Bits

Binary questions

Everybody knows that computers use binary for numbers and arithmetic, but:

Why?

Computer scientists need to know something about binary, but:

How much?

Computers are good at binary arithmetic and translation to and from decimal, so why not leave them to it?

Economics

One reason computers use binary is economics

A few early computers used decimal, but it needed more circuitry, and more time, than using binary

Binary is just simpler, for a computer

Information

A second reason for using binary is that a binary number is made up of bits

The bit is the fundamental unit of information, and it makes sense to store all kinds of data in the same way

Computers use bit patterns to represent everything: instructions, numbers, characters, pixels, ...

The word "binary" means "to do with bits", whether numerical or not (e.g. binary file = non-text file)

How does the computer know?

A common question, when people look at computer architecture for the first time, is "how does the computer know whether a memory location holds an instruction, number, character or pixel?" ***It doesn't***

If the current operation is "execute", the bits are treated as an instruction; if "add", as a number, if "print", as a character, if "display", as a pixel

So, the knowledge of what each lump of memory represents is embedded implicitly in the program's instructions

Bit manipulation

Computer scientists need to know about binary, because bit manipulation is needed by programmers in:

- understanding architecture to program well
- operating systems and device drivers
- small devices such as smart phones
- networking, protocols, the Web
- efficient programs e.g. cryptography
- file formats, e.g. audio, video, compression
- pixel manipulation in graphics, image processing

Need to know

What do you need to know about binary:

- arithmetic? ***no***
- counting? ***yes***
- handling of negative numbers? ***yes***
- translation to/from decimal? ***no***
- translation limits? ***yes***

And bit manipulation:

- pack numbers into groups of bits ***yes***
- unpack bits into a signed/unsigned number ***yes***
- floating point numbers? ***very little***

Decimal Counting

With a decimal 4-digit counter, the rightmost digit rolls round, and there may be carries:

$\boxed{2}\boxed{3}\boxed{9}\boxed{9}$ $\boxed{2}\boxed{4}\boxed{0}\boxed{0}$

Each position has `10` possible digits, so the counter can display `10 x 10 x 10 x 10 = 10000` different numbers, from `0000` to `9999`

To avoid overflow (wrap-around) mistakes, you need to avoid counting up from `9999` or down from `0000`

Binary Counting

With a binary 4-bit counter, the rightmost digit rolls round, and there may be carries:

$\boxed{1}\boxed{0}\boxed{1}\boxed{1}$ $\boxed{1}\boxed{1}\boxed{0}\boxed{0}$

Each position has `2` possible digits, so the counter can display `2 x 2 x 2 x 2 = 16` different numbers, from `0000` to $\texttt{1111}_2$ (`0..15`)

To avoid overflow (wrap-around) mistakes, you need to avoid counting up from $\texttt{1111}_2$ or down from `0000`

Bytes

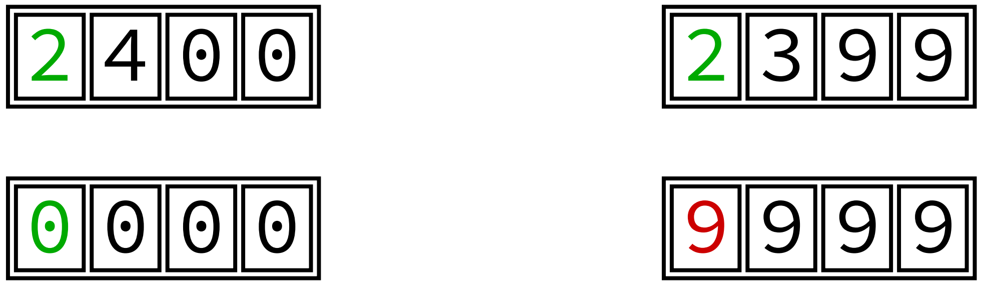
A byte is like a binary counter with `8` digit positions

So it has `2 x 2 x 2 x 2 x 2 x 2 x 2 x 2 =` 2^8 `= 256` different possibilities

They run from `00000000` to 11111111_2 `= 255`

Decimal negatives

Having a minus sign in front is not natural for mechanical counters or computers - instead, half the possibilites are reserved as negative



By counting down from `0`, we can see that `9999` represents `-1`: first digits `5..9` indicate negative numbers, using the ***same*** counter

Working it out

How do you work out what `7385` means?

You subtract from `0000`, and forget everything except the four right most digits, to get `-2615`

What range of numbers does the counter cover?

From `5000 = -5000` to `4999`

To avoid overflow, avoid counting down from `5000` or up from `4999`

This is called ten's complement arithmetic

Binary negatives

Half the possibilities are reserved as negative

$\boxed{0}\boxed{1}\boxed{0}\boxed{0}$ $\boxed{0}\boxed{0}\boxed{1}\boxed{1}$

$\boxed{0}\boxed{0}\boxed{0}\boxed{0}$ $\boxed{1}\boxed{1}\boxed{1}\boxed{1}$

By counting down from `0`, we can see that 1111_2 represents `−1`: first digit `1` indicates a negative number, and the arithmetic circuitry in the processor is (almost) identical

Working it out

How do you work out what 1101_2 means?

You subtract from 0000, and forget everything except the four right most digits, to get $-0011_2 = -3$

What range of numbers does the counter cover?

From $1000_2 = -1000_2 = -8$ to $0111_2 = 7$

For bytes, the range is $10000000_2 = -2^7 = -128$ up to $01111111_2 = 2^7-1 = 127$

This is called two's complement arithmetic

How does the computer know?

When a number is stored in a byte, how does the computer know whether it unsigned (`0..255`) or signed (`-128..127`)? ***It doesn't***

You tell the computer to do unsigned/signed arithmetic or to print out the number unsigned/signed or whatever

The knowledge resides in the instructions

Integers

Computers also use two-byte integers, giving an unsigned range `0..65535` or signed range `-32768..32767`

Computers also use four-byte integers, giving an unsigned range `0..4294967295`, i.e. about `4` billion, or signed range `-2147483648..2147483647`

Computers also use eight-byte integers, giving $0..2^{64}-1$, i.e. about `18` quintillion, or $-2^{63}..2^{63}-1$

Sex

It has never been clear whether multi-byte integers should be stored big-endian or little-endian - the choice is sometimes called the sex of the computer, (though nobody knows which is which, and some are bi)

Decimal numbers in English are written big-endian, but (a) simple arithmetic is done right to left (b) in a calculator, typed digits emerge from the right and (c) there is a story that we stole the notation from documents in a right-to-left Arabic languages, without realising we should have reversed it

Does it matter?

When does it matter whether a computer is big- or little-endian? ***Answer: rarely***

- if you store integers in binary files
- if you send integers over the net
- if you re-interpret an integer in memory as an array of bytes or vice versa, e.g. with pixels

Hex

Hex, short for hexadecimal, is base 16. It is used as a shorthand for binary (1 hex digit = 4 bits)

```
int n = 0x3C0;  // 0011 1100 0000
```

Beware: `0377` in C means octal, now obsolete

Hex is used when emphasizing bit patterns, but is often used inappropriately, e.g. character `0x3C0` instead of `960` for π or colour `0x00FF00` instead of `(0%,100%,0%)` for green

Example: hex printing

To print an int in hex, in order to check its bit pattern:

```
printf("%08x\n", n);
```

`%x` means print in hex

`%8x` means 8 columns

`%08x` means leading zeros, not spaces

For 1, 2, 4, 8 byte integers, use `%02x`, `%04x`, `%08x`, `%016lx` (add letter `l` for long arguments)

C integer types

Integer variables in C have ***roughly*** types:

- `char` (one byte, one ascii character)
- `unsigned char`
- `short` (two bytes)
- `unsigned short`
- `int` (four bytes)
- `unsigned int`
- `long` (eight bytes)
- `unsigned long`

Advice: use unsigned char

If you are manipulating bytes, you could just use the `char` type

But you don't know whether it is signed or unsigned (the standard says it depends on the computer)

And if it is signed, `char c = 0x80` may give a compiler warning because hex constants are unsigned

So I recommend defining a `byte` type:

```
typedef unsigned char byte;
```

Warning

Technically, C types are represented in "the most convenient way on the current computer" - in practice:

`char` is sometimes unsigned - use `signed char` or `unsigned char` for bytes

`short` is almost always two bytes

`int` is almost always four bytes (past 2, future 8)

`long` is usually eight bytes, but is four bytes on 32-bit systems and native 64-bit Windows (so use Cygwin)

Variations

Sometimes "the most convenient representation" is right

But for truly portable software, it isn't, so for example, the `stdint.h` header provides types ending with `_t`:

- `int8_t`, `int16_t`, `int32_t`, `int64_t`
- `uint8_t`, `uint16_t`, `uint32_t`, `uint64_t`

And, e.g, `stdlib.h` provides `size_t` meaning "best type to hold sizes, up to the memory limit"

The headers vary, so your programs don't have to!

Coercion

When different types are combined, there are subtle rules of conversion, called coercion, that are applied implicitly by the C compiler

Conversion to a bigger type includes sign extension, e.g. if a negative `short` is copied into an `int`, the top 16 bits are set to 1 so that it represents exactly the same number:

```
short s = -42;
int n = s;
if (n == -42) printf("ok\n");
```

Assignment

In an assignment `x = ...`, if the type of the right hand side is bigger than the variable can hold, the extra bits are thrown away.

However, depending on the compiler, if the right hand side is a constant, and strict options are used, there may be a warning if the value changes:

```
short s = 65535;
```

The number `65535` consists of 16 digits, so it fits in a `short` variable. But the value becomes negative.

Casts

This can be fixed ***if you know what you are doing*** by explicitly casting a value of one type to another:

```
short s = (short) 65535;
```

You can also specify the type of constants:

```
long n = 4L * 1024L * 1024L * 1024L;
```

Without the L, the right hand side would be an `int` which wouldn't hold the result accurately.

Casts indicate that some dirty trick is being used, so they should be very rare!

Integer promotion

You also need to be aware of integer promotion.

In integer expressions, all variables, constants and intermediate values are 'promoted'.

That means they are expanded to `int` (or possibly `uint`, `long` or `ulong`). This matches what processors typically do, when they hold integer values in registers.

Bit operators

The bit operators in C are:

```
&          bitwise and
|          bitwise or
^          bitwise xor
~          bitwise not
<<         shift left
>>         shift right
```

C uses the pow function, not ^, for powers

The ~ operator changes all the bits

Masking

The `&` operator is most often used for masking

That means isolating just some of the bits from a pattern

Suppose `n` holds 11101011_2 and we want to split this into two blocks of four bits each

The hex constant `0x0F` represents the rightmost four bits, and `n & 0x0F` gives 1011_2

The hex constant `0xF0` represents the other four bits, and `n & 0xF0` gives 11100000_2

Example: testing odd

To test whether an integer is odd:

```
if ((n & 0x1) == 0x1) ...;
```

You could write `(n & 1) == 1`, but it is usually more readable to use hex constants during bit manipulation, to emphasise the bit patterns

Advice: use lots of brackets round bit operations, because the precedences of the bit operators are "wrong" (like `||`, `&&` instead of `+`, `*`)

Shifting right

The `>>` operator shifts a number to the right by a given number of bits.

If `n` holds bit pattern $\texttt{10110}_2$ or $\texttt{10111}_2$, then `n >> 1` gives $\texttt{1011}_2$.

That means `n >> 1` divides `n` by `2` (discarding any remainder), `n >> 2` divides `n` by `4`, and so on.

Use `n / 2` when doing arithmetic, `n >> 1` when manipulating bits, and trust the compiler to choose the best instruction.

Shift right ambiguity

When `>>` is used to shift a ***negative*** number to the right, what happens?

If a number is signed, does the `>>` do sign extension to preserve the sign?

The C standard doesn't require a processor to have an instruction which does that, so the result is undefined.

So, `>>` ***should only be used on unsigned numbers***.

Shifting left

The `<<` operator shifts a number to the left by a given number of bits.

If `n` holds bit pattern $\texttt{1011}_2$, then `n << 1` gives $\texttt{10110}_2$.

That means `n << 1` multiplies `n` by `2`, `n << 2` multiplies `n` by `4`, and so on (except for overflow).

Use `n * 2` when doing arithmetic, `n << 1` when manipulating bits, and trust the compiler to choose the best instruction.

Shift left ambiguity

When `<<` is used to shift a ***negative*** number to the left, what happens?

There is no ambiguity about what the resulting bit pattern should be, on the vast majority of processors anyway, but a number could switch from unsigned to signed or vice versa.

This counts as another undefined situation, which the sanitize option will give an error or warning for.

So, `<<` ***should only be used on unsigned numbers***.

Packing

Suppose that compression is needed in a file, or a network packet, or a program with lots of data

Then you might want to pack several pieces of data into one variable

For example, in graphics, a colour is often three numbers, each `0..255`, for red, green and blue components (ignoring opacity) packed into one integer

Example: Colour packing

Let's write a function using the | (or) operator and shifts to pack the three component numbers into one integer

```
// Pack three components, each 0..255, into a colour
int pack(int r, int g, int b) {
    unsigned int ur = r, ug = g, ub = b;
    return (ur << 16) | (ug << 8) | ub;
}
```

Programmers often write x+y instead of x|y, which is the same ***if there no common bits***, but it is more readable to use | when manipulating bits

Example: Colour packing

```
unsigned int ur = r, ug = g, ub = b;
```

Unsigned copies of the arguments need to be made, so that it is unsigned integers that are shifted

It would be a ***mistake*** to define these as `unsigned char`, because then they would get promoted to `int` before being shifted

Even if the arguments `r`, `g`, `b` were declared as `unsigned char`, they would need to be copied into `unsigned int` variables

Unpacking

To unpack some numbers that have been packed, you can use shifting and masking:

```
// Unpack the three components from a colour
void unpack(int c, int rgb[3]) {
    unsigned int uc = c;
    rgb[0] = (uc >> 16) & 0xFF;
    rgb[1] = (uc >> 8) & 0xFF;
    rgb[2] = uc & 0xFF;
}
```

You can mask and then shift, but then the masks are longer

Signed pieces

Sometimes, the pieces to be packed and unpacked are signed and can be negative

Suppose one `int` is to be used to hold (x,y) coordinates, where each coordinate is a signed 16 bit number (range `-32768..32767`)

Example: packing coordinates

Here is a function to pack two coordinates:

```c
// Pack two signed 16-bit coordinates
int pack(int x, int y) {
    unsigned int ux = x, uy = y;
    int p = ((ux & 0xFFFF) << 16) | (uy & 0xFFFF);
    return p;
}
```

If an `int` is guaranteed to be 32 bits, then the first mask is unnecessary (shifting discards bits that don't fit)

The resulting position variable may be negative (if `x` is negative)

Hex constants

```
int p = ((ux & 0xFFFF) << 16) | (uy & 0xFFFF);
```

A hex constant `0xFFFF` is always positive

It is normally promoted to `int`, so it becomes signed

But here, it is promoted to `unsigned int` to match the other integers

Sign extension

Unpacking is more difficult, because the leading `1` bits in a negative number have to be recovered explicitly

There are several ways to do this:

- explicit
- double shift
- numerical
- use minus one
- use the ~ operator

The last one is the best

Explicit sign extension

Suppose `x` holds 16 bits, with the leftmost of the 16 a 1, e.g. `00000000000000001101011001111011` and we want to make it a negative `int`

The most obvious explicit way is with:

```
x = 0xFFFF0000 | x;
```

This adds in the missing bits but (a) it is easy to get the hex bit pattern wrong, and (b) it depends on knowing that an int has 32 bits

Double shift

Another way of doing it that is recommended in many places is:

```
x = (x << 16) >> 16;
```

This is no good because of the ambiguity of shifts for negative numbers, and it won't work on all platforms

Numerical approach

Here's a numerical approach:

```
x = x - 65536;
```

This works because `x - 65536` is `x + (-65536)` and `-65536` is `0xFFFF0000`

But (a) it is obscure and (b) it is bad style because it is better to do bit manipulation with bit operators rather than with arithmetic

Use minus one

Here's a clever approach:

```
x = (-1 << 16) | x;
```

This works because `-1` consists of 32 ones

But with some compilers it will generate a warning because `-1` is a 'numerical' operation

Use the ~ operator

A similar clever approach is:

```
x = (~0U << 16) | x;
```

The constant `0U` is zero, forced to have the type `unsigned int`

This works because `~` swaps 0s and 1s, so `~0U` consists of 32 ones

This is pure bit manipulation with no assumption of 32 bits in it, so it can be used quite generally

Example: unpacking coords

Here is a function to unpack two coordinates:

```
// Unpack two signed 16-bit coordinates
void unpack(int p, int xy[2]) {
    unsigned int up = p;
    int x = (up >> 16) & 0xFFFF;
    if ((x & 0x8000) != 0) x = (~0U << 16) | x;
    xy[0] = x;
    int y = up & 0xFFFF;
    if ((y & 0x8000) != 0) y = (~0U << 16) | y;
    xy[1] = y;
}
```

Advice:

A summary of general tips is:

- use `-fsanitize=undefined` to ensure cross-platform safety
- use unsigned integers when shifting
- use hex constant and bit operators, avoiding decimal constants and arithmetic operators
- use lots more brackets than usual