Chapter 04

명령어 집합의 분류와 주소 지정 방식

01 피연산자의 수와 명령어 집합

02 명령어와 메모리

03 주소 지정 방식

04 복잡도에 따른 명령어 집합

05 picoMIPS 명령어 집합 구조

요약

연습문제

학습목표

- 피연산자의 수에 따른 각종 명령어 집합과 트래픽 사이의 관계를 분석한다.
- 메모리 속성, 데이터 정렬, 엔디언 방식에 대해 살펴본다.
- 데이터가 실제 위치한 유효 주소를 결정하는 각종 주소 지정 방식을 학습한다.
- RISC와 CISC 명령어 집합 구조의 장단점을 비교 · 분석한다.
- 기본 아키텍처로 사용할 picoMIPS의 명령어 형식과 종류를 살펴본다.

Preview

명령어 집합은 일반적으로 명령어에 포함된 피연산자의 수에 따라 분류하거나, 명령어의 복잡성, 즉 종류의 다양성이나 강력함 정도에 따라 분류한다. 이 장에서는 이와 같은 방식으로 분류된 명령어 집합을 비교하고, 간단한 프로그램을 수행할 때 발생하는 메모리 트래픽을 살펴볼 것이다. 그리고 명령어에 명시된 피연산자 정보로 데이터에 접근할 수 있는 방법을 의미하는 주소 지정 방식을 학습하고, 이 책에서 기본 아키텍처로 사용할 picoMIPS의 명령어 형식을 알아본다.

01 피연산자의 수와 명령어 집합

명령어는 일반적으로 어떤 연산을 수행할 것인가를 나타내는 연산 부호와 연산될 데이터에 대한 정보를 포함하는 피연산자로 구성된다. 연산 부호는 특별한 경우가 아니면 하나의 필드에 존재하지만, 피연산자는 여러 개의 필드로 구성될 수 있다. 명령어는 내부에 명시적으로 나타난 피연산자의 수에 따라 0-주소 명령어, 1-주소 명령어, 2-주소 명령어 등으로 분류한다. 0-주소 명령어는 스택 컴퓨터tack computer에서, 1-주소 명령어는 누산기 컴퓨터accumulator computer에서, 2-주소 명령어나 3-주소 명령어는 범용 레지스터 컴퓨터general purpose register computer에서 사용된다.

현대 컴퓨터의 전형적인 모델인 프로그램 내장식 컴퓨터는 CPU와 메모리를 연결하는 부분, 즉 폰노이만 병목이 컴퓨터의 성능을 좌우한다. 3장 5절에서 메모리-메모리 컴퓨터의 계산식 $y = ax^2 + bx + c$를 수행할 때 폰노이만 병목에 발생하는 트래픽을 살펴보았다. 이 계산식을 통해 각종 아키텍처에서 폰노이만 병목에 발생하는 트래픽을 분석해보자.

1 누산기 컴퓨터

메모리-메모리 컴퓨터는 CPU 내부에 데이터를 저장하는 기억장치가 없다. 이 메모리-메모리 컴퓨터에 하나의 데이터를 저장하기 위한 레지스터를 추가한 컴퓨터를 누산기 컴퓨터라 하고, 그 레지스터를 누산기라고 한다. 누산기 컴퓨터는 다음과 같은 특징을 가지고 있다.

- 누산기를 묵시적 피연산자로 사용한다. 누산기는 CPU 내부에 데이터를 저장할 수 있는 유일한 레지스터이므로 누산기를 명시할 필요가 없기 때문이다.
- 적재 혹은 저장 명령어가 아니면 누산기는 근원지 겸 목적지 레지스터로 사용된다.
- 메모리를 명시적 피연산자로 사용한다. 데이터가 다수의 워드로 구성된 메모리 중 하나의 워드에 있기 때문에 데이터가 있는 워드의 위치를 명시해야 한다. 명시적 피연산자는 저장 명령어일 경우 메모리가 목적지가 되지만 대부분의 경우에는 근원지가 된다.
- 누산기 컴퓨터는 CPU 내에 기억장치가 하나의 워드 용량인 누산기만 있으므로 단순하고 저렴하다.

누산기 컴퓨터는 하나의 레지스터만 사용할 뿐인데도 메모리 트래픽의 감소 효과가 크다. 이는 전화기에 재발신 버튼(실제로는 하나의 전화번호에 해당하는 메모리)만 추가하더라도 버튼을 누르는 횟수가 매우 줄어드는 것과 동일한 이치이다. 누산기 컴퓨터의 대표적인 예는 PDP-8, Motorola 6809 등이지만 현재는 사용하지 않는다.

누산기 컴퓨터가 연산할 때 발생되는 트래픽을 분석하기 위해 다음과 같은 조건을 사용한다. 이는 3장 5절에서 살펴본 메모리-메모리 컴퓨터의 경우와 동일한 가정이다.

- **메모리 주소** : 16비트, 즉 2바이트
- **데이터 크기** : 32비트, 즉 4바이트
- **연산 부호** : 8비트, 즉 1바이트

위 조건에 따르면 명령어는 최대 $2^8(= 256)$개, 데이터는 4바이트, 메모리의 용량은 최대 64K개의 데이터인 256K 바이트를 사용할 수 있다. 누산기 컴퓨터를 위한 가상의 어셈블리어를 사용하여 $y = ax^2 + bx + c$를 프로그래밍하면 [표 4-1]과 같다.

표 4-1 누산기 컴퓨터의 연산

명령어		의미
연산 부호	피연산자	
lda	a	acc ← M[a]
mul	x	acc ← M[a] × M[x]
mul	x	acc ← M[a] × M[x]^2
sta	y	y ← acc의 내용
lda	b	acc ← M[b]
mul	x	acc ← M[b] × M[x]
add	c	acc ← M[b] × M[x] + M[c]
add	y	acc ← M[a] × M[x]^2 + M[b] × M[x] + M[c]
sta	y	y ← acc의 내용

여기서 lda는 메모리의 내용을 누산기에 적재하는 명령어, sta는 누산기의 내용을 메모리에 저장하는 명령어, mul과 add는 각각 곱셈, 덧셈 명령어이다. acc는 누산기를 의미하고, 변수 a,

b, c, x, y는 모두 피연산자로 데이터가 위치한 메모리의 주소를 명시한다. 그리고 M[a], M[x] 등은 메모리의 a번지, x번지 등에 있는 데이터를 의미한다.

[표 4-1]에 나타난 바와 같이 누산기 컴퓨터의 명령어는 명시적 피연산자가 하나이므로 누산기 컴퓨터를 1-주소 명령어 컴퓨터라고도 한다. 그럼 위와 같은 프로그램을 누산기 컴퓨터에서 실행할 때 CPU와 메모리 사이에 발생하는 트래픽이 어느 정도인지 분석해보자.

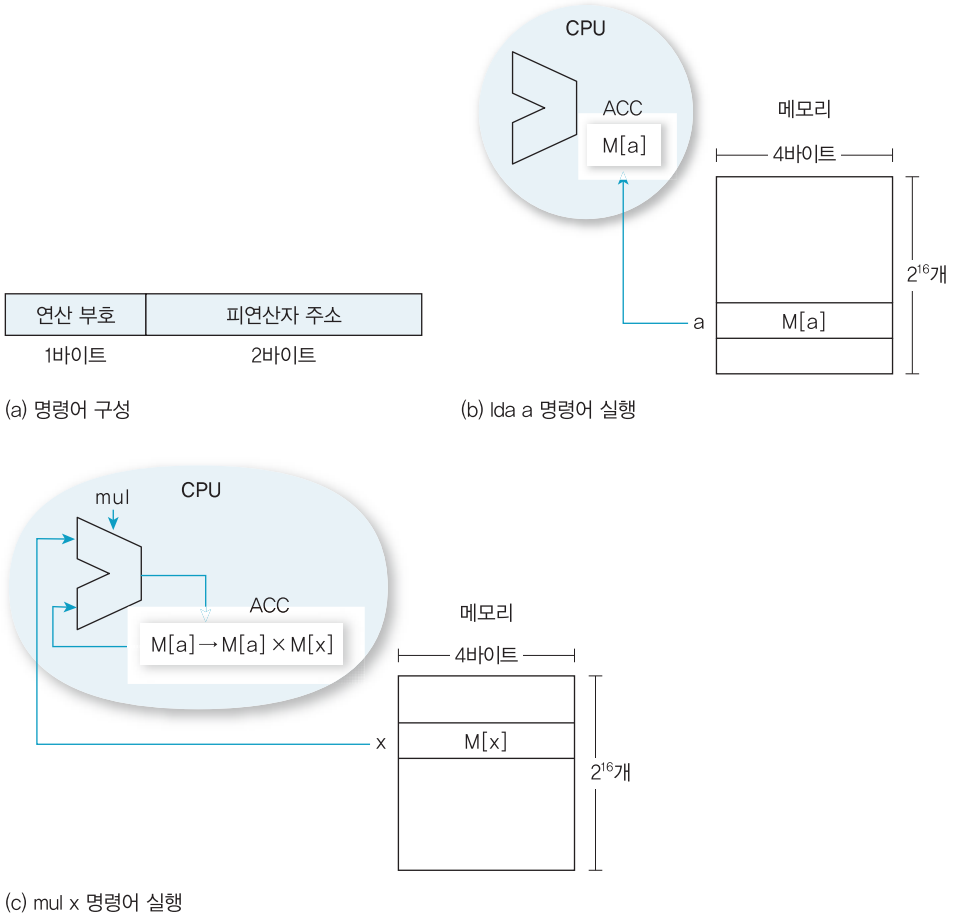


그림 4-1 누산기 컴퓨터의 명령어 구성과 데이터 트래픽

먼저 명령어를 메모리에서 인출하는 데 필요한 트래픽 발생량을 살펴보자. 모든 명령어는 [그림 4-1]의 (a)와 같이 연산 부호와 하나의 명시적 피연산자로 구성되어 있다. 변수 a, x 등은 모두

피연산자로 데이터가 위치한 메모리의 주소를 명시한다. 연산 부호는 1바이트이고 피연산자는 2바이트이므로 명령어의 크기는 3(= 1 + 2)바이트이다. 따라서 9개의 명령어를 인출하는 데 CPU와 메모리 사이에 27(= 3 × 9)바이트의 트래픽이 발생한다.

다음으로, 연산을 수행할 때 데이터 이동에 필요한 트래픽 발생량을 살펴보자. 9개의 명령어는 두 가지 종류의 연산으로 구분할 수 있다. lda와 sta는 [그림 4-1]의 (b)와 같이 메모리와 누산기 사이에서 데이터를 이동하는 명령어로 4바이트의 메모리 트래픽을 발생시킨다. 그리고 mul과 add는 각각 메모리에서 CPU로 가져온 데이터와 누산기의 내용에 대해 곱셈 및 덧셈 연산을 수행한 후 그 결과를 누산기에 저장하는 명령어이다. 따라서 이 명령어도 lda와 sta처럼 4바이트의 메모리 트래픽을 발생시킨다. 즉, 프로그램에 있는 모든 명령어는 데이터를 참조하기 위해 메모리에 한 번만 접근하면 되므로 명령어마다 4바이트의 데이터 트래픽이 발생한다. 따라서 9개의 명령어가 발생시키는 총 데이터 트래픽은 36(= 4 × 9)바이트이다.

명령어 인출을 위한 27바이트와 데이터 전송을 위한 36바이트를 더하여 총 메모리 트래픽은 63바이트가 발생한다. 3장 5절의 메모리-메모리 컴퓨터와 비교해보라. 하나의 데이터를 저장하기 위해 CPU에 누산기만 추가했는데 폰노이만 병목에 대한 부담이 약 56.8%로 줄어들었다. 이는 CPU 내부에 소규모 기억장치를 추가하더라도 메모리 트래픽에 적지 않은 영향을 준다는 것을 의미한다. 결과적으로 CPU 내부에 추가된 기억장치가 명령어 집합 설계에 큰 영향을 미친다는 것을 알 수 있다.

2 스택 컴퓨터

스택 컴퓨터는 CPU 내부에 다수의 데이터를 임시로 저장하기 위해 스택을 사용하는 컴퓨터를 말한다. 스택 컴퓨터는 다음과 같은 특징을 가지고 있다.

- 스택에 저장된 모든 데이터는 위치에 따라 접근 시간이 다르다. 스택의 하위에 있는 데이터에 접근하려면 먼저 상위에 있는 모든 데이터를 차례로 제거해야 하므로 상위에 있는 데이터보다 접근 시간이 길다.
- 대부분의 연산은 스택의 최상위 또는 차상위 데이터를 사용하여 수행한다. 따라서 명령어는 스택의 최상위에 위치한 1~2개 데이터를 묵시적 피연산자로 사용할 수 있다.
- 최근 연산한 데이터는 스택의 최상위에 저장된다.
- 보관할 데이터의 양이 스택 용량보다 크면 스택의 최하위에 있는 데이터부터 삭제해야 하므

로 성능이 많이 떨어진다.

스택 컴퓨터는 메모리에 접근하기 위해 다음과 같은 명령어를 사용한다.

- **push** : 메모리에서 스택의 최상위로 데이터를 이동하는 명령어로, 누산기 컴퓨터의 lda와 같은 적재 명령이다.
- **pop** : 스택의 최상위에 있는 데이터를 메모리로 이동하는 명령어로, 누산기 컴퓨터의 sta와 같은 저장 명령이다.

스택 컴퓨터의 대표적인 예는 B5500, HP3000/70 등이지만, 누산기 컴퓨터와 마찬가지로 현재 특별한 경우가 아니면 사용하지 않는다.

누산기 컴퓨터의 경우와 동일한 가정하에 연산 도중 발생하는 트래픽을 분석해보자. 스택 컴퓨터를 위한 가상 어셈블리어를 사용하여 $y = ax^2 + bx + c$를 프로그래밍하면 [표 4-2]와 같다.

표 4-2 스택 컴퓨터의 연산

명령어		의미
연산 부호	피연산자	
push	a	tos : M[a]
push	x	tos : M[x], M[a]
push	x	tos : M[x], M[x], M[a]
mul		tos : $M[x]^2$, M[a]
mul		tos : M[a] × $M[x]^2$
push	b	tos : M[b], M[a] × $M[x]^2$
push	x	tos : M[x], M[b], M[a] × $M[x]^2$
mul		tos : M[b] × M[x], M[a] × $M[x]^2$
push	c	tos : M[c], M[b] × M[x], M[a] × $M[x]^2$
add		tos : M[b] × M[x] + M[c], M[a] × $M[x]^2$
add		tos : M[a] × $M[x]^2$ + M[b] × M[x] + M[c]
pop	y	y ← tos의 내용

여기서 tos는 스택의 최상위top of stack를 나타낸다. [표 4-2]에 제시된 명령어의 수행 과정을 그림으로 나타내면 [그림 4-2]와 같다.

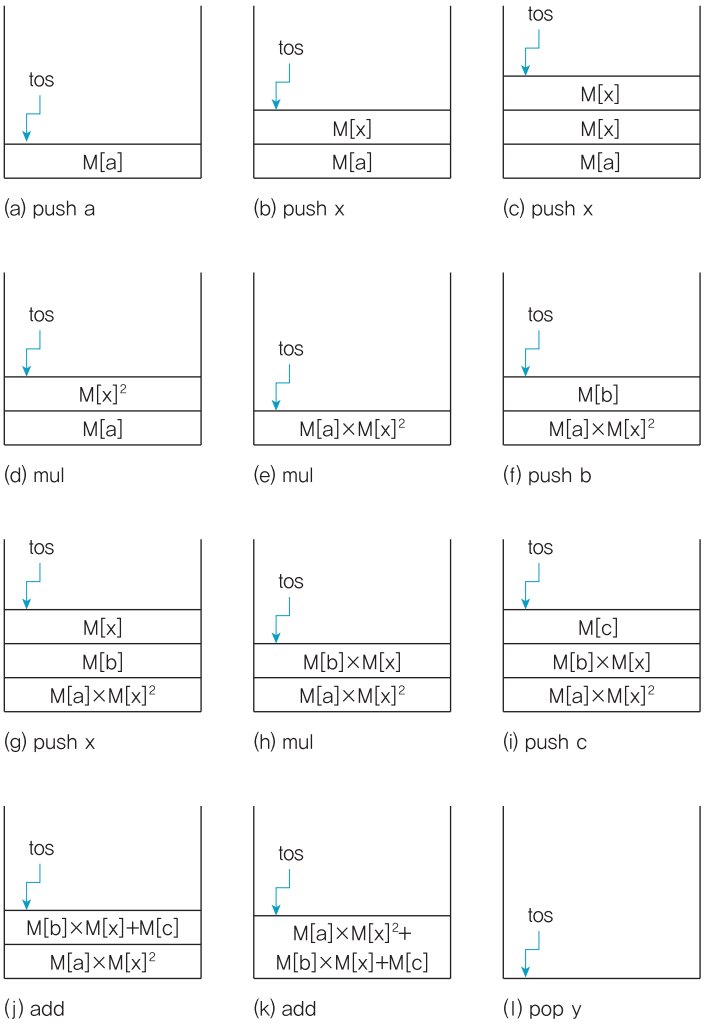


그림 4-2 스택 컴퓨터의 연산 과정과 스택 상태

push 명령어에 의해 메모리의 내용이 CPU의 스택 최상위에 입력되고, pop 명령어에 의해 스택 최상위의 내용이 메모리에 저장된다. 그리고 push와 pop 외의 명령어는 지정된 장소인 스택의 최상위 데이터 1~2개를 사용하여 연산하므로 피연산자를 명시할 필요가 없다. 따라서 스택 컴퓨터를 0-주소 명령어 컴퓨터라고도 하는 것이다. 그림 위와 같은 프로그램을 스택 컴퓨터

에서 실행할 때 CPU와 메모리 사이에 발생하는 트래픽이 어느 정도인지 분석해보자.

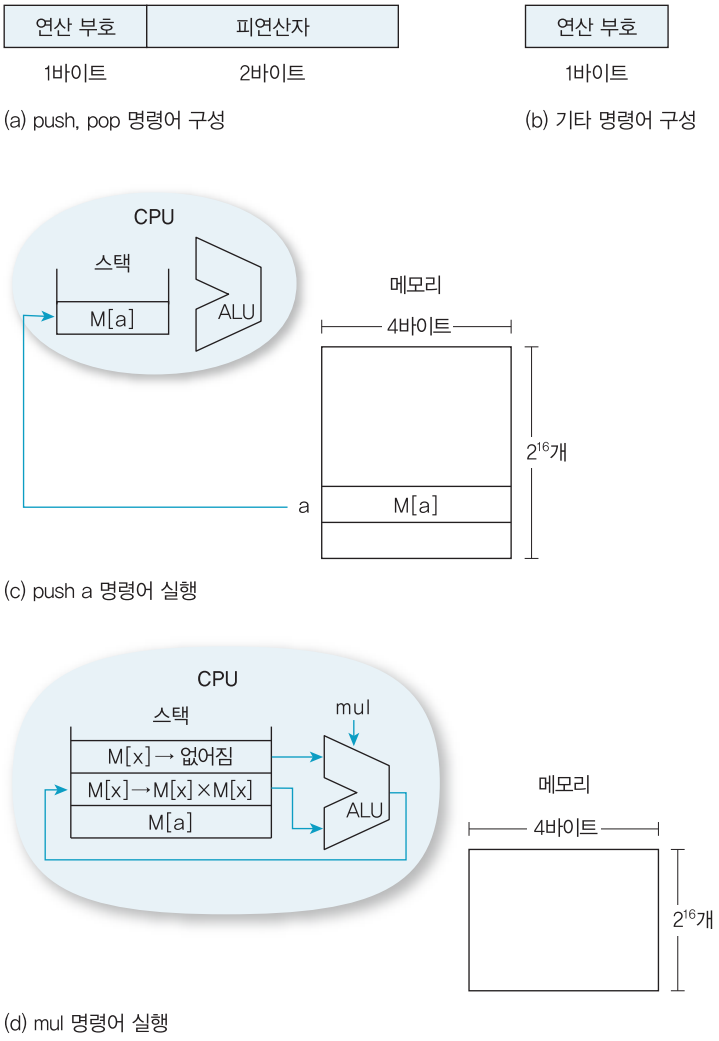


그림 4-3 스택 컴퓨터의 명령어 구성과 데이터 트래픽 발생

명령어를 메모리에서 인출하는 데 필요한 트래픽 발생량을 먼저 살펴보자. push와 pop 명령어는 [그림 4-3]의 (a)와 같이 연산 부호와 하나의 명시적 피연산자를 가지고 있다. 따라서 1바이트의 연산 부호와 2바이트의 피연산자 하나로 구성되므로 명령어의 크기는 3(= 1 + 2)바이트이다. 반면에 mul과 add 명령어는 [그림 4-3]의 (b)와 같이 1바이트의 연산 부호로만 구성

되어 있으므로 명령어의 크기가 1바이트이다. 결과적으로 push, pop 명령어 7개와 mul, add 명령어 5개를 인출해야 하므로 CPU와 메모리 사이에 26(= 3 × 7 + 1 × 5)바이트의 트래픽이 발생한다.

다음으로, 연산을 수행할 때 데이터 이동에 필요한 트래픽 발생량을 살펴보자. push와 pop 명령어는 [그림 4-3]의 (c)와 같이 각각 하나의 데이터를 메모리에서 스택으로 전송하거나 스택에서 메모리로 전송한다. 따라서 push와 pop 명령어는 4바이트의 트래픽을 발생시킨다. 반면에 mul과 add 명령어는 [그림 4-3]의 (d)와 같이 피연산자가 없으므로 데이터 트래픽이 없다. 따라서 push와 pop 명령어 7개만 데이터 트래픽을 발생시키므로 총 데이터 트래픽은 28(= 4 × 7)바이트이다.

명령어 인출을 위한 26바이트와 데이터 전송을 위한 28바이트를 더하여 총 메모리 트래픽은 54바이트가 발생한다. 이와 같은 연산을 수행할 경우, 다수의 데이터를 저장하는 스택 컴퓨터는 하나의 데이터만 저장할 수 있는 누산기 컴퓨터보다 폰노이만 병목에 대한 부담을 줄여준다. 그러나 스택의 특성 때문에 애플리케이션에 따라 누산기 컴퓨터보다 스택 컴퓨터의 성능이 나쁠 수도 있다.

3 범용 레지스터 컴퓨터

범용 레지스터 컴퓨터는 CPU 내부에 다수의 데이터를 임시로 저장하기 위해 범용 레지스터general purpose register를 사용하는 컴퓨터를 말한다. 범용 레지스터 컴퓨터는 다음과 같은 특징을 가지고 있다.

- 범용 레지스터는 스택과 달리 모든 레지스터의 접근 시간이 동일하며, 순서에 상관없이 임의로 접근할 수 있다.
- 누산기나 스택과 달리 레지스터가 명시적 피연산자로 사용된다. 데이터가 다수의 레지스터 중 하나에 저장되기 때문에 레지스터 주소를 명시해야 한다.

범용 레지스터 컴퓨터는 피연산자의 수에 따라 2-주소 명령어 컴퓨터나 3-주소 명령어 컴퓨터가 될 수도 있다. 다음은 2-주소 명령어와 3-주소 명령어의 전형적인 예이다.

```
add r1, r2          ; r1 ← r1 + r2 (2-주소 명령어)
add r1, r2, r3      ; r1 ← r2 + r3 (3-주소 명령어)
```

2-주소 명령어는 명령어에 포함된 피연산자가 2개이며, 2개의 피연산자 중 하나는 근원지 겸 목적지로 사용된다. 2-주소 명령어를 수행하면 근원지 겸 목적지로 사용하는 레지스터의 내용이 수정되기 때문에 근원지 내용 1개를 더 이상 사용할 수 없게 된다. 즉, add 명령어를 수행하기 전의 r1 값이 연산 결과에 의해 파괴된다. 그러나 피연산자 필드가 2개만 필요하므로 명령어의 길이가 짧아지는 이점이 있다. 3-주소 명령어는 2-주소 명령어에 비해 크기가 길어지지만 연산을 수행한 후에도 근원지 레지스터의 내용이 변하지 않는다.

범용 레지스터 컴퓨터가 다음 명령어만 메모리에 접근할 수 있도록 제한하는 경우 적재 · 저장 명령어 컴퓨터load-store machine라고 한다.

- **적재 명령어** : 데이터를 메모리에서 CPU 레지스터로 전송한다.
- **저장 명령어** : 데이터를 CPU 레지스터에서 메모리로 전송한다.

적재 · 저장 명령어 컴퓨터는 add 혹은 mul 명령어의 피연산자로 메모리 주소를 사용할 수 없다는 데 유의하라. 레지스터 주소가 메모리 주소보다 짧기 때문에 범용 레지스터 컴퓨터의 명령어 길이는 메모리-메모리 컴퓨터의 명령어 길이보다 짧다. 따라서 명령어를 위한 메모리 트래픽이 줄어든다. 또한 범용 레지스터 컴퓨터는 사용 빈도가 높은 데이터를 레지스터에 두기 때문에 데이터를 위한 메모리 트래픽이 줄어든다. 이는 전화기의 단축 버튼처럼 전체 전화번호를 사용하지 않고 한두 자리 버튼을 사용하는 경우와 유사하다. 범용 레지스터 컴퓨터의 대표적인 예는 최근 컴퓨터를 포함한 VAX 11, IBM 360 등으로, 오늘날의 컴퓨터는 대부분 적재 · 저장 명령어 컴퓨터에 속한다.

연산 과정에서 발생하는 트래픽을 분석하기 위해 누산기 컴퓨터의 경우와 동일한 가정에 다음을 추가한다.

- 최대 2개의 피연산자 사용, 즉 2-주소 명령어
- 16개의 레지스터 사용
- 적재 · 저장 명령어 컴퓨터

범용 레지스터 컴퓨터를 위한 가상의 어셈블리어를 사용하여 $y = ax^2 + bx + c$를 프로그래밍하면 [표 4-3]과 같다.

여기서 a, b, c, x, y는 데이터가 위치한 메모리 주소를 명시하는 변수이며, r1~r4는 레지스터를 의미한다. [표 4-3]에 나타난 바와 같이 모든 명령어는 2개의 명시적 피연산자를 가진 2-주

표 4-3 범용 레지스터 컴퓨터의 연산

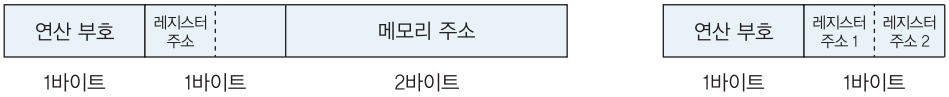
명령어		의미
연산 부호	피연산자	
load	r1 x	Reg[1] ← M[x]
load	r2 a	Reg[2] ← M[a]
load	r3 b	Reg[3] ← M[b]
load	r4 c	Reg[4] ← M[c]
mul	r2 r1	Reg[2] ← M[a] × M[x]
mul	r2 r1	Reg[2] ← M[a] × M[x]^2
mul	r3 r1	Reg[3] ← M[b] × M[x]
add	r3 r2	Reg[3] ← M[a] × M[x]^2 + M[b] × M[x]
add	r4 r3	Reg[4] ← M[a] × M[x]^2 + M[b] × M[x] + M[c]
store	r4 y	M[y] ← r4의 내용

소 명령어를 사용한다. 또한 적재 · 저장 명령어 컴퓨터로서 load와 store 명령어만 a, b 같은 주소를 사용하여 메모리에 접근한다. 그럼 위와 같은 프로그램을 범용 레지스터 컴퓨터에서 실행할 때 CPU와 메모리 사이에 발생하는 트래픽이 어느 정도인지 분석해보자.

명령어를 메모리에서 인출하는 데 필요한 트래픽 발생량을 먼저 살펴보자. load와 store 명령어는 [그림 4-4]의 (a)와 같이 연산 부호와 2개의 명시적 피연산자(레지스터 주소 1개와 메모리 주소 1개)를 가지고 있다. 레지스터가 16개이므로 4비트의 주소 비트가 필요하다. 그리고 가정에 따라 연산 부호가 1바이트이고 메모리 주소가 2바이트이므로 load와 store 명령어는 3바이트와 4비트로 구성된다. 그런데 컴퓨터의 최소 단위 정보는 바이트 단위이므로 load와 store 명령어의 길이는 4바이트가 된다. 또한 mul과 add 명령어는 [그림 4-4]의 (b)와 같이 1바이트의 연산 부호와 2개의 레지스터로 구성되기 때문에 명령어의 길이는 2(= 1 + 0.5 + 0.5)바이트이다. 결과적으로 load, store 명령어 5개와 mul, add 명령어 5개를 인출하는 데 CPU와 메모리 사이에 30(= 4 × 5 + 2 × 5)바이트의 트래픽이 발생한다.

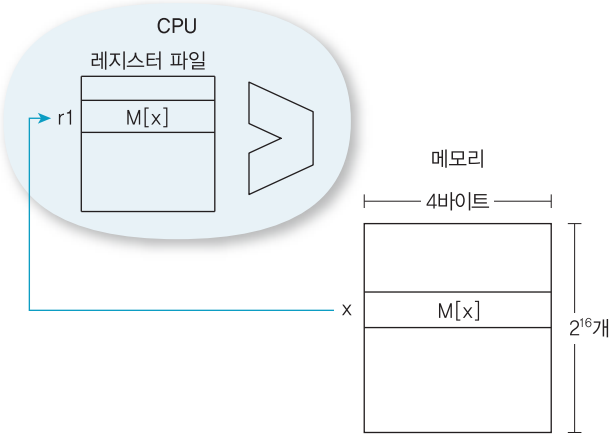
다음으로, 연산을 수행할 때 데이터 이동에 필요한 트래픽 발생량을 살펴보자. load와 store 명령어는 [그림 4-4]의 (c)와 같이 하나의 데이터, 즉 4바이트를 메모리와 레지스터 사이에 전송

한다. 그러나 mul과 add 명령어는 [그림 4-4]의 (d)와 같이 레지스터에 있는 데이터만 사용하므로 메모리에 접근할 필요가 없으니 데이터 트래픽이 발생하지 않는다. 따라서 load와 store 명령어 5개만 데이터 트래픽을 발생시키므로 총 데이터 트래픽은 20(= 4 × 5)바이트이다.

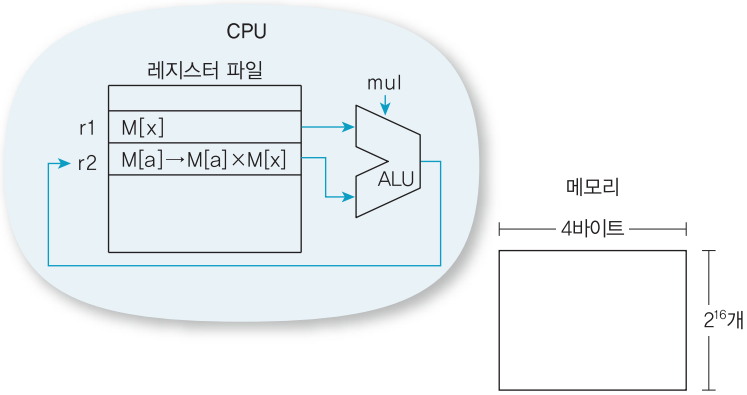


(a) 적재 · 저장 명령어 구성

(b) 적재 · 저장 외의 명령어 구성



(c) load r1, x 명령어 실행



(d) mul r2, r1 명령어 실행

그림 4-4 범용 레지스터 컴퓨터의 명령어 구성과 데이터 트래픽 발생

명령어 인출을 위한 30바이트와 데이터 전송을 위한 20바이트를 더하여 총 메모리 트래픽은 50바이트가 발생한다. 누산기 컴퓨터, 스택 컴퓨터와 비교해보면 범용 레지스터 컴퓨터의 메모리 트래픽이 가장 적다. 비록 $y = ax^2 + bx + c$에 대한 경우 메모리 트래픽이 크게 차이가 없지만, 실제 프로그램의 경우 범용 레지스터 컴퓨터는 다른 아키텍처에 비해 트래픽을 크게 감소시킨다. 따라서 오늘날의 컴퓨터는 범용 레지스터 컴퓨터 방식을 사용한다. 범용 레지스터의 수가 많으면 데이터에 대한 접근 시간이 짧아지고 메모리 트래픽이 줄어드는 효과가 있다. 하지만 범용 레지스터가 너무 많으면 다음과 같은 문제가 발생한다.

- 레지스터 주소가 길어져서 명령어의 길이도 늘어난다.
- 컴파일러가 모든 레지스터를 효과적으로 이용하기 어렵다.
- 문맥 교환context switching 비용이 증가한다.
- 레지스터 개수에 따라 하드웨어의 비용이 상승한다.

저자 한마디 **문맥 교환**

문맥 교환은 CPU를 사용 중인 하나의 프로세스가 다른 프로세스로 하여금 CPU를 사용할 수 있도록 자신의 프로세스 상태, 즉 문맥(프로그램 카운터, 스택 포인터, 레지스터 등의 내용)을 메모리에 보관하고 새로운 프로세스의 상태를 메모리에 적재하는 작업을 말한다. 프로세스의 문맥은 프로세스 제어 블록에 기록되어 있다. 문맥을 교환하는 동안에는 유용한 작업을 수행할 수 없기 때문에 CPU 내에 레지스터와 같은 기억장치가 많으면 더 큰 부담이 발생한다.

02 명령어와 메모리

명령어는 수행해야 할 연산에 대한 데이터 또는 데이터에 대한 정보를 명시해야 한다. 데이터는 일차적으로 메모리에 위치하므로 명령어는 메모리와 밀접한 관계가 있다. 이 절에서는 메모리의 속성, 메모리 내부의 데이터 정렬, 메모리에 데이터를 저장하는 순서인 엔디언 방식에 대해 살펴본다.

1 메모리 속성

명령어와 데이터는 컴퓨터의 메모리에 적재되어야 CPU가 사용할 수 있다. 따라서 명령어를 살펴보려면 메모리의 속성을 이해해야 하며, 메모리의 논리적 구성을 살펴보려면 먼저 메모리와 관련된 다음 세 가지 용어를 알아야 한다.

■ 워드

워드는 명령어와 데이터를 포함할 수 있는 메모리 구성 요소 중 하나이다. 워드는 고정된 개수의 비트라는 메모리 소자memory cell로 이루어진다. 기술의 발전과 컴퓨터의 사양에 따라 다르지만 대부분의 컴퓨터는 아직까지 하나의 워드가 32비트로 구성되어 있다. 초기의 컴퓨터는 하나의 워드가 8비트나 16비트였고, 심지어 4비트인 경우도 있었다. 최근에는 하나의 워드가 64비트인 컴퓨터도 많이 사용되고 있다.

■ 주소

메모리는 주소라는 고유의 번호를 사용하여 명령어나 데이터를 참조한다. 알기 쉽게 비유하자면 메모리는 그릇의 집합, 그릇 속의 내용물은 데이터, 그릇에 표시된 번호는 주소라고 할 수 있다.

예를 들어, 가게에 사과 16개가 있는데 1개씩 또는 8개씩 판매한다고 하자. 사과를 1개씩 판매하는 경우 낱개씩 포장해야 한다. 따라서 [그림 4-5]의 (a)와 같이 사과 포장 용기에 4비트의 주소가 필요하다. 한편 사과를 8개씩 판매하는 경우에는 8개 단위의 박스로 포장할 수 있다. 따라서 [그림 4-5]의 (b)와 같이 사과 포장 용기가 2개면 충분하므로 1비트의 주소가 필요하다. 그

런데 8개 단위로 포장하면 주소를 위한 비트가 적게 필요하지만 사과를 낱개로 거래할 수 없다. 즉, 큰 단위로 데이터를 다루면 주소가 짧아지지만 작은 단위의 데이터를 취급하기가 어렵다. 따라서 데이터 취급 단위와 주소 크기를 적절히 조정해야 한다.

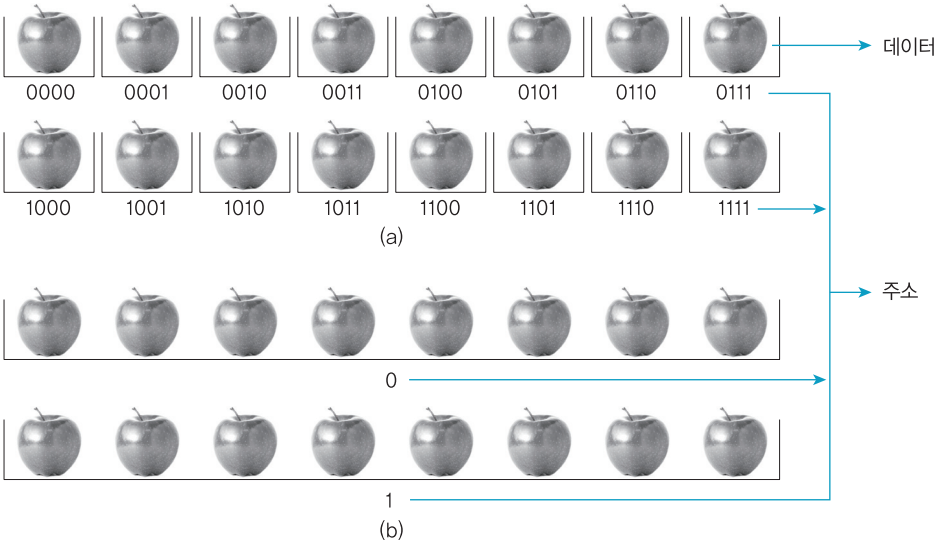


그림 4-5 데이터와 주소

■ 주소 지정 단위

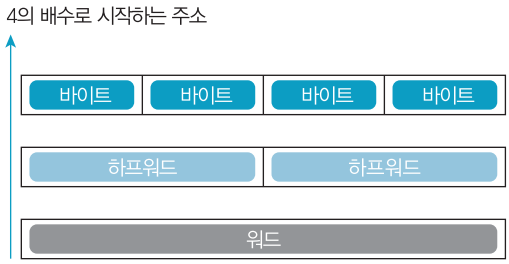
주소 지정 단위addressable unit는 그릇의 크기에 해당하며 주소 해상도address resolution를 결정한다. 주소 해상도는 아키텍처가 직접 명시할 수 있는 정보의 최소 단위를 의미한다. iAPX-432와 같은 일부 아키텍처는 최소 주소 지정 단위로 비트도 사용할 수 있었다. 이런 경우 비트 단위로 메모리에 직접 접근할 수 있지만 대부분의 아키텍처는 비트 단위의 주소 해상도를 사용하지 않는다. 비트 단위까지 주소를 분해하면 주소를 나타내는 데 많은 비트가 필요할 뿐만 아니라 데이터 정렬에 추가 시간이 필요하기 때문이다.

오늘날의 아키텍처는 대부분 8비트, 16비트, 또는 그 배수 단위의 주소 해상도를 사용한다. 8비트나 16비트 단위의 주소 해상도는 비트 단위의 주소 해상도에 비해 주소의 길이를 3비트나 4비트 줄일 수 있고 정렬 시간도 줄일 수 있다. 일반적으로 메모리의 주소 지정 단위는 8비트로 구성된 바이트를 사용하는데 이를 바이트 주소라고 한다. 주소를 구성하는 비트의 수는 메모리의 기본 단위에 대한 최대 개수를 결정한다. 만약 주소의 기본 단위가 바이트이고 주소 비트가 n이면 메모리는 최대 2^n개의 바이트를 포함할 수 있다.

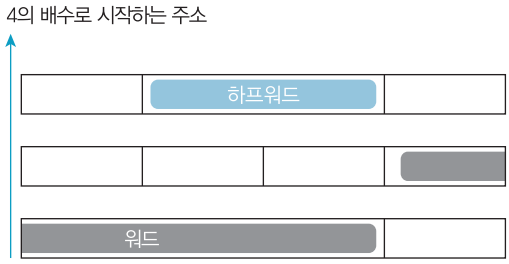
대부분의 아키텍처는 일관성consistency과 간결성simplicity을 위해 명령어와 데이터에 대해 동일한 주소 해상도를 사용한다. 하지만 iAPX-432와 같은 아키텍처는 명령어가 어떤 비트 경계에서도 시작될 수 있고, 명령어의 길이도 8비트 혹은 16비트의 배수가 아니다. 이런 경우 명령어의 밀도를 높일 수는 있지만, 명령어의 주소를 명시하기 위해 더 많은 비트를 사용해야 한다. 또한 어떤 비트 경계에서도 명령어가 시작될 수 있으므로 명령어를 인출하려면 한 번 이상 메모리에 접근해야 할 수도 있다. 이 때문에 오늘날의 컴퓨터는 비트 단위의 연산을 위해 특별한 명령어를 제공하지만 비트 단위의 주소 해상도를 사용하지 않는다.

2 메모리 정렬

정렬된 메모리aligned memory는 주소 해상도가 바이트라 할지라도 [그림 4-6]의 (a)와 같이 명령어와 데이터를 강제로 정렬하는 방식을 말한다. 즉, 2바이트로 구성된 명령어와 데이터는 2의 배수로 시작되는 주소를 갖고, 4바이트로 구성된 명령어와 데이터는 4의 배수로 시작되는 주소를 갖도록 강제하는 방식이다. 이는 컴퓨터를 구현할 때 실행 속도를 높이기 위한 요구 사항이다.



(a) 정렬된 데이터



(b) 정렬되지 않은 데이터

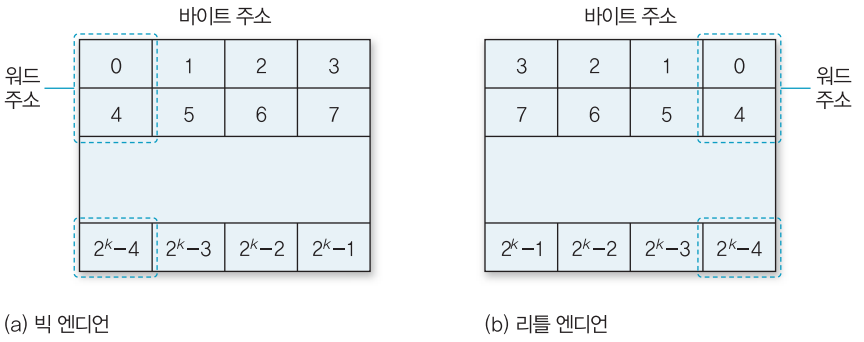
그림 4-6 데이터의 정렬

[그림 4-6]의 (b)와 같이 하프워드나 워드 데이터가 2의 배수나 4의 배수가 아닌 주소에 위치할 경우 데이터가 정렬되지 않았다고 한다. 특히 정렬되지 않은 워드 데이터를 적재하려면 2개의 메모리 읽기 사이클을 실행해야 한다. 반대로 워드 데이터가 정렬되어 있다면 1개의 메모리 읽기 사이클만 수행하면 된다. 따라서 대부분의 아키텍처는 명령어나 데이터를 정렬함으로써 성능을 높인다.

3 엔디언 방식

엔디언endian은 여러 개의 연속된 대상을 1차원 공간에 배열하는 방법을 말한다. 예를 들어, 32비트 컴퓨터의 경우 메모리 워드에 32개의 비트를 배열하거나 4개의 바이트를 배열하는 방법이다. 일반적으로 엔디언은 바이트를 배열하는 후자를 의미한다. 엔디언 방식에서 비트, 바이트, 워드의 순서는 어느 방법이든 성능에 차이가 없다고 알려져 있다. 그러나 컴파일된 실행 파일을 다른 아키텍처로 이식할 때는 워드 내부의 비트 혹은 바이트의 순서가 달라지기 때문에 의미가 있다.

바이트를 메모리 워드 내부에서 배열하는 방식은 다양하지만 주로 [그림 4-7]과 같은 빅 엔디언big endian 방식과 리틀 엔디언little endian 방식을 사용한다. 빅 엔디언 방식은 왼쪽에서 오른쪽으로 바이트를 배열하는 것으로, 사람이 숫자를 쓰는 방법과 마찬가지로 큰 단위의 바이트가 앞자리에 위치한다. 이 방식에서는 각 워드의 최대 유효 비트(MSBmost significant bit)가 포함된 바이트의 주소가 워드 주소가 된다. 이에 반해 리틀 엔디언 방식은 빅 엔디언 방식과 반대 방향으로 바이트를 배열하므로 작은 단위의 바이트가 앞자리에 위치하여 각 워드의 최소 유효 비트(LSBleast significant bit)가 포함된 바이트의 주소가 워드 주소가 된다. 각 워드의 주소는 빅 엔디언의 경우 가장 왼쪽 바이트의 주소이고, 리틀 엔디언의 경우 가장 오른쪽 바이트의 주소이다.



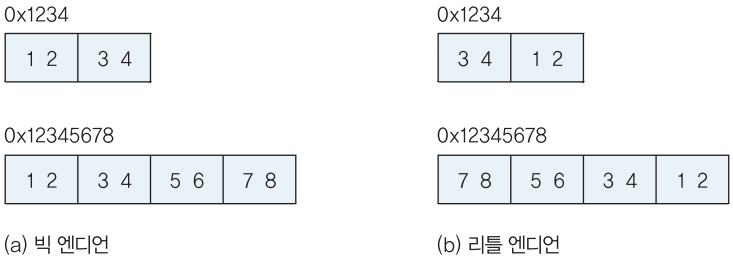
(a) 빅 엔디언

(b) 리틀 엔디언

그림 4-7 빅 엔디언과 리틀 엔디언

빅 엔디언 방식의 대표적인 예는 MIPS, IBM, Motorola, Sun SPARC를 비롯한 대부분의 RISC 아키텍처이며, 리틀 엔디언 방식의 대표적인 예는 Intel, DEC, VAX-11 등이다. ARM 프로세서를 포함한 오늘날의 아키텍처는 빅 엔디언과 리틀 엔디언을 선택할 수 있도록 옵션을 제공하기도 한다.

좀 더 구체적인 예를 통해 살펴보자. 두 가지 종류의 16진법 데이터 1234_{16}와 12345678_{16}을 빅 엔디언과 리틀 엔디언으로 표현하면 [그림 4-8]과 같다. 2개의 바이트로 구성된 1234_{16}의 경우, 빅 엔디언 방식은 12_{16}를 포함하는 바이트가 34_{16}를 포함하는 바이트보다 앞에 위치하지만, 리틀 엔디언 방식은 34_{16}를 포함하는 바이트가 12_{16}를 포함하는 바이트보다 앞에 위치한다.



(a) 빅 엔디언 (b) 리틀 엔디언

그림 4-8 빅 엔디언과 리틀 엔디언의 데이터

저자 한마디 빅 엔디언과 리틀 엔디언의 유래

빅 엔디언과 리틀 엔디언은 조너선 스위프트Jonathan Swift의 소설 『걸리버 여행기』에 나오는 소인국 릴리퍼트의 이야기 중 달걀의 뭉툭한 끝big-end을 먼저 깨는 사람과 뾰족한 끝little-end을 먼저 깨는 사람 사이의 격론에서 따온 이름이다.

03 주소 지정 방식

컴퓨터에서 연산을 수행하려면 필요한 데이터의 위치를 알아야 한다. 그런데 데이터를 저장하는 공간은 메모리, 레지스터 등 다양한 장소가 될 수 있다. 메모리의 경우 메모리를 구성하는 다수의 워드나 바이트 중에서 정렬 제약 사항을 준수한다면 데이터가 어디에든 저장될 수 있다. 따라서 명령어는 피연산자 필드를 사용하여 데이터의 위치에 대한 정보를 제공한다. 주소 지정 방식addressing mode은 명령어의 일부인 피연산자 필드를 사용하여 데이터가 실제 위치한 유효 주소effective address를 결정하는 방법이다. 현존하는 아키텍처는 효율적으로 데이터의 위치를 지정하기 위해 다양한 주소 지정 방식을 제공한다.

주소 지정 방식의 종류는 매우 다양하지만 모든 방식을 열거하는 것은 큰 의미가 없으므로 여기서는 주로 사용하는 방식에 대해 살펴본다. 주소 지정 방식은 피연산자 정보를 가지고 몇 단계를 거쳐 원하는 데이터에 접근할 수 있느냐에 따라 단계별로 분류할 수 있다. 일반적으로 주소 확정을 위한 단계가 깊을수록 데이터를 참조하는 데 시간이 오래 걸리지만, 더 많은 데이터를 다양한 방법으로 보관할 수 있다는 장점도 있다.

명령어의 피연산자 필드와 데이터 실제 위치의 관계를 나타내기 위해 다음과 같은 표기를 사용한다.

- **r** : 명령어 내의 피연산자 필드가 명시하는 레지스터 주소
- **a** : 명령어 내의 피연산자 필드가 명시하는 메모리 주소
- **ea** : 참조되는 데이터를 포함하는 장소의 실제 주소, 즉 유효 주소
- **Reg[x]** : 레지스터 x의 내용
- **M[x]** : 메모리 x번지의 내용

1 0-단계 주소 지정 방식

0-단계 주소 지정 방식은 데이터가 있는 위치를 파악하기 위해 특별한 과정이 필요 없는 방식으로 데이터의 유효 주소를 명시하지 않는다. 0-단계 주소 지정 방식은 크게 즉치 주소 지정 방식

과 묵시 주소 지정 방식으로 구분된다.

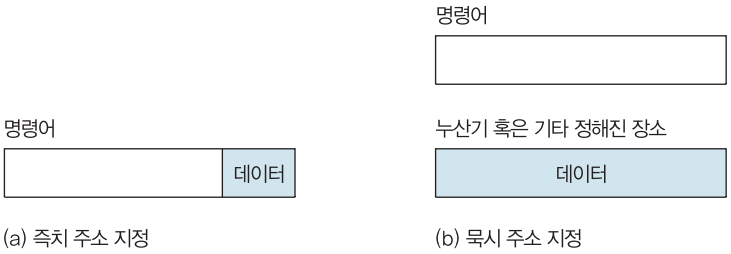


그림 4-9 0-단계 주소 지정 방식

■ 즉치 주소 지정

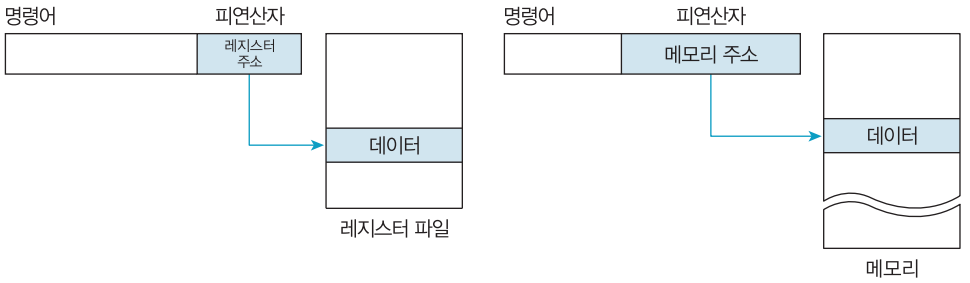
즉치 주소 지정immediate addressing mode은 즉시 주소 지정, 즉석 주소 지정, 상수 주소 지정 등으로도 불린다. 이 방식은 [그림 4-9]의 (a)와 같이 명령어의 피연산자 필드에 데이터가 포함되어 있으므로 주소 지정과는 상관이 없다. 따라서 명령어를 CPU로 인출만 하면 데이터가 바로 보이기 때문에 데이터를 즉시 이용할 수 있다. 그러나 명령어의 일부를 사용하므로 제한된 비트로 표현될 수 있는 0과 1 같은 작은 정수 데이터를 위한 공간으로 유용하다. 또한 명령어에 내장되어 있기 때문에 상수와 같이 변경하지 않는 데이터를 위해 사용된다.

■ 묵시 주소 지정

묵시 주소 지정implied addressing mode은 [그림 4-9]의 (b)와 같이 누산기나 프로그램 계수기 등과 같은 정해진 기억장치에 데이터가 저장된다. 이 방식은 항상 정해진 장소에 데이터가 있기 때문에 데이터의 위치를 명시할 필요가 없다. 그리고 유효 주소를 계산할 필요는 없지만 데이터를 사용하려면 하나의 기억장치를 읽어야 하며, 제한된 기억장치에 국한되기 때문에 많은 데이터에 대해 사용할 수 없다.

2 1-단계 주소 지정 방식

1-단계 주소 지정 방식은 데이터의 위치, 즉 유효 주소를 확정하기 위해 주소 계산이나 읽기 연산 과정을 한 번 수행하는 방식이다. 1-단계 주소 지정 방식부터는 데이터의 위치를 확정하기 위해 명령어의 피연산자 필드를 주소로 사용한다.



(a) 레지스터 직접 주소 지정 (b) 직접 주소 지정

그림 4-10 1-단계 주소 지정 방식

■ 레지스터 직접 주소 지정

레지스터 직접 주소 지정register-direct addressing은 레지스터 주소 지정register addressing이라고도 하며, [그림 4-10]의 (a)와 같이 데이터가 레지스터 파일 중의 한 곳에 있고 데이터가 있는 레지스터의 주소를 명령어의 피연산자 필드에 명시하는 방식이다. 이 방식은 높은 효율과 빠른 속도로 대부분의 아키텍처에서 사용한다. 레지스터 파일이 온칩on-chip으로 구현되어 메모리나 캐시보다 매우 빠르지만 용량이 작기 때문에, 자주 사용하는 소수의 데이터를 레지스터에 저장하여 빨리 접근하고자 할 때 사용한다. 레지스터 직접 주소 지정 방식은 레지스터 주소가 짧아서 짧은 피연산자 필드를 요구할 뿐만 아니라 데이터 접근 속도도 빠르다. 그러나 레지스터의 개수를 초과하는 데이터 집단을 취급할 때 효율적으로 사용하기 어렵다. 이 방식의 데이터 주소 확정 과정은 다음과 같이 나타낼 수 있다.

$$ea = r$$

■ 직접 주소 지정

직접 주소 지정direct addressing은 [그림 4-10]의 (b)와 같이 데이터가 있는 메모리의 주소를 명령어의 피연산자 필드에 명시하는 방식이다. 데이터를 참조하려면 피연산자 필드에 명시된 메모리 주소를 사용하여 메모리에 접근하면 된다. 메모리 주소 공간 전체를 사용하려면 메모리의 절대 주소 길이만큼 긴 피연산자 필드가 필요하므로 명령어가 너무 길어진다. 따라서 명령어의 길이를 줄이려면 메모리 주소 공간의 일부만 사용하거나 메모리 용량을 줄여야 한다. 직접 주소 지정 방식은 소형 시스템이나 초기 컴퓨터에 사용되기도 했지만 오늘날에는 거의 사용하지 않는다. 이 방식의 데이터 주소 확정 과정은 다음과 같이 나타낼 수 있다.

$$ea = a$$

3 2-단계 주소 지정 방식

2-단계 주소 지정 방식은 데이터의 유효 주소를 확정하기 위해 주소 계산이나 읽기 연산 과정을 두 번 수행한다. 이 방식은 데이터가 메모리에 저장되므로 많은 데이터의 위치를 명시할 수 있다. 그러나 데이터의 주소를 확정하기 위해 2-단계 과정을 거쳐야 하므로 0-단계 혹은 1-단계 주소 지정 방식보다 데이터를 참조하는 데 더 많은 시간이 걸린다. 대표적인 2-단계 주소 지정 방식은 레지스터 간접 주소 지정 방식과 메모리 간접 주소 지정 방식이며, 이를 조합한 색인 주소 지정, 베이스 주소 지정, 베이스-색인 주소 지정, PC 상대 주소 지정 방식 등도 있다.

■ 레지스터 간접 주소 지정

레지스터 간접 주소 지정register-indirect addressing은 [그림 4-11]의 (a)와 같이 피연산자 필드가 레지스터 주소를 명시하며, 레지스터의 내용이 데이터를 위한 메모리의 주소를 명시한다. 따라서 피연산자 필드에 명시된 레지스터를 읽은 다음, 레지스터의 내용이 가리키는 주소를 사용하여 메모리에 접근함으로써 데이터를 참조할 수 있다. 이 방식의 데이터 주소 확정 과정은 다음과 같이 나타낼 수 있다.

$$ea = Reg[r]$$

■ 메모리 간접 주소 지정

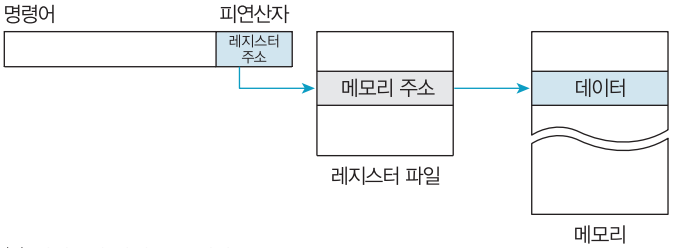
메모리 간접 주소 지정memory-indirect addressing은 [그림 4-11]의 (b)와 같이 데이터가 있는 메모리의 주소를 피연산자 필드에 명시하는 방식으로, 간략하게 간접 주소 지정indirect addressing이라고도 한다. 이 방식은 직접 주소 지정의 단점인 메모리 주소 공간이나 메모리 용량의 제약을 해결해주지만 데이터를 참조하기 위해 두 번에 걸친 메모리 접근이 필요하다. 따라서 오늘날의 컴퓨터 시스템에서는 사용하지 않는다. 이 방식의 데이터 주소 확정 과정은 다음과 같이 나타낼 수 있다.

$$ea = M[a]$$

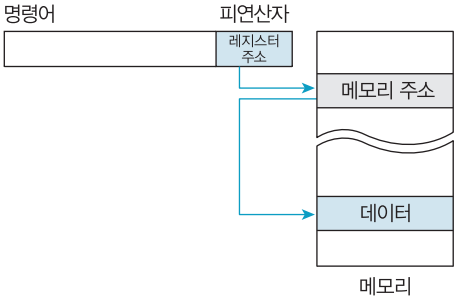
■ 변위 주소 지정

변위 주소 지정displacement addressing은 [그림 4-11]의 (c)와 같이 명령어에 포함된 2개의 피연산자 필드를 사용하여 데이터가 있는 메모리의 주소를 확정하는 방식으로, 색인 주소 지정과 베이스 주소 지정이 있다. 이 방식의 데이터 주소 확정 과정은 다음과 같이 나타낼 수 있다.

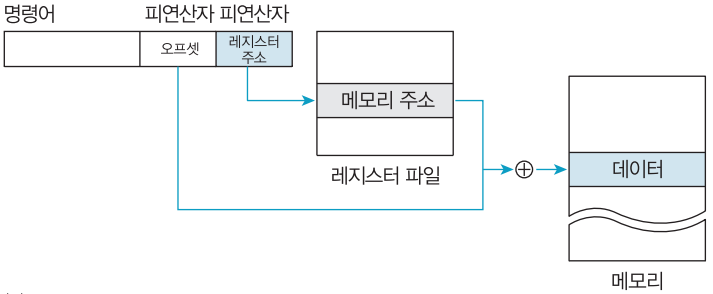
$$ea = Reg[r] + a$$



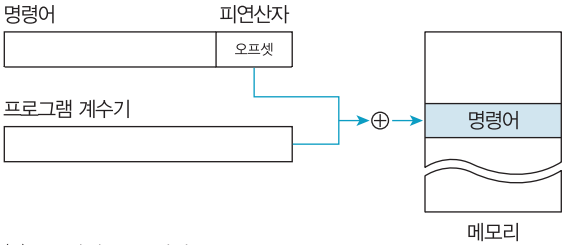
(a) 레지스터 간접 주소 지정



(b) 메모리 간접 주소 지정



(c) 변위 주소 지정



(d) PC 상대 주소 지정

그림 4-11 2-단계 주소 지정 방식

- **색인 주소 지정**indexed addressing : 2개의 피연산자 필드 중 하나는 베이스 주소로 사용할 메모리 주소를 포함하고, 다른 하나는 변윗값으로 사용할 색인 레지스터 주소를 명시한다. 데이터의 유효 주소를 얻기 위해 명령어에 포함된 2개의 피연산자 필드를 먼저 파악한 후 베이스 주소 값과 색인 레지스터의 내용을 더해야 한다. 그리고 이 덧셈 결과를 메모리 주소로 사용하여 메모리에 저장된 데이터를 참조한다. 이 방식은 배열 구조와 같이 컴파일 시간에 베이스 주소가 알려지지만 배열의 각 구성에 대한 위치가 실행 시간에 결정될 때 주로 사용한다. 색인 주소 지정 방식의 변형으로 선색인 간접 주소 지정pre-index indirect addressing과 후색인 간접 주소 지정post-index indirect addressing 등이 있다.
- **베이스 주소 지정**base addressing : 2개의 피연산자 필드 중 하나는 베이스 주소로 사용할 베이스 레지스터 주소를 명시하고, 다른 하나는 변윗값으로 사용할 메모리 주소를 명시한다. 색인 주소 지정의 경우 베이스 주소가 피연산자 필드에 있는 메모리 주소이지만, 베이스 주소 지정의 경우 베이스 주소가 베이스 레지스터가 가리키는 메모리 주소이다. 이 방식은 데이터 블록의 베이스 주소를 실행 시간에 알 수 있지만 데이터의 상대적 위치는 컴파일 시간에 알 수 있는 경우에 흔히 사용한다. 예를 들면, 함수를 사용할 때 전달되는 매개변수의 시작 주소는 컴파일 시간에 모르지만 매개변수 사이의 상대적 위치는 알 수 있는 경우이다.

■ PC 상대 주소 지정

PC 상대 주소 지정PC-relative addressing은 [그림 4-11]의 (d)와 같이 베이스 레지스터로 프로그램 계수기를 사용하는 베이스 주소 지정의 특별한 경우에 해당한다. 이 방식은 프로그램 계수기를 명시할 필요가 없으므로 변위 주소를 명시할 하나의 피연산자 필드만 있으면 된다. PC 상대 주소 지정 방식은 주로 분기 명령어에서 사용하는데, 대부분의 경우 분기할 명령어가 현재 참조된 명령어에서 가까운 곳에 있기 때문에 상대적인 변윗값으로 명령어의 피연산자 필드를 이용한다. 프로그램 계수기의 내용과 피연산자 필드에 있는 주소 변위를 합한 값이 분기할 명령어의 주소가 된다. 이 방식의 데이터 주소 확정 과정은 다음과 같이 나타낼 수 있다.

$$ea = PC + a$$

04 복잡도에 따른 명령어 집합

하드웨어가 이해할 수 있는 어휘를 의미하는 명령어 집합은 복잡도에 따라 크게 CISC 구조와 RISC 구조로 나뉜다. 2장에서 CISC 구조와 RISC 구조가 컴퓨터의 성능에 미치는 영향을 살펴봤지만 둘 중에서 어느 것이 더 낫다고 말할 수는 없다. 최근에는 각각의 장점을 도입하여 두 기술이 점차 통합되고 있는 추세이다. 이 절에서는 CISC와 RISC의 탄생 과정을 살펴보고 각각의 특징과 장단점을 비교해본다.

1 CISC의 탄생과 특징

초기의 컴퓨터는 다음 두 가지 문제를 해결하기 위해 명령어의 수가 더 많고 복잡한 방향으로 발전해 나갔다.

- 메모리가 매우 고가이고 속도가 느렸다.
- 프로그래밍 기술이 발전되지 않아서 소프트웨어 개발에 어려움이 있었다.

당시에는 메모리 용량을 작게 차지하는 프로그램을 구성할 수 있어야 좋은 아키텍처였다. 코드 밀도를 높이고, 소규모 라인의 프로그램으로 많은 작업을 수행하기 위해 강력한 명령어를 명령어 집합에 포함하는 추세였다. 특히 8장에서 소개할 마이크로프로그래밍의 등장으로 예전에 다수의 명령어로 처리할 기능을 하나의 강력한 명령어로 쉽게 구현하게 되었다. 이에 따라 메모리에서 다수의 명령어 대신에 하나의 명령어를 인출함으로써 메모리의 느린 속도에 따른 영향을 피할 수 있게 되어 실행 효율을 높일 수 있었다.

또한 프로그래밍 기술이 발전하기 이전이므로 소프트웨어 개발 비용도 만만치 않았다. 이를 해결하기 위해 강력하고 복잡한 고급 프로그래밍 언어가 개발되었고, 고급언어를 더 잘 지원하도록 강력한 명령어를 추가한 컴퓨터를 설계하게 되었다. 더 나은 고급언어를 지원하기 위해 추가된 대표적인 명령어는 VAX-11의 index, case, call 명령어와 MC68020의 chk 명령어 등이다. 또한 이보다 더 복잡한 VAX-11의 poly, crc 명령어, iAPX-432의 send 명령어 등도 있다.

이렇듯 많은 컴퓨터 제조 회사들은 새롭고 더욱 강력한 모델로 명령어 집합의 크기를 계속 증가시켰는데, 이와 같은 강력한 모델의 명령어 집합 구조를 복합 명령어 집합 구조(CISC complex instruction set computer)라고 한다. 그러나 높은 코드 밀도와 강력한 명령어를 추구하려면 [표 4-4]의 컴퓨터와 같이 가변 명령어 형식을 사용할 수밖에 없다. 가변 명령어 형식은 많은 종류의 주소 지정 방식을 동반하며, 또한 연산 부호를 해독할 때까지 명령어의 길이를 알 수 없다. 따라서 CISC 구조를 사용하면 명령어를 효율적으로 인출할 수 없고 데이터 전송 효율도 나빠진다.

표 4-4 일부 CISC 컴퓨터의 특징

구분	IBM 370/168	VAX-11/780	iAPX-432
도입 연도	1973	1978	1982
명령어의 종류	208	303	222
마이크로 코드 메모리의 용량	420KB	480KB	64KB
명령어의 크기	16~48비트	16~456비트	6~321비트

출처 : 『Computer Architecture Design』

2 RISC의 탄생

프로그램의 실행 궤적execution trace을 사용하여 CISC 구조의 복잡한 명령어와 주소 지정 방식이 실제로 얼마나 유효한가에 대해 많은 연구가 수년간 진행되었다. 연구 결과에 의하면 CISC 구조의 강력한 연산과 복잡한 주소 지정 방식의 사용 빈도가 매우 낮으며, 일부 데이터 형식은 거의 사용되지 않는 것으로 나타났다. 심지어 컴파일러도 CISC 구조의 강력한 명령어 집합을 충분히 사용하지 않고 일부만 사용할 가능성이 크다. 여기에는 여러 가지 이유가 있지만 다음과 같은 고급언어의 반복문 예를 살펴보자.

```
for (int i = 0; i < 100; i++) {
    ...
}
```

for 문장의 마지막에서 i의 값은 고급언어마다 다르다. 즉, i 값이 정의되지 않을 수도 있고, 마지막 반복될 때의 값인 100 또는 마지막 반복의 값보다 큰 101이 될 수도 있다. 따라서 모든 고급언어에 대한 for 문장에 대처할 수 있는 CISC 프로세서를 구현하는 것은 무의미하다. 결과적

으로 컴파일러는 일부 고급언어에 대해서는 for 문장을 for 명령어로 컴파일할 수 있지만 대부분의 고급언어에 대해서는 단순 명령어의 조합을 사용할 수밖에 없다.

복잡한 연산은 CPU 설계에 영향을 미치고, 복잡한 피연산자 구조는 메모리 구성과 주소 지정 방식에 영향을 미치며, 복잡한 프로그램 제어는 제어장치의 구성과 파이프라인pipeline 구현에 영향을 미친다. 복잡하고 강력한 명령어 집합은 오히려 간단한 명령어를 포함한 모든 명령어의 해석과 실행 시간을 증가시킨다. 복잡하고 강력한 연산을 거의 사용하지 않는데도 이와 같은 명령어를 명령어 집합에 포함하는 것이 바람직한지 컴퓨터 설계자들은 의문을 품게 되었고, 다른 시각으로 명령어 집합 구조에 접근하기 시작했다.

단순한 명령어는 하나의 사이클 내에서 실행될 수 있으며, 또한 짧은 사이클 시간도 허용한다. 따라서 컴퓨터 설계자들은 단순한 연산, 단순한 주소 지정 방식, 작은 상수의 즉치 피연산자 혹은 변윗값, 소수의 데이터 형식, 소수의 명령어 형식을 사용하는 컴퓨터를 설계하기 시작했다. 거의 사용하지 않는 비효율적인 복합 명령어를 제공하기보다 컴파일러로 하여금 기본 명령어의 최적 조합을 이용하도록 발상을 전환한 것이다. 즉, 명령어 집합 구조가 컴파일러에 대해 해결책 대신 기본 명령어를 제공하도록 했다.

아울러 메모리의 지속적인 가격 하락은 코드 밀도의 중요성을 떨어뜨리고, 메모리의 속도 향상은 폰노이만 병목에 대한 영향을 줄여준다. 궁극적으로 고급언어에 가까운 명령어를 제공함으로써 생기는 시맨틱 갭semantic gap의 축소는 성능에 악영향을 미칠 수 있다.

저자 한마디 **시맨틱 갭**

시맨틱 갭은 고급언어와 컴퓨터 연산의 차이를 의미한다. 예를 들어, 고급언어로 행렬 연산을 기술하더라도 실제로 컴퓨터에서는 선형 계산을 차례대로 수행한다. 이처럼 고급언어로 표현한 것과 실제로 수행한 연산이 상반되는데 이를 시맨틱 갭이라고 한다.

---

3 RISC의 특징

프로그램의 실행 특징에 의하면 강력한 명령어나 복잡한 주소 지정 방식 등은 사용 빈도가 낮기 때문에 하드웨어를 비효율적으로 사용한다. 이는 아키텍처와 컴퓨터 구현, 하드웨어와 소프트웨어, 컴파일 시간과 실행 시간의 상호 조정을 통해 성능을 향상할 수 있음을 시사한다. 또한 데이터의 임계 경로critical path에 대한 지연 시간 감소는 시스템의 성능 향상을 도모하는 것이다.

명령어, 주소 지정 방식, 데이터 형식 등의 종류가 많고 복잡하면 마이크로프로그램을 위한 제어 저장소로 많은 칩 면적이 필요해진다. 따라서 이를 줄이면 하드웨어 복잡도가 낮아지므로 속도가 빠른 고정결선식 제어(8장에서 설명)를 도입할 수 있다. 강력하고 복합적인 명령어가 아니라 단순하지만 더 빨리 실행되는 소수의 명령어를 사용하는 아키텍처를 축약 명령어 집합 구조(RISC[reduced instruction set computer])라고 한다. RISC 방식은 다음과 같은 특징을 가지고 있어 CPI와 사이클 시간이 줄어든다.

- **단순 명령어** : 전형적으로 하나의 사이클 내에서 실행되기 때문에 실행 속도가 빠르다. RISC 명령어는 마이크로 명령어[microinstruction]와 거의 일치하며, 복잡한 연산은 단순 명령어의 조합으로 구현한다.
- **짧은 사이클 시간** : 단순 명령어를 실행하는 데이터 경로가 간단하므로 명령어 실행에 필요한 사이클 시간이 짧다.
- **적재 · 저장 구조** : 적재 및 저장 명령어를 통해서만 메모리의 접근을 허용하며, 나머지 명령어는 메모리에 접근할 수 없고 오로지 레지스터 기반으로 연산을 한다. 자주 접근하는 데이터가 레지스터에 저장되어 있으므로 데이터 인출 시간이 짧아진다.
- **고정 길이 명령어** : 명령어의 길이가 고정되어 있으므로 명령어를 해독하기 전에 다음 명령어의 주소를 알 수 있다. 따라서 명령어 인출 과정이 단순하고 빠르다.
- **단순 명령어 형식** : 명령어의 형식이 단순하여 명령어를 빨리 해독할 수 있다.
- **제한된 종류의 주소 지정 방식** : 데이터의 유효 주소를 계산할 필요가 없거나 유효 주소에 대한 계산 속도를 향상할 수 있다.
- **하버드 아키텍처** : 명령어와 데이터가 독립적인 경로를 사용하기 때문에 메모리 대역폭이 증대된다.

이와 같은 특징을 가진 RISC 구조는 더욱 효율적인 최적화 컴파일러를 개발할 수 있도록 하며, 명령어 집합을 단순화하므로 제어장치를 간단하게 한다. 또한 9장에서 다룰 파이프라이닝을 더욱 효과적으로 구현할 수 있도록 한다. 오늘날 대부분의 컴퓨터는 RISC 접근 방식을 부분적으로나 전체적으로 이용하고 있으며, 대표적인 예로 IBM 801을 비롯해 버클리대학의 RISC I, RISC II, 스탠퍼드대학의 MIPS, 그리고 상용 아키텍처로 ARM, AMD29000, SPARC, MC88000, 인텔 80960 등이 있다.

4 RISC와 CISC의 전망

CISC 구조의 명령어는 고급언어에 가까운 구조로서 하드웨어를 강조하지만, RISC 구조의 명령어는 기계어에 가까운 구조로서 소프트웨어를 강조한다. 그러나 RISC 접근 방식이 도입된 후 RISC 프로세서가 CISC 프로세서보다 성능이 우수한가에 대한 의문이 생겼다. 2장에서 살펴본 컴퓨터의 성능에 의하면 최종적으로 고려해야 할 것은 CPI나 사이클 시간이 아니라 애플리케이션의 실행 시간이기 때문이다. 칩 밀도가 높아지고 하드웨어 속도가 빨라짐에 따라 [표 4-5]에 제시된 RISC 구조와 CISC 구조의 경계가 무너지고 있다. 칩 밀도가 매우 증가한 시점에 복잡한 명령어를 해독하기 위한 복잡한 회로도 더 이상 심각한 문제가 되지 않기 때문이다. 또한 효율적인 파이프라이닝 처리를 위해 CISC 구조도 복잡한 명령어를 내부적으로 단순 명령으로 분할하거나 RISC 기술을 적극 도입하고 있다. 이처럼 오늘날의 컴퓨터는 RISC 구조와 CISC 구조의 장점을 통합하는 추세이다.

표 4-5 RISC와 CISC

구분	RISC	CISC
명령어 형식	고정적	가변적
명령어 종류	적음	많음
명령어 길이	고정적	가변적
적재 · 저장 구조	사용	미사용
주소 지정 방식	단순하고 소수	복잡하고 다수
회로 구성	복잡함	단순함
장점	구현 용이, 파이프라이닝에의 효율적 적용	호환성 양호, 코드 밀도 양호
예	MIPS, ARM, PowerPC	인텔 x86, DEC VAX 11/780

05 picoMIPS 명령어 집합 구조

아키텍처를 학습하기 위해 하나의 명령어 집합 구조를 선택해야 한다. 그런데 RISC 구조는 CISC 구조에 비해 단순하여 쉽게 배울 수 있다. 따라서 이 책에서는 RISC의 많은 특징을 가지고 있는 32비트 MIPS 아키텍처를 모방한 picoMIPS를 기본 아키텍처로 사용한다. 비록 완전한 아키텍처는 아니지만 picoMIPS는 명령어의 실행 과정을 이해하거나 데이터 경로를 설계하기 위해 구상한 16비트 아키텍처이다. MIPS 아키텍처는 스탠퍼드대학에서 제안한 대표적인 RISC 프로세서 중의 하나이다. 이 절에서는 picoMIPS가 제공하는 명령어 형식과 명령어의 종류에 대해 살펴본다.

1 picoMIPS의 개요

전형적인 picoMIPS 명령어인 덧셈 명령어를 살펴보자. [그림 4-12]는 덧셈 명령어 add $1, $2, $3에 대한 고급언어와 기계어의 관계를 보여준다. add는 연산 부호로 덧셈 연산을 의미하고, $1~$3은 1~3번 레지스터를 가리킨다. 이 명령어는 2번 레지스터와 3번 레지스터의 내용을 더하여 덧셈 결과를 1번 레지스터에 저장하는 것이다. 이는 고급언어의 a = b + c를 컴파일한 결과에 대응하는 어셈블리어이며, 0000 010 011 001 010_2는 기계어이다.

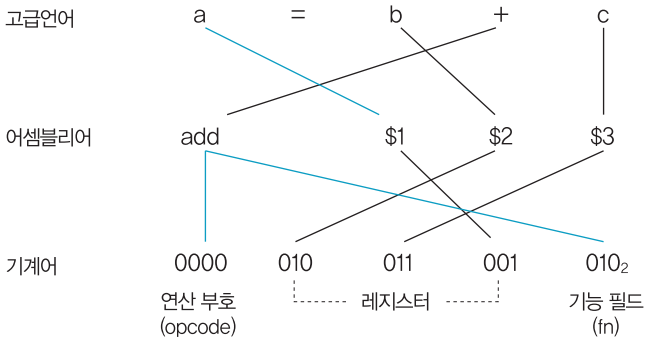


그림 4-12 전형적인 picoMIPS 명령어

picoMIPS 아키텍처는 다음과 같은 특징을 가지고 있다.

- 16비트 아키텍처로 워드의 길이가 16비트이다.
- 기본적으로 3-주소 명령어 형식을 사용한다.
- 적재 · 저장 명령어 구조이다. 따라서 적재 명령어와 저장 명령어만 메모리에 접근할 수 있고 나머지 명령어는 레지스터에 복제된 데이터만 처리할 수 있다.
- 바이트 순서는 빅 엔디언 방식을 사용한다. 워드 내의 비트 번호는 최소 유효 비트에서 시작하므로 가장 오른쪽 비트가 비트 번호 0이다.
- 데이터는 메모리에 정렬되어 저장된다. 따라서 명령어를 참조하거나 워드 크기의 데이터를 참조하려면 한 번의 메모리 접근만 필요하다.
- 레지스터가 8개이므로 레지스터의 주소를 명시하기 위해 3비트를 사용한다.

2 picoMIPS의 명령어 형식과 종류

모든 picoMIPS 명령어는 16비트로 구성된다. 그리고 [그림 4-13]과 같은 세 가지 명령어 형식을 사용하는데 이는 MIPS 명령어 형식과 유사하다.

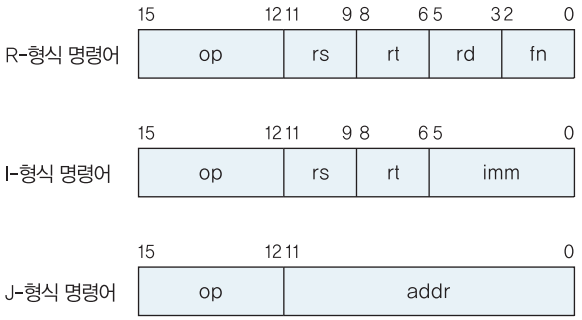


그림 4-13 picoMIPS의 명령어 형식

picoMIPS는 RISC 구조로 명령어 형식이 단순하고 일률적이다. 연산 부호가 세 가지 명령어 형식 모두에서 공통적이고 세 종류의 명령어를 구별하는 데 사용된다. [표 4-6]에 picoMIPS에서 사용하는 명령어 집합의 일부를 제시했다. 명령어의 9~11비트가 명시하는 레지스터 rs는 읽기 전용이며, 명령어의 6~8비트가 명시하는 레지스터 rt는 읽기 · 쓰기가 모두 가능하고, 명령어의 3~5비트가 명시하는 레지스터 rd는 쓰기만 가능하기 때문에 데이터 경로를 단순하게 설계할 수 있다.

표 4-6 picoMIPS 명령어의 예

명령어	형식	op	fn	의미
and $rd, $rs, $rt	R	0000	000	$rd = $rs and $rt
or $rd, $rs, $rt	R	0000	001	$rd = $rs or $rt
add $rd, $rs, $rt	R	0000	010	$rd = $rs + $rt
sub $rd, $rs, $rt	R	0000	011	$rd = $rs − $rt
mul $rd, $rs, $rt	R	0000	100	$rd = $rs × $rt
div $rd, $rs, $rt	R	0000	101	$rd = $rs / $rt
addi $rt, $rs, imm	I	1010		$rt = $rs + imm
subi $rt, $rs, imm	I	1011		$rt = $rs − imm
lw $rt, imm($rs)	I	0100		$rt = M[$rs + imm × 2]
sw $rt, imm($rs)	I	0101		M[$rs + imm × 2] = $rt
beq $rs, $rt, imm	I	0001		if($rs − $rt == 0) then PC += imm × 2
j addr	J	0011		PC += addr × 2

R-형식 명령어는 rs[register source]와 rt[register target] 필드에 명시된 2개의 레지스터를 사용하여 연산하며, 결과를 rd[register destination] 필드에 명시된 레지스터에 저장한다. 기능 필드(fn[function])는 연산 부호를 확장하기 위한 비트로 산술논리장치[ALU]가 수행할 연산을 정의한다. 따라서 R-형식 명령어는 다음과 같은 연산을 수행한다. 여기서 함수 f( )는 연산 부호와 기능 필드를 사용하여 해독된 연산을 의미한다.

$$Reg[rd] \leftarrow f(Reg[rs], Reg[rt])$$

I-형식 명령어는 6비트로 구성된 imm 필드를 포함한다. 이 필드는 즉치값을 명시하며 −32에서 31 사이의 상수를 나타낼 수 있다. I-형식 명령어는 명령어의 종류마다 다음과 같은 연산을 수행한다.

- **적재 명령어** : $Reg[rt] \leftarrow M[Reg[rs] + imm \times 2]]$
- **저장 명령어** : $M[Reg[rs] + imm \times 2] \leftarrow Reg[rt]$
- **조건 분기 명령어** : if(condition(Reg[rs], Reg[rt]) == true) then $PC \leftarrow PC + imm \times 2$

• **기타 명령어** : Reg[rt] ← f(Reg[rs], imm)

적재 · 저장 명령어나 조건 분기 명령어가 포함된 imm 필드는 모두 워드 단위로 메모리에 정렬되어 있기 때문에 워드 주소로 표현한다. 그러나 메모리는 바이트 주소를 사용한다. picoMIPS 아키텍처에서 하나의 워드는 2바이트이므로 imm 필드의 오른쪽 끝에 하나의 0을 추가하면, 즉 2를 곱하면 바이트 주소가 된다. 예를 들어, 조건 분기 명령어에서 imm 필드가 4(= 000100_2)라면 네 번째 워드를 말하기 때문에 바이트 주소로 8번지라는 의미이다. 적재 · 저장 명령어는 imm 필드를 오프셋 주소로 해석하며, 레지스터 rs의 값과 imm × 2를 합한 값을 읽기나 쓰기를 위한 메모리 주소로 사용한다. 조건 분기 명령어의 경우에는 rs와 rt가 가리키는 레지스터를 사용하여 조건을 점검한 후, 조건을 만족하면 프로그램 계수기의 내용을 갱신한다. 그리고 기타 명령어에 포함된 imm 필드는 R-형식 명령어의 rt 필드와 동일한 역할을 수행한다. 즉, 명시된 연산을 수행하기 위해 레지스터 rs의 내용과 imm 필드 값을 사용하며, 결과를 레지스터 rt에 저장한다.

J-형식 명령어는 무조건 분기 명령어에서 사용되며, 다음과 같이 addr 필드에 명시된 워드 주소 값을 사용하여 PC 상대 주소 지정 방식으로 분기한다. R-형식의 조건 분기 명령어와 마찬가지로 addr은 워드 주소를 의미하므로 다음과 같이 addr × 2를 사용하여 프로그램 계수기를 갱신한다.

무조건 분기 명령어 : PC ← PC + addr × 2

요약

1 명령어 내부에 명시적으로 나타난 피연산자의 수에 따라 0-주소 명령어, 1-주소 명령어, 2-주소 명령어 등으로 분류한다. 0-주소 명령어는 스택 컴퓨터에서 사용되고, 1-주소 명령어는 누산기 컴퓨터에서 사용되며, 2-주소 명령어나 3-주소 명령어는 범용 레지스터 컴퓨터에서 사용된다.

2 CPU 내부에 있는 소규모 기억장치라도 메모리 트래픽에 적지 않은 영향을 미치며, 결과적으로 명령어 집합 설계에 영향을 미친다.

3 범용 레지스터 컴퓨터 중에서 적재 명령어와 저장 명령어만 메모리에 접근할 수 있도록 제한하는 컴퓨터를 적재 · 저장 명령어 컴퓨터라고 한다.

4 범용 레지스터 컴퓨터는 메모리 주소에 비해 짧은 레지스터 주소를 사용하므로 명령어의 길이가 축소되고, 또한 사용 빈도가 높은 데이터가 레지스터에 있기 때문에 메모리 트래픽이 줄어드는 효과가 크다.

5 주소 지정 단위는 아키텍처에 의해 직접 명시될 수 있는 정보의 최소 단위를 명시하며, 주소 해상도를 결정한다.

6 메모리 정렬은 데이터와 명령어가 자신의 길이에 대한 배수의 주소에 위치하도록 강제하는 방식으로, 컴퓨터를 구현할 때 실행 속도를 높이기 위해 필요한 사항이다.

7 엔디언은 하나의 워드에 포함된 바이트를 배열하는 방법을 의미한다. 빅 엔디언 방식은 큰 단위의 바이트가 앞자리에 위치하고, 리틀 엔디언 방식은 작은 단위의 바이트가 앞자리에 위치한다.

8 주소 지정 방식은 명령어의 일부를 사용하여 데이터가 실제 위치한 유효 주소를 결정하는 방법이다. 즉치 주소 지정 방식은 데이터가 명령어 내부에 포함되어 있고, 묵시 주소 지정 방식은 누산기와 같이 지정된 장소에 데이터가 저장된다. 레지스터 직접 주소 지정 방식은 데이터가 있는 레지스터의 주소를 피연산자 필드에 명시하고, 직접 주소 지정 방식은 데이터가 있는 메모리의 주소를 피연산자 필드에 명시한다. 레지스터 간접 주소 지정 방식은 피연산자 필드가 레지스터 주소를 명시하며, 레지스터의 내용은 데이터를 위한 메모리의 주소를 명시한다. 명령어에 포함된 2개의 피연산자 필드를 사용하는 주소 지정 방식은 색인 주소 지정과 베이스 주소 지정이 있으며, PC 상대 주소 지정 방식은 베이스 레지스터로 프로그램 계수기를 사용하는 베이스 주소 지정의 특별한 경우에 해당한다.

9 RISC 구조는 자주 사용하는 소수의 명령어만을 명령어 집합에 포함하여 하나의 사이클에 실행시키는 방식의 아키텍처이고, CISC 구조는 강력하고 복잡한 명령어까지 명령어 집합에 포함하여 코드 밀도를 높이는 아키텍처이다. RISC 구조의 명령어는 기계어에 가까운 구조로서 소프트웨어를 강조하지만, CISC 구조의 명령어는 고급언어에 가까운 구조로서 하드웨어를 강조한다.

연습문제

1 다음 중 즉치 주소 지정 방식과 관계없는 것은?

① 상수 주소 지정 방식이라고도 한다.
② 명령어의 일부를 사용하여 데이터를 표현한다.
③ 고급언어에서 상수를 명시할 빈도수가 매우 낮기 때문에 효용성이 떨어진다.
④ picoMIPS에서 I-형식 명령어의 imm 필드가 사용한다.

2 다음 주소 지정 방식 중 속도가 가장 빠른 것은?

① 즉치 주소 지정
② 레지스터 직접 주소 지정
③ 메모리 간접 주소 지정
④ 색인 주소 지정

3 다음 중 0-주소 명령어를 사용하는 컴퓨터 구조는?

① 누산기 컴퓨터 ② 스택 컴퓨터
③ 마이크로컴퓨터 ④ 범용 레지스터 컴퓨터

4 다음 중 주소 지정 방식이 아닌 것은?

① 직접 주소 지정 ② 버퍼 주소 지정
③ 레지스터 주소 지정 ④ 즉시 주소 지정

5 다음 중 0-주소 명령어 컴퓨터와 가장 관련 있는 것은?

① 스택 ② 큐
③ 베이스 레지스터 ④ 색인 레지스터

6 다음 중 명령어 설계 과정과 가장 거리가 먼 것은?

① 연산 부호의 종류 ② 주소 지정 방식
③ 메모리의 대역폭 ④ 워드의 크기

7 다음 중 연산을 수행한 후에 입력 데이터가 모두 보존되는 방식은?

① 0-주소 명령어 컴퓨터 ② 1-주소 명령어 컴퓨터
③ 2-주소 명령어 컴퓨터 ④ 3-주소 명령어 컴퓨터

8 스택 컴퓨터에서 스택의 최상위 데이터는 ( ) 주소 지정 방식이다.

9 "CISC 구조는 복잡하고 강력한 연산을 단순 명령어의 조합으로 구현한다." 이는 맞는가, 틀린가?

10 피연산자 필드를 짧게 명시하는 것이 중요한 이유는 무엇이며, 오늘날의 컴퓨터는 어떤 방법을 사용하여 피연산자 필드를 짧게 하는가?

11 RISC 아키텍처의 주된 아이디어는 무엇인가?

12 대부분의 RISC 아키텍처에서 3-주소 명령어를 주로 사용하는 이유는 무엇인가?

13 주소 해상도는 무엇인가? 명령어와 데이터를 위해 동일한 주소 해상도를 사용해야 하는가? 비트 단위의 주소 해상도를 사용하지 않는 이유는 무엇인가?

14 레지스터 직접 주소 지정 방식이 흔히 사용되는 이유는 무엇인가?

15 데이터 1234_{16}를 빅 엔디언과 리틀 엔디언으로 표시하라.

16 4장 3절과 동일한 가정하에 3-주소 명령어를 사용할 때 범용 레지스터 컴퓨터에서 발생하는 메모리 트래픽을 분석하라.