ECS404U Computer Systems and Networks

Week 3: Digital Representation

Digital Representation

- Number of things we can represent
- Positive whole numbers.
- Conversion between bases, decimal, octal, hexadecimal, binary
- Long addition and multiplication in different bases.
- (Positive and) Negative whole numbers (2s complement)
- Real Numbers
- Characters
- Sound and vision (MP3, JPEG, MP4 and DVD).

This week

- Number of things we can represent
- Positive whole numbers.
- Conversion between bases, decimal, (octal), hexadecimal, binary
- Long addition and multiplication in different bases (binary).

1’s and 0’s

- We all know that computers represent things using 1’s and 0’s, a system called **binary**.
- 1’s and 0’s can be:
  - low or high voltage
  - pulse or no pulse (on wire)
  - light or no light (on optical fibre)
  - hole or no hole (CD or DVD)
- But using binary is a **choice**

Binary is a choice

- We could decide to build computers that used:
  - decimal
  - ordinary characters

  as their internal representations.
- Why don’t we?

But, first fundamental

- how we represent things is a choice

Second fundamental

- If we only have a finite number of characters, then there are only a finite number of representations possible.

Viewing this one way

- Given k bits, there are only n = 2^k possible bit sequences of length k, so only 2^k different possible codes of length k to represent things with.

Viewing this another way

- Given n things to represent, we need at least k = $\log_2 n$ bits to write distinct codes for all of the n things.

- A real(-ish) example: much of the internet is still working with IPv4 addressing which uses 32 bits. That allows 4 billion addresses, which is not enough to give every networked device a different address. IPv6 uses 128 bits.

Why are there 2^k bit sequences of length k?

00 01 10 11

00 01 10 11

00 01 10 11

0 00 01 10 11 1 00 01 10 11

Each time you add a bit, the number of possible bit sequences doubles: for each shorter sequence you get two longer ones, one putting 0 on the front, and the other 1.

Counting…

- If you have one bit, your sequence is 0 or 1, so you have $2 = 2^1$ possible sequences.

- If you have two bits, then we predict $2 * 2 = 4 = 2^2$ possible sequences. We can check this: 00, 01, 10, 11

- If you have three bits, then we predict $2 * 4 = 8 = 2^3$ possible sequences. We can check this too: 000, 001, 010, 011, 100, 101, 110, 111,

- It follows that when we have k bits, then there are $n=2^k$ possible sequences.

Representation

- If we have three bits available, we can form $2^3=8$ different bit patterns.

- So we can represent (code up, name) $2^3=8$ different things.

- There does not have to be any reason why we choose a particular bit pattern to represent a particular thing.

Representation	Thing	Representation	Thing
000	😎	100	😡
001	😶	101	😠
010	😀	110	🤓
011	😬	111	😱

Representation	Thing	Representation	Thing
000	😎	100	😡
001	😶	101	😠
010	😀	110	🤓
011	😁	111	😱

There is no real reason why I have chosen these emojis, or given them these numbers.

But if I want to code up more, then I need more bits.

Four bits would give me 16 emojis.

We can turn this round: How many bits

- If I am allowed a k bit representation, I can represent up to 2^k things.

The converse of this is:

- If I have more than 2^k things, then I need more than k bits to have representations for all of them.

How many bits?

- Does it take to code every second in the day?
  - There are 86,400 seconds in the day
  - 64 * 1000 < 86 * 1000 < 128 * 1000
  - $2^6 * 2^{10} < 86{,}400 < 2^7 * 2^{10}$
  - $2^{16} < 86{,}400 < 2^{17}$
  - So 17 bits.

How many bits?

- Would it take to give every electron in the universe its own IP address?
  - Scientists believe there are about 10^{80} electrons in the universe.
  - This is $(10^3)^{26.7} = (2^{10})^{26.7} = 2^{267}$
  - So 267 bits.
  - So the IP address would have to contain 267 bits

01100101011001010110010101100101
01100101011001010110010101100101
01100101011001010110010101100101
01100101011001010110010101100101
01100101011001010110010101100101
01100101011001010110010101100101
01100101011001010110010101100101
01100101011001010110010101100101
01100101011

Representation

Before we start to think about how computers represent things like letters and numbers, it’s worth taking time to realise not everyone does it the same way.

Alphabets

- English uses a latin alphabet.
- Other European languages use variants with accents
- Other languages use entirely different alphabets: Greek, Russian, Arabic
- Some languages use syllabaries: Hindi, Tamil
- Still others complex character sets: Chinese

Character Sets

- Standard character sets include punctuation, numerals and non-printing characters, as well as letters.
- And they distinguish between upper and lower case…
- The first ubiquitous computer character set (ASCII) basically used American English… and the rest of the world complained.

Number Systems

- The system we use for writing down numbers is called arabic, because it came to Europe from the Middle East (it was actually invented in India), though it had precursors (eg Greek).
- Before this system, there were lots of others, Roman, several in Ancient Greece, ones for each civilisation you can think of.
- Lots of these had basic problems: size limits (ie could not represent really big numbers), problems with calculation (translate numbers to abacus and then calculate, translate back).
- So the arabic system is much smarter than you think at first.

Arabic

- Positional: what a numeral represents depends on where it is in the number.
- 3234 contains two occurrences of the numeral 3. One denotes “thirty” and the other denotes “three thousand”. Neither denotes “three”.
- It has a numeral 0
- There are simple algorithms for addition, subtraction, multiplication that work on arbitrary numbers. Children learn these in primary school.

Representations

- Computers are taking us into a new world.
- Maybe in the new world we’d want to do things differently.
- Why do we use a basic unit of 10 (base 10)?

Base 10

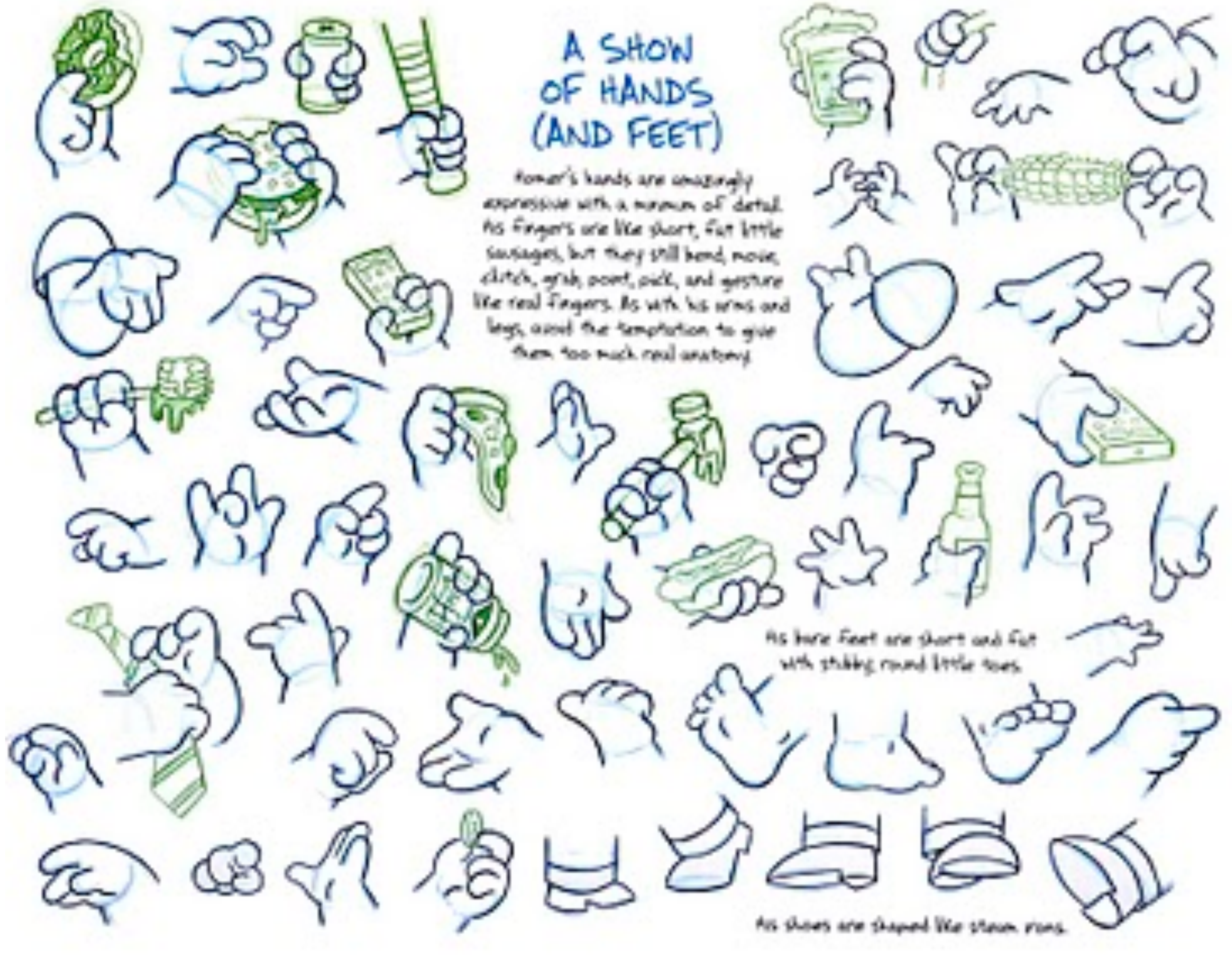
- From a mathematician’s point of view, base 10 is a fairly rubbish base to use.
- 12 would be much better.

A thought experiment

- Let’s think about other possibilities.
- How many fingers do cartoon characters have?

Here’s a picture to help

and here's another


A SHOW OF HANDS (AND FEET)
Homer's hands are amazingly expressive with a minimum of detail. His fingers are like short, fat little sausages, but they still bend, move, clutch, grab, point, pick, and gesture like real fingers. As with his arms and legs, avoid the temptation to give them too much real anatomy.
His bare feet are short and fat with stubby, round little toes.
His shoes are shaped like steam irons.

So let's take this logically

- If we count in 10's, then ...
- over in cartoonworld, they count in 8's

Interlude: cartoon world in detail

Counting in 8's

- So over in cartoonworld when they write 10, they mean the number we call 8.
- When they write 14 (one four), they mean the number we call 12 (1*8+4=12)
- When they write 25, they mean the number we call 21 (2*8+5=21).

Going the other way

- The number we write as 10, they write as 12, because it is 1*8 + 2
- Similarly, our 35 is 4*8 + 3, so they write 43

And on...

- It goes on.. over in cartoonworld, when they write 100, they mean
- $1*8^2 + 0*8^1 + 0*8^0$
- Remember that $8^0 = 1$.

And on...

- So they work in powers of 8:
- $8^0 = 1$, $8^1 = 8$, $8^2 = 64$, $8^3 = 512$, $8^4 = 4096$, ..

- They would systematically use a different representation for numbers.

- And if you think about it, there’s no reason why they should not use an 8-based number system, just as we use a 10-based one.

- Mathematicians call this working in base 8.

A larger example

- When a cartoonworld character writes 2013 they mean:
- $2 * 8^3 + 0 * 8^2 + 1 * 8^1 + 3 * 8^0$
- = 2*512 + 0*64 + 1*8 + 3*1
- = 1024 + 0 + 8 + 3
- = 1035

Some notation

- We'll write 2013_8 to show we mean that we are counting in 8's (mathematicians call this working to base 8)
- We'll write 1035_{10} to show we mean that we are counting in 10's (working to base 10)
- So we've just seen that $2013_8 = 1035_{10}$

How to convert

- To convert a cartoonworld, base 8, number to our notation: 1452
- make a note of the relevant powers of 8: 1, 8, 64, 512
- Number is:

  1 * 512 + 4 * 64 + 5 * 8 + 2 * 1

  = 512 + 256 + 40 + 2

  = 810

How to convert

- To convert one of our numbers to cartoonworld, base 8, keep dividing by 8 and taking the remainder to build up the number:

$1973_{10} = 8 * 246_{10} + 5$

$= 8 * (8 * 30_{10} + 6) + 5_8$

$= 8^2 * 30_{10} + 65_8$

$= 8^2 * (8 * 3_{10} + 6) + 65_8$

$= 8^3 * (3_{10}) + 665_8$

$= 3665_8$

Successive division

- This method is called successive division.
- What we do is keep dividing (by 8) and get the number we want by assembling the remainders.

Successive division

1973 divided by 8 is 246 remainder 5

246 divided by 8 is 30 remainder 6

30 divided by 8 is 3 remainder 6

3 divided by 8 is 0 remainder 3

$1973_{10}=3665_8$

How to convert

- These are different algorithms.
- They don’t have to be.
- But we are more familiar with calculations in base 10.

Exercises

- Convert the following base 10 numbers to cartoonworld format: base 8
- 40
- 52
- 128
- 1500

- Convert the following cartoonworld (base 8) numbers to base 10:
- 30
- 500
- 173
- 1052

But how do the cartoonworld characters add and subtract?

- Obviously we have some new rules about what to do with single digit numbers.
- But for larger ones they can use the same long addition, (subtraction), multiplication (and division) algorithms we do.
- The important ones for us are addition and multiplication.

Long addition

- Remember how we do it (I've put in the carries):

$$
\begin{array}{r}
7263 \\
5168\ + \\
\hline
124_{1}3_{1}1
\end{array}
$$

Over in cartoonworld

- It works just the same way:
- $6+3 = 9_{10} = 11_8$: Put 1, carry 1
- $2+5+1 = 8_{10} = 10_8$: Put 0, carry 1

$$\begin{array}{rl} 4126 & \\ 5253 & + \\ \hline 114_{1}0_{1}1 & \end{array}$$

Let’s check

- 4126_8 = 4*512 + 64*1 + 2 * 8 + 6

  = 2048 + 64 + 16 + 6 = 2048 + 80 + 6 = 2134
- 5253_8 = 5 * 512 + 2 * 64 + 5 * 8 + 3

  = 2560 + 128 + 40 +3 = 2600 + 128 + 3 = 2731
- Adding these we get 4865_{10}
- Our answer was 11401_8
- 11401_8 = 1 * 4096 + 1 * 512 + 4 * 64 + 1

  = 4096 + 512 + 256 + 1 = 4608 + 256 + 1 = 4854 + 1

  = 4865_{10}
- Check.

Multiplication

- Long multiplication works in just the same way
- I’m not going to give you an example of the kind of long multiplication you did in primary school.
- We’ll just go straight to cartoonworld.

Multiplication

$$\begin{array}{rl}
1\ 3\ 5\ 6 & \\
2\ 4 & * \\
\hline
5_1\ 6_2 7_3 0 & \\
2\ 7\ 3_1\ 4_1 0 & \\
\hline
3_1 5_1\ 2_1 3\ \ 0 &
\end{array}$$

Check again

- $1356_8 = 1 * 512 + 3 * 64 + 5 * 8 + 2$

  $= 512 + 192 + 40 + 2 = 512 + 234 = 750_{10}$
- $24_8 = 2 * 8 + 4 = 16 + 4 = 20_{10}$
- $750 * 20 = \underline{15000_{10}}$
- $35230_8 = 3*4096 + 5*512 + 2 * 64 + 3 * 8$

  $= 12288 + 2560 + 128 + 24$

  $= 12288 + 2560 + 152$

  $= 12340 + 2560$

  $= \underline{15000_{10}}$
- Check.

The key point

- If stuff works in Base 10 (our normal)
- and it works in Base 8 (cartoonworld)
- then it works in Base 2 (computers).

End of interlude: cartoon world

But what has this to do with Computers?

- As we all know, computers work in Binary.
- They use bits, which are either 0 or 1
- In other words they work in Base 2.

Bits and Bytes

- One binary digit is called a **bit**
- **Eight (8) bits** are called a **byte**
- The smallest directly addressable unit of memory is usually at least a byte (ie you always get at least 8 bits), and is always a whole number of bytes.

Bits and Bytes

- Watch out for the difference:
- Storage is usually given in **bytes**:
  - a terabyte hard drive
- Network capacity is usually given in **bits**:
  - gigabit ethernet

Numbers

Different kinds of numbers

- Computers support different kinds of numbers:
  - unsigned integer
  - signed integer
  - floating point reals
- They come in different flavours (relating to the storage allocated).
  - these are usually 32 or 64 bit (single or double), but sometimes 128 bit and sometimes smaller.

Different kinds of numbers

- We will concentrate on the basic classes:
  - unsigned integer
  - signed integer
  - floating point reals

Unsigned integers and bit sequences

Unsigned integers and bit sequences

- These are the simplest.
- Unsigned integer means a whole number that only has a size and not a sign (positive or negative).
- Think of this as a positive integer: positive whole number.

Unsigned integers and bit sequences

- Computers do not make much distinction between unsigned integers and (fixed length) bit sequences.

- If your computer supports 32-bit unsigned integers, then it supports integers from 0 to 2^{32}-1 = 4294967295 (4,294,967,295)

Unsigned integers and bit sequences

- A bit sequence can be read as a positive integer written in binary, and hence an unsigned integer.

- An unsigned integer, can be expressed in binary, given leading 0’s if necessary, and hence produces a bit sequence.

- There is an exact one to one correspondence between unsigned integers from 0 to 4,294,967,295 and 32-bit bit sequences.

0	0000 0000 0000 0000 0000 0000 0000 0000
1	0000 0000 0000 0000 0000 0000 0000 0001
2	0000 0000 0000 0000 0000 0000 0000 0010
3	0000 0000 0000 0000 0000 0000 0000 0011
4	0000 0000 0000 0000 0000 0000 0000 0100
5	0000 0000 0000 0000 0000 0000 0000 0101
6	0000 0000 0000 0000 0000 0000 0000 0110
7	0000 0000 0000 0000 0000 0000 0000 0111
8	0000 0000 0000 0000 0000 0000 0000 1000
…	…
4294967295	1111 1111 1111 1111 1111 1111 1111 1111

Positive numbers

To represent a positive number:

- write it in Binary (Base 2), eg 14_{10}=1101
- encode as a sequence of bits

Skills: conversion between binary and decimal

- Given a binary sequence, you should understand that it represents a number and be able to convert it to its decimal equivalent.

- Given an unsigned integer in decimal, you should be able to convert it to binary, either as a binary number (no leading 0’s), or a bit sequence of given length.

Method: Conversion from binary to decimal.

- Take a number in binary: 1011011
- Write the powers of 2 above or below the digits (we're using a table).

1	0	1	1	0	1	1
64	32	16	8	4	2	1

Method 1: Conversion from binary to decimal.

- Take a number in binary: 1011011
- Add all the entries that have a 1 above:
  - 64 + 16 + 8 + 2 + 1 = 91

1	0	1	1	0	1	1
64	32	16	8	4	2	1

Class Exercise

- 10011

Vote

1. 37
2. 19
3. something else

Class Exercise

- 10011

1	0	0	1	1
16	8	4	2	1

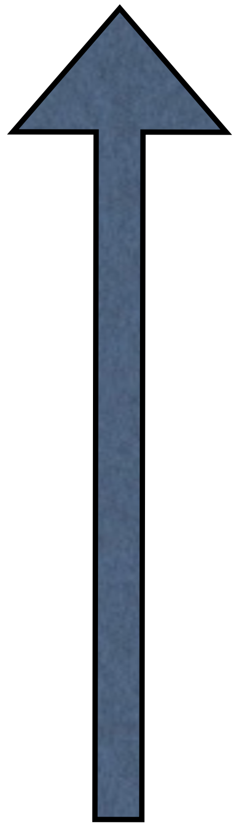
16 + 2 + 1 = 19

Method 2: Conversion from decimal to binary

- This method is called **“successive division”**
- It uses a loop.
- In the loop body you divide by two and keep the remainder.
- Then you put all the remainders together to get the final result.

Successive division

91	div 2	remainder
	45	1
	22	1
	11	0
	5	1
	2	1
	1	0
	0	1



Result: 1011011

Class exercise

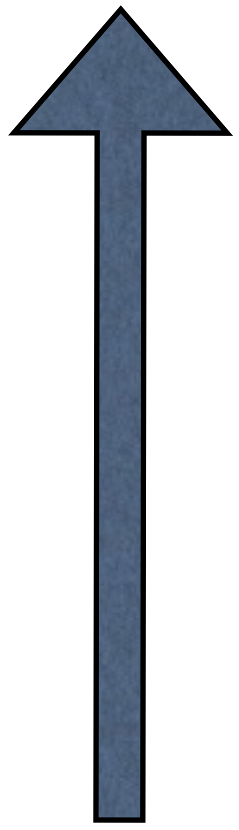
Successive division

- Successive division can be used to translate numbers into any given base.
- Instead of dividing by 2 we divide by the base.

Example: Base 10

Successive division

1,865	div 10	remainder
	186	5
	18	6
	1	8
	0	1



Method 3: Conversion from binary to decimal.

- Reverse the successive division method.
- Take a number in binary: 1011011

Method 3: Conversion from binary to decimal.

- Reverse the successive division method.
- Take a number in binary: 1011011

	count*2	+
1	0	1
0	2	2
1	4	5
1	10	11
0	22	22
1	44	45
1	90	91

Result: 91

Method 3: Conversion from binary to decimal.

- This is the way you'd implement it on a computer.
- Take a number in binary: 1011011

	count*2	+
1	0	1
0	2	2
1	4	5
1	10	11
0	22	22
1	44	45
1	90	91

Result: 91

Method 3: Conversion from binary to decimal.

- Compare tables…

91	div 2	remainder
	45	1
	22	1
	11	0
	5	1
	2	1
	1	0
	0	1

	count*2	+
1	0	1
0	2	2
1	4	5
1	10	11
0	22	22
1	44	45
1	90	91

Result: 91

Binary

- Although you can convert binary to decimal, it's a good idea to learn the binary for 0..10.

No	Binary	No	Binary
0	0000	8	1000
1	0001	9	1001
2	0010	10	1010
3	0011	11	1011
4	0100	12	1100
5	0101	13	1101
6	0110	14	1110
7	0111	15	1111

Hexadecimal (Hex)

- Computers are good at reading long strings of binary.
- People are not.
- A 32 bit string: 1011 0001 0111 1100 0101 0000 1001 0010
- This is unintelligible, so we often present it in base **16** (**hexadecimal** or **hex**)

Hexadecimal (Hex)

- Computers are good at reading long strings of binary.
- People are not.
- A 32 bit string: 1011 0001 0111 1100 0101 0000 1001 0010
- This is unintelligible, so we often present such strings in base **16** (**hexadecimal** or **hex**)

Hexadecimal (Hex) digits

Decimal	Binary	**Hex**	Decimal	Binary	**Hex**
0	0000	**0**	8	1000	**8**
1	0001	**1**	9	1001	**9**
2	0010	**2**	10	1010	**a**
3	0011	**3**	11	1011	**b**
4	0100	**4**	12	1100	**c**
5	0101	**5**	13	1101	**d**
6	0110	**6**	14	1110	**e**
7	0111	**7**	15	1111	**f**

Hexadecimal (Hex) digits

Decimal	Binary	**Hex**	Decimal	Binary	**Hex**
0	0000	**0**	8	1000	**8**
1	0001	**1**	9	1001	**9**
2	0010	**2**	10	1010	**a**
3	0011	**3**	11	1011	**b**
4	0100	**4**	12	1100	**c**
5	0101	**5**	13	1101	**d**
6	0110	**6**	14	1110	**e**
7	0111	**7**	15	1111	**f**

Also use ABCDEF for 10-15.

Skills: conversion between binary and hexadecimal

- Given a hexadecimal sequence, you should be able to convert it to binary.
- Given a binary sequence, you should be able to convert it to its hexadecimal equivalent.

Skills: conversion between binary and hexadecimal

- Given a hexadecimal sequence, you should be able to convert it to binary.
- Given a binary sequence, you should be able to convert it to its hexadecimal equivalent.
- This is easier than converting between binary and decimal.

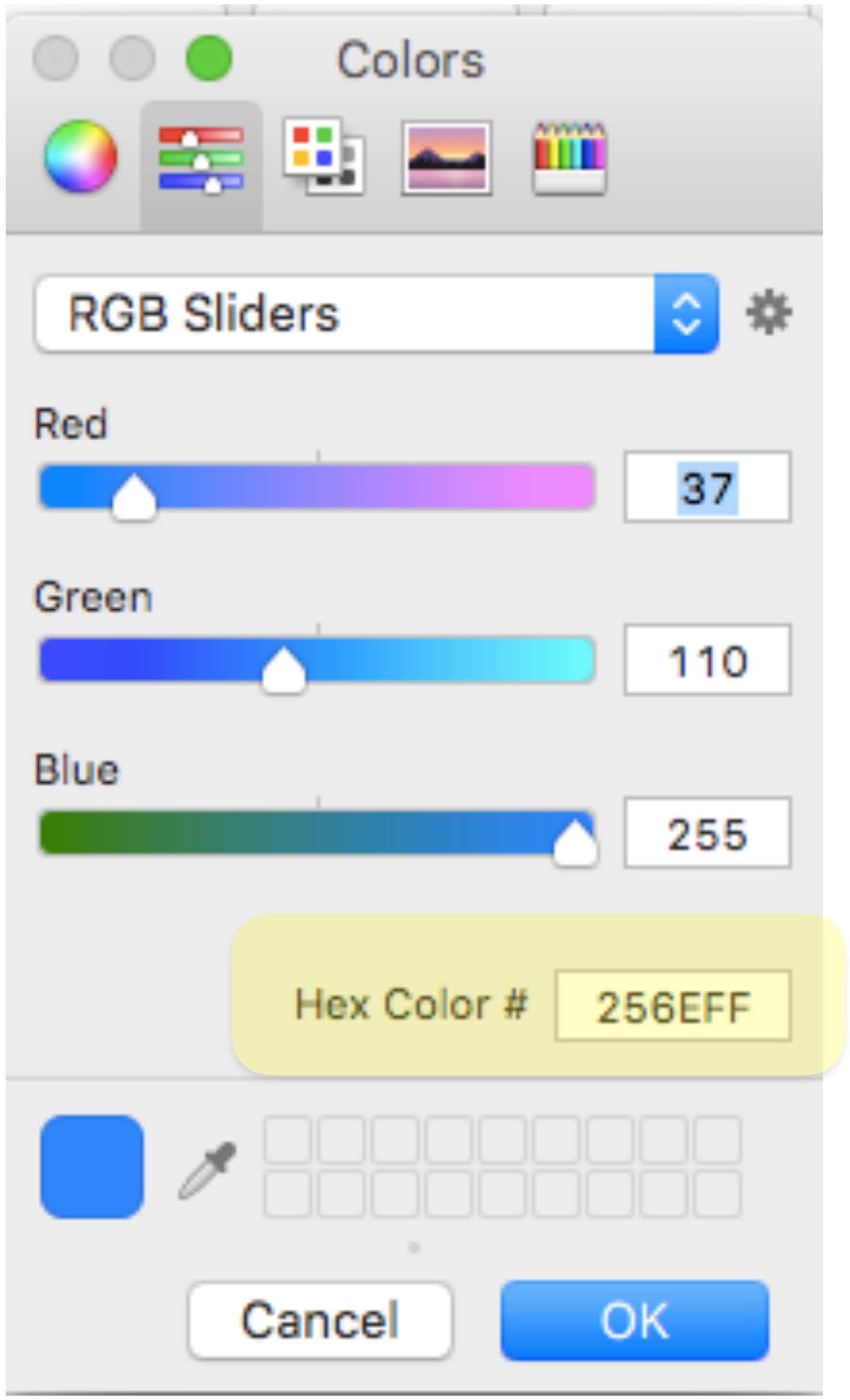
Method 4: Conversion between binary and hexadecimal

- Each hexadecimal digit is worth exactly four bits (see table). (So two hexadecimal digits are eight bits or one byte).

- This is because $16 = 2^4$

- So we can convert a hexadecimal string to binary character by character, converting each hexadecimal digit to four bits.

Method 4: Conversion from hexadecimal to binary


Colors

RGB Sliders

Red: 37

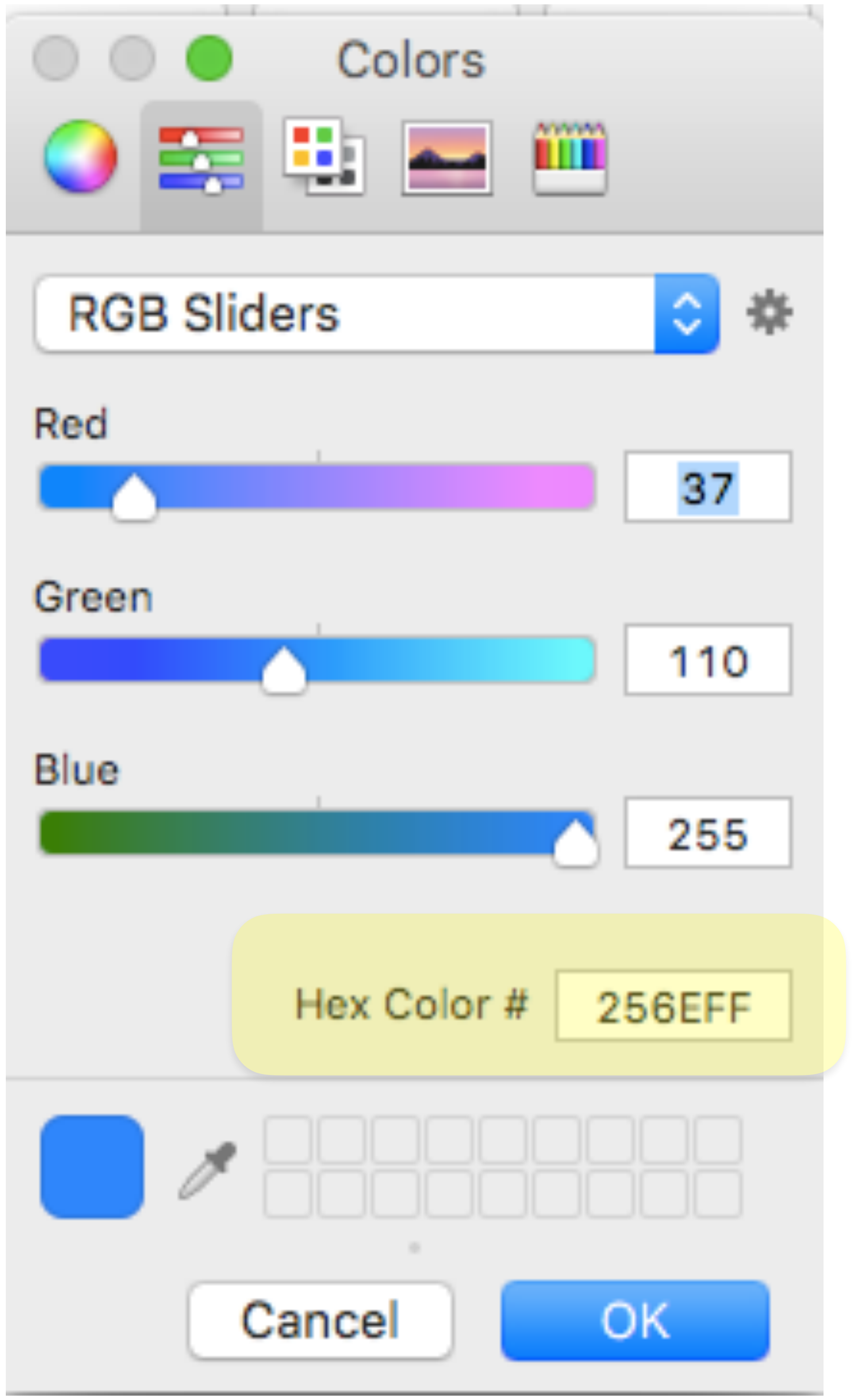
Green: 110

Blue: 255

Hex Color #: 256EFF

- Here is an example from "the wild".
- Hex Color is given as 256EFF.
- This is 6 hex digits, or two hex digits for each of Red Green and Blue.
- That makes 3 bytes or 24 bits.

Conversion from hexadecimal to binary



Colors

RGB Sliders

Red: 37

Green: 110

Blue: 255

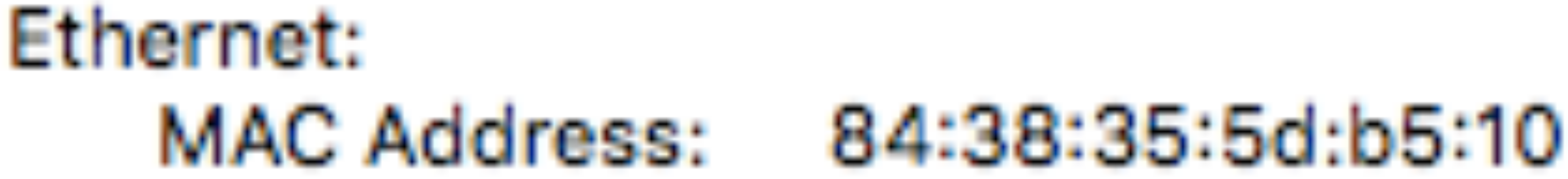
Hex Color # 256EFF

2	0010
5	0101
6	0110
E	1110
F	1111
F	1111

256EFF is

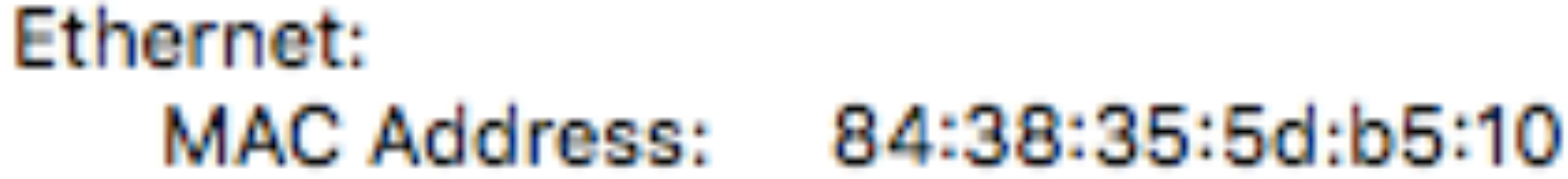
001001010110111011111111

Conversion from hexadecimal to binary


Ethernet:
MAC Address: 84:38:35:5d:b5:10

- This is a “MAC” Address
- 12 Hex digits
- Represents 48 bits.
- Exercise: translate to binary bit sequence

Conversion from hexadecimal to binary



```
Ethernet:
    MAC Address:    84:38:35:5d:b5:10
```

1000 0100 : 0011 1000 : 0011 0101 : 0101 1101 : 1011 0101 : 0001 0000

Method 5: Conversion from binary to hexadecimal

- Starting at right (least significant) divide binary into groups of four.
- Add leading 0’s to left hand group if necessary.
- Convert each group of four into Hex.

Method 5: Conversion from binary to hexadecimal

101111100010110

- Starting at right (least significant) divide binary into groups of four.

101 1111 0001 0110

Method 5: Conversion from binary to hexadecimal

101111100010110

- Starting at right (least significant) divide binary into groups of four.
- Add leading 0’s to left hand group if necessary.

101 1111 0001 0110

0101 1111 0001 0110

Method 5: Conversion from binary to hexadecimal

101111100010110

- Starting at right (least significant) divide binary into groups of four.
- Add leading 0's to left hand group if necessary.
- Convert each group of four into Hex.

0101 1111 0001 0110

5f16

Hex and binary

$$\underbrace{1\ 0\ 1\ 0}_{a}\ \underbrace{1\ 1\ 1\ 1}_{f}\ \underbrace{0\ 1\ 1\ 0}_{6}\ \underbrace{1\ 0\ 0\ 1}_{9}$$

- Each group of 4 bits corresponds to a single hex digit.
- In hex this number (bit pattern) is af69

Method 6: Binary Addition

- Add binary numbers using the standard long addition algorithm

$$\begin{array}{r} 1101 \\ 1010 + \\ \hline 10111 \end{array}$$

Addition

- Check result:
- $1101_2 = 13_{10}$
- $1010_2 = 10_{10}$
- $10111_2 = 23_{10}$

Subscript tells you which base we are using.

$$\begin{array}{r} 1101 \\ 1010\ + \\ \hline 10111 \end{array}$$

Addition: carries

- You may need to use carries..

```
  100101
  101110 +
   11000 carries
________
 1010011
```

Addition: carries

- Check:
- $100101_2 = 37_{10}$
- $101110_2 = 46_{10}$
- $1010011_2 = 83_{10}$

```
   100101
   101110 +
    11000 carries
 _________
  1010011
```

Multiplication

- Multiply positive binary numbers using the standard long multiplication

```
     1,101
     1,010 *
----------
      0000
    11,010
    000000
 1,101,000
----------
10,000,010
```

Check result

- $1101_2 = 1*2^3+1*2^2+0*2+1 = 1*8 + 1*4 + 1 = 13_{10}$
- $1010_2 = 1*2^3 + 1*2 = 8+2 = 10_{10}$
- $10000010 = 1*2^7 + 1*2 = 128 + 2 = 130_{10}$
- Check.

Notice

```
       1,101
       1,010 *
  ------------
        0000
      11,010
      000000
   1,101,000
  ------------
  10,000,010
```

Rows here are either 0 or just 1101 shifted to the left.

This makes the algorithm easy to implement in a machine.

Simplicity

- When you were at primary school you learned how to add two small numbers together (numbers less than ten).
- Then you learned an algorithm for adding large numbers - long addition.
- Similarly you learned subtraction and multiplication on single digit numbers, and then how to extend those to many digit numbers - long subtraction and multiplication.
- (Long division is a little different).

Long addition

- The long addition algorithm also works in arbitrary number bases.
- It tells you how to add two numbers expressed in positional notation if you already know how to add single digits.
- In other words it can be used for numbers in binary.
- (And similarly for long subtraction and multiplication).

But why do computers use binary?

Why do computers use binary?

There are lots of reasons why computers use binary.

- They boil down to:
    - simplicity
    - reliability
    - efficiency

Binary

Makes for simpler more reliable internals.

- It’s much easier to produce a light that is on or off than one that has (say) ten levels.
- It’s easier to decide whether it is on or off, than to decide which of ten levels it is shining at.
- We can use greater lengths of cable before our signal gets corrupted.
- We can send signals down the cables faster.
- And exactly the same things apply for eg voltage levels.

Binary is more stable

- We're trying to make the machinery as small as possible, and push it as fast as possible (and we've always been doing this).
- Since it's very small, and built to the limit of our engineering, components that are supposed to be the same don't behave quite the same.
- For example, we're making stuff out of very small transistors. The transistors are built out of silicon plus dopants. The transistors are so small that there are only a few atoms of dopant in each transistor. So the transistor behaviour depends on just how many, and just where they are.
- So it is much, much harder to build circuits that make use of several voltage levels, not just two.

Binary

- Remember: information is stored as a voltage
- Voltages do not suddenly magically change
- Changing a voltage requires that we move some charge (electrons) from one place to another.
- This does not happen instantly. It takes time for the charge to move, build up, stabilise and supply the voltage we want.
- If we try to run the circuit very quickly, then the price we pay is that the result of one operation is not stable before we start the next.
- This means that the voltage we get as an output has an error in it.

Errors

- Let’s suppose we’re running a decimal system at 5v
- Then we want to read voltage levels at 0v, 0.5v,1v,1.5v,2v, 2.5v,...
- But in fact we’ll get e.g. 2.5v +/- say 0.1v
- Now try adding two of these and we expect to get error of 2*0.1v = 0.2v
- Add two of these again, and we get a possible error of 0.4v
- This means you can’t reliably tell which number the voltage is supposed to represent.
- For example: 1.7v could represent 1.5v or 2v.

Errors, again

- That means we have to design a circuit that will stabilise the results into one of ten bins before we pass them on.
- This is hard.
- It is much easier to design circuits that are stable at the whole 5v or 0v (i.e. to use binary).

Binary

- We will see later that algorithms, such as addition and multiplication are simpler for binary than they are for decimal.

- And binary fits very well with our basic control structures.

- And what's more, using another system with a fixed number of characters would not materially improve efficiency of representation. (We could represent each of those characters with a fixed length binary string).

Simplicity

- If you want to do addition, then you can use the long addition method *provided* you can add single digits.
- Here is the single digit addition table for binary:

+	0	1
0	0	1
1	1	10

- Compare

+	0	1
0	0	1
1	1	10

- with the table you learnt at school.

+	0	1	2	3	4	5	6	7	8	9
0	0	1	2	3	4	5	6	7	8	9
1	1	2	3	4	5	6	7	8	9	10
2	2	3	4	5	6	7	8	9	10	11
3	3	4	5	6	7	8	9	10	11	12
4	4	5	6	7	8	9	10	11	12	13
5	5	6	7	8	9	10	11	12	13	14
6	6	7	8	9	10	11	12	13	14	15
7	7	8	9	10	11	12	13	14	15	16
8	8	9	10	11	12	13	14	15	16	17
9	9	10	11	12	13	14	15	16	17	18

It’s easier to get a complete design in hardware

- For a complete design we need to be able to get all possible functions.
- In decimal, there are ten (10) 1-digit constants.
- There are ninety (90) non-constant functions of one single-digit argument producing a single-digit result.
- There are 10^{100} functions of two single-digit arguments producing single-digit results (only 190 of which are constant, or really just functions of one argument).

It’s easier to get a complete design in hardware

- For a complete design we need to be able to get all possible functions.
- In binary, there are two (2) 1-digit constants.
- There are two (2) non-constant functions of one single-digit argument producing a single-digit result.
- There are 16 functions of two single-digit arguments producing single-digit results (6 of which are constant, or really just functions of one argument). **Moreover we can get all of them from standard boolean functions “and” “or” and “not”.**

So binary it is.