Number systems & Bit operations

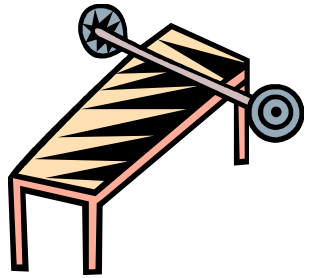
http://xkcd.com/99/

Overview

- Last time:
  - Representing positive integers in base 2, 8, 16
- Bit operations
  - AND, OR, NOT, XOR
  - Shifting
- Binary representation of negative integers
- Addition, subtraction, overflow

Bit operation: AND

If Fred is strong AND Bob is strong,
we will win the football game,
otherwise we will lose
("strong" means benches >= 275 pounds)



Fred benches 135, Bob benches 80 – Will we win?

Fred benches 190, Bob benches 275 – Will we win?

Fred benches 290, Bob benches 210 – Will we win?

Fred benches 290, Bob benches 310 – Will we win?

Bit operation: AND

	Fred strong?	Bob strong?	Win game?
Fred benches 135, Bob benches 80	0	0	0
Fred benches 190, Bob benches 275	0	1	0
Fred benches 290, Bob benches 210	1	0	0
Fred benches 290, Bob benches 310	1	1	1

Bit operation: AND

	Fred strong?	Bob strong?	Win game?
Fred benches 135, Bob benches 80	0	0	0
Fred benches 190, Bob benches 275	0	1	0
Fred benches 290, Bob benches 210	1	0	0
Fred benches 290, Bob benches 310	1	1	1

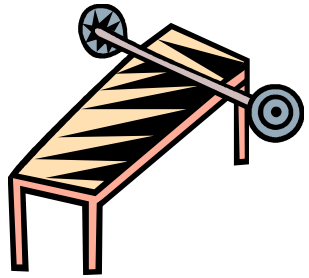
Bitwise AND operator in C

Truth table

x	y	x & y
0	0	0
0	1	0
1	0	0
1	1	1

Bit operation: OR

If Fred is strong OR Bob is strong,
we will win the football game,
otherwise we will lose
("strong" means benches >= 275 pounds)



Fred benches 135, Bob benches 80 – Will we win?

Fred benches 190, Bob benches 275 – Will we win?

Fred benches 290, Bob benches 210 – Will we win?

Fred benches 290, Bob benches 310 – Will we win?

Bit operation: OR

	Fred strong?	Bob strong?	Win game?
Fred benches 135, Bob benches 80	0	0	0
Fred benches 190, Bob benches 275	0	1	1
Fred benches 290, Bob benches 210	1	0	1
Fred benches 290, Bob benches 310	1	1	1

Bitwise OR operator in C

Truth table

x	y	x | y
0	0	0
0	1	1
1	0	1
1	1	1

Bit operation: NOT

- NOT inverts bits:

  $0 \rightarrow 1$

  $1 \rightarrow 0$

Bitwise NOT operator in C

x	~x
0	1
1	0

Bit operation: XOR

- XOR - exclusive or
- Likes it when bits are different (0 and 1, or 1 and 0)
- Hates it when bits are the same (both 0 or both 1)

Bitwise XOR operator in C

x	y	x ^ y
0	0	0
0	1	1
1	0	1
1	1	0

Bit operation: XOR

- Reversible, great for encryption/decryption
  - plaintext message ^ key = encrypted message
  - encrypted message ^ key = plaintext message
- XOR with constant 1 flips a bit
  - Handy if you don't have NOT

x	1 (constant)	x ^ 1
0	1	1
1	1	0

Bit whacking bigger numbers

- AND, OR, NOT, XOR can operate on bigger #'s

AND

	0	1	1	0	1	1	0	0
&	1	0	1	1	0	1	0	1
	0	0	1	0	0	1	0	0

OR

	0	1	1	0	1	1	0	0
|	1	0	1	1	0	1	0	1
	1	1	1	1	1	1	0	1

NOT

~	1	0	1	1	0	1	0	1
	0	1	0	0	1	0	1	0

XOR

	0	1	1	0	1	1	0	0
^	1	0	1	1	0	1	0	1
	1	1	0	1	1	0	0	1

Bit whacking bigger numbers

- Converting binary numbers to decimal:

AND

		0	1	1	0	1	1	0	0
	&	1	0	1	1	0	1	0	1
		0	0	1	0	0	1	0	0

108 & 181 = 36

OR

		0	1	1	0	1	1	0	0
	|	1	0	1	1	0	1	0	1
		1	1	1	1	1	1	0	1

108 | 181 = 253

NOT

	~	1	0	1	1	0	1	0	1
		0	1	0	0	1	0	1	0

~181 = 74

XOR

		0	1	1	0	1	1	0	0
	^	1	0	1	1	0	1	0	1
		1	1	0	1	1	0	0	1

108 ^ 181 = 217

Why bit-whack?

- Often used for bit masking
- Example:
  - One byte used to store 8 on/off user settings

```
const unsigned char AUTO_SAVE        = 1;     0000 0001 = 1
const unsigned char CONFIRM_EXIT     = 2;     0000 0010 = 2
const unsigned char SHOW_TOOLBAR     = 4;     0000 0100 = 4
const unsigned char STARTUP_TIP      = 8;     0000 1000 = 8
const unsigned char COMPRESS_FILES   = 16;    0001 0000 = 16
const unsigned char ENCRYPT_FILES    = 32;    0010 0000 = 32
const unsigned char PASSWORD_PROTECT = 64;    0100 0000 = 64
const unsigned char TRACK_CHANGES    = 128;   1000 0000 = 128
```

Why bit-whack?

```
const unsigned char AUTO_SAVE         = 1;
const unsigned char CONFIRM_EXIT      = 2;
const unsigned char SHOW_TOOLBAR      = 4;
const unsigned char STARTUP_TIP       = 8;
const unsigned char COMPRESS_FILES    = 16;
const unsigned char ENCRYPT_FILES     = 32;
const unsigned char PASSWORD_PROTECT  = 64;
const unsigned char TRACK_CHANGES     = 128;
```

```
unsigned char setting = AUTO_SAVE | COMPRESS_FILES | TRACK_CHANGES;
```

Start with three of the settings enabled.

```
setting = setting | SHOW_TOOLBAR;
```

Enable the toolbar if it isn't already enabled.

```
if (setting & TRACK_CHANGES) { /* do something */ }
```

Do something if track changes is turned on.

```
setting ^= SHOW_TOOLBAR;
```

Toggle the show toolbar setting.

Left bit shifting

- Left shift, **x << y**
  - Bits in x move y positions left
  - Zeros stuffed on right side (least significant bit)
  - Every move left doubles values
  - Left shift of value 3:

```
0000 0000 0000 0011 << 1
0000 0000 0000 0110          = 6
```

```
0000 0000 0000 0011 << 2
0000 0000 0000 1100          = 12
```

```
0000 0000 0000 0011 << 3
0000 0000 0001 1000          = 24
```

Right bit shifting

- Right shift, **x >> y**
  - Bits in x move y positions right
  - Zeros stuffed on left side (most significant bit)
  - Every move right halves value (integer div by 2)
  - Right shift of value 35:

```
0000 0000 0010 0011 >> 1
0000 0000 0001 0001          = 17
```

```
0000 0000 0010 0011 >> 2
0000 0000 0000 1000          = 8
```

```
0000 0000 0010 0011 >> 3
0000 0000 0000 0100          = 4
```

Bit masking

- Bit-whacking to obtain part of number
- Example:

  16-bit number, get least and most significant nibbles

```
unsigned short number     = 12345;
unsigned short lowNibble  = (unsigned short)  (number & 0x000F);
unsigned short highNibble = (unsigned short) ((number & 0xF000) >> 12);
```

```
  0011 0000 0011 1001 = 12345
& 0000 0000 0000 1111 = 0x000F
  0000 0000 0000 1001 = 9
```

lowNibble

```
  0011 0000 0011 1001 = 12345
& 1111 0000 0000 0000 = 0xF000
  0011 0000 0000 0000 = 12288

  0011 0000 0000 0000 >> 12
  0000 0000 0000 0011 = 3
```

highNibble

Signed integers

Sign and magnitude

- Duh, let's just use first bit for sign
  - 0 positive, 1 negative
  - Rest for absolute value

binary	value
0000 1011	+11
1000 1011	-11
0000 0000	+0
1000 0000	-0
0111 1111	+127
1111 1111	-127

IBM 7090, 1961

Sign and magnitude

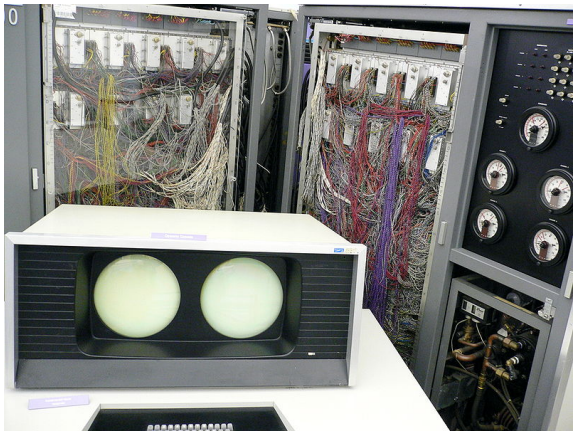
- Disadvantages:
  - Two representations for zero
  - Addition/subtraction complicated

binary	value
0000 1011	+11
1000 1011	-11
0000 0000	+0
1000 0000	-0
0111 1111	+127
1111 1111	-127

```
if (sign1 == sign2)
{
    // add the magnitude of the numbers
    // reapply the sign
}
else
{
    // compare the magnitudes
    // subtract smaller magnitude from larger
    // determine sign of result and apply
}
```

One's complement

- Positive numbers as normal
- Negative numbers flip all bits in positive counterpart
- Easier to add/subtract
- Still have +0 and -0



CDC 6000 series supercomputer, 3 MIPS (1964)

binary	value
0000 1011	+11
1111 0100	-11
0000 0000	+0
1111 1111	-0
0111 1111	+127
1000 0000	-127

Two's complement

- Positive numbers as normal
- Negative numbers flip bits and add one
- Very easy to add/subtract
- Only one version of zero!
- Used by all modern computers

binary	value
0000 1011	+11
1111 0101	-11
0000 0000	+0
1111 1111	-1
0111 1111	+127
1000 0000	-128

Apple MacBook Pro, ~53000 MIPS (2011)

Two's complement

- Decimal to two's complement binary
  - Width in bits (w) needs to be specified!
  - Positive number:
    - Most significant bit is 0, rest as normal
    - Max value, $+2^{(w-1)}-1$
  - Negative number:
    - First find binary of positive number
    - Complement all bits
    - Add 1 to the number in binary
    - Min value, $-2^{(w-1)}$

Adding in binary

- 0 + 0 = 0
- 0 + 1 = 1
- 1 + 0 = 1
- 1 + 1 = 0 carry out
- 1 + 1 + carry in = 1 carry out

		1		1						
	0	0	1	0	1	1	0	0		44
+	0	0	1	0	1	0	0	1	+	41
	0	1	0	1	0	1	0	1		85

		1				1	1			
	0	0	1	0	1	0	1	1		43
+	0	0	1	0	0	0	1	1	+	35
	0	1	0	0	1	1	1	0		78

Two's complement examples

- Find the 8-bit two's complement for -57:

```
find +57 in binary   0011 1001

flip the bits         1100 0110
add 1                +0000 0001
                      1100 0111
```

- Find the decimal for the 8-bit two's complement binary number 1100 1100:

```
                      1100 1100

flip the bits         0011 0011
add 1                +0000 0001
                      0011 0100
convert binary to decimal, 0011 0100 = 52
add the negative sign, -52
```

Two's complement examples

- Find the 8-bit two's complement for +57:

```
find +57 in binary, 0011 1001
done
```

- Find the decimal for the 8-bit two's complement binary number 0100 1100:

```
leading bit is 0, so must be positive
convert binary to decimal, 0100 1100 = +76
done
```

Adding in two's complement

- Add two's complement binary representation
- Ignore any carry out from most significant bit

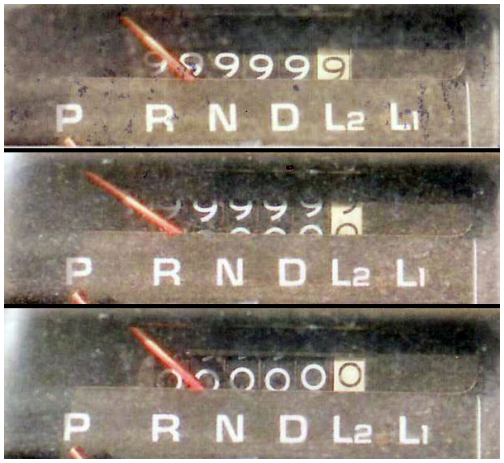
1	1	1	1	1	1	1	1	
	0	0	0	0	1	1	1	1
+	1	1	1	1	1	0	1	1
	0	0	0	0	1	0	1	0

	15
+	-5
	10

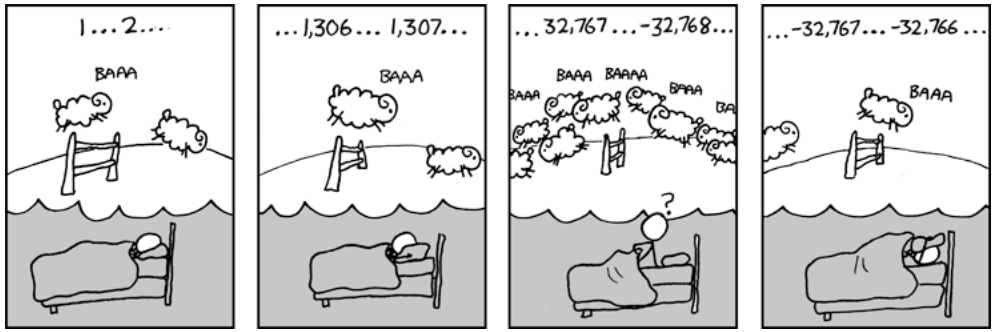
- Subtraction, e.g. 15 - 5
  - Find negative version of second number
  - Use addition, 15 + (-5)

Overflow

```
signed char count = 125;
while (count < 128)
{
   printf("%d\n", count);
   count++;
}
```

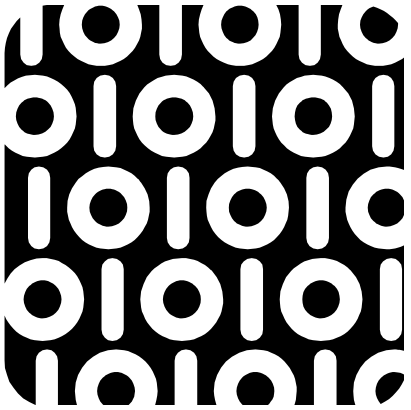


125	0111 1101
126	0111 1110
127	0111 1111
-128	1000 0000
-127	1000 0001
-126	1000 0010
…	…
-2	1111 1110
-1	1111 1111
0	0000 0000
1	0000 0001
2	0000 0010
…	…



http://xkcd.com/571/

Summary



- Learned bit manipulations
  - AND, OR, NOT, XOR
  - Shifting left and right
  - Packing data, e.g. 8 options into 1 byte
  - Masking out part of a number, e.g. low nibble
- Ways to represent negative integers
  - Sign/magnitude, 1's complement, 2's complement
- Binary addition & subtraction
  - Watch out for overflow