

# Unit 2: Computer architecture

## 1. Binary number systems

What is the simplest form of information you could transmit as a message?

- A word? - A humble 5 letter word has 26<sup>5</sup> possible combinations. That's almost 12 million.
- A letter? - 26 possibilities (52 if we distinguish between upper and lower case)
- A numerical digit? - Still 10 possibilities
- A light bulb! - just 2 possible states! on or off.

In Computer Science, this **on** or **off** value is known as a bit, and can be thought of the literal presence or absence of an electrical charge within a transistor. We tend to simplify it in programming to refer to it as a **1** or **0**.

Our modern computer processors are made of millions of these transistors, each of which is on or off at a given moment in time, and it is on this foundation that our entire world of computing is built upon! Computers take the simple on/off values of a transistor, and scale them up to create the complexity we know and love.

Given each bulb can have two states, "on or off", or 2 possible combinations, how many total possible combinations are possible with 8 light bulbs?



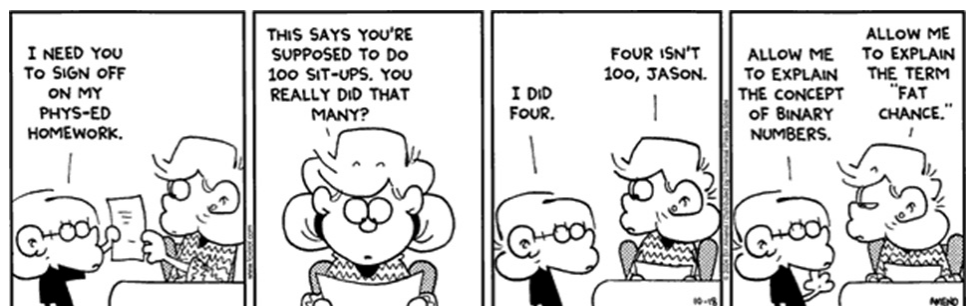
A cluster of 8 bits is known as one byte.

If we replace the off/on of electrical charge with 0/1, we can establish a number system. The number system built around bits is known as the binary number system.



Heliosphere (2017): Binary - How to make a computer: Part II  
<https://www.youtube.com/watch?v=NRKORzi5tnM> (7m15)

So far we've been talking in binary, which is a base 2 number system (2 possible values per place column). Historically we are also familiar with the decimal number system, or base 10. In Computer Science there are others for you to be familiar with that we will now look at.



## Octal

- Octal was briefly mentioned in the previously watched video at <https://youtu.be/NRKORzi5tnM>
- Octal is a base 8 numbering system. Think of it as the numbering system we would have if humans only had 4 fingers (including the thumb) per hand instead of 5! You can only count from 0 to 7 after which you run out of digits and so need to increment the next column (the "tens" column).
- The least significant column is the 1s, the next column is the number of 8s, and the column after that is the number of 64s ( $8^2$ ).
- The convenience of Octal is that base 8 makes octal counting equivalent to a 3 digit binary number!

Dec	Hex	Oct	Bin
0	0	000	0000
1	1	001	0001
2	2	002	0010
3	3	003	0011
4	4	004	0100
5	5	005	0101
6	6	006	0110
7	7	007	0111
8	8	010	1000
9	9	011	1001
10	A	012	1010
11	B	013	1011
12	C	014	1100
13	D	015	1101
14	E	016	1110
15	F	017	1111

## Hexadecimal

- Hexadecimal is a base 16 numbering system. Base 16 means it has 16 numerals for each column.
- The least significant column is the 1s, the next column is the number of 16s, and the column after that is the number of 256s ( $16^2$ ).
- Hexadecimal is even more convenient than octal because each hexadecimal numeral can represent a 4 digit binary number, resulting in a two digit hexadecimal being able to convey one byte!

### Practice conversions!

Decimal	Binary	Octal	Hexadecimal
140			
71			
	0110 1010		
	1001 1111		
		300	
		123	
		222	
			A4
			8F
			B9

## 2. Representing data

### 2.1 ASCII

As powerful as computers can be using bits and bytes, humans don't intuitively operate on that level, we use characters and words, so there needed to be a way for letters (and thereby words) to be represented somehow within a computer.

To achieve this, an arbitrary table was decided on in the 1960s called the ASCII table. This table still forms the basis of much of today's character representation system. The significance of this table is that a "string of letters" is really just a sequence of binary that, according to this lookup table, can be used to represent alphanumeric text.

01001000    01100101    01101100    01101100    01101111  
 H            e            |            |            o

#### Decimal - Binary - Octal - Hex – ASCII Conversion Chart

Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII
0	00000000	000	00	NUL	32	00100000	040	20	SP	64	01000000	100	40	@	96	01100000	140	60	`
1	00000001	001	01	SOH	33	00100001	041	21	!	65	01000001	101	41	A	97	01100001	141	61	a
2	00000010	002	02	STX	34	00100010	042	22	"	66	01000010	102	42	B	98	01100010	142	62	b
3	00000011	003	03	ETX	35	00100011	043	23	#	67	01000011	103	43	C	99	01100011	143	63	c
4	00000100	004	04	EOT	36	00100100	044	24	\$	68	01000100	104	44	D	100	01100100	144	64	d
5	00000101	005	05	ENQ	37	00100101	045	25	%	69	01000101	105	45	E	101	01100101	145	65	e
6	00000110	006	06	ACK	38	00100110	046	26	&	70	01000110	106	46	F	102	01100110	146	66	f
7	00000111	007	07	BEL	39	00100111	047	27	'	71	01000111	107	47	G	103	01100111	147	67	g
8	00001000	010	08	BS	40	00101000	050	28	(	72	01001000	110	48	H	104	01101000	150	68	h
9	00001001	011	09	HT	41	00101001	051	29	)	73	01001001	111	49	I	105	01101001	151	69	i
10	00001010	012	0A	LF	42	00101010	052	2A	*	74	01001010	112	4A	J	106	01101010	152	6A	j
11	00001011	013	0B	VT	43	00101011	053	2B	+	75	01001011	113	4B	K	107	01101011	153	6B	k
12	00001100	014	0C	FF	44	00101100	054	2C	,	76	01001100	114	4C	L	108	01101100	154	6C	l
13	00001101	015	0D	CR	45	00101101	055	2D	-	77	01001101	115	4D	M	109	01101101	155	6D	m
14	00001110	016	0E	SO	46	00101110	056	2E	.	78	01001110	116	4E	N	110	01101110	156	6E	n
15	00001111	017	0F	SI	47	00101111	057	2F	/	79	01001111	117	4F	O	111	01101111	157	6F	o
16	00010000	020	10	DLE	48	00110000	060	30	0	80	01010000	120	50	P	112	01110000	160	70	p
17	00010001	021	11	DC1	49	00110001	061	31	1	81	01010001	121	51	Q	113	01110001	161	71	q
18	00010010	022	12	DC2	50	00110010	062	32	2	82	01010010	122	52	R	114	01110010	162	72	r
19	00010011	023	13	DC3	51	00110011	063	33	3	83	01010011	123	53	S	115	01110011	163	73	s
20	00010100	024	14	DC4	52	00110100	064	34	4	84	01010100	124	54	T	116	01110100	164	74	t
21	00010101	025	15	NAK	53	00110101	065	35	5	85	01010101	125	55	U	117	01110101	165	75	u
22	00010110	026	16	SYN	54	00110110	066	36	6	86	01010110	126	56	V	118	01110110	166	76	v
23	00010111	027	17	ETB	55	00110111	067	37	7	87	01010111	127	57	W	119	01110111	167	77	w
24	00011000	030	18	CAN	56	00111000	070	38	8	88	01011000	130	58	X	120	01111000	170	78	x
25	00011001	031	19	EM	57	00111001	071	39	9	89	01011001	131	59	Y	121	01111001	171	79	y
26	00011010	032	1A	SUB	58	00111010	072	3A	:	90	01011010	132	5A	Z	122	01111010	172	7A	z
27	00011011	033	1B	ESC	59	00111011	073	3B	;	91	01011011	133	5B	[	123	01111011	173	7B	{
28	00011100	034	1C	FS	60	00111100	074	3C	<	92	01011100	134	5C	\	124	01111100	174	7C	
29	00011101	035	1D	GS	61	00111101	075	3D	=	93	01011101	135	5D	]	125	01111101	175	7D	}
30	00011110	036	1E	RS	62	00111110	076	3E	>	94	01011110	136	5E	^	126	01111110	176	7E	~
31	00011111	037	1F	US	63	00111111	077	3F	?	95	01011111	137	5F	_	127	01111111	177	7F	DEL

This work is licensed under the Creative Commons Attribution-ShareAlike License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-sa/3.0/>

ASCII Conversion Chart.doc Copyright © 2008, 2012 Donald Weiman 22 March 2012

**Practice: Convert the following ASCII to it's binary representation.**

01110100 - 01101000 - 01101001 - 01110011  
00100000 - 01101001 - 01110011 - 00100000  
01110011 - 01101100 - 01101111 - 01110111  
00100000 - 01100111 - 01101111 - 01101001  
01101110 - 01100111

## 2.2 Unicode

ASCII was incredibly useful and opened up a world of computing accessible to a lot of people, but there are still significant limitations. ASCII was built on an 8 bit / 1 byte conversion table. That means there are only 256 possible characters that can be used for conversion. While this is generally fine for Latin based languages such as English, it imposes restrictions on how multilingual computing is capable of being.

The solution to overcome this was the development of the UNICODE standard, which was published in 1991. UNICODE is a 16 bit lookup table (65536 possible values). While this means it takes 2 bytes to store every letter, the cost of data storage has fallen significantly enough that that is not a major problem. The upside is it means all Asian characters etc can now be represented.

To look at the Unicode table for yourself, check out @ <http://unicode-table.com/en/>



Computerphile (2013) Characters, Symbols and the Unicode  
Miracle by Tom Scott!  
<https://www.youtube.com/watch?v=MijmeoH9LT4> (9m36)

## 2.3 Negative integers

### NOT REQUIRED FOR CURRENT DIPLOMA COMPUTER SCIENCE COURSE

In the examples we have been looking at so far, we have used the entire byte to represent a number: 8 bits to represent values 0 to 255. In reality computers need to be able to cater to negative values as well, so the most significant bit is actually reserved to indicate the sign (positive or negative) of the number. This system is known as two's complement, or having a signed integer. To use the full size of the binary number for a positive only value is known as having an unsigned integer.

0000 0011	3
0000 0010	2
0000 0001	1
0000 0000	0
1111 1111	-1
1111 1110	-2
1111 1101	-3
1111 1100	-4

Notice that this will mean the number range is greater for the negatives than the positives. For an 8 bit integer, the decimal values will range from +127 to -128.

Converting negative decimals to two's complement binary (example using -13):

- Obtain the positive value of your decimal number (13)
- Subtract one (12)
- Convert to binary (0000 1100)
- Flip all the bits (1111 0011)

Converting two's complement binary to negative decimal (example using 1110 1110):

- Flip all the bits (0001 0001)
- Convert to decimal (17)
- Add one (18)
- Add the negative sign (-18)

By using two's complement, binary addition and subtraction work quite simply. Some examples:

Addition of  $68 + 12$

```
    1 1    (carry over row)
  0100 0100 (68)
+ 0000 1100 (12)
  0101 0000 (80)
```

Subtraction of  $68 - 12$ , which is really  $68 + (-12)$

```
    1 1    (carry over row)
  0100 0100 (+68)
+ 1111 0100 (-12)
  0011 1000 (+56)
```

Subtraction of  $12 - 68$ , which is really  $12 + (-68)$

```
    111 1    (carry over row)
  0000 1100 (+12)
+ 1011 1100 (-68)
  1100 1000 (-56)
```

## 2.4 Floating point numbers

Tom Scott has another excellent introduction to this concept in the following Computerphile video.



Computerphile (2014) Floating Point Numbers by Tom Scott  
<https://www.youtube.com/watch?v=PZR11fStY0> (9m15)

A good written summary can be found here

floating-point-gui.de (undated): Floating Point Numbers  
<https://floating-point-gui.de/formats/fp/>

For our purposes, you don't need to be able to do manual conversions with floating point numbers, you just need to understand the concept, its limitations and workarounds (as Tom Scott outlines in his video).

A good illustration of the problems with floating point numbers would be to run the following code

```
# Python
a = 0.1
b = a + a + a
print(b) # What do you expect to print? What actually prints?
```

What is the cause of the unexpected output from the above code snippet? As a programmer, how should you mitigate this in your applications?

Internally, floating point numbers are stored like scientific notation by recording two integers. The first represents the value component, the second represents the power of the exponent. In this way, they can store very large and very small numbers but with a limited degree of accuracy. The 64 bit floating point number uses 1 bit for the sign (positive/negative), 8 bits for the exponent, and 55 bits for the significant number.

For example, the speed of light may be represented as  $3.0 \times 10^8$  (m/s) using scientific notation, so as a floating point number this would be the integers 3 and 8. The catch is to remember everything is using binary numbers. This means the bits in the "value" component represent  $1/2$ , then  $1/4$ , then  $1/8$  and so forth; and the exponent is for powers of 2 rather than 10. Our example number of  $3.0 \times 10^8$  is actually  $1.00011110000110100011 \times 2^{28}$ .

Try some numbers for yourself here, <https://www.exploringbinary.com/floating-point-converter/>

## 2.5 Color

If you've used Photoshop, you have probably seen colours expressed as #FF0000. You should now be able to recognise this type of number as hexadecimal.

What is the value of the number? Colours in the computer are actually split into RGB – Red, Green Blue. One unsigned byte (256 values) for each.

So, #FF0000 is actually:

FF for Red, 00 for Green, 00 for Blue, or, ...

1111 1111 for Red, 0000 0000 for Green, 0000 0000 for Blue

Note:  $256^3 = 16'777'216$  colour combinations

## 2.6 Time

Computers store time internally as the number of seconds that have lapsed from an arbitrarily agreed epoch (zero-point) of midnight, 1st January 1970 UTC.

32 bit computers take their name by the fact their internal calculations are performed using an integer size of 32 bits. A signed 32bit integer has a range of -2,147,483,648 to 2,147,483,647.

That means that a little after 2 billion seconds have lapsed from the start of the 1970s, a 32 bit computer would be unable to accurately store an integer that represented the time! In fact, it would clock over from being 1970 plus 2 billion seconds to becoming 1970 minus 2 billion seconds! When do we reach this limit? 03:14:07 UTC on 19 January 2038!

The subsequent second, any computer still running a 32 bit signed system will clock over to 13 December 1901, 20:45:52.

While your personal computer may be a 64 bit system, so you think you are safe, there are a lot of systems still around that we all rely on that have 32 bit internals. This is particularly true of embedded systems in transportation infrastructure, electrical grid control, pumps for water and sewer systems, internal chips on cars and other machinery, even a lot of Android mobile phones (though admittedly the changes of one of them still being in use in 20 years is unlikely!). If you research into the "2038 problem" you'll discover just how many critical systems are still vulnerable.

## Practice

Looking for more practice? This website has a number of online quizzes for you to convert between number systems and practice your binary arithmetic.

[http://www.free-test-online.com/binary/binary\\_numbers.htm](http://www.free-test-online.com/binary/binary_numbers.htm)



### 3. Logic gates & circuits

So we've seen that binary can be used to store numbers, text, and colour codes, what we can use binary for much more than storing values; binary also forms the basis of all the logic functionality that occurs within computers.


We do this through what are commonly known as logic gates. All gates can be simplified down into the three of AND, OR and NOT but there are six gates we will learn to love in this course:

AND  
NAND

OR  
NOR

NOT  
XOR

The following video provides a great introduction into how you can easily create your own logic gates and how they work.



Heliosphere (2016) Relays and Logic Gates -  
How to Make a Computer: Part I  
<https://www.youtube.com/watch?v=fB85NrUBBhQ>

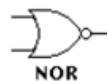
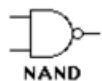
The six logic gates, their symbols, and their truth tables are as follows:



A	B	A & B
0	0	0
0	1	0
1	0	0
1	1	1

A	B	A   B
0	0	0
0	1	1
1	0	1
1	1	1

A	$\bar{A}$
0	1
1	0



A	B	A NAND B	A NOR B	A XOR B
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	0	0	0

To help you try to remember what the various symbols look like, it might be helpful to remember the ANDroid way (cheesy I know)...



Like PEMDAS in mathematics, an order of precedence exists for equations involving gates. The order of precedence is:

1. NOT
2. AND (NAND)
3. OR (NOR, XOR)

Logic equations can either use the written name of the relevant logic gates, or they could be expressed using boolean notation as per the following table. Unfortunately there are several different notations that you may come across.

NOT	$A'$	$\neg A$	$\bar{A}$	$\neg A$
AND	$AB$	$A*B$	$A\bullet B$	$A \wedge B$ $A \cap B$
OR	$A+B$	$A \vee B$	$A \cup B$	
NAND	$(AB)'$	$\overline{AB}$		
NOR	$(A+B)'$	$\overline{A+B}$		
XOR	$A \oplus B$	$A @ B$		
XNOR	$(A \oplus B)'$	$\overline{A \oplus B}$	$(A @ B)'$	$\overline{A @ B}$

As a result, the following are all ways you could represent an AND gate:

Logic equations	Truth table	Logic diagram															
$X = A \text{ AND } B$ $X = AB$ $X = A \& B$ $X = A \cap B$ $X = A \bullet B$	<table border="1"> <thead> <tr> <th>A</th> <th>B</th> <th>X</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>0</td> <td>0</td> </tr> <tr> <td>0</td> <td>1</td> <td>0</td> </tr> <tr> <td>1</td> <td>0</td> <td>0</td> </tr> <tr> <td>1</td> <td>1</td> <td>1</td> </tr> </tbody> </table>	A	B	X	0	0	0	0	1	0	1	0	0	1	1	1	
A	B	X															
0	0	0															
0	1	0															
1	0	0															
1	1	1															

You may be required to convert from any one of these methods, to any other method.

ie: Logic diagram <--> logic equation <--> truth table

## Practice

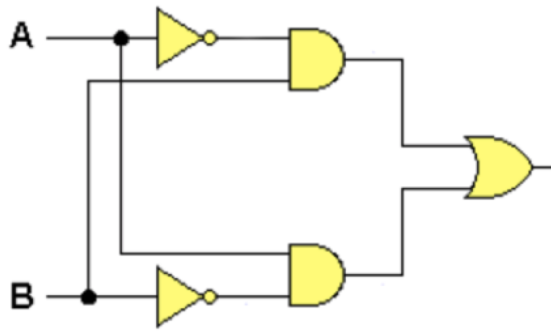
Question 1. Convert this truth table to logic diagram and logic equation

A	B	C	X
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

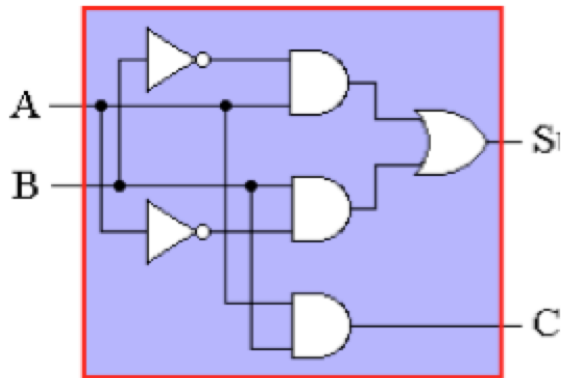
Question 2. Convert this truth table to logic diagram and logic equation

A	B	C	X
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Question 3. Determine the truth table and logic equation.



Question 4. Determine the truth table and logic equation.



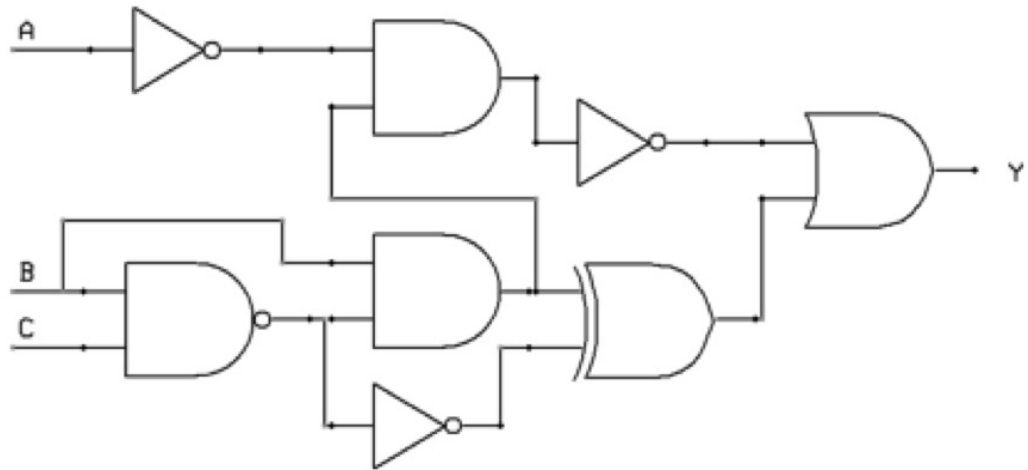
Question 5. Find the logic equation and logic diagram

A B C X

0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Question 6. Find the values for Y in the truth table

A	B	C	Y
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	



Question 7. Find the logic equation and diagram

A	B	C	X
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Question 8. Find the logic diagram and truth table.

$$X = \text{not } A \text{ and } B \text{ or } A \text{ and not } B$$

Question 9. Find the logic diagram and truth table.

$$X = (A | B) \& (\text{not } C | B)$$

## 4. The CPU

"A computer processor does moronically simple things – it moves a byte from memory to register, adds a byte to another byte, moves the result back to memory. The only reason anything substantial gets completed is that these operations occur very quickly. To quote Robert Noyce, 'After you become reconciled to the nanosecond, computer operations are conceptually fairly simple.'"

(code: The Hidden Language of Computer Hardware and Software by Charles Petzold)

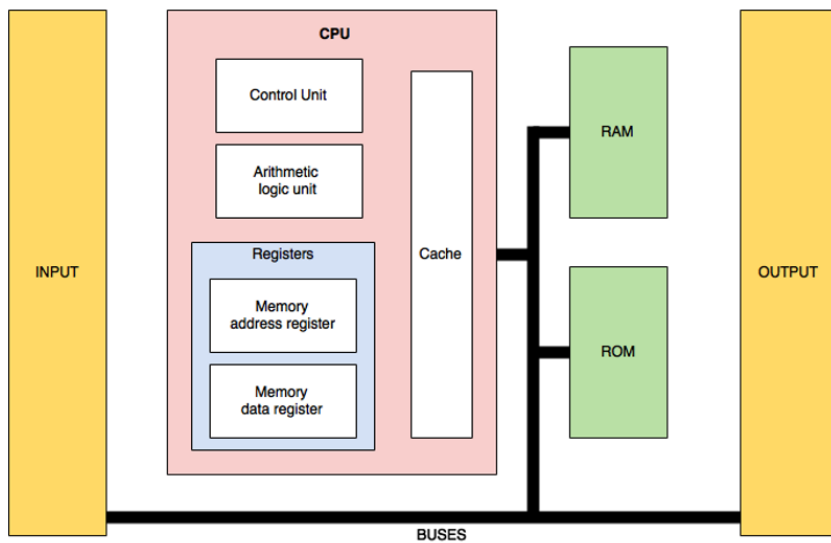
The functions it can perform are not complicated, it's power comes from it's speed. They are just created through millions/billions of logic gates working together.

What is the speed of a typical CPU today? What does that speed "mean"?

A CPU can add, subtract, multiply, divide, load from memory, save to memory. From those simple building blocks we get the computers we have today.

## 4.1 Structure of the CPU

The internal structure of a CPU can be presented by the pink area in the diagram below:



Inside the CPU:

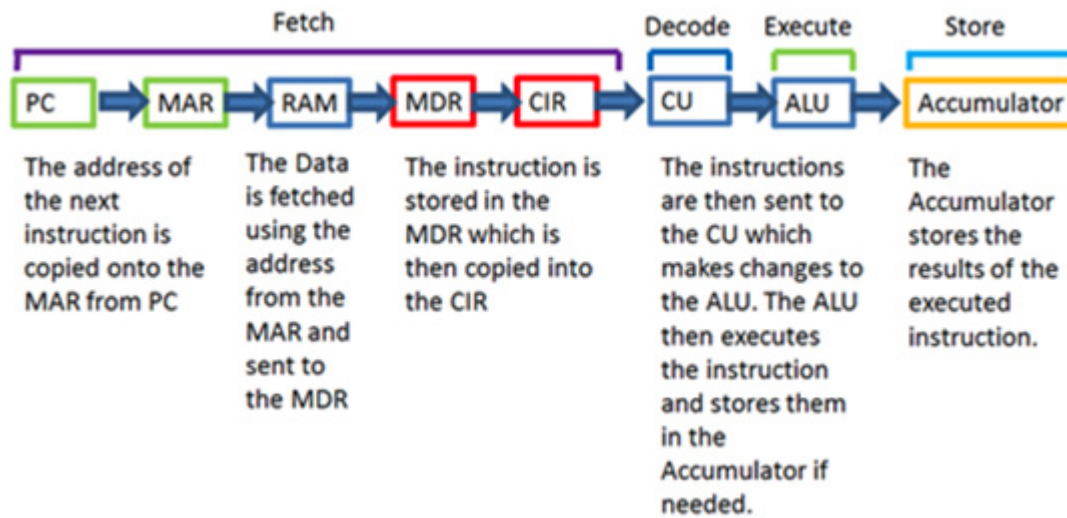
- CU - Control Unit - controls the operations of the processor.
- ALU - Arithmetic logic unit - Performs arithmetic operations
- Register – A small memory location to temporarily hold information as part of the operations process.
- MAR - Memory Address Register - the address in RAM that the data is to be stored in, or fetched from.
- MDR - Memory Data Register - data to be stored, or has just been fetched, from memory.
- Cache – Fast (small) memory that reduces time to access items from memory. Keeps the most frequently used items.

Outside the CPU:

- Input – Keyboard, mouse, scanner, mic etc
- Output – Monitor, printer, speakers etc
- Primary Memory
  - RAM – “Working memory” the documents you are currently working on
  - ROM – “Startup program” to boot the computer. cf. BIOS/CMOS
- Secondary Memory
  - HDD, CD/DVD, USB – Non-volatile bulk storage. Needed to store data between execution / power on
- Bus – transfers data from one part of the computer to another

## 4.2 Fetch-decode-execute

Within the CPU, a cycle known as fetch-decode-execute controls the operations:



- Fetch - Each instruction is stored in memory and has its own address. The processor takes this address number from the program counter, which is responsible for tracking which instructions the CPU should execute next.
- Decode - All programs to be executed are translated into Assembly instructions. Assembly code must be decoded into binary instructions, which are understandable to your CPU. This step is called decoding.
- Execute - While executing instructions the CPU can do one of three things: Do calculations with its ALU, move data from one memory location to another, or jump to a different address.
- Store - The CPU must give feedback after executing an instruction and the output data is written to the memory.

From <https://turbofuture.com/computers/What-are-the-basic-functions-of-a-CPU>



The Fetch-Execute Cycle: What's Your Computer Actually Doing? by Tom Scott  
<https://www.youtube.com/watch?v=Z5JC9Ve1sfl> (9:03)

The Evolution Of CPU Processing Power Part 1: The Mechanics Of A CPU  
<https://www.youtube.com/watch?v=sK-49uz3IGg> (14:04)



## 5. Memory

We have looked at a lot of different types of memory. Let's briefly compare each type to make some generalisations about their different properties.

Memory type	Speed	Capacity	Cost	Storage duration
CPU register	Very fast	Very small (a few bytes)	Very expensive (built into the CPU)	For immediate use
Cache	Very fast	Small (a few MB)	Very expensive	Immediate use
RAM	Fast	Large-ish (8 GB)	About USD 1c/MB	Short term (seconds to minutes)
SSD	Moderate	Large 100s of GB	About USD 0.2c/MB	Long term, non-volatile
HDD	Slow	Large TBs	Cheap! About USD 0.005c/MB	Long term, non-volatile
Tape	Excruiciatingly slow	Many TBs or Petabytes	Extremely cheap!	Several years

Question: What is the difference between volatile and non-volatile memory?

## 6. Operating systems & applications

We continue our abstraction journey upward. We started with the humble electron, called it a bit, scaled up into bytes, started grouping bits together to build logic circuits, grouped bits together into bytes, and then started performing operations on them with ALUs inside CPUs and storing the results in memory. Finally we move into looking at the software that manages all this hardware for us, the operating system.

Operating System (OS) can be defined as a set of programs that manage computer hardware resources and provide common services for application software. The operating system acts as an interface between the hardware and the programs requesting I/O.

That said, we will be looking at operating systems in more depth in unit 6.

For now, there is just one question: What are the main functions of an operating system?

- To manage hardware resources, and
- To provide services to the applications

What are the hardware resources that require managing? What are some of the services an OS provides to applications? Rather than repeating myself, see my notes in unit 6 for this as it is all addressed there.

Finally, after we have an operating system to manage the hardware, we get to run our application software to "do stuff"!

What are some of the common application uses available with computing? Some common categories include:

- Word processors
- Spreadsheets
- Database management systems
- Email
- Web browsers
- Computer aided design (CAD)
- Graphic processing
- Video & audio editing

Names of a few of the main ones for each category? A key differentiator between each?

What are some of the more common features across most applications?

- Toolbars, menus, dialogue box, graphical user interface (GUI)

Which features are provided by the OS, and which by the application?

# Past paper questions for review

(refer to separate document)