



C Programming – 3

임베디드스쿨2기

Lv2과정

2021. 04. 01

박태인

1. 비트 연산자(1)

1) AND

C 프로그램 내에서는 & 로 사용하고 있다.

관계 연산자에서는 && 로 사용된다.

동작 방식은 서로 같은 자리수의 비트가 1(참)인지에 따라 결정됨.

진리표

```
  1 0 1 0 0 (20)
& 0 1 1 1 0 (14)
-----
  0 0 1 0 0 (4)
```

2) OR

C 프로그램 내에서는 | 로 사용한다.

관계 연산자에서는 || 로 사용된다.

동작 방식은 같은 자리수의 비트에 하나라도 1(참)이 있다면 참이 된다.

진리표

```
  1 0 1 0 0 (20)
| 0 1 1 1 0 (14)
-----
  1 1 1 1 0 (30)
```

1. 비트 연산자(2)

1) NOT

C 프로그램 내에서는 ~ 로 사용한다.
관계 연산자는 ! 로 사용된다.
동작 방식은 **무조건 비트를 반전** 시킨다.

진리표

~ 1 0 1 0 0 (20)

111111...0 1 0 1 1

앞자리가 1111...으로 반전되는 이유?

↳ 레지스터 비트수는 ? 64 비트

그러므로 **int(32비트)** 라 가정 했을 때 실질적으로 위의 5비트 앞에 27 비트의 0이 배치되어 있다 !

결국 NOT 연산은 비트 반전이 발생하므로 상위에 있는 나머지 비트들도 살펴봐야 한다.

기본적으로 변수들은 레지스터에 할당 됩니다.
그런데 메모리 계층구조상 레지스터의 용량은? 작다.
작기 때문에 많은 분량을 수용할 수가 없습니다.
그러므로 상대적으로 용량이 큰 메모리(DRAM)을 사용하게 됩니다.
(하지만, **실질적인 연산 체계는 레지스터가 담당한다 - ALU**)

레지스터에 값이 써져 있지 않다면 ?

플로팅 상태 라고 하는데 이런 상태를 원치 않으니

↳ 뒷 페이지 설명.

내부적으로 **회로에 값을 설정하지 않으면 0 으로 셋팅** 되게 되어 있다.

1. 비트 연산자(3)

1) NOT

C 프로그램 내에서는 ~ 로 사용한다.
관계 연산자는 ! 로 사용된다.
동작 방식은 **무조건 비트를 반전** 시킨다.

진리표

~ 1 0 1 0 0 (20)

111111...0 1 0 1 1

앞자리가 1111...으로 반전되는 이유?

↳ 레지스터 비트수는 ? 64 비트

그러므로 **int(32비트)** 라 가정 했을 때 실질적으로 위의 5비트 앞에 27 비트의 0이 배치되어 있다 !

결국 NOT 연산은 비트 반전이 발생하므로 상위에 있는 나머지 비트들도 살펴봐야 한다.

기본적으로 변수들은 레지스터에 할당 됩니다.
그런데 메모리 계층구조상 레지스터의 용량은? 작다.
작기 때문에 많은 분량을 수용할 수가 없습니다.
그러므로 상대적으로 용량이 큰 메모리(DRAM)을 사용하게 됩니다.
(하지만, **실질적인 연산 체계는 레지스터가 담당한다 - ALU**)

레지스터에 값이 써져 있지 않다면 ?

플로팅 상태 라고 하는데 이런 상태를 원치 않으니

↳ 뒷 페이지 설명.

내부적으로 **회로에 값을 설정하지 않으면 0 으로 셋팅** 되게 되어 있다.

1. 비트 연산자(4)

1) XOR

C 프로그램 내부에서는 ^ 로 표시한다.
안타깝게도 관계 연산자는 없다.

동작이 특이한데 비트가 서로 교차될 경우에만 1(참)이 되며,
비트가 서로 교차하지 않는다면 0(거짓)이 된다.
이 특성 때문에 암호화 과정에서 많이 사용하게 된다.

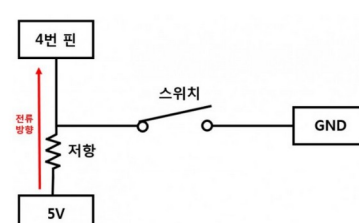
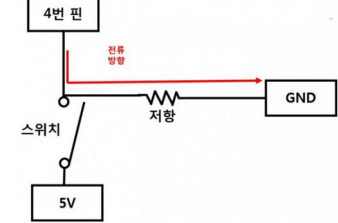
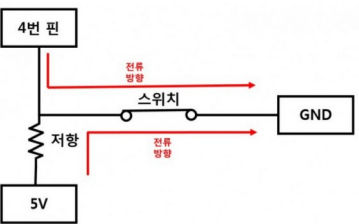
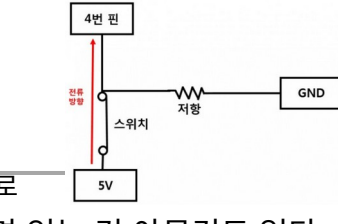
진리표

1	0	1	0	0	(20)
^	0	1	1	1	0 (14)
<hr/>					
1	1	0	1	0	(26)

❖ 플로팅(floating) 상태란?

영어 단어의 뜻과 같이 '떠 있는, 부동적인, 유동적인'이라는 의미를 가지고 있고,
스위치가 떨어져 있을 때와 스위치가 붙어 있을 때 0과 1의 값 사이를 떠다니면서 명확한 값을 출력하지 못하는 경우를 말함.

→ 이러한 플로팅(floating) 상태 해결 방법으로 풀업, 풀다운 저항을 사용한다.

종 류	풀업 방식(Pull up) [저항을 VCC 단자에 연결]	풀 다운 방식(Pull down) [저항을 GND 단자에 연결]
스위치 OPEN 상태	스위치가 열려 있기 때문에 전류는 GND가 아닌 입출력핀(4번)으로 흐르게 되고 따라서 입출력 핀에는 1(HIGH)의 값 을 읽을 수 있다. 	스위치가 열려 있게 되면 VCC와 회로는 단절되기 때문에 입출력핀(4번)에서 흐르는 전류는 GND로 향하게 됩니다. 따라서 입출력 핀은 0 (LOW)을 출력 합니다. 
스위치 CLOSE 상태	GND는 모든 전류가 도착하는 전압이 가장 낮은 지점이기 때문에 모든 전류는 GND 방향으로 흐르게 됩니다. 따라서 입출력핀(4번)에 흐르는 전류가 없기 때문에 이때는 0(LOW)가 출력 된다. 	스위치가 닫히게 되면 VCC부분이 회로와 연결 됩니다. 하지만 원래라면 VCC에서 흐르는 전류는 GND로 흘러야 하지만 GND 부분에 설치된 저항으로 인해 GND로 흐르지 못하고 스위치하면 얻는 건 아무것도 없다. 입출력 핀으로 출력 1(HIGH) 

2. 집중 해서 살펴 볼 부분(1)

1) NOT의 동작

0의 보수, 1의 보수, 2의 보수 ... 보기 싫어요.

$$+1 + (-1) = 0$$

보통 숫자는 부호 비트가 최상단에 배치되어 있고 하위로 숫자를 표현하게 되어 있다.
결국 부호비트가 1이면 음수가 되고 0 이면 양수가 된다.

그렇다면 **-1은 1000 0001(8비트 가정)이라 할 수 있는 것인가?**

0000 0001 (+1)

- 1000 0001 (-1)

1000 0010 ????? 0000 0000 ???

두 수를 더 했지만 0을 만족하지 못하므로 이것을 -1 이라 할 수 없다.

11111 111 올림 발생

0000 0001 (+1)

- 1111 1111 (- 1)

0000 0000 (0)

하드웨어 레지스터에 보면 EFLAGS(Intel), CPSR | SPSR (ARM)에 캐리비트가 셋 됨.

여기 에서 어떤 수를 음수로 표현하기 위한 규칙을 찾아야 한다(다음페이지)

2. 집중 해서 살펴 볼 부분(2)

1) NOT의 동작

자, 어떤 수를 음수로 표현하기 위한 규칙은 아래와 같다.

1-1. 뒤쪽에서 가장 먼저 나오는 1을 찾는다.

1-2. 맨 뒤에서 1까지의 숫자들은 그대로 유지한다.

1-3. 1 이후의 숫자들은 전부 반전 시킨다.

11111 11	11111 1
0000 1010 (+10)	0001 0100 (+20)
+ 1111 0110 (-10)	1110 1100 (-20)
-----	-----
0000 0000 (0)	0000 0000 (0)

돌발 퀴즈) 숫자 37을 지금 방식을 통해 -37로 만들어 보자.

11111 111
0010 0101 (+37)
+ 1101 1011 (-37)

0000 0000 (0)

돌발 퀴즈) -55, 2진수 만들어 보자.

$$1100\ 1001 (-X) = -55$$
$$0011\ 0111 (+X) = 1 + 2 + 4 + 16 + 32 = 55$$

2. 집중 해서 살펴 볼 부분(3)

1) NOT의 동작

NOT을 분석해 보자!

~10

~ 0000 1010 (+ 10)

1111 0101 (-X)

1111 0101(-X) = -11

0000 1011(+X) = 1 + 2 + 8 = 11

풀이) ~10은 10에서 모든 비트를 반전 시킨다. 이러한 2진수가 음수 몇인지 알아내기 위해 반대로 +X로 바꾸어 본다.
그러면 +11이 나오고 -X는 -11 이었다는 것을 알 수 있다.

돌발 퀴즈 : ~39를 위의 방식을 적용해서 연습해 보자!

~ 0010 0111 (+39)

1101 1000 (-X)

1101 1000 (-X) = -40

0010 1000(+X) = 32 + 8 = 40

결론) ~(+X)는 -(X+1) 가 된다.

2. 집중 해서 살펴 볼 부분(4)

2) XOR의 동작

A = 65

a = 97

1 0 0 0 0 0 1 (65) => 'A'

[^]0 1 0 0 0 0 0 (32)

1 1 0 0 0 0 1 (97) => 'a'

1 1 0 0 0 0 1 (97) => 'a'

[^]0 1 0 0 0 0 0 (32)

1 0 0 0 0 0 1 (65) => 'A'

돌발 퀴즈 : 70에 **XOR 0x20(32)**를
적용하는 동작을 직접 그려보도록 합니다.

1 0 0 0 1 1 0 (70) => 'F'

[^]0 1 0 0 0 0 0 (32)

1 1 0 0 1 1 0 (102) => 'f'

1 1 0 0 1 1 0 (102) => 'f'

[^]0 1 0 0 0 0 0 (32)

1 0 0 0 1 1 0 (70) => 'F'

결론 : XOR 0x20(32)를 하면
대소문자의 반대 값이 나온다 !

2. 집중 해서 살펴 볼 부분(5)

2) AND NOT 이란

사용처는 2의 n승 단위 정렬을 하고자 하는 경우에 매우 유용하다.

보통 리눅스 커널의 페이지 프레임이 4KB(4096) = 2^{12}

페이지 크기를 정렬하는 목적으로 많이 사용한다.

$$1000 \& \sim(2^n - 1)$$

$$23 = 16 + 4 + 2 + 1$$

$$11\ 1110\ 1000\ (1000) = 8 + 32 + 64 + 128 + 256 + 512 = ?\ 1000$$

$$1111\ (15) = 10000 - 1 = 16 - 1$$

$$11111\ (31) = 100000 - 1 = 32 - 1$$

$$11\mathbf{0}11\ (27) = \mathbf{31} - \mathbf{4} = 27$$

$$100\ 0000\ 0000\ (1024)$$

$$11\ 1111\ 1111\ (1023)$$

$$1\ 0111\ (23)$$

$$11\ 1110\ 1000\ (1000)$$

$$(2^n - 1) \Rightarrow 1111\dots111\ (2\text{진수})$$

$$16 \Rightarrow 10000 - 1 = 1111\ (2\text{진수})$$

$$32 \Rightarrow 100000 - 1 = 11111\ (2\text{진수})$$

$$64 \Rightarrow 1000000 - 1 = 111111\ (2\text{진수})$$

$$\begin{array}{l} 15\ (2^n - 1) \\ 31 \\ 63 \end{array}$$

결국 위의 흐름을 따르면 $\sim(2^n - 1) \Rightarrow 11111\dots111000\dots000$ 의 형태를 가지게 된다.

$$11\ 1110\ 1000\ (1000)$$

$$\& 11\ 1100\ 0000\ \sim(2^n - 1)$$

$$11\ 1100\ 0000 = 64 + 128 + 256 + 512 = 320 + 640 = 960$$

2. 집중 해서 살펴 볼 부분(6)

Gdb 디버깅 방법(1)

1. 먼저 프로그램에 -g 옵션을 붙여서 컴파일 한다.
2. gdb 실행파일명
3. b main (break main)
4. r (reset)
5. disas (어셈블리 코드를 볼 수 있다)

여기서 어셈블리어를 볼 수 있다.

다음 시간에는 이 내용을 기반으로 C언어 설계의 전반을 살펴보도록 한다.

```
gcc -g -o func func.c  
gdb func
```

: -g -o func func.c
↳ 형태 유의, 순서 유의

```
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04) 9.2  
Copyright (C) 2020 Free Software Foundation, Inc.  
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>  
This is free software: you are free to change and redistribute it.  
There is NO WARRANTY, to the extent permitted by law.  
Type "show copying" and "show warranty" for details.  
This GDB was configured as "x86_64-linux-gnu".  
Type "show configuration" for configuration details.  
For bug reporting instructions, please see:  
<http://www.gnu.org/software/gdb/bugs/>.  
Find the GDB manual and other documentation resources online at:  
<http://www.gnu.org/software/gdb/documentation/>.  
  
For help, type "help".  
Type "apropos word" to search for commands related to "word"...  
Reading symbols from func...  
(No debugging symbols found in func)
```

```
(gdb) b main  
Breakpoint 1 at 0x115c
```

```
(gdb) r  
Starting program: /home/taein/proj/es02/Lv01-02/TaeinPark/homework/c/03/func  
Breakpoint 1, 0x000055555555515c in main ()
```

```
(gdb) disas  
Dump of assembler code for function main:  
=> 0x000055555555515c <+0>:    endbr64  
0x0000555555555160 <+4>:    push    %rbp  
0x0000555555555161 <+5>:    mov     %rsp,%rbp  
0x0000555555555164 <+8>:    sub     $0x10,%rsp  
0x0000555555555168 <+12>:   movl    $0x3,-0x8(%rbp)  
0x000055555555516f <+19>:   mov     -0x8(%rbp),%eax  
0x0000555555555172 <+22>:   mov     %eax,%edi  
0x0000555555555174 <+24>:   callq   0x555555555149 <my_func>  
0x0000555555555179 <+29>:   mov     %eax,-0x4(%rbp)  
0x000055555555517c <+32>:   mov     -0x4(%rbp),%eax  
0x000055555555517f <+35>:   mov     %eax,%esi  
0x0000555555555181 <+37>:   lea     0xe7c(%rip),%rdi    # 0x555555556004  
0x0000555555555188 <+44>:   mov     $0x0,%eax  
0x000055555555518d <+49>:   callq   0x555555555050 <printf@plt>  
0x0000555555555192 <+54>:   mov     $0x0,%eax  
0x0000555555555197 <+59>:   leaveq  %eax  
0x0000555555555198 <+60>:   retq  
End of assembler dump.
```

2. 집중 해서 살펴 볼 부분(7)

Gdb 디버깅 방법(2)

L : C 레벨로 현재 코드를 볼 수 있음

```
(gdb) l
8      {
9          return num + 6;
10     }
11
12     int main(void)
13     {
14         int num1 = 3, res;
15
16         res = my_func(num1);
17     }
```

원 C 코드

```
int my_func(int num)
{
    return num + 6;
}

int main(void)
{
    int num1 = 3, res;

    res = my_func(num1);

    printf("res = %d\n", res);

    return 0;
}
```

N : 명령어 한 줄 실행(함수 호출 건너뛰기)

```
(gdb) l
8      {
9          return num + 6;
10     }
11
12     int main(void)
13     {
14         int num1 = 3, res;
15
16         res = my_func(num1);
17     }
(gdb) n
14         int num1 = 3, res;
(gdb) n
16         res = my_func(num1);
(gdb) n
18         printf("res = %d\n", res);
(gdb) n
res = 9
20         return 0;
(gdb) n
21     }
(gdb) n
__libc_start_main (main=0x5555555515c <main
                  rtld_fini=<optimized out>, stack_end=0x7
342     ../csu/libc-start.c: No such file or
```

2. 집중 해서 살펴 볼 부분(8)

Gdb 디버깅 방법(3)

S : 명령어 한 줄 실행(함수 호출 타고 들어감)

```
(gdb) l
8      {
9          return num + 6;
10     }
11
12     int main(void)
13     {
14         int num1 = 3, res;
15
16         res = my_func(num1);
17
18         printf("res = %d\n", res);
19     }
20
21     __printf__ (format=0x555555556004 "res = %d\n") at printf.c: No such file or directory.
22
23     (gdb) s
24     my_func (num=21845) at func.c:8
25     {
26
27     (gdb) s
28         return num + 6;
29     }
30     (gdb) s
31     main () at func.c:18
32     18         printf("res = %d\n", res);
33     (gdb) s
34     __printf__ (format=0x555555556004 "res = %d\n") at printf.c: No such file or directory.
```

원 C 코드

```
int my_func(int num)
{
    return num + 6;
}

int main(void)
{
    int num1 = 3, res;

    res = my_func(num1);

    printf("res = %d\n", res);

    return 0;
}
```


2. 집중 해서 살펴 볼 부분(9)

Gdb 디버깅 방법(4)

Ni : 어셈블리 명령어 한 줄을 실행함(함수 호출 건너 뛴)

```
(gdb) disas
Dump of assembler code for function main:
0x00005555555515c <+0>:    endbr64
0x000055555555160 <+4>:    push    %rbp
0x000055555555161 <+5>:    mov     %rsp,%rbp
0x000055555555164 <+8>:    sub     $0x10,%rsp
0x000055555555168 <+12>:   movl    $0x3,-0x8(%rbp)
=> 0x00005555555516f <+19>:   mov     -0x8(%rbp),%eax
0x000055555555172 <+22>:   mov     %eax,%edi
0x000055555555174 <+24>:   callq   0x55555555149 <my_func>
0x000055555555179 <+29>:   mov     %eax,-0x4(%rbp)
0x00005555555517c <+32>:   mov     -0x4(%rbp),%eax
0x00005555555517f <+35>:   mov     %eax,%esi
0x000055555555181 <+37>:   lea     0xe7c(%rip),%rdi    # 0
0x000055555555188 <+44>:   mov     $0x0,%eax
0x00005555555518d <+49>:   callq   0x55555555050 <printf@plt>
0x000055555555192 <+54>:   mov     $0x0,%eax
0x000055555555197 <+59>:   leaveq  %eax
0x000055555555198 <+60>:   retq
End of assembler dump.
```

함수가
호출 되지
않음



```
(gdb) ni
0x000055555555172 16      res = my_func(num1);
(gdb) ni
0x000055555555174 16      res = my_func(num1);
(gdb) ni
0x000055555555179 16      res = my_func(num1);
(gdb) ni
18      printf("res = %d\n", res);
(gdb) disas
Dump of assembler code for function main:
0x00005555555515c <+0>:    endbr64
0x000055555555160 <+4>:    push    %rbp
0x000055555555161 <+5>:    mov     %rsp,%rbp
0x000055555555164 <+8>:    sub     $0x10,%rsp
0x000055555555168 <+12>:   movl    $0x3,-0x8(%rbp)
0x00005555555516f <+19>:   mov     -0x8(%rbp),%eax
0x000055555555172 <+22>:   mov     %eax,%edi
0x000055555555174 <+24>:   callq   0x55555555149 <my_func>
0x000055555555179 <+29>:   mov     %eax,-0x4(%rbp)
=> 0x00005555555517c <+32>:   mov     -0x4(%rbp),%eax
0x00005555555517f <+35>:   mov     %eax,%esi
0x000055555555181 <+37>:   lea     0xe7c(%rip),%rdi    # 0x
0x000055555555188 <+44>:   mov     $0x0,%eax
0x00005555555518d <+49>:   callq   0x55555555050 <printf@plt>
0x000055555555192 <+54>:   mov     $0x0,%eax
0x000055555555197 <+59>:   leaveq  %eax
0x000055555555198 <+60>:   retq
```

원 C 코드

```
int my_func(int num)
{
    return num + 6;
}

int main(void)
{
    int num1 = 3, res;

    res = my_func(num1);

    printf("res = %d\n", res);

    return 0;
}
```

포기하면 얻는 건 아무것도 없다.

2. 집중 해서 살펴 볼 부분(10)

Gdb 디버깅 방법(5)

Si : 어셈블리 명령어 한 줄을 실행함(함수 호출 타고 들어감)

함수
호출!!

```
(gdb) si
my_func (num=21845) at func.c:8
8      {
(gdb) disas
Dump of assembler code for function my_func:
=> 0x000055555555149 <+0>:    endbr64
    0x00005555555514d <+4>:    push    %rbp
    0x00005555555514e <+5>:    mov     %rsp,%rbp
    0x000055555555151 <+8>:    mov     %edi,-0x4(%rbp)
    0x000055555555154 <+11>:   mov     -0x4(%rbp),%eax
    0x000055555555157 <+14>:   add     $0x6,%eax
    0x00005555555515a <+17>:   pop     %rbp
    0x00005555555515b <+18>:   retq
End of assembler dump.
(gdb) si
0x00005555555514d      8      {
(gdb) disas
Dump of assembler code for function my_func:
=> 0x000055555555149 <+0>:    endbr64
    0x00005555555514d <+4>:    push    %rbp
    0x00005555555514e <+5>:    mov     %rsp,%rbp
    0x000055555555151 <+8>:    mov     %edi,-0x4(%rbp)
    0x000055555555154 <+11>:   mov     -0x4(%rbp),%eax
    0x000055555555157 <+14>:   add     $0x6,%eax
    0x00005555555515a <+17>:   pop     %rbp
    0x00005555555515b <+18>:   retq
```

```
(gdb) disas
Dump of assembler code for function main:
0x00005555555515c <+0>:    endbr64
0x000055555555160 <+4>:    push    %rbp
0x000055555555161 <+5>:    mov     %rsp,%rbp
0x000055555555164 <+8>:    sub     $0x10,%rsp
0x000055555555168 <+12>:   movl    $0x3,-0x8(%rbp)
0x00005555555516f <+19>:   mov     -0x8(%rbp),%eax
0x000055555555172 <+22>:   mov     %eax,%edi
=> 0x000055555555174 <+24>:   callq   0x55555555149 <my_func>
0x000055555555179 <+29>:   mov     %eax,-0x4(%rbp)
0x00005555555517c <+32>:   mov     -0x4(%rbp),%eax
0x00005555555517f <+35>:   mov     %eax,%esi
0x000055555555181 <+37>:   lea     0xe7c(%rip),%rdi    # 0
0x000055555555188 <+44>:   mov     $0x0,%eax
0x00005555555518d <+49>:   callq   0x55555555050 <printf@plt>
0x000055555555192 <+54>:   mov     $0x0,%eax
0x000055555555197 <+59>:   leaveq  %rsi
0x000055555555198 <+60>:   retq
```

원 C 코드

```
int my_func(int num)
{
    return num + 6;
}

int main(void)
{
    int num1 = 3, res;

    res = my_func(num1);

    printf("res = %d\n", res);

    return 0;
}
```

2. 집중 해서 살펴 볼 부분(6)

3) 함수

- 프로토 타입

리턴 타입 : 어떠한 데이터 타입을 return 할 것 인지

인자 타입 : 어떠한 데이터 타입을 입력으로 수용 할 것인지

함수 이름 : 호출할 이름

- 돌발 퀴즈 : float 타입의 변수 3개를 서로 더하도록 함수를 만들어보자!

```
#include <stdio.h>

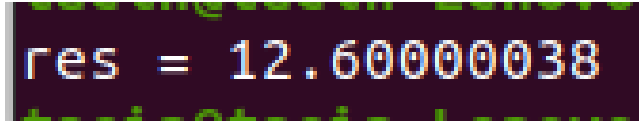
float my_func(float a, float b, float c)
{
    return a + b + c;
}

int main(void)
{
    float num1 = 3.3f, num2 = 3.7f, num3 = 5.6f, res;

    res = my_func(num1, num2, num3);

    printf("res = %.8f\n", res);

    return 0;
}
```



res = 12.60000038

Printf 에서 %f 가 아닌 %.8f 를 하면 소수점 8자리 까지 나타내게 되는데 7,8 자리의 수는 오차가 존재할 경우 나타난 것이다.

3. 코딩 연습(1)

1) bit_and

```
#include <stdio.h>

int main(void)
{
    int num_a = 10;
    int num_b = 7;

    printf("%d and %d = %d\n", num_a, num_b, num_a & num_b);

    return 0;
}
```

```
taein@taein-Lenovo-ideapad-700-15ISK:~/proj/es02/Lv01-02/TaeinPark/homework/c/03$ ./bit_and
10 and 7 = 2
```

1 0 1 0 (10)
& 0 1 1 1 (7) => 비트 연산

0 0 1 0 (2)

3. 코딩 연습(2)

bit_and_not

```
int main(void)
{
    int num_a, num_b, res, div;

    printf("원하는 숫자를 입력 하세요: ");

    scanf("%d", &num_a);

    printf("배수단위로 정렬하고자 하는 범위를 입력하세요: ");

    scanf("%d", &num_b);

    res = num_a & ~(num_b - 1);
    printf("res = %d\n", res);

    div = res / num_b;
    printf("res(%d) / num_b(%d) = div(%d)\n", res, num_b, div);
    printf("div(%d) * num_b(%d) = res(%d)\n", div, num_b, div * num_b);

    return 0;
}
```

num_a = 원하는 숫자 최대 범위

num_b = 배수 단위 정렬하고자 하는 범위 ($2^n - 1$)

└ 64 입력 → ($2^n - 1$) 이므로, $1000000 - 1 = 111111$ (2진수)

└ 여기에 $\sim(2^n - 1)$ 하면 11111...1100000

└ 따라서 11 1110 1000 (1000)

 & 11 1100 0000 $\sim(2^n - 1)$

11 1100 0000 = 64 + 128 + 256 + 512 = 320 + 640 = 960 (**res**)

└ $64 * 15(\text{div}) = 960$

```
taein@taein-Lenovo-ideapad-700-15ISK:~/proj/es02/
원하는 숫자를 입력 하세요: 1000
배수단위로 정렬하고자 하는 범위를 입력하세요: 64
res = 960
res(960) / num_b(64) = div(15)
div(15) * num_b(64) = res(960)
```

3. 코딩 연습(3)

bit_not

```
#include <stdio.h>

int main(void)
{
    int num_a = 20;

    printf("not %d = %d\n", num_a, ~num_a);

    return 0;
}
```

```
taein@taein-1
not 20 = -21
```

~ 0001 0100 (20)

1110 1011 (-X)

1110 1011 (-X) = -21

0001 0101 (+X) = 16 + 4 + 1 = 21

1 : 제일 처음 1은 그대로 1로 내려오고, 앞의 수(파란색 수) 들은 역으로 교 환!

3. 코딩 연습(4)

bit_or

```
#include <stdio.h>

int main(void)
{
    int num_a = 20;
    int num_b = 14;

    printf("%d or %d = %d\n", num_a, num_b, num_a | num_b);

    return 0;
}
```

```
taein@taein-Len
20 or 14 = 30
```

$$\begin{array}{r} 0001\ 0100\ (20) \\ | 0000\ 1110\ (14) \\ \hline 0001\ 1110\ (30) \end{array}$$

OR : 하나라도 1 이면 1, (1 1 이어도 올림 1이 되진 않음)

3. 코딩 연습(5)

bit_xor

```
#include <stdio.h>

int main(void)
{
    int num_a = 20;
    int num_b = 14;

    printf("%d xor %d = %d\n", num_a, num_b, num_a ^ num_b);

    return 0;
}
```

```
taein@taein-Len
20 xor 14 = 26
```

$$\begin{array}{r} 0001\ 0100\ (20) \\ \wedge\ 0000\ 1110\ (14) \\ \hline 0001\ 1010\ (26) \end{array}$$

XOR : 1 , 0 이 교차시에만 1

3. 코딩 연습(6)

기본 함수

```
#include <stdio.h>

// 리턴 타입 : int
// 함수 이름 : my_func
// 인자 타입 : int

int my_func(int num)
{
    return num + 6;
}

int main(void)
{
    int num1 = 3, res;

    res = my_func(num1);

    printf("res = %d\n", res);

    return 0;
}
```

```
taein@ta
res = 9
taein@ta
```

3. 코딩 연습(7)

입력이 2개인 함수

```
#include <stdio.h>

// f = g(x,y) ==> 입력이 두 개인 함수
int my_func(int a, int b)
{
    return a + b;
}

int main(void)
{
    int num1 = 3, num2 = 7, res;

    //함수 이름을 통해 함수를 실질적으로 호출
    //res = num1 + num2;
    res = my_func(num1, num2);

    printf("res = %d\n", res);

    return 0;
}
```

```
taeIn@taei
res = 10
taeIn@taei
```

3. 코딩 연습(8)

쉬프트 연산자

```
#include <stdio.h>

int my_func(int num)
{
    // 쉬프트 연산자로 비트를 전체적으로 1칸 밀어 올림
    // num << 1 은 1칸, num << 2 는 2칸, num << 3 은 3칸
    // 비트의 2^n을 의미하므로 결국 2^n 비트 이동 칸 수 만큼 숫자가 곱해진다.

    // num >> 1 은 1칸, num >> 2는 2칸 이동이 동일하다.
    // 그러나 이 방식은 우측 이동이다.
    // 그러므로 나누기와 같은데 주의점은
    // 소수점을 취급하지 않아 0.5 같은건 버린다.
    return num >> 1;
}

int main(void)
{
    int num = 3, res;

    res = my_func(num);

    printf("res = %d\n", res);

    return 0;
}
```

```
res = 1
```

0011 → 0001 : >> 이므로 전체적으로 움직여 지고 제일 좌측에 0 생김

```
return num << 1;
```

0011 → 0110 : << 이므로 전체적으로 움직여 지고 제일 우측에 0 생김

```
res = 6
```


3. 코딩 연습(9)

대소문자

```
#include <stdio.h>

int main(void)
{
    char sec_char;
    printf("암호 글자를 한 개 입력 하세요 : ");
    scanf("%c", &sec_char);

    printf("입력한 글자는 = %c\n", sec_char);
    printf("또한 ASCII로 표현하면 = %d\n", sec_char);

    // 대문자 소문자 변환하기
    // 먼저 대문자인지 파악하고 + 32
    // 소문자면 -32

    // 0x20은 16진수 표기이며 10진수로 32에 해당합니다.
    // 16 x 2 = 32
    printf("대문자는 소문자가 되고 소문자는 대문자가 된다. \n");
    printf("원래는 %c, 지금은 %c\n", sec_char, sec_char ^ 0x20);

    return 0;
}
```

```
taein@taein-Lenovo-ideapad-700-15ISK: ~/proj/es0
암호 글자를 한 개 입력 하세요 : c
입력한 글자는 = c
또한 ASCII로 표현하면 = 67
대문자는 소문자가 되고 소문자는 대문자가 된다.
원래는 c, 지금은 c
taein@taein-Lenovo-ideapad-700-15ISK: ~/proj/es0
암호 글자를 한 개 입력 하세요 : d
입력한 글자는 = d
또한 ASCII로 표현하면 = 100
대문자는 소문자가 되고 소문자는 대문자가 된다.
원래는 d, 지금은 D
```

Q. 질문

11 1110 1000 (1000)
& 11 1100 0000 $\sim(2^n - 1)$

11 1100 0000 = 64 + 128 + 256 + 512 = 320 + 640 = 960
☞ 이와 같은 이유 때문에 2^n 단위로 숫자를 정렬 할 수 있게 된다.

⇒ 질문 : 2^n 에서 -1을 하는 이유는 10000 -> 1111 이런식 으로 나타내고 ~을 했을 때 1111..11000...00 으로 나타낼 수 있기 때문이고,
이러한 것이 2^n 단위의 숫자로 나타낼 수 있기 때문 인 것 같습니다.
그런데, 1000 이라는 숫자가 위 처럼 $\sim(2^n - 1)$ 이 2진수로 11 1100 0000 가 된다는 것은 어떻게 알 수 있는 것 인가요??