Chapter 2

Bit, Bytes, and Data Types

2.1 Bit Models

It may seem strange, but in computers, there are actually several different types of "255"s.

- *A* `char` 255.
- *An* `int` 255.
- *A* `float` 255.
- *A* `double` 255.
- *An* `unsigned int` 255.
- *A* `signed short int` 255.
- *A* `long double` 255.

All values in a computer are stored as binary numbers (i.e. groups of bits), but how many and what bits may be very different.

Type Sizes (32-bit architecture)

sizeof.c

Doubles for 64-bit arch

```
printf("     Type\tBytes\tBits\n");
print_range("    short int \t"
print_range("          int \t"
print_range("        int * \t"
print_range("     long int \t"
print_range("   long int * \t"
print_range("   signed int \t"
print_range(" unsigned int \t"
printf("\n");
print_range("        float \t"
print_range("      float * \t"
print_range("       double \t"
print_range("     double * \t"
print_range("  long double \t"
printf("\n");
print_range("  signed char \t"
print_range("         char \t"
print_range("       char * \t"
print_range("unsigned char \t", sizeof(unsigned char));
```

```
Type          Bytes    Bits
    short int   2        16
          int   4        32
        int *   4        32
     long int   4 8    64 32
   long int *   4        32
   signed int   4        32
 unsigned int   4        32

        float   4        32
      float *   4        32
       double   8        64
     double *   4        32
  long double   12 16 128 96

  signed char   1        8
         char   1        8
       char *   4        32
unsigned char   1        8
```

Bit Models

bittypes.c

```
int main(void)
{
  c = i = f = d = ui = ssi = ld = 255;

  print_bits("c = ", &c, sizeof(char));
  print_bits("i = ", &i, sizeof(int));
  print_bits("ssi = ", &ssi, sizeof(signed short int));
  print_bits("f = ", &f, sizeof(float));
  print_bits("d = ", &d, sizeof(double));
  print_bits("ld = ", &ld, sizeof(long double));
}
```

```
c = 11111111
i = 00000000 00000000 00000000 11111111
ssi = 00000000 11111111
f = 01000011 01111111 00000000 00000000
d = 01000000 01101111 11100000 00000000 00000000 00000000 00000000
00000000
ld = 00000000 00000000 01000000 00000110 11111111 00000000 00000000
00000000 00000000 00000000 00000000 00000000
```

Different Bit Models

Each binary number below represents 255 in a different format.

```
c =     11111111

i =     00000000000000000000000011111111

ssi =   0000000011111111

f =     01000011011111110000000000000000

d =     01000000011011111110000000000000
        00000000000000000000000000000000

ld =    00000000000000000100000000000110
        11111111000000000000000000000000
        00000000000000000000000000000000
```

Integer Bit Models

Magnitude Only

The position of the bit determines its value. The sum of the bits determines the number's value.

bit	7	6	5	4	3	2	1	0
value	128	64	32	16	8	4	2	1

ex. $\boxed{1}\boxed{00}\boxed{1}\boxed{1}\boxed{00}\boxed{1}$ =

128 + 0 + 0 + 16 + 8 + 0 + 0 + 1

= 153

Hexadecimal and Octal Codes

Since binary numbers can be long and cumbersome to write. It can be advantageous to right them in hexidecimal notation which is simply a compression code for binary.

In C, binary numbers (constants) can be input and output as hexidecimal (or octal) numbers. C uses the syntax of a leading `0x` to denote hex constants and a leading `0` to denote octal constants.

The Hexadecimal Code

binary	hex	binary	hex	binary	hex
0000	0x0	0110	0x6	1100	0xC
0001	0x1	0111	0x7	1101	0xD
0010	0x2	1000	0x8	1110	0xE
0011	0x3	1001	0x9	1111	0xF
0100	0x4	1010	0xA		
0101	0x5	1011	0xB		

ex. 1001100110011001

= 1001 1001 1001 1001 = 0x9999

hex.c

Integer Bit Models
Signed Magnitude

Signed Magnitude employs a *sign bit* to determine whether the number is positive or negative.

The sign bit is the most significant—leftmost—bit.

bit	7	6	5	4	3	2	1	0
value	+/-	64	32	16	8	4	2	1

ex. *what does* 10011001 *equal?*

− 0 + 0 + 16 + 8 + 0 + 0 + 1

= − 25

What does* 00011001 *equal?

What does* 10000000 *equal?

Integer Bit Models
One's-Complement

The One's-Complement also contains an uppermost sign bit. If the number is positive, we simply create the binary value just as we did in the signed magnitude case.

If the number is negative, however, the one's complement is formed by subtracting the magnitude of the number from an equivalent number of all 1's, and adding a sign bit of 1.

Since 1 − 0 is 1 and 1 − 1 is 0, we can form the ones complement by simply complementing each bit.

ex. ***How do we write −7 ?***

7 = 00000111, so

−7 = 11111000

What does 00000111 + 11111000 equal?

What does 10000000 equal?

Integer Bit Models
Two's-Complement

Two's-Complement is called a *complement* because the number "complements" its negative (i.e. positive) value.

That is, when a value and its negative (two's complement) are added together, they form 2^n.

For example: Add `103` to −`103`

`103 = 01100111, -103 = 10011001`

```
  01100111
+ 10011001
----------
 100000000
```

$100000000 = 2^8 = 256 = 0$

How does this equal 0?

Creating Two’s-Complement Numbers

Like Signed Magnitude and One’s-Complement, Two’s-Complement also adds a *sign bit*—most significant bit—to determine whether the number is positive or negative.

If the sign positive, we simply form the number just as we did in the signed magnitude and one’s-complement cases.

If the sign is negative, however, we have to produce the two’s-complement of the number. This can be done in several different ways. The two simplest methods are shown here.

Creating Two's-Complement Numbers

The first method is to first find the one's complement of the number by complementing each bit, and then adding one to the result.

ex. *What is* -7 *in two's-complement form?*

$$7 = 00000111, \text{ so}$$
$$-7 = 11111000 + 1$$
$$= 11111001$$

Notice $7 + -7 = \begin{array}{r} 00000111 \\ +\ 11111001 \\ \hline \end{array}$
$$= 100000000$$

What does 100000000 *equal?*

Creating Two's-Complement Numbers

The second method is to simply start copying the bits from right (*least significant*) to left (*most significant*) until the first 1 is encountered.

Then, after writing down the first 1, complement each bit.

ex. ***How do we write −16 in two's-complement form ?***

First, create 16.

$$16 = 2^4 = 00010000$$

Then, write down 0's till first 1 is encountered.

10000

Finally, invert the rest of the bits to give.

11110000

00001111

What does 11110001 equal?

What does 10000000 equal?

Two's-Complement Program

twoscomp.c

```
print_bits(char *s, void *mem, unsigned char len)
{
  /* … */
}
/* … */
int main(void)
{  char a, b, c, i, j, k;

   a = 23;     print_bits("a = ",&a, sizeof(a));
   b = 17;     print_bits("b = ",&b, sizeof(b));
   c = a + b;  print_bits("c = ",&c, sizeof(c));

   printf("\n");

   i = -23;    print_bits("i = ",&i, sizeof(i));
   j = -17;    print_bits("j = ",&j, sizeof(j));
   k = i + j;  print_bits("k = ",&k, sizeof(k));
}
```

```
 a = 23 = 00010111
 b = 17 = 00010001
 c = 40 = 00101000

i = -23 = 11101001
j = -17 = 11101111
k = -40 = 11011000
```

Byte Ordering

Since *most* computers today are Byte-addressable, each address in the CPU corresponds to exactly eight bits.

Then how does a computer store and retrieve numbers which require more than a Byte, such as a name or a 32-bit integer?

To be efficient, the computer should only put one address on the bus. But the 32-bit integer takes up *four* addresses.

Computers specify the lowest address to reference a block of memory. For example, strings of characters are stored starting with the first character at the lowest address.

For example: `sprintf((char *)0x0100, “TIGERS!”);`

Address	Data 0	1	2	3
⋮				
0x0100	84	73	71	69
0x0104	82	83	33	00
0x0108	00	00	00	00
0x010C	00	00	00	00
⋮				

=

Address	Data 0	1	2	3
⋮				
0x0100	'T'	'I'	'G'	'E'
0x0104	'R'	'S'	'!'	'\0'
0x0108	00	00	00	00
0x010C	00	00	00	00
⋮				

Byte Ordering

But how is a four-byte *integer* stored?

There are two logical choices:

- **Store the most significant Byte at the most significant (highest) address, or**
- **Store the least significant Byte at the most significant address.**

Let's say `a` is located at address 0x0108 and we execute the instruction:

```
a = 0x001234AB;
```

Address	Data 0	1	2	3
⋮				
0x0100	'T'	'I'	'G'	'E'
0x0104	'R'	'S'	'!'	'\0'
0x0108	00	12	34	AB
0x010C	00	00	00	00
⋮				

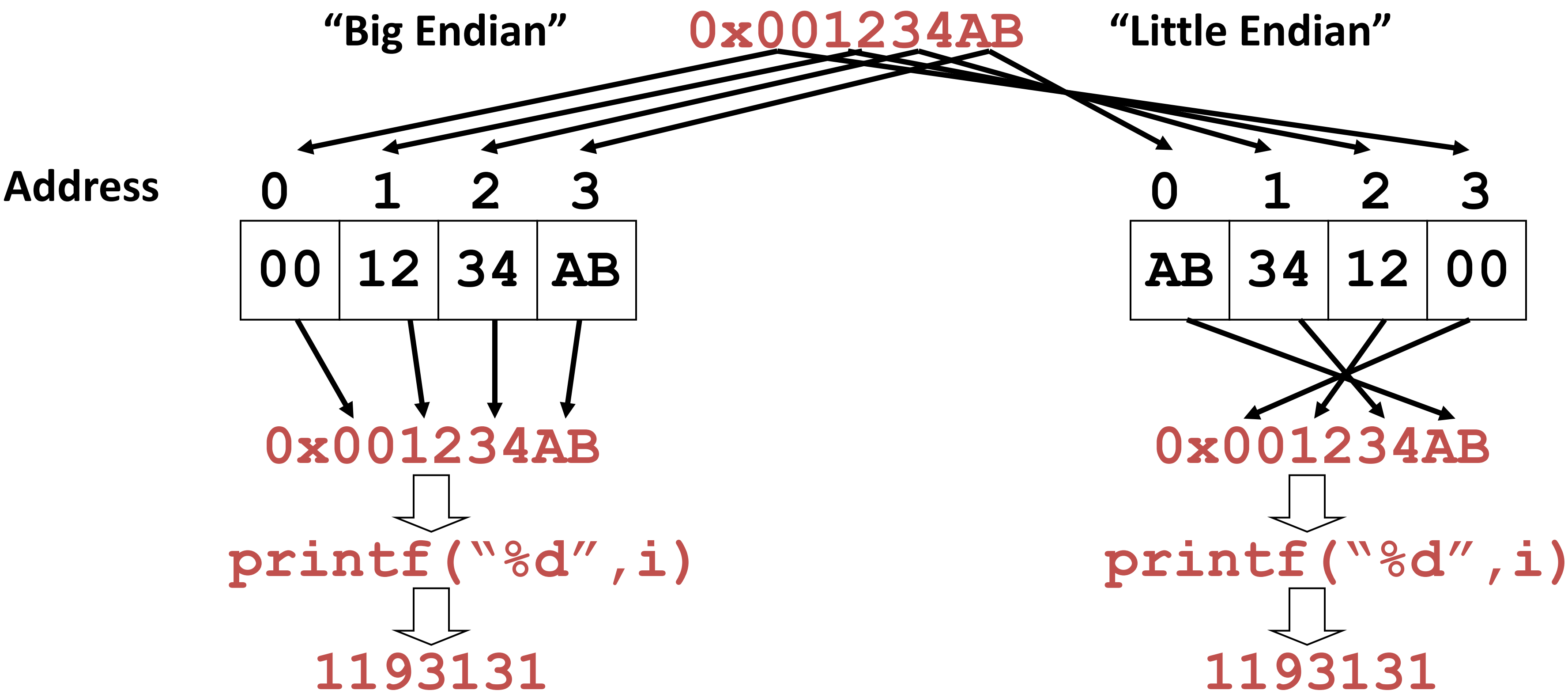
"Big Endian"

Address	Data 0	1	2	3
⋮				
0x0100	'T'	'I'	'G'	'E'
0x0104	'R'	'S'	'!'	00
0x0108	AB	34	12	00
0x010C	00	00	00	00
⋮				

"Little Endian"

Byte Ordering

Consider storing a four-byte integer in both types of computers:

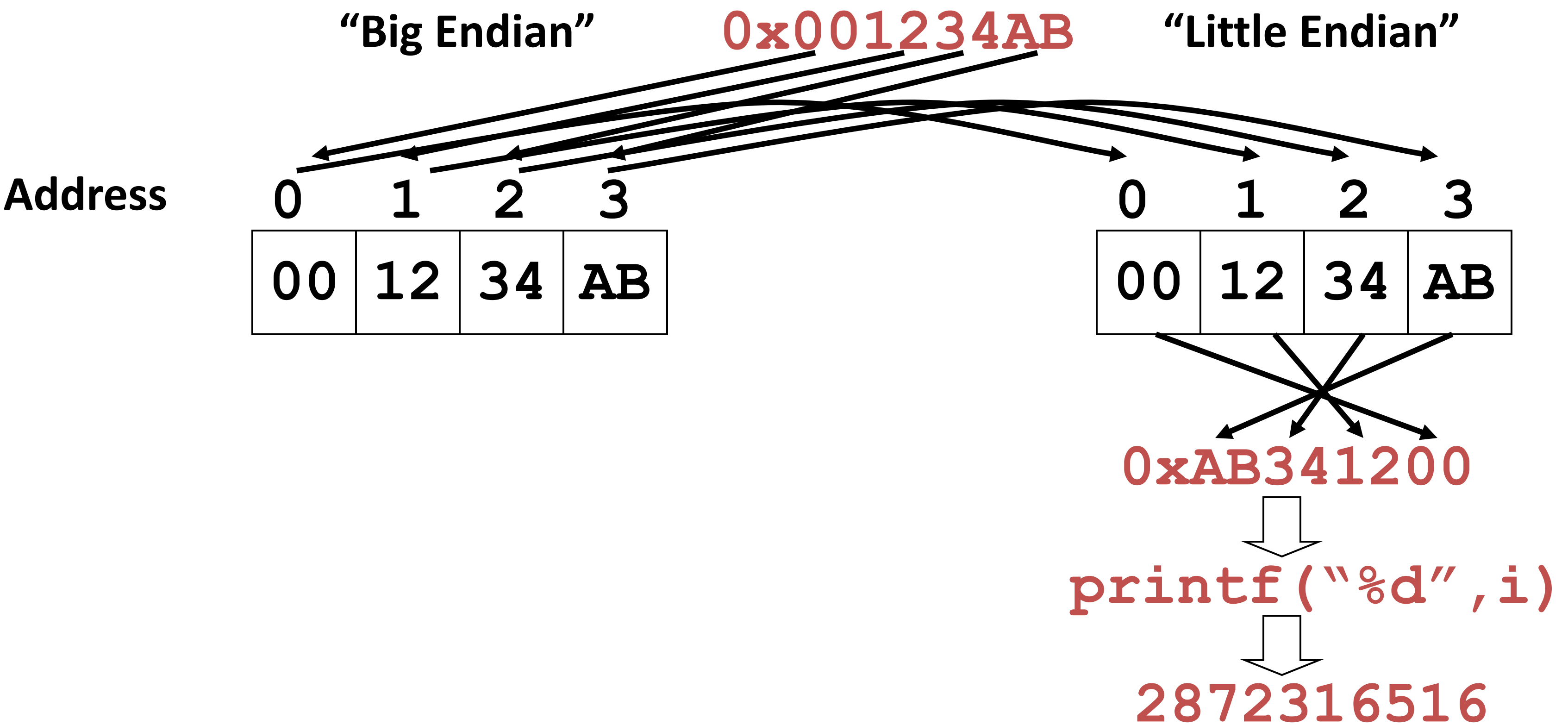


Now, consider calling `printf("%d", i)` to print both numbers:

They both work because the `printf()` function written for each computer's hardware "knows" how each number is stored.

Byte Ordering

Now consider a byte-to-byte transfer from one type system to the other:



Since the number was transferred from one machine to another by block, the number is *stored*–and thus printed—incorrectly.

Decimal Bit Models
Fixed Point

Fixed point is like magnitude bit model except that part of the bits (a fixed number) are used for the fractional part.

bit	7	6	5	4	3	2	1	0
value	16	8	4	2	1	½	¼	$^1/_8$

ex. 10011001 =

16 + 0 + 0 + 2 + 1 + 0 + 0 + $^1/_8$

= 19.125

Decimal Bit Models

Floating Point

Floating Point (as the name implies) doesn't use a *fixed* number of bits for the integer and fractional parts of the number. This allows for very large and very small numbers. An IEEE standard says:

$$x = +/- \; 1.f \cdot 2^{e-127}$$

bit	31	30-23	22-0
value	+/-	*e*	*f*

Example: ***What does*** `0xABC00123` ***in memory represent?***

Floating Point Example

$$x = +/- \ 1.f \cdot 2^{e-127}$$

bit	31	30-23	22-0
value	+/-	*e*	*f*

Example: 0xABC00123

1010 1011 1100 0000 0000 0001 0010 0011

10101011110000000000000100100011

$$1.10000000000000100100011 \cdot 2^{01010111-127}$$

$$(2^{0} + 2^{-1} + 2^{-15} + 2^{-18} + 2^{-22} + 2^{-23}) \cdot 2^{87-127}$$

$$(2^{0} + 2^{-1} + 2^{-15} + 2^{-18} + 2^{-22} + 2^{-23}) \cdot 2^{-40}$$

$$(2^{-40} + 2^{-41} + 2^{-55} + 2^{-58} + 2^{-62} + 2^{-63}) \cdot 2^{0}$$

$$= 1.364273 \cdot 10^{-12}$$

Creating Floating Point Numbers

1 – Write the sign bit: `0` = non-negative, `1` = negative.

2 – Write the magnitude of number in fixed point binary.

3 – Normalize to a number between `1.0` and `1.111111`...
i.e. Move the decimal place to one digit from the left.

4 – Take *`f`* as the value to the right of the decimal place, padded with zeroes.

5 – Add `127` to the exponent and use as *`e`*.

Floating Point Example

Store the number $x = 12.3456 \cdot 10^{-3}$ **as a floating point number.**

1. **Bit 31 = 0, since the number is positive.**
2. $12.3456 \cdot 10^{-3}$

   1 2345 6789 0123 4567

   = 0.0000 0011 0010 1001 0001 0100

   1100 11~~00 0011 1111~~...

   8901 23
3. $= 1.10010100100010100110011 \cdot 2^{-7}$
4. $f =$ 100 1010 0100 0101 0011 0011.
5. $e = -7 + 127 = 120 =$ 01111000 = 011 1100 0

x = 00111100 01001010 01000101 00110011

Floating Point Program

float.c

```
int main(void)
{   float f;
    int *ptri;

    ptri = (int *)&f;

    *ptri = 0xABC00123;

    print_bits("f = ",&f, sizeof(f));
    printf("f = %d\n", *(int *)&f);
    printf("f = 0x%X\n", *(int *)&f);
    printf("f = %f\n", f);
    printf("f = %e\n\n", f);

    f = 12.3456E-3;
    print_bits("f = ",&f, sizeof(f));
    printf("f = %d\n", *(int *)&f);
    printf("f = 0x%X\n", *(int *)&f);
    printf("f = %f\n", f);
    printf("f = %e\n", f);
}
```

Floating Point Precision

precision.c

precision2.c

```
double d1, d2; float f1, f2;

int main(void)
{ int i;

  f1 = d1 = 1.2345678901234567890;
  f2 = d2 = 1.0/3.0;

  for (i=0; i<40; i++)
  { f1 *= 2;  f1 -= (int)f1;
    printf("f1 = %0.20f\n", f1);
  }

  /* … */

  for (i=0; i<40; i++)
  { d1 *= 2;  d1 -= (int)d1;
    printf("d1 = %0.20lf\n", d1);
  }

  /* … */
```

Text Bit Models
ASCII Characters

ASCII (American Standard Code for Information Interchange) was developed to produce a standard for storing and printing English text.

Therfore, it contains a code for all characters on a keyboard, along with control characters needed to communicate with other devices.

http://www.asciitable.com/

ASCII Example

How do we write `Go Tigers!” *in ASCII?*

`0x47 0x6F 0x20 0x54 0x69 0x67 0x65 0x72 0x73 0x21`

What is `01000011` `01010101` *in memory if stored as ASCII?*

`0x43 0x55`

`C U`

What does `'C'` *mean in a C progam?*

ASCII Program

```
void print_c(void)
{  printf("c = %c = %d = 0x%X", c, c, c);
   print_bits(" = ",&c, sizeof(c));
   printf("\n");
}
int main(void)
{
   c = 65;  print_c();
   for (c=7; c<=13; c++) print_c();
   c = '6';  print_c();
   c = '\'';  print_c();
   c = '\\';  print_c();
   uc = 200;  print_c();
   c = 200; print_c();
}
```

ASCII.c

2.2 Bitwise Operations

The *cell size* of a typical computer is a *byte*. That is, a group of eight bits. Therefore, each address specifies a byte of data and the smallest data type in C is a `char` which is eight bits.

How then do we access and modify single bits in memory using C instructions?

Bitwise Operators

C allows for several "bit-wise" operators. These operators perform Boolean operations *bit-by-bit* on integer values.

`&`	`AND`	**Bit is `1` only if both bits are `1`.**
`|`	`OR`	**Bit is `1` if either bit is `1`.**
`^`	`XOR`	**Bit is `1` if either bit is `1`, but not both.**
`~`	`NOT`	**Bits are the complements of the operand's bits.**
`<<`	Left Shift	**Shifts bits to the left.**
`>>`	Right Shift	**Shifts bits to the right.**

Bitwise Operator Examples

```
a=10100101 = 0xA5, b=11110000 = 0xF0

a & b = 10100101
        11110000
        --------
        10100000 = 0xA0

a | b = 10100101
        11110000
        --------
        11110101 = 0xF5

a ^ b = 10100101
        11110000
        --------
        01010101 = 0x55

 ~a   = ~10100101  = 01011010 = 0x5A
```

Bitwise Operator Program

bitwise.c

```
int main(void)
{   UCHAR a, b, c;

    a = 0xFA;
    b = 0x5F;

 …

    printf("\nc = a & b\n");
    c = a & b;
    printf("c = %3d = 0x%02X", c, c);
    print_bits(" = ",&c, sizeof(c));

    printf("\nc = a | b\n");
    c = a | b;
    printf("c = %3d = 0x%02X", c, c);
    print_bits(" = ",&c, sizeof(c));

    printf("\nc = a ^ b\n");
    c = a ^ b;
    printf("c = %3d = 0x%02X", c, c);
    print_bits(" = ",&c, sizeof(c));
```

Bitwise Operator Examples

```
a=10100101 = 0xA5, b=11110000 = 0xF0

a << 3 = 10100101 << 3
       = 10100101 << 3
       = 00101000
       = 00101000 = 0x28

b >> 6 = 11110000 >> 6
       = 11110000 >> 6
       = 00000011
       = 00000011 = 0x03
```

What are the new bit values when shifting left or right?

Bitwise Shifting

New lower-order bits when shifting left are always 0's.

When shifting right, however, the new high-order bits will be 1's if the variable is *signed*, and the sign bit is 1.

```
signed int a = 12, b = -12;
unsigned int c = 12, d = -12;

int main(void)
{
  print_data();

  printf(">>=3\n");
  a >>= 3;  b >>= 3;  c >>= 3;  d >>= 3;

  print_data();

  printf("<<=3\n");
  a <<= 3;  b <<= 3;  c <<= 3;  d <<= 3;

  print_data();
}
```

shift.c

What happened here?

Bitmask Operations

Since most machines are Byte addressable, single bits in memory cannot be addressed (*referenced*). To modify single bits, these bitwise operators can be used.

Since `X OR 0 = X` and `X OR 1 = 1`, the OR operation can be used to set certain bits, but leave others in the same Byte unmodified.

Since `X AND 0 = 0` and `X AND 1 = X`, the OR operation can be used to clear (*reset*) certain bits, but leave others in the same Byte unmodified.

Bitmask Examples

To set or clear bits, a bitmask (denoted in *hexadecimal* notation) is used to reference which bits should be modified (and which should remain unchanged).

To set the highest and lowest order bits of a `char` named `a`, the bitmask `0x81` should be used.

```
a = 0x14;       // a = 00010100
a = a | 0x81; // a = 10010101 = 0x95
```

To clear the highest and lowest order bits of `a`, the complement of bitmask `0x81` should be used.

```
a = 0xE3;        // a = 11100011
a = a & ~0x81; // a = 01100010 = 0x62
```

Bitmask Examples

To check the status of bits, these same operations can be used.

To check and see if bit 2 of a `char` named `a` is `TRUE`, we can use the bitmask `0x04` .

```
if ((a & 0x04) != 0) printf("Motor On");
```

To check to see if the four highest bits of `char a` are set we can use the bitmask `0xF0` .

```
if ((a & 0xF0) == 0xF0) printf("All Fans On");
```

Manipulating Bits

Operation	C Code
Set Nth bit	`x = x | ( 1 << N);`
Clear Nth bit	`x = x & ( ~ ( 1 << N ));`
Read Nth bit	`( x & ( 1 << N ) ) >> N;`

Recall CS notation: least significant bit is numbered 0

bitmask.c

bitmask_neg.c

```
char a;
int i;
a = 17;
a |= 1 << 3;          // set 3rd bit
printf("a = %d\n", a);

a &= ~ (1 << 4);      // clear 4th bit
printf("a = %d\n", a);

// print bits
printf("a = ");
for (i = 7; i >= 0; i--)
     printf("%d", (a & (1 << i)) >> i);    // read i-th bit
printf("\n");
```

Extend to work with block of bits

Bitmask Operations Program
bitmask2.c

```
int main(void)
{   UCHAR a;
    a = 0x00;
    printf("\na = %3d = 0x%02X", a, a); print_bits(" = ",&a, sizeof(a));
    printf("a = a | (BIT_6 | BIT_4 | BIT_2 | BIT_0);\n");
    a = a | (BIT_6 | BIT_4 | BIT_2 | BIT_0);
    printf("a = %3d = 0x%02X", a, a); print_bits(" = ",&a, sizeof(a));
    a = 0xFF;
    printf("\na = %3d = 0x%02X", a, a); print_bits(" = ",&a, sizeof(a));
    printf("a = a & ~(BIT_7 | BIT_5 | BIT_3 | BIT_1);\n");
    a = a & ~(BIT_7 | BIT_5 | BIT_3 | BIT_1);
    printf("a = %3d = 0x%02X", a, a); print_bits(" = ",&a, sizeof(a));
    printf("\n");
    for (a=0; a<0xFF; a++)
    {   if ((a & 0xAA) == 0xAA)
        {   printf("Match:  ");
            printf("a = %3d = 0x%02X", a, a); print_bits(" = ",&a, sizeof(a));
        }
    }
    getchar();
}
```

But hard to use in loops

Functions vs. Tables

Up till now, I used a lookup table to print bits since we had not covered bit masking operations yet.

```
char *HexToBin[16] =
{ "0000", "0001", "0010", "0011", "0100", "0101", "0110", "0111",
  "1000", "1001", "1010", "1011", "1100", "1101", "1110", "1111"
};

void print_bits(char *text, void *mem, UCHAR len)
{ UCHAR *addr; UCHAR i; unsigned int l;

  // *** Print Binary Value of Memory *** //
  printf(text);
  for (addr=(UCHAR *)mem + len - 1; addr>=(UCHAR *)mem; addr--)
  {  printf("%4s", &HexToBin[*addr>>4][0]);
     printf("%4s ", &HexToBin[*addr % 16][0]);
}
  printf("\n");
}
```

Functions vs. Tables 2

Now consider the same function, but written with bitmasking instead...

bitmask3.c

```
void print_bits(char *text, void *mem, UCHAR len)
{ UCHAR *addr; UCHAR i; char j;

  // *** Print Binary Value of Memory *** //
  printf(text);
  for (addr=(UCHAR *)mem + len - 1; addr>=(UCHAR *)mem; addr--)
  {  for (j = 7; j>=0; j--) printf((*addr>>j) & 0x01 ? "1" : "0");
  }
  printf("\n");
}
//-------------------------------------------------------------------

int main(void)
{  UCHAR i;

   printf("Enter a character...  ");
   scanf("%c", &i);

   printf("The binary value for %c is ");
   print_bits(" = ", &i, sizeof(i));
}
```

What is the advantage and disadvantage of each method?

Both depend on Little-Endian architecture

Complicated use of void pointer
(we will study pointers in great detail later)

Adv: functions independent of type

"Real-world" Bitmask Example

Consider an example where a byte of data—an ASCII character called `c`—needs to be written to an LCD display one nibble at a time using an embedded processor with programmable I/O pins.

Unfortunately, the engineer felt the need to connect the four LCD hardware lines to pins 3, 4, 5, and 6 of the port `PortOut` instead of 0, 1, 2, and 3.

How do we write the code to write the upper nibble and then the lower nibble to the proper pins of the port, without changing the other bits, and given that we write the high order nibble first?

`LCD.c`

2.3 Memory Maps

Memory maps are listings of all the variables and their addresses for a piece of code.

For example, consider the following allocation:

```
short int si = 0x8421;
int i = 0x12345678;
float f = 16;
double d = 256;
char c = 'C';
```

memory.c

Memory Maps

```
short int si = 1000;
int i= 0x12345678;
float f = 16;
double d = 256;
char c = 'C';
```

For all homework and exam problems in this class do not consider alignment and do not reorder variables

Modern compilers often reorder memory to align to the bus size and to use memory more efficintly.

Using DevC++ the allocation above produces this memory map:

gcc produces a considerably different map

Variable	Address
si	4210704
i	4210708
f	4210712
d	4210720
c	4210728

Memory Map Example

```
si  4210704 1000
i   4210708 0x12345678
f   4210712 16
```

```
d 4210720 256
c 4210728 'C'
```

```
4210704 - 11101000
4210705 - 00000011
4210706 - 00000000
4210707 - 00000000
4210708 - 01111000
4210709 - 01010110
4210710 - 00110100
4210711 - 00010010
4210712 - 00000000
4210713 - 00000000
4210714 - 10000000
4210715 - 01000001
```

```
4210716 - 00000000
4210717 - 00000000
4210718 - 00000000
4210719 - 00000000
4210720 - 00000000
4210721 - 00000000
4210722 - 00000000
4210723 - 00000000
4210724 - 00000000
4210725 - 00000000
4210726 - 01110000
4210727 - 01000000
4210728 - 01000011
```

Are these values correct? (for DevC++ only)

Memory Map for HW and Exams

```
short int si = 0x8421;
int i = 0x12345678;
float f = 16;
double d = 256;
char c = 'C';
```

Label	Address	Value
si	400-401	1000
i	402-405	305419869
f	406-409	16.0
d	410-417	256.0
c	418	67