



## C언어 – HW1

임베디드스쿨1기

Lv1과정

2020. 07. 21

손표훈

## 2. C언어의 필요성

---

- (1) C언어란? 기계어(1과 0으로 구성된 명령어 체계)를 인간이 직관적으로 이해할 수 있게 해주는 언어
- (2) C언어 그대로 사람이 의도한 내용을 전달하지 못함  
"컴파일러"를 통해 C -> 기계어로 변환하여 전달
- (3) C언어 말고도 어셈블리어라고 기계어와 1:1 대응 되는 언어가 있다.
- (4) 앞에 나온 임베디드 시스템에서 **MCU**에 C언어로 설계된 펌웨어를 탑재하여 시스템을 제어한다.

# 1. Microprocessor? Microcontroller?

- (1) Microprocessor : 그림1과 같이 **범용** 컴퓨터 시스템에 사용되는 칩을 말한다.
- (2) Microprocessor는 칩 자체만으로 아무것도 할 수 없다.
- (3) **외부에서** 여러 주변기기들이 연결 되어야만 사용자의 목적에 맞는 시스템을 구성할 수 있다.



그림1. 범용 컴퓨터 시스템

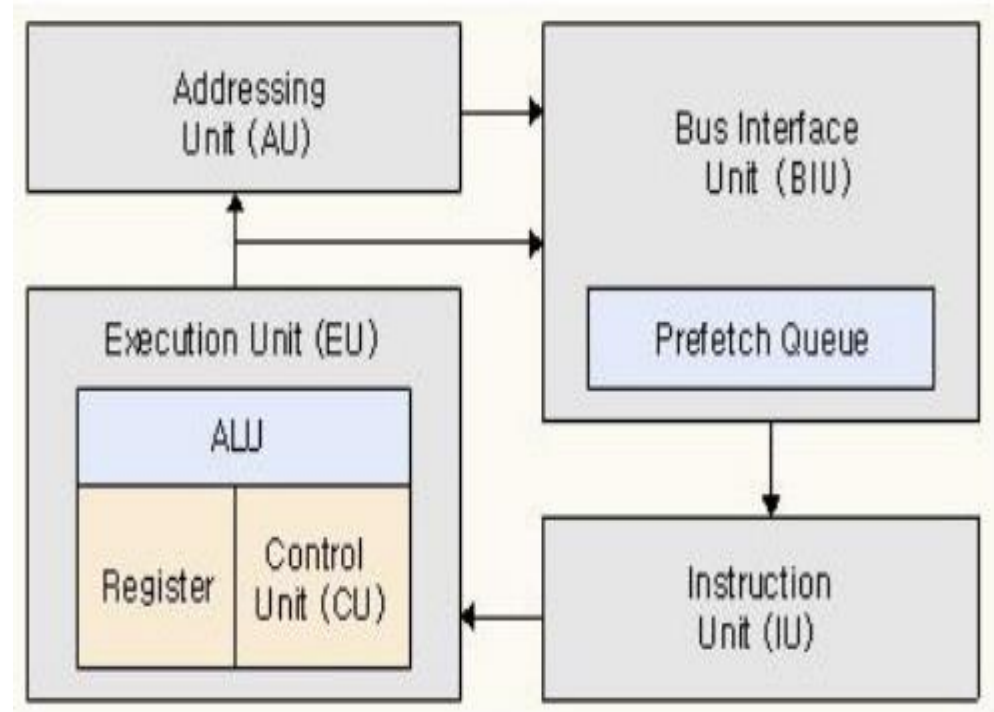


그림2. Microprocessor 구조

# 1. Microprocessor? Microcontroller?

(3) Microcontroller : 아래 그림1과 같이 "**특정기능**"을 위한 임베디드 시스템에 사용되는 칩을 말한다.

(4) Microcontroller는 그림2와 같이 주변장치들이 "**하나의 칩**"에 CPU와 연결되어있다.

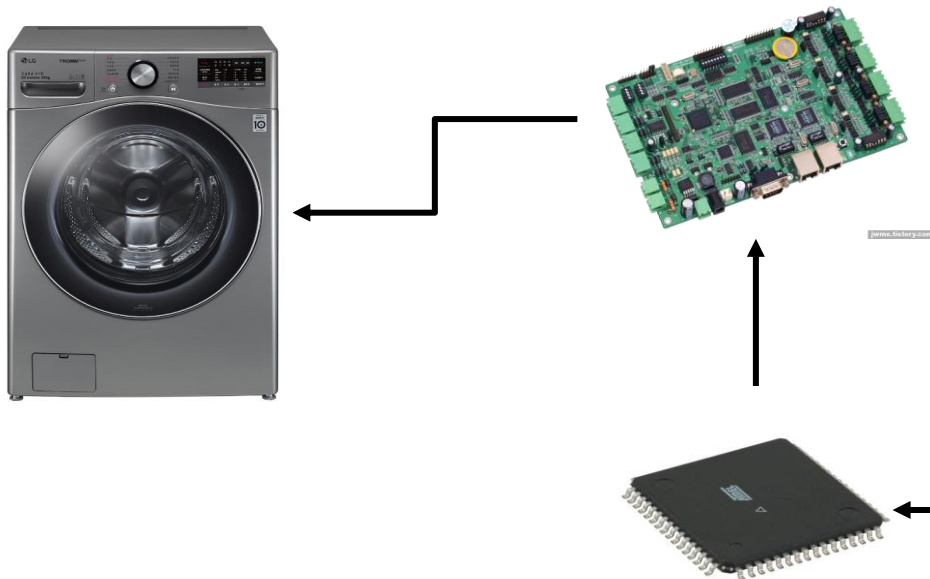


그림1. 임베디드 시스템 예시(세탁기)

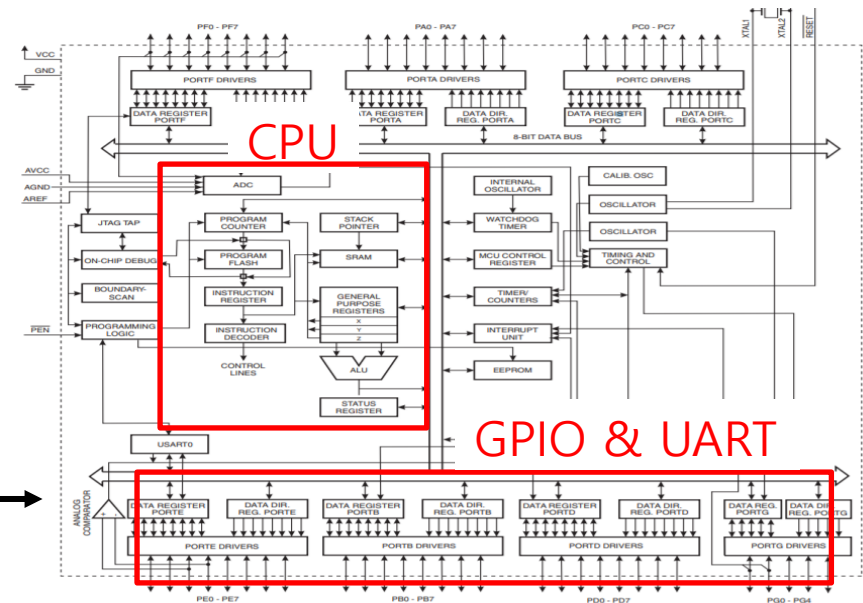


그림2. Microcontroller(Atmega128)

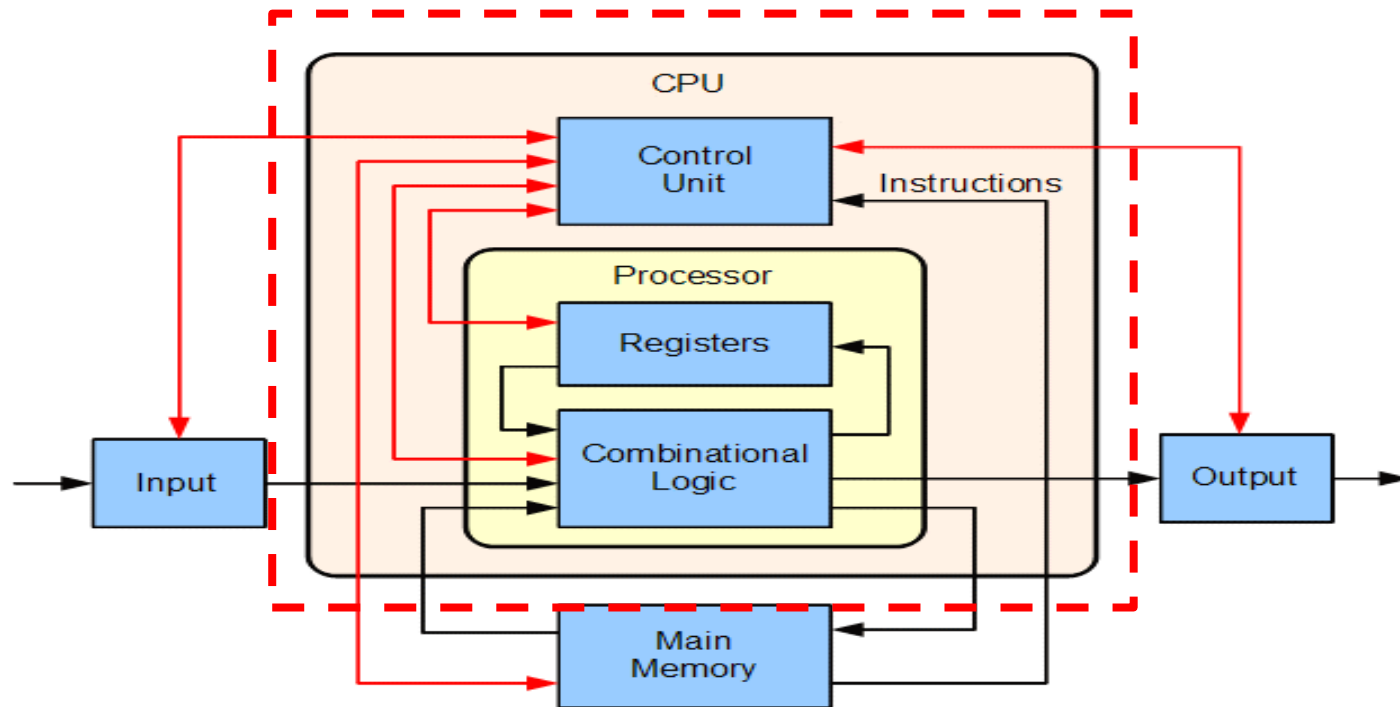
## 2. 폰 노이만 구조?

\*추가내용 : 컴퓨터 하드웨어



## 2. 폰 노이만 구조?

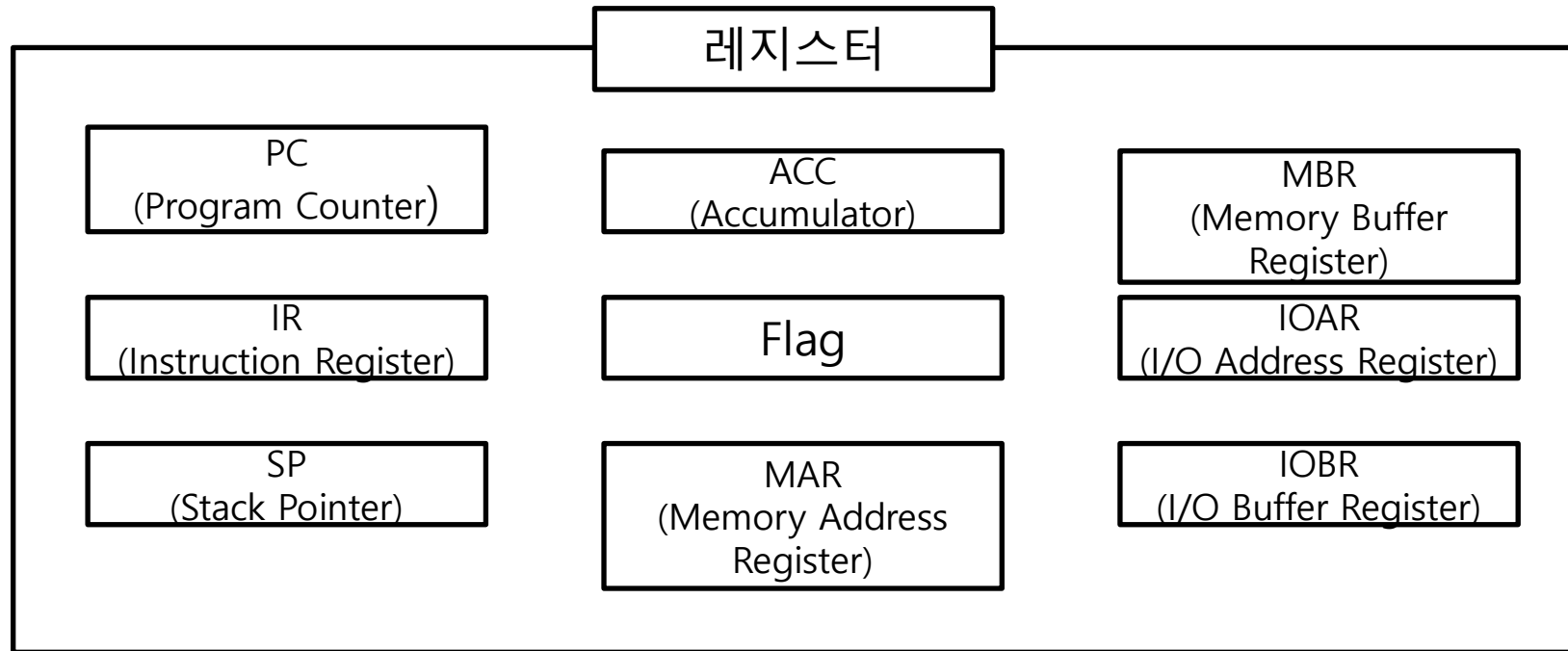
- (1) 중앙처리 장치란? 명령어를 기억장치로 부터 불러와 해석, 명령어 수행을 위한 전체적인 제어를 담당하며, ALU, CU, Register Set으 구성됨.



- CPU와 메모리간의 연결은 **BUS**라는 양방향으로 데이터를 주고 받을 수 있는 일종의 Wire로 연결되어 있다.
- CPU를 구성하는 각 부분들에 대해 알아보자!

## 2. 폰 노이만 구조?

- ① 레지스터? CPU내에서 처리할 명령어와 연산데이터, 각종 처리 데이터를 저장한다.  
플립플롭으로 구성되어 있으며, 병렬 입/출력으로 구성되어있다.  
고속으로 데이터를 입/출력 할 수 있다.



- PC : CPU CLK에 따라 증가하며 다음에 실행하기 위해 인출할 명령어의 메모리상의 주소 값을 저장하며, 명령어 인출 후 **명령어 길이만큼 증가**된다.
- IR : 현재 실행되고 있는 명령어를 저장한다(명령어 파이프라인 구성 시 IR을 병렬로 여러 개 사용하여 구성한다)

## 2. 폰 노이만 구조?

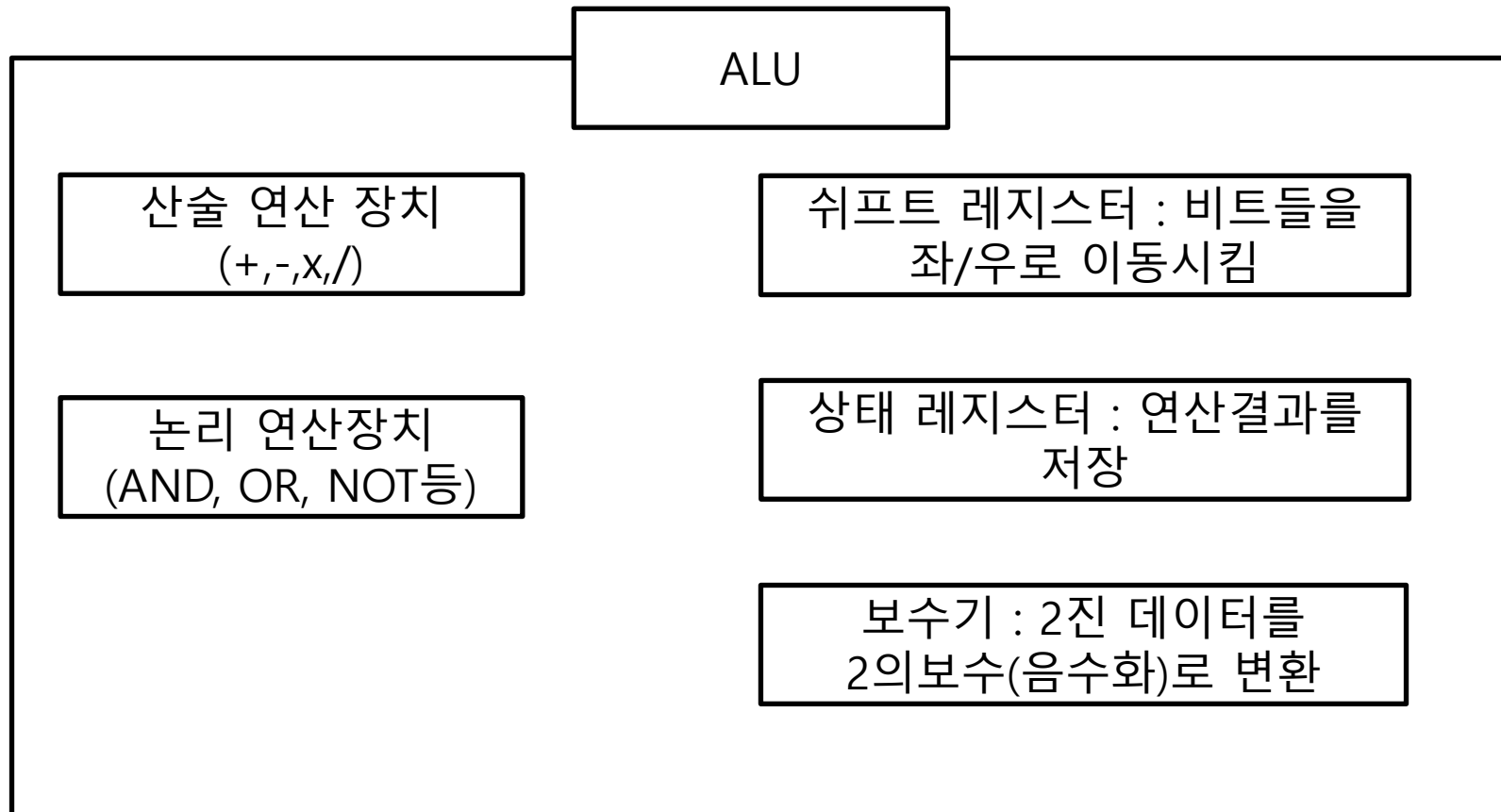
---

- SP : 데이터 세그먼트의 STACK영역에 저장된 서브루틴(함수)이 실행되면 해당되는 서브루틴의 주소를 저장한다.
- ACC : ALU의 계산결과를 저장하는 레지스터이다.
- Flag Register : 연산결과를 비트단위로 할당하여 저장한다.  
Ex) interrupt flag : 1이면 CPU에 인터럽트 활성화 상태임을 알 수 있다.
- MAR : 접근할 메모리 주소를 임시로 저장한다.
- MBR : MBR에 저장된 데이터를 메모리에 저장하거나 메모리로 부터 받은 데이터를 임시로 저장한다.
- IOAR : I/O장치의 주소를 임시로 저장한다.
- IOBR : I/O장치로 부터 받거나 보낼 데이터를 임시로 저장한다.



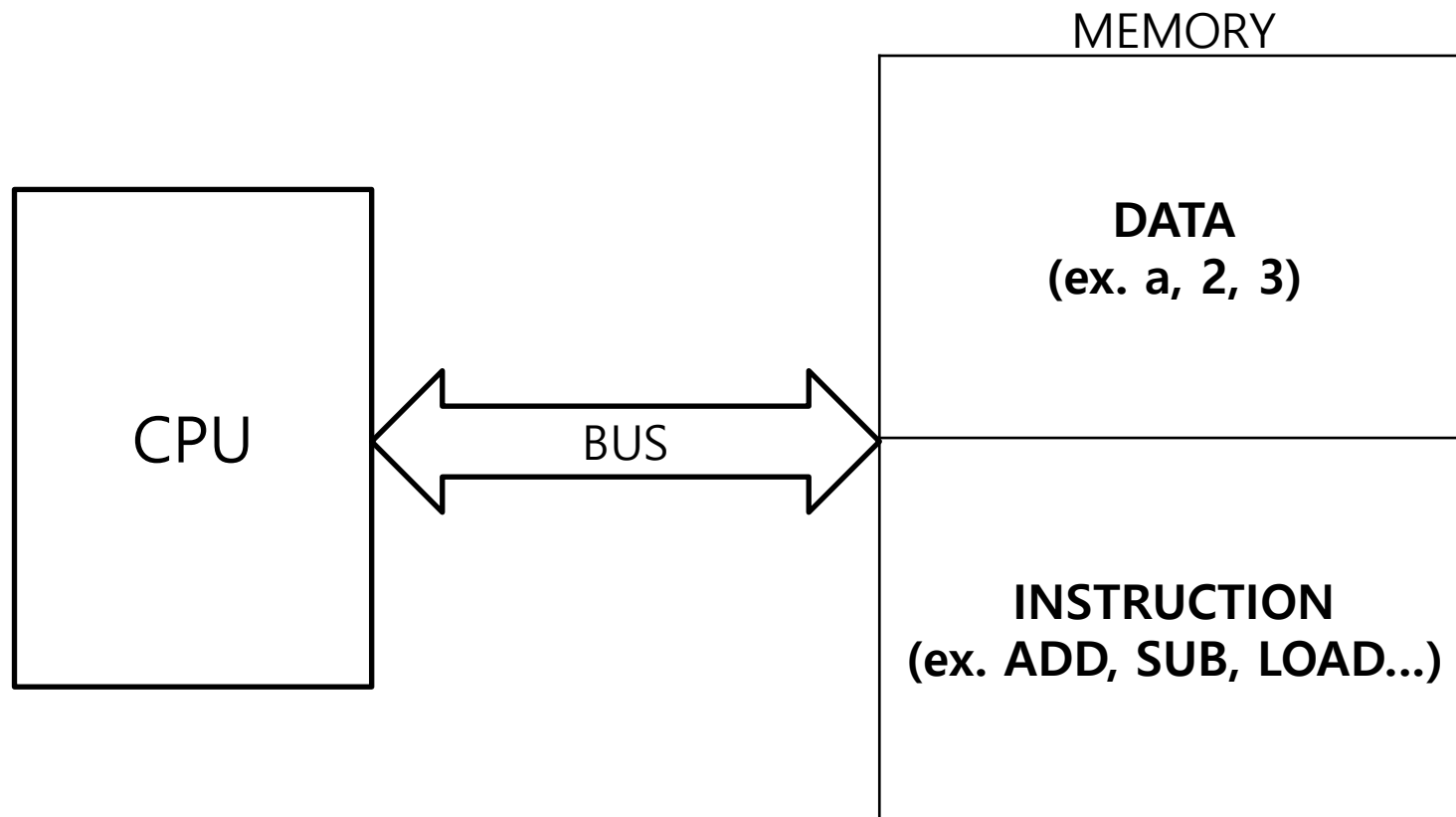
## 2. 폰 노이만 구조?

- ② Control Unit(CU)? 실행할 명령어의 해석과 명령어에 따라 주변장치들(메모리, 입/출력장치 등)에 제어신호를 보낸다.  
CPU의 핵심 구성품이다.
- ③ ALU(Arithmetic Logic Unit)? ALU는 아래 그림과 같은 구성품들로 구성되어 해당 구성품의 기능을 수행한다.



## 2. 폰 노이만 구조?

- (1) 폰 노이만 구조 : 동일 메모리내에 데이터와 명령어가 같이 들어있다.  
메모리에 프로그램 내장, 순차 처리방식  
하나의 버스로 데이터와 명령어를 CPU에서 처리한다



## 2. 폰 노이만 구조?

(2) 폰 노이만 구조 단점 : 병목현상 -> 폰노이만 구조에서 **파이프라이닝 기법** 사용시 메모리 지연발생

(3) 파이프라이닝 기법이란? 우선 아래 CPU의 명령어 실행과정을 살펴보자

Ex. 명령1 : A+B, 명령2 : C+D, 명령3 : E+F

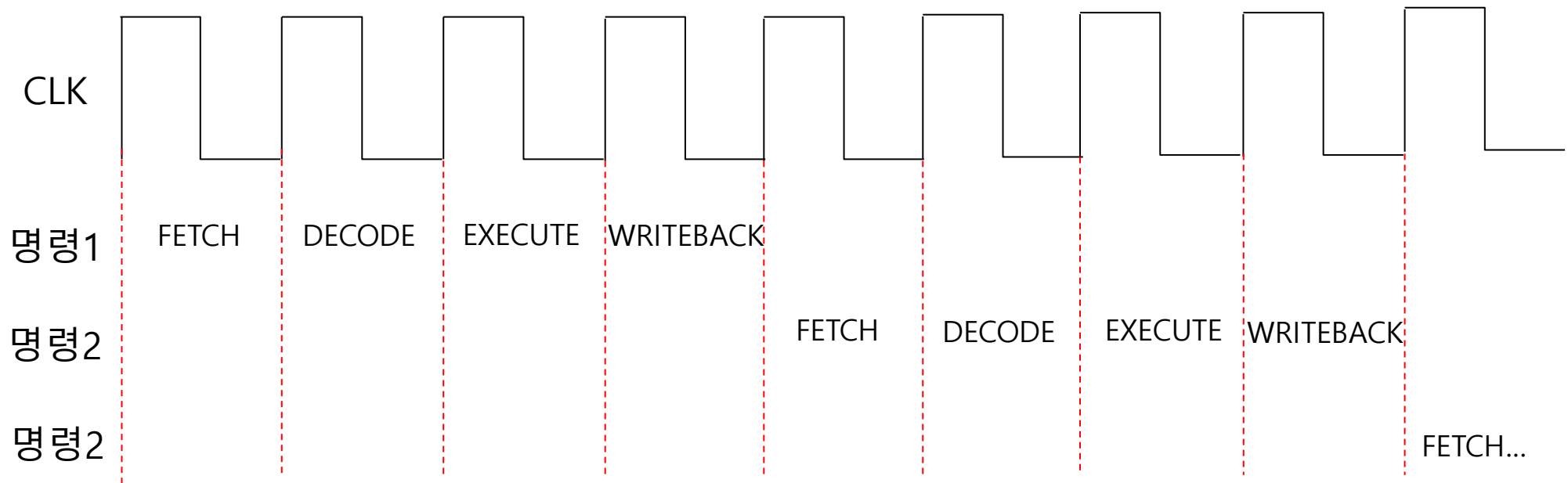


그림1. 파이프라인 없는 명령어 실행 단계

- > CPU는 명령어 처리시 FETCH(명령어인출), DECODE(해석), EXECUTE(실행), WRITEBACK(쓰기) 4단계로 처리한다.
- > CPU의 순차처리방식으로 인해 명령어 3개 처리시 12클럭 소요 (명령어 처리단계는 명령어에 따라 다름)

## 2. 폰 노이만 구조?

### (3) 파이프라이닝 기법이란?

Ex. 명령1 : A+B, 명령2 : C+D, 명령3 : E+F

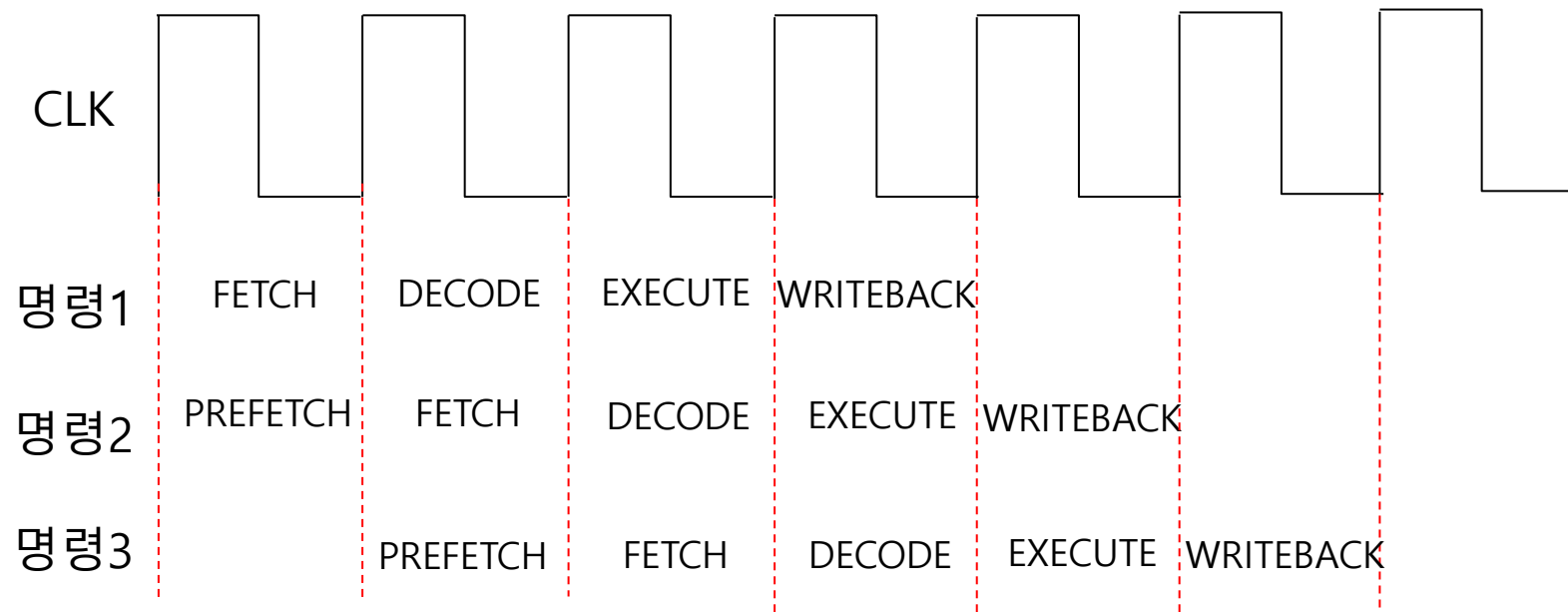


그림1. 파이프라인 적용 명령어 실행 단계

-> 하나의 명령어가 실행되는 중 다음 명령어를 시작하는 식으로 동시에 여러 명령어를 실행하는 기법

-> 6클럭에 3개의 명령어가 끝남(파이프라이닝 없이 하면  $4 \times 3 = 12$ 클럭 소요)

## 2. 폰 노이만 구조?

### (4) 파이프라인 해저드

Ex. 명령1 : A+B, 명령2 : C+D, 명령3 : E+F

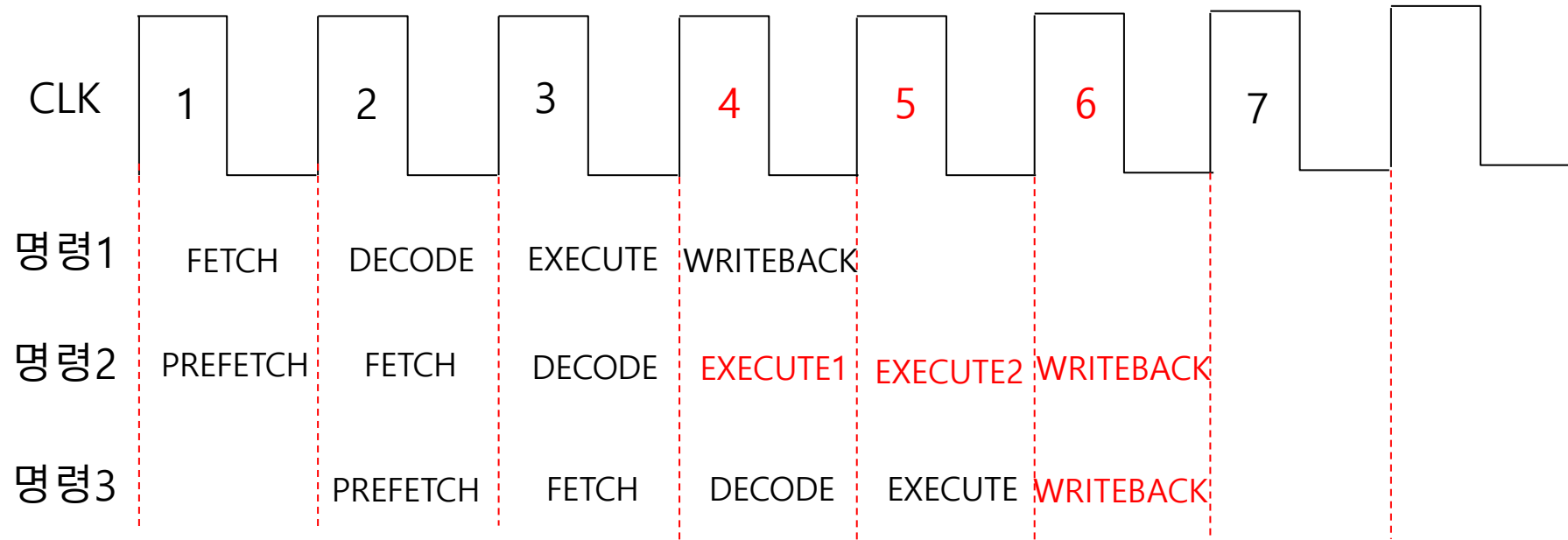


그림1. 파이프라인 적용 명령어 실행 단계

- > 명령2의 실행이 한 클럭에 완료되지 않으면 6번째 클럭에서 명령2,3이 동시에 메모리에 접근할 때, **명령3의 실행 지연 발생.**
- > **단일버스를** 사용하는 폰노이만 구조는 데이터를 동시에 읽기/쓰기 불가능 (**파이프라인 구조적 해저드**)
- > 해결 : **하버드구조(데이터 메모리와 명령어 메모리를 분리)**

## 2. 폰 노이만 구조?

(3) 그 외 파이프라인 해저드

- 데이터 해저드 : 이전 명령어의 연산결과가 다음 명령어의 입력으로 사용되는 경우 발생

Ex) 명령어1 :  $C = A + B$ , 명령어2 :  $E = C + D$

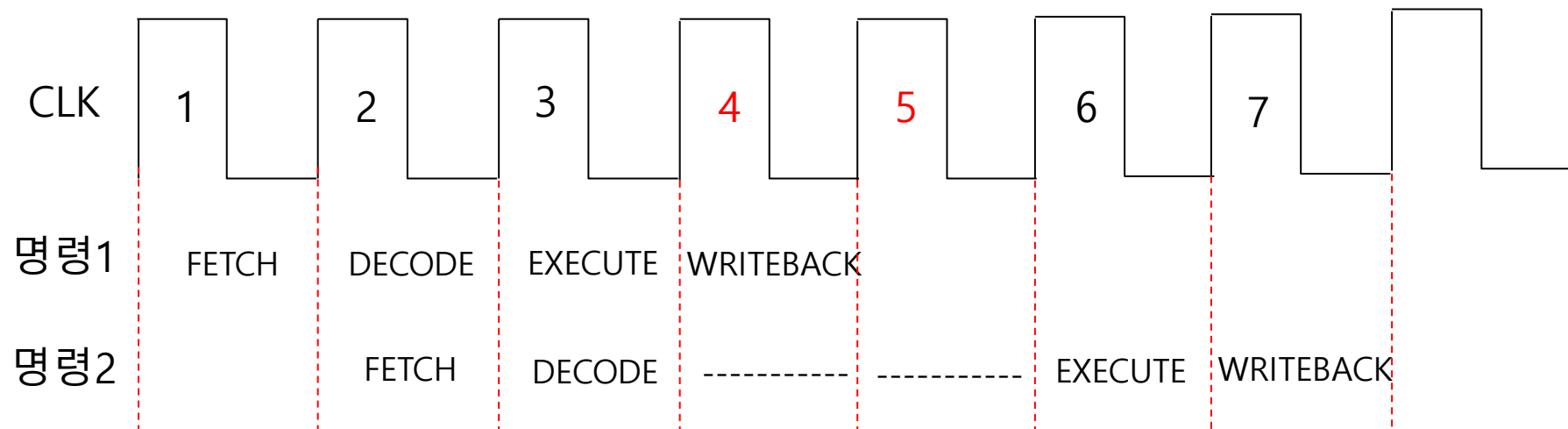


그림1. 데이터 해저드

- 명령어1의 결과는 4클럭에서 완료가 된다.
- 명령어1의 연산이 완료될 때 까지 명령어2의 실행이 지연된다.

## 2. 폰 노이만 구조?

- 제어 해저드 : If-else문과 같은 조건 분기 명령어(Branch)를 사용할 경우 발생가능



그림1. 제어 해저드

- > if-else문 사용시 조건이 성립/불성립에 따라 if문 또는 else문의 주소로 JUMP를 하게 됨.
- > 점프할 주소가 확정되기 전까지 다음 명령어의 처리가 지연된다.

\*그림출처 : <https://ezbeat.tistory.com/454>

### 3. 하버드 구조?

- (1) 하버드구조 : 데이터 메모리와 명령어 메모리가 **물리적으로 분리**되어 있음  
데이터버스와 명령어 버스가 **각각** 구성됨
- (2) 하버드 구조 단점 : 버스구조가 복잡하여 설계가 어려움(고비용)

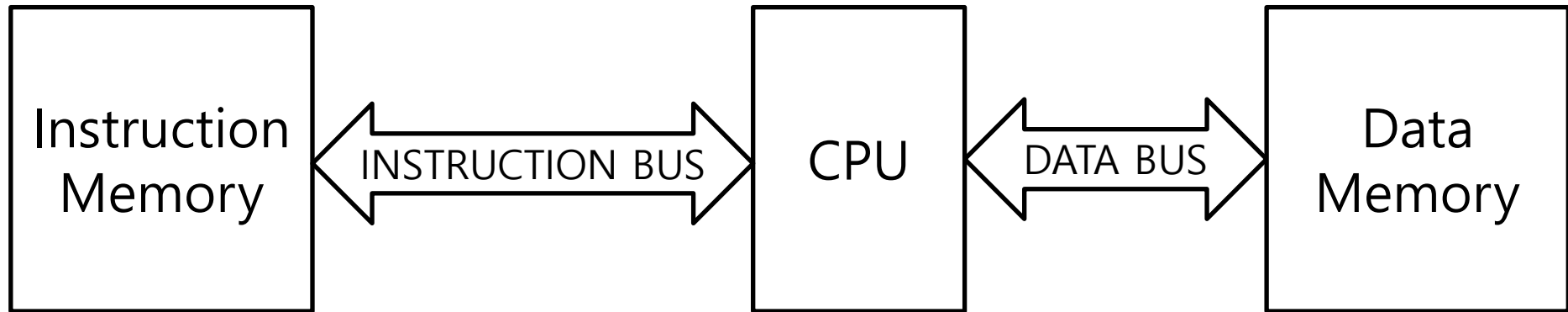


그림1. 하버드 구조

- (3) 데이터 버스와 명령어 메모리가 분리되어 있어 파이프라이닝 기법 활용에 최적화  
(데이터를 동시에 읽기/쓰기가 가능!!)
- (4) **RISC**와 같이 축소된 명령어 구조에 적합하다.



## 4. RISC vs CISC

- (1) RISC? Reduced Instruction Set Computer로 말그대로 명령어를 축소시킨 컴퓨터다.
  - CPU의 메모리 접근을 위해 **LOAD/STORE** 명령어를 사용한다.
  - 축약된 명령어 체계로 1클럭에 명령어 1개가 수행된다.
  - 명령어의 길이가 **고정적**
- (2) 반대로 CISC는 복잡한 명령어체계를 가진 컴퓨터다.
  - RISC의 명령어와 중복되는 부분이 많지만 내포하는 의미는 더 많다
  - CPU의 메모리 접근을 위해 **MOVE** 명령어를 사용한다.
  - 명령어의 길이가 **가변적**
- (3) CISC vs RISC

\*MOVE명령어 구성 : MOVE DESTINATION(register or address), SOURCE(data or register, address)  
EX)  $A = B + C$

-> RISC 명령어 구조에서  
Line1 LOAD R1, B  
Line2 LOAD R2, C  
Line3 ADD R3, R1, R2  
Line4 STORE R3, A

-> CISC 명령어 구조에서  
Line1 MOVE A, B  
Line2 ADD A, C

- (4) RISC에서 4번의 명령어가 필요하지만 CISC는 2번의 명령어만에 결과값이 도출된다.  
-> **CISC의 ADD명령에 STORE까지 포함된다...**

## 4. RISC vs CISC

---

### (5) CISC의 장점

- 코드의 사이즈가 작다(메모리 공간 활용에 용이)
- 메모리 접근 횟수가 적다(메모리 접근 속도가 빠르다)

### (6) CISC의 단점

- 코드사이즈가 가변적으로 파이프라인 기법에 불리하다(이전 병목현상 참고)
- 복잡한 명령어 구현을 위해 하드웨어적으로도 구성이 복잡(비용 증가)
- 오늘날 집적 기술의 향상으로 메모리 용량도 늘어났다(메모리 공간 활용에 큰 이점x)
- 명령어의 길이가 가변적이라 명령어 처리에 많은 클럭이 소모된다

### (7) RISC의 장점

- 명령어가 세분화된 과정으로 나뉘어져 파이프라인 기법에 유리하다.
- 하드웨어 구성에 있어 단순하다(저비용)
- 명령어의 길이가 고정되어 있어 한 클럭에 하나의 명령어가 수행된다.

### (8) RISC의 단점

- 코드의 사이즈가 크다(메모리 공간 활용에 CISC에 비해 불리)
- 메모리 접근 횟수가 크다(메모리 접근 속도가 느림)

## 5. 변수

- (1) 변수란? 특정 데이터가 저장되지 않은 메모리 영역의 지정된 이름!
- (2) 변수는 문자 또는 숫자로 구성 될 수 있다(변수명은 반드시 문자나 밑줄로 시작해야함)

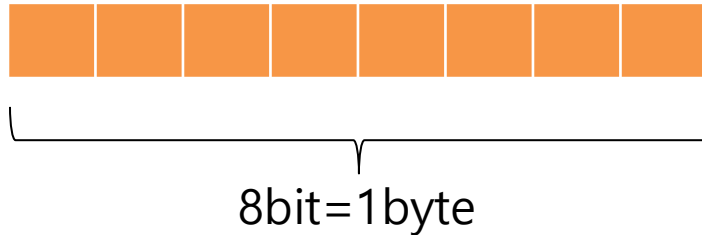
```
//char RefAxis;
//Axis Position = { 0, };
//double Angle=0.0F;
float Matrix[MatrixSize][MatrixSize] = { 0, };
//float DeterminantResult;
float Value[3] = { 0, };

float yn=0;
static float yn_1=0;
const float LPF_CONST1 = 0.083026;
const float LPF_CONST2 = 0.916974;
```

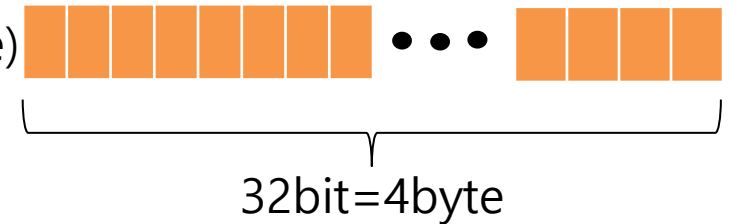
## 5. 변수

- (3) C의 변수에는 특정 유형(Data Type)이 존재, 메모리 크기와 레이아웃(메모리계층)을 결정  
※ int크기는 CPU마다 다름!

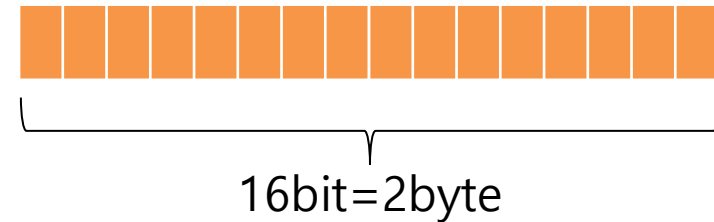
char형



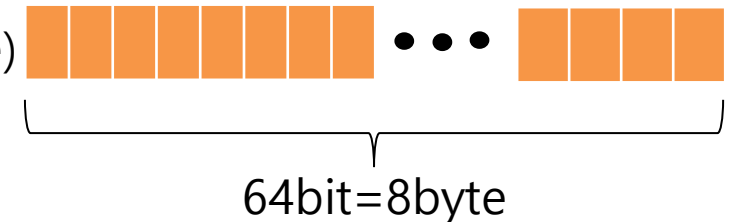
float형(4byte)



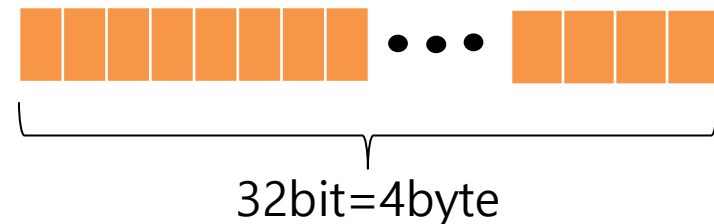
int형(2byte)



double형(8byte)



int형(4byte)



void형(?byte)

아무런 Type도아님  
(크기도 정해지지 않음)

## 5. 변수

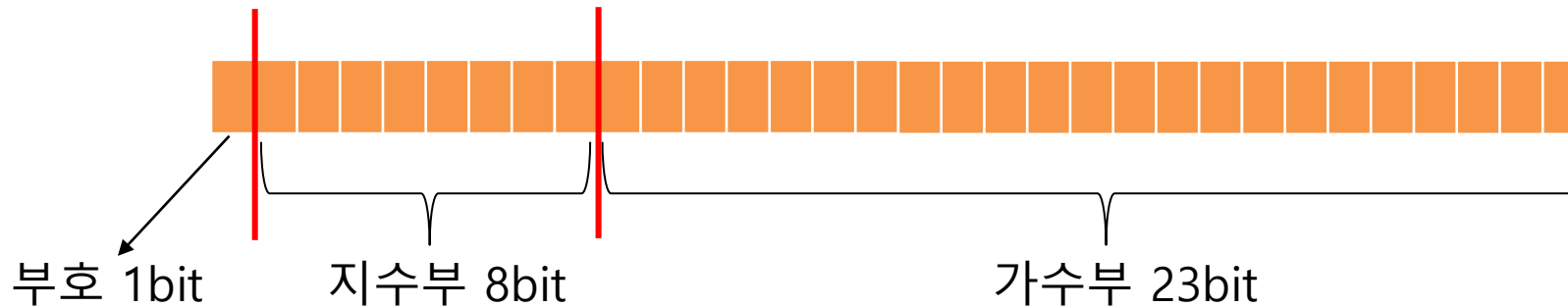
(4) float(single-precision), double(double-precision)이다.

single-precision : 단정밀도 32bit

double-precision : 배정밀도 64bit(단정밀도의 두 배)

(5) 둘 다 실수를 표현함.

(6) 자세한 실수표현 계산법은 <https://whatisthenext.tistory.com/146> 참조



## 6. 데이터 유형(Data Type)

- 데이터 유형에 따라 메모리에 차지하는 공간과 저장된 비트 방식이 다르다  
(signed or unsigned)
- unsigned가 없으면 signed

유형	크기	범위
char	1byte	-126 ~ 127
unsigned char	1byte	0 ~ 255
int	1byte	-32,768 ~ 32,767 or -2,147,483,648 ~ 2,147,483,647
unsigned int	2 or 4byte	0 ~ 65,535 or 0 ~ 4,294,967,295
short	2 or 4byte	-32,768 ~ 32,767
unsigned short	2byte	0 ~ 65,535
long	2byte	-9,223,372,036,854,775,808 ~ 9,223,372,036,854,775,807
unsigned long	8byte	0 ~ 16,446,744,073,709,551,615
Float	4byte	1.2E-38 ~ 3.4E+38
double	4byte	2.3E-308 ~ 1.7E+308
long double	8byte	3.4E-4932 ~ 1.1E+4932

# 7. 데이터 유형(Data Type)

## - Signed 데이터형과 Unsigned 데이터형의 범위를 초과하면?

```
char a = 127;
unsigned char b = 255;

printf("unsigned char = %d\n", a);
printf("signed char = %d\n", b);

a = a + 1;
b = b + 1;

printf("!!!!Over Flow!!!!\n");
printf("unsigned char = %d\n", a);
printf("signed char = %d\n", b);

a = -129;
b = -1;

printf("!!!!Under Flow!!!!\n");
printf("unsigned char = %d\n", a);
printf("signed char = %d\n", b);

return 0;
```

```
unsigned char = 127
signed char = 255
!!!!Over Flow!!!!
unsigned char = -128
signed char = 0
!!!!Under Flow!!!!
unsigned char = 127
signed char = 255

C:\Users\SON\source\repos\Test
이 창을 닫으려면 아무 키나 누르
```

- 1) Overflow :  $127 = 0111\ 1111$ ,  
 $0111\ 1111 + 1 \rightarrow 1000\ 0000$ 이며,  
앞의 부호비트가 1이며, decimal로  
바꾸면 -128이 된다.
- 2) Underflow :  $-128 = 1000\ 0000$   
 $1000\ 0000 - 1 \rightarrow 0111\ 1111$ 이며,  
앞의 부호비트가 0이고, decimal로  
바꾸면 127이 된다.

unsigned형도 마찬가지로이다.

## 8. 데이터 유형 변환(Type Casting)

- (1) Type Casting은 데이터 타입을 다른 데이터 타입으로 바꾸는 것  
- 표현하고자 하는 자료의 크기를 확장 또는 축소 시 사용한다

(1) 아래 그림1과 같이 변수명 앞에 "(원하는 데이터 유형)"를 붙여 사용

(2) Casting 지시를 하지 않으면 그림2처럼 컴파일러는 범위가 큰 자료형으로 강제로 Casting한다

```
#include <stdio.h>

int main(void)
{
    char a = 127;

    printf("a = %f\n", (float)a);

    return 0;
}
```

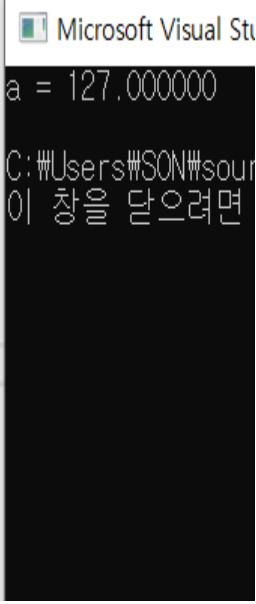


그림1

```
#include <stdio.h>

int main(void)
{
    char a = 127;
    float b = a;

    printf("a = %d\n", a);
    printf("b = %f\n", b);

    return 0;
}
```

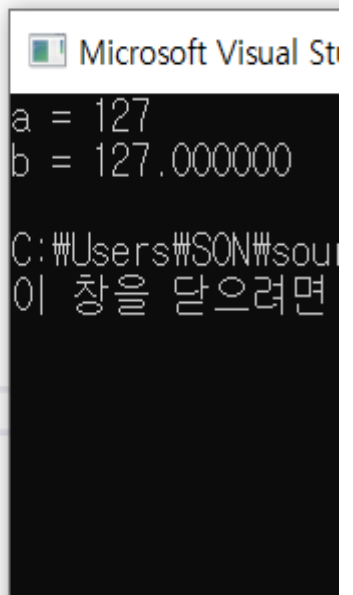


그림2