Bit Level Operators in C

C Programming and Software Tools

N.C. State Department of Computer Science

Computer Science
NC STATE UNIVERSITY

Contents

- “Bit Twiddling”
- Bit Operators
- Differences between Logical and Bitwise Operators
- Shift Operators
- Examples

Hexadecimal reminder

Hex digit	Binary	Decimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
A	1010	10
B	1011	11
C	1100	12
D	1101	13
E	1110	14
F	1111	15

0xDEADBEEF

1101 1110 1010 1101 1011 1110 1110 1111

0x02468ACE

0000 0010 0100 0110 1000 1010 1100 1110

0x13579BDF

0001 0011 0101 0111 1001 1011 1101 1111

“Bit Twiddling”

- C has operators that treat operands simply as sequences of bits
- Q: Why do bit level operations in C (or any language)?
- A#1: lets you pack information as efficiently as possible
- A#2: some processing is faster to implement with bit-level operations than with arithmetic operators

“Bit Twiddling”… (cont’d)

- Ex: image processing
  - pack 64 B&W pixel values into a single **`long long`** operand, and process 64 pixels with one instruction
  - mask one image with another to create overlays
- Other applications:
  - data compression,
  - encryption
  - error correction
  - I/O device control
  - …

Working in Binary With C?

- There is no standard way to…
  - …write a constant in binary

```
i = 01011011;
```

☠ *common source of bugs* ☠
thinking sequence of 1's and 0's means base 2

  - …input an ASCII-encoded binary string and convert to an integer

```
scanf("%b", &i);
```

  - …output an integer as an ASCII-encoded binary string

```
printf("%b", i);
```

- Alternatives?
  - Use octal or hexadecimal representation

BitOps: Unary

- Bit-wise complement (~)
  - operand must be integer type
  - result is ones-complement of operand (flip every bit)
  - ex:

```
  ~0x0d    // (binary 00001101)
== 0xf2    // (binary 11110010)
```

Not the same as Logical NOT (!) or sign change (-)

```
char i, j1, j2, j3;
i = 0x0d;     // binary 00001101
j1 = ~i;      // binary 11110010
j2 = -i;      // binary 11110011
j3 = !i;      // binary 00000000
```

BitOps: Two Operands

- Operate bit-by-bit on operands to produce a result operand of the same length
- And (**&**): result 1 if both inputs 1, 0 otherwise
- Or (**|**): result 1 if either input 1, 0 otherwise
- Xor (**^**): result 1 if one input 1, but not both, 0 otherwise
- Operands must be of type integer

Two Operands... (cont'd)

- Examples

```
  0011 1000
& 1101 1110
  ---------
  0001 1000
```

```
  0011 1000
| 1101 1110
  ---------
  1111 1110
```

```
  0011 1000
^ 1101 1110
  ---------
  1110 0110
```

Differences: Logical and Bit Ops

Results?

```
int a, b, c,
    d, e, f;

int i = 30;
int j = 0;
a = i && j;
b = !j;
c = !i;

float x = 30.0;
float y = 0.0;
d = x || y;
e = !y;
f = !x;
```

☠ *common source of bugs* ☠

difference between logical and bit-level operators

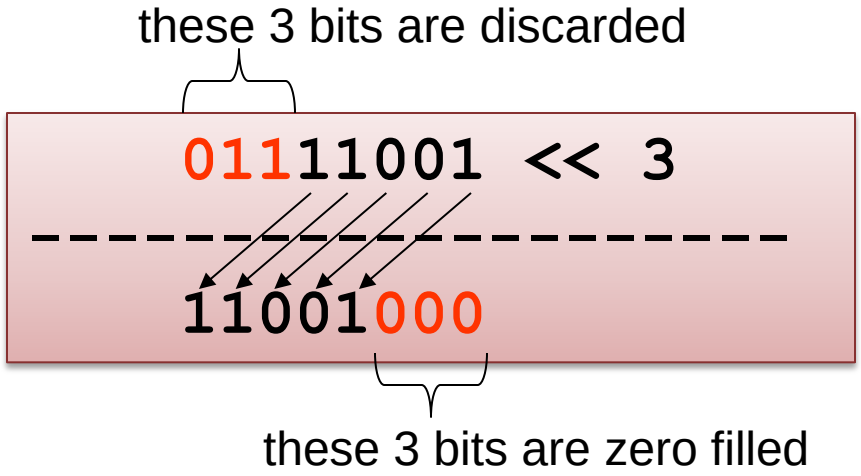
Difference? Problems?

```
int a, b, c,
    d, e, f;

int i = 30;
int j = 0;
a = i & j;
b = ~j;
c = ~i;

float x = 30.0;
float y = 0.0;
d = x | y;
e = ~y;
f = ~x;
```

Shift Operations

- **x << y** is left (logical) shift of **x** by **y** positions
  - **x** and **y** must both be integers
  - **x** should be unsigned or positive
  - 0 <= **y** <= number of bits in **x**
  - **y** leftmost bits of **x** are discarded
  - zero fill **y** bits on the right

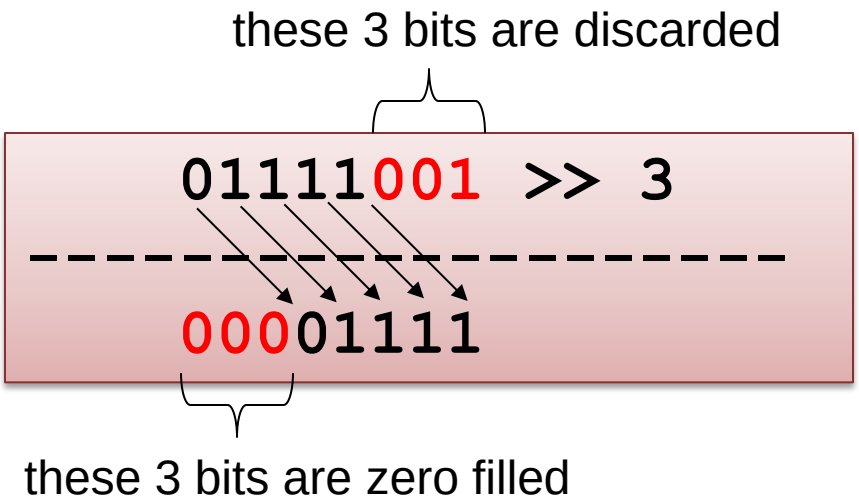
☠ *common source of bugs* ☠
logical shifts on negative numbers



ShiftOps... (cont'd)

- `x >> y` is right (logical) shift of `x` by `y` positions
  - `y` rightmost bits of `x` are discarded
  - zero fill `y` bits on the left

☠ *common source of bugs* ☠
logical shifts on negative numbers



ShiftOps... (cont'd)

- It is occasionally useful to know that...
  - right logical shift of an unsigned number x by y positions is equivalent to dividing x by 2^y
  - left logical shift of an unsigned number x by y positions is equivalent to multiplying x by 2^y

```
unsigned char j, k, m;
j = 121;
k = j << 3
m = j >> 3;
printf("%d %d %d\n", j, k, m);
```

Other Useful Bit Operations

- Complementing, Anding, Oring, and Xoring bits are all provided directly by C operators
- What about the following?
  - clearing all or selected bits to 0's, or setting all or selected bits to 1's
  - testing if all or selected bits are 0's, or 1's
  - counting the number of bits that are 0's, or that are 1's
  - copying all or selected bits from x to y
  - copying a bit or bits from position i of x to position j of y

Clearing Bits to 0's

- Using C operators:
  - & with 0 will clear, & with 1 means "no change"
- So, create a mask with 0's where you want to clear, and 1's everywhere else

If input is...	And mask is...	Then input & mask =
0	0	0 (no change)
0	1	0 (no change)
1	0	0 (clear)
1	1	1 (no change)

Clearing... (cont'd)

- How would you clear (to **0**) all the bits in a **`char`**?

```
unsigned char m = 0x00;
a = a & m;
```

```
  a: 0011 1011
& m: 0000 0000
---------------
  a: 0000 0000
```

- How would you clear the right two bits (without changing the other bits)?

```
unsigned char m = 0xFC;
a = a & m;
```

```
  a: 0011 1011
& m: 1111 1100
---------------
  a: 0011 1000
```

Setting Bits to 1's

- Using C operators:
  - | with 1 will set, | with 0 means "no change"
- So, create a mask with 1's where you want to set, and 0's everywhere else

If input is...	And mask is...	Then input | mask =
0	0	0 (no change)
0	1	1 (set)
1	0	1 (no change)
1	1	1 (no change)

Setting... (cont'd)

- How would you set (to **1**) all the bits in a **`char`** ?

```
unsigned char m = 0xFF;
a = a | m;
```

```
  a: 0011 1110
| m: 1111 1111
--------------
  a: 1111 1111
```

- How would you set the right two bits without changing the other bits?

```
unsigned char m = 0x03;
a = a | m;
```

```
  a: 0011 1110
| m: 0000 0011
--------------
  a: 0011 1111
```

Complementing (Inverting) Bits

- Using C operators:
  - ^ with 1 will complement, ^ with 0 means "no change"
- So, create a mask with 1's where you want to complement, and 0's everywhere else

If input is...	And mask is...	Then input ^ mask =
0	0	0 (no change)
0	1	1 (complement)
1	0	1 (no change)
1	1	0 (complement)

Complementing... (cont'd)

- How would you complement (invert) all the bits in a **`char`** ?

```
unsigned char m = 0xFF;
a = a ^ m;

a = ~a; //also works
```

```
  a: 0011 1110
^ m: 1111 1111
--------------
  a: 1100 0001
```

- How would you complement the right two bits without changing the other bits?

```
unsigned char m = 0x03;
a = a ^ m;
```

```
  a: 0011 1110
^ m: 0000 0011
--------------
  a: 0011 1101
```

Testing Bits for 1's

- Using C operators:
  1. & with 1 will where you want to test, & with 0 elsewhere
  2. then check if result == mask
- So, create a mask with 1's where you want to test, and 0's everywhere else

If input is...	And mask is...	Then input & mask =
0	0	0 (matches mask)
0	1	0 (won't match mask)
1	0	0 (matches mask)
1	1	1 (matches mask)

Test... (cont'd)

- How would you test (if == 1) all the bits in a **`char`** ?

```
unsigned char m = 0xFF;
if ((a & m) == m)
    ...
```

```
  a: 0011 1110
& m: 1111 1111
--------------

     0011 1110
```

- How would you test if the right two bits == 1?

```
unsigned char m = 0x03;
if ((a & m) == m)
    ...
```

```
  a: 0011 1110
& m: 0000 0011
--------------

     0000 0010
```

Not equal to m

Counting the Bits That Are 1's

- Using C operators:
  1. you already know how to test if a specific bit == 1
  2. do this for each bit, one at a time
  3. each time the bit == 1, add 1 to a counter
- A movable mask
  - (1 << i) creates a mask with a 1 in the ith position from the right, and 0 everywhere else

Test... (cont'd)

- How would you count the number of bits == 1 in an unsigned char?

```
unsigned char m;
unsigned int cnt = 0;
for (i = 0; i < 8; i++) {
    m = 1 << i;
    if ((a & m) == m)
        cnt += 1;
}
```

Testing Bits for 0's

- Using C operators:
  - (you try it)
- How would you test (if == 0) all the bits in a char?

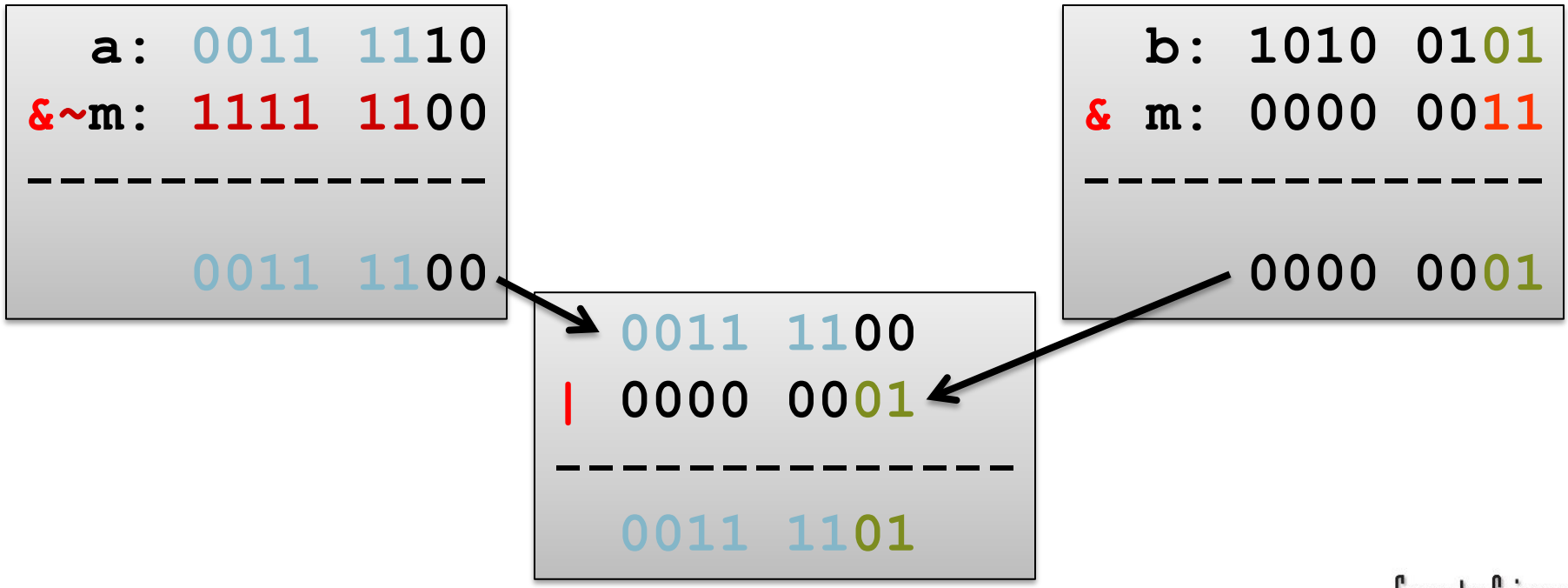
```
???
```

- How would you test if the two right bits == 0?

```
???
```

Copying Selected Bits (from b to a)

- Using C operators:
  - clear all the bits in **a** you do want to replace
  - clear all the bits in **b** you don't want to copy
  - OR **a** with **b** to get result



Exercise 06a

Bitwise operators

- Show the code to set the middle 4 bits of **`char a`** to 1 (without changing the other bits)

- Show the code to copy the middle 4 bits of c to d (without changing the other bits).
  What is d (in binary) afterwards?

```
unsigned char c = 0xd6, d = 0x6c;
```

- What is the value (in binary and hex) of b after executing:

```
b = 0x1D;
m = 0xB7;
b = b ^ m;
```


Reminder: Go to course web page for link to exercise form.
Paste code into ideone.com and submit the link.