

## 2 기계어

1. 기계 명령: CPU가 인식할 수 있도록 비트 패턴으로 인코딩된 명령      **Instruction Set – The vocabulary of commands understood by a given architecture.**
2. 기계어(Machine Language):  
기계가 인식할 수 있는 모든 명령의 집합
3. 컴퓨터가 처리할 수 있는 데이터는 2진수일뿐임  
즉 메모리에 저장된 내용은 단순한 비트열임  
**가령)** 00011101 :  
컴퓨터 하드웨어의 상황에 따라 다양하게 해석됨  
(명령어? 데이터? 숫자? 등)

## 2 기계어 – High level, Assembly, Machine Language

### 1. High Level Language

- \* User-friendly languages which are similar to a human language with vocabulary of words and symbols.
- \* Instructions in a High-Level Language are called **statements**.
- \* A program written in a high-level language is called the **source code**.
- \* `area = radius * radius * 3.141592;`

```
count = 0;
while (n > 1) {
    count++;
    if (n & 1)
        n = n*3 + 1;
    else
        n = n/2;
}
```

### 2. Assembly Language

- \* Assembly Language uses short descriptive words (mnemonic) to represent each of the Machine Language instructions.
- \* `load r1, radius`      // r1 레지스터에 radius(변수, 메모리)값을 적재  
`multiply r1, r1`      // r1 곱하기 r1 하여 Accumulator(레지스터)에 저장  
`multiply 3.141592`      // Acc의 값과 3.141592를 곱하여 Acc에 저장  
`store area`      // Acc의 내용(값)을 area(변수, 메모리)에 저장

## 2 기계어 – High level, Assembly, Machine Language

### 3. Machine Language

- \* A computer's native language is called Machine Language.
- \* Machine language is the most primitive or basic programming language that starts or takes instructions in the form of raw binary code.

```
count = 0;
while (n > 1) {
    count++;
    if (n & 1)
        n = n*3 + 1;
    else
        n = n/2;
}
```

High Level Language

```
        movl    $0, %ecx
loop:   cmpl    $1, %edx
        jle     endloop
        addl    $1, %ecx
        movl    %edx, %eax
        andl    $1, %eax
        je      else
        movl    %edx, %eax
        addl    %eax, %edx
        addl    %eax, %edx
        addl    $1, %edx
        jmp     endif
else:   sarl    $1, %edx
endif:  jmp     loop
endloop:
```

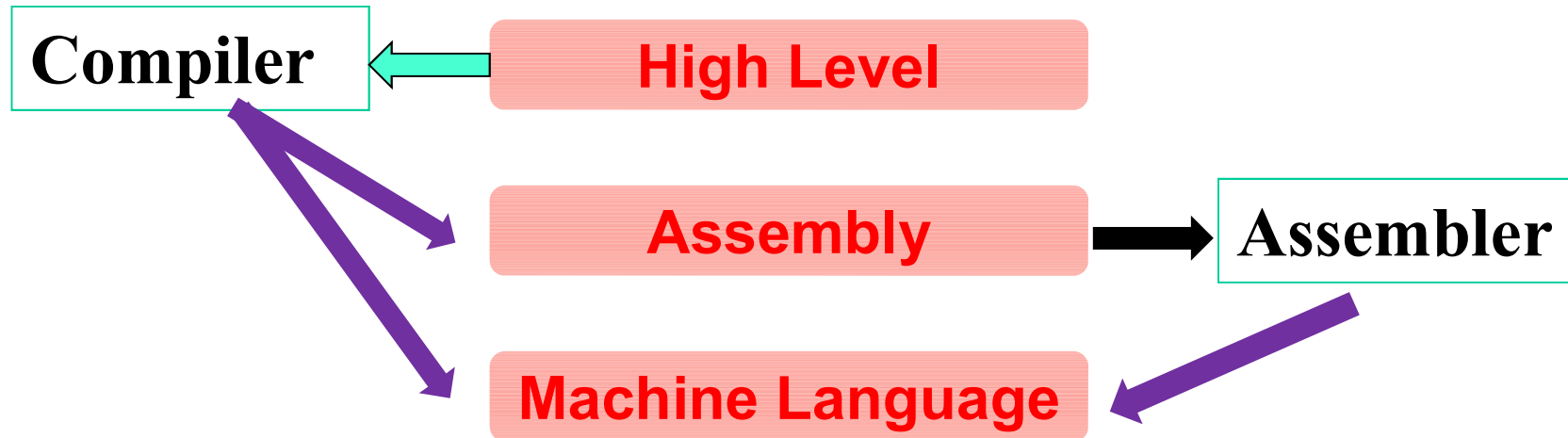
Assembly Language

0000	0000	0000	0000	0000	0000	0000	0000
0000	0000	0000	0000	0000	0000	0000	0000
9222	9120	1121	A120	1121	A121	7211	0000
0000	0001	0002	0003	0004	0005	0006	0007
0008	0009	000A	000B	000C	000D	000E	000F
0000	0000	0000	FE10	FACE	CAFE	ACED	CEDE
1234	5678	9ABC	DEF0	0000	0000	F00D	0000
0000	0000	EEEE	1111	EEEE	1111	0000	0000
B1B2	F1F5	0000	0000	0000	0000	0000	0000

Machine Language

## 2 기계어 – High level, Assembly, Machine Language

### 4. Compiler and Assembler

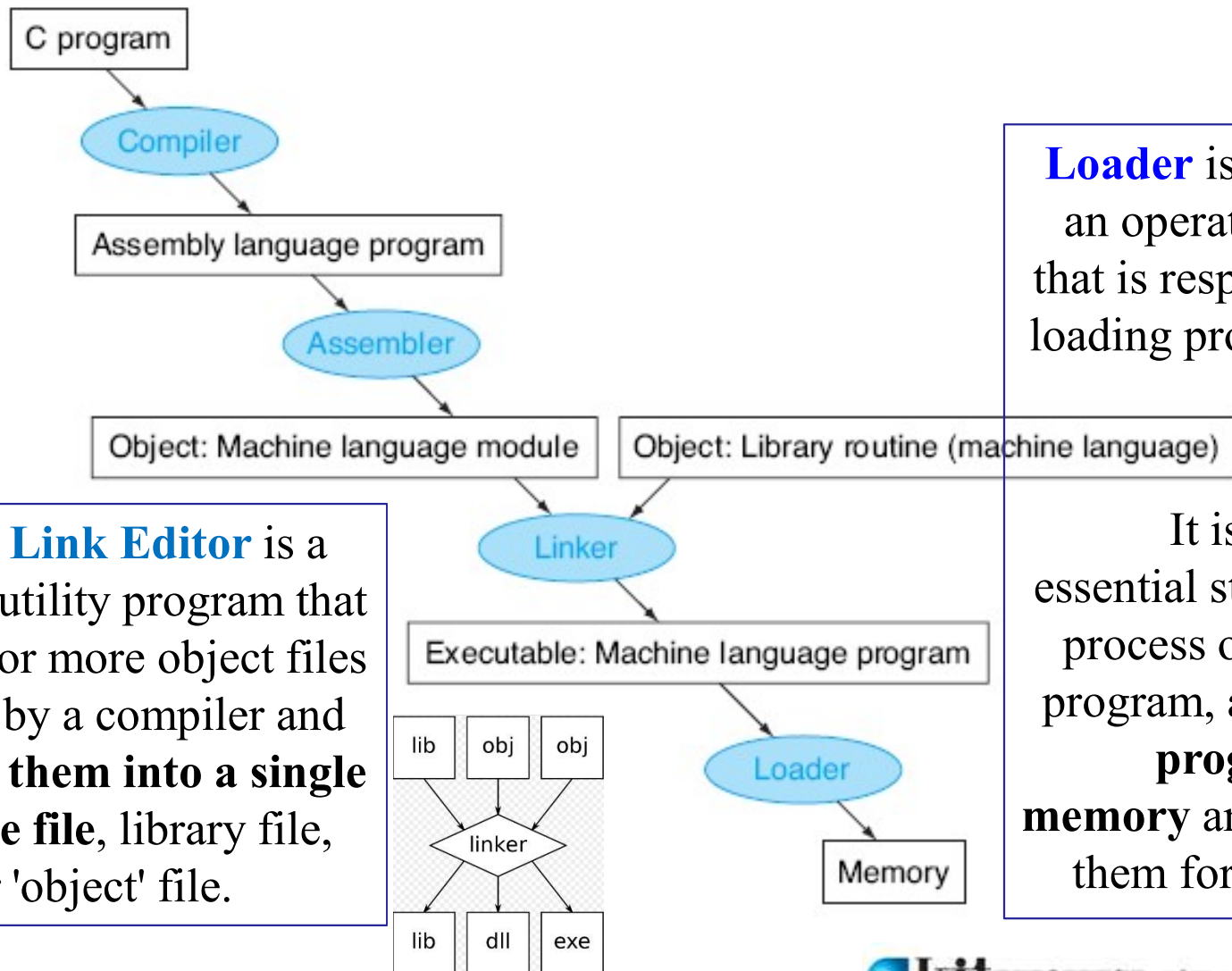


### \* Compiler, Interpreter

An **interpreter** is a computer program that **directly executes**, i.e. performs instructions written in a programming or scripting language, without requiring them previously to have been compiled into a machine language program.

## 2 기계어 – High level, Assembly, Machine Language

### 5. Linker and Loader



**Linker** or **Link Editor** is a computer utility program that takes one or more object files generated by a compiler and **combines them into a single executable file**, library file, or another 'object' file.

**Loader** is the part of an operating system that is responsible for loading programs and libraries.

It is one of the essential stages in the process of starting a program, as **it places programs into memory** and prepares them for execution.

## 2.1 기계어: 명령의 종류

- 데이터 전송(data transfer):**  
한 장소에서 다른 장소로 데이터를 복사한다(load, store)
- 연산(arithmetic/logic):**  
기존의 비트 패턴을 사용하여 새로운 비트 패턴을 계산한다(add, and, not)
- 제어(control):**  
프로그램 실행을 지시한다(jmp, jsr)

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADD <sup>+</sup>	0001				DR			SR1		0	00			SR2		
ADD <sup>+</sup>	0001				DR			SR1		1			imm5			
AND <sup>+</sup>	0101				DR			SR1		0	00			SR2		
AND <sup>+</sup>	0101				DR			SR1		1			imm5			
BR	0000			n	z	p										PCoffset9
JMP	1100						000			BaseR						000000
JSR	0100				1											PCoffset11
JSRR	0100				0		00			BaseR						000000
LD <sup>+</sup>	0010				DR											PCoffset9
LDI <sup>+</sup>	1010				DR											PCoffset9
LDR <sup>+</sup>	0110				DR			BaseR								offset6
LEA <sup>+</sup>	1110				DR											PCoffset9
NOT <sup>+</sup>	1001				DR			SR								111111
RET	1100						000			111						000000
RTI	1000															000000000000
ST	0011				SR											PCoffset9
STI	1011				SR											PCoffset9
STR	0111				SR			BaseR								offset6
TRAP	1111						0000									trapvect8
reserved	1101															

## 2.2 기계어: 명령의 종류

### 1. 명령어 형식의 종류

0-주소 명령어 

연산코드
------

 (예: PUSH, POP)

1-주소 명령어 

연산코드	오퍼랜드
------	------

 (예: ADD A)

2-주소 명령어 

연산코드	오퍼랜드 1	오퍼랜드 2
------	--------	--------

 (예: ADD A, B)

3-주소 명령어 

연산코드	오퍼랜드 1	오퍼랜드 2	오퍼랜드 3
------	--------	--------	--------

 (예: ADD A, B, C)

#### □ 연산 코드(Operation code)

- CPU가 수행할 연산을 지정

연산 코드	오퍼랜드 ( <i>addr</i> )
-------	----------------------

#### □ 오퍼랜드(Operand)

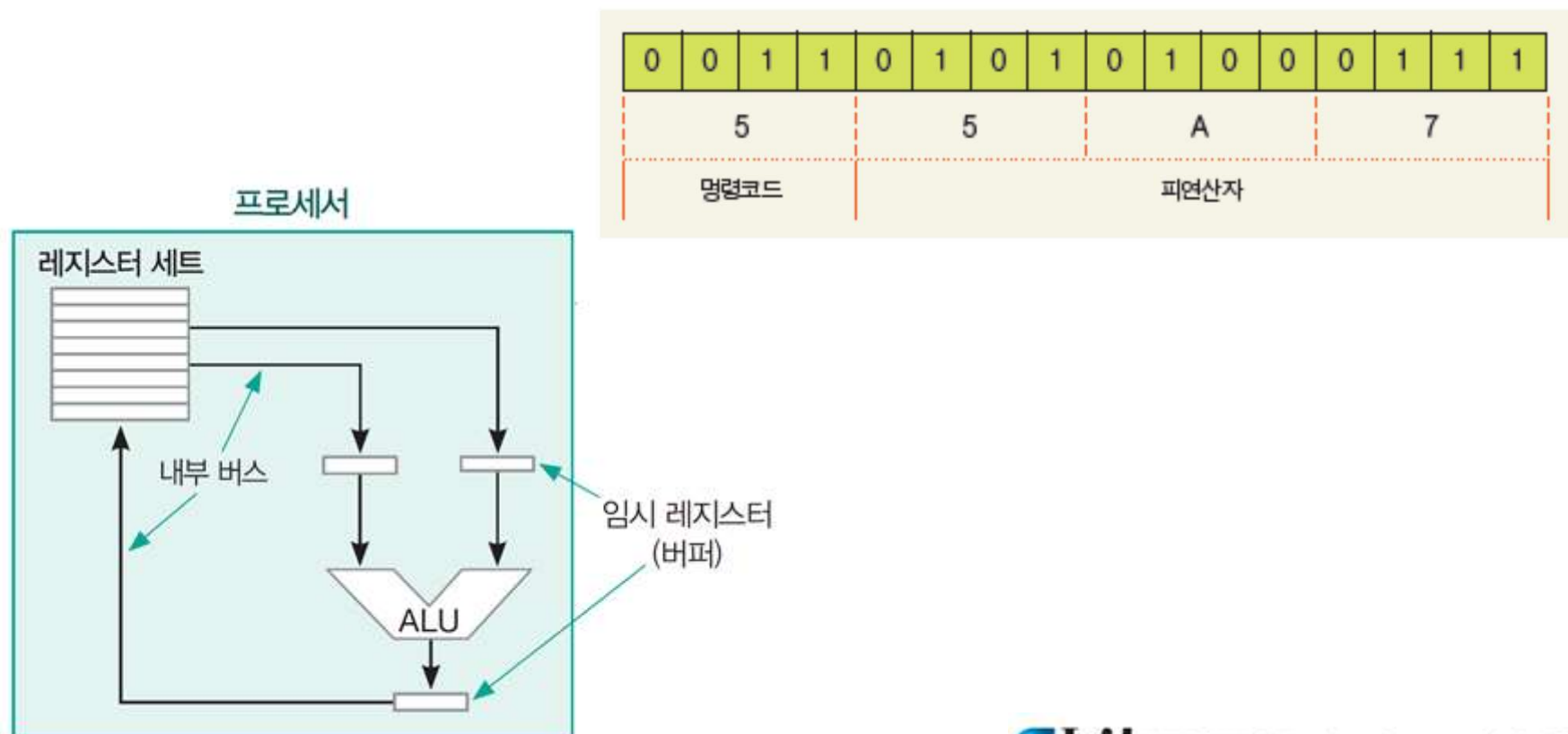
- 명령어 실행에 필요한 데이터가 저장된 주소(*address*)



## 2.2 기계 명령의 요소들

1. 명령코드(opcode): 실행할 명령을 지정 (덧셈 뺄셈 등)
2. 피연산자(operand): 명령에 관한 추가 정보를 제공
  - A. 피연산자에 대한 해석은 명령 코드에 따라 다름

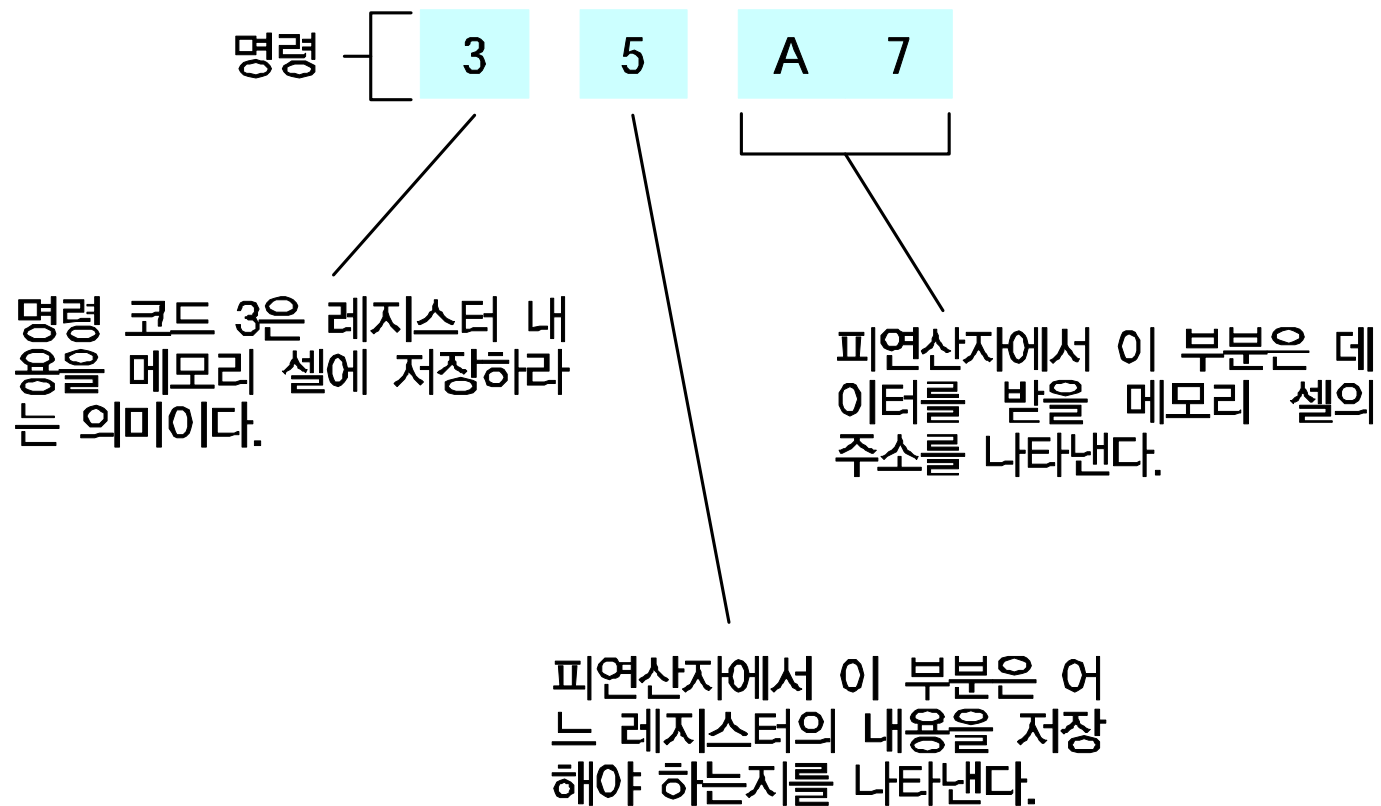
### 3. 16비트 컴퓨터 경우의 예





## 2.2 기계어 명령 0x35A7의 해석

1. CPU 하드웨어마다 명령이 다르고, 의미도 다르다.
2. 따라서 프로그래밍 언어가 생성하는 기계어 코드도 다르다.



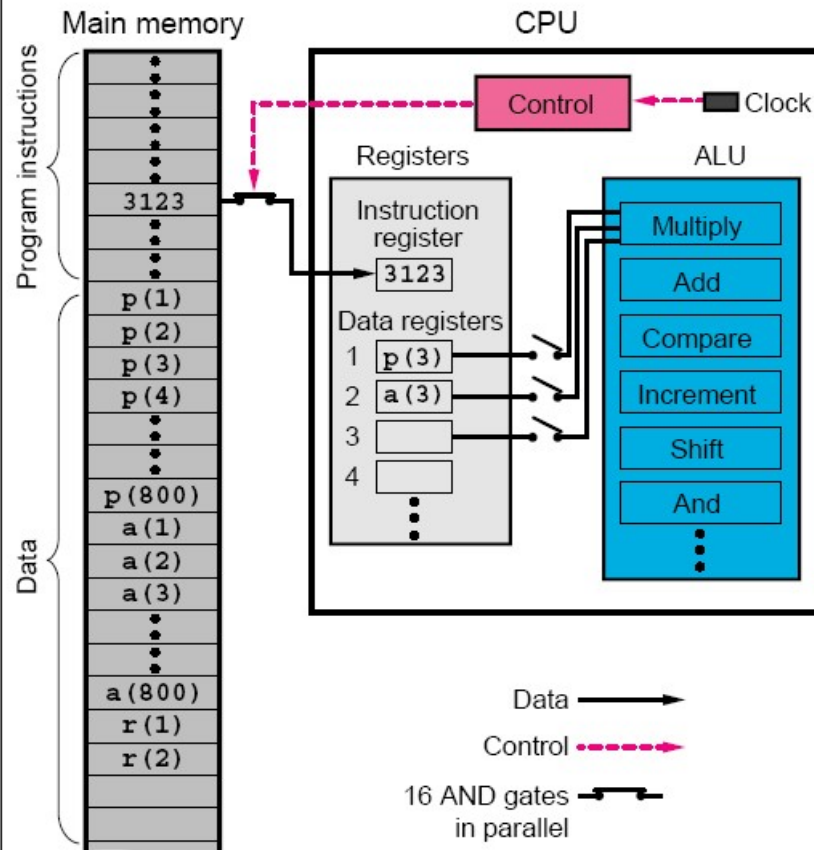
## 2.2 명령들에 대한 인코딩 버전 (덧셈 예제 경우)

인코딩된 명령	해설
156C	주소가 6C인 메모리 셀에 들어있는 비트 패턴으로 5번 레지스터를 채운다.
166D	주소가 6D인 메모리 셀에 들어있는 비트 패턴으로 6번 레지스터를 채운다.
5056	5번 레지스터와 6번 레지스터의 내용에 대해 2의 보수 덧셈을 수행하고, 그 결과를 0번 레지스터에 넣는다.
306E	0번 레지스터의 내용을 주소가 6E인 메모리 셀에 저장한다.
C000	멈춘다.

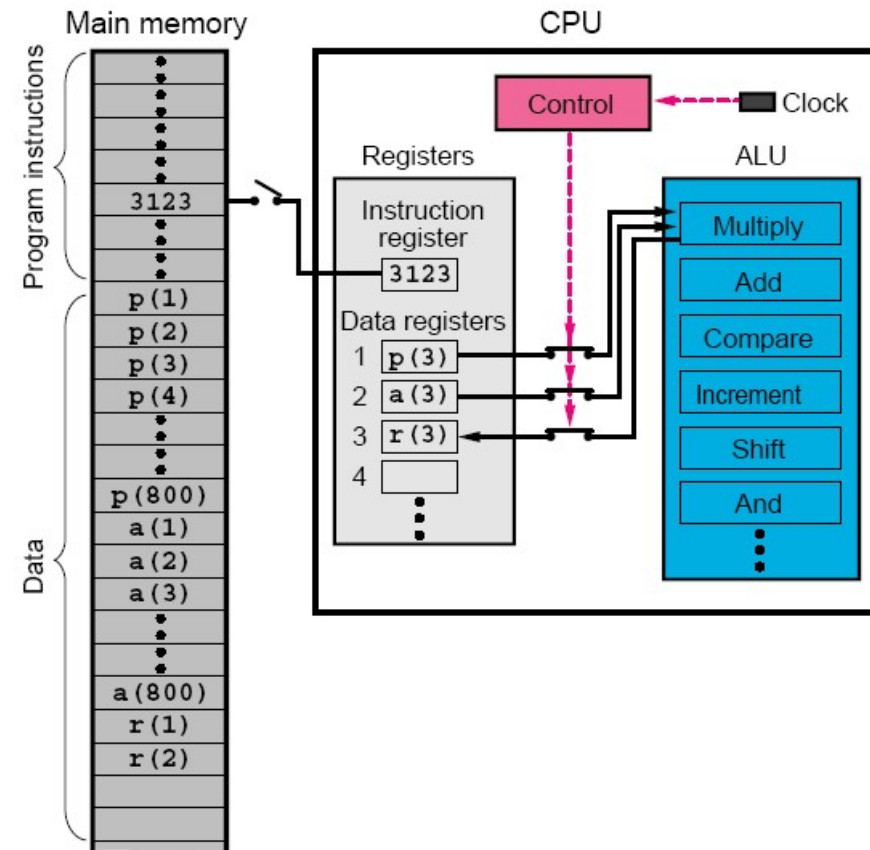
## 2.3 프로그램의 실행 과정

### Execution of One Instruction

(a) Fetch instruction to multiply



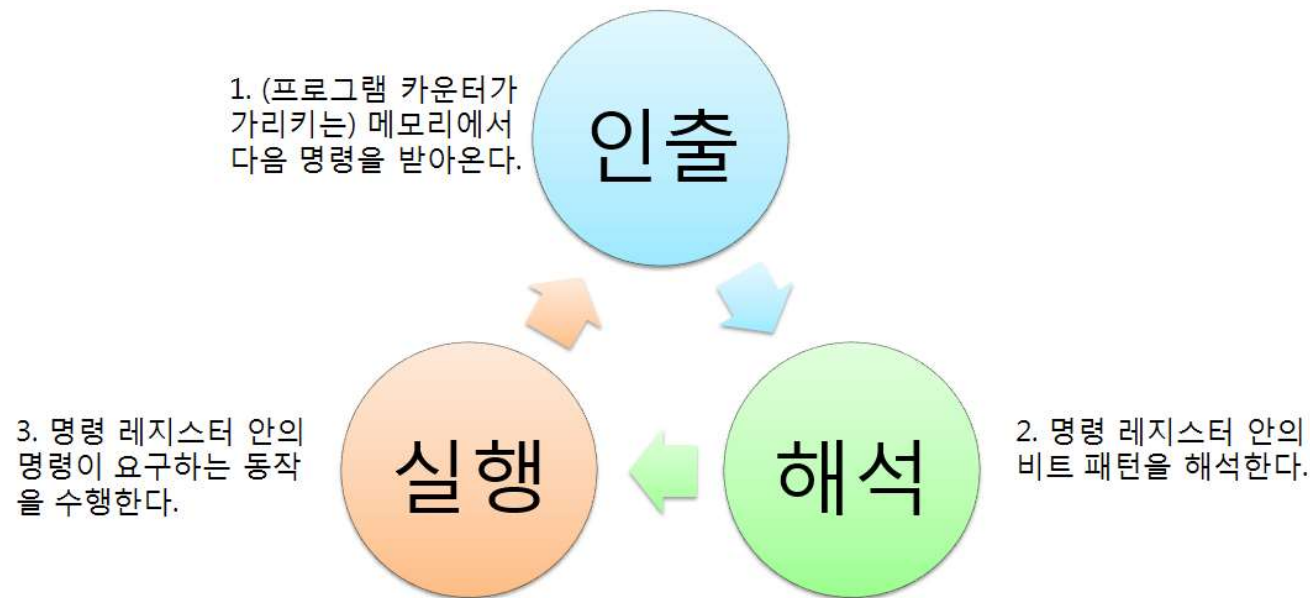
(b) Execute multiplication



## 2.3 프로그램의 실행 - 기계 주기

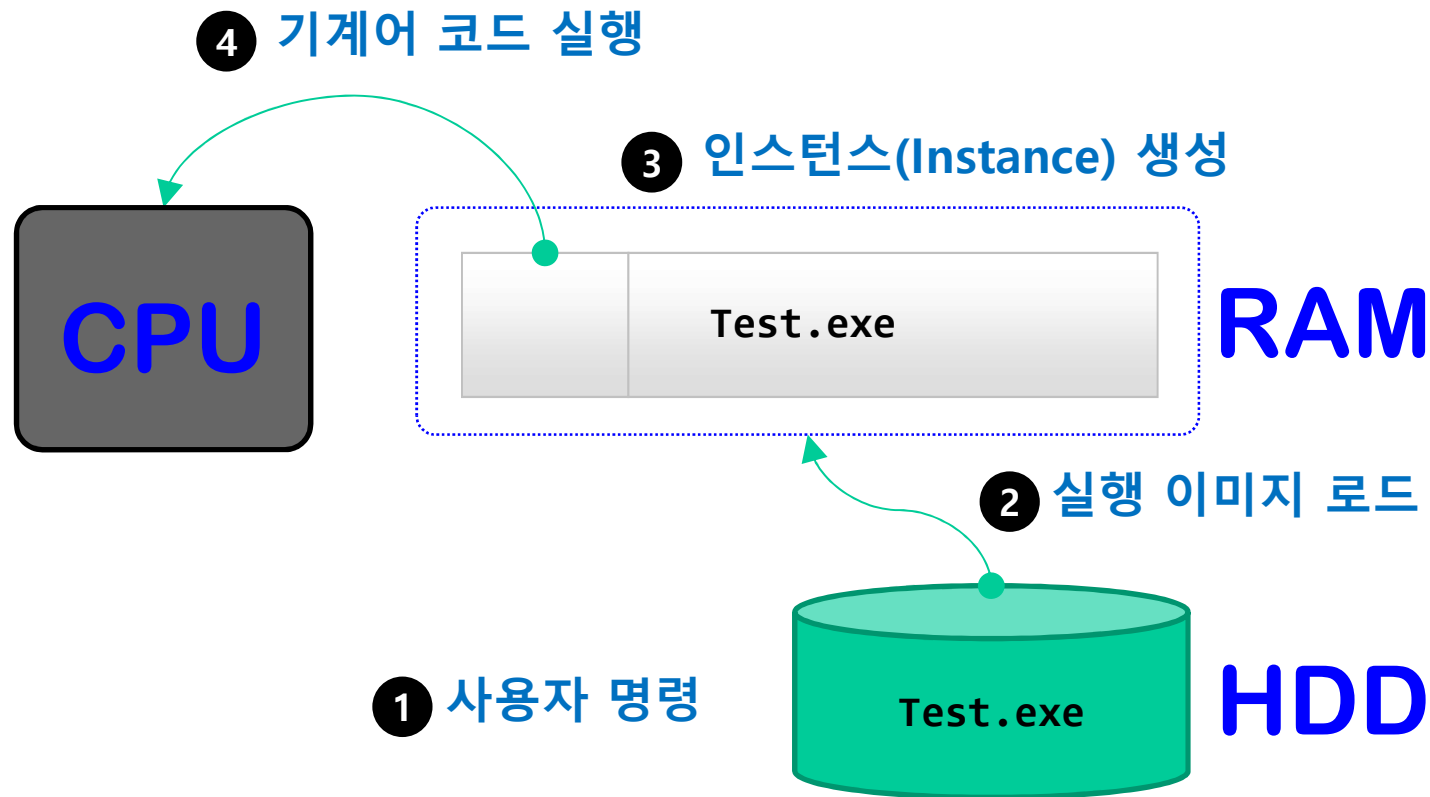
### 1. 기계 주기(machine cycle) :

- A. 인출(Fetch)
- B. 해석(Decode)
- C. 실행(Execute)



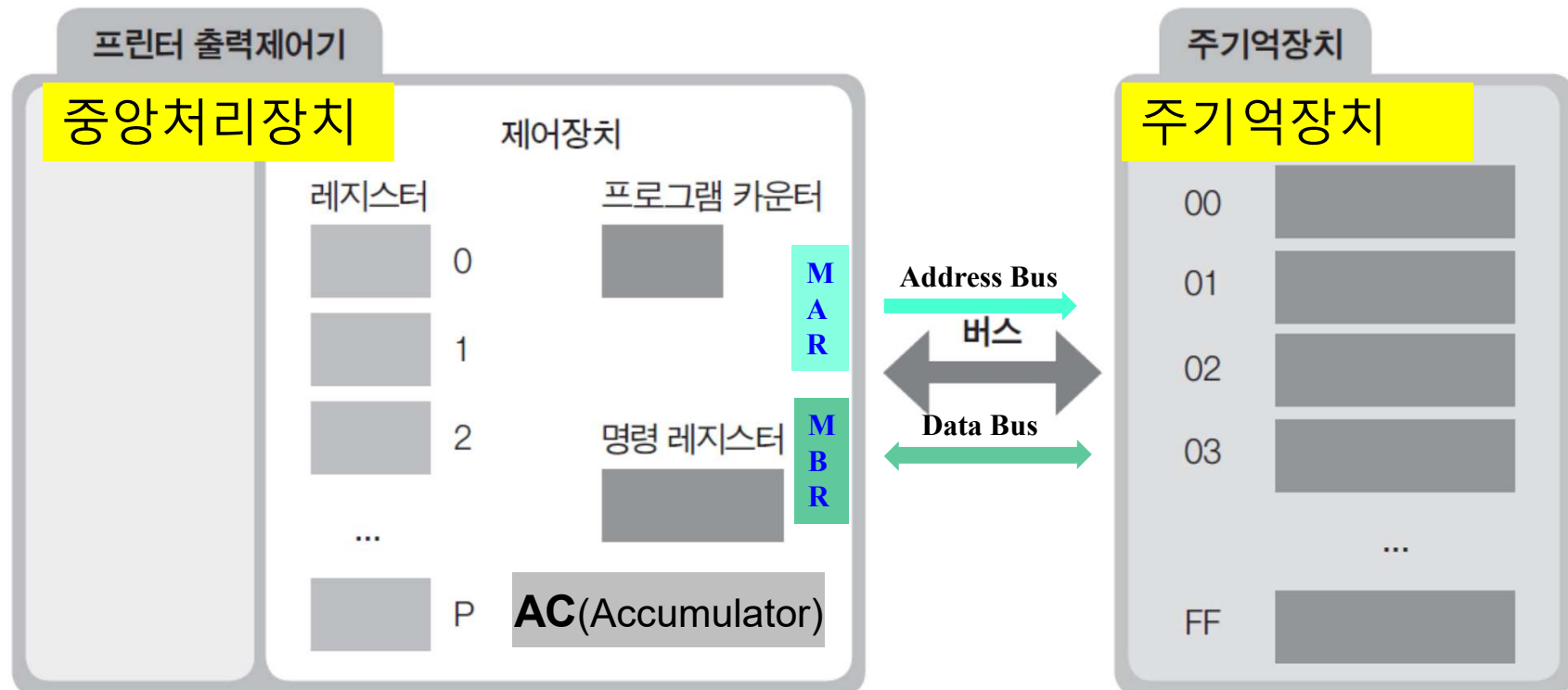
- D. Clock에 의하여 동기화되며, 기계 주기를 반복 수행하여 프로그램 실행

## 2.3 프로그램의 실행 과정



## 2.3 가상 컴퓨터 시스템 가정

1. 레지스터: 연산을 위한 값을 가지고 있는 소규모 기억장치
2. 명령 레지스터: 현재 수행할 명령을 보관
3. 프로그램 카운터: 다음 실행할 명령의 주소를 보관



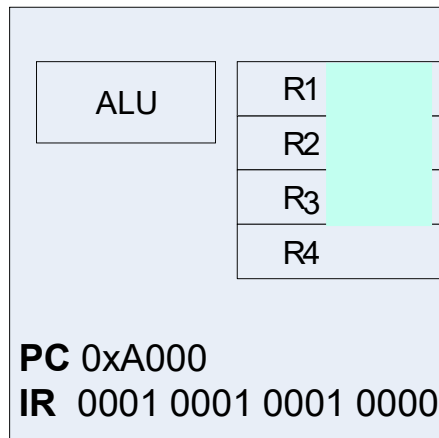
**MAR** (Memory Address Register)

**MBR** (Memory Buffer Register)

## 2.3 프로그램의 실행 과정-1

### PC 0xA000 실행 전

#### CPU



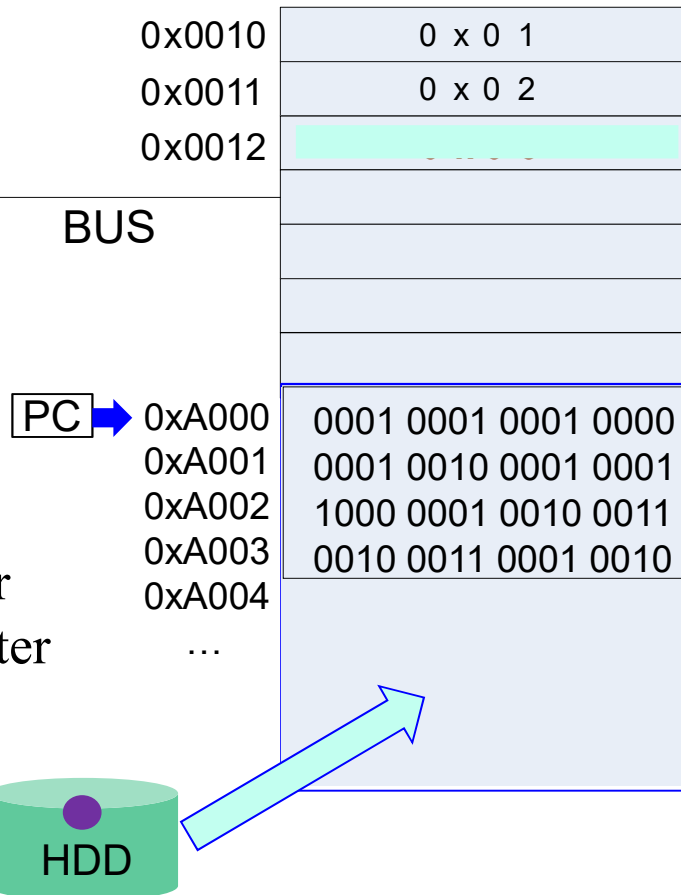
#### Special Register

PC - Program Counter

IR - Instruction Register

AC - Accumulator

#### RAM



#### Symbol Table

변수 a	0x0010
변수 b	0x0011
변수 c	0x0012

절대 주소 / 상대 주소

```
Int a = 1 ;
Int b = 2 ;
Int c = a + b;
```

```
LOAD R1 0x0010
LOAD R2 0x0011
ADD R1 R2 R3
SAVE R3 0x0012
```

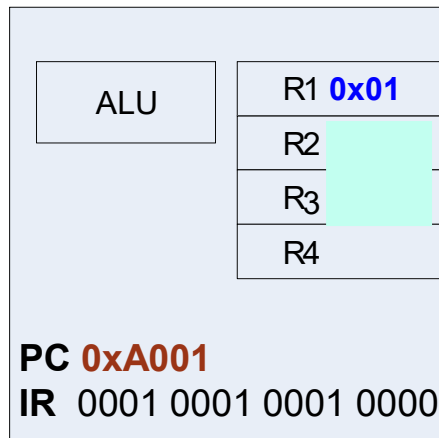
```
0001 0001 0001 0000
0001 0010 0001 0001
1000 0001 0010 0011
0010 0011 0001 0010
```



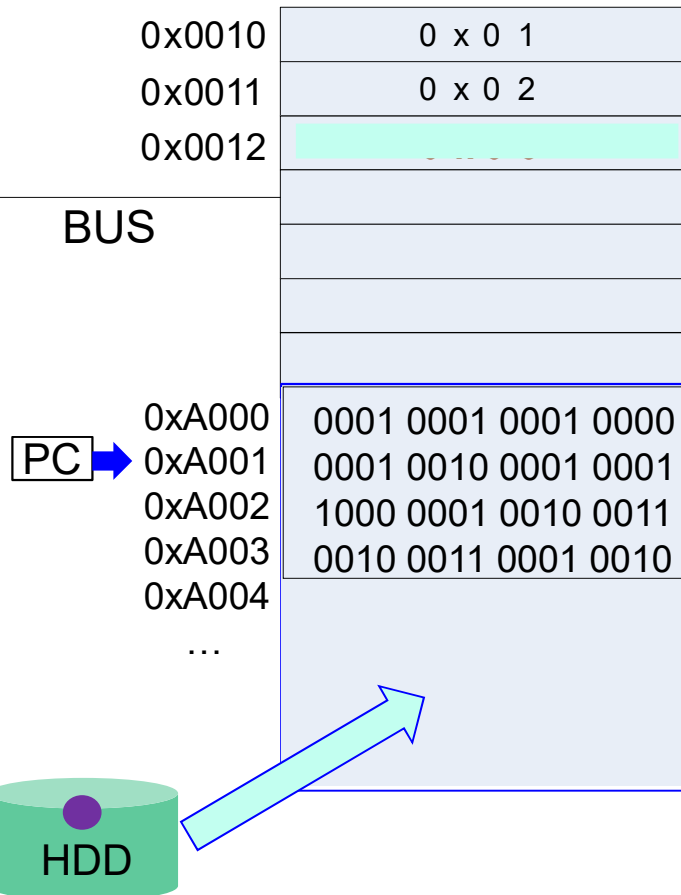
## 2.3 프로그램의 실행 과정-2

### PC 0xA000 실행 후

#### CPU



#### RAM



#### Symbol Table

변수 a	0x0010
변수 b	0x0011
변수 c	0x0012

절대 주소 / 상대 주소

```
Int a = 1 ;
Int b = 2 ;
Int c = a + b;
```

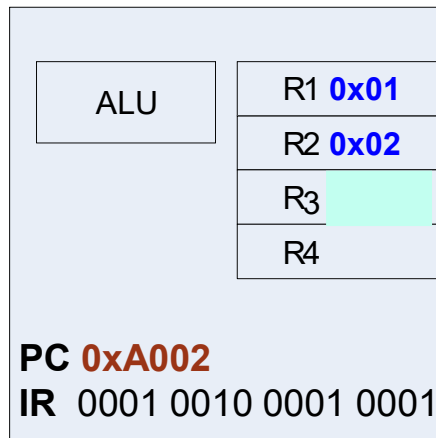
```
LOAD R1 0x0010
LOAD R2 0x0011
ADD R1 R2 R3
SAVE R3 0x0012
```

```
0001 0001 0001 0000
0001 0010 0001 0001
1000 0001 0010 0011
0010 0011 0001 0010
```

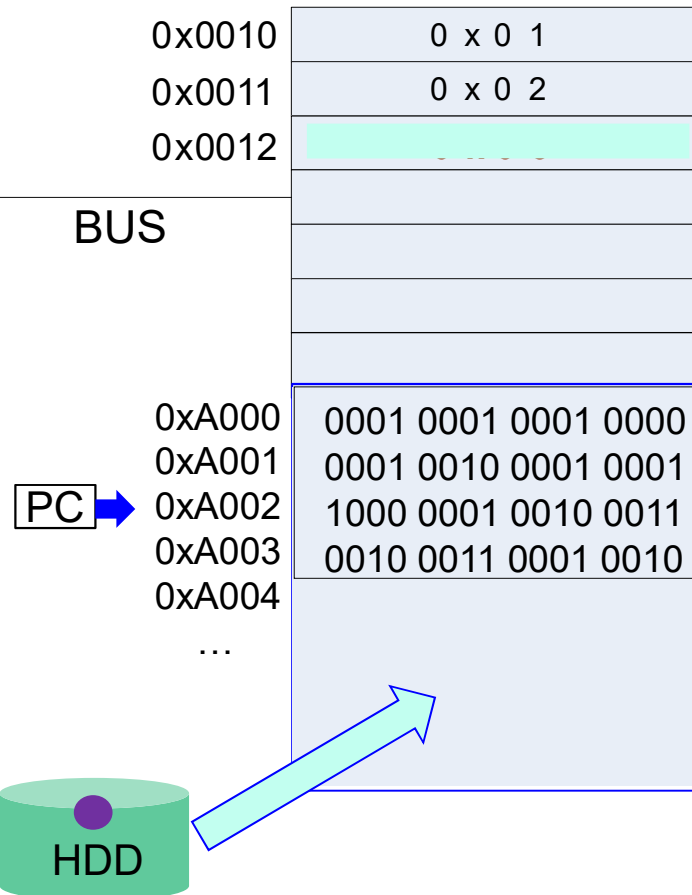
## 2.3 프로그램의 실행 과정-3

### PC 0xA001 실행 후

#### CPU



#### RAM



#### Symbol Table

변수 a	<b>0x0010</b>
변수 b	<b>0x0011</b>
변수 c	<b>0x0012</b>

절대 주소 / 상대 주소

```
Int a = 1 ;
Int b = 2 ;
Int c = a + b;
```

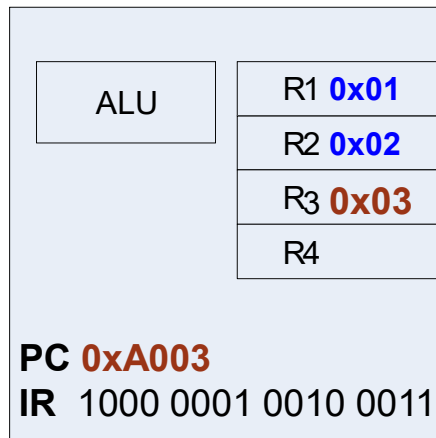
```
LOAD R1 0x0010
LOAD R2 0x0011
ADD R1 R2 R3
SAVE R3 0x0012
```

```
0001 0001 0001 0000
0001 0010 0001 0001
1000 0001 0010 0011
0010 0011 0001 0010
```

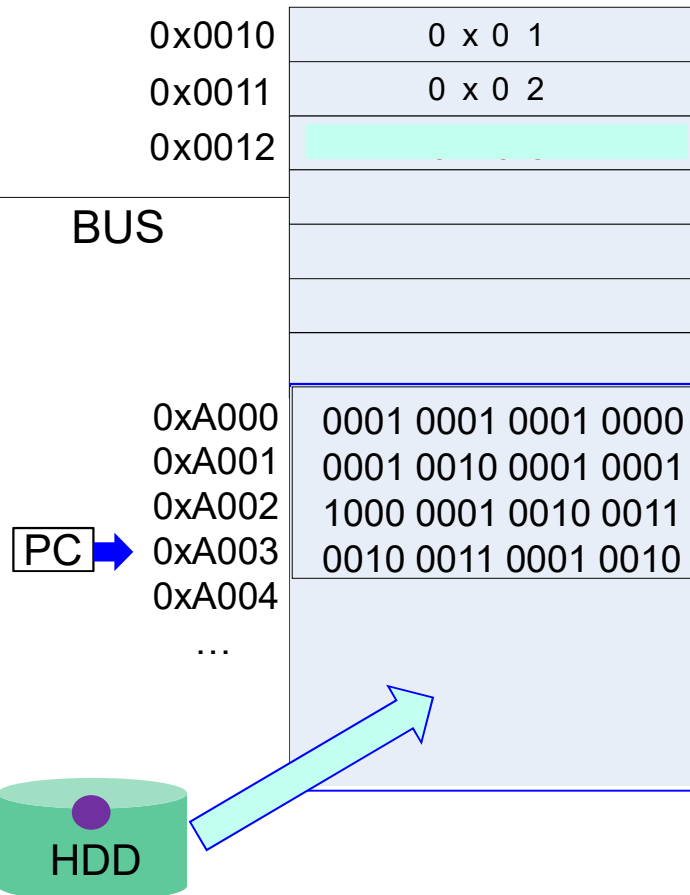
## 2.3 프로그램의 실행 과정-4

### PC 0xA002 실행 후

#### CPU



#### RAM



#### Symbol Table

변수 a	<b>0x0010</b>
변수 b	<b>0x0011</b>
변수 c	<b>0x0012</b>

절대 주소 / 상대 주소

```
Int a = 1 ;
Int b = 2 ;
Int c = a + b;
```

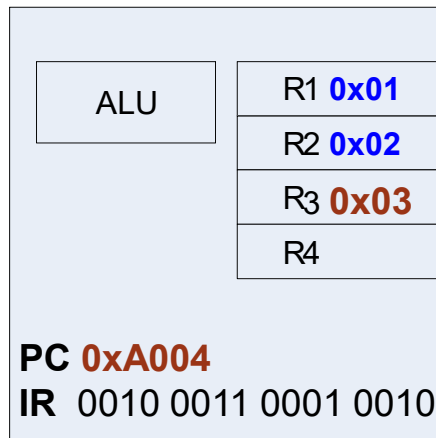
```
LOAD R1 0x0010
LOAD R2 0x0011
ADD R1 R2 R3
SAVE R3 0x0012
```

```
0001 0001 0001 0000
0001 0010 0001 0001
1000 0001 0010 0011
0010 0011 0001 0010
```

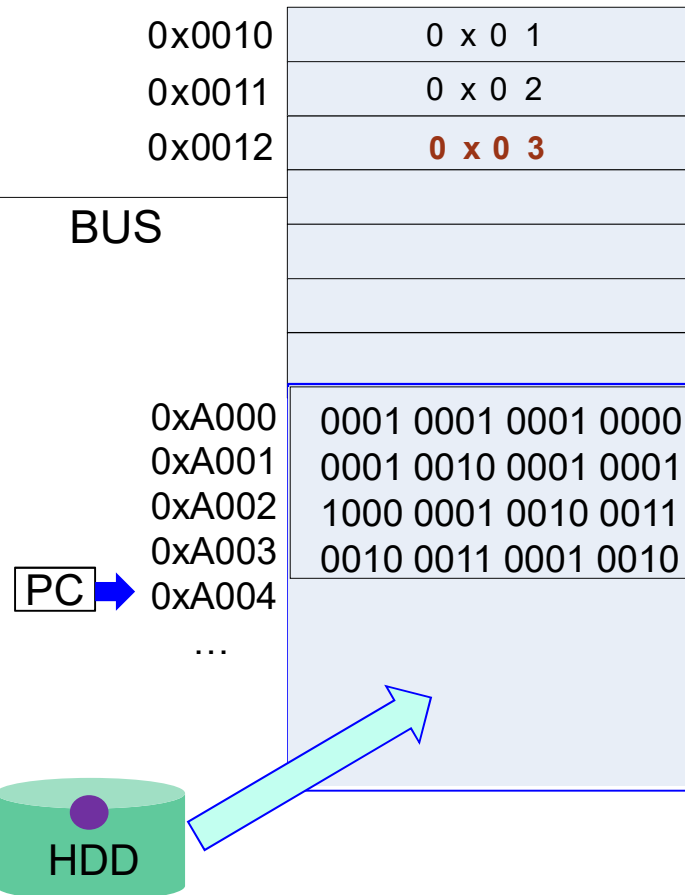
## 2.3 프로그램의 실행 과정-5

### PC 0xA003 실행 후

#### CPU



#### RAM



#### Symbol Table

변수 a	<b>0x0010</b>
변수 b	<b>0x0011</b>
변수 c	<b>0x0012</b>

절대 주소 / 상대 주소

```
Int a = 1 ;
Int b = 2 ;
Int c = a + b;
```

```
LOAD R1 0x0010
LOAD R2 0x0011
ADD R1 R2 R3
SAVE R3 0x0012
```

```
0001 0001 0001 0000
0001 0010 0001 0001
1000 0001 0010 0011
0010 0011 0001 0010
```

## 2.4 기계어 철학의 양대 축

### 1. RISC(Reduced Instruction Set Computing)

- A. 단순하고, 빠르고, 효율적인 소수의 명령
- B. 예: ARM, Apple, Motorola의 PowerPC

### 2. CISC(Complex Instruction Set Computing)

- A. 편리하고 강력한 다수의 명령
- B. 예: Intel의 Pentium

#### ● CISC와 RISC의 차이점

#### CISC

- 복합명령어를 사용함으로써 전체 명령어의 수를 줄일 수 있으므로 프로그램의 실행시간을 줄일 수 있음.
- 다양한 명령어 형식을 갖고 있기 때문에 연산코드를 해석하고 실행할 때 제어신호를 발생시키는 제어장치를 복잡하게 만들.

#### RISC

- 각 명령어의 길이를 가능한 한 짧게 함으로써 각 명령어의 실행시간을 최소화 할 수 있음.
- 많은 처리량과 빠른 속도를 얻는 것을 목적으로 하며, 제어장치는 비교적 간단하고 하드웨어로 구성되는 것이 일반적.

## 2.4 기계어 철학의 양대 축

### ● CISC와 RISC의 특징

#### CISC

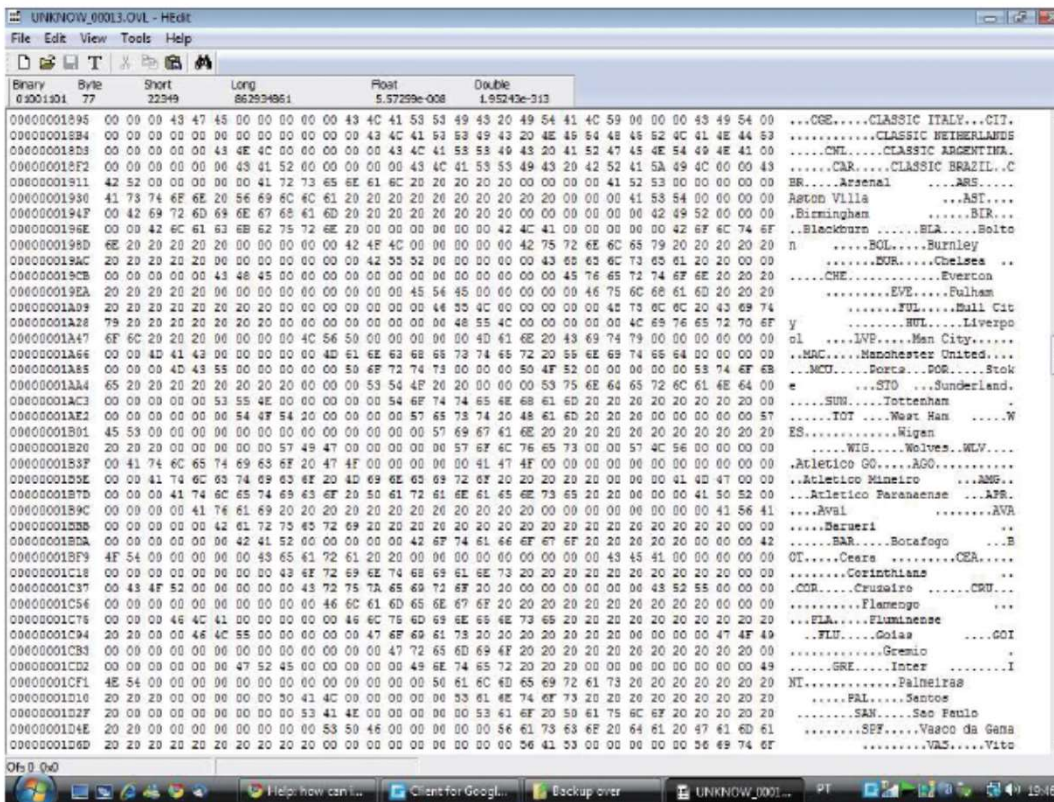
- 명령어의 수가 많다.
- 주소지정방식이 다양하다.
- 명령어 형식은 여러 개의 길이를 갖는다.
- 대부분의 명령어는 직접적으로 기억장치 액세스를 할 수 있다.
- 명령어는 기본적인 연산과 복잡한 연산을 모두 수행한다.

#### RISC

- 명령어의 수가 최소화되어 있다.
- 주소지정방식의 수가 제한되어 있다.
- 명령어 형식은 모두 같은 길이를 갖는다.
- 기억장치 액세스는 로드와 스토어 명령에 의해서만 가능하다.
- 명령어는 기본적인 연산기능만을 수행한다.

### 3 프로그램 및 데이터

1. 주기억장치 안에는 여러 개의 프로그램이 서로 다른 영역에 위치하여 동시에 저장 – **Multi-Programming**  
보호 문제
2. 프로그램과 데이터 구분이 어려움  
(실행 시점의 해석에 의해 결정됨)



Code(Program) Area  
Data Area

접근 주소가  
- 범위를 벗어나면