Machine-level Representation of Programs

Jo, Heeseung

Program?

짬뽕라면

준비시간 :10분, 조리시간 :10분

재료

라면 1개, 스프 1봉지, 오징어 1/4마리, 호박 1/4개, 양파 1/2개, 양배추 1장, 당근 1/4개, 물 3컵 (600cc)

Data

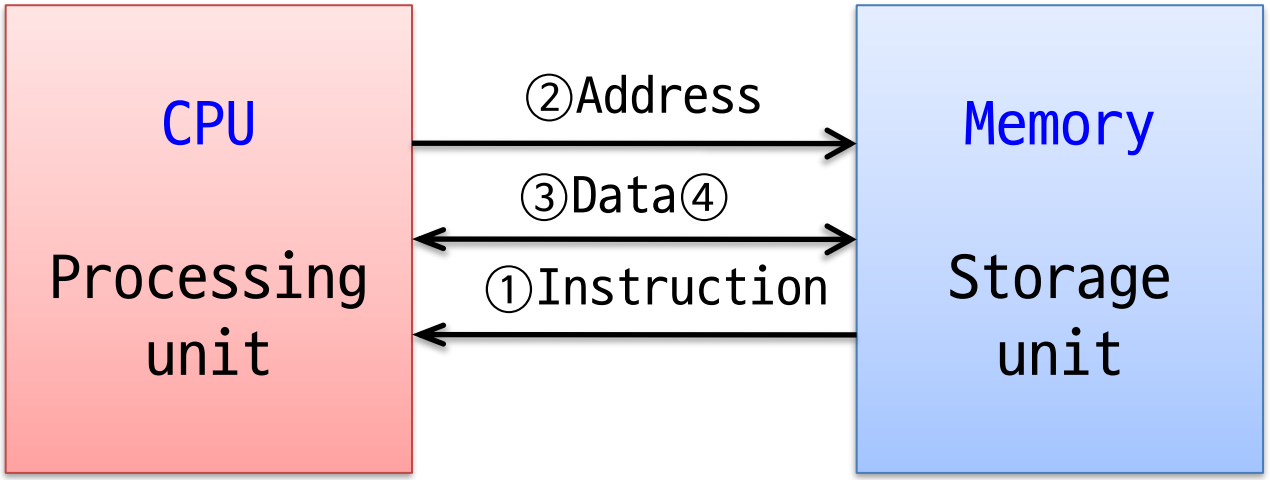
만드는 법

1. 오징어는 껍질을 벗기고 깨끗하게 씻어 칼집으로 모양을 낸다.
2. 호박, 양파, 양배추는 모두 채썬다.
3. 냄비에 물 3컵을 붓고 끓인다.
4. 물이 끓으면 스프를 넣고 오징어와 야채를 넣어 충분히 맛이 우러나도록 5분 정도 끓여준다.
5. 끓으면 면을 넣어 익힌다.

Instructions

Introduction (1)

Computer system



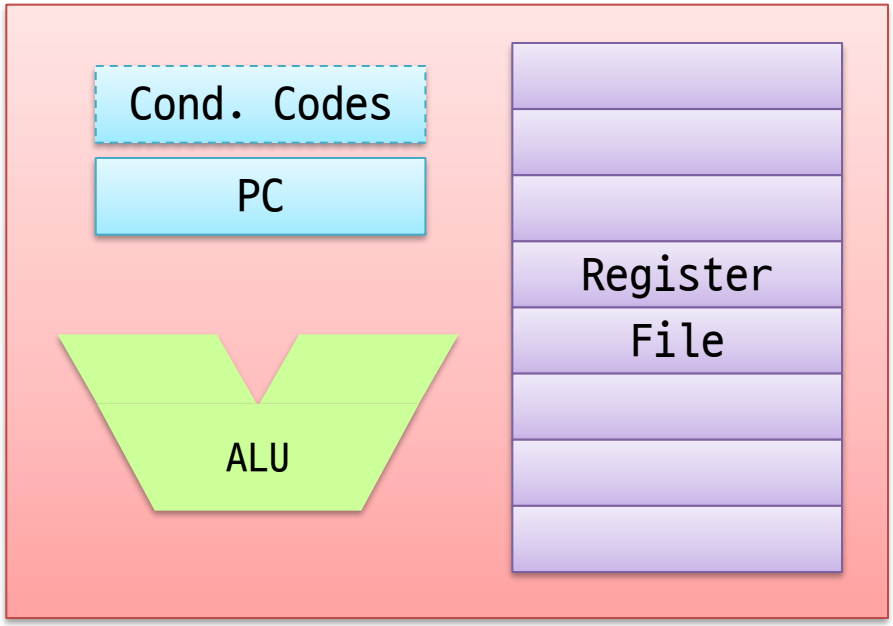
Data movement
Arithmetic & logical ops
Control transfer

Byte addressable array
Program code + data
OS code + data, ...

Introduction (2)

CPU (Central Processing Unit)

- PC (Program Counter)
  - Address of next instruction
  - Called "EIP" (IA-32) or "RIP" (x86-64)
- Register File
  - Heavily used program data
- Condition codes
  - Store status information about most recent arithmetic operation
  - Used for conditional branching

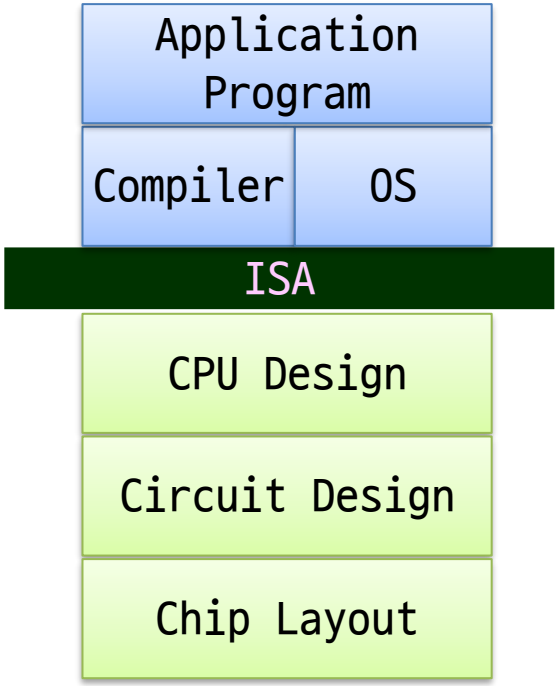


Introduction (3)

Instruction Set Architecture (ISA)

- An important part of "Architecture"
- Above: how to program machine
  - Processors execute instructions in sequence
- Below: what needs to be built
  - Use variety of tricks to make it run fast

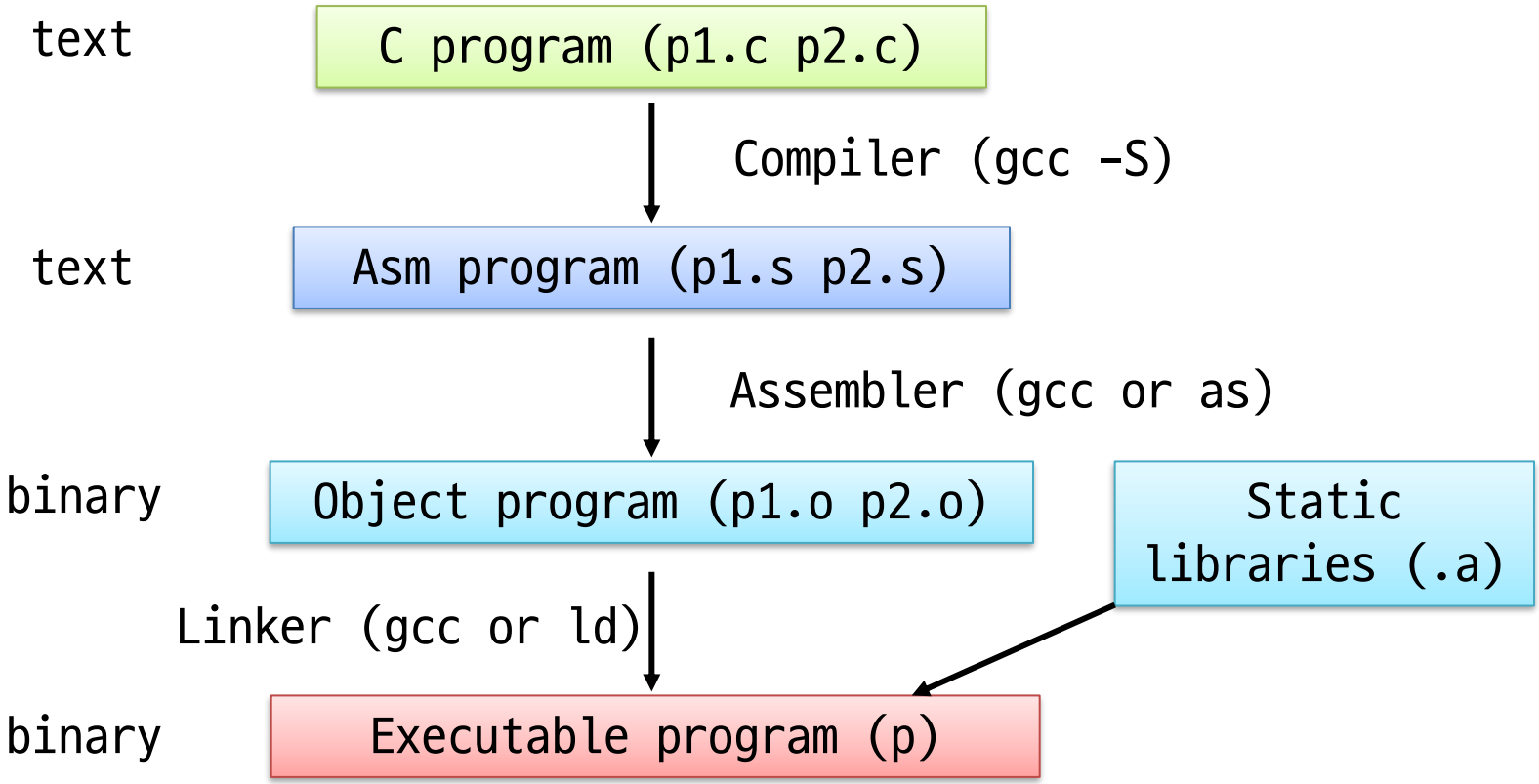
- Instruction set
- Processor registers
- Memory addressing modes
- Data types and representations
- ...



Turning C into Object Code

```
gcc -O p1.c p2.c -o p
```

- Use optimizations (-O)
- Put resulting binary in file p



Compiling into Assembly

```
gcc -O -S sum.c
```

sum.c

```
int sum(int x, int y)
{
  int t = x + y;
  return t;
}
```

sum.s

```
_sum:
    pushl %ebp
    movl  %esp,%ebp
    movl  12(%ebp),%eax
    addl  8(%ebp),%eax
    movl  %ebp,%esp
    popl  %ebp
    ret
```

Some compilers use single instruction "leave"

```
0x401040 <sum>:
        0x55 0x89 0xe5 0x8b 0x45 0x0c 0x03 0x45
        0x08 0x89 0xec 0x5d 0xc3
```

Object Code

Assembler

- Translates .s into .o
- Binary encoding of each instruction
- Nearly-complete image of executable code
- Missing linkages between code in different files

Linker

- Resolves references between files
- Combines with static run-time libraries
  - e.g., code for malloc(), printf(), etc.
- Some libraries are dynamically linked
  - Linking occurs when program begins execution

```
0x401040 <sum>:
0x55
0x89
0xe5
0x8b
0x45
0x0c
0x03
0x45
0x08
0x89
0xec
0x5d
0xc3
```

- Total of 13 bytes
- Each instruction 1, 2, or 3 bytes
- Starts at address 0x401040

Machine Code Example

C code

- Add two signed integers

```
int t = x + y;
```

Assembly

```
addl 8(%ebp),%eax
```

- Add two 4-byte integers
  - "Long" words in GCC parlance
  - Same instruction whether signed or unsigned
- Operands
  - x: Register %eax
  - y: Memory M[%ebp+8]
  - t: Register %eax

Object code

- 3-byte instruction
- Stored at address 0x401046

```
0x401046: 03 45 08
```

Disassembling (1)

Disassembler: objdump -d sum

- Useful tool for examining object code
- Analyzes bit pattern of series of instructions
- Produces approximate rendition of assembly code
- Can be run on either a.out (complete executable) or .o (object code) file

```
00401040 <_sum>:
   0:        55                   push   %ebp
   1:        89 e5                mov    %esp,%ebp
   3:        8b 45 0c             mov    0xc(%ebp),%eax
   6:        03 45 08             add    0x8(%ebp),%eax
   9:        89 ec                mov    %ebp,%esp
   b:        5d                   pop    %ebp
   c:        c3                   ret
```

Disassembling (2)

Using gdb (GNU debugger)

```
$ gdb sum
(gdb) disassemble sum
Dump of assembler code for function sum:
0x401040 <sum>:          push %ebp
0x401041 <sum+1>:        mov  %esp,%ebp
0x401043 <sum+3>:        mov  0xc(%ebp),%eax
0x401046 <sum+6>:        add  0x8(%ebp),%eax
0x401049 <sum+9>:        mov  %ebp,%esp
0x40104b <sum+11>:       pop  %ebp
0x40104c <sum+12>:       ret
```

Disassemble procedure, sum

```
$ gdb sum
(gdb) x/13b sum
0x401040:
  0x55  0x89
  0xe5  0x8b
  0x45  0x0c
  0x03  0x45
  0x08  0x89
  0xec  0x5d
  0xc3
```

Examine 13 bytes starting at sum

Computer Systems Review

기계어란?

CPU는 비트패턴으로 인코딩된 명령들을 해석할 수 있도록 설계되어 있음

기계어(machine language) : 인코딩체계 + 명령집합

- 기계가 인식할 수 있는 모든 기계 명령의 집합

기계 명령(machine instruction)

- 기계어에서 표현되는 기계 수준 명령
- 아주 간단한 명령들이지만, 다 모이면 복잡한 일을 수행
- 모든 프로그램은 결국 기계 명령들로 변환되어야 함

기계어 철학의 양대 축

컴퓨터 설계 시 기계 명령을 얼마나 간단히 만들 것인가?

단순하고, 빠르고, 효율적인 명령을 갖추도록 할까?

RISC

vs.

많은 수의 복잡하지만 강력한 명령을 갖추도록 할까?

CISC

명령의 종류

RISC 구조이든 CISC 구조이든 기계 명령들은 3그룹으로 분류

데이터 전송(Data transfer)

- 한 장소에서 다른 장소로 데이터를 복사

연산(Arithmetic/Logic)

- 기존의 비트 패턴을 사용하여 새로운 비트 패턴을 계산

제어(Control)

- 프로그램 실행을 지시

종류 1 : 데이터 전송

데이터 전송 그룹

- 데이터를 컴퓨터 내의 어느 한 장소에서 다른 장소로 옮길 것을 요청하는 명령들로 구성

실제로는 전송(transfer)가 아니라, 복사(copy)

- LOAD : MEM → CPU
- STORE/SAVE : CPU → MEM
- MOVE : MEM → MEM

I/O 명령

- CPU나 주기억장치가 아닌 프린터, 키보드, 디스플레이 화면, 디스크 장치와의 통신을 위한 명령들
- 별도로 취급

종류 2 : 연산

연산그룹

- 제어장치가 연산장치에 어떤 작업을 하도록 요청하는 명령들로 구성

종류

- AND/OR/XOR
  - 부울 연산
- SHIFT/ROTATE
  - 레지스터의 내용을 그 안에서 오른쪽이나 왼쪽으로 이동하는 연산들
- ADD/SUB/MULT/DIV
  - 사칙 연산

종류 3 : 제어

제어그룹

- 데이터를 조작하는 대신 프로그램의 실행을 조종하는 명령들로 구성

JUMP/BRANCH : 대표적

- 조건부 점프(conditional jump) : C의 if, while, for 문
- 무조건 점프(unconditional jump) : C의 goto 문

덧셈 명령 예

메모리에 저장된 값들에 대한 덧셈 C = A + B

1. A를 메모리에서 가져와 레지스터에 저장

   ```
   LOAD R1, MEM_A
   ```

2. B를 메모리에서 가져와 또 다른 레지스터에 저장

   ```
   LOAD R2, MEM_B
   ```

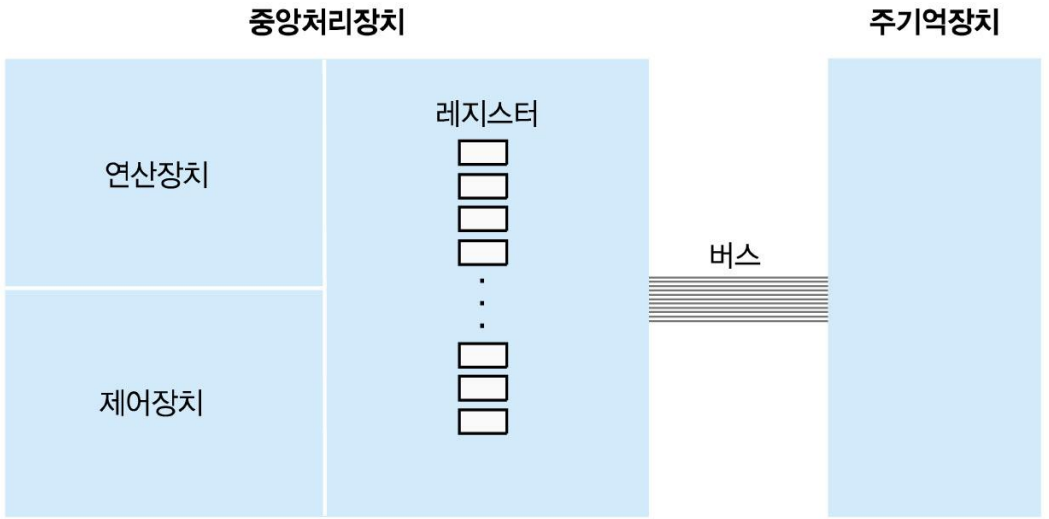
3. R1, R2 레지스터들 더해 R3 레지스터에 저장하도록 덧셈 회로를 작동

   ```
   ADD R3 R1 R2   // R3 = R1 + R2
   ```

4. 결과를 메모리 C 위치에 저장

   ```
   STORE R3, MEM_C
   ```

5. 멈춤

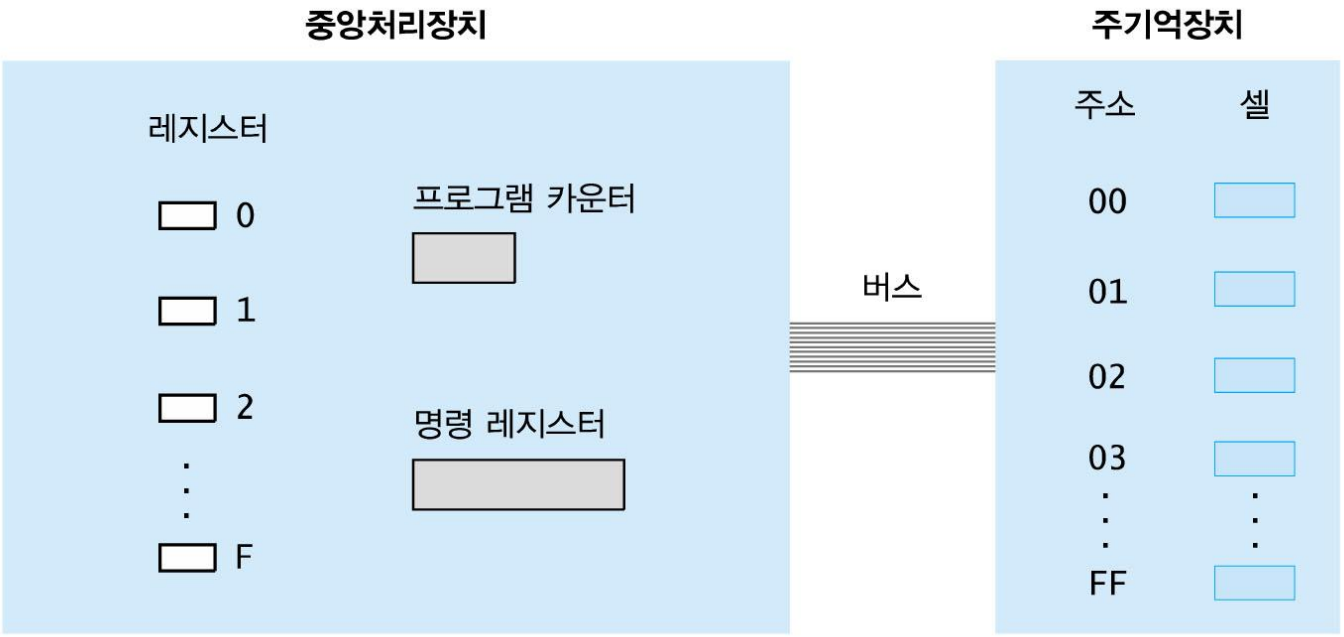


가상 기계어

컴퓨터에서 명령들은 어떻게 인코딩될까?

가상 컴퓨터를 만들어보자!

- 주기억장치 : 256개의 주기억 장치 셀(cell)
- 범용 레지스터 : 16개



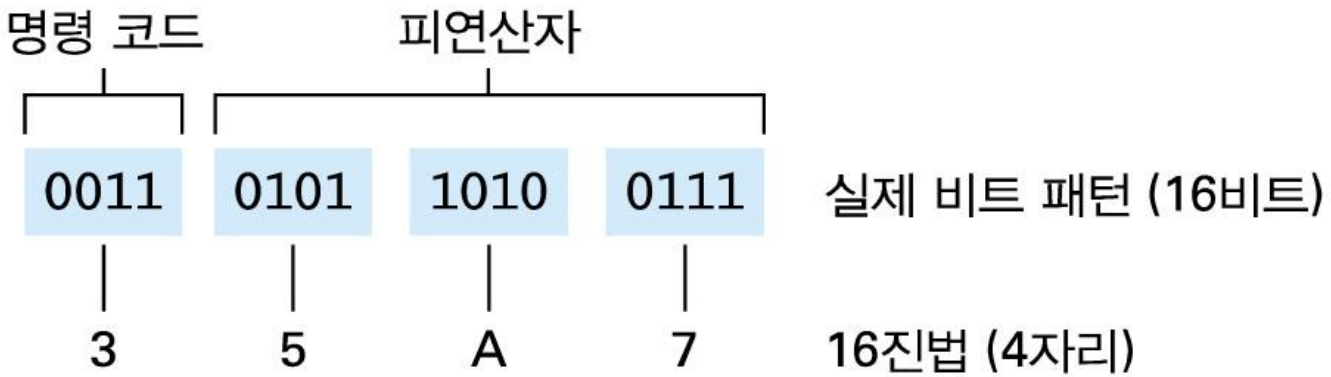
가상 기계어

기계 명령 구조

- 명령 코드(op-code, operation code) : 어느 명령인지 지정
- 피연산자(operand) : 명령어 대상 (명령코드가 사용할 정보)

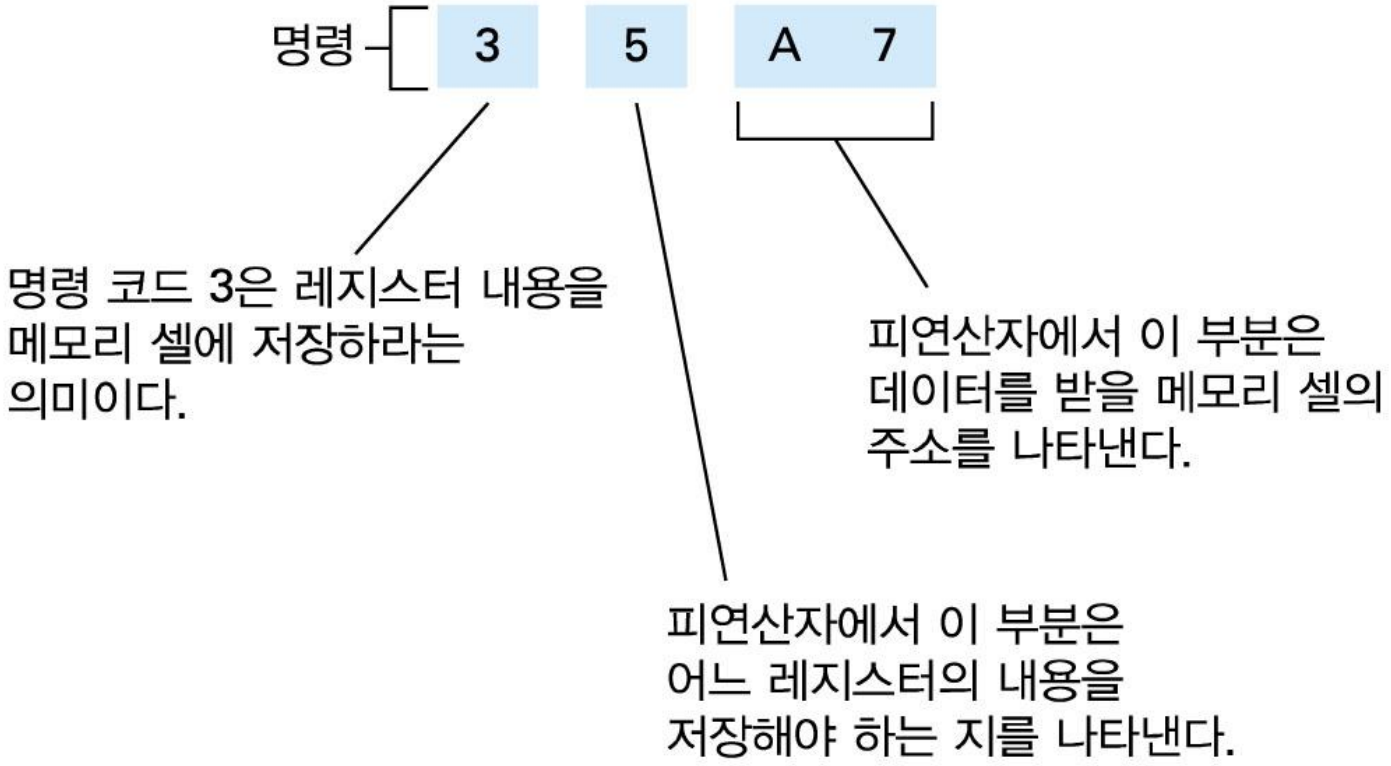
가상 컴퓨터에서 사용되는 기계어 : 2 바이트 = 16 비트

- 명령코드(op-code) : 최초의 4 비트 (16개의 명령 가능)
- 피연산자(operand) : 4 (레지스터 번호) + 8 (메모리주소/값)
  - 4 bit = 16개의 레지스터 사용가능
  - 8 bit = 256B의 메모리 사용가능



명령 35A7의 해석

"5번 레지스터의 내용을 메모리 주소 A7에 저장하라"



가상 명령의 구성

LOAD R, MEM 1RXY

- register R ← address XY에 든 값

LOAD R, VAL 2RXY

- register R ← XY 가 표현하는 1 byte 값

STORE R, MEM 3RXY

- address XY ← register R에 든 값

MOVE R, S 40RS

- register S ← register R에 든 값

가상 명령의 구성

ADD R, S, T 5RST

- 2's complement addition
- register R ← register S + register T

ADDF R, S, T 6RST

- floating point addition
- register R ← register S + register T

가상 명령의 구성

OR R, S, T　　7RST　(bit-wise OR)

- register R ← register S OR register T

AND R, S, T　　8RST　(bit-wise AND)

- register R ← register S AND register T

XOR R, S, T　　9RST　(bit-wise XOR)

- register R ← register S XOR register T

bit-wise operation: bit 단위 (C의 |, &, ^)

logical operation: byte 단위 (C의 ||, &&)

가상 명령의 구성

ROT R, X AR0X

- register R의 값을 X-bit 만큼 rotate right

JUMP R, ADDR BRXY

- if (register R = register #0), jump to address XY

unconditional jump B0XY

HALT C000

- 무조건 실행 중단

기계 명령 사용 예

메모리에 저장된 값들에 대한 덧셈 C = A + B 에 대한 명령

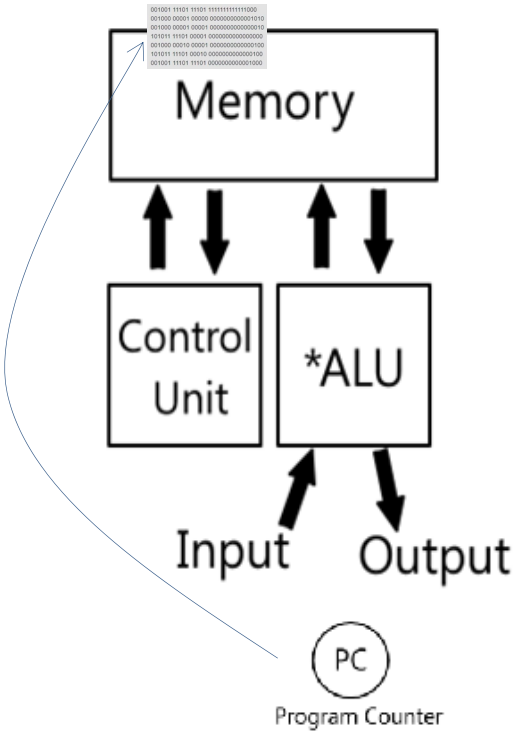
인코딩된 명령	해설
156C	주소가 6C인 메모리 셀에 들어있는 비트 패턴을 5번 레지스터에 LOAD
166D	주소가 6D인 메모리 셀에 들어있는 비트 패턴을 6번 레지스터에 LOAD
5056	5번 레지스터와 6번 레지스터의 내용에 대해 2의 보수 덧셈을 수행하고, 그 결과를 0번 레지스터에 넣음
306E	0번 레지스터의 내용을 주소가 6E인 메모리 셀에 STORE
C000	멈춘다

```
LOAD R, MEM                              1RXY
      register R ← address XY에 든 값

LOAD R, VAL                              2RXY
      register R ← XY 가 표현하는 1 byte 값

STORE R, MEM                             3RXY
      address XY ← register R에 든 값

ADD R, S, T                              5RST
      2's complement addition
      register R ← register S + register T

HALT                                     C000
      무조건 실행 중단
```

프로그램의 실행

CPU가 프로그램을 실행시키려면?

- 프로그램들이 메모리에 위치
  - 프로그램이 실행되려면 먼저 프로그램이 실행 가능한 상태로 준비되어 있어야 함
- 컴퓨터는 필요한 대로 명령들을 메모리에서 CPU로 복사
- 일단 CPU로 옮겨진 명령은 해석되고 실행됨



*ALU(Arithmetic Logic Unit)

```
001001 11101 11101 1111111111111000
001000 00001 00000 0000000000001010
001000 00001 00001 0000000000000010
101011 11101 00001 0000000000000000
001000 00010 00001 0000000000000100
101011 11101 00010 0000000000000100
001001 11101 11101 0000000000001000
```

용도 지정 레지스터

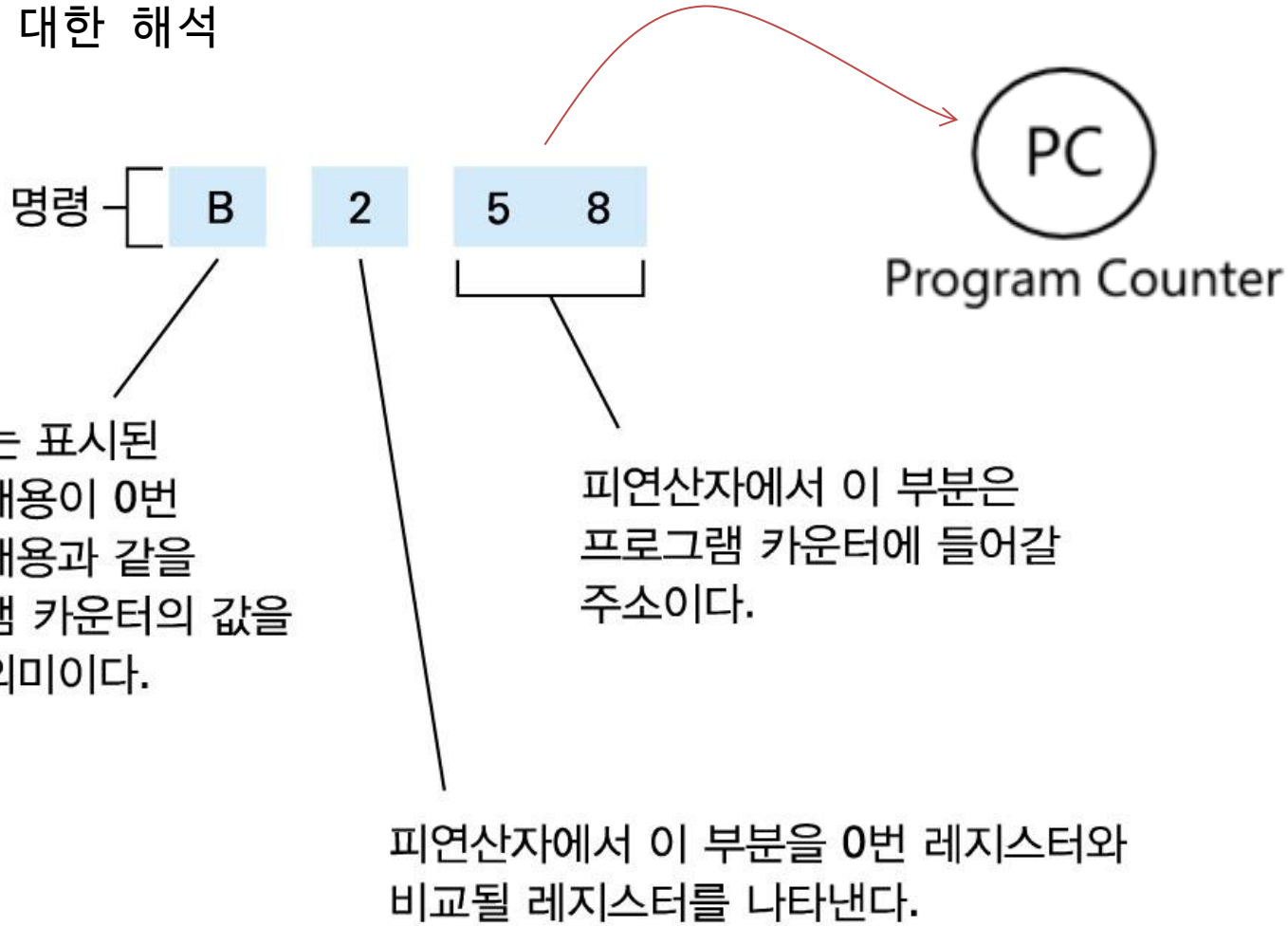
프로그램의 실행은 2개의 용도 지정 레지스터로 제어

- 프로그램 카운터(PC : Program Counter) : 8 비트
  - 다음에 실행될 명령의 주소
  - 컴퓨터가 현재 프로그램의 어느 부분에 와 있는지 추적하는 수단으로 사용
- 명령 레지스터(IR; Instruction Register) : 16 비트
  - 현재 실행 중인 명령을 보관

용도 지정 레지스터

프로그램카운터의 역할

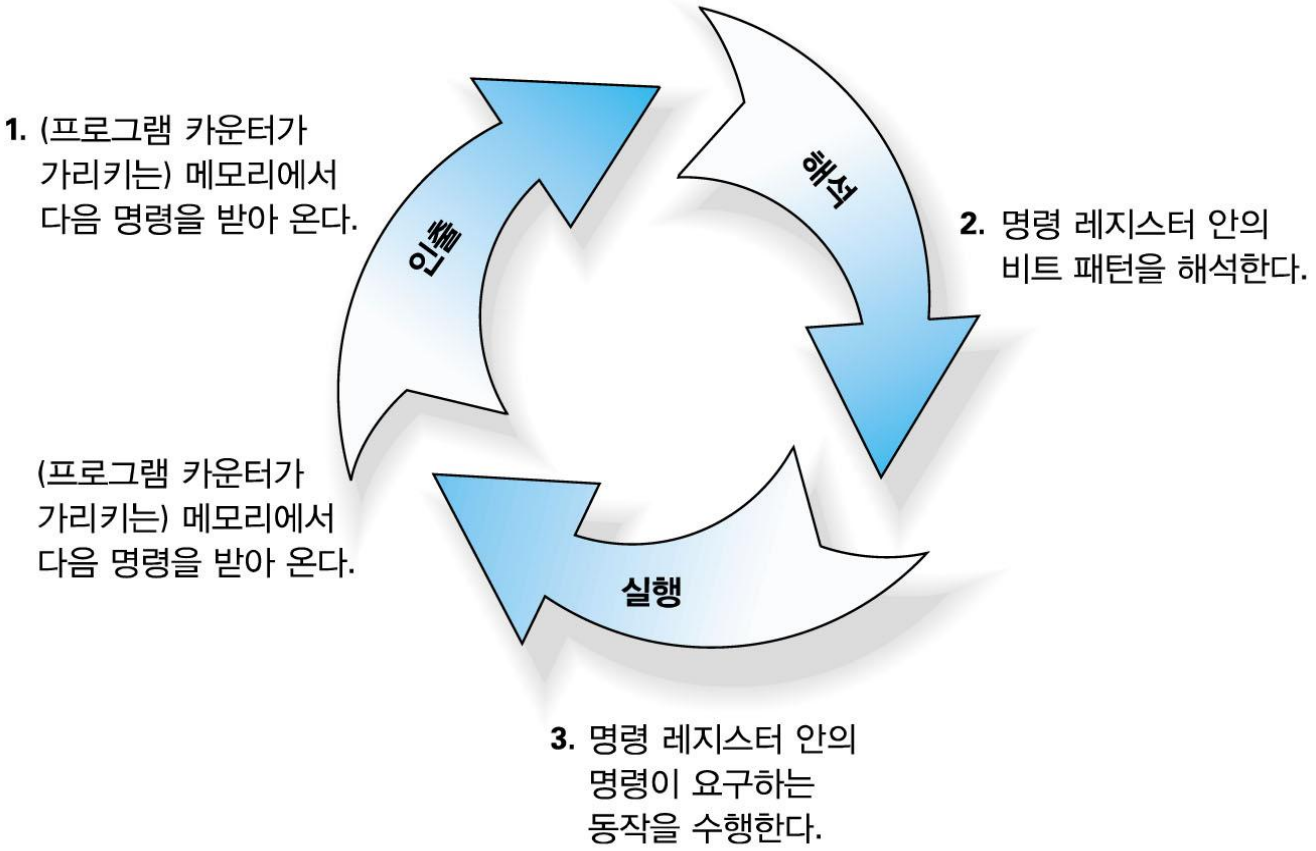
- 명령 B258에 대한 해석



기계 주기

기계 주기(machine cycle)

- CPU의 작업은 기계주기라 불리는 3단계 과정을 반복함으로써 알고리즘을 실행시킴



프로그램 실행의 예 : 더하기 연산

ex) C = A + B ;

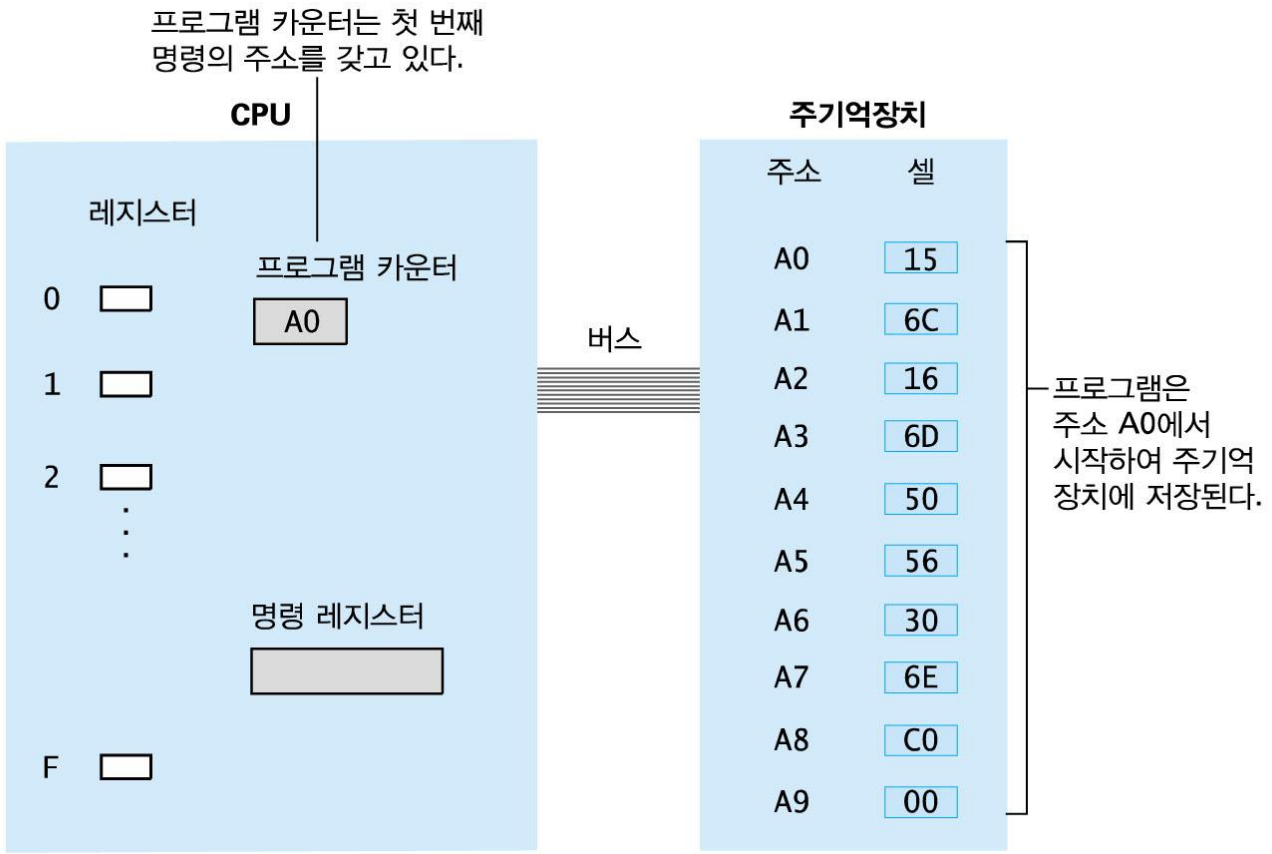
156C : LOAD

166D : LOAD

5056 : ADD

306E : STORE

C000 : HALT

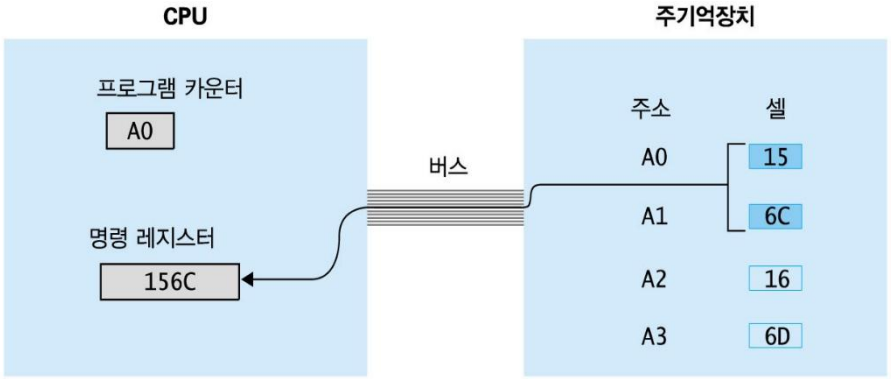


주기억장치에 저장되어 실행 준비된 프로그램

프로그램 실행의 예

156C : LOAD

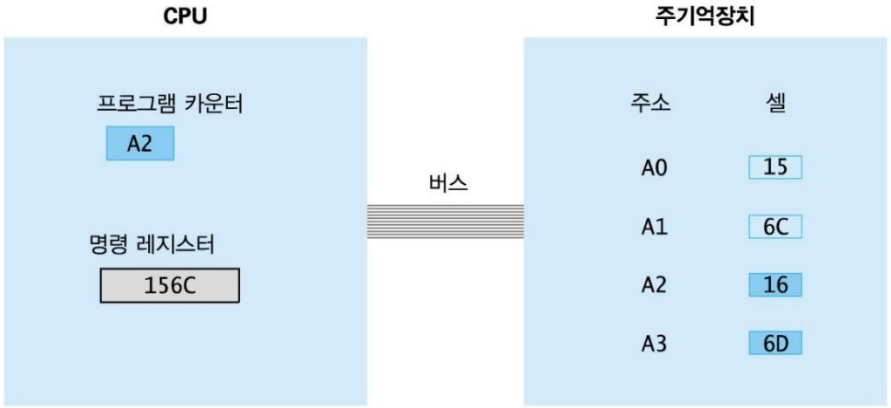
PC: A0	MEM A0: 15
	MEM A1: 6C
인출	IR ← 156C,
	PC ← A2
해석	LOAD R5, 6C로 해석
실행	R5 ← MEM 6C 내용



a. 인출 단계를 시작할 때 주소 A0에서 시작되는 명령을 메모리에서 가져와서 명령 레지스터에 넣는다.

166D : LOAD

PC: A2	MEM A2: 16
	MEM A3: 6D
인출	IR ← 166D
	PC ← A4
해석	LOAD R6, 6D로 해석
실행	R6 ← MEM 6D 내용



b. 그런 다음 프로그램 카운터가 증가되어 다음 명령을 가리킨다.

프로그램 실행의 예 : 더하기 연산

ex) C = A + B ;

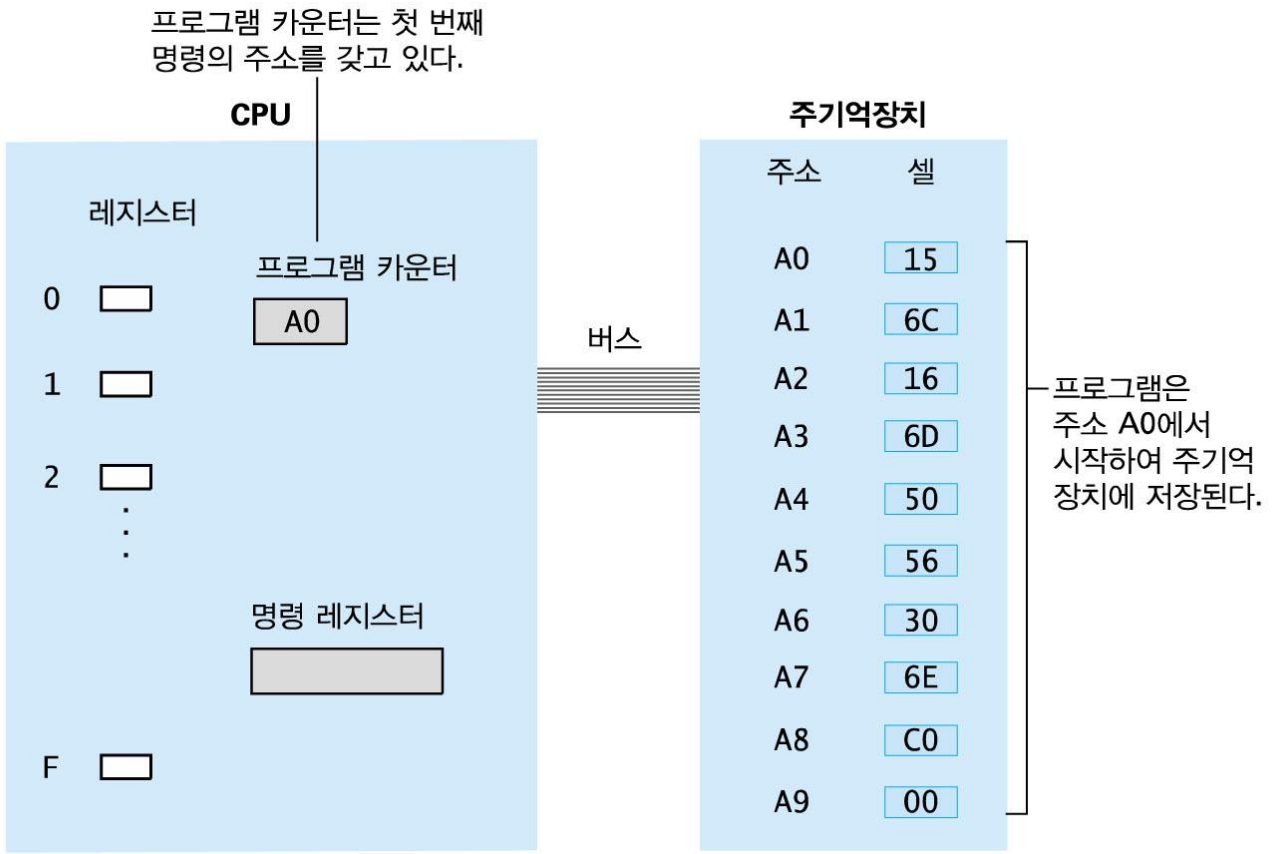
156C : LOAD

166D : LOAD

5056 : ADD

306E : STORE

C000 : HALT



주기억장치에 저장되어 실행 준비된 프로그램