Bit Level Operators in C

C Programming and Software Tools
N.C. State Department of Computer Science
Contents

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

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

\[ \text{0xDEADBEEF} \]

\[ \text{0x02468ACE} \]

\[ \text{0x13579BDF} \]
“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
  
  
  ```c
  i = 01011011;
  ```

  – ...input an ASCII-encoded binary string and convert to an integer

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

  – ...output an integer as an ASCII-encoded binary string

  ```c
  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:
    \[
    \sim 0x0d \quad // \quad (binary \quad 00001101) \\
    \sim 0x0d \quad // \quad (binary \quad 11110010)
    \]

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

```c
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

\[
\begin{array}{c}
0011 1000 \\
\& \\
1101 1110 \\
\hline
0001 1000 \\
\end{array}
\]

\[
\begin{array}{c}
0011 1000 \\
| \\
1101 1110 \\
\hline
1111 1110 \\
\end{array}
\]

\[
\begin{array}{c}
0011 1000 \\
^ \\
1101 1110 \\
\hline
1110 0110 \\
\end{array}
\]
Differences: Logical and Bit Ops

Results?

```c
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;
```

Difference? Problems?

```c
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
  - $0 \leq y \leq \text{number of bits in } x$
  - $y$ leftmost bits of $x$ are discarded
  - zero fill $y$ bits on the right

```
01111001 << 3
-------------------
11001000
```

- common source of bugs
- logical shifts on negative numbers
ShiftOps... (cont’d)

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

\[
\begin{array}{c}
01111001 \gg 3 \\
\hline
00001111
\end{array}
\]

- 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 \( 2^y \)

```c
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

<table>
<thead>
<tr>
<th>If input is...</th>
<th>And mask is...</th>
<th>Then input &amp; mask =</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0 (no change)</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0 (no change)</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0 (clear)</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1 (no change)</td>
</tr>
</tbody>
</table>
Clearing... (cont’d)

• How would you clear (to 0) all the bits in a char?

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

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

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

<table>
<thead>
<tr>
<th>If input is...</th>
<th>And mask is...</th>
<th>Then input</th>
<th>mask =</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0 (no change)</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1 (set)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1 (no change)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1 (no change)</td>
<td></td>
</tr>
</tbody>
</table>
Setting... (cont’d)

• How would you set (to 1) all the bits in a char?

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

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

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

<table>
<thead>
<tr>
<th>If input is...</th>
<th>And mask is...</th>
<th>Then input ^ mask =</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0 (no change)</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1 (complement)</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1 (no change)</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0 (complement)</td>
</tr>
</tbody>
</table>
Complementing... (cont’d)

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

```c
unsigned char m = 0xFF;
a = a ^ m;
a = ~a; //also works
```

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

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

<table>
<thead>
<tr>
<th>If input is...</th>
<th>And mask is...</th>
<th>Then input &amp; mask =</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0 (matches mask)</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0 (won’t match mask)</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0 (matches mask)</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1 (matches mask)</td>
</tr>
</tbody>
</table>
Test... (cont’d)

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

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

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

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

```c
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

\[
\begin{align*}
  a & : 0011 \ 1110 \\
  \& \ \sim m & : 1111 \ 1100 \\
  \hline
  0011 \ 1100 \\
\end{align*}
\]

\[
\begin{align*}
  b & : 1010 \ 0101 \\
  \& \ m & : 0000 \ 0011 \\
  \hline
  0000 \ 0001 \\
\end{align*}
\]

\[
\begin{align*}
  & 0011 \ 1100 \\
  \mid & 0000 \ 0001 \\
  \hline
  0011 \ 1101
\end{align*}
\]
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?

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

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

```c
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.