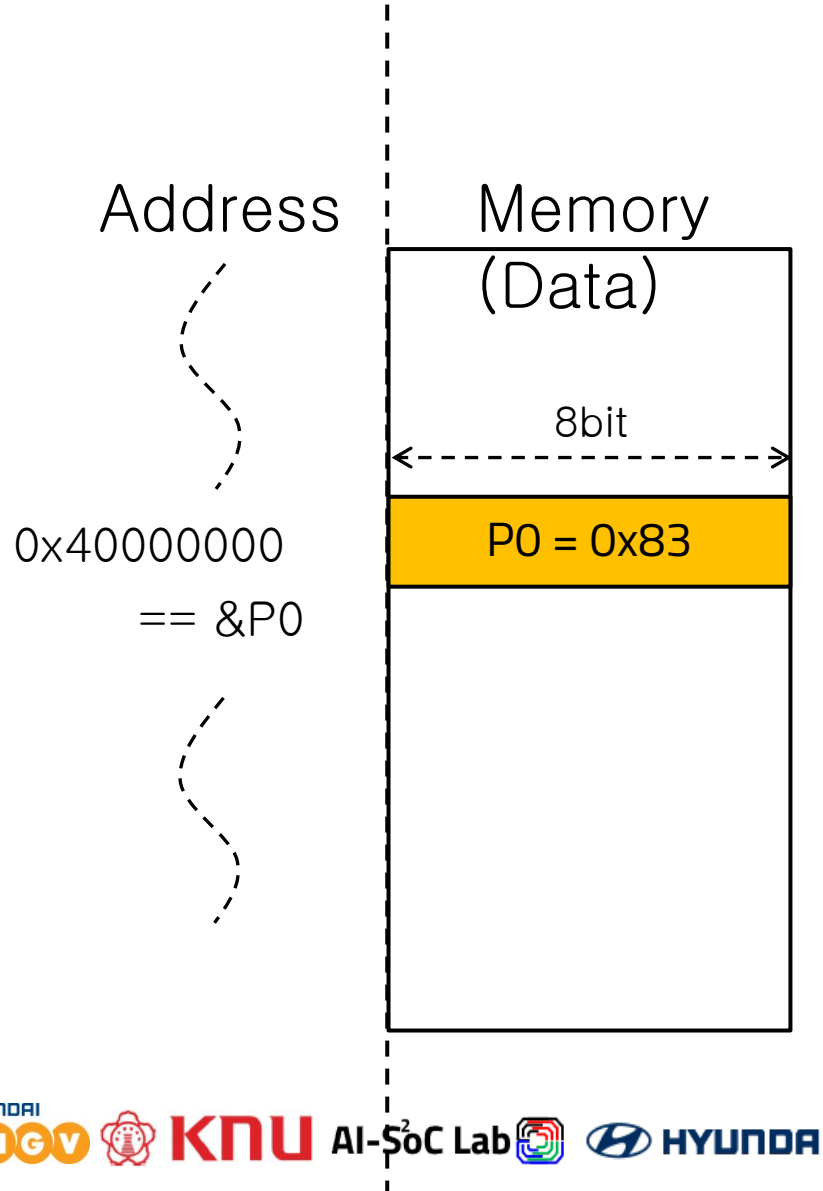
시스템 프로그래밍을 위한 C언어

- Bit Manipulation to Control Hardware -

현대자동차 입문교육
박대진 교수

Pointer: Method to Access Registers in C Language



Region on Memory vs. its location

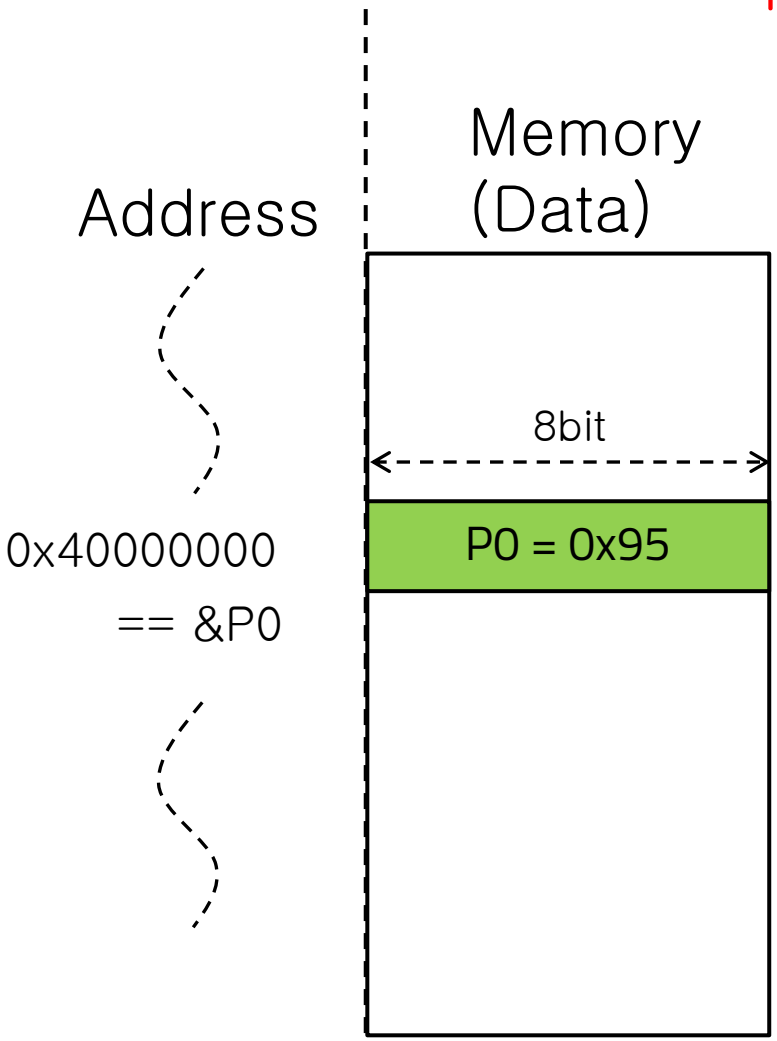
printf("%X", P0) ⟶ 0x83

printf("%X", &P0) ⟶ 0x40000000

Writing a value into memory region

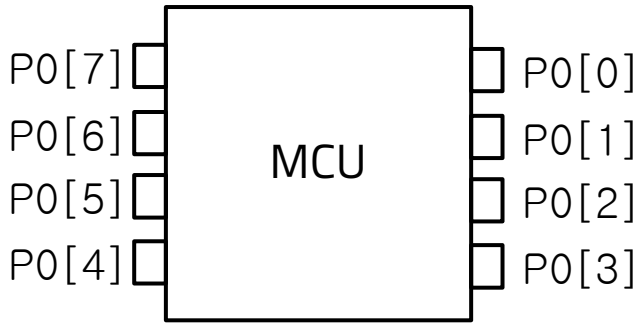
```
P0 = 0x83;

*(&P0) = 0x83;

(*(unsigned char*)0x40000000) = 0x83;
```

Toggling Individual Bits in Register Data



P0 at 0x40000000

7	6	5	4	3	2	1	0
P0[7]	P0[6]	P0[5]	P0[4]	P0[3]	P0[2]	P0[1]	P0[0]

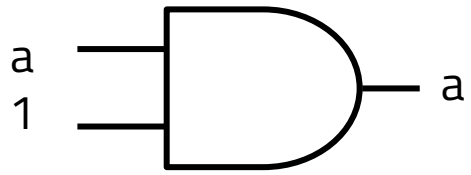


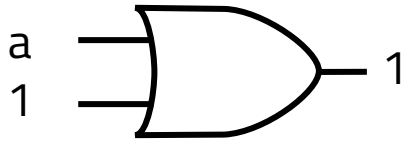
Setting P0[7],P[4],P[2],P[0] with 1

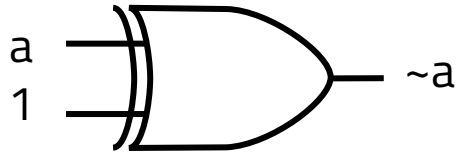
```
#include “header.h”
#define P0 (*(volatile unsigned char*)0x40000000)
P0 = 0x95;
```

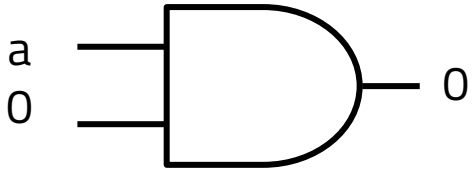
7	6	5	4	3	2	1	0
1	0	0	1	0	1	0	1

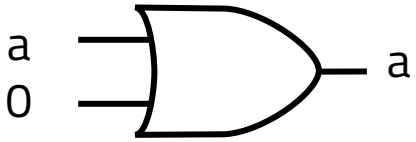
Logical Operation

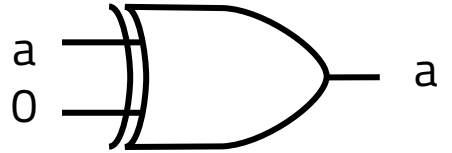












Bit Manipulation for Bit-Clear

We want to clear 4th bit, still don't touch other bits

P0 = 0x95;

7	6	5	4	3	2	1	0
1	0	0	1	0	1	0	1

P0 = P0 & 0xEF;

Bit-by-bit 'and' logical operation

P0 → 1 0 0 1 0 1 0 1

& → 1 1 1 0 1 1 1 1

0xEF → b1110_1111

→ 1 0 0 0 0 1 0 1

Three alternatives to clear specific bit of register data

- P0 = P0 & ~(0x10);
- P0 = P0 & ~(1<<4)
- P0 &= ~(1<<4)

Bit Manipulation for Bit-Set

We want to set 3rd bit, still don't touch other bits

P0 = 0x85;

7	6	5	4	3	2	1	0
1	0	0	0	0	1	0	1

P0 = P0 | 0x08;

Bit-by-bit 'or' logical operation

P0

or

0x08 → b0000_1000

1	0	0	0	0	1	0	1
0	0	0	0	1	0	0	0
1	0	0	0	1	1	0	1

Three alternatives to set specific bit of register data

```
P0 = P0 | (0x08);
P0 = P0 | (1<<3)
P0 |=      (1<<3)
```

Bit Manipulation for Bit Toggle

We want to toggle 3rd bit, still don't touch other bits

P0 = 0x85;

7	6	5	4	3	2	1	0
1	0	0	0	0	1	0	1

P0 = P0 ^ 0x08;

Bit-by-bit 'or' logical operation

P0

or

0x08 → b0000_1000

1	0	0	0	0	1	0	1
0	0	0	0	1	0	0	0
1	0	0	0	1	1	0	1

Three alternatives to set specific bit of register data

```
P0 = P0 ^ (0x08);
P0 = P0 ^ (1<<3)
P0 ^=     (1<<3)
```

Bit Masking-based Bit Manipulation

```c
#include <stdio.h>

int main() {
    unsigned char P0 = 0x95;

    P0 = P0 & ~(1<<4);
    printf("P0: 0x%02X\n", P0);

    P0 = P0 & ~(0x80);
    printf("P0: 0x%02X\n", P0);
    return 0;
}
```

union, struct Bit Slice Representation

```c
struct BITS8 {
    unsigned char b0 : 1;
    unsigned char b1 : 1;
    unsigned char b2 : 1;
    unsigned char b3 : 1;
    unsigned char b4 : 1;
    unsigned char b5 : 1;
    unsigned char b6 : 1;
    unsigned char b7 : 1;
};
union PORT0 {
    unsigned char U;
    struct BITS8  B;
};
```

```c
union PORT0 port0;
port0.U = 0xFF;
port0.B.b4 = 0; // clear bit at index 4
printf("port0: 0x%02X\n", port0.U);
```

Bit Set Check

원하는 위치의 비트가 1인지 비교할려면 전체 정수값으로 비교값을 표현해주어야 함
0인지 비교하고, 아닌경우에 내가 원하는 조건을 처리하도록 구현하는 것을 추천

```
// expecting 1, --> more complex
if((port0.U & (1<<EOC_IDX)) == 8) // port0[3] bit is 1
    printf("End of Conversion\n") ;
else // otherwise, port0[3] is 0
    printf("port0[3] is still 0\n");

// bit compare with 0, otherwise we expect value, more simple
if((port0.U & (1<<EOC_IDX)) == 0) // port0[3] bit is still 0
    printf("port0[3] is still 0\n");
else // otherwise, port0[3] is 1
    printf("End of Conversion\n") ;
```

Continuous Check via Memory Bus

- 하드웨어는 버스에 연결되어 있다.
- 하드웨어에 접근할 때 메모리 버스를 경유하므로 latency 가 발생한다.
- 그래서 if로 딱 한번만 비교하고 넘어가면 안된다
- 지속적인 비교를 하기 위해 while문을 사용하고,
- 그 값이 0이 되는 조건으로 변환하여,
- 0이 아니게 되면 계속 버스를 경유하여 하드웨어의 값을 읽어내도록 해야 함

```
// check memory bus idle or flag check..
// (via memory mapped-IO based hardware access)
while ((port0.U & (1<<EOC_IDX)) == 0); // port0[3] is still 0, on ADC conversion

// port0[3] is 1, so, while(false) --> stop loop
// so, go through here,
printf("End of Conversion (while self check technique)\n") ;
```

멀티 비트 슬라이스에 값을 셋팅

- 해당 비트 슬라이스외에 다른 값은 유지한채로 해당 비트들만 업데이트 해야 함.
- 해당 슬라이스를 0으로 클리어하고, or연산으로 해당 비트만 업데이트 할수 있음
- 비트 마스크를 만들기 위해 해당 비트 슬라이스를 1로 채운뒤 inversio하고 해당 위치로 shift해야 함

```
// multi-bit slice update
port0.U &= ~(0xF << ADC_DATA_IDX);
port0.U |=  (0x3 << ADC_DATA_IDX);
printf("port0: 0x%02X\n", port0.U);
```