Low-level operations in C

C was invented as a high-level systems programming language
- Higher than assember, but still close to the machine

Binary data representation
- `int` type: two's complement
- `float` type: IEEE 754
- hexadecimal: another representation for binary

Hexadecimal constant:
- `int i = 0x1234abcd;`
- Can use upper or lower case for digits

Read or print hex values:
- `scanf ("%x", &i);`
- `printf ("%x", i);`

Manipulating individual bits:
- bitwise logical operators
- bitshift operators

Bitwise operators

Logical operators

`&&, ||` and, or

operate on entire value

```
int x = 0, y = 1;
(x && y)      value 0
(x || y)      value 1
```

May want to work with individual bits

```
int x = 2, y = 7;
```

`(x && y)` value 1

What about bits?									**value**
x	**0000**	**0000**	**0000**	**0000**	**0000**	**0000**	**0000**	**0010**	**2**
y	**0000**	**0000**	**0000**	**0000**	**0000**	**0000**	**0000**	**0111**	**7**

Bitwise and

`x & y`	**0000**	**0000**	**0000**	**0000**	**0000**	**0000**	**0000**	**0010**	**2**

Bitwise or

`x | y`	**0000**	**0000**	**0000**	**0000**	**0000**	**0000**	**0000**	**0111**	**7**

Bitwise xor (exclusive-or)

`x ^ y`	**0000**	**0000**	**0000**	**0000**	**0000**	**0000**	**0000**	**0101**	**5**

Value of a bit is 1 if only one bit is 1

Complement

~x	**1111**	**1111**	**1111**	**1111**	**1111**	**1111**	**1111**	**1101**
~y	**1111**	**1111**	**1111**	**1111**	**1111**	**1111**	**1111**	**1000**

Each bit is "flipped" to opposite value

How is this related to negative value?

Bitshift operators

`x << n` Shift bits of x left by n digits
Insert 0's on the right

`x >> n` Shift bits of x right by n digits
If unsigned or non-negative, insert 0's on left
If signed, may be system dependent

Important: x DOES NOT CHANGE!
(just like x + 2 does not change x)
`x <<= n` } change x
`x >>= n` }

x and n must be int

Examples:

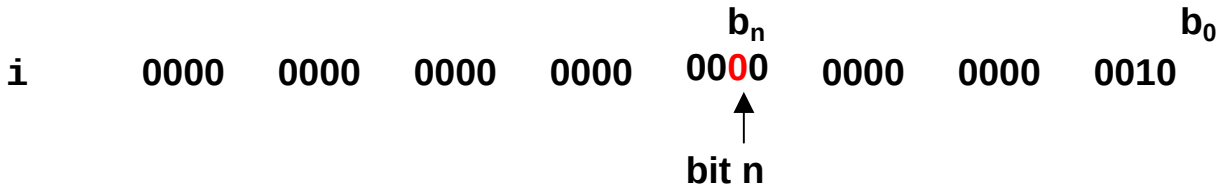
```
int x = 5;
```

x	0000	0000	0000	0000	0000	0000	0000	0101
x << 3	0000	0000	0000	0000	0000	0000	0010	1000
x >> 2	0000	0000	0000	0000	0000	0000	0000	0001

What arithmetic operations do these correspond to?

Bit operations: Test a bit

Problem: given int i, is bit n set (equal to 1)?



How can we test if this bit is 1?

We can use & operator with a "mask" variable:


```
if (i & mask)
      printf ("yes");
else
      printf ("no");
```

What is the problem with this?

We would need 32 different masks, depending on the value of n!

Answer:

```
      i  & (1 << n)        /* idiom */
```

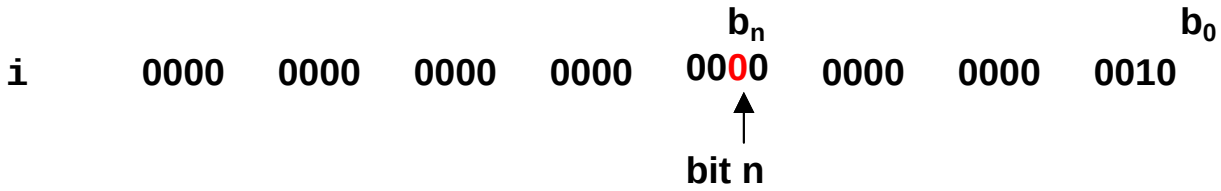
or

```
      (i >> n) & 1
```

(Shift nth bit to first location, compare to mask of 1)

Bit operations: Set a bit

Problem: given int i, set bit n to 1



How can we make sure this bit is set to1 (and not affect any other bits)?

We can use | operator with a "mask":


```
i |= mask;
```

What is the problem with this?

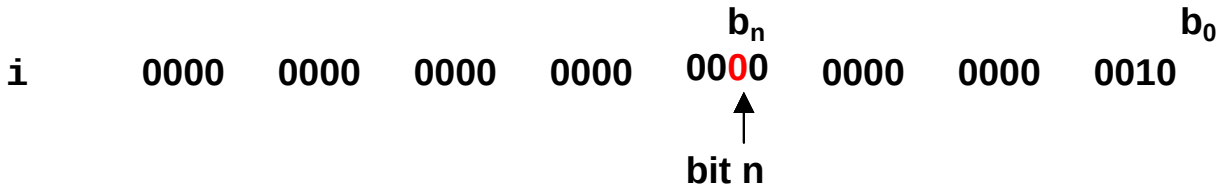
Answer:

```
i |= (1 << n);      /* idiom */
```

(Shift 1 bit n places to the left, then apply or operator to the result and i).

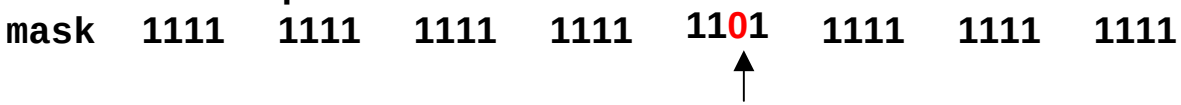
Bit operations: Clear a bit

Problem: given int i, set bit n to 0



How can we make sure this bit is 0 (and not affect any other bits)?

We can use & operator with a "mask":




```
i &= mask;
```

What is the problem with this?

Answer:

```
i &= ~(1 << n);     /* idiom */
```

Shift 1 bit n places to the left
Flip bits to get all 1's except 0 in bit n
Apply & operator to clear bit n

Cast operator

Problem: given int i, access a particular byte

```
int i = 0x1234abcd;
```

Is this big-endian or little-endian? How can we tell?

By definition, if we look at the bits in i, the leftmost bits are

0001 0010 (big- or little-endian)

Value is independent of byte order

Another way to look at 4 bytes:

```
char c[4];
```

What about

```
c[0] = (char) i;
```

Doesn't do what we want, because cast converts value from `int` to `char`

What type of value is `char` really?

Need to look at the individual memory locations as `char`

How do we refer to memory locations?

```
int * iptr = &i;
char * byte_ptr = (char *) &i;
```

Converts pointer, not data

Now we can increment `byte_ptr` to look at each byte within the `int`

Address	big endian Contents	little endian Contents
1000	12	cd
1001	34	ab
1002	ab	34
1003	cd	12
1004		
1005		
1006		
1007		

Endian test

```
main ()
{
   int i = 0x1234abcd, n;
   unsigned char * byte = (unsigned char *) &i;

   for (n = 0; n < 4; n++)
      printf ("%x ", *(byte + n));

   return 0;
}
```

Address	big endian Contents	little endian Contents
1000	12	cd
1001	34	ab
1002	ab	34
1003	cd	12
1004		
1005		
1006		
1007		

Why unsigned char?

Output (IBM PC):

`cd ab 34 12`

Output (Sun):

`12 34 ab cd`

Test bits: float

How would we test the bits of a float value?

```
float f = 1024;
int n;
for (n = 0; n < 32; n++)
      if (f  & (1 << n))
         printf ("1");
      else
         printf ("0");
```

Compile error!

Need a way to look at f as if it were int:

```
float f = 1024;
int n, i;
i = (int) f;
for (n = 0; n < 32; n++)
      if (i  & (1 << n))
         printf ("1");
      else
         printf ("0");
```

Can't use cast on f: value is converted to int.

Cast a pointer:

```
int * iptr = (int *)&f;
i = *iptr;
```