9장 비트수준 접근

비트수준 접근

- 모든 내용은 메모리에 0과 1의 이진값으로 저장
- 비트: 이진 값의 각자리
- 바이트: 8개 비트
- 워드: 메모리에서 읽혀지는 단위
- 비트열 예제
  - 0101 1101
  - 93_{10} ($=2^6+ 2^4+ 2^3+ 2^2+ 2^0$)
  - $5D_{16}$ ($0101 = 5_{16}$, $1101 = D_{16}$)
  - 비트열을 다룰때 16진수를 많이 사용

비트 단위 연산자

- int, short, long, long long, char 등과 같은 정수형 수식에 사용됨
- 시스템 종속적

비트 단위 연산자

논리 연산자	(단항) 비트단위 보수	~
	비트단위 논리곱	&
	비트단위 배타적 논리합	^
	비트단위 논리합	|
이동 연산자	왼쪽 이동	<<
	오른쪽 이동	>>

~ 연산자

- 1의 보수 연산자, 비트단위 보수 연산자
- 피 연산자의 각 비트를 0은 1로, 1은 0으로 함
- 예

```
int a = 0x10F07;// 0x10F07 = 69383
```

- a의 이진수 표현

00000000 00000001 00001111 00000111

- ~a의 이진수 표현

11111111 11111110 11110000 11111000

// = -69384(0xFFFEF0F8)

- 참고 : !a = 0

비트 단위 이진 논리 연산자

- 이진 논리 연산자
  - &(논리곱), |(논리합) , ^(배타적 논리합)
  - 두 피연산자는 대응되는 비트끼리 연산됨

a	b	a & b	a ^ b	a | b
0	0	0	0	0
1	0	0	1	1
0	1	0	1	1
1	1	1	0	1

비트 단위 이진 논리 연산자

프로그램 9.1

```
#include <stdio.h>
int main(void){
    int   a = 33333,  b = -77777, c = 0, d = 0, e = 0;
    c = a & b;
    d = a | b;
    e = a ^ b;
    printf("a : %x, b : %x\n", a, b);
    printf("a & b : %x\n", c);
    printf("a | b : %x\n", d);
    printf("a ^ b : %x\n", e);
    return 0;
}
```

비트 단위 이진 논리 연산자

- a, b의 비트 표현

```
a 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 1 0 0 0 1 1 0 1 0 1   33333
b 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 0 1 0 0 0 0 0 1 0 1 1 1 1 -77777
```

- c = a & b

```
a 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 1 0 0 0 1 1 0 1 0 1   33333
  & & & & & & & & & & & & & & & & & & & & & & & & & & & & & & &       &
b 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 0 1 0 0 0 0 0 1 0 1 1 1 1 -77777
c 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 0 0 1 0 1   32805
```

비트 단위 이진 논리 연산자

- d = a | b

```
a 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 1 0 0 0 1 1 0 1 0 1    33333
  | | | | | | | | | | | | | | | | | | | | | | | | | | | | |      |
b 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 0 1 0 0 0 0 0 1 0 1 1 1   -77777
d 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 0 1 0 0 1 0 0 0 1 1 1 1 1   -77249
```

- e = a ^ b

```
a 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 1 0 0 0 1 1 0 1 0 1    33333
  ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^      ^
b 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 0 1 0 0 0 0 0 1 0 1 1 1   -77777
e 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 1 0 1 0 0 1 0 0 0 0 1 1 0 1 0   -110054
```

프로그램 결과

```
a : 8235, b : fffed02f
a & b : 8025
a | b : fffed23f
a ^ b : fffe521a
```

마스킹 연산

- & 연산자를 사용하여 주어진 비트열의 특정 비트를 0으로 만드는 것
- 마스크 : 마스킹 연산을 위해 사용되는 상수나 변수
- 예제

```
int i, mask = 1;
// mask : 00000000000000000000000000000001
for(i=0; i<10; ++i)
   printf("%d", i & mask);
// i가 홀수면 1, 짝수면 0 출력
```

마스크

- 예제

  (v & 0x4) ? 1 : 0

  - 0x4 : 00000000 00000000 00000000 00000100
  - v의 3번째 비트가 1일 때 1

- 예제

  v & 255

  - 255 : 00000000 00000000 00000000 11111111
  - v의 하위 1 바이트 값

이동 연산자

- 지정된 값의 비트열을 이동시키는 연산자
- 이동 연산자의 두 피연산자는 정수 수식이어야 함
- 왼쪽 이동 연산자(<<)
- 오른쪽 이동 연산자(>>)

왼쪽 이동 연산자

- **수식1 << 수식2**
  - 수식1의 비트 표현을 수식2가 지정하는 수만큼 왼쪽으로 이동
  - 왼쪽 이동으로 발생하는 빈 공간은 0으로 채워짐
  - 2의 거듭제곱 효과를 볼 수 있음

왼쪽 이동 연산자

- 예

```
int c = 7;
// c = 00000000 00000000 00000000 00000111
```

```
c << 1   00000000000000000000000000001110          14
c << 8   00000000000000000000011100000000        1792
c << 30  11000000000000000000000000000000 -1073741924
```

오른쪽 이동 연산자

- 수식1 >> 수식2
  - 수식1의 비트 표현을 수식2가 지정하는 수 만큼 오른쪽으로 이동
  - 수식1이 unsigned 형이면 상위비트로 0이 들어옴
    - 2의 거듭제곱으로 나누기 연산 효과
  - 수식1이 signed 형이면 시스템 종속적
    - 상위 비트로 0이 들어올 수도 있고, 1이 들어올 수도 있음

오른쪽 이동 연산자

- 예

```
unsigned int    c = 0xf0000000;    // c = 4026531840
// c = 11110000 00000000 00000000 00000000
```

```
c >> 1   01111000000000000000000000000000  2013265920
c >> 8   00000000111100000000000000000000    15728640
c >> 30  00000000000000000000000000000011           3
```

오른쪽 이동 연산자

- 예

```
int    c = 0xf0000000;      // c = -268435456
// c = 11110000 00000000 00000000 00000000
```

- 시스템에 따라 다음 둘 중 하나로 동작

```
c >> 8   11111111111100000000000000000000    -1048576
c >> 8   00000000111100000000000000000000    15728640
```

복합 배정 연산자

- **복합 배정 연산자**
  - &=, ^=, |=, <<=, >>=
- **예**

```
a &= 255;
a = a & 255;
```

예제 프로그램

프로그램 9.2

```
#include <stdio.h>
void bit_print(int a){
    int     i;
    int     n = sizeof(int) * 8;
    int     mask = 1 << (n - 1);
    for (i = 1;  i <= n;  ++i)  {
        putchar(((a & mask) == 0) ? '0' : '1');
        a <<= 1;
        if (i % 8 == 0 && i < n)
            putchar(' ');
    }
}
```

예제 프로그램

프로그램 9.2

```
int main(void){
    int i;
    printf("정수를 입력하세요 : ");
    scanf("%d", &i);
    printf("%d 비트열 : ", i);
    bit_print(i);
    printf("\n");
    return 0;
}
```

프로그램 결과

```
$ bit_pr
정수를 입력하세요 : 1
1 비트열 : 00000000 00000000 00000000 00000001
$ bit_pr
정수를 입력하세요 : -1
-1 비트열 : 11111111 11111111 11111111 11111111
$ bit_pr
정수를 입력하세요 : 230490
230490 비트열 : 00000000 00000011 10000100 01011010
```

패킹과 언패킹

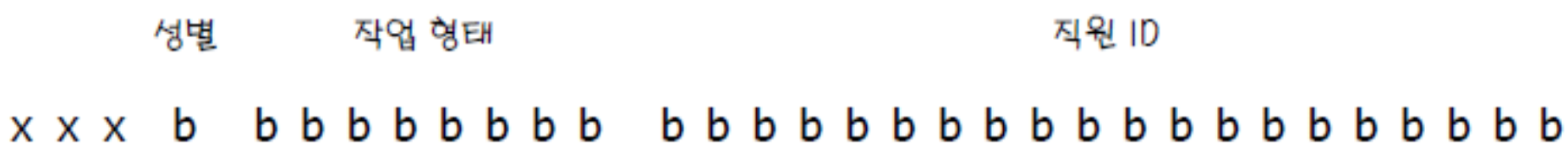
- 패킹 : 여러 정보를 비트 단위 연산자를 사용하여 적은 바이트로 압축하는 것
  - 메모리 절약
  - 전송시간 줄임
- 언패킹 : 패킹된 정보를 사용하기 전에 정보를 추출하는 것

패킹

- 직원 관리 프로그램에서 다음과 같은 정보를 다루어야 한다고 가정
  - 직원ID : 6자리(10진수)
  - 작업형태: 200가지
  - 성별
- 이 세가지 정보는 unsigned int에 패킹 가능
  - 직원ID : 20비트
  - 작업형태 : 8비트
  - 성별 : 1비트
  - 총 : 29비트

패킹

- unsigned int 사용 예



패킹

함수 9.1

```
unsigned pack_employee_data(unsigned id_no,
                            unsigned job_type, char gender)
{
    unsigned  employee = 0;
    employee |= id_no;
    employee |= job_type << 20;
    employee |= ((gender == 'm' || gender == 'M') ? 0 : 1) << 28;
    return employee;
}
```

언패킹

- 패킹된 정보는 사용하기 전에 언패킹 해야 함
- 적절한 마스크 필요
- 직원 관리 프로그램에서
  - 직원 ID를 위한 마스크 : 20개의 1
  - 작업 형태를 위한 마스크 : 8개의 1
  - 성별을 위한 마스트 : 1개의 1

함수 9.2

```
void print_employee_data(unsigned employee)
{
    unsigned id_no, job_type;
    char gender;
    id_no = employee & 0xFFFFF;
    job_type = (employee >> 20) & 0xFF;
    gender = (employee >> 28) & 1;
    printf("ID : %u", id_no);
    printf("작업 형태 : %u", job_type);
    printf("성별 : %s\n", gender? "여" : "남");
}
```

비트 필드

- 구조체나 공용체에서 int 형이나 unsigned 형의 멤버에 비트 수(폭)를 지정하는 것
- 폭은 콜론 다음에 음수가 아닌 정수형 상수 수식으로 지정되고, 최대 값은 멤버 변수의 비트 수와 같음
- 예제

```
struct employee {
  unsigned id_no : 20;
  unsigned job_type : 8;
  unsigned gender : 1;
}
```

비트 필드

- 컴파일러는 멤버들을 최소의 공간으로 패킹함
- 예제

```
struct employee {
    unsigned id_no : 20;
    unsigned job_type : 8;
    unsigned gender : 1;
}
```

  - struct employee는 4 바이트에 저장
  - 크기를 측정할 때에는 sizeof 연산자 사용

비트 필드

- 비트 필드를 사용하는 구조체에 일반 멤버도 있을 수 있음

```
struct employee_with_name {
    char name[30];              // 일반멤버
    unsigned id_no : 20;
    unsigned job_type : 8;
    unsigned gender : 1;
}
```

비트 필드

- 컴파일러는 비트 필드를 워드 경계에 걸치지 않게 메모리에 할당

```
struct abc {
    int a : 1, b : 16, c : 16;
} x;
```

  - a, b : 첫 번째 워드에 할당
  - c : 두 번째 워드에 할당

주의사항

- int 형 비트 필드는 시스템에 따라 unsigned int 비트 필드로 다루어짐
  - unsigned 비트 필드만을 사용하는 것이 좋음
- 비트 필드 배열은 허용되지 않음
- 비트 필드에 주소연산자 &를 적용할 수 없음
- 포인터가 직접 비트 필드를 포인트 할 수 없음

비트 필드

- 패딩과 정렬을 위해 이름 없는 비트 필드나 폭이 0인 비트 필드를 사용할 수 있음

```
struct small_integers {
    unsigned        i1 : 7, i2 : 7, i3 : 7,
                       : 11,
                    i4 : 7, i5 : 7, i6 : 7;
};
struct abc {
  unsigned          a : 1, : 0, b : 1, : 0, c : 1;
};
```

비트 필드

프로그램 9.3

```
typedef struct {
   unsigned   b0 : 8, b1 : 8, b2 : 8, b3 : 8;
} word_bytes;
typedef struct {
  unsigned
       b0  : 1, b1  : 1, b2  : 1, b3  : 1, b4  : 1, b5  : 1,
       b6  : 1, b7  : 1, b8  : 1, b9  : 1, b10 : 1, b11 : 1,
       b12 : 1, b13 : 1, b14 : 1, b15 : 1, b16 : 1, b17 : 1,
       b18 : 1, b19 : 1, b20 : 1, b21 : 1, b22 : 1, b23 : 1,
       b24 : 1, b25 : 1, b26 : 1, b27 : 1, b28 : 1, b29 : 1,
       b30 : 1, b31 : 1;
} word_bits;
typedef union {
   int         i;
   word_bytes  byte;
   word_bits   bit;
} word;
```

프로그램 9.3

```
void bit_print(int);
int main(void)
{
    word   w = {0};
    w.bit.b8 = 1;
    w.byte.b0 = 'a';
    printf("w.i = %d\n", w.i);
    bit_print(w.i);
    return 0;
}
```

비트 필드

```
w.i = 353
00000000 00000000 00000001 01100001

또는

w.i = 1635778560
01100001 10000000 00000000 00000000
```