CS1021 Introduction to Computing I
1. Computer Memory

Rebekah Clarke
clarker7@scss.tcd.ie

Memory

A **bit** (***b****inary dig****it***) is a unit of information which has only two possible states

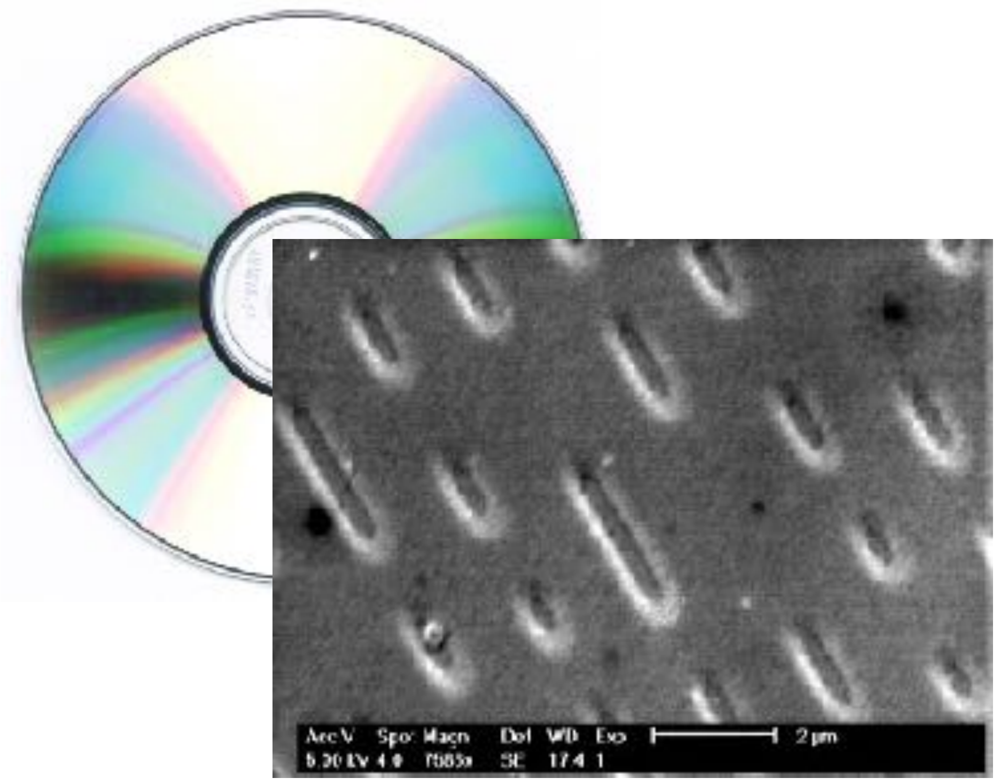
0 or 1, on or off, true or false, purple or gold, sitting or standing….

Bits are the fundamental unit of data storage in a computer

Computer Memory is (usually) implemented as a collection of electronic switches which take one of two states

Other devices use different physical implementations of bits e.g.

Hard disk drives use magnetism



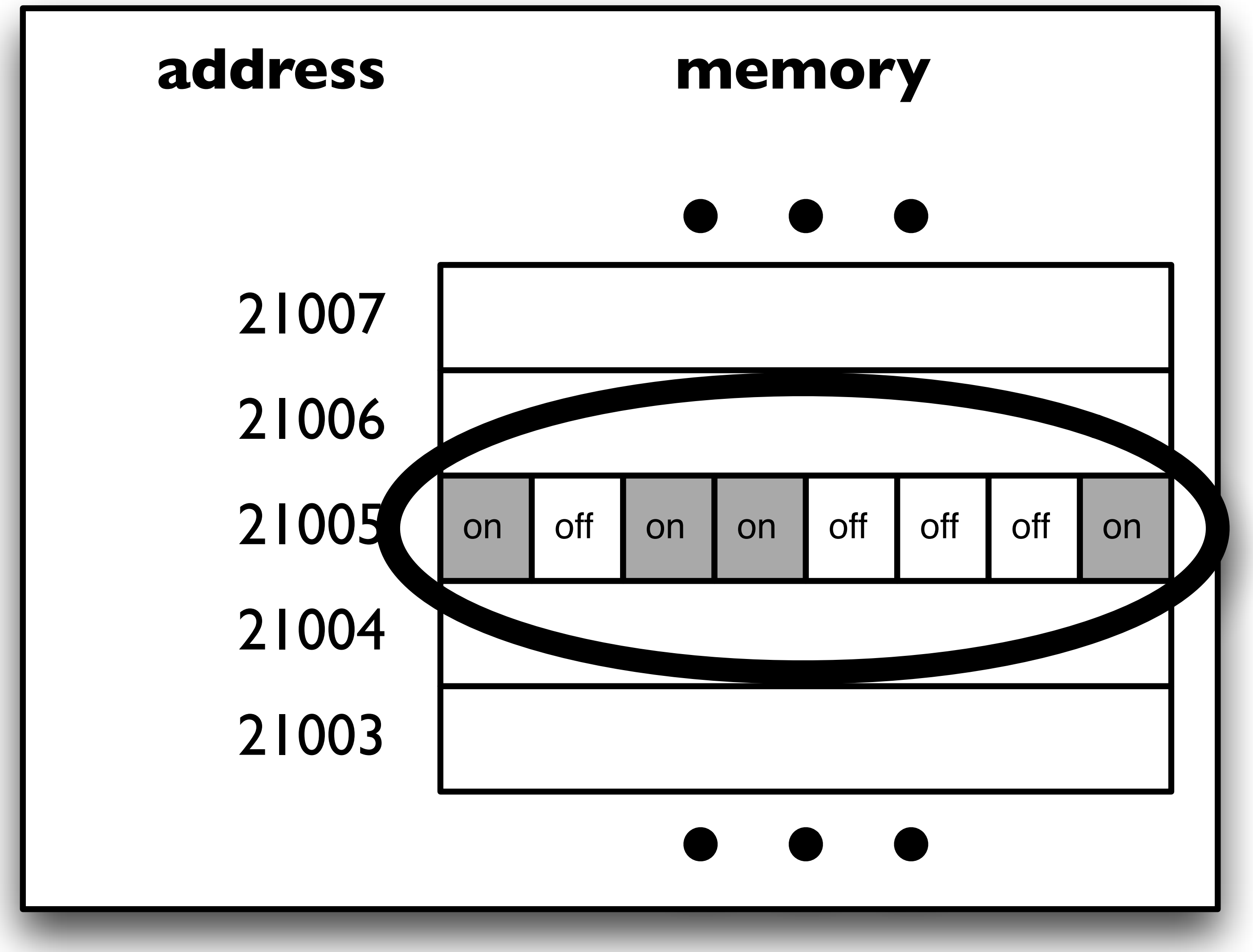
CDs use dented grooves

Bytes

Accessing each bit individually isn't very useful.
We want to store data that can take a wider range of values,

- e.g. the value 214, the letter "b", an image

By grouping bits together we can store a wider range of unique values (wider = more than 2)



Smallest "addressable" unit of memory storage storage is the byte

8 bits = 1 byte

Each Memory address refers to a region of 8 bits

Decimal Numeral System (base 10)

Usually use decimal (base-10) numeral system

Symbols (digits) that can represent ten integer values

0, 1, 2, 3, 4, 5, 6, 7, 8, 9

Represent integer values larger than 9 by using two or more digits

10, 11, 12, ..., 112, ..., 247

e.g.: 247

$= (7 \times 10^0) + (4 \times 10^1) + (2 \times 10^2)$

2 is the **Most Significant Digit**

7 is the **Least Significant Digit**

Binary Numeral System (base 2)

Computer systems store information electronically using bits (binary digits)

Each bit can be in one of two states, which we can take to represent the binary (base-2) digits 0 and 1

So, the binary number system is a natural number system for computing

Using a single bit, we can represent integer values 0 and 1

i.e. two different values

Using two bits, we can represent 00, 01, 10, 11

i.e. four different values

Converting between binary and decimal

Converting from binary to decimal

e.g. $100101 = (1 \times 2^5)+(0 \times 2^4)+(0 \times 2^3)+(1 \times 2^2)+(0 \times 2^1)+(1 \times 2^0) = 37$

100101

$2^5=32$ $2^4=16$ $2^3=8$ $2^2=4$ $2^1=2$ $2^0=1$

Converting from decimal to binary

2	37	1
2	18	0
2	9	1
2	4	0
2	2	0
2	1	1
	0	

read from bottom

$= 100101_2$

Numeral Systems

Given n decimal digits, what range of non-negative integers can we represent?

$[0 \ldots (10^n - 1)]$

e.g., n = 3 allows us to represent values in the range [0 ... 999]

In General, a number system with base b and n bits can represent $\mathbf{b^n}$ numbers (including 0)

The **range** of a number system with n bits is from 0 to $b^n - 1$

e.g. 4-bit Binary $0 \ldots 2^4 - 1$ i.e. 0…15

How many unique values can we represent with, for example, eight bits?

[0 ... 11111111] in binary notation

$[0 \ldots (2^8 - 1)] = [0 \ldots 255]$ in decimal notation

256 unique values

Base subscript notation

The same sequence of symbols can have a different meaning depending on the base being used

Use subscript notation to denote the base being used

$12_{10} = 1100_2$

$1_{10} = 1_2$

Using binary all the time would become quite tedious

e.g. The CS1021 exam is worth 1000110_2% of the final mark

Hexadecimal Numeral System

Base-16 (hexadecimal or “hex”) is a more convenient number system for computer scientists:

With binary, we needed 2 symbols (0 and 1)

With decimal, we needed 10 symbols (0, 1, ..., 9)

With hexadecimal, we need 16 symbols

Use the same ten symbols as the decimal system for the first ten hexadecimal digits

Borrow the first six symbols from the alphabet for the last six symbols

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

Why is hexadecimal useful?

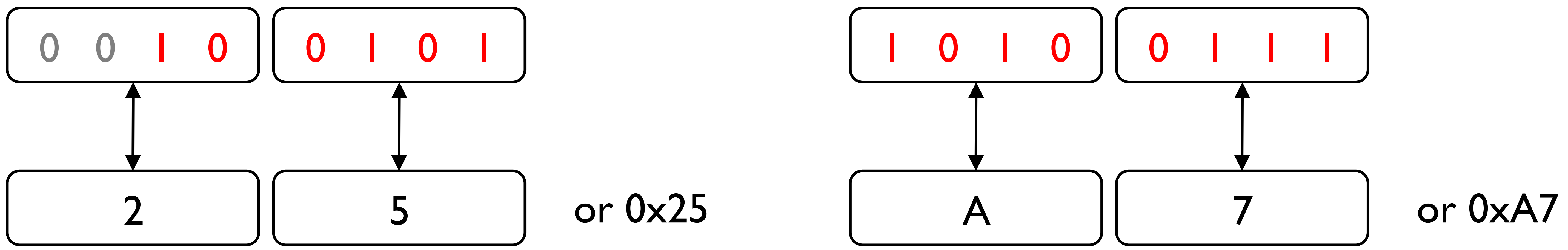
16 is a power of 2 (2^4), so **exactly one hex digit can represent the same sixteen values as four binary digits**

Converting between binary and hex

One hexadecimal digit represents the same number of values as four binary digits

conversion between hex and binary is trivial

the hexadecimal notation a convenient one for us



Hexadecimal is used by convention when referring to memory addresses: e.g. address 0x1000, address 0x4002

Decimal, Binary and Hexadecimal

base 10	base 2	base 16
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Alternative base notation

Without a fancy word processor, we won't be able to use the subscript notation to represent different bases

1010_2

How would we tell a computer whether we mean 1010 or 1010?

Instead we can prefix values with symbols that provide additional information about the base

In **ARM Assembly Language** (which we will be using) we use the following notation:

1000	No prefix usually means decimal
0x1000	Hexadecimal (used often)
&1000	Alternative hexadecimal notation
2_1000	Binary
n_1000	Base n

Bytes, Halfwords and Words

Remember

8 bits = 1 byte

with 8 bites we can represent 2^8 = 256 unique values

Sometimes useful to group more (than 8) bits together to store an even wider range of unique values

2 bytes = 16 bits = 1 halfword

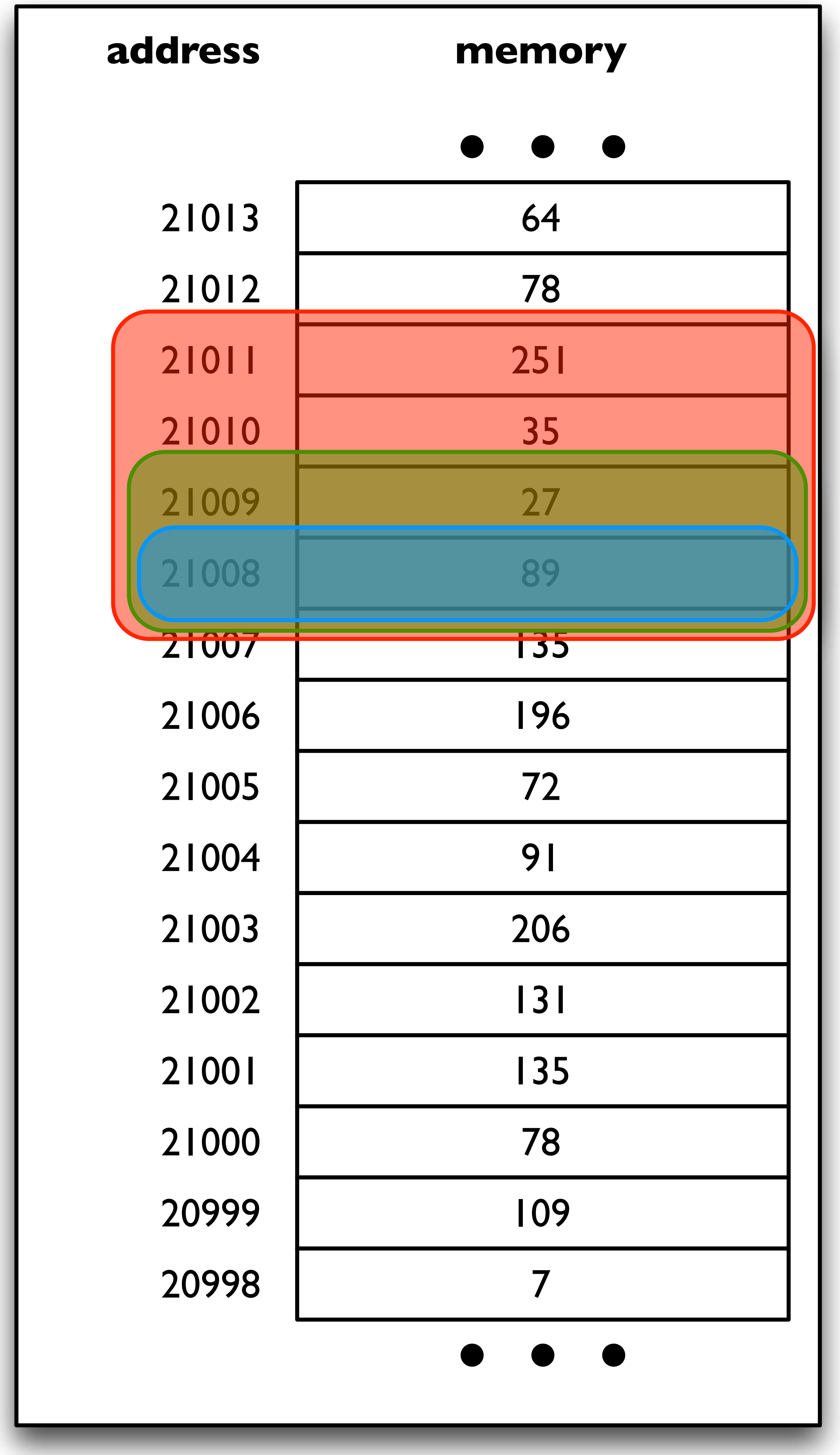
4 bytes = 32 bits = 1 word

When we refer to memory locations by address (using the ARM microprocessor), we can only do so in units of **bytes**, **halfwords** or **words**

the byte at address 21008

the halfword at address 21008

the word at address 21008



Larger units

Larger units of information storage

1 **kilobyte** (kB) = 2^{10} bytes = 1,024 bytes

1 **megabyte** (MB) = 1,024 KB = 2^{20} bytes = 1,048,576 bytes

1 **gigabyte** (GB) = 1,024 MB = 2^{30} bytes = ...

The following units of groups of bits are also used, usually when expressing **data rates** (e.g. Mbits/s):

1 **kilobit** (kb) = 1,000 bits

1 **megabit** (Mb) = 1,000 kilobits = 1,000,000 bits

IEC prefixes, KiB, MiB, GiB, …

Note: The storage industry standard is to display capacity in decimal.

Representing text

So far, we have only considered how computers store (non-negative) integer values using binary digits

What about representing other information, for example text composed of alphanumeric symbols?

‘T’, ‘h’, ‘e’, ‘ ’, ‘q’, ‘u’, ‘i’, ‘c’, ‘k’, ‘ ’, ‘b’, ‘r’, ‘o’, ‘w’, ‘n’, ‘ ’, ‘f’, ‘o’, ‘x’, …

We’re still restricted to storing binary digits (bits) in memory

To store alphanumeric symbols or “characters”, we can assign each character a value which can be stored in binary form in memory

ASCII

American Standard Code for Information Interchange

ASCII is a standard used to encode alphanumeric and other characters associated with text

- e.g. representing the word “hello” using ASCII

‘H’	‘E’	‘L’	‘L’	‘O’
0x48	0x45	0x4C	0x4C	0x4F

Each character is stored in a single byte value (8 bits)

- 1 byte = 8 bits means we can have a possible 256 characters
- In fact, ASCII only uses 7 bits, giving 128 possible characters
- Only 96 of the ASCII characters are **printable**
- Remaining values are **control codes**

ASCII

	0	1	2	3	4	5	6	7
0	NUL	DLE	SPACE	0	@	P	`	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	“	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB		7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	|
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	_	o	DEL

e.g. “E” = 0x45

ASCII

The value 0 is not the same as the character ‘0’

Similarly, the value 1 is not the same as the character ‘1’

1+1 = 2 but ‘1’+‘1’=?

The ASCII characters ‘0’, ‘1’, ... are used in text to display values in human readable form, **not for arithmetic**

Upper and lower case characters have different codes

The first printable character is the space symbol, ‘ ’ and it has code 32_{10} (sometimes written ␣)

It is almost always more efficient to store a value in its “value” form than its ASCII text form

- the value 10_{10} (or 1010_2) requires 1 byte
- the ASCII characters ‘1’ (0x31) followed by ‘0’ (0x30) require 2 bytes (1 byte each)
- we cannot perform arithmetic, comparison, etc. directly using the ASCII characters