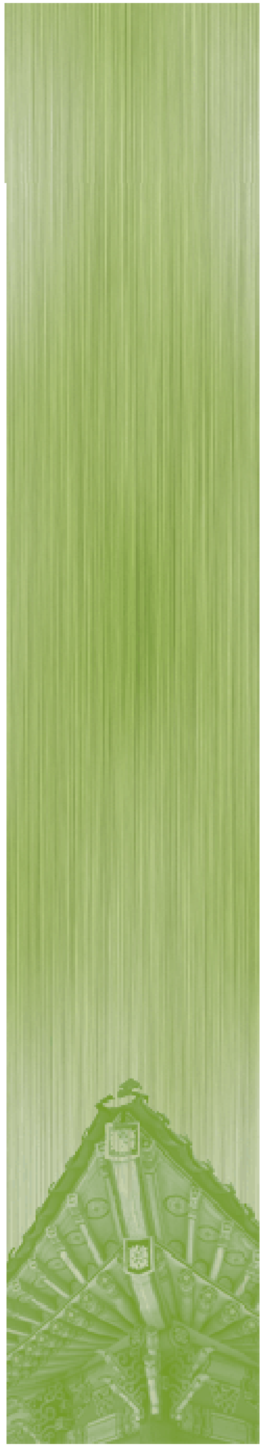


Representing and Manipulating Integers Part I

Jin-Soo Kim (jinsookim@skku.edu)
Computer Systems Laboratory
Sungkyunkwan University
http://csl.skku.edu

Unsigned Integers

- **Encoding unsigned integers**

$$B = [b_{w-1}, b_{w-2}, \ldots, b_0]$$

x = 0000 0111 1101 0011$_2$

$$D(B) = \sum_{i=0}^{w-1} b_i \cdot 2^i$$

D(x) = $2^{10} + 2^9 + 2^8 + 2^7$
$+ 2^6 + 2^4 + 2^1 + 2^0$
= 1024 + 512 + 256 + 128
+ 64 + 16 + 2 + 1
= 2003

- **What is the range for unsigned values with *w* bits?**

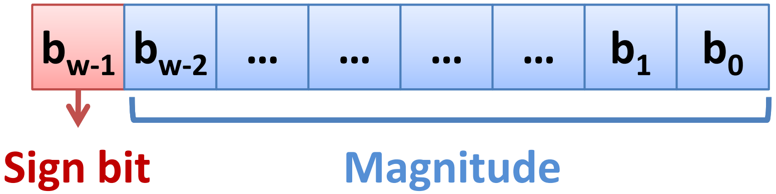
Signed Integers (1)

- **Encoding positive numbers**
  - Same as unsigned numbers

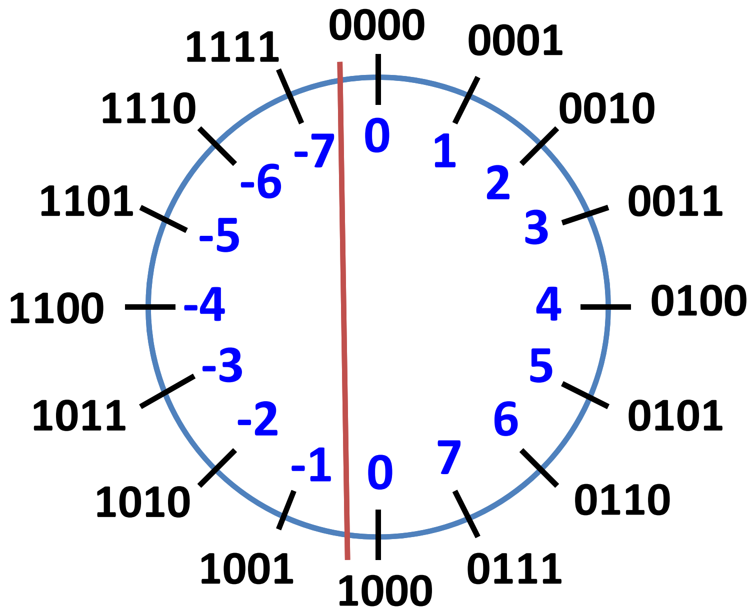
- **Encoding negative numbers**
  - Sign-magnitude representation
  - Ones' complement representation
  - Two's complement representation

Signed Integers (2)

- **Sign-magnitude representation**



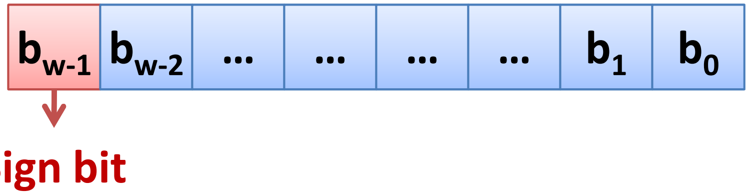
$$S(B) = (-1)^{b_{w-1}} \cdot \left(\sum_{i=0}^{w-2} b_i \cdot 2^i \right)$$



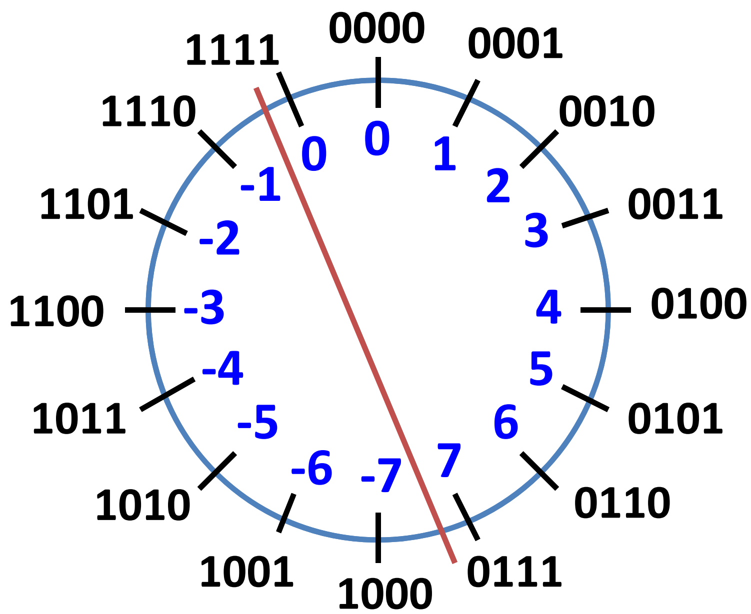
- Two zeros
  - [000…00], [100…00]
- Used for floating-point numbers

Signed Integers (3)

- **Ones' complement representation**



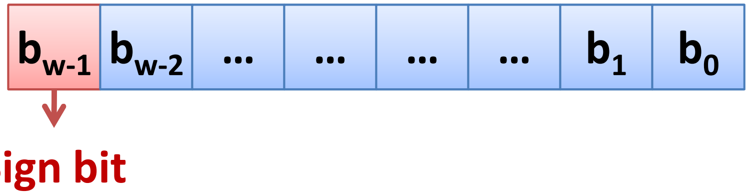
$$O(B) = -b_{w-1}(2^{w-1} - 1) + \left(\sum_{i=0}^{w-2} b_i \cdot 2^i \right)$$



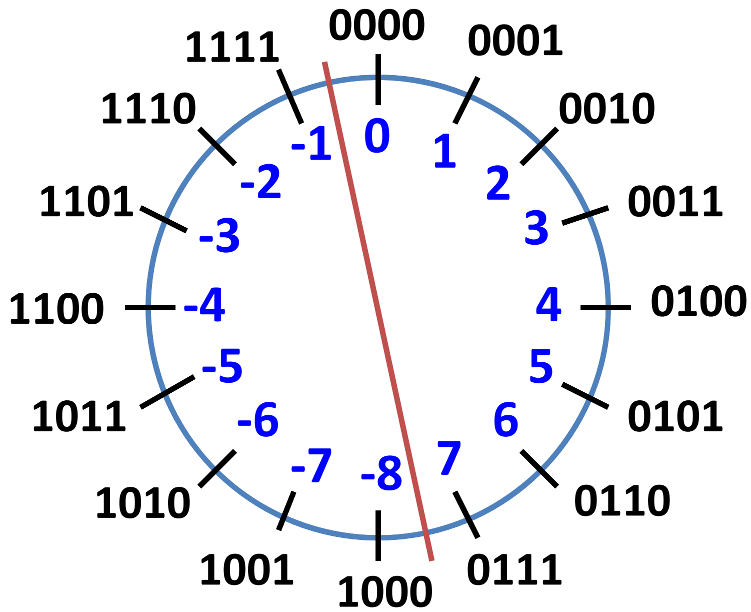
  - Easy to find ***–n***
  - Two zeros
    - [000…00], [111…11]
  - No longer used

Signed Integers (4)

- **Two's complement representation**



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



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

Signed Integers (5)

- **Two's complement representation (cont'd)**
  - Following holds for two's complement

~x + 1 == -x

  - Complement
    - Observation: **~x + x == $1111…11_2$ == -1**
  - Increment

~x + x == -1

~x + x + (-x + 1) == -1 + (-x + 1)

~x + 1 == -x

Numeric Ranges (1)

- **Unsigned values**
  - UMin = 0 [000…00]
  - UMax = $2^w - 1$ [111…11]
- **Two's complement values**
  - TMin = -2^{w-1} [100…00]
  - TMax = $2^{w-1} - 1$ [011…11]

Values for w = 16

	Decimal	Hex	Binary
UMax	65535	FF FF	11111111 11111111
TMax	32767	7F FF	01111111 11111111
TMin	-32768	80 00	10000000 00000000
-1	-1	FF FF	11111111 11111111
0	0	00 00	00000000 00000000

Numeric Ranges (2)

- **Values for different word sizes**

	w = 8	w = 16	w = 32	w = 64
UMax	255	65,535	4,294,967,295	18,446,744,073,709,551,615
TMax	127	32,767	2,147,483,647	9,223,372,036,854,775,807
TMin	-128	-32,768	-2,147,483,648	-9,223,372,036,854,775,808

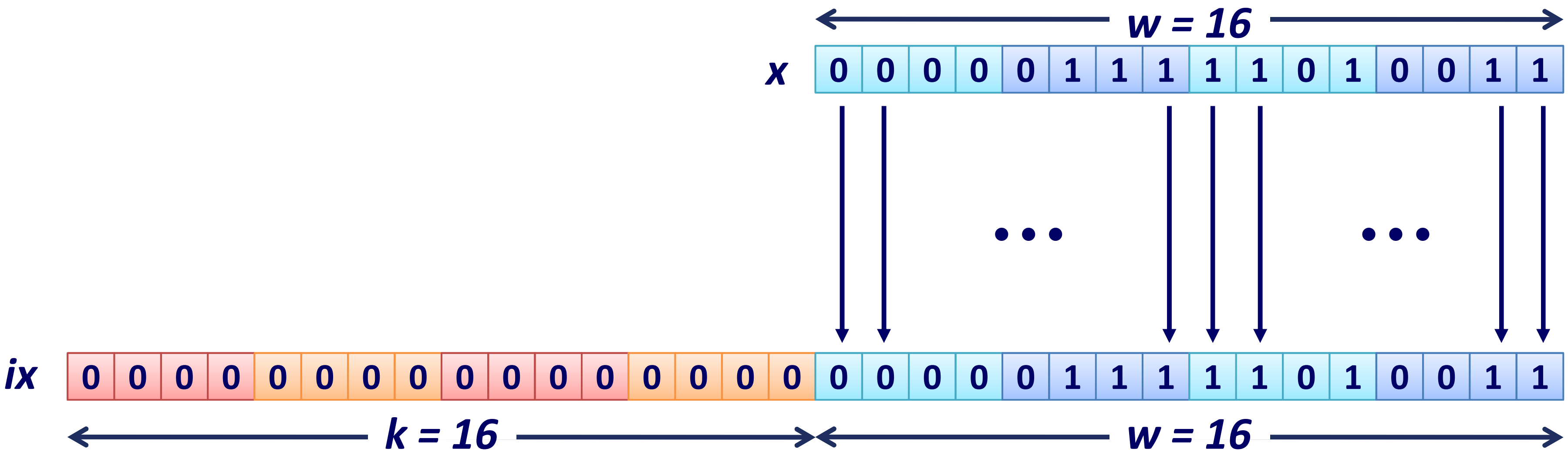
- **Observations**
  - |TMin| = TMax + 1 (Asymmetric range)
  - UMax = 2 * TMax + 1

- **In C programming**
  - `#include <limits.h>`
  - `INT_MIN, INT_MAX, LONG_MIN, LONG_MAX, UINT_MAX, …`
  - Values platform-specific

Type Conversion (1)

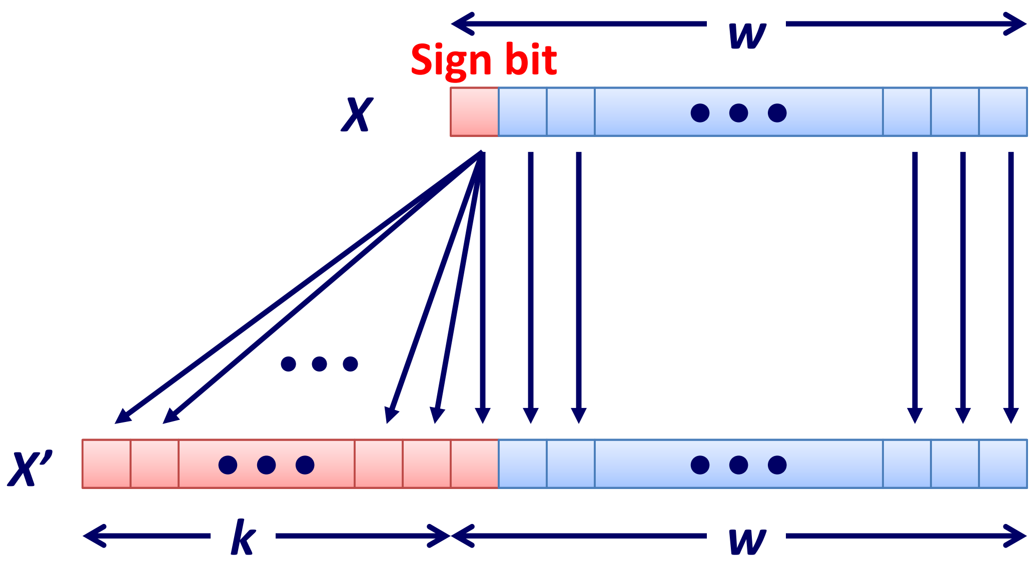
- **Unsigned: *w* bits → *w+k* bits**
  - Zero extension: just fill ***k*** bits with 0's

```
unsigned short  x  =  2003;
unsigned       ix  =  (unsigned) x;
```



Type Conversion (2)

- **Signed: *w* bits → *w+k* bits**
  - Given ***w***-bit signed integer ***x***
  - Convert it to ***w+k***-bit integer with same value
- **Sign extension**
  - Make ***k*** copies of sign bit



Type Conversion (3)

- **Sign extension example**
  - Converting from smaller to larger integer type
  - C automatically performs sign extension

```
short int x  = 2003;
int       ix = (int) x;
short int y  = -2003;
int       iy = (int) y;
```

	Decimal	Hex	Binary
x	2003	07 D3	00000111 11010011
ix	2003	00 00 07 D3	00000000 00000000 00000111 11010011
y	-2003	F8 2D	11111000 00101101
iy	-2003	FF FF F8 2D	11111111 11111111 11111000 00101101

Type Conversion (4)

- **Unsigned & Signed: *w+k* bits → *w* bits**
  - Just truncate it to lower ***w*** bits
  - Equivalent to computing ***x*** mod ***2^w***

```
unsigned int    x  = 0xcafebabe;
unsigned short  ix = (unsigned short) x;
int             y  = 0x2003beef;
short           iy = (short) y;
```

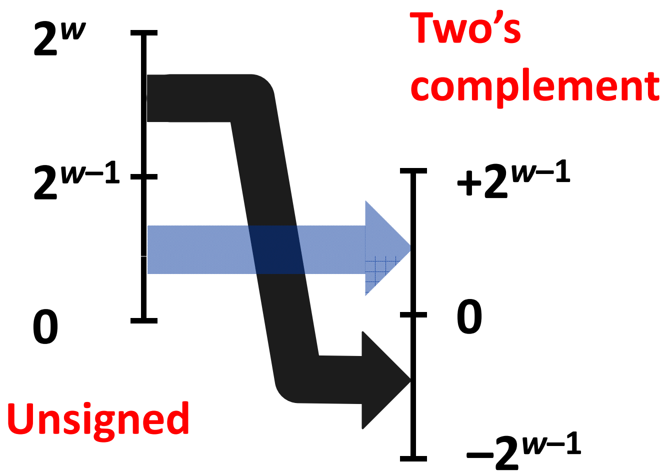
	Decimal	Hex	Binary
x	3405691582	CA FE BA BE	11001010 11111110 10111010 10111110
ix	47806	BA BE	10111010 10111110
y	537116399	20 03 BE EF	00100000 00000011 10111110 11101111
iy	-16657	BE EF	10111110 11101111

Type Conversion (5)

- **Unsigned → Signed**
  - The same bit pattern is interpreted as a signed number

$$U2T_w(x) = \begin{cases} x, & x < 2^{w-1} \\ x - 2^w, & x \geq 2^{w-1} \end{cases}$$

```
unsigned short x  = 2003;
short          ix = (short) x;
unsigned short y  = 0xbabe;
short          iy = (short) y;
```



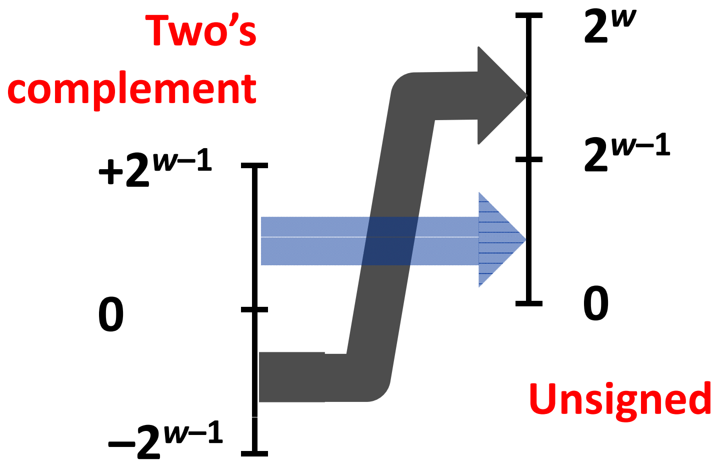
	Decimal	Hex	Binary
x	2003	07 D3	00000111 11010011
ix	2003	07 D3	00000111 11010011
y	47806	BA BE	10111010 10111110
iy	-17730	BA BE	10111010 10111110

Type Conversion (6)

- **Signed → Unsigned**
  - Ordering inversion
  - Negative → Big positive

$$T2U_w(x) = \begin{cases} x + 2^w, & x < 0 \\ x, & x \geq 0 \end{cases}$$

```
short          x = 2003;
unsigned short ix = (unsigned short) x;
short          y = -2003;
unsigned short iy = (unsigned short) y;
```



	Decimal	Hex	Binary
x	2003	07 D3	00000111 11010011
ix	2003	07 D3	00000111 11010011
y	-2003	F8 2D	11111000 00101101
iy	63533	F8 2D	11111000 00101101

Type Casting in C (1)

- **Constants**
  - By default, considered to be signed integers
  - Unsigned if have "U" or "u" as suffix
    - 0U, 12345U, 0x1A2Bu
- **Type casting**
  - Explicit casting
  - Implicit casting via
    - Assignments
    - Procedure calls

```
int         tx, ty;
unsigned    ux, uy;
tx = (int) ux;
uy = (unsigned) ty;
```

```
int f(unsigned);
tx = ux;
f(ty);
```

Type Casting in C (2)

- **Expression evaluation**
  - If mix unsigned and signed in single expression, signed values implicitly cast to **unsigned**.
  - Including comparison operations <, >, ==, <=, >=

Expression	Type	Evaluation
0 == 0U	unsigned	True
-1 < 0	signed	True
-1 < 0U	unsigned	?
-1 > -2	signed	?
(unsigned) -1 > -2	unsigned	?
2147483647 > -2147483647-1	signed	?
2147483647U > -2147483647-1	unsigned	?
2147483647 > (int) 2147483648U	signed	?

Type Casting in C (3)

- Example 1-1

```
int main ()
{
        unsigned i;
        for (i = 10; i > 0; i--)
                printf ("%u\n", i);
}
```

- Example 1-2

```
int main ()
{
        unsigned i;
        for (i = 10; i >= 0; i--)
                printf ("%u\n", i);
}
```

Type Casting in C (4)

- **Example 2**

```
int sum_array (int a[], unsigned len)
{
        int  i;
        int  result = 0;

        for (i = 0; i <= len - 1; i++)
                result += a[i];

        return result;
}
```

Type Casting in C (5)

- **Example 3-1**

```
void copy_mem1 (char *src, char *dest, unsigned len)
{
        unsigned i;
        for (i = 0; i < len; i++)
              *dest++ = *src++;
}
```

- **Example 3-2**

```
void copy_mem2 (char *src, char *dest, unsigned len)
{
        int   i;
        for (i = 0; i < len; i++)
              *dest++ = *src++;
}
```

Type Casting in C (6)

- **Example 3-3**

```
void copy_mem3 (char *src, char *dest, unsigned len)
{
        for (; len > 0; len--)
                *dest++ = *src++;
 }
```

- **Example 3-4**

```
void copy_mem4 (char *src, char *dest, unsigned len)
{
        for (; (int) len > 0; len--)
                *dest++ = *src++;
 }
```

Type Casting in C (7)

- **Example 4**

```
#include <stdio.h>

int  main  ()
{
        unsigned char     c;

        while  ((c  =  getchar())  !=  EOF)
                putchar  (c);
}
```

Type Casting in C (8)

- **Lessons**
  - There are many tricky situations when you use unsigned integers – hard to debug
  - Do not use just because numbers are nonnegative
  - Use only when you need collections of bits with no numeric interpretation ("flags")
  - Few languages other than C support unsigned integers