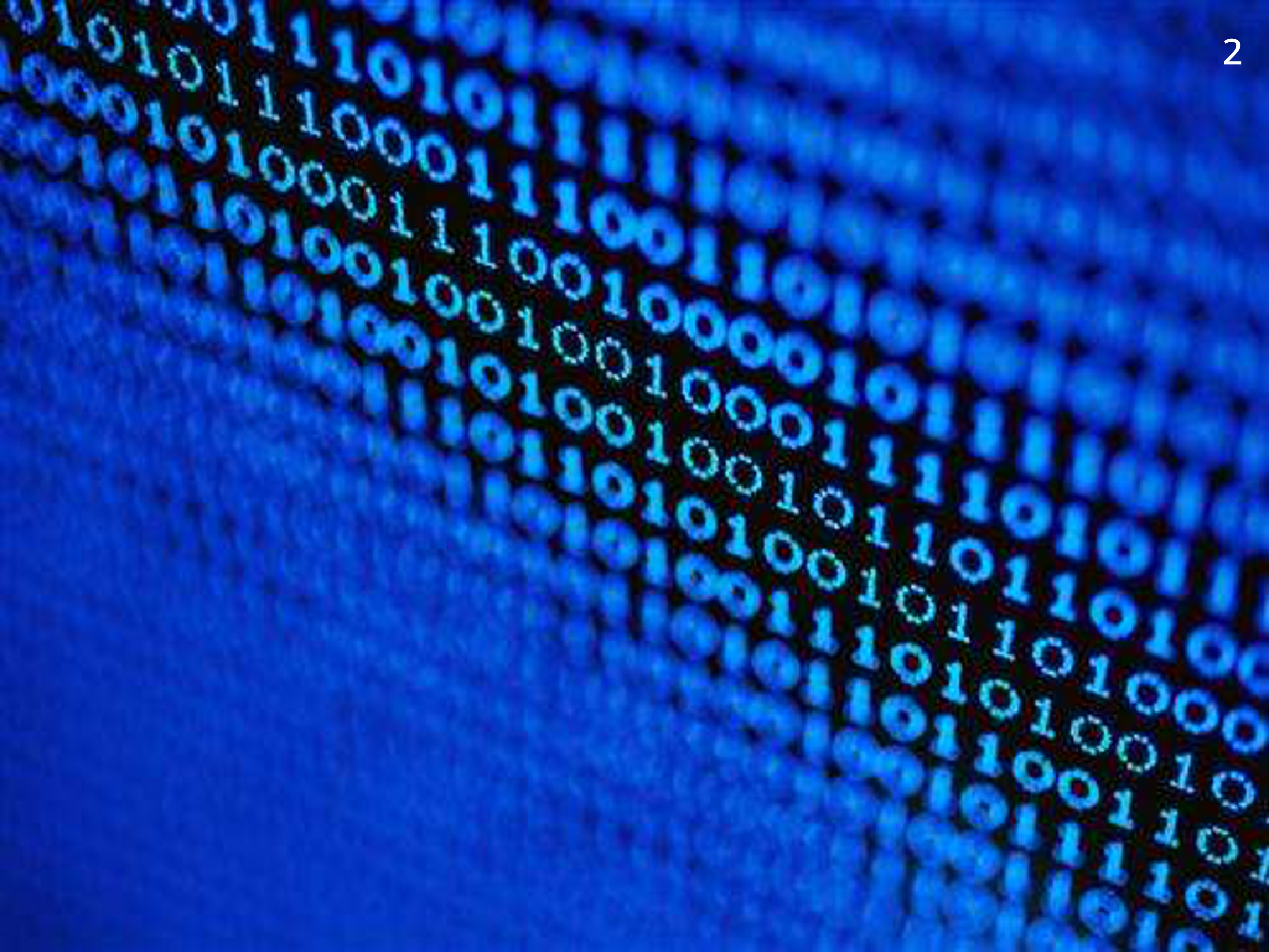
Bits



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)

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 the layout of a program 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 or unpack groups of bits *yes*
- treat bits as a signed/unsigned number *yes*
- floating point numbers? *very little*

Decimal Counting

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

2	3	9	9

2	4	0	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 counter, the rightmost digit rolls round, and there may be carries:

1	0	1	1

1	1	0	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 counter with 8 digit positions

So it has $2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 = 2^8 = 256$ 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

2	4	0	0		2	3	9	9
0	0	0	0		9	9	9	9

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 possibilites are reserved as negative

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

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` or signed range `-2147483648..2147483647`, i.e. about four billion

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

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 `(0,255,0)` 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 (ascii character)
- `unsigned char`
- `short` two bytes
- `unsigned short`
- `int` four bytes
- `unsigned int`
- `long` eight bytes
- `unsigned long`

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 64-bit Windows, so use `long long` to be sure

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");
```

Casting

In *some* of the cases where the bit pattern means something different, you get a warning:

```
short s = 65535;  compile with -pedantic
```

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 = 42L;
```

Bit operators

The bit operators in C are:

```
&          and
|          or
^          xor      C has no power operator!
~          not
<<         shift left
>>         shift right
```

Example: find the low bits

To pick out the rightmost 4 bits from a number, as an integer from 0 to 15:

```
int d0 = n & 0x0F;
```

To pick out the next 4 bits:

```
int d1 = (n >> 4) & 0x0F;
```

And the next 4 bits:

```
int d2 = (n >> 8) & 0x0F;
```

Example: packing

To pack 4-bit numbers into an int:

```
int n = (d2 << 8) | (d1 << 4) | d0;
```

OR

```
int n = (d2 << 8) + (d1 << 4) + d0;
```

In principle, the first is better because it is all bits - the second relies on the assumption that the things being added have no bits in common, and therefore no carries - but the second style is extremely common

Example: testing

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 bit notation throughout

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