Chapter

01

마이크로프로세서의 원리

Principles of Microprocessor

마이크로프로세서, 마이크로컨트롤러, 마이크로컴퓨터 _1.1
고급 언어와 저급 언어 _1.2
프로그램 실행 원리 _1.3
명령어 셋 _1.4
저장형 프로그램 컴퓨터 _1.5
RISC와 CISC _1.6

연습문제

학습목표

- 마이크로프로세서와 마이크로컨트롤러의 차이를 알 수 있다.
- 마이크로프로세서에서 프로그램이 수행되기 위한 명령어 형식과 명령어 실행 절차를 알 수 있다.
- 프로그램을 수행하려면 어떤 종류의 명령어가 필요한지 알 수 있다.
- 현대 컴퓨터의 기본 모델인 저장형 프로그램 컴퓨터의 한계점과 그 한계를 완화하려는 방법을 알 수 있다.
- CISC와 상대되는 RISC의 발전 계기와 RISC형 마이크로프로세서의 특징을 알 수 있다.

‘마이크로프로세서’, ‘마이크로컨트롤러’, ‘마이크로컴퓨터’라는 용어들은 서로 유사해 보이지만 조금씩 다르다. 이 장에서는 이 용어들에 대해 간단히 설명한 후, 연산 처리 기능을 향상하기 위한 마이크로프로세서 기술을 중심으로 살펴본다. 연산을 처리하기 위한 마이크로프로세서의 내부 구조는 어떻게 이루어져 있는지, 명령어와 주소 방식은 어떠한 관계를 맺고 있는지, RISC형 마이크로프로세서와 CISC형 마이크로프로세서는 어떤 차이가 있는지, 메모리 계층 구조가 필요한 이유는 무엇인지 등을 중심으로 간단히 살펴본다.

1.1 마이크로프로세서, 마이크로컨트롤러, 마이크로컴퓨터

마이크로프로세서

마이크로프로세서(microprocessor)는 마이크로(micro)와 프로세서(processor)가 결합된 용어로서 '매우 작은'(micro)이라는 의미와 '처리기'(processor)라는 의미를 포함하고 있다. 마이크로프로세서는 크기가 매우 작고, 뛰어난 계산 능력을 가진 장치로서, IC 집적기술, 컴퓨터 구조기술, 시스템 프로그래밍 기술을 함께 묶어 단일 칩으로 집적화한 반도체 소자다.

메모리, 디스크 메모리, 키보드, 모니터, 전원 장치 등과 같은 부가적인 부품들과 장치를 PCB(Printed Circuit Board)에 결합하여 만든 개인용 컴퓨터와 노트북의 마이크로프로세서는 자동차의 엔진과 같은 중요한 부품이다. 데스크톱 컴퓨터, 휴대용 노트북 등은 게임 수행, 문서 작성, 이미지 처리, 웹서핑 등과 같은 응용프로그램의 빠른 처리를 요구한다.

마이크로프로세서는 빠른 산술연산, 논리연산, 데이터 처리를 위한 구조로 최적화된 기술이 집약된 소자다. 1940년대 말 ENIAC의 진공관과 릴레이를 사용한 전기적인 컴퓨터가 개발된 이래 재료, 수학적 개념, 전자 집약기술, 사회적 요구를 수렴한 다양한 마이크로프로세서가 사용되고 있으며, 앞으로도 꾸준한 연구를 통해 더욱 향상된 마이크로프로세서가 등장할 것이다.

마이크로컨트롤러

마이크로컨트롤러(microcontroller)는 마이크로(micro)와 컨트롤러(controller)가 결합된 용어로서 '매우 작은'(micro)이라는 의미와 '제어기'(controller)라는 의미를 포함하고 있다. 마이크로컨트롤러는 크기가 매우 작고, 여러 가지 기능들을 쉽게 제어할 수 있는 장치로서 IC 집적기술, 컴퓨터 구조기술, 시스템 프로그래밍 기술, 손쉬운 제어 처리 기능을 함께 묶어 단일 칩으로 집적화한 반도체 소자다.

마이크로컨트롤러는 마이크로프로세서의 연산 처리 기능에 제어 기능을 추가한 것을 말한다. 하드웨어 측면에서 보면 산술 및 논리연산의 성능을 향상시키기 위한 회로와 제어에 필요한 회로를 포함한다. 또 외부 디지털 전압에 대한 입출력 기능, 메모리 기능, 타이머 기능, PWM(Pulse Width Modulation) 펄스 생성 기능, 입력신호 캡처 기능, A/D(Analog Digital) 변환 기능, 통신 기능 등도 포함될 수 있다. 이렇게 단일 칩에 많은 기능을 포함하면 외부

부가 장치 연결이 줄어들기 때문에 소형 제어시스템을 쉽게 만들 수 있다.

마이크로컨트롤러는 마이크로프로세서보다 더 많은 기능이 요구되지만, 값싼 장난감에서부터 산업용 장치에 이르기까지 넓은 범주를 대상으로 제어할 수 있다는 특징을 가지고 있다. 마이크로컨트롤러 제조 회사는 넓은 제어 범주를 가격과 기능별로 분류하고, 내부에 포함하는 제어 기능을 차별화하여 공급하고 있다. 또 부가 제어 기능에 따라 가격을 달리하여 응용 분야에 적합한 마이크로컨트롤러를 선택할 수 있도록 시리즈 형태로 제공하고 있다.

이상에서 살펴본 마이크로프로세서와 마이크로컨트롤러의 주요한 차이를 간단히 요약하면 다음과 같다.

■ **마이크로프로세서**

프로그램을 신속하게 실행하기 위한 목적으로, 내부 구조를 최적화한 것이다. 다양한 입출력 장치를 추가로 연결하고 자원을 효율적으로 관리하는 운영체제와 연동하면 마이크로컴퓨터가 된다.

■ **마이크로컨트롤러**

프로그램을 실행하면서 장치를 효과적으로 제어하기 위한 목적으로, 내부 구조를 최적화한 것이다. 다양한 입출력 장치를 내부에 포함하고 있기 때문에 외부에서 추가되는 장치를 줄일 수 있고, 제어 장치를 소형화할 수 있으며, 가격까지 낮출 수 있다는 장점이 있다.

마이크로컴퓨터

마이크로컴퓨터(microcomputer)는 '마이크로(micro)'와 '컴퓨터(computer)'가 결합된 용어로서 '매우 작은(micro)' 컴퓨터 시스템을 의미한다. 마이크로프로세서 또는 마이크로컨트롤러만으로는 프로그램을 실행할 수 없다. PCB에 있는 마이크로프로세서나 마이크로컨트롤러에 입출력 소자 또는 장치를 연결하면 매우 작은 컴퓨터, 즉 마이크로컴퓨터가 완성된다. 마이크로컴퓨터는 상대적으로 대형 슈퍼컴퓨터와 대비되는 용어로서, 그 대표적인 예로는 데스크톱 컴퓨터, 휴대용 노트북, 간단한 제어용 컴퓨터 및 PMP, 스마트폰을 들 수 있다.

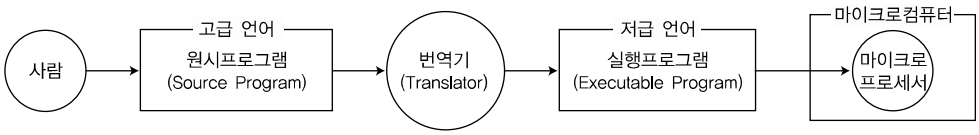
1.2 고급 언어와 저급 언어

1.2.1 번역기의 필요성

사람과 사람은 말을 통해 의사를 전달한다. 서로 다른 말을 쓰는 사람들이 의사전달을 할 때는 공통 언어를 사용한다. 만약 어느 한 쪽이 공통 언어를 모른다면, 통역사를 통해 의사를 전달해야 한다. 그렇다면 사람은 컴퓨터에 어떻게 의사를 전달할까?

컴퓨터에 내장된 마이크로프로세서는 2진(0 또는 1)값을 갖는 명령어가 차례대로 실행되면서 작업을 처리한다. 여기서 명령어란, 컴퓨터 시스템이 수행할 연산과 이에 사용될 데이터를 정의한 단위를 뜻하며, 원하는 작업을 처리하기 위해 이들 명령어를 차례로 배열한 것을 '프로그램'이라고 한다.

[그림 1-1]은 프로그래머(Programmer)가 작성한 프로그램을 마이크로프로세서가 수행할 수 있는 실행프로그램으로 변환하는 과정을 나타낸 것이다. 하지만 프로그래머가 2진 명령어로 프로그램을 직접 나열하는 것은 매우 번거로운 일이다. 프로그래머는 특별한 경우를 제외하고 사람의 말과 유사한 형식을 갖고 있기 때문에 논리적으로 프로그래밍하기 쉬운 고급 언어(High level language)를 이용하여 원하는 작업을 원시프로그램(Source Program)으로 작성한다. 원시프로그램은 통역사에 해당되는 번역기(Translator)에 의해, 저급 언어(Low level language)인 2진 명령어로 변환된 후 컴퓨터가 인식할 수 있는 기계어(Machine language)의 실행프로그램(Executable Program)으로 다시 변환된다.



[그림 1-1] 사람의 의사를 마이크로프로세서로 전달하는 과정

기계어는 마이크로프로세서 내부의 실행 동작을 명령어 단위로 표현한 것이기 때문에 이 명령어를 이해하면 마이크로프로세서 내부 구조를 쉽게 이해할 수 있다.

1.2.2 기계어와 어셈블리어

기계어와 어셈블리어는 마이크로프로세서 구조에 근거하여 구성된 컴퓨터 언어로서, 사람의 언어 형식과 거리가 멀고, 컴퓨터 구조와 밀접하므로 저급 언어로 분류된다.

기계어(Machine Language)는 특정 비트에, 특정 의미가 있는 2진값을 설정하는 명령어를 나열한 것으로, 원하는 동작을 프로그래밍하여 기계에 직접 사용할 수 있다는 특징이 있다. 기계어의 명령어 길이는 명령어 셋 구조(Instruction Set Architecture)에 따라 8비트, 16비트 또는 32비트로 이루어져 있다.

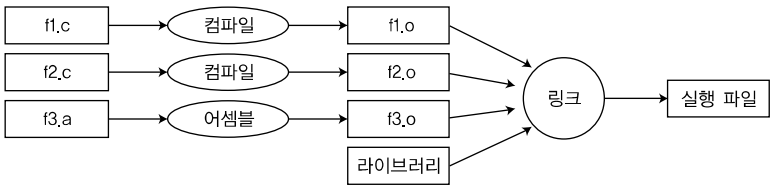
어셈블리어(Assembly Language)는 0과 1로 나열되는 기계어를 사람이 연상하기 쉬운 니모닉(Mnemonic)과 연산 대상이라는 영문 단어로 바꿔 표현한 것을 말한다. 이는 명령 동작을 대표하는 단어 또는 연산 대상을 대표하는 단어를 이용하여 기계어를 효과적으로 표현할 수 있다. 어셈블리어로 작성된 원시프로그램은 어셈블러(Assembler)라는 번역기(Translator)를 이용하여 기계어로 변환한다. 아래 [표 1-1]은 어셈블리어로 작성된 원시프로그램을 기계어로 변환한 명령어를 나타낸 것이다.

[표 1-1] 어셈블리어와 변환된 기계어

어셈블리어	기계어
movw r30, r22	1111 1101 0000 0001
subi r30, 0x00	1110 0000 0101 0000

1.2.3 원시프로그램에서 실행 파일 생성

프로그래머는 특별한 경우를 제외하고 C 언어와 같이 사람이 다루기에 편리한 논리적인 체계와 예약된 단어를 사용하는 고급 언어를 이용하여 원시프로그램을 작성한다. 특별히 하드웨어 정밀 제어가 필요한 부분은 어셈블리어로도 작성할 수 있다. [그림 1-2]는 여러 개의 원시프로그램에서 실행 파일이 생성되는 과정을 나타낸 것이다.



[그림 1-2] 원시프로그램에서 실행 파일이 생성되는 과정

컴파일과 어셈블

컴파일은 고급 언어로 작성된 원시프로그램을 기계어의 목적 파일(Object File)로 번역하는 과정을 말하는데, 이 번역 과정을 전담하는 프로그램을 '컴파일러(Compiler)'라 한다. 어셈블리어로 작성된 원시프로그램을 어셈블러(Assembler)를 이용하여 어셈블하면 목적 파일이 생성된다. 여기서 어셈블(assemble)이란, 어셈블리어로 작성된 프로그램을 기계어로 변환하는 것을 말한다.

링크

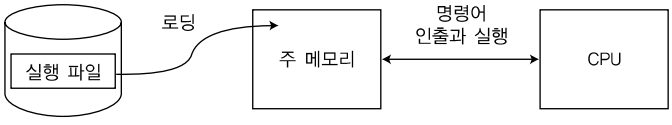
여러 개의 파일로 구성된 원시프로그램을 하나의 실행 파일로 만들기 위해서는 원시프로그램의 다른 파일 사이에 변수 또는 함수를 참조할 수 있어야 한다. [그림 1-2]에서 f1.o와 f2.o 목적 파일은 f1.c와 f2.c에서 표현된 다른 파일 간의 변수 및 함수 참조가 연결될 수 있는 정보를 저장한 채 기계어로 변환된 상태다. 이를 연결하여 통합된 실행 파일로 만드는 과정을 '링크(link)'라 한다. 링크를 전담하는 프로그램을 '링커(linker)'라 한다.

라이브러리

프로그래머는 실행을 위한 프로그램을 모두 작성할 수 있지만 같은 기능을 매번 다시 작성해야 하는 불편함이 있다. 따라서 공통으로 자주 사용하는 기능은 표준화된 이름과 매개 변수가 있는 함수로 작성한 후, 컴파일하여 목적 파일을 생성하고 링크 과정에서 이를 연결하여 사용한다. 컴파일된 목적 파일을 모아놓은 파일을 '라이브러리 파일'이라 한다. 보통 sin, cos, tan, printf 등과 같은 함수는 컴파일러를 제공하는 회사에서 공급한다.

실행 파일과 로딩

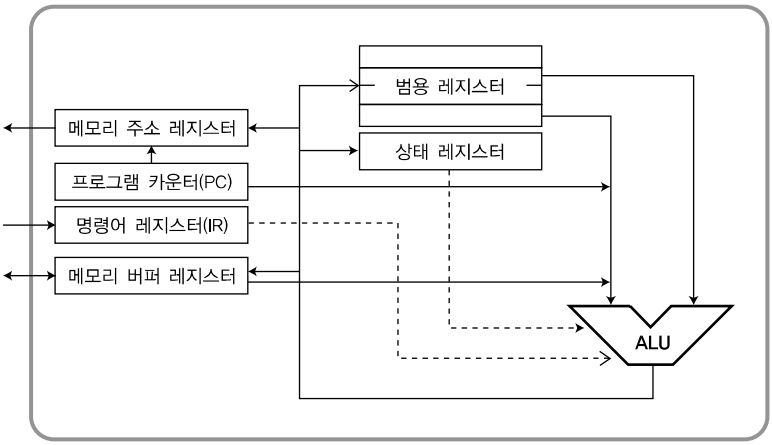
원시프로그램이 컴파일되어 완성된 실행 파일은 디스크 메모리에 존재한다. 실행 파일이 중앙 처리 장치(CPU, Central Processing Unit)에서 실행되기 위해서는 주 메모리에 탑재되어야 하는데, 이 과정을 '로딩(Loading)'이라 하고, 이를 전담하는 프로그램을 '로더(Loader)'라 한다. [그림 1-3]은 디스크 메모리에 있는 실행 파일을 주 메모리로 로딩하고, CPU에서 인출하여 실행하는 경로를 나타낸 것이다.



[그림 1-3] 실행 파일의 로딩과 실행

1.3 프로그램 실행 원리

프로그램은 마이크로프로세서에서 명령어가 차례대로 실행하면서 수행된다. [그림 1-4]는 3개의 버스 구조를 갖는 CPU에서 데이터를 연산 처리하는 구조를 나타낸 것이다. 특수 레지스터, 범용 레지스터 그리고 CPU 외부의 메모리 사이에 ALU(Arithmetic Logic Unit)를 통한 연산을 통해 명령어가 실행된다.



[그림 1-4] CPU 내부 구조

1.3.1 특수 레지스터와 범용 레지스터

마이크로프로세서 내부에는 데이터 연산을 위해 사용되는 범용 레지스터(General register)와 특수 레지스터(Special register)가 있다.

범용 레지스터

CPU 연산을 빠르게 처리하기 위해 ALU와 직접 연결되어 있으며, 연산 대상이 되는 오퍼랜드값을 가진다.

특수 레지스터

마이크로프로세서에서 명령어를 실행할 때 필요한 범용 데이터가 아닌 특수한 데이터를 처리하기 위한 레지스터를 말한다. [그림 1-4]에 나타낸 프로그램 카운터, 명령어 레지스터, 상태 레지스터, 메모리 주소 레지스터, 메모리 버퍼 레지스터가 이에 해당한다.

■ **프로그램 카운터** (PC, Program Counter)

인출할 명령어가 있는 메모리의 주소를 갖는 특수 레지스터를 말한다. 프로그램 메모리에서 한 개의 명령어 인출이 끝나면, 그 명령어의 크기가 더해진 값으로 자동 변경되어 다음 명령어 인출을 위한 주소를 갖게 된다.

■ **명령어 레지스터** (IR, Instruction Register)

프로그램 메모리에서 인출된 명령어를 기억하고 있는 특수 레지스터를 말한다. 인출된 명령어에는 특정 비트에 연산 동작(opcode), 연산 대상(operand), 연산 결과(result) 어드레싱 모드 등의 정보를 설정하고 있기 때문에 명령어를 실행하는 과정에 사용된다.

■ **상태 레지스터** (SR, Status Register)

명령어를 실행한 후의 연산 결과 정보는 상태 레지스터에 기록된다. 기록된 상태 레지스터의 각 비트는 연속되는 다음 명령어 실행에 영향을 끼친다. 예를 들어, 아래의 간단한 C 언어 프로그램을 살펴보자. 오른쪽은 어셈블리어로 번역된 코드다.

[프로그램 1-1] C 언어 프로그램의 예

1	if(a >= 100)	cpi r24, 0x64
2	a = 0x10;	brcs .+6
3		ldi r24, 0x10

(1) cpi(compare immediate) 명령을 수행하고, 결과는 상태 레지스터의 캐리 비트(carry bit)에 저장된다. cpi 명령을 수행하면, 변수 a가 저장된 레지스터 r24에서 100(16진수로 0x64에 해당되는 값)을 빼고 그 과정에서 오버플로가 발생하면 캐리 비트값은 1로 변한다.
(2) brcs .+6 명령은 상태 레지스터의 캐리 비트값이 1이면, 현재의 프로그램 카운터보다 6바이트 후의 분기된 곳을 실행한다(연산 결과에 따라 실행되는 위치가 달라진다).
(3) 2번 라인에서 캐리 비트값이 0이면, 연속된 ldi 명령을 수행한다.

■ **메모리 주소 레지스터** (MAR)**와 메모리 버퍼 레지스터** (MBR)

명령어 실행 과정에 필요한 특수 레지스터라기보다는 데이터 또는 주소를 임시로 저장하기

위한 특수 레지스터다. 외부 메모리와 데이터를 입출력하기 위해서 메모리 주소 레지스터(MAR, Memory Address Register)에 메모리 주소를 놓고 메모리에서 출력되는 데이터를 임시로 메모리 버퍼 레지스터(MBR, Memory Buffer Register)에 넣어 사용한다.

1.3.2 명령어 실행

프로그램은 명령어의 집합으로서 메모리에 있고, 데이터는 연산 대상(operand) 또는 연산 결과(result)로서 다른 메모리 공간에 있게 된다. 이를 각각 '프로그램 메모리'와 '데이터 메모리'라 한다.

명령어 실행 단계

마이크로프로세서에서 명령어가 실행되는 단계는 다음과 같다.

❶ 명령어 인출 단계

프로그램 카운터(PC)가 지시하고 있는 프로그램 메모리에 있는 명령어를 인출(Fetch)하여 명령어 레지스터(IR)에 넣는 단계다. 명령어는 연산 동작(opcode), 연산 대상(operand), 연산 결과(result)를 지칭하는 비트 정보로 구성되어 있다.

❷ 명령어 분석 단계

명령어 레지스터(IR)에 저장된 정보를 디코딩하여 레지스터 파일, 버스 라인 등에 필요한 신호를 생성한다. CPU에서는 여러 개의 범용 레지스터를 사용하는데, 이를 '레지스터 파일(Register file)'이라 한다.

❸ 오퍼랜드 인출 단계

디코딩 결과에 따라 연산에 사용될 오퍼랜드(operand)의 위치가 결정된다. 오퍼랜드는 명령어 자체에 포함되어 있거나, CPU 내부의 범용 레지스터에 있거나, 메모리에 있을 수 있다. 메모리에 있는 경우 오퍼랜드를 메모리로부터 인출하여 CPU 내부의 메모리 버퍼 레지스터로 옮기는 단계가 오퍼랜드 인출 단계다. 메모리로부터 데이터를 CPU 내부로 옮기기 위해서는 부가의 클록이 요구되고 명령어 실행 시간이 길어진다. 오퍼랜드가 명령어에 포함되어 있거나 레지스터에 있는 경우, 이 단계는 생략된다.

❹ 연산 실행 및 결과 생성 단계

명령어 레지스터의 연산 동작(opcode)에 따라 조합 논리 회로로 구성된 ALU에서 범용 레지

스터값, 명령어에 포함된 연산 대상값 또는 메모리에서 읽은 오퍼랜드값 등을 사용하여 산술 또는 논리연산한 값을 어큐뮤레이터(Accumulator) 또는 범용 레지스터로 결과를 기록하는 단계다.

❺ 결과 기록 단계

명령어를 디코딩한 결과 연산된 결과를 저장하는 위치가 메모리이면, 이 단계가 수행된다. 모든 명령어에 이 단계가 포함되지는 않는다.

명령어 인출-실행 사이클

명령어 실행 단계 중에서 오퍼랜드 인출 단계를 별도의 명령어 load, 결과 기록 단계를 별도의 명령어 store로 두면 명령어의 길이를 짧게 할 수 있다. 또 명령어 유형에 따라 실행 단계를 단축할 수 있다. 이런 특징을 활용한 것이 RISC(Reduced Instruction Set Computer) 구조다. RISC에 대해서는 1.6절을 참고하기 바란다.

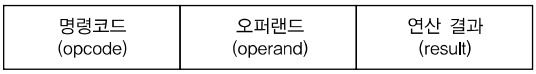
프로그램은 명령어가 메모리에서 인출(Fetch)되어 실행(Execution)되는 사이클이 반복되면서 수행된다. 이를 '명령어 인출-실행 사이클(Fetch-execute cycle)'이라 한다.

1.4 명령어 셋

명령어 셋은 명령어의 집합으로서 마이크로프로세서의 특징을 대표적으로 나타낸다. 명령어에 많은 내용이 포함될수록 다양한 기능을 수행할 수 있지만 명령어의 길이와 명령어 인출 시간 또한 길어지게 된다. 반대로 명령어 길이를 짧게 하면 같은 기능을 위해 여러 명령어를 수행시켜야 하는 단점이 발생한다.

1.4.1 3주소, 2주소, 1주소, 0주소 마이크로프로세서

명령어에 포함되는 비트 필드(bit field)는 [그림 1-5]와 같이 명령코드(opcode), 오퍼랜드(operand), 연산 결과(result)로 이루어져 있다.



[그림 1-5] 명령어 형식

명령코드는 해당 명령어로 실행되어야 할 연산 동작을 나타낸다. 산술연산 또는 논리연산에 해당하는 연산 동작은 조합 논리 회로로 구성된 ALU에서 수행된다. 명령코드 필드의 비트는 ALU에서 수행될 연산 회로를 결정하는 값으로, ALU의 동작을 선택하는 입력 신호에 해당된다. 한편, 오퍼랜드는 연산에 필요한 데이터를 나타낸다. [그림 1-5]의 연산 결과(result)는 명령코드와 오퍼랜드로 얻은 값을 저장할 위치를 나타낸다. 마이크로프로세서는 연산 대상인 오퍼랜드 인출과 연산 결과 저장을 위한 3주소, 2주소, 1주소, 0주소 필드를 갖는 명령어 유형으로 구분할 수 있다.

3주소 마이크로프로세서 명령어

'ADD a, b, c'는 3개의 오퍼랜드 주소 a, b, c로 주소 a의 데이터와 주소 b의 데이터를 ADD한 후 주소 c에 결과를 기록한다. 하나의 주소가 10비트를 차지하면 3개의 주소는 30비트가 필요하다. CPU는 아래와 같이 메모리에서 a, b 오퍼랜드를 인출하여 ADD한 뒤 결과 오퍼랜드 c를 메모리에 저장한다.

```
c ← a + b
```

2주소 마이크로프로세서 명령어

'ADD a, b'는 2개의 오퍼랜드 주소 a, b로 주소 a의 데이터와 주소 b의 데이터를 ADD하여 주소 b에 결과를 기록한다. 3주소 방식보다 명령어의 길이가 짧다. CPU는 아래와 같이 메모리에서 a, b 오퍼랜드를 인출하여 ADD한 뒤 결과를 오퍼랜드 b 메모리에 저장한다.

```
b ← a + b
```

1주소 마이크로프로세서 명령어

'ADD a'는 특수 레지스터 어큐뮤레이터(Accumulator)를 묵시적으로 이용하여 연산을 수행한다. 어큐뮤레이터에 연산의 중간 결과값을 갖게 하여 명령어에 언급되는 주소 수를 줄이는 효과가 있다. CPU는 아래와 같이 메모리에서 a 오퍼랜드를 인출하여 AC와 ADD한 뒤 결과를 AC에 저장한다.

```
AC ← a + AC
```

0주소 마이크로프로세서 명령어

0주소 마이크로프로세서 명령어 'ADD'는 오퍼랜드용 주소를 따로 두지 않는다. 그러나 스택(stack)이라는 곳에 연산 대상의 데이터를 미리 가져다 놓아야 한다. ADD 연산은 Push되어 있는 2개의 오퍼랜드가 필요하므로 묵시적으로 스택의 Top에서 2개의 데이터를 꺼내(Pop) 연산한 후, 결과를 스택의 Top에 넣는다(Push).

스택에서 오퍼랜드를 읽는 주소를 '스택 포인터(SP)'라 한다. 여기서 [SP]는 스택 포인터 주소 SP에 있는 메모리의 데이터를 나타낸다. CPU는 아래 절차에 의해 연산된다. #a는 명령어에 기록되어 있는 데이터를 나타낸다.

명령 순서	연산 동작
PUSH #a	SP ← SP + 1, [SP] ← #a
PUSH #b	SP ← SP + 1, [SP] ← #b
ADD	AC ← [SP], SP ← SP − 1 AC ← AC + [SP] [SP] ← AC

1.4.2 어드레싱 모드

마이크로프로세서에서 연산을 위한 오퍼랜드를 지칭하는 방법인 어드레싱 모드(Addressing mode)는 아래와 같이 구분할 수 있다.

즉시 주소방식

명령어에 연산 대상이 되는 데이터값을 갖고 있다. 이때 명령어 레지스터(IR)의 데이터 비트는 ALU의 연산 데이터 경로로 연결되어 즉시 사용된다. 아래 명령은 명령어에 포함된 2진 데이터 23을 오퍼랜드로 즉시 사용하는 즉시 주소방식을 나타낸 것이다.

명령어 예시	연산 동작
ADD #23	AC ← AC + 23

직접 주소방식

명령어에 포함된 주소는 주 메모리의 연산 대상 데이터에 대한 주소를 갖는다. 명령어 주소 필드의 비트에 메모리 주소가 있기 때문에 메모리에서 직접 오퍼랜드를 인출할 수 있다. [a]는 주소 a에 있는 메모리 데이터를 나타낸다.

명령어 예시	연산 동작
ADD a	AC ← AC + [a]

간접 주소방식

명령어에 포함된 주소는 오퍼랜드를 직접 인출할 수 있는 주소가 아니다. 명령어에 기록된 주소로 인출한 값은 실제 오퍼랜드를 인출할 수 있는 주소다. 즉, 명령어에 포함된 메모리 주소는 오퍼랜드의 간접 주소가 된다. 주소 a의 메모리 데이터를 MAR이라는 임시 주소 레지스터에 넣은 후, 다시 MAR 주소를 사용하여 메모리에서 오퍼랜드를 인출한다.

명령어 예시	연산 동작
ADD [a]	MAR ← [a], AC ← AC + [MAR]

레지스터 직접 주소방식

명령어에 포함된 주소는 주 메모리의 주소가 아닌 레지스터 파일 중 하나를 가리킨다. 레지스터 파일의 레지스터가 32개면 5비트로 특정 레지스터를 가리킬 수 있으므로 주 메모리의 주

소를 지칭하는 직접 주소방식(Direct addressing)보다 적은 비트가 사용된다. 또 CPU 내부에 레지스터 파일이 있으므로 오퍼랜드의 ALU에 도달하는 시간이 매우 빨라 연산을 신속하게 할 수 있다. 아래 사례는 레지스터 R1에 있는 데이터를 오퍼랜드로 사용한다.

명령어 예시	연산 동작
ADD R1	AC ← AC + R1

레지스터 간접 주소방식

명령어에 포함된 주소는 레지스터 파일의 특정 레지스터를 가리킨다. 그 레지스터에 들어 있는 값은 주 메모리의 주소로서, 오퍼랜드는 이 주소로부터 인출된다. 따라서 명령어에 포함된 레지스터 파일의 주소는 주 메모리의 오퍼랜드에 대한 간접 주소다. 명령어에 표시된 레지스터값을 MAR이라는 임시 주소 레지스터에 넣은 후, 다시 MAR 주소를 사용하여 메모리에서 오퍼랜드를 인출한다.

명령어 예시	연산 동작
ADD [R1]	MAR ← R1, AC ← AC + [MAR]

변위 주소방식

명령어에 두 개의 주소 필드를 가진다. 하나는 베이스(또는 인덱스) 레지스터를 가리키는 주소이고, 다른 하나는 변위(displacement)다. 배열 또는 메모리의 특정 위치를 베이스(또는 인덱스) 레지스터를 기준으로 하고, 변위 차에 해당되는 메모리 주소가 된다.

변위 주소방식은 배열 또는 스택에서 데이터를 참조할 때 많이 사용한다. 배열의 시작 주소를 베이스(또는 인덱스) 레지스터에 두고 변위값 거리에 있는 배열 원소를 참조하는 데 사용한다. 또 스택 포인터를 기준으로 함수 호출 복귀 주소, 매개 변수 전달에 사용되는 메모리 주소를 생성하는 데 사용한다. 명령어에 표시된 레지스터값과 변위값 disp를 MAR이라는 임시 주소 레지스터에 넣은 후, 다시 MAR 주소를 사용하여 메모리에서 오퍼랜드를 인출한다.

명령어 예시	연산 동작
ADD R1, disp	MAR ← R1 + disp, AC ← AC + [MAR]

상대 주소방식

변위 주소방식과 유사하지만, 기준되는 레지스터로 프로그램 카운터를 사용한다는 점이 다르다. C 언어 프로그램에서 if( ) 조건문, while( ) 반복문이 사용되면 명령어의 인출이 항상 순차적이지는 않다. 상대 주소방식은 조건이 만족되었을 때와 프로그램 카운터의 값을 특정 변위값 차이만큼의 프로그램 메모리 주소로 갱신해야 할 때 주로 사용된다. 명령어에 표시된 변위값 disp로 프로그램 카운터값을 갱신하는 사례는 아래와 같다.

명령어 예시	연산 동작
BNE disp	if Not_Zero then PC ← PC + disp

1.4.3 명령코드 동작

명령코드 필드에 의해 결정되는 명령어 동작은 연산, 데이터 이동, 프로그램 흐름제어와 함께 동작해야 한다.

연산

산술연산은 덧셈, 뺄셈, 곱셈, 나눗셈과 같은 산술연산 동작을 포함한다. 덧셈과 뺄셈은 ALU에서 회로로 쉽게 구현할 수 있지만, 곱셈과 나눗셈은 다소 복잡한 회로로 구현된다.

성능이 좋은 마이크로프로세서일수록 ALU에서 한 번에 수행되는 연산 비트가 크다. 덧셈과 뺄셈을 사용하면 소프트웨어적으로 곱셈과 나눗셈 연산을 수행할 수 있다. 그러나 여러 명령어를 실행해야 하므로 많은 시간이 소요된다. 곱셈 또는 나눗셈 연산을 ALU에서 하드웨어 회로로 구현하면 CPU는 복잡해지지만, 하나의 명령어만을 실행하기 때문에 연산을 빠르게 수행할 수 있다. 부동소수점 수에 대한 산술연산은 정수형 수의 연산에 비해 과정이 복잡하다. 부동소수점 연산을 하드웨어적으로 구현하여 CPU에 포함하면 빠른 연산을 수행할 수 있고, CPU 성능이 향상된다.

논리연산은 OR, AND, XOR, 좌로 비트천이, 우로 비트천이, 보수 변환 등의 논리연산 동작을 포함한다.

데이터 이동

메모리의 데이터가 산술 또는 논리연산이 되기 위해서는 CPU 안으로 값이 이동해야 한다. CPU 안에서는 레지스터와 레지스터 사이의 데이터가 이동할 수 있다. 이때 연산된 결과는

CPU에서 메모리로 이동되어야 한다. CPU에서는 논리값에 해당되는 디지털 전압으로 출력 신호를 출력할 수 있을 뿐만 아니라 메모리에서 다른 메모리로 정보를 이동할 수도 있다.

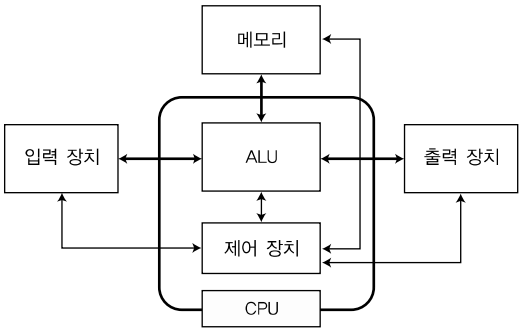
프로그램 흐름제어

프로그램 메모리에 차례대로 저장된 명령어는 일반적으로 프로그램 카운터(PC)가 차례로 증가하면서 인출되어 실행된다. 조건부 분기 명령 또는 반복 명령은 조건을 검사한 결과에 따라 실행되는 명령어의 위치가 순차적이지 않을 수 있다.

프로그램의 흐름을 바꾸는 분기 명령으로 프로그램 카운터(PC)값이 변경되면 명령어 인출 위치가 바뀌고 프로그램의 흐름이 달라진다. 상대 주소방식(Relative Addressing)은 이러한 프로그램 흐름제어 명령어에 많이 사용된다.

1.5 저장형 프로그램 컴퓨터

폰 노이만 아키텍처(von Neumann architecture)로 설계된 컴퓨터 모델을 '저장형 프로그램(Stored Program) 컴퓨터'라 한다. 저장형 프로그램 컴퓨터는 [그림 1-6]과 같이 구성되며, 메모리에 명령어와 데이터를 저장하고 읽기와 쓰기 동작을 반복하면서 원하는 연산을 수행한다. 굵은 선은 데이터 경로(Data path)를, 가는 신호선은 제어 신호를 나타낸다. CPU 내부에서 제어 신호는 명령어 인출-실행 사이클(Fetch-execution cycle) 과정에 필요한 데이터와 제어 신호가 된다. ALU와 제어 장치는 CPU(Centeral Processing Unit)의 기본 요소가 된다. CPU 외부에 메모리를 비롯한 입출력 장치를 연결하면 외부와 인터페이스하는 컴퓨터 시스템으로 확장된다.



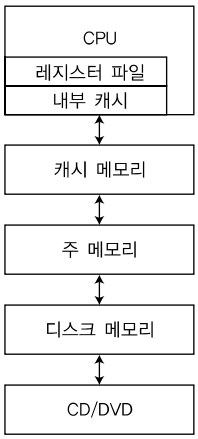
[그림 1-6] 저장형 프로그램 컴퓨터 개념도

1.5.1 저장형 프로그램 컴퓨터의 한계

반도체와 컴퓨터 기술이 발전하면서 메모리 용량이 증가되고, CPU 속도도 향상되고 있다. CPU 구조 기술이 발달할수록, 처리되는 데이터가 많아질수록 메모리와 CPU 사이에는 더 많은 데이터가 왕래한다. CPU는 명령어와 데이터를 메모리로부터 가져와야만 연산을 수행할 수 있다. 그러나 단일 CPU의 저장형 프로그램 컴퓨터는 정해진 연결선만을 통해 명령어와 데이터를 메모리로부터 전송할 수 있다. 따라서 CPU와 메모리 사이의 한정된 데이터 전송 속도에 의해 병목 현상(Bottle-neck)이 나타나게 되고, 이로 인해 CPU 안에서 연산 능력이 향상된다고 하더라도 연산 속도의 향상에는 한계가 드러난다. 이 한계를 연장하기 위해 메모리 계층을 이용하거나 병렬 처리 컴퓨터를 이용하기도 한다.

1.5.2 메모리 계층

저장형 프로그램 컴퓨터에서는 메모리의 병목 현상과 메모리 속도, 속도 대비 가격을 고려하여 메모리를 계층적으로 배치한다. [그림 1-7]과 같이 CPU 가까이에 빠른 데이터 인출과 저장을 할 수 있는 메모리를 배치하고, 멀리에는 데이터 인출 속도는 늦지만 많은 용량을 저장할 수 있는 메모리를 배치한다. 캐시(Cache)처럼 개별 데이터 인출과 저장이 빠른 메모리는 비트당 가격이 비싸지만, CD/DVD와 같은 보조 기억 장치는 블록 단위로 인출과 저장을 해야 하므로 비트당 가격이 상대적으로 싸다.



[그림 1-7] 메모리 계층(Memory Hierarchy)

CPU 연산에 사용되는 데이터는 범용 레지스터(General register)에 두고 명령어의 각 비트값 설정에 따라 신속하게 시스템 버스(System bus)를 오가면서 산술연산, 논리연산, 데이터 이동 등의 명령을 처리한다.

명령어의 집합인 프로그램과 데이터는 저장형 프로그램 컴퓨터 개념에 따라 메모리에 위치한다. 지역성 원리(Principle of locality)에 따라 곧 처리될 프로그램과 데이터 일부를 CPU 근처의 속도가 빠른 메모리에 놓고 처리한다.

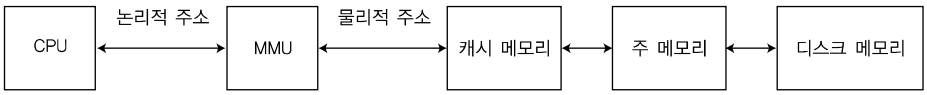
지역성은 시간과 공간 두 관점에서 성립된다. 시간적 지역성(Temporal locality)은 한번 참조된 데이터는 다시 참조될 가능성이 크다는 것을 의미하고, 공간적 지역성(Spatial locality)은 참조된 데이터에 인접된 위치가 참조될 가능성이 크다는 것을 의미한다. 지역성(locality)에 의해 참조되는 데이터 영역은 특정 부위에 확률적으로 집중되기 때문에 메모리 계층 구조를 두고, 자주 참조되는 데이터를 빠른 캐시에 놓고 사용한다.

1.5.3 가상 메모리

실행될 프로그램 또는 데이터는 디스크 메모리에 두었다가 실행이 시작되면 주 메모리(Main memory)로 이동시킨다. 이때 모든 프로그램과 데이터를 이동시키지 않고 사용되는 일부만을 이동하여 사용하다가 프로그램 또는 데이터가 더 요구되는 상황이 발생하면 그때 추가로 이동하여 사용한다. 가상 메모리(Virtual Memory)로 확장된 메모리 공간은 운영체제(Operation System)가 관리한다. 가상 메모리는 주 메모리의 크기와는 상관없이 명목적으로 사용자의 메모리가 확장된 것처럼 주 메모리와 디스크 메모리를 연동하여 논리 주소를 가진다.

논리 주소를 물리 주소로(하드웨어적으로) 빠르게 변환하기 위해서는 [그림 1-8]과 같이 메모리 관리 장치(MMU, Memory Management Unit)를 사용한다. CPU는 논리 주소로 프로그램 또는 데이터에 접근하지만, MMU에서는 논리 주소를 주 메모리에 적재된 물리 주소로 TLB(Translation Lookaside Block)라는 테이블을 이용하여 빠르게 변환시킨다.

주 메모리와 디스크 메모리 사이는 페이지 테이블(Page Table)이라는 것이 있는데, 이는 데이터의 일관성을 유지하는 역할을 한다. TLB에서 요구하는 데이터가 주 메모리에 적재되어 있지 않으면 페이지 펄트(Page Fault)가 발생하고, 운영체제의 도움으로 적재되지 않은 데이터는 페이지 단위로 디스크 메모리에서 주 메모리로 옮겨진다.



[그림 1-8] MMU를 이용한 메모리 계층

1.6 RISC와 CISC

컴퓨터 구조는 발전 단계에서 CPU 성능을 향상시키기 위해 다양한 기능을 포함해 왔다. 상황이 이렇다 보니 다양한 어드레싱 모드(Addressing Mode)가 제공되었고, 프로그래머는 마이크로프로세서의 특징을 잘 활용하여 CPU 성능을 크게 개선할 수 있게 되었다.

어떤 회사는 특정 기능을 개선하여 하드웨어에 포함하고, 구동을 위한 특수 명령어를 제공하기도 했다. 그러나 역설적이게도 프로그래머는 CPU 구조를 샅샅이 공부하여 프로그래밍하지 않고 알고리즘을 구체화할 수 있는 고급 언어로 손쉽게 작성하는 것을 선호한다. 즉, CPU 성능 향상을 위해 포함된 다양한 기능이 기대한 만큼 활용되지 못하는 결과를 갖는 일이 많아진 것이다.

과거에는 제공되는 특징을 최적화할 수 있는 컴파일러를 만들면 성능을 더 높일 수 있을 것으로 생각했다. 하지만 컴파일러 설계자 또한 고급 언어의 일반적인 경우를 최적화된 어셈블 코드로 생성하기가 쉽지 않다. 고급 언어를 이용한 일반적인 프로그래밍 기법이 아닌 특정한 형태로 작성된 경우에는 최적화에 가까운 코드를 생성할 수는 있지만, 범용으로 사용하기는 어렵다.

이 절에서는 RISC형 마이크로프로세서를 중심으로 설명하고, 이와 상반되는 구조의 CISC형 마이크로세서는 간단히 의미만 설명하기로 한다.

1.6.1 RISC

RISC(Reduced Instruction Set Computer)는 기존의 컴퓨터 구조를 CISC(Complex Instruction Set Computer)로 규정하고, 이와 상대되는 개념으로 발전했다. 이는 명령어 셋(Instruction Set)을 줄임으로써 CPU 하드웨어 구조를 CISC와 비교하여 간단히 만들 수 있다는 취지에서 시작되었다.

단순한 어드레싱 모드를 사용하고 명령어 길이를 짧게 하면, 프로그램 메모리에서 인출되는 명령어 크기를 줄일 수 있기 때문에 명령어 인출 시간이 줄어든다.

RISC형 마이크로프로세서의 특징은 다음과 같다.

파이프라인 기법(Pipelining)을 이용하여 한 사이클에 한 명령어 실행

명령어 인출(Instruction fetch), 명령어 판독(Decoding), 연산(Execution), 결과 저장(Writing back) 단계를 한 클록 사이클에 처리될 수 있는 구조로 설계하면 [그림 1-9]와 같이 파이프라인으로 처리할 수 있다. 실행되는 명령어의 단계별 결과 정보는 사각형으로 표시된 플립플롭에 저장되어 연속된 클록 사이클에 의해 다음 단계로 넘겨진다.

i+2번째 클록 사이클에서 명령어가 실행되고 있는 상황을 스냅샷(snapshot)으로 관찰해보면, 명령어 i+2는 명령어 인출 단계를, 명령어 i+1은 명령 판독 단계를, 명령어 i는 연산 단계를, 명령어 i-1은 결과 저장 단계를 수행하고 있다는 것을 알 수 있다.

프로그램의 수행 순서를 바꾸는 흐름제어 명령어가 아니고 각 단계마다 한 클록이 소요되면, 평균 한 사이클마다 한 개의 명령어가 된다. 따라서 16[MHz]를 사용하는 파이프라인에서 한 사이클에 한 명령어가 실행되는 마이크로프로세서는 최대 평균 16[MIPS]의 속도를 낼 수 있다.

여기서 잠깐!

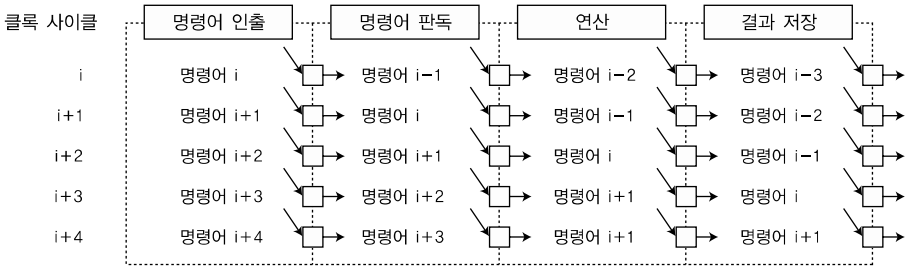
+ MIPS

가장 널리 알려진 마이크로프로세서의 성능 측정 단위 중 하나는 MIPS(Million Instructions per Second)다. 초당 실행되는 명령어 개수는 백만 개 단위로 표시한다. 그런데 같은 작업을 서로 다른 마이크로프로세서로 처리하기 위한 명령어 개수는 같지 않으므로, MIPS가 크다고 해서 성능이 우수하다고 말할 수는 없다.

+ FLOPS

과학 연산에 많이 사용되는 부동소수점 연산은 복잡하기 때문에 하드웨어를 사용하지 않고 정수형 연산만으로 처리할 경우, 명령어가 많이 필요하다. MIPS가 크더라도 생성되는 명령어 개수가 많아지면 프로그램 실행 시간은 여전히 길어지게 된다.

명령어 i가 흐름제어 명령어이면, 다음에 수행될 명령어는 연속으로 인출되어 이전 단계에 수행되고 있는 명령어 i+1이 아닐 수 있으므로 파이프라인에서 얻어진 결과는 무시된다. 이때는 처음부터 명령어를 인출해야 하므로 추가 클록이 필요하다.



[그림 1-9] 각 단계에서 클록 사이클에 따라 수행되는 명령어

고정된 짧은 길이의 명령어

[그림 1-9]에 나타낸 각 파이프라인 단계에서는 시간 지연이 가장 오래 걸리는 임계 경로를 기준으로 클록 사이클의 주기를 결정한다. 메모리 또는 캐시에서 명령어 인출 시간을 단축하기 위해서는 한 번의 클록 주기에 명령어를 인출해야 한다. CPU 시스템 버스 크기와 같게 명령어의 크기를 고정해야만 단일 클록에서 명령어 레지스터로 옮길 수 있다.

단순한 어드레싱 모드 사용

CISC형 마이크로프로세서의 어드레싱 모드에는 직접 주소방식, 간접 주소방식, 레지스터 간접 주소방식, 변위 주소방식과 같이 명령어에서 메모리를 직접 참조할 수 있는 필드가 있다. 이때는 명령어의 길이가 길어진다. 메모리에 저장된 오퍼랜드를 인출하는 명령어와 인출하지 않는 명령어가 같이 있기 때문에 경우에 따라 메모리를 참조할 수도 있고, 그렇지 않을 수도 있다. 단계별로 같은 시간을 보장하여 파이프라인화하기에는 적합하지 않다. 파이프라인 구조를 활용하는 데는 일반 명령어의 어드레싱 모드로 메모리에서 오퍼랜드를 가져오는 직접 주소방식, 간접 주소방식, 레지스터 간접 주소방식, 변위 주소방식을 사용하지 않는다. 즉, CPU 내부의 레지스터를 오퍼랜드로 이용하거나 명령어에 포함된 데이터를 오퍼랜드로 이용한다.

RISC형 마이크로프로세서에서는 메모리 참조를 위한 명령어를 따로 두어 명령어의 길이와 파이프라인의 각 단계에서 요구되는 시간 지연을 줄임으로써 클록 사이클의 주기를 줄인다. 메모리에서 데이터를 인출할 때는 load 명령어를, CPU 내부의 값을 저장할 때는 store 명령어를 사용한다.

load와 store 메모리 참조 명령에 단순한 어드레싱 모드를 사용하면, CPU 내부 하드웨어 복잡도를 줄이고 명령어의 길이를 동일하게 유지할 수 있다. 가장 많이 사용하는 어드레싱 모드에는 즉시 주소방식, 레지스터 간접 주소방식, 변위 주소방식이 있다.

간단한 연산 동작의 적은 명령어 셋

명령어에서 산술과 논리와 관련된 간단한 연산 동작을 사용하면, [그림 1-9]의 연산 단계에서 짧은 시간에 수행될 수 있다. 즉, 짧은 주기의 클록 사이클을 사용할 수 있기 때문에 파이프라인을 이용한 명령어 실행 시간을 줄일 수 있다. 구현되는 명령어 개수가 적으면 각 명령어의 기능을 만족하기 위한 하드웨어 복잡도를 줄일 수 있다.

데이터 종속 보장을 위한 nop 명령 삽입 기능

아래와 같은 연속된 명령을 생각해보자. 100번지 주소의 오퍼랜드를 r1에 옮기고 r1과 r0 값을 더해 r0에 넣는 연산이다.

```
load r1, 100
add r0, r1
```

[그림 1-9]의 파이프라인 구조에서 load 명령의 결과 저장 단계가 끝나야만 r1에 주소 100의 오퍼랜드값이 옮겨진다. 연속된 add 명령어의 연산 단계에서는 아직 직전 명령의 결과가 r1에 종속된 데이터로 갱신되지 않았기 때문에 연산을 계속할 수 없다. 컴파일러는 이러한 문제 해결을 위해 nop 명령을 삽입하여 r1에 값이 갱신될 때까지 시간을 지연할 수 있는 코드를 생성해야 한다.

```
load r1, 100
nop
add r0, r1
```

선인출(Pre-fetch)과 예측 실행

조건 검사에 따라 프로그램의 흐름이 달라질 수 있는 branch 명령은 파이프라인의 최종 단계까지 실행되어야만 다음 명령어의 위치를 확신할 수 있다. branch 명령의 분기 위치는 연속된 다음 명령어 또는 다른 위치가 될 수 있다.

다음에 실행될 명령어 위치가 늦게 결정되면 연속된 명령어가 결정될 것으로 예측하기 때문에 선인출하여 실행한다. 연속된 명령어로 분기 위치가 결정되면 수행된 파이프라인 단계만큼 시간을 절약한 효과가 나타난다. 만일 다른 곳에 분기 위치가 결정되면 현재까지 예측 실행된 파이프라인 실행 결과를 모두 무시하고 새 위치부터 명령어를 인출하기 시작한다.

RISC형 마이크로프로세서에 적합한 컴파일러

RISC형 마이크로프로세서는 짧은 명령어 길이, 파이프라인 구조, 간단한 어드레싱 모드, 데이터 종속 보장을 위한 nop 명령어 삽입 등의 특징을 고려한 컴파일러 설계가 필요하다. CISC형 마이크로프로세서보다 작은 명령어 셋을 이용하여 고급 언어를 어셈블리어로 번역한다.

1.6.2 CISC

CISC(Complex Instruction Set Computer)는 RISC(Reduced Instruction Set Computer)와 상대되는 단어로, RISC 개념이 만들어지면서 CISC 용어가 자연스럽게 만들어졌다. CISC는 여러 단계의 동작을 실행시킬 수 있는 구조를 가지고 있으며, 다양한 어드레싱 모드를 사용하여 하나의 명령어로 복잡한 동작을 실행할 수 있다는 특징이 있다. 하나의 명령으로 복잡한 명령을 수행할 수 있기 때문에 편리해 보이지만, 복잡한 하드웨어가 필요하다는 단점이 있다.

RISC와 비교하여 CISC형 마이크로프로세서의 특징을 간단히 설명하면 다음과 같다.

- 가변 길이의 명령어 형식을 가진다.
- 명령어의 길이가 길다.
- 다양한 어드레싱 모드를 사용한다.
- 복잡한 연산 동작을 수행할 수 있는 다양한 명령어로 구성된 명령어 셋을 가진다.

현대에는 IC 집적기술과 컴퓨터 구조기술을 바탕으로 RISC에서 사용한 파이프라인 구조 등의 개념을 CISC에서도 채택하여 성능을 향상시키는 데 이용하고 있다. 즉, 마이크로프로세서가 고성능화되면서 RISC와 CISC의 구별이 점점 모호해지고 있는 것이다.

Chapter_01 연습문제

1.1 마이크로프로세서, 마이크로컨트롤러, 마이크로컴퓨터의 차이는 무엇인가?

1.2 사람은 어떤 과정을 거쳐 마이크로프로세서가 인식할 수 있는 기계어로 작성한 실행프로그램을 생성할 수 있는가?

1.3 고급 언어와 저급 언어의 차이는 무엇인가?

1.4 컴파일러, 어셈블러, 링커, 로더를 설명하라.

1.5 여러 개의 원시프로그램에서 실행 파일이 생성되기까지 과정을 설명하라.

1.6 다음 특수 레지스터의 용도를 설명하라.

프로그램 카운터, 명령어 레지스터, 상태 레지스터

1.7 명령어의 실행 절차를 설명하라.

1.8 명령어 인출-실행 사이클이란 무엇인가?

1.9 명령어에 포함되는 비트 필드를 설명하라.

1.10 3주소, 2주소, 1주소, 0주소 마이크로프로세서의 특징을 각각 설명하라.

1.11 다음의 어드레싱 모드(Addressing mode)를 설명하라.

(a) 즉시 주소방식
(b) 직접 주소방식
(c) 간접 주소방식

(d) 레지스터 직접 주소방식
(e) 변위 주소방식
(f) 상대 주소방식

1.12 명령어 셋에 포함되는 명령어 종류를 나열하고 설명하라.

1.13 저장형 프로그램 컴퓨터의 개념을 설명하라.

1.14 저장형 프로그램 컴퓨터의 한계는 무엇 때문에 발생하는가?

1.15 메모리 계층은 왜 필요하며, 어떻게 동작하는지를 설명하라.

1.16 마이크로프로세서의 성능을 간단하게 언급할 수 있는 다음 단어를 설명하라.

```
MIPS, FLOPS
```

1.17 RISC형 마이크로프로세서와 CISC형 마이크로프로세서의 차이를 설명하라.

1.18 RISC형 마이크로프로세서에 사용되는 기술을 나열하고 설명하라.

1.19 명령어가 고정된 짧은 길이를 가지면 명령어가 처리되는 속도가 향상될 수 있다. 그 이유를 설명하라.

1.20 단순한 어드레싱 모드를 사용하면 하드웨어의 복잡도를 줄일 수 있다. 그 이유를 설명하라.

1.21 선인출(Pre-fetch)과 예측 실행이 갖는 장점을 설명하라.