# 05장 타겟 머신의 구조

## 【5.4.4】 구조의 두 가지 예: x86과 MIPS

CISC와 RISC 구조 계열의 예로 x86과 MIPS를 다루며, 주소 지정, 레지스터 집합, 명령어를 순서대로 살펴본다.

### 메모리 접근과 주소 지정 방식

모든 RISC 기계와 마찬가지로 MIPS는 불러오기/저장하기 구조를 가진다. 메모리 접근 명령어는 변위 주소 지정만을 지원한다. 0의 변위를 사용해서 레지스터 간접 주소 지정도 포괄한다. 또 0을 기반으로 삼아서(레지스터 0에 직접 물리적으로 연결) 처음 64K의 메모리에서의 절대 주소 지정을 포괄할 수도 있다. 이와 달리 x86에서 대부분의 명령어는 피연산자 중 하나(둘 모두는 아님)를 메모리에서 얻을 수 있다. 이러한 참조를 위해 9개의 다른 주소 지정 방식을 이용할 수 있다. 가장 일반적인 경우는 기준화 색인(scaled indexed) 주소 지정이다. 이 주소 지정 방식은 기준 레지스터 $R_b$, 색인 레지스터 $R_i$, 변위 d, 기준화 요소 s를 사용한다. s의 값은 1, 2, 4, 8 중 하나여야 하므로 2비트로 인코딩할 수 있다. 피연산자의 실제 주소는 $(R_b) + d + (R_i) \times s$다. 여기서 (R)은 레지스터 R의 내용을 나타낸다. 다른 x86 주소 지정 방식은 모두 이 일반적인 경우를 단순화한 것들이다.

MIPS 명령어는 세 개의 주소를 사용한다. X86 명령어는 두 개의 주소만을 사용하며 계산의 결과는 레지스터나 메모리 중 하나일 수 있는 하나의 피연산자에 덮어쓴다.

## 레지스터

**예 5.20**

x86과 MIPS
레지스터 집합

두 기계의 레지스터 집합을 ◎(심화학습에 있는) 그림 5.10에 나타냈다.

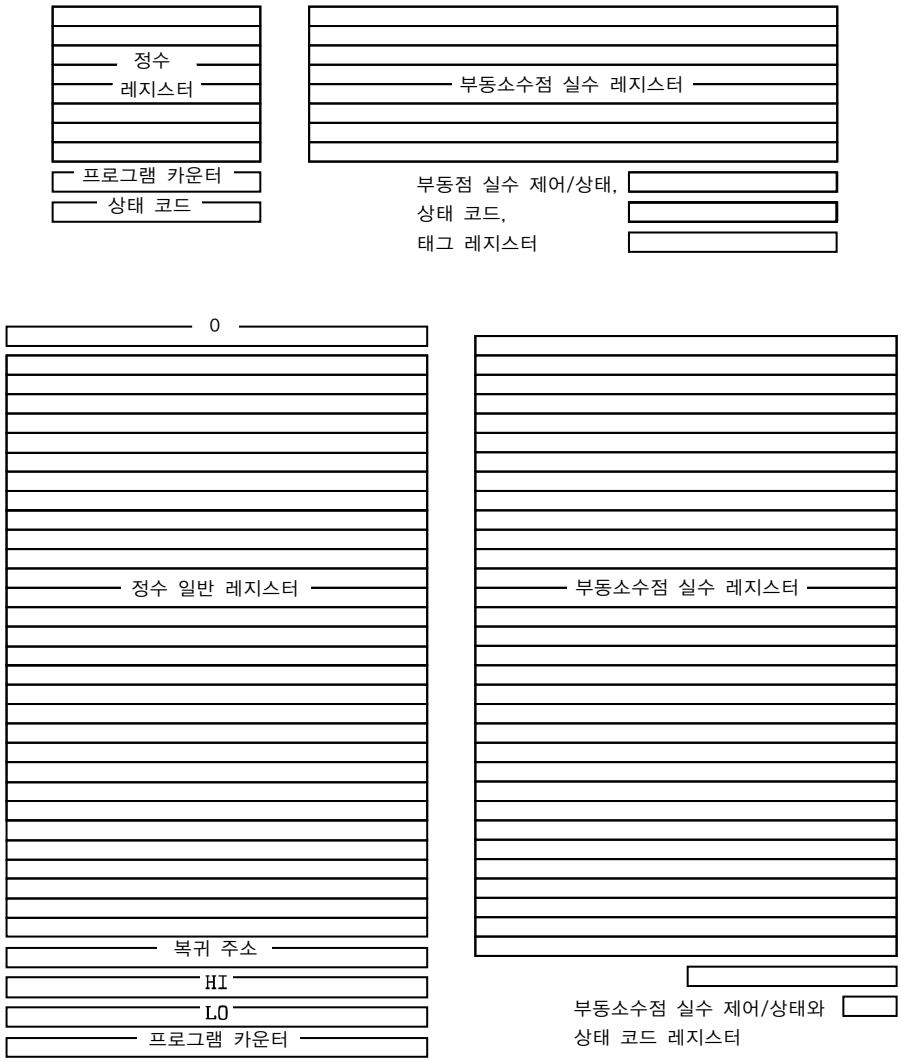


**그림 5.10 | x86(위)와 MIPS IV(아래)의 레지스터 집합.** 두 경우 모두 사용자 수준 프로그래머가 관심을 가지는 레지스터들만 나타냈다. 두 구조의 구현은 운영체제의 특수 목적 레지스터 사용을 포함한다. 여기에 나타내지 않은 것은 x86의 SSE 확장과 함께 도입된 128비트 "스트리밍 레지스터" 8개와 MIPS의 MDMX 확장과 함께 도입된 192비트 누산기다. 또 생략한 것은 폐기된 80286 주소 지정 시스템을 지원하는 8개의 x86 세그먼트 레지스터다. 현대 컴파일러는 이 레지스터를 사용하지 않는다.

둘의 가장 큰 차이점은 RISC 기계의 절대적인 레지스터 개수로 CISC 기계보다 약 4배 정도 많다. 정수 레지스터도 MIPS에서 더 넓지만 두 기계 모두 시간이 지남에 따라 "넓어졌다" 1978년에 나온 8086은 16비트 정수 레지스터를 사용했다(8086은 이전

의 8비트 8080과의 소스 코드는 호환성은 가지지만 바이너리 호환성은 가지지 않았다). 인텔은 1985년에 80386의 레지스터를 32비트로 확장했다. 이와 달리 32비트 정수 레지스터를 가지는 MIPS는 1984년에 나왔으며 1991년에 64비트로 확장되었다. 2003년에 나온 AMD64(옵테론) 구조는 x86을 64비트 레지스터로 확장한다. 여기서는 이러한 확장은 다루지 않는다.

X86은 8개의 32비트 정수 레지스터와 프로그램 카운터, 상태 코드 등을 포함하는 프로세서 상태 워드를 가진다. 역사적인 이유 때문에 정수 레지스터는 `eax`, `ebx`, `ecx`, `edx`, `esi`, `edi`, `esp`, `ebp`로 명명된다. 이들은 대부분의 명령어에서 서로 교환 가능하게 사용할 수 있지만 어떤 명령어들은 레지스터들을 특별한 방식으로 사용한다. 예를 들어 레지스터 `eax`와 `edx`는 묵시적으로 정수 곱셈과 나눗셈 연산의 목적지 레지스터다. 레지스터 `ecx`는 특정 루프 제어 명령어가 묵시적으로 읽고 갱신한다. 레지스터 `esi`와 `edi`는 메모리에 있는 문자열을 복사, 검색, 비교하는 명령어들이 묵시적으로 사용한다. 레지스터 `esp`는 스택 포인터로 사용되며 `push`, `pop`, 서브루틴 호출/복귀 명령어가 이 레지스터를 읽고 쓴다. 레지스터 `ebp`는 주로 프레임 포인터로 사용된다. 스택 프레임을 할당하고 해제하게 설계된 명령어가 이 레지스터를 조작한다.

16비트 코드와의 하위 호환성을 위해 정수 레지스터 8개의 하위 절반은 `ax`, `bx`, `cx`, `dx`, `si`, `di`, `sp`, `bp`라는 별도의 이름을 가진다. 이 중 4개(`ax`, `bx`, `cx`, `dx`)는 `ah`, `al`, `bh`, `bl`, `ch`, `cl`, `dh`, `dl`과 같이 상위와 하위 절반에 대해 별도의 이름을 가진다.

부동소수점 실수 명령어는 별개의 80비트 부동소수점 실수 레지스터 집합을 조작한다. IEEE 부동소수점 실수 상태와 제어, 부동소수점 실수 상태 코드, 다양한 부동소수점 실수 레지스터에 있는 값이 정규, 비정규, NaN, 쓰레기 값 중 무엇인지를 나타내는 "태그" 비트를 위한 레지스터도 있다. 모든 계산은 확장된 정밀도에서 수행된다. 값을 메모리에서 읽어올 때는 IEEE 단정도와 배정도 부동소수점 실수로부터 변환하며, 메모리에 쓸 때는 IEEE 단정도와 배정도 부동소수점 실수로 변환한다.

최근의 x86 프로세서 계열은 산술 연산을 작은 정수나 부동소수점 실수 피연산자의 벡터상에서 수행할 수 있게 해주는 명령어 집합 확장(MMX와 SSE)을 지원한다. 이러한 확장을 더 깊게 다루지는 않지만 8개의 MMX 벡터 레지스터가 x86 부동소수점 실수 레지스터의 하위 64비트와 겹친다는 사실은 알아두는 것이 좋다. 8개의 SSE 벡터는 새로운 것으로 각 벡터는 128비트다.

MIPS는 정수와 부동소수점 실수 레지스터를 32개씩, 총 64개의 레지스터와 함께 프로그램 카운터, 곱셈과 나눗셈 명령어에서 사용하는 LO와 HI라는 특수 레지스터 한 쌍, x86과 유사한 부동소수점 실수 제어와 상태 레지스터, 8비트 부동소수점 실수 상태 코드 레지스터를 가진다. 정수 연산의 결과에 기반한 분기는 값을 검사하거나 비교하고 그 결과에 기반해서 분기하는 조합 명령어를 사용한다. 정수 상태 코드는 없다. 초기 세대의 MIPS에서 정수 레지스터는 32비트였다. 좀 더 최근의 세대는 64비트의

정수 레지스터를 가진다. 배정도(64비트) 부동소수점 실수 산술은 초기 MIPS 때부터 사용할 수 있었으나 초기 세대에서는 부동소수점 실수 레지스터를 쌍으로 함께 사용해야 했다.

MIPS의 정수 레지스터 `r0`는 항상 0이다. 이러한 설계 전략을 통해 명령어 인코딩을 다양하게 단순화할 수 있다. 예를 들어 한 레지스터에서 다른 레지스터로 값을 이동하기 위해 `r0`와 함께 `add`(또는 `sub`나 `or`)를 수행할 수 있다. 별도의 명령어가 필요하지 않다. 값의 부호를 바꾸려면 `r0`에서 그 값을 빼면 된다. 무조건적으로 분기하기 위해 `r0=r0`인지 "검사"할 수도 있다. MIPS 레지스터에서 `r0` 외에 유일하게 달리 취급하는 레지스터는 서브루틴 호출 명령어, `jal`(건너뛰고 연결, jump and link)이다. 이 명령어는 복귀 주소를 항상 레지스터 `r31`에 저장한다.

MIPS의 정수 곱셈 명령어는 레지스터 `LO`와 `HI`에 결과를 기입한다(n비트 피연산자가 주어지면 곱셈 결과에는 2n비트가 필요할 수 있다). 나눗셈은 항상 `LO`와 `HI`에 몫과 나머지를 생성한다. 특수 `move` 명령어는 `LO`와/또는 `HI`로부터 정수 레지스터로 복사한다. 앞서 살펴봤듯이 x86은 이러한 목적으로 레지스터 `eax`와 `edx`를 다중 정의한다.

x86의 MMX나 SSE 확장과 유사한 방식으로 최근의 MIPS 프로세서는 소규모의 정수와 부동소수점 실수 벡터 연산을 지원한다. 이러한 연산을 위한 자료는 부동소수점 실수 레지스터와 새로운 192비트 벡터 누산기에 저장한다. 누산기는 정수 벡터 곱의 결과가 오버플로우 없이 전부 더해질 수 있게 해준다.

**레지스터 규약 하드웨어:** 레지스터 일부의 특수 처리 외에 x86과 MIPS 설계자들은 추가적인 규약을 소프트웨어로 강제하게 권장한다. x86에서 레지스터 `ebp`는 컴파일러가 특수 프레임 관리 명령어를 사용하든지 사용하지 않든지 일반적으로 프레임 포인터를 위해 사용한다. 함수 값은 레지스터 `eax`(또는 64비트 반환 값의 경우 레지스터 쌍 `eax:edx`)에 반환된다. 레지스터 `ebx`, `esi`, `edi`를 수정하는 서브루틴은 이전 값을 메모리에 저장하고 복귀 전에 반드시 복구해야 한다. `eax`, `ecx`, `edx`의 값을 필요로 하는 호출자는 반드시 호출 전에 이 값을 저장해야 한다(호출 순서는 5.5.2절에서 좀 더 자세히 다룬다).

MIPS 규약은 훨씬 더 정교하다. 레지스터 `r1`은 어셈블러를 위해 남겨두며 어셈블러는 이 레지스터를 사용해서 특정 유사명령어를 실제 명령어 나열로 확장한다. 레지스터 `r2`와 `r3`는 수식 값 계산과 함수 복귀에 사용된다. 레지스터 `r4..r7`은 서브루틴 매개변수를 위해 사용한다. 레지스터 `r16..r23`은 "피호출자 저장" 레지스터로 x86의 `ebx`, `esi`, `edi`와 유사하다. 레지스터 `r8..r15`, `r24`와 `r25`는 "호출자 저장" 레지스터다. 레지스터 `r26`과 `r27`은 운영체제 커널이 사용하게 남겨둔다. 사용자 프로그램의 관점에서 볼 때 이 두 레지스터의 값은 자발적으로 변경될 수 있다. 레지스터 `r29`과 `r30`은 각기 스택 포인터와 프레임 포인터를 위해 사용한다.

## 명령어

주어진 명령어 집합의 명령어 개수를 세는 것은 어렵지만(x86은 상태 코드의 다른 조합 16개 중 어느 것으로나 분기할 수 있다. 이는 x86이 16개의 조건 분기 명령어를 가졌다는 것인가, 아니면 16가지 변형을 가지는 하나의 조건 분기 명령어를 가졌다는 것인가?) x86이 MIPS보다 많은 수의 명령어를 가지며 더 복잡하다는 것은 여전히 분명하다. MIPS에는 없는 x86의 기능에는 다음과 같은 것이 있다.

- 이진화 십진수(BCD) 산술[1]
- 문자열 검색, 비교, 복사 연산
- 비트 검사와 설정 연산
- 비트 스트링 검색과 복사 연산
- 잡다한 "조합" 명령어: 이러한 명령어들은 특정 다중 명령어 나열과 동일한 작업을 수행하지만 코드 공간을 덜 차지하며 대개 좀 더 실행된다. 예를 들어 바이트와 레지스터 스왑(swap), 서브루틴 호출과 복귀, 스택 연산, 루프 제어 등이 있다.
- 폐기된 80286 세그먼트 메모리 시스템을 지원하기 위한 명령어

한편 MIPS는 다음과 같은 기능을 제공한다.

- 명령어 2개의 나열과 함께 32비트 값을 레지스터에 불러오게 할 수 있는 "빌딩 블록" 명령어
- 대부분의 산술, 논리, 메모리 접근(불러오기/저장하기) 명령어에 대한 별도의 32비 및 64비트 버전
- 무효화 분기(5.5.1절에서 다룸)
- 조건 트랩: 이를 통해 동적 의미 오류 시 운영체제로 빠르게 문맥 전환할 수 있다.

그러나 명령어의 개수나 유형의 차이보다 더 중요한 것은 이러한 명령어가 어떻게 인코딩되느냐다. 대부분의 CISC 기계와 마찬가지로 x86은 코드 크기를 최소화하는 데 상당히 주안점을 두며(그러므로 실행 시간에 메모리가 필요) 이 때문에 상대적으로 어려운 명령어 디코딩을 감수한다. 1에서 17바이트까지의 길이를 가지는 명령어들은 다양한 내부 형식을 가진다. 유사한 항목들이라 하더라도 동일한 길이일 필요도 없으며

주1. BCD 형식은 1니블(nibble, 4비트, 즉 1바이트의 절반)을 각 십진수 자리수에 할당한다. 많은 CISC 기계(그리고 적어도 하나의 RISC인 PowerPC)와 마찬가지로 x86은 BCD 표현을 이진 형태로 변환하거나 이진 형태로부터 변환하지 않고 바로 BCD 표현에 대한 산술을 수행한다. 이 기능은 특히 자료를 숫자와 문자열을 모두 자료로 취급하는 상업 및 재정 응용에서 유용하다. 아스키 숫자 열을 BCD로 변환하거나 BCD에서 아스키 숫자 열로 변환하는 것은 이진수로 변환하거나 이진수로부터 변환하는 것보다 훨씬 쉽다.

다른 명령어에서 동일한 오프셋에 나올 필요도 없다. 피연산자 명시자는 주소 지정 방식에 따라 길이가 달라진다. 명령어가 여러 번 반복하거나 80286 주소 공간의 다른 세그먼트에 있는 피연산자에 접근하거나 주 메모리로의 원자적 접근을 위해 버스를 잠그게 하기 위해 특정 명령어들의 동작을 수정하려고 명령어 앞에 1바이트 접두어 코드를 붙일 수 있다.

부동소수점 실수 연산은x86 명령어 집합에서 가장 장식적인 요소일 것이다. 최초의 부동소수점 실수 코프로세서인 인텔 8087의 설계자들은 부동소수점 실수 레지스터를 스택으로 처리함으로써 부동소수점 실수 명령어의 공간을 유지하게 했다. 예를 들어 부동소수점 실수 불러오기 명령어는 목적지 레지스터를 명시하지 않는다. 대신 피연산자를 `st(0)`으로 지명된 레지스터 스택 상단에 푸쉬한다. 레지스터 `st(0)`에서 `st(6)`까지의 이전 내용은 아래로 이동하게 되며 예를 들어 `st(4)`에서 사용되던 것은 이제 `st(5)`에 있게 된다. `st(7)`의 내용은 손실된다. 산술 명령어도 이와 유사하게 스택을 조작한다. 예를 들어 가장 간단한 `add` 명령어는 피연산자나 목적지 레지스터를 명시하지 않는다. 대신 `st(0)`과 `st(1)`에서 묵시적으로 피연산자를 읽고 `st(0)`의 값을 덧셈 결과로 대체한다. `add` 명령어의 한 변형은 피연산자를 팝한다. 즉, 레지스터 `st(2)`에서 `st(7)`까지의 내용이 한 자리 위로 이동하며 `st(0)`에는 이전 `st(0)`과 `st(1)`의 덧셈 결과를 저장한다. 추가적인 명령어 변형들은 명시적으로 명명된 스택 하단부의 레지스터나 메모리 위치로부터 피연산자 하나를 가져온다. 뺄셈과 나눗셈과 같이 교환 법칙이 성립하지 않는 연산자들은 피연산자를 둘 중 하나의 순서로 사용하는 변형을 가진다.

부동소수점 산술 명령어(그리고 비교와 검사 명령어)는 부동소수점 실수 상태 코드를 갱신하지만 이러한 코드에 기반해 제어 흐름을 변경하는 특수 분기 명령어는 존재하지 않는다. 대신 부동소수점 실수 상태 워드를 16비트 정수 레지스터 중 하나에 복사하기 위해서는 `fnstsw` 명령어를 사용해야 한다. 복사 목적지가 되는 정수 레지스터는 다음의 비트 검사 명령어가 바람직한 상태 코드를 접근할 수 있는 곳이다. 그 후 `fnstsw` 명령어는 분기를 결정하는 데 사용하는 정수 상태 코드를 설정한다.

대부분의 RISC 기계와 마찬가지로 MIPS는 비교적 간단한 인코딩을 가지는 고정 길이의 32비트 명령어를 사용한다. 처음 6비트는 op코드를 명시한다. 나머지 비트는 (1) 26비트 건너뛰기 변위, (2) 레지스터 명시자와 16비트 상수 한 쌍, (3) 세 개의 레지스터 명시자와 일부가 특수 목적 명령어에 의해 사용되는 11개의 부가적인 비트를 포함한다. 다수의 레지스터-레지스터 연산이 사용되지 않는 비트를 포함하며 가령 x86에서 40비트로 명시할 수 있는 연산이 MIPS에서는 두 개의 완전한 명령어(64비트)를 필요로 할 수도 있다. X86과 마찬가지로 MIPS도 IEEE 754 부동소수점 실수 표준을 구현하지만 MIPS의 부동소수점 실수 명령어 집합은 훨씬 더 간단하고 이해하기 쉽다(정수 집합과 매우 유사하다). 부동소수점 실수 분기 명령어는 부동소수점 실수 상태 코드에 직접 접근할 수 있다.

## ✓ 확인문제

36. x86과 MIPS 구조의 가장 일반적인(복잡한) 주소 지정 방식을 설명하라.

37. x86과 MIPS는 각기 몇 개의 정수 레지스터와 부동소수점 실수 레지스터를 제공하는가? 이 레지스터들의 크기는 얼마인가?

38. MIPS의 레지스터 0의 유용성을 설명하라.

39. x86과 MIPS의 레지스터 사용 규약을 요약하라.

40. x86은 제공하지만 MIPS는 제공하지 않는 "복잡한" 명령어를 최소한 세 개 나열하라.

41. MIPS는 제공하지만 x86은 제공하지 않는 "간단한" 명령어를 하나 명명하라.

42. x86의 부동소수점 실수 스택을 기술하라.

43. x86과 MIPS 간 명령어 인코딩의 차이점을 요약하라.