특수절 : 나만의 컴퓨터 만들기

다음 몇 가지 문제에서 고급 언어 프로그래밍 세계에서 벗어난 일시적인 전환의 내용을 다룰 것이다. 컴퓨터를 "완전히 열고" 내부 구조를 살펴볼 것이다. 기계어 프로그래밍을 소개하고, 여러 기계어 프로그램을 작성할 것이다. 이 특별한 소중한 경험을 만들기 위해 기계어 프로그램을 실행할 수 있는 컴퓨터(시뮬레이션에 근거한 소프트웨어의 기술)를 만들 것이다.

7.18 (Machine Language Programming: 기계어 프로그래밍) Simpletron이라 부르는 컴퓨터를 만들어 보자. 이름에 나타나듯이 이것은 단순한 기계지만 우리가 될 수 있는 강력한 것이다. 이 Simpletron은 그것을 이해하는 유일한 언어인 Simpletron Machine Language 또는 SML으로 프로그램을 실행한다.

Simpletron이 계산에서 정보를 사용하거나 여러 방법에서 예시화하기 전에 정보가 입력된 "특별한 레지스터"인 누산기(accumulator)를 포함한다. Simpletron에 모든 정보는 word(단어)로 다루어진다. 단위 **+3364, -1293, +0007, -0001**등과 같은 부호가 있는 4개의 십진수를 나타낸다. Simpletron는 100-단어 메모리로 되어있으며, 이들 단어는 위치 수 **00, 01, ..., 99**에 의해 참조된다.

SML 프로그램을 실행하기 전에 메모리에 프로그램을 로드(load: 불러오기)하거나 위치시켜야 한다. 모든 SML 프로그램의 첫 번째 명령(또는 문장)은 항상 위치 00에 있게 된다.

SML에 쓰여진 각 명령은 Simpletron에 메모리의 하나의 단어를 차지한다.(명령은 부호가 있는 4개의 십진수이다) SML 명령의 부호는 항상 양(+)이지만, 데이터의 부호는 양(+) 또는 음(-)이라고 가정한다. Simpletron의 메모리에 있는 각각의 위치는 명령, 프로그램에 의해 사용된 데이터 값, 메모리에 사용되지 않은 (정의되지 않은) 영역을 나타낸다. 각 SML 명령의 첫 번째 두 숫자는 연산이 수행되는 것을 나타내는 연산 코드(operation code) 이다. SML 연산 코드는 그림 7.31에 요약되어 있다.

연산 코드	의미
입/출력 연산:	
#define Read 10	터미널의 단어를 메모리에 특별한 위치로 읽어라.
#define WRITE 11	메모리에 특별한 위치의 단어를 터미널에 출력하라.
로드/저장 연산:	
#define ADD 30	메모리에 특별한 위치의 단어를 누산기로 로드하라.
#define STORE 21	누산기의 단어를 메모리의 특별한 위치에 저장하라.
산술 연산:	
#define ADD 30	메모리에 특별한 위치의 단어를 누산기(누산기에 결과를 둔다)에 단어에 더하라.
#define SUBSTRACT 31	메모리에 특별한 위치의 단어를 누산기(누산기에서 결과를 둔다)에 단어를 빼어라.
#define DIVIDE 32	메모리에 특별한 위치의 단어를 누산기(누산기에서 결과를 둔다)에 단어를 나누어라.
#define MULTIPLY 33	메모리에 특별한 위치의 단어를 누산기(누산기에서 결과를 둔다)에 단어에 곱하라.
제어 연산 전달:	
#define BRANCH 40	메모리에 특별한 위치에서 브랜치(가지로 갈라짐)하라.
#define BRANCHNEG 41	만약 누산기가 음이라면 메모리에 특별한 위치에서 브랜치하라.
#define BRANCHZERO 42	만약 누산기가 0이라면 메모리에 특별한 위치에서 브랜치하라.
#define HALT 43	정지(Halt), 즉 프로그램은 일을 끝낸다.

그림 7.31 Simpletron Machine Language(SML) 연산 코드

SML 명령의 마지막 두 숫자는 연산자가 적용하기 위해 단어를 포함하는 메모리 위치 주소인 연산자(operand)이다. 그러면 이제 여러 가지 단순 SML 프로그램을 생각해 보자.

예 1 위치	수	지시
00	**+1007**	(A를 읽어라)
01	**+1008**	(B를 읽어라)
02	**+2007**	(A를 로드하라)
03	**+3008**	(B를 더하라)
04	**+2109**	(C를 저장하라)
05	**+1109**	(C를 써라)
06	**+4300**	(정지)
07	**+0000**	(변수 A)
08	**+0000**	(변수 B)
09	**+0000**	(결과 C)

이 SML 프로그램은 키보드로부터 두 수를 읽고 합을 계산하고 출력한다.

명령 **+1007**은 키보드로부터 첫 번째 수를 읽고(0으로 초기화되는) 위치 **07**에 그것을 놓는다. **+1008**은 다음 수를 위치 **08**에서 읽는다. 로드(load) 명령인 **+2007**은 첫 번째 수를 누산기에 놓고 더하기(add) 명령인 **+3008**은 두 번째 수를 누산기에 있는 수에 더한다. 모든 SML 산술 명령은 누산기에 그 결과를 남겨놓는다. 저장(store) 명령인 **+2109**는 결과를 쓰기(write) 명령으로부터 메모리 위치 **09**에 놓는다. **+1109**는 수를 가지고 그것(부호화있는 4개 숫자 십진수)을 출력한다. 정지(halt) 명령인 **+4300**은 실행을 종결한다.

예 2 위치	수	지시
00	**+1009**	(A를 읽어라)
01	**+1010**	(B를 읽어라)
02	**+2009**	(A를 로드하라)
03	**+3110**	(A를 빼라)
04	**+4107**	(음의 07을 브랜치하라)
05	**+1109**	(A를 써라)
06	**+4300**	(정지)
07	**+1110**	(B를 써라)
08	**+4300**	(정지)
09	**+0000**	(변수 A)
10	**+0000**	(변수 B)

SML 프로그램은 키보드로부터 두 수 읽고 더 큰 값을 결정하고 출력한다. C의 if문장과 같은 제어 전달로서 명령 **+4107**을 사용하는 것에 주의하라. 그러면 다음 각각의 일을 수행하는 SML 프로그램을 작성하라.

a) 10개의 양의 수를 읽기 위해 감시 제어 반복을 사용하고 합을 계산하고 출력하라.
b) 7개 수, 양수, 음수를 읽기 위해 카운터 제어 반복을 사용하고 평균을 계산하고 출력하라.
c) 일련의 수를 읽고 가장 큰 수를 결정하고 출력하라. 첫 번째 수는 얼마나 많은 수가 처리되었는

지 나타내는 것을 읽는다.

7.19 (computer Simulator: 컴퓨터 시뮬레이터) 우선 이것이 부당한 것처럼 보일지 모르지만 이 문제의 독자는 자신의 컴퓨터를 만들게 될 것이다. 그렇지만 아직 독자는 여러 부품들을 납땜하지 못할 것이다. 오히려 독자는 Simpletron의 소프트웨어 모델(software model)을 만들기 위해 소프트웨어에 기초한 시뮬레이션(software-based simulation)의 강력한 기술을 사용할 것이다. 그리고 실망하지 않을 것이다. 독자의 Simpletron 시뮬레이터는 여러분이 사용하는 컴퓨터를 Simpletron으로 바꿀 것이고 실제 연습문제 7.18에서 썼던 SML 프로그램을 실행, 테스트, 디버그 할 수 있을 것이다.

Simpletron 시뮬레이터를 실행할 때 다음과 같은 출력으로 시작해야 한다.

***** Welcome to Simpletron! *****
***** Please enter your program one istruction *****
***** (or data word) at a time. I will type the *****
***** location number and a question mark (?) *****
***** You then type the word for that location. *****
***** Type the sectinel -99999 to stop entering *****
***** your program ******

100개의 요소를 가진 1차원 배열 **memory**로 Simpletron의 메모리를 시험하라. 시뮬레이터가 실행하고 있다고 가정하고, 연습문제 7.18의 예 2 프로그램에 나타난 것처럼 다음의 다이얼로그(dialog)를 검사하자:

00 ? +1009
01 ? +1010
02 ? +2009
03 ? +3110
04 ? +4107
05 ? +1109
06 ? +4300
07 ? +1110
08 ? +4300
09 ? +0000
10 ? +0000
11 ? -99999
***** Program loading completed *****
***** Program execution begins *****

SML 프로그램은 바로 배열 **memory**에 놓여진다(또는 로드된다). Simpetron은 독자의 Simpletron 프로그램을 실행한다. 실행은 위치 **00**에서 명령으로 시작하며 제어의 전달에 의한 프로그램의 다른 부분에 직접적이 아니라면 C처럼 순차적으로 계속된다.

누산기 레지스터를 표현하기 위해 변수 **accumulator**를 사용하라. 수행되고 있는 명령을 포함하는 메모리에 위치를 탐지하기 위해 변수 **instructionCounter**를 사용하라. 수행되고 있는 현재의 연산을 나타내기 위해 왼쪽 명령의 두 개 숫자 단어인 변수 **operationCode**를 사용하라. 현재 명령이 연산되는 메모리 위치를 나타내기 위해서 변수 **operand**를 사용하라. 그러므로 **operand**는 수행하고 있는 현재의 가장 오른쪽 명령의 숫자이다. 명령을 직접적으로 메모리에서부터 실행하지 말아라. 오히려 메모리를 **instructionRegister**이 호출되는 변수로 수행되기 위해 다음 명령을 전달하라. 왼쪽 두 번째 수수자를 "꺼내어(pick off)"서 변수 **operationCode**에 놓고 오른쪽 두 번재 숫자를 "꺼내어"서 **operand**에 놓는다.
Simpletron이 실행할 때 특별한 레지스터가 다음과 같이 초기화된다.

accumulator	**+0000**
instructionCounter	**00**
instructionRegister	**+0000**
operationCode	**00**
operand	**00**

메모리 위치 **00**에 **+1009**의 첫 번째 SML 명령의 실행을 "읽는다(walk through)". 이것은 명령 실행 싸이클을 호출한다.
instructionCounter는 수행되기 위해 다음 명령의 위치를 알려준다. C 문장을 사용함으로써 **memory**에서 그 위치의 내용을 꺼낸다(fetch).

```
instructionRegister = memory [ instructionCounter ];
```

연산코드와 피연산자는 그 문장을 통해 명령 레스터를 추출한다.

```
operationCode = instructionRegister / 100;
operand = instructionRegister % 100;
```

Simpletron은 연산 코드가 실제 읽기(read) (Write 대 load 등)를 결정해야 한다. **switch**는 12개의 SML 연산을 통해 구분한다.
switch구조에서 다양한 SML 명령의 행동은 다음과 같이(우리는 독자에게 다른 것을 제시한다) 시뮬레이트한다(simulate).

읽기 : **scanf("%d" , &memory[operand]);**
로드 : **accumulator = memory[operand];**
더하기 : **accumulator += memory[operand];**
다양한 브랜치 명령: 다음에 이것들을 간단히 살펴볼 것이다.
정지 : 이 명령은 다음과 같은 메시지를 출력한다.

```
*** Simpletron execution terminated ***
```

그리고 메모리의 완전한 내용뿐만 아니라 각 레지스터의 이름과 내용을 출력한다. 이러한 출력은 종종 컴퓨터 덤프(computer dump: 컴퓨터 덤프는 오래된 컴퓨터에는 없다.)라 부른다. 덤프 함수를 프로그램하기 위해 간단한 덤프 형식은 그림 7.32에 나타나 있다. Simpletron 프로그램 실행 후의 덤프는 종결된 실행의 순간에 명령의 실제 값과 데이터 값들을 보여준다.

```
REGISTERS :
accumulator                +0000
instructionCounter            00
instructionRegister        +0000
operationCode                 00
operand                       00

MEMORY :
          0     1     2     3     4     5     6     7     8     9
  0  +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
 10  +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
 20  +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
 30  +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
 40  +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
 50  +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
 60  +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
 70  +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
 80  +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
 90  +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
```

그림 7.32 샘플 덤프

프로그램의 첫 번째 명령의 실행으로 위치 **00**에 **+1009**을 수행하여 보자. 우리가 나타낸 것처럼 **switch** 문장은 이것을 C 문장을 수행하여 시뮬레이트한다.

```
scanf( "%d", &memory[ operand ]);
```

질문 표시(?)는 scanf가 입력을 위해 사용자의 프롬프트에서 실행되기 전에 화면에 나타나야 한다. Simpletron은 값의 형식을 위해 사용자를 기다리고 나서 리턴키(Retrun key)를 누른다. 그런 다음 값은 위치 **09**로 읽혀진다.

이점에서 첫 번째 명령의 시뮬레이션이 완성된다. 모든 나머지는 다음 명령을 실행하기 위해 Simpletron를 준비한다. 막 수행된 명령이 제어의 전달이 아니기 때문에, 다음과 같은 드문 명령 카운터 레지스터의 증가가 필요하다.

```
++instructionCounter;
```

이것은 첫 번째 명령의 시뮬레이트된 실행을 완성한다. 전체 과정(즉, 명령 실행 싸이클)은 실행된

다음 명령을 꺼내와서(fetch) 새롭게 시작한다.

브랜치 명령 즉 제어의 전환이 어떻게 시뮬레이션 되는지 생각해보자.

우리가 해야 할 모든 일은 적절하게 명령 counter에 값을 조정하는 것이다. 그러므로 비조건인 브랜치 명령(**40**)은 **switch** 내에서 시뮬레이트된다.

```
instructionCounter = operand ;
```

조건적 "누산기가 0이라면 브랜치하라." 명령은 다음과 같이 시뮬레이트 된다.

```
if ( accumulator == 0)
    instructionCounter = operand;
```

이 시점에서 독자는 Simpletron 시뮬레이터를 구현해야 하고 연습문제 7.18에서 작성한 SML 프로그램의 각각을 실행해야 한다. 추가적인 특징으로 SML을 꾸밀 것이고 시뮬레이터에 이것들을 제공한 것이다.

시뮬레이터는 오류의 다양한 형식을 체크해야 한다. 예를 들어, 프로그램 로딩 동안 Simpletron의 **memory**에 각 사용자 형식의 수는 **-9999**에서 **+9999**범위가 되어야 한다. 시뮬레이터는 입력된 각 수가 이 범위로 테스트 하기 위해 **while** 루프를 사용해야 한다. 만약 그렇지 않으면 사용자가 정확한 수를 입력할 때까지 수를 다시 입력하기 위해 사용자 프롬프트를 계속 유지해야 한다.

실행하는 동안 시뮬레이터는 0으로 나누려고 시도하고 유효하지 않은 연산 코드를 실행하려고 시도하고, 누산기가 오버플로우(즉 **+9999**보다 크거나 **-9999**보다 작은 값의 결과의 산술연산)가 되는 것 같은 다양한 심각한 오류를 체크해야 한다. 이러한 심각한 오류를 치명적인 오류(fatal error)라고 부른다. 치명적인 오류가 발견되면, 시뮬레이터는 다음과 같은 오류 메시지를 출력해야 한다.

```
*** Attempt to divide by zero ***
*** Simpletron execution abnomally terminated ***
```

그리고 이전에 논의된 형식에서 완전한 컴퓨터 덤프를 출력해야 한다. 이것은 프로그램에서 사용자가 가 오류의 위치를 검색하게 도와줄 것이다.