CS 241
Data Organization using C
Binary and Bit Operators

Instructor: Joel Castellanos
e-mail: joel@unm.edu
Web: http://cs.unm.edu/~joel/
Office: Farris Engineering Center, Room 2110

Workflow Tips

1) Use 2 or 3 PuTTy windows: code, compile, and input.
2) vim: If error on line 45, use <esc>:45.
3) vim: Go to top of file: <esc>:0 or gg. Bottom: <esc>G.
4) vim: Search <esc>/string.
5) When debugging, use many printf statements so you ***know*** where your program is flowing and what the values of intermediate fields are.
6) Edit data input file to just one line. When that line works, replace it with a new line – that way you do not generate too much debug output.
Quiz: Reading 2.9 Bitwise Operators

In the C programming language, the ^ operator performs:

a) Bitwise AND
b) Bitwise OR
c) Bitwise Exclusive OR
d) Two's Complement
e) One's Complement

Combinations and Permutations

In English we use the word "combination" loosely, without thinking if the order of things is important. In other words:

"My fruit salad is a combination of apples, grapes and bananas" In this statement, order does not matter: "bananas, grapes and apples" or "grapes, apples and bananas" make the same salad.

"The combination to the safe is 472". Here the order is important: "724" would not work, nor would "247".

In Computer Science we use more precise language:

If the order doesn't matter, it is a Combination.

If the order does matter it is a Permutation.

1. Repetition is Allowed: such as the lock above. It could be “333”.
2. No Repetition: for example the first three people in a running race. Order does matter, but you can't be first and second.
Ideal Serial Binary Signal

Real Binary Signal (Maxim MAX5581)
Information in a Binary Signal

1 Bit
2 Permutations

<table>
<thead>
<tr>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

2 Bits
4 Permutations

<table>
<thead>
<tr>
<th>0 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1</td>
</tr>
<tr>
<td>1 0</td>
</tr>
<tr>
<td>1 1</td>
</tr>
</tbody>
</table>

3 Bits
8 Permutations

<table>
<thead>
<tr>
<th>0 0 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1</td>
</tr>
<tr>
<td>0 1 0</td>
</tr>
<tr>
<td>0 1 1</td>
</tr>
<tr>
<td>1 0 0</td>
</tr>
<tr>
<td>1 0 1</td>
</tr>
<tr>
<td>1 1 0</td>
</tr>
<tr>
<td>1 1 1</td>
</tr>
</tbody>
</table>

4 Bits
16 Permutations

<table>
<thead>
<tr>
<th>0 0 0 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 1</td>
</tr>
<tr>
<td>0 0 1 0</td>
</tr>
<tr>
<td>0 0 1 1</td>
</tr>
<tr>
<td>0 1 0 0</td>
</tr>
<tr>
<td>0 1 0 1</td>
</tr>
<tr>
<td>0 1 1 0</td>
</tr>
<tr>
<td>0 1 1 1</td>
</tr>
<tr>
<td>1 0 0 0</td>
</tr>
<tr>
<td>1 0 0 1</td>
</tr>
<tr>
<td>1 0 1 0</td>
</tr>
<tr>
<td>1 0 1 1</td>
</tr>
<tr>
<td>1 1 0 0</td>
</tr>
<tr>
<td>1 1 0 1</td>
</tr>
<tr>
<td>1 1 1 0</td>
</tr>
<tr>
<td>1 1 1 1</td>
</tr>
</tbody>
</table>

Parallel Binary Signals: Flash Memory Chip

There are 37 pins on each side.
- 17 Pins for: Read/Write Flag, Data In bits (8), and Data Out bits (8).
- 23 pins for other special purposes such as ground, power supply, etc.
- 34 pins remain for specifying memory addresses.

How many bytes can be addressed?
Numbers in Base Ten and Base Two

**Base 10**

\[ 5307 = 5 \times 10^3 + 3 \times 10^2 + 0 \times 10^1 + 7 \times 10^0 \]
\[ = 5000 + 300 + 0 + 7 \]

**Base 2**

\[ 1011 = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 \]
\[ = 8 + 0 + 2 + 1 \]

Examples of Binary Numbers

\[
\begin{array}{cccccccc}
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 \\
512 & 256 & 128 & 64 & 32 & 16 & 8 & 4 & 2 & 1 \\
\end{array}
= 35
\]

\[
\begin{array}{cccccccc}
0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\
512 & 256 & 128 & 64 & 32 & 16 & 8 & 4 & 2 & 1 \\
\end{array}
= 63
\]

\[
\begin{array}{cccccccc}
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
512 & 256 & 128 & 64 & 32 & 16 & 8 & 4 & 2 & 1 \\
\end{array}
= 64
\]

\[
\begin{array}{cccccccc}
1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\
512 & 256 & 128 & 64 & 32 & 16 & 8 & 4 & 2 & 1 \\
\end{array}
= 867
\]
Quiz: Binary

The binary number, 00101010, in base-ten is:

a) 101010
b) 1010
c) 128
d) 75
e) 42

Quiz: Binary

The binary number, 10101010, in base-ten is:

a) 170
b) 76
c) 52
d) 47
e) 42
Hexadecimal: Base-16

Hexadecimal (or hex) is a base-16 system that uses sixteen distinct symbols, most often the symbols 0–9 to represent values zero to nine, and A, B, C, D, E, F to represent values ten to fifteen.

Base 16

\[
0x53AC = 5 \times 16^3 + 3 \times 16^2 + 10 \times 16^1 + 12 \times 16^0
\]
\[
= 5 \times 4096 + 3 \times 256 + 10 \times 16 + 12 \times 1
\]
\[
= 20,480 + 768 + 160 + 12
\]
\[
= 21,420
\]

Why Hexadecimal?

Hexadecimal is trivially more compact than base-10, but significantly more compact than base-2.

Since 16 is a power of 2, it is very easy to convert between Binary and Hexadecimal.

Base 16

0x01239ACF: Four bytes:

\[
\begin{align*}
01 & 23 & 9A & CF \\
0000 & 0001 & 0010 & 0011 & 1001 & 1010 & 1100 & 1111
\end{align*}
\]
Hexadecimal Literals

```c
#include <stdio.h>

void main(void)
{
    printf("%d\n", 0x1); 1
    printf("%d\n", 0x2); 2
    printf("%d\n", 0x3); 3
    printf("%d\n", 0x8); 8
    printf("%d\n", 0x9); 9
    printf("%d\n", 0xA); 10
    printf("%d\n", 0xB); 11
    printf("%d\n", 0xC); 12
    printf("%d\n", 0xD); 13
    printf("%d\n", 0xE); 14
    printf("%d\n", 0xF); 15
    printf("%d\n", 0x10); 16
    printf("%d\n", 0x11); 17
    printf("%d\n", 0x12); 18
}
```

Powers of 2: char, int

```c
#include <stdio.h>

void main(void)
{
    char i=0;
    char a=1;
    unsigned char b=1;
    int c = 1;

    for (i=1; i<22; i++)
    {
        a = a * 2;
        b = b * 2;
        c = c * 2;
        printf("%2d) %4d %3d %7d\n", 
               i, a, b, c);
    }
}
```
Powers of 2: int, long

```c
#include <stdio.h>

void main(void)
{
    char i=0;
    int  c=1;
    long d = 1;

    for (i=1; i<65; i++)
    {
        c = c * 2;
        d = d * 2;
        printf("%2d) %11d %20ld\n", i, c, d);
    }
}
```

<table>
<thead>
<tr>
<th>i</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>29</td>
<td>536870912</td>
<td>536870912</td>
</tr>
<tr>
<td>30</td>
<td>1073741824</td>
<td>1073741824</td>
</tr>
<tr>
<td>31</td>
<td>-2147483648</td>
<td>2147483648</td>
</tr>
<tr>
<td>32</td>
<td>0</td>
<td>4294967296</td>
</tr>
<tr>
<td>33</td>
<td>0</td>
<td>8589934592</td>
</tr>
<tr>
<td>61</td>
<td>0</td>
<td>2305843009213693952</td>
</tr>
<tr>
<td>62</td>
<td>0</td>
<td>4611686018427387904</td>
</tr>
<tr>
<td>63</td>
<td>0</td>
<td>-9223372036854775808</td>
</tr>
<tr>
<td>64</td>
<td>0</td>
<td>4294967296</td>
</tr>
</tbody>
</table>

Format code: \(1d\) for long decimal

Bitwise Operators

- &  bitwise AND \(1010 & 0011 = 0010\)
- |  bitwise inclusive OR \(1010 | 0011 = 1011\)
- ^  bitwise exclusive OR \(1010 ^ 0011 = 1001\)
- ~  bitwise NOT \(~1010 = 0101\)
- << left shift \(00000100 << 3 = 00100000\)
- >> right shift \(00000100 >> 2 = 00000001\)
Shift Operator Example

1. `void main(void)`
2. `{ int i;
3.   for (i=0; i<8; i++)
4.   { unsigned char n = 1 << i;
5.     printf("n=%d\n", n);
6.   } }
7. }

Output:

```
n=1
n=2
n=4
n=8
n=16
n=32
n=64
n=128
```

Quiz: Bitwise AND Operator

1. `#include <stdio.h>`
2. 
3. `void main(void)`
4. `{ 
5.   printf("%d\n", 26 & 28);
6. }

The output is:

<table>
<thead>
<tr>
<th></th>
<th>128</th>
<th>64</th>
<th>32</th>
<th>16</th>
<th>8</th>
<th>4</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>a)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>b)</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>c)</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>d)</td>
<td>12</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>e)</td>
<td>24</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

```

```
0 0 0 1 1 1 0 0 & 0 0 0 1 1 1 0 0
```

```
0 0 0 1 1 0 0 0
```
**Quiz: Bitwise OR Operator**

1. `#include <stdio.h>`
2. 
3. `void main(void)`
4. {
5.     `printf("%d\n", 26 | 28);`
6. }

The output is:

- a) 26
- b) 28
- c) 30
- d) 42
- e) 54

**Convert 77 to an 8-bit Binary String**

- $2^7 = 128$ is > 77 put a ‘0’ in the 128’s place.
- $2^6 = 64$ is <= 77, put a ‘1’ in the 64’s place.
  - AND, subtract 64: $77 - 64 = 13$.
- $2^5 = 32$ is > 13. Put a ‘0’ in the 32’s place.
- $2^4 = 16$ is > 13. Put a ‘0’ in the 16’s place.
- $2^3 = 8$ is <= 13. Put a ‘1’ in the 8’s place.
  - AND subtract 8: $13 - 8 = 5$.
- $2^2 = 4$ is <= 5. Put a ‘1’ in the 4’s place.
  - AND subtract 2: $5 - 4 = 1$.
- $2^1 = 2$ is > 1. Put a ‘0’ in the 2’s place.
- $2^0 = 1$ is <=1. Put a ‘1’ in the 1’s place.
  - AND subtract 1: $1 - 1 = 0$. 

00011100
Convert unsigned char to Binary Array

```c
#include <stdio.h>
void main(void)
{
    char bits[9];
    bits[8] = '\0';
    unsigned char n=83;
    unsigned char powerOf2 = 128;
    int i;
    for (i=0; i<=7; i++)
    {
        if (n >= powerOf2)
        {
            bits[i] = '1';
            n = n-powerOf2;
        }
        else bits[i] = '0';
        powerOf2 /= 2;
    }
    printf("%s\n", bits);
}
```

01010011

The Mask

In computer science, a **mask** or **bitmask** is data that is used for a bitwise AND operation.

Using a mask, multiple bits in a byte, short, int, long or other machine type can be set blocked in a single bitwise operation.

```
<table>
<thead>
<tr>
<th>Original Value</th>
<th>Mask</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 1 0 1 0 1</td>
<td>0 0 0 0 0 1 1 1</td>
<td>0 0 0 0 0 0 1 0 1</td>
</tr>
</tbody>
</table>
```
The Mask

```c
void main(void)
{
    unsigned short x = 30; //00011110
    unsigned short mask = 0xF7; //11110111

    // Turn OFF 8 bit. If 8 bit is already OFF, x is unchanged.
    x = x & mask;
    printf("%d\n", x); // prints: 22

    // Turn OFF all bits except the first 3.
    x = x & 0x07; // 0x07 = 00000111
    printf("%d\n", x); // prints: 6 (00000110)

    // True (non-zero) if 8 bit is ON in x.
    if (x & (~mask)) printf("true\n");
}
```

Using the Mask: Binary Array

1. #include <stdio.h>
2. void main(void)
3. {
4.     char bits[9];
5.     bits[8] = '0';
6.     unsigned char n = 83;
7.     unsigned char powerOf2 = 128;
8.     int i;
9.     for (i=0; i<=7; i++)
10.    { if (n & powerOf2) bits[i] = '1';
11.        else bits[i] = '0';
12.        powerOf2 = powerOf2 >> 1;
13.    }
14.    printf("%s\n", bits);
15.}

In the first method, whenever a power of 2 is found, it is subtracted from n. This method never changes n.

Output:

```
00010001
```
Addition: Base 10 and Binary

\begin{align*}
1 & \quad 1 \\
2 & + 5 \\
9 & + 6 \\
\hline
8 & + 5 \\
\end{align*}

Overflow Addition

```c
#include <stdio.h>

void main (void)
{
    char i=0;
    char a = 123, b = 252;
    unsigned char x = 123, y = 252;
    for (i=1; i<=7; i++)
    {
        a++; b++; x++; y++;
        printf("%4d %4d %4d %4d\n", a, x, b, y);
    }
}
```

\begin{align*}
0 & \quad 1 \\
1 & + 0 \\
1 & + 0 \\
1 & + 0 \\
\hline
1 & + 0 \\
\end{align*}
Quiz: Bitwise Operator

1. `#include <stdio.h>
2. 
3. `void main(void)`
4. `{ 
5.    printf("%d\n", 70 & 73);
6. }

The output is:

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>64</td>
<td>32</td>
<td>16</td>
<td>8</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

&

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1. a) 3  
2. b) 42 
3. c) 64  
4. d) 142 
5. e) 143

Quiz: `<<` and `&`

1. `#include <stdio.h>
2. `void main(void)`
3. {
4.    char bits[9];
5.    bits[8] = '\0';
6.    unsigned char n=37;
7.    unsigned char p = 128;
8.    int i;
9.    for (i=0; i<=7; i++)
10.    { if (n & p) bits[i] = '1';
11.        else bits[i] = '0';
12.        p = p >> 1;
13.    } 
14.    printf("%s\n", bits);
15. }
Quiz: << and &

1) void main(void)
2) { unsigned char a=37;
3)     int i;
4) 
5)     for (i=7; i>=0; i--)
6)         { unsigned char n = 1 << i;
7)             if (!(a & n)) printf("%d, ", n);
8)     }
9)     printf("\n");

a) 64, 32, 16, 4, 1,
b) 64, 16, 8, 4, 2, 1,
c) 64, 16, 8, 4, 1,
d) 128, 64, 16, 8, 1,
e) 128, 64, 16, 8, 2,

Two's Complement

From ordinary binary: Flip the bits and Add 1.

<table>
<thead>
<tr>
<th>Ordinary Binary</th>
<th>Decimal</th>
<th>Two's Complement</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 0001</td>
<td>1</td>
<td>1111 1111</td>
<td>-1</td>
</tr>
<tr>
<td>0000 0010</td>
<td>2</td>
<td>1111 1110</td>
<td>-2</td>
</tr>
<tr>
<td>0000 0011</td>
<td>3</td>
<td>1111 1101</td>
<td>-3</td>
</tr>
<tr>
<td>0000 0100</td>
<td>4</td>
<td>1111 1100</td>
<td>-4</td>
</tr>
<tr>
<td>0000 0101</td>
<td>5</td>
<td>1111 1011</td>
<td>-5</td>
</tr>
<tr>
<td>0000 0010</td>
<td>6</td>
<td>1111 1010</td>
<td>-6</td>
</tr>
<tr>
<td>0000 0111</td>
<td>7</td>
<td>1111 1001</td>
<td>-7</td>
</tr>
<tr>
<td>0000 0001</td>
<td>1</td>
<td>1111 1111</td>
<td>-1</td>
</tr>
<tr>
<td>0000 0010</td>
<td>2</td>
<td>1111 1110</td>
<td>-2</td>
</tr>
<tr>
<td>0000 0011</td>
<td>3</td>
<td>1111 1101</td>
<td>-3</td>
</tr>
<tr>
<td>0000 0100</td>
<td>4</td>
<td>1111 1100</td>
<td>-4</td>
</tr>
<tr>
<td>0000 0101</td>
<td>5</td>
<td>1111 1011</td>
<td>-5</td>
</tr>
<tr>
<td>0000 0010</td>
<td>6</td>
<td>1111 1010</td>
<td>-6</td>
</tr>
<tr>
<td>0000 0111</td>
<td>7</td>
<td>1111 1001</td>
<td>-7</td>
</tr>
</tbody>
</table>
**Two's Complement Addition**

\[
\begin{array}{c}
29 \\
+ -29 \\
\hline
0
\end{array} + \begin{array}{c}33 \\
\hline
0
\end{array} = \begin{array}{c}66 \\
\hline
0
\end{array}
\]

\[
\begin{array}{c}
7 \\
+ -4 \\
\hline
3
\end{array} + \begin{array}{c}33 \\
\hline
0
\end{array} = \begin{array}{c}39 \\
\hline
0
\end{array}
\]

There are 10 types of people: those who understand binary and those who don't.