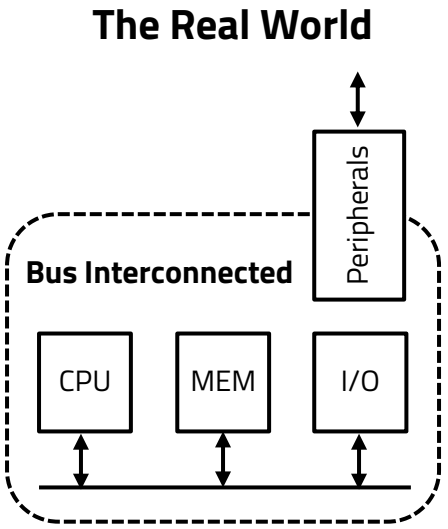
# 시스템 프로그래밍을 위한 C언어

## - Code Memory에 명령어 (컴파일된 기계코드) 배치 (Layout) 및 Data  Memory에 변수 할당 (Allocation)

현대자동차 입문교육
박대진 교수

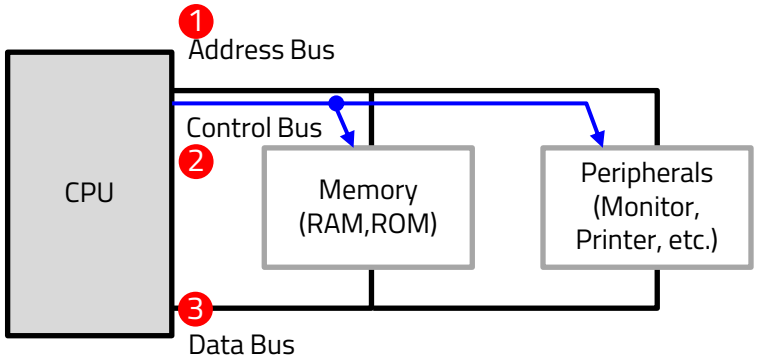
# uP-based System has three parts

- **Central Processing Unit (CPU)**
  - Same to uP in PC domain
- **Memory**
  - Storage for Program (Code, Instructions)
  - Buffer for Data (Stack, Heap, Constant)
- **Input/Output (I/O) & Peripherals Devices**
  - Provides data to CPU from outside world
  - Generates meaningful data



# Bus-based Communications

- **Connection between blocks.**
  - The bus inside a system carries information from place to place
  - 1) Address Bus
    - Is used to identify the memory location
  - 2) Control Bus
    - is used to tell what type of command is, where to write/read,
      - Specifically, Memory Read/Write, Peripherals(I/O포함) Read/Write
  - 3) Data Bus
    - Is used to by CPU to get data from / to send data to I/O devices



# Memory

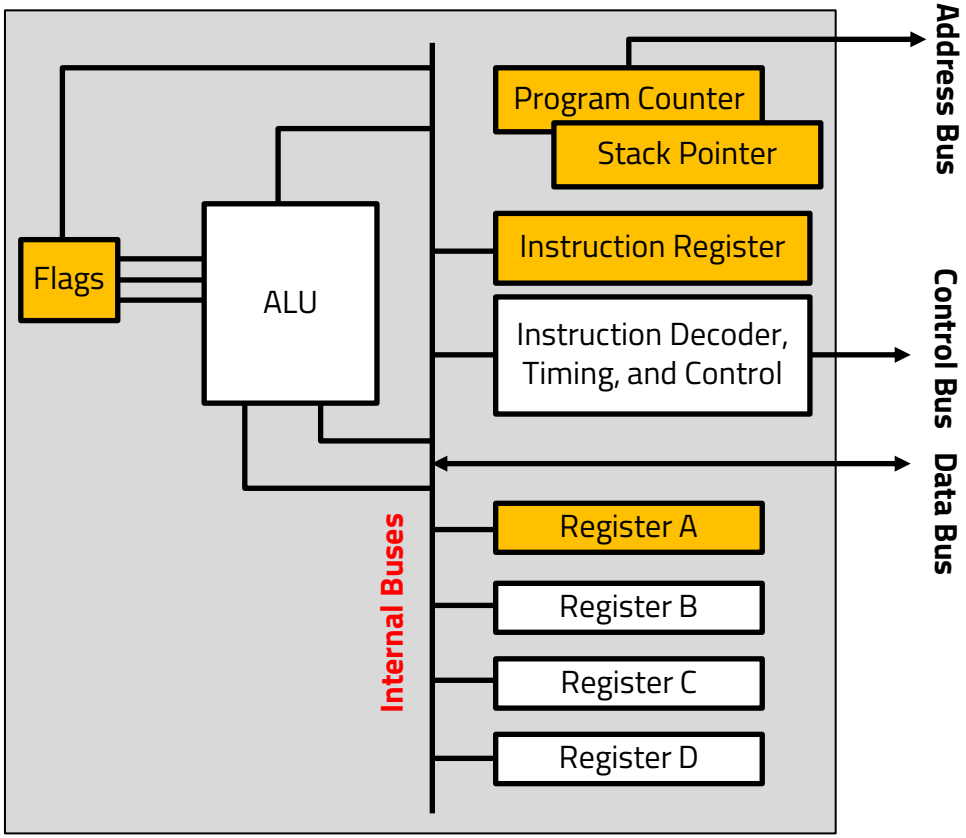
- Volatile
  - DRAM
  - SRAM
  - Register (F/F)
- Non-volatile (ROM, but, programmable)
  - SSD (Stacks of Flash)
  - Flash
  - EEPROM

**단기 기억**
**Ex) 100+134+152**
**= 234+152**
**= 386**

**장기 기억**
**Ex) 미분방정식 풀이 절차**

# 5 Important Registers in CPU

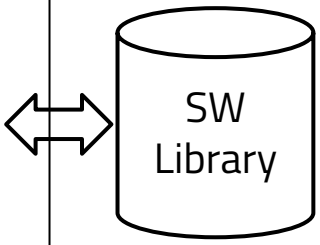
- **Program Counter (PC)**
- **Stack Pointers (SP)**
- **Instruction Register (IR)**
- **Registers (A,B, ….)**
  - Accumulator
  - Operands
- **Flag Register**



# ISA (Register로 인코딩): Hardware API

- **Hardware에 접근하기 위한 인터페이스 (도구)**
  - ISA에 정의된 명령어를 이용하여 Register에 값을 쓰면, 하드웨어의 동작이 바뀐다
  - Register의 값을 읽으면, 하드웨어의 상태를 파악할 수 있다.

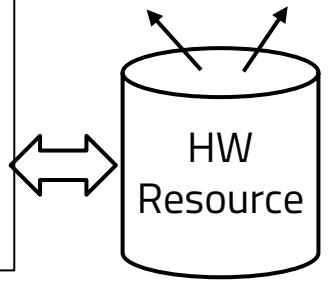
```
#include <stdio.h>
#include <list.h>
int main() {
        ...
    sort(...);
    add_list(...);
        printf(..)
    ...
}
```




**SW라이브러리에서 제공되는 함수(API)를 이용하여 SW 동작을 제어한다.**

SW 관점의 API : Functions

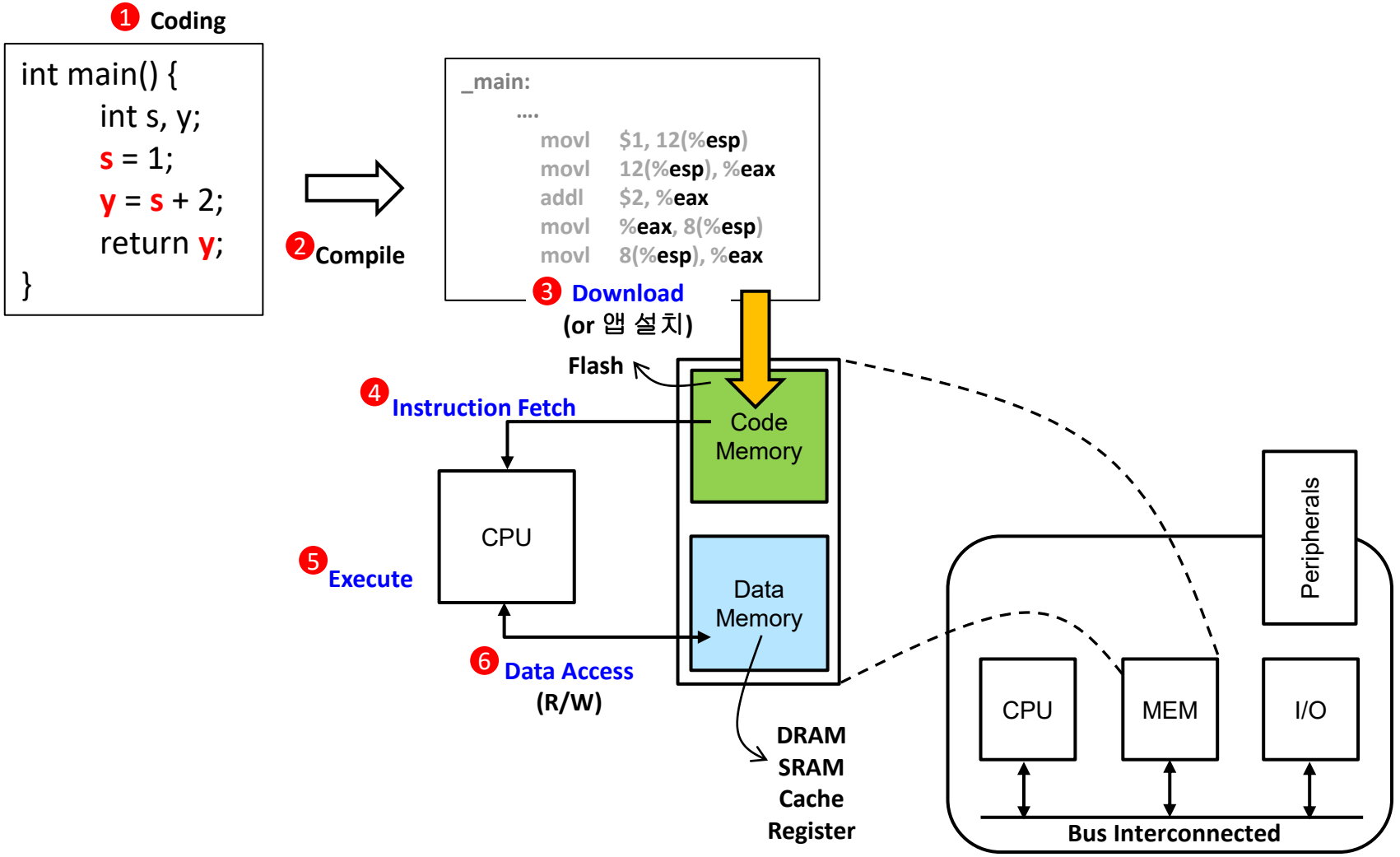
```
_main:
    ....
        movl   $1, 12(%esp)
        movl   12(%esp), %eax
        add    $2, %eax
        movl   %eax, 8(%esp)
        movl   8(%esp), %eax
```



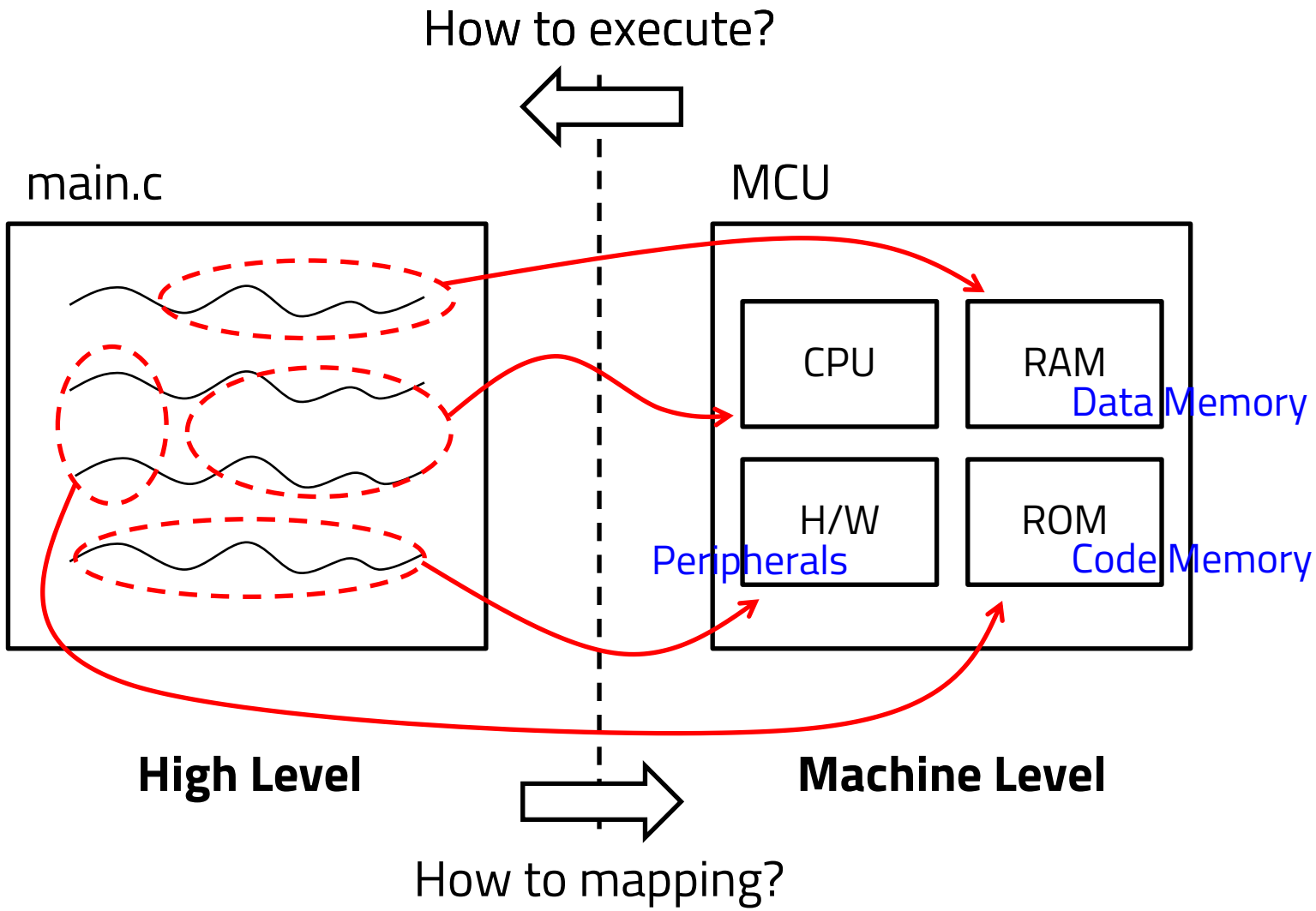

**Target HW내부에서 제공되는 Registers에 접근하여 HW 동작을 제어한다**

HW 관점의 API : Registers

# 코드와 데이터 (변수)


1 Coding
int main() {
int s, y;
s = 1;
y = s + 2;
return y;
}
2 Compile
_main:
....
movl $1, 12(%esp)
movl 12(%esp), %eax
addl $2, %eax
movl %eax, 8(%esp)
movl 8(%esp), %eax
3 Download
(or 앱 설치)
Flash
4 Instruction Fetch
Code Memory
CPU
5 Execute
Data Memory
6 Data Access
(R/W)
DRAM
SRAM
Cache
Register
Peripherals
CPU
MEM
I/O
Bus Interconnected

# C코드는 MCU의 각 하드웨어로 매핑된다.



# Data Type별 메모리 차지하는 크기

| 바이트수 | 바이트 | 정수형 | 실수형 |
|---|---|---|---|
| 1 | Byte | char | |
| 2 | Half word | short | |
| | | Int (optional) | |
| 4 | Word | int | float |
| | | long | |
| 8 | Double word | long long | double |
| 16 | Long double word | | long double |

# Summary: Size of Integer

## Signed integer

| Data type | Memory size | Minimum value | Maximum value |
|---|---|---|---|
| char | 8bit (1byte) | $-2^7$=-128 | $2^7$-1=127 |
| short | 16bit (2byte) | $-2^{15}$=-32,768 | $2^{15}$-1=32,767 |
| int | 32bit (4byte)* | $-2^{31}$=-2,147,483,648 (0x80000000) | $2^{31}$-1=2,147,483,647 (0x7fffffff) |
| long | 32bit (4byte) | $-2^{31}$=-2,147,483,648 | $2^{31}$-1=2,147,483,647 |

## Unsigned integer

| Data type | Memory size | Minimum value | Maximum value |
|---|---|---|---|
| unsigned char | 8bit (1byte) | 0 | $2^8$-1=255 |
| unsigned short | 16bit (2byte) | 0 | $2^{16}$-1=65,535 (0xffff) |
| unsigned int | 32bit (4byte) | 0 | $2^{32}$-1=4,294,967,295 (0xffffffff) |
| unsigned long | 32bit (4byte) | 0 | $2^{32}$-1=4,294,967,295 |

# Data Overflow

- Data exceeds the size of its type used in compiler

| Data type | Memory size | Minimum value | Maximum value |
|---|---|---|---|
| int | 32bit (4byte) | $-2^{31}$=-2,147,483,648 (0x80000000) | $2^{31}$-1=2,147,483,647 (0x7fffffff) |
| unsigned int | 32bit (4byte) | 0 | $2^{32}$-1=4,294,967,295 (0xffffffff) |

- Ex.

```
#include <stdio.h>
main( )
{
   int i = 2147483647; // 2^31 – 1
   printf("%d\n%d\n%d\n", i, i+1, i+2);
}
```

```
result>
i:    2147483647  (0  11……..…. 11)
i+1: -2147483648  (1  00…….…..00)
i+2: -2147483647  (1  00…….…..01)
```

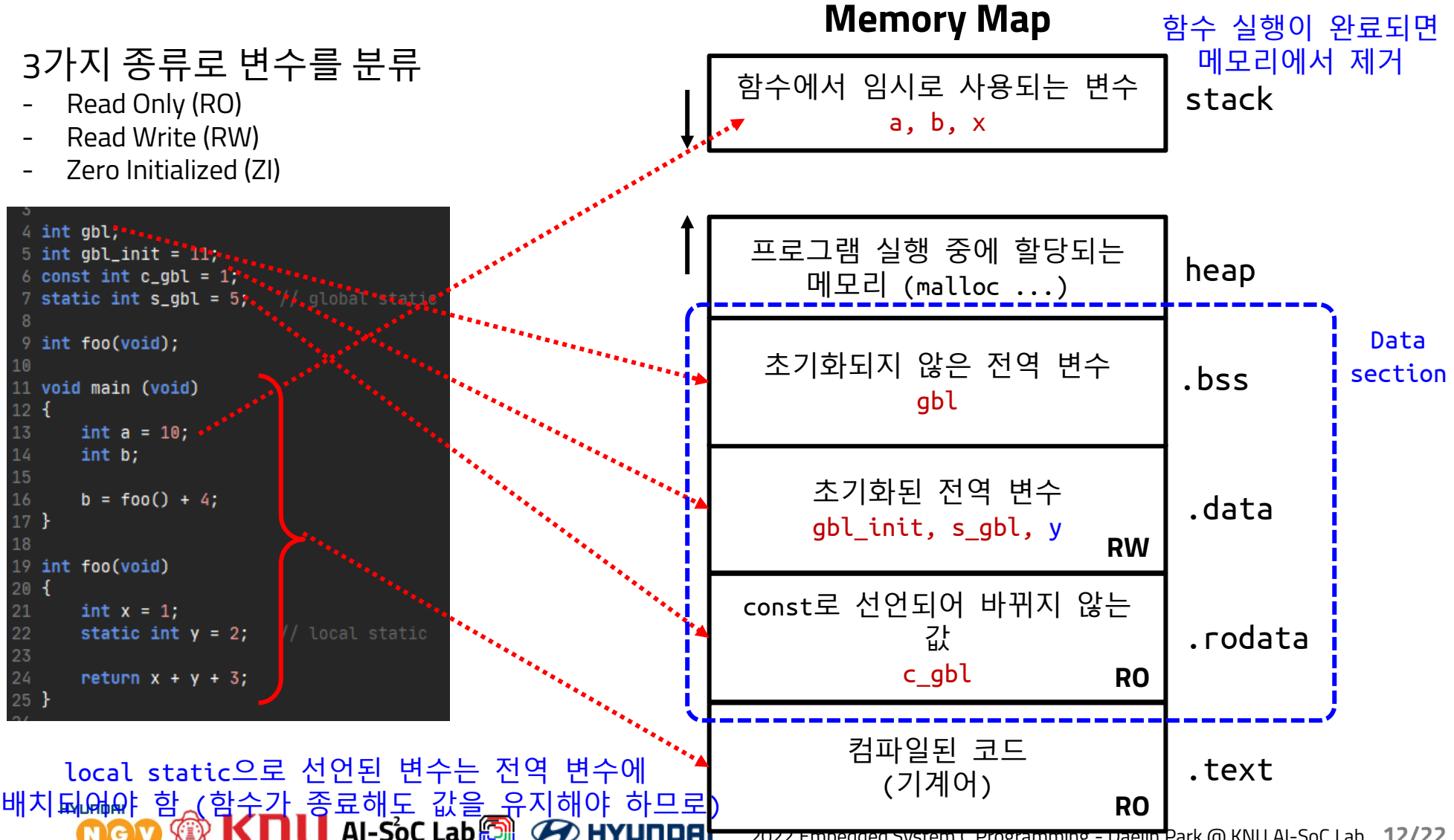
# 변수와 Memory Map

## 컴파일을 거친 변수들이 메모리에 저장되는 규칙

3가지 종류로 변수를 분류

- Read Only (RO)
- Read Write (RW)
- Zero Initialized (ZI)

```
4  int gbl;
5  int gbl_init = 11;
6  const int c_gbl = 1;
7  static int s_gbl = 5;      // global static
8
9  int foo(void);
10
11 void main (void)
12 {
13     int a = 10;
14     int b;
15
16     b = foo() + 4;
17 }
18
19 int foo(void)
20 {
21     int x = 1;
22     static int y = 2;      // local static
23
24     return x + y + 3;
25 }
```

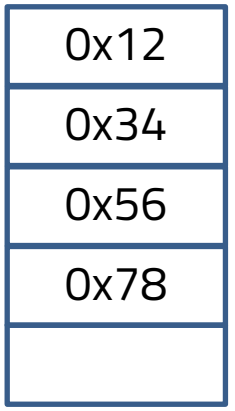



local static으로 선언된 변수는 전역 변수에 배치되어야 함 (함수가 종료해도 값을 유지해야 하므로)

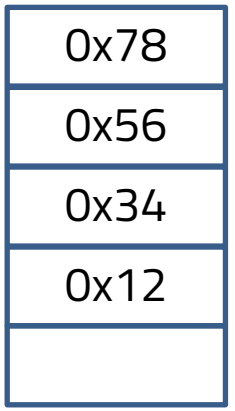
# 멀티바이트 변수 memory allocation

unsigned int x = 0x12345678;

- Big endian



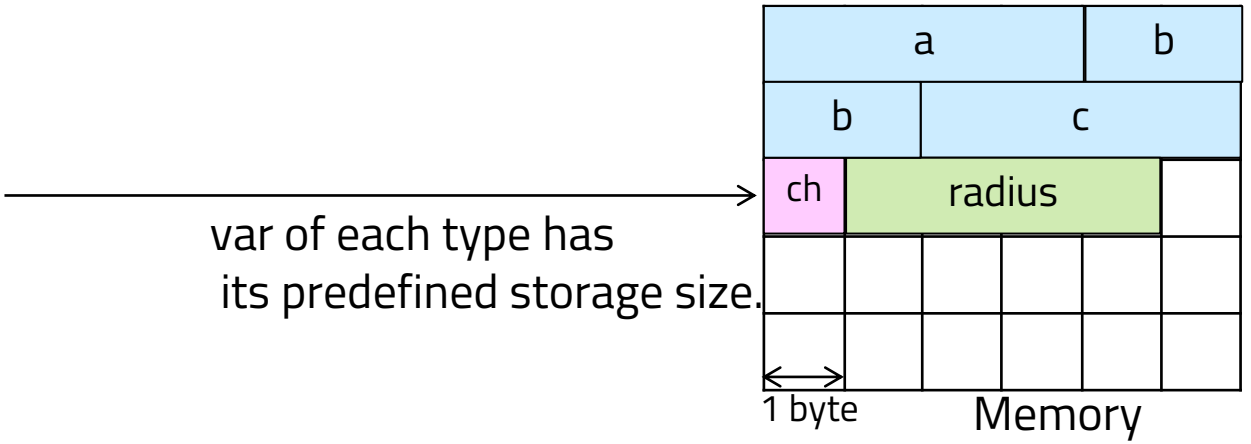

- Little endian



# Memory Allocation:
# Compiler does, vs Programmer does.

- Two Ways to manage life time of object storage
  - Complier manages extent based on storage class specifier in declaration.

```
int a, b, c;
char ch;
float radius;
```




  - Dynamic allocation and de-allocation by programmers
    - Use of special library routines such as malloc() and free()

# 구조체 memory allocation

```
struct test {
    int a;      ← 0x12345678
    short b;    ← 0xFBCD
    char c;     ← 0xEE
    int d;      ← 0x12345678
    char e;     ← 0xAB
    char f;     ← 0xCC
    char g;     ← 0xEE
    short h;    ← 0xDDFF
};
```

Aligned access

| 3 | 2 | 1 | 0 | |
|---|---|---|---|---|
| 0x12 | 0x34 | 0x56 | 0x78 | 0x20000000 |
| | 0xEE | 0xFB | 0xCD | 0x20000004 |
| 0x12 | 0x34 | 0x56 | 0x78 | 0x20000008 |
| | 0xEE | 0xCC | 0xAB | 0x2000000C |
| | | 0xDD | 0xFF | 0x20000010 |

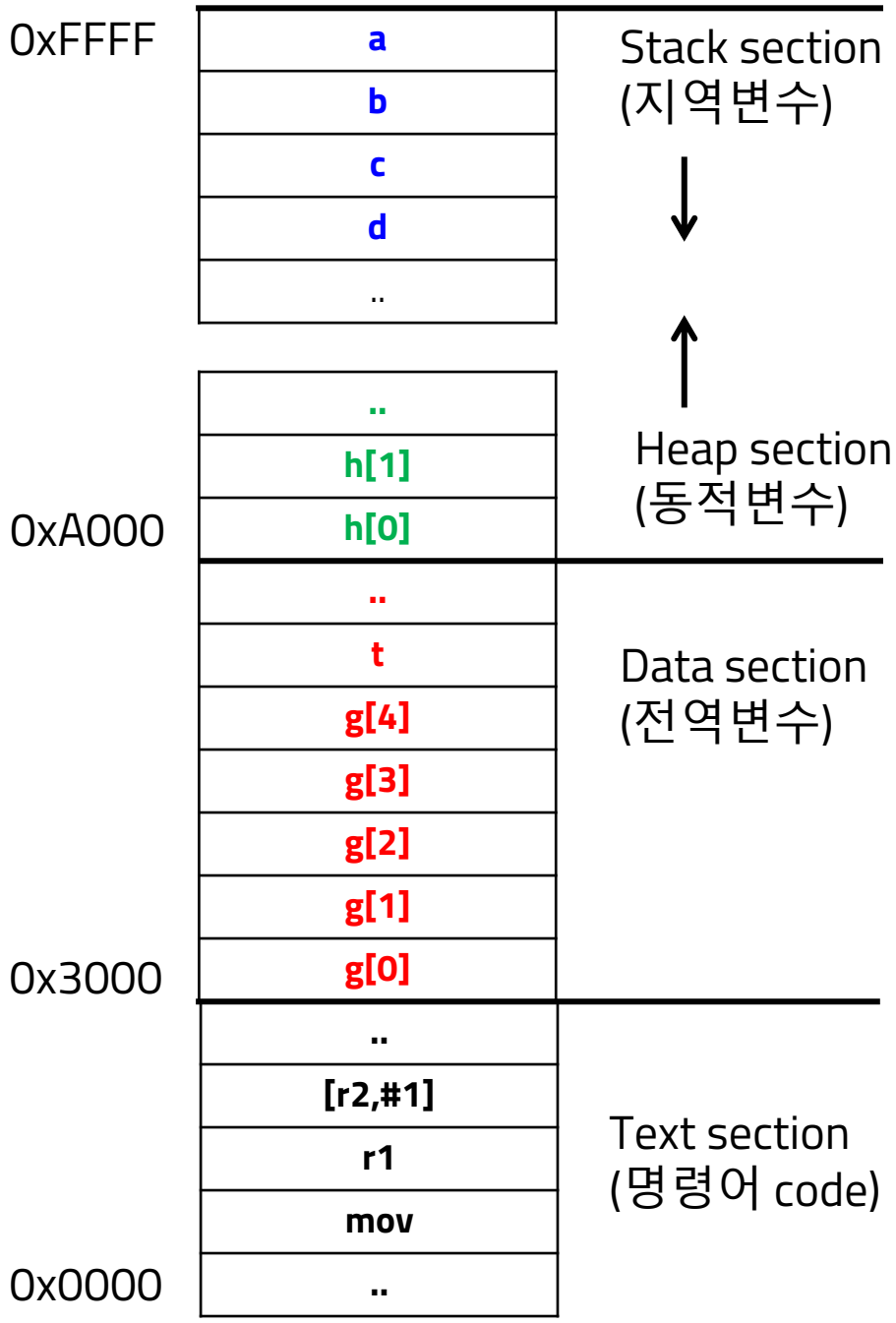
Unaligned access (packed)

| 3 | 2 | 1 | 0 | |
|---|---|---|---|---|
| 0x12 | 0x34 | 0x56 | 0x78 | 0x20000000 |
| 0x78 | 0xEE | 0xFB | 0xCD | 0x20000004 |
| 0xAB | 0x12 | 0x34 | 0x56 | 0x20000008 |
| 0xDD | 0xFF | 0xEE | 0xCC | 0x2000000C |
| | | | | 0x20000010 |

# Embedded F/W의 RAM/ROM Allocation

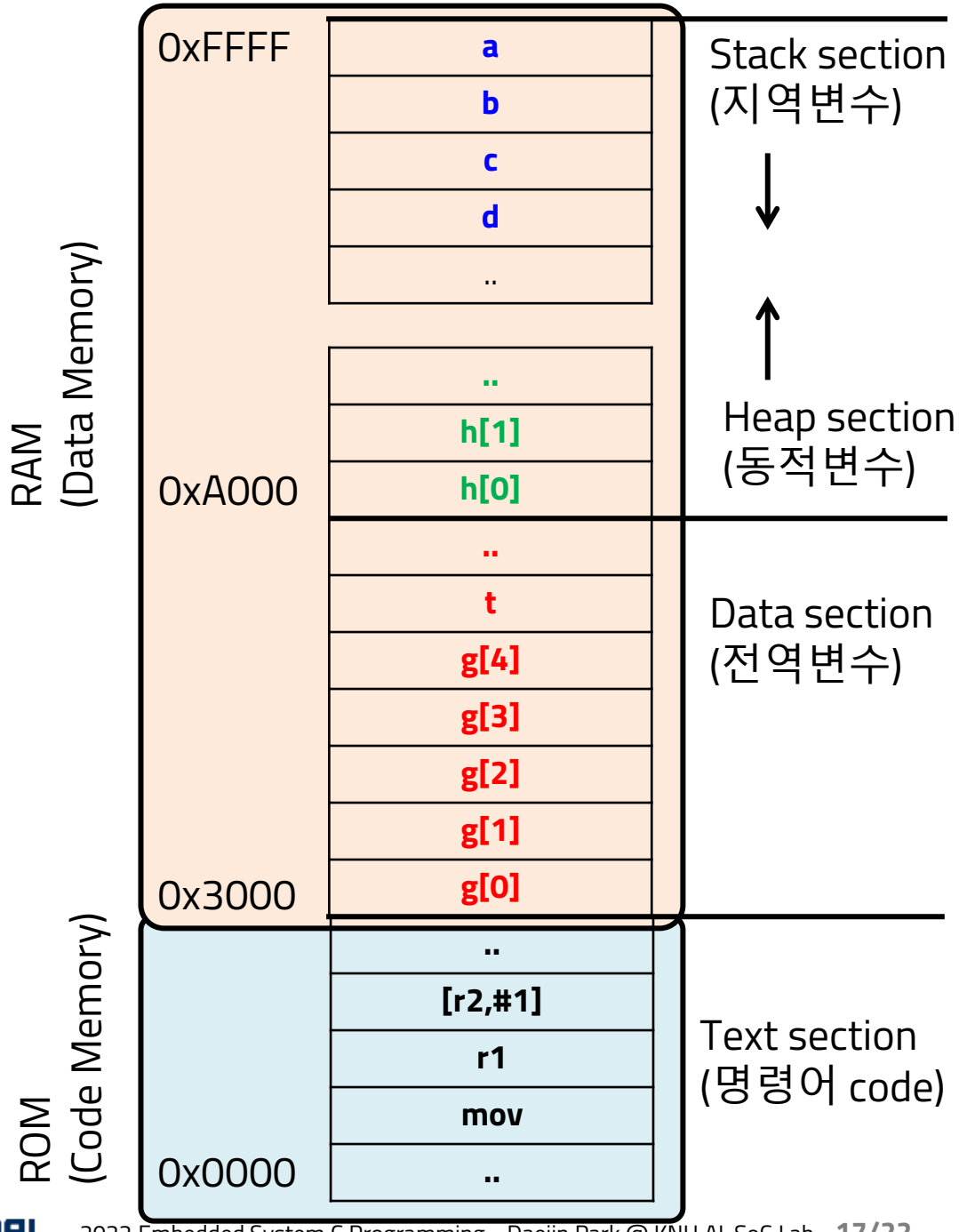
## : Memory Layout

```
char g[5] = {1,2,3,4,5};
Int k;
void main() {
    int a, b, c, d;
    static char t = 7;

    b = 10;
    c = 20;
    a = b + c;
    char* h = (char*)malloc(2);
    h[0] = 7;
    h[1] = 8;

    d = g[2] + a;
    d = d + h[1];

}
```



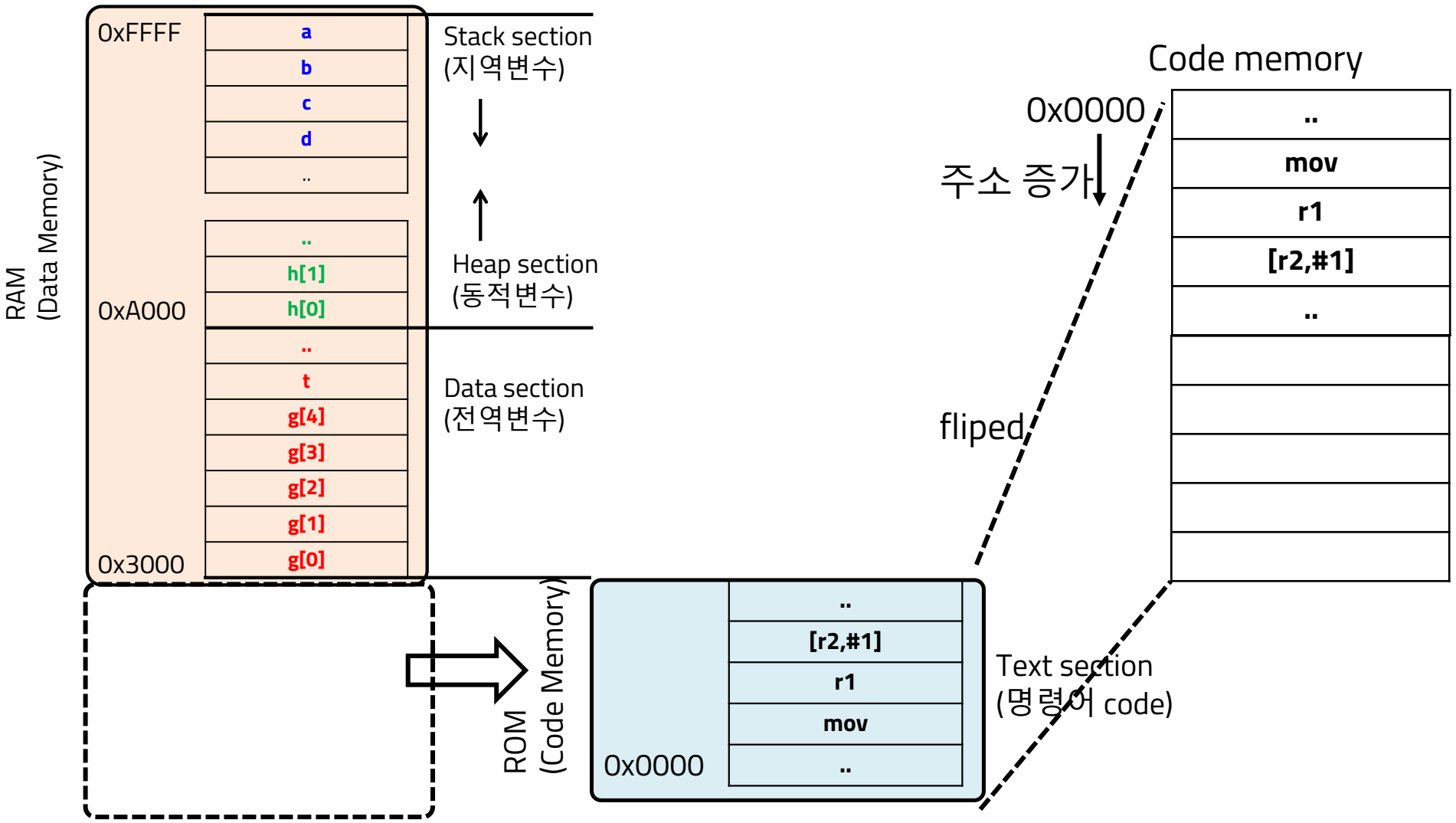
# Embedded F/W의 RAM/ROM Allocation

```
char g[5] = {1,2,3,4,5};
Int k;
void main() {
    int a, b, c, d;
    static char t = 7;

    b = 10;
    c = 20;
    a = b + c;
    char* h = (char*)malloc(2);
    h[0] = 7;
    h[1] = 8;

    d = g[2] + a;
    d = d + h[1];

}
```



# Embedded F/W의 RAM/ROM Allocation

## Harvard Architecture – Data/Code Memory분리 (동시 접근 가능)

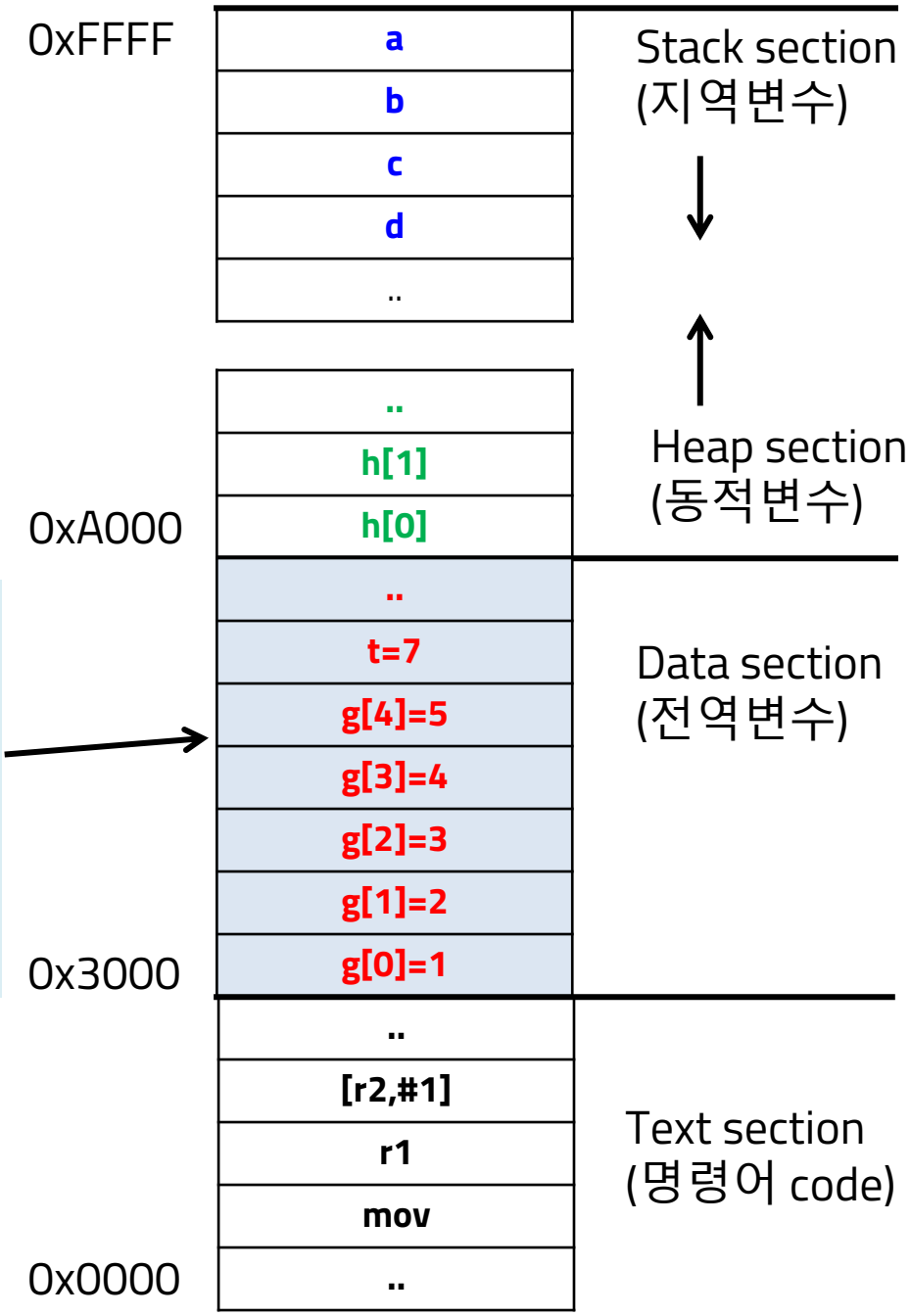


# Embedded F/W의 RAM/ROM Allocation

## : Data Section 초기화 (전역변수)

```
char g[5] = {1,2,3,4,5};
Int k;
void main() {
    int a, b, c, d;
    static char t = 7;

    b = 10;
    c = 20;
    a = b + c;
    char* h = (char*)malloc(2);
    h[0] = 7;
    h[1] = 8;

    d = g[2] + a;
    d = d + h[1];

}
```

```
mov r0, #3000
// start of data section
mov [r0,#0], #1 // g[0]
mov [r0,#1], #2 // g[1]
mov [r0,#2], #3 // g[2]
mov [r0,#3], #4 // g[3]
mov [r0,#4], #5 // g[4]
mov [r0,#5], #7 // t
```

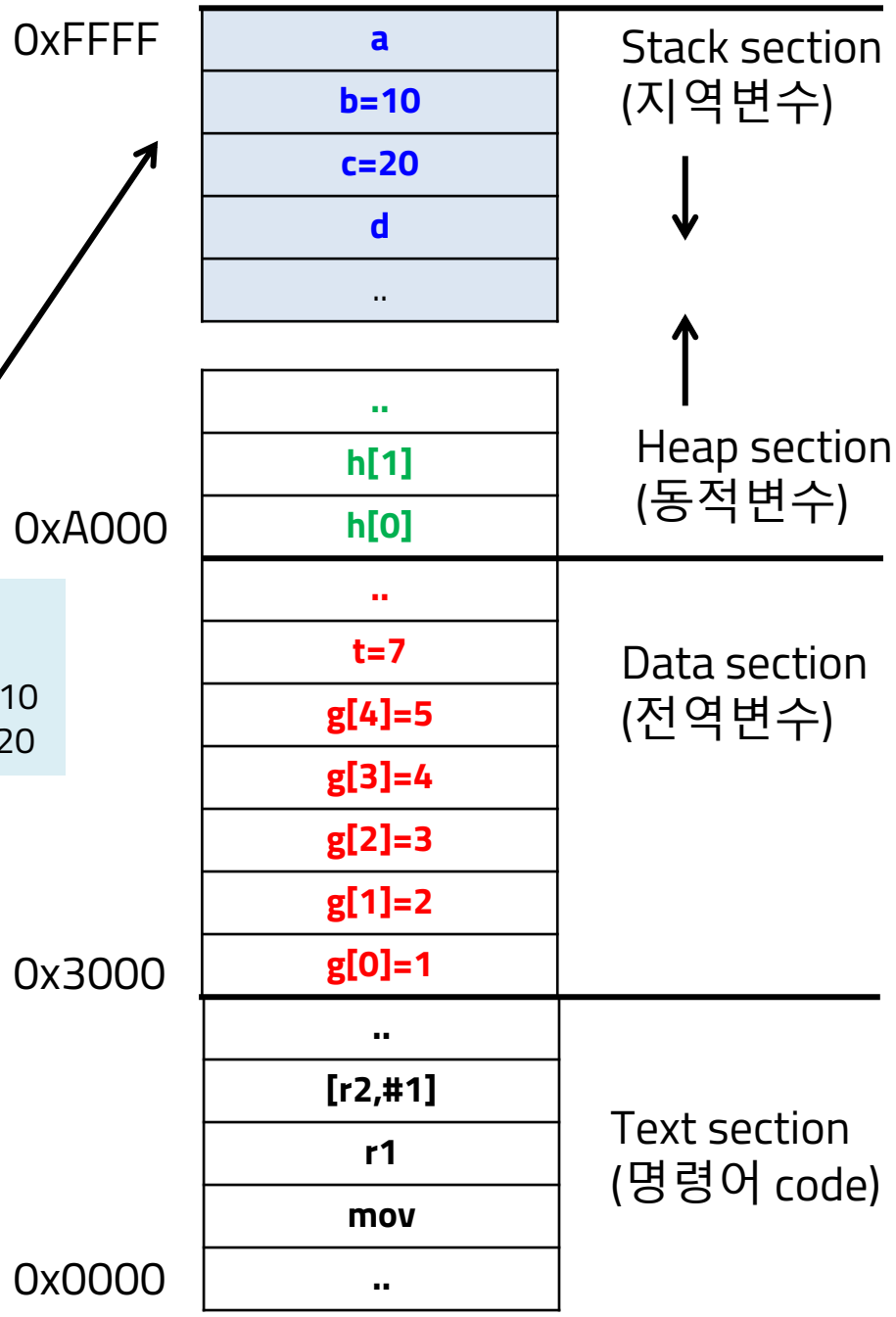


# Embedded F/W의 RAM/ROM Allocation

## : Stack Section 초기화 (지역변수)

```
char g[5] = {1,2,3,4,5};
Int k;
void main() {
    int a, b, c, d;
    static char t = 7;

    b = 10;
    c = 20;
    a = b + c;
    char* h = (char*)malloc(2);
    h[0] = 7;
    h[1] = 8;

    d = g[2] + a;
    d = d + h[1];

}
```

```
mov sp, #FFFF
// start of stack section
 mov [sp,#-4], #10 // b = 10
 mov [sp,#-8], #20 // c = 20
```

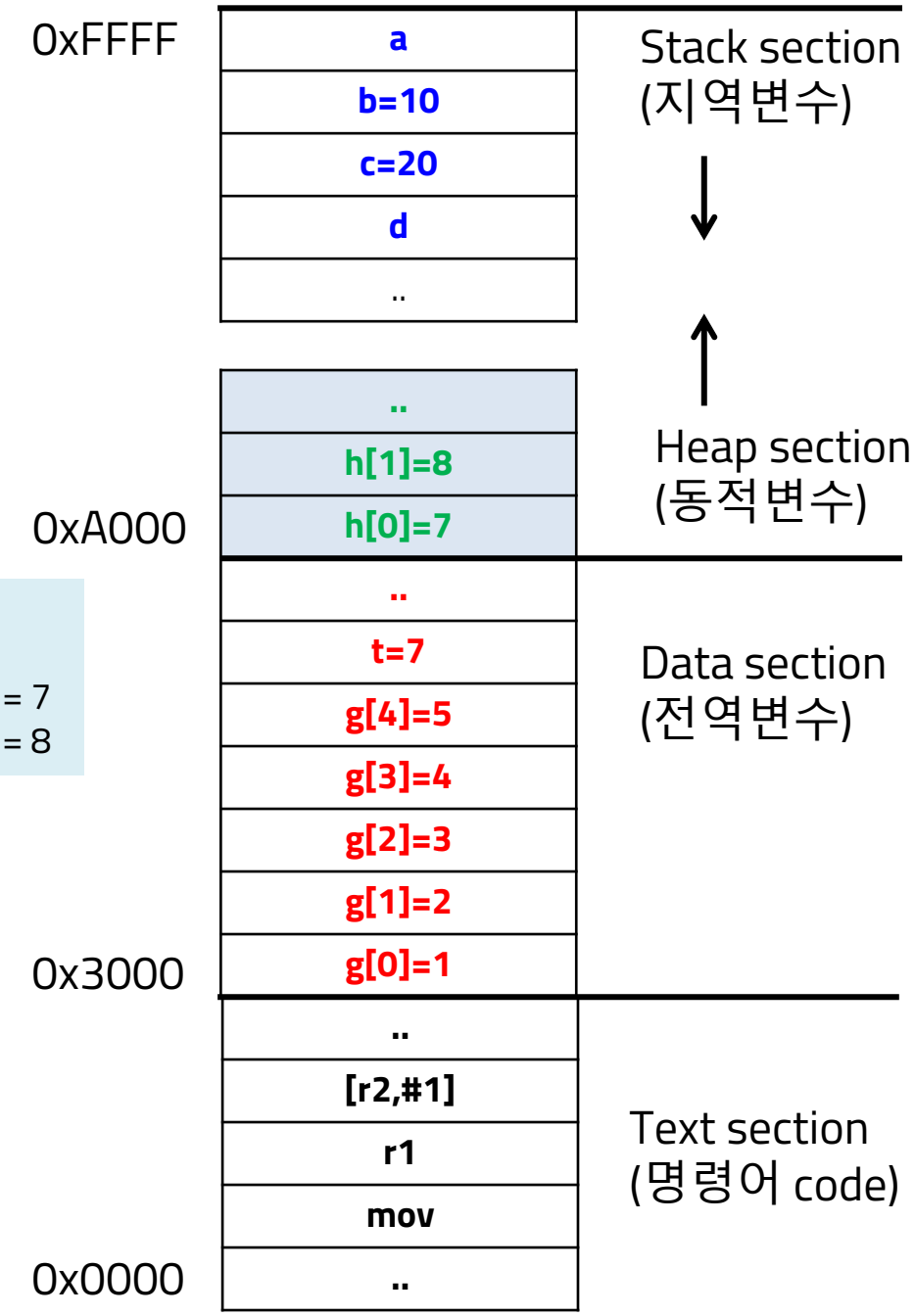


# Embedded F/W의 RAM/ROM Allocation

## : Heap Section 초기화 (동적변수)

```
char g[5] = {1,2,3,4,5};
Int k;
void main() {
    int a, b, c, d;
    static char t = 7;

    b = 10;
    c = 20;
    a = b + c;
    char* h = (char*)malloc(2);
    h[0] = 7;
    h[1] = 8;

    d = g[2] + a;
    d = d + h[1];

}
```

```
mov r7, #A000
// start of heap section
mov [r7,#0], #7  // h[0] = 7
mov [r7,#1], #8  // h[1] = 8
```



# Embedded F/W의 RAM/ROM Allocation

## : 연산처리 (Code)

```
char g[5] = {1,2,3,4,5};
Int k;
void main() {
    int a, b, c, d;
    static char t = 7;

    b = 10;
    c = 20;
    a = b + c;
    char* h = (char*)malloc(2);
    h[0] = 7;
    h[1] = 8;

    d = g[2] + a;
    d = d + h[1];

}
```

```
reset:
  mov r0, #3000 // start of data section
  mov [r0,#0], #1 // g[0]
  mov [r0,#1], #2 // g[1]
  mov [r0,#2], #3 // g[2]
  mov [r0,#3], #4 // g[3]
  mov [r0,#4], #5 // g[4]
  mov [r0,#5], #7 // t
  mov pc, @main // call main

main:
  mov sp, #FFFF // start of stack section
  mov [sp,#-4], #10 // b = 10
  mov [sp,#-8], #20 // c = 20
  mov r1, [sp,#-4]    // load b
  mov r2, [sp,#-8]    // load c
  add r3, r1, r2      // b + c
  mov [sp,#0], r3     // a = b + c

  mov r7, #A000 // start of heap section
  mov [r7,#0], #7  // h[0] = 7
  mov [r7,#1], #8  // h[1] = 8

  mov r1, [r0,#2]   // load g[2]
  mov r2, [sp,#0]   // load a
  add r1, r1, r2    // g[2] + a
  mov r3, [r7,#1]   // load h[1]
  add r1, r1, r3    // d + h[1]
  mov [sp,#-12], r1  // d = d+h[1]
```