

Jin-Soo Kim
(jinsoo.kim@snu.ac.kr)

Systems Software &
Architecture Lab.
Seoul National University

Spring 2022

Bitwise Operators



Bitwise Operation

■ Logical operator

- Bitwise complement: $\sim$
- Bitwise AND: $\&$
- Bitwise inclusive OR: $|$
- Bitwise exclusive OR: $\wedge$

X = 4190: 0001 0000 0101 1110

Y = 103: 0000 0000 0110 0111

X & Y: 0000 0000 0100 0110

X | Y: 0001 0000 0111 1111

X ^ Y: 0001 0000 0011 1001

■ Shift operator

- Left shift: $\ll$
- Right shift: $\gg$

NOT	
X	$\sim X$
0	1
1	0

AND		
X	Y	$X \& Y$
0	0	0
0	1	0
1	0	0
1	1	1

OR		
X	Y	$X Y$
0	0	0
0	1	1
1	0	1
1	1	1

XOR		
X	Y	$X \wedge Y$
0	0	0
0	1	1
1	0	1
1	1	0

Operator Precedence & Associativity

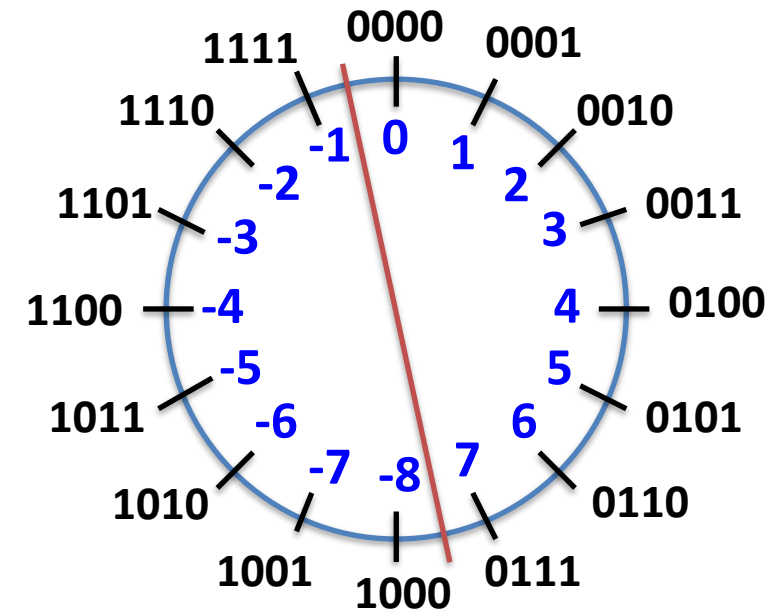
Operator	Associativity
() [] ++ (<i>postfix</i>) -- (<i>postfix</i>)	Left to right
+ (<i>unary</i>) - (<i>unary</i>) ++ (<i>prefix</i>) -- (<i>prefix</i>) ! & (<i>address</i>) * (<i>dereference</i>) ~	Right to left
* / %	Left to right
+ -	Left to right
<< >>	Left to right
< <= > >=	Left to right
== !=	Left to right
&	Left to right
^	Left to right
	Left to right
&&	Left to right
	Left to right
?:	Right to left
= += -= *= /= %= <<= >>= &= ^= =	Right to left
, (<i>comma operator</i>)	Left to right

Two's Complement (I)

- Unique zero
- Easy for hardware
 - leading 0 ≥ 0
 - leading 1 < 0
- Used by almost all modern machines



$$O(B) = -b_{w-1} \cdot 2^{w-1} + \left(\sum_{i=0}^{w-2} b_i \cdot 2^i \right)$$



Asymmetric range:

$$-2^{n-1} \sim 2^{n-1} - 1$$

Two's Complement (2)

- For $-n$, take the bitwise complement of n and add 1 to it:
 - $\sim n + n == 1111\dots11_2 == -1 \rightarrow \sim n + 1 == -n$

`short int n;`

Value of n	Binary representation	Binary complement	Two's complement of -n	Value of -n
1	00000000 00000001	11111111 11111110	11111111 11111111	-1
7	00000000 00000111	11111111 11111000	11111111 11111001	-7
8	00000000 00001000	11111111 11110111	11111111 11111000	-8
9	00000000 00001001	11111111 11110110	11111111 11110111	-9
-9	11111111 11110111	00000000 00001000	00000000 00001001	9
0	00000000 00000000	11111111 11111111	00000000 00000000	0

Bitwise Binary Operators

Declarations and initializations

```
int a = 33333, b = -77777;
```

Expression	Binary representation	Value
a	00000000 00000000 10000010 00110101	33333
b	11111111 11111110 11010000 00101111	-77777
a & b	00000000 00000000 10000000 00100101	32085
a ^ b	11111111 11111110 01010010 00011010	-110054
a b	11111111 11111110 11010010 00111111	-77249
~(a b)	00000000 00000001 00101101 11000000	77248

Some "Boolean" Properties

- $a \& 0 == 0$
- $a \& (-1) == a$
- $a \& a == a$
- $a \& (\sim a) == 0$
- $\sim(\sim a) == a$
- $a \& b == b \& a$
- $a | b == b | a$
- $a \wedge b == b \wedge a$
- $a \& (b | c) == (a \& b) | (a \& c)$
- $\sim(a | b) == \sim a \& \sim b$
- $a | 0 == a$
- $a | (-1) == -1$
- $a | a == a$
- $a | (\sim a) == -1$
- $a \wedge 0 == a$
- $a \wedge (-1) = \sim a$
- $a \wedge a == 0$
- $a \wedge (\sim a) == -1$
- $a \& (b \& c) == (a \& b) \& c$
- $a | (b | c) == (a | b) | c$
- $a \wedge (b \wedge c) == (a \wedge b) \wedge c$
- $a | (b \& c) == (a | b) \& (a | c)$
- $\sim(a \& b) == \sim a | \sim b$

Example: Bufferless Swap

- Note: $(a \oplus b) \oplus b == a \oplus (b \oplus b) == a$

```
#include <stdio.h>

int main(void)
{
    int a = 3, b = 9;          /* a <- 0011, b <- 1001 */

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

    a = a ^ b;                 /* a <- 0011 ^ 1001 = 1010 */
    b = a ^ b;                 /* b <- 1010 ^ 1001 = 0011 */
    a = a ^ b;                 /* a <- 1010 ^ 0011 = 1001 */
                                /* also try: a ^= b ^= a ^= b */
    printf("a = %d, b = %d\n", a, b);
    return 0;
}
```

An Interview Question

- Given a set of numbers where all elements occur even a number of times except one number, find the odd occurring number
 - findodd(12, 36, 7, 12, 9, 36, 7, 12, 12) → 9

```
#include <stdio.h>

int findodd(int a[], int n)
{
    int i, res = 0;

    for (i = 0; i < n; i++)
        res ^= a[i];
    return res;
}
```

```
int main(void)
{
    int odd;
    int a[] = {12, 36, 7, 12, 9, 36, 7, 12, 12};

    odd = findodd(a, sizeof(a)/sizeof(int));
    printf("%d\n", odd);
    return 0;
}
```

Shift Left Operator

▪ $expr1 \ll expr2$

- Shift left $expr1$ by $expr2$ bits (remaining bits on the right side are filled with **zeroes**)

Declarations and initializations

```
int c = 90, d = -12;
```

Expression	Binary representation	Action
c	00000000 00000000 00000000 01011010	unshifted
c << 1	00000000 00000000 00000000 10110100	left-shifted 1
c << 4	00000000 00000000 0000101 10100000	left-shifted 4
c << 31	00000000 00000000 00000000 00000000	left-shifted 31
d	11111111 11111111 11111111 11110100	unshifted
d << 28	01000000 00000000 00000000 00000000	left-shifted 28

(Logical) Shift Right Operator

- `expr1 >> expr2` (when `expr1` is an unsigned integer)
 - Shift right `expr1` by `expr2` bits (remaining bits on the left side are filled with zeroes)

Declarations and initializations

```
unsigned int u = 90, v = 0xdeadbeef;
```

Expression	Binary representation	Action
<code>u</code>	00000000 00000000 00000000 01011010	unshifted
<code>u >> 1</code>	00000000 00000000 00000000 00101101	right-shifted 1
<code>u >> 4</code>	00000000 00000000 00000000 0000101	right-shifted 4
<code>v</code>	11011110 10101101 10111110 11101111	unshifted
<code>v >> 8</code>	00000000 11011110 10101101 10111110	right-shifted 8
<code>v >> 24</code>	00000000 00000000 00000000 11011110	right-shifted 24

(Arithmetic) Shift Right Operator

- `expr1 >> expr2` (when `expr1` is a signed integer)
 - Shift right `expr1` by `expr2` bits (remaining bits on the left side are filled with *sign bits*)

Declarations and initializations

```
int m = 90, n = 0xdeadbeef;
```

Expression	Binary representation	Action
<code>m</code>	00000000 00000000 00000000 01011010	unshifted
<code>m >> 1</code>	00000000 00000000 00000000 00101101	right-shifted 1
<code>m >> 4</code>	00000000 00000000 00000000 0000101	right-shifted 4
<code>n</code>	11011110 10101101 10111110 11101111	unshifted
<code>n >> 8</code>	11111111 11011110 10101101 10111110	right-shifted 8
<code>n >> 24</code>	11111111 11111111 11111111 11011110	right-shifted 24

A Mask

- A constant or variable that is used to extract desired bits from another variable or expression
- Examples
 - The constant value 1:
used to determine the low-order bit of an `int` expression
 - The constant value 255:
a mask for the low-order byte

```
/* For n >= 0,  
   returns 1 if n is an odd number */  
int odd(int n)  
{  
    int mask = 1;  
    return (n & mask);  
}
```

```
/* For n >= 0, returns n % 256 */  
int mod256(int n)  
{  
    int mask = 255;  
    return (n & mask);  
}
```

Example: Printing an Integer Bitwise

```
#include <stdio.h>
#define BITS_PER_BYTE 8

void bit_print(int a)
{
    int i;
    int n = sizeof(int) * BITS_PER_BYTE;
    int mask = 1 << (n-1);

    for (i = 1; i <= n; i++)
    {
        putchar(((a & mask)? '1' : '0'));
        a <<= 1;          /* a = a << 1; */
        if (i % BITS_PER_BYTE == 0 && i < n)
            putchar(' ');
    }
}
```

Example: Extracting Bits from an Integer

- For a given integer,
 - Extract bit 31
 - Extract bits 30 ~ 23
 - Extract bits 22 ~ 0



```
#include <stdio.h>

int bit31(int n) {
    return (n >> 31) & 1;
}

int bit30_23(int n) {
    return (n >> 23) & 0xff;
}

int bit22_0(int n) {
    return n & 0x7fffffff;
}
```

Enumeration Types (I)

■ User defined type

- Provides a means of naming a finite set of **integers** and of declaring identifiers of the set
- `enum day {Sun, Mon, Tue, Wed, Thu, Fri, Sat}; // 0, 1, 2, 3, 4, 5, 6`
- `enum fruit {apple, pear, orange, lemon}; // 0, 1, 2, 3`
- `enum fruit {apple=7, pear, orange, lemon}; // 7, 8, 9, 10`

■ Variable declaration

- `enum day d1, d2`
`d1 = Fri;`
`if (d2 == Thu) ... /* if (d2 == 4) ... */`

Enumeration Types (2)

```
#include <stdio.h>
enum day {Sun, Mon, Tue, Wed, Thu, Fri, Sat};

enum day find_next_day(enum day d) {
    switch (d) {
        case Sun: return Mon;
        case Mon: return Tue;
        case Tue: return Wed;
        case Wed: return Thu;
        case Thu: return Fri;
        case Fri: return Sat;
        case Sat: return Sun;
    }
}

int main(void) {
    enum day d;
    for (d = Sun; d <= Sat; d++)
        printf("next day of %d is %d\n", d, find_next_day(d));
    return 0;
}
```

```
enum day find_next_day2(enum day d) {
    return ((d + 1) % (Sat - Sun + 1));
}
```