THE INSTITUTE OF EDUCATION

Computer Science

Diarmuid Ó Muirgheasa

Leaving Cert

Higher Level

2020/2021

Volume 2 – 1950s-1970s

CONTENTS

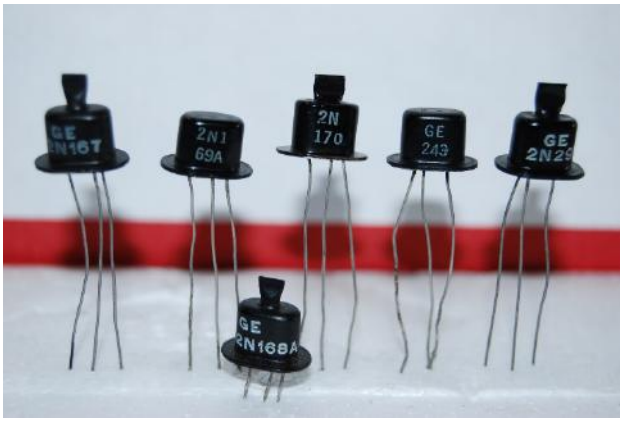
Contents

1950s - State of Play

Let's stop and take stock of where we are in the story.

By 1953 many of the key components are in place for the development of modern computing. The era of theoretical machines designed by Babbage and Turing has passed and programmable machines like Colossus and ENIAC have been constructed and successfully used to defeat the Axis powers in World War 2.

At Bell Labs in the USA solid state transistors have been developed, allowing for vast improvements in computing power and reliability compared with the vacuum tubes used up to this point.



While all this was going on, however, computers were still very much not a part of daily life for the vast majority of people in the world. Consider the example cited in the "Computer Science for Leaving Certificate" textbook, of a person in Ireland who wants to book a flight to New York, and comparing that experience from 1950 to 2020.

Quite aside from the cost difference, the logistical challenges of booking a flight in 1950 were significant. A phone call was required to find flight times and costs, and it is likely that one would have to physically attend a travel agent or airport to pay for a flight. Tickets would have been posted in advance as well - there was no such thing as boarding passes on your phone in the 1950s!

The journey itself would have been much more arduous. No mobile access to train/bus timetables, and traffic updates would have been dramatically less available as well. Nowadays it is trivial to compare, in real time, travel times across various modes of transport. Even navigating airport terminals without the benefit of computerised arrivals/departures boards would have been much more difficult. Communication from the airplane would have been impossible for passengers (WiFi is now commonplace on airplanes), while communication from New York to Dublin would have involved expensive long distance telephone calls.

Exchanging money in the 1950s could of course only be done in person, either at the bank in advance of travel, or at whatever rate was offered in whatever airport you happen to find yourself. Travellers often carried "traveller's cheques" as a means to safely transport large amounts of currency - these were dramatically more expensive and less flexible than modern currency cards.

Booking a hotel was also difficult in the 1950s, both because of communication challenges in making a booking and because details about the hotel were unlikely to be readily available. Again, modern technology has rendered these worries largely obsolete. There are hundreds of photos available of almost any conceivable accommodation option in any corner of the world, alongside reviews by other travellers, maps showing distances to local amenities, and so on and so on. If you wanted to book a hotel, you could probably do so on your phone in a matter of minutes.

1953 - Invention of High Level Programming Languages

One of the key advancements required to bring us from 1950s technology into the 21st century was the invention of high level languages. These allow programmers to be much more efficient and productive, and also open programming/coding up to a wider range of people than just specialist computer scientists. Engineers, mathematicians, business people and many others can get value from computers, without necessarily having to spend years learning about how they work.

Loosely speaking, a "low level" language is one which is tightly coupled to the underlying hardware. In a low level language the programmer needs to consider specific memory locations, and must consider certain elements of the underlying hardware. Low level languages (e.g. assembly language) are generally difficult for humans to understand, but very close to what a computer can understand.

A "high level" language is much closer to natural human language. Most languages used nowadays are considered high level languages - eg Python, Java, Javascript, C++, etc. These languages allow us to code without worrying about the underlying computer architecture. Not only does this make coding faster and easier, it also makes our code much more portable. It is easy to run Python or Javascript on many different platforms, whereas assembly language must be written for specific processor architectures.

High level languages also allow us to quickly and relatively easily implement conditionals ("if" statements) and loops, which are key features of all high level languages.

High level languages	**Low level languages**
Similar to human language - relatively easy for humans to read and write.	Not very human readable. Difficult to write and debug.
Abstracts away memory management.	The programmer must manually manage memory allocation.
Generally quite mobile. Code can be written once and with minor modifications can generally run on many different computer types.	Not mobile. Low level languages are tied directly to hardware, so different processor architectures may have totally different languages with limited similarities.
Efficiency varies between languages, but broadly less efficient than low level languages.	Efficiency depends on the programmer, but low level languages are generally considered much more efficient than high level languages.
Category includes most modern languages e.g. Java, Swift, Go, C++, Python, Javascript and many, many more.	Assembly language, machine code.

There is a key term used to describe the role high level languages play, and that is “abstraction”. Abstraction allows us to forget about complicated implementation details and focus on what matters to us *right now*. We’ll come back to abstraction soon.

The first high level language was implemented by Grace Hopper in 1953, working on the UNIVAC computer for the US Navy. It was called A-0.

A-0 never achieved widespread use, but it ultimately evolved into the very widely used COBOL (COmmon Business-Oriented Language). Meanwhile IBM was working on FORTRAN (FORmula TRANslation). Led by John Backus, this language was developed in 1954 and became an industry standard for many decades. FORTRAN is even still in use today in certain applications, almost 70 years later!

> *“The innovation to use a keyboard to enter data and construct programs, revolutionised the world of computing, as far back as 1956. At MIT, researchers began experimenting with direct keyboard input to computers, a precursor to today's normal mode of operation.*
>
> *Typically, computer users of the time fed their programs into a computer using punched cards or paper tape. Doug Ross wrote a memo advocating direct access in February. Ross contended that a Flexowriter – an electrically-controlled typewriter – connected to an MIT computer could function as a keyboard input device due to its low cost and flexibility. An experiment conducted five months later on the MIT Whirlwind computer confirmed how useful and convenient a keyboard input device could be.”*
>
> - *From “Computer Science - Evolutions of Computers in Society”*

Hello World - comparison of different languages (for illustrative purpose only - not examinable! Source: helloworldcollection.de)

```
/* Hello world in ARM assembly (Android devices) */
.data
msg:
    .ascii      "Hello, World!\n"
len = . - msg
.text
.globl _start
_start:
    mov     %r0, $1
    ldr     %r1, =msg
    ldr     %r2, =len
    mov     %r7, $4
    swi     $0
    mov     %r0, $0
    mov     %r7, $1
    swi     $0
```

```
; Hello World for Intel Assembler (MSDOS)

mov ax,cs
mov ds,ax
mov ah,9
mov dx, offset Hello
int 21h
xor ax,ax
int 21h

Hello:
  db "Hello World!",13,10,"$"
```

```
      * Hello World in COBOL

*****************************
IDENTIFICATION DIVISION.
PROGRAM-ID. HELLO.
ENVIRONMENT DIVISION.
DATA DIVISION.
PROCEDURE DIVISION.
MAIN SECTION.
DISPLAY "Hello World!"
STOP RUN.
*****************************
```

```
# Hello world in Python 3 (aka Python 3000)

print("Hello World")
```

Interpreted v Compiled Languages

Languages like Python and Javascript are *interpreted* languages. This means that the code you write is translated into machine code as the program runs - there is no separate step to translate the code. Generally speaking this means that these languages are a little slower, but there are flexibility advantages too. Python code will normally run on any computer which runs Python with no changes required. This is not always true of compiled languages.

In addition, interpreted languages will generally run until they hit an error, so if you have an error on line 20 you can expect lines 1 through 19 to run correctly.

A *compiled* language, like C or Java, has an extra step between writing code and running code, called *compiling*. This is the part where your code is converted to machine code. This adds a little extra effort up-front, but the resulting code generally runs much faster than an interpreted language can. The act of compiling will also highlight any syntax errors in your code. Note that this does NOT mean that compiled languages don't crash or experience errors. You may still experience "run time"

errors, or simply incorrect output from mistakes in your code. It is only syntax errors that will be caught at compile time.

- Instructions & Programs: Crash Course Computer Science #8 (10:35) taoc.ie/crash-course-instructions-programs - basic overview of the internals of a theoretical, simple CPU, and how a very basic assembly language might work with it.
- The First Programming Languages: Crash Course Computer Science #11 (11:51) taoc.ie/crash-course-programming-languages - why we need high level languages
- High-level Languages (Bits and Bytes, Episode 6) (2:44) taoc.ie/high-level-languages
- Interpreters and Compilers (Bits and Bytes, Episode 6) (3:35) taoc.ie/interpreters-compilers

Abstraction

Abstraction can be a little tricky to get one's head around. In short, it's the idea that allows us to ignore certain details while concentrating on others. When we write "myVariable = 3" in Python we don't consider specifically where that data is being stored. There is some bit of Python magic which worries about that detail for us. This is known as abstraction.

This concept also applies outside of computer science. When we book a train ticket we are abstracting away all the underlying details of operating a rail network. They aren't relevant to us, the passenger! We just book the 7:05 train, and when we arrive at the station we expect to find a train there for us at 7:05. We don't need to give any thought to ensuring the train driver is out of bed in time - that detail is abstracted away.

The founder of Trello and Stack Overflow, Joel Spolsky, wrote the following blog post in 2002, but it remains equally relevant today (key piece reproduced here - follow the link at the bottom to get the full blog post). Note that we will mention TCP/IP again later when discussing the internet - it's a key building block of global communications nowadays:

> *"There's a key piece of magic in the engineering of the Internet which you rely on every single day. It happens in the TCP protocol, one of the fundamental building blocks of the Internet.*
>
> *TCP is a way to transmit data that is reliable. By this I mean: if you send a message over a network using TCP, it will arrive, and it won't be garbled or corrupted.*
>
> *We use TCP for many things like fetching web pages and sending email. The reliability of TCP is why every email arrives in letter-perfect condition. Even if it's just some dumb spam.*
>
> *By comparison, there is another method of transmitting data called IP which is unreliable. Nobody promises that your data will arrive, and it might get messed up before it arrives. If you send a bunch of messages with IP, don't be surprised if only half of them arrive, and some of those are in a different order than the order in which they were sent, and some of them have been replaced by alternate messages, perhaps containing pictures of adorable baby orangutans, or more likely just a lot of unreadable garbage that looks like that spam you get in a foreign language.*

Here's the magic part: TCP is built on top of IP. In other words, TCP is obliged to somehow send data reliably using only an unreliable tool.

To illustrate why this is magic, consider the following morally equivalent, though somewhat ludicrous, scenario from the real world.

Imagine that we had a way of sending actors from Broadway to Hollywood that involved putting them in cars and driving them across the country. Some of these cars crashed, killing the poor actors. Sometimes the actors got drunk on the way and shaved their heads or got nasal tattoos, thus becoming too ugly to work in Hollywood, and frequently the actors arrived in a different order than they had set out, because they all took different routes. Now imagine a new service called Hollywood Express, which delivered actors to Hollywood, guaranteeing that they would (a) arrive (b) in order (c) in perfect condition.

The magic part is that Hollywood Express doesn't have any method of delivering the actors, other than the unreliable method of putting them in cars and driving them across the country. Hollywood Express works by checking that each actor arrives in perfect condition, and, if he doesn't, calling up the home office and requesting that the actor's identical twin be sent instead. If the actors arrive in the wrong order Hollywood Express rearranges them. If a large UFO on its way to Area 51 crashes on the highway in Nevada, rendering it impassable, all the actors that went that way are rerouted via Arizona and Hollywood Express doesn't even tell the movie directors in California what happened. To them, it just looks like the actors are arriving a little bit more slowly than usual, and they never even hear about the UFO crash.

That is, approximately, the magic of TCP. It is what computer scientists like to call an abstraction: a simplification of something much more complicated that is going on under the covers. As it turns out, a lot of computer programming consists of building abstractions. What is a string library? It's a way to pretend that computers can manipulate strings just as easily as they can manipulate numbers. What is a file system? It's a way to pretend that a hard drive isn't really a bunch of spinning magnetic platters that can store bits at certain locations, but rather a hierarchical system of folders-within-folders containing individual files that in turn consist of one or more strings of bytes.

Back to TCP. Earlier for the sake of simplicity I told a little fib, and some of you have steam coming out of your ears by now because this fib is driving you crazy. I said that TCP guarantees that your message will arrive. It doesn't, actually. If your pet snake has chewed through the network cable leading to your computer, and no IP packets can get through, then TCP can't do anything about it and your message doesn't arrive. If you were curt with the system administrators in your company and they punished you by plugging you into an overloaded hub, only some of your IP packets will get through, and TCP will work, but everything will be really slow.

This is what I call a leaky abstraction. TCP attempts to provide a complete abstraction of an underlying unreliable network, but sometimes, the network leaks through the abstraction and you feel the things that the abstraction can't quite protect you from. This is but one example of what I've dubbed the Law of Leaky Abstractions:

All non-trivial abstractions, to some degree, are leaky.

Abstractions fail. Sometimes a little, sometimes a lot. There's leakage. Things go wrong. It happens all over the place when you have abstractions.

[...]"

- *© Joel Spolsky, joelonsoftware.com/2002/11/11/the-law-of-leaky-abstractions/*

The Future of Jobs

The future of jobs: http://taoc.ie/future-of-jobs, 5:34.

The first 8 minutes of the BBC video Will Robots Take Our Jobs (http://taoc.ie/will-robots-take-our-jobs, 24:28), examines the impact of automated ports, focusing on Rotterdam. The remaining 20 minutes discusses the wider impact of CS on our world of roles and careers.

The following piece is from Michael Dertouzos, speculating on the impact of CS on our world in the future. It was 1995, and he had just come from a meeting in MIT, chaired by Tim Berners-Lee, the inventor of the world wide web.

> *"In a quiet but relentless way, information technology would soon change the world so profoundly that the movement would claim its place in history as a socioeconomic revolution equal in scale and impact to the two industrial revolutions. Information technology would alter how we work and play, but more important, it would revise deeper aspects of our lives and of humanity: how we receive health care, how our children learn, how the elderly remain connected to society, how governments conduct their affairs, how ethnic groups preserve their heritage, whose voices are heard, even how nations are formed. It would also present serious challenges: poor people might get poorer and sicker; criminals and insurance companies and employers might invade our bank accounts, medical files, and personal correspondence. Ultimately, the Information Revolution would even bring closer together the polarized views of technologists who worship scientific reason and humanists who worship faith in humanity. Most people had no idea that there was a tidal wave rushing toward them."*

Discussion Topics

These are six areas in which, it is claimed, Computer Science improves the world we live in today. Do you agree or disagree with each area?

1. Solving problems and improving solutions
2. Protecting people and organisations
3. Furthering education
4. Improving communication
5. Organizing & streamlining philanthropy
6. Positively impacting every area of society

Further Resources

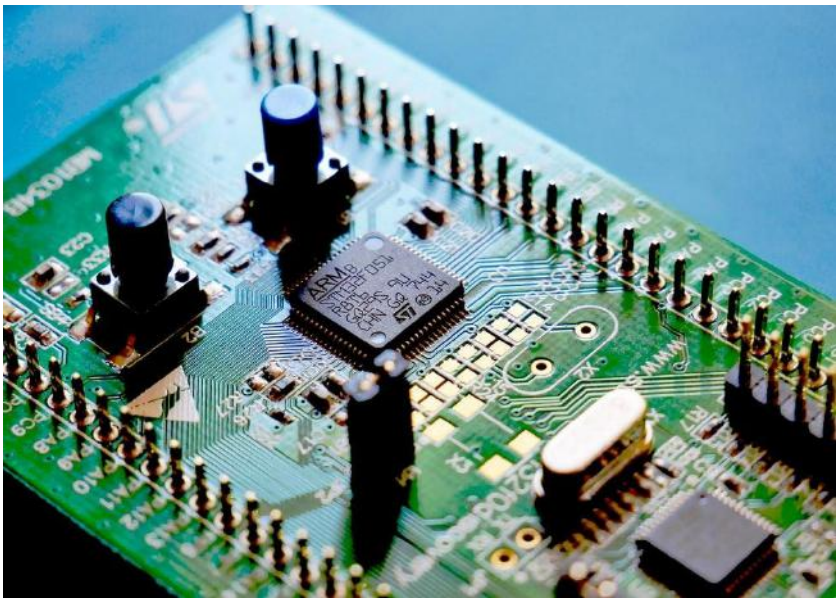
Twitter, if carefully curated, can provide real insight in many areas. The following accounts are worth following for frequent commentary on technology, society and the future of jobs:

- Patrick McKenzie (@patio11)
- Scott Galloway (@profgalloway)
- Kara Swisher (@karaswisher)
- Patrick Collison (@patrickc)

1958 - Integrated Circuits

In 1958 there was a further seismic shift in computer technology, as **Integrated Circuits** (IC - also referred to as **microchips**) were invented simultaneously and independently by Jack Kilby, of Texas Instruments, and Robert Noyce, of Fairchild Semiconductors. (The history of the development of IC is rather muddy, and several others lay claim to various developments which have contributed to the modern understanding of an integrated circuit. Jack Kilby was awarded the Nobel Prize in 2000 for his contribution, and specifically acknowledged the then-deceased Robert Noyce as a co-inventor).

Crash Course Computer Science: Integrated Circuits and Moore's Law (13:49) taoc.ie/ic-moores-law

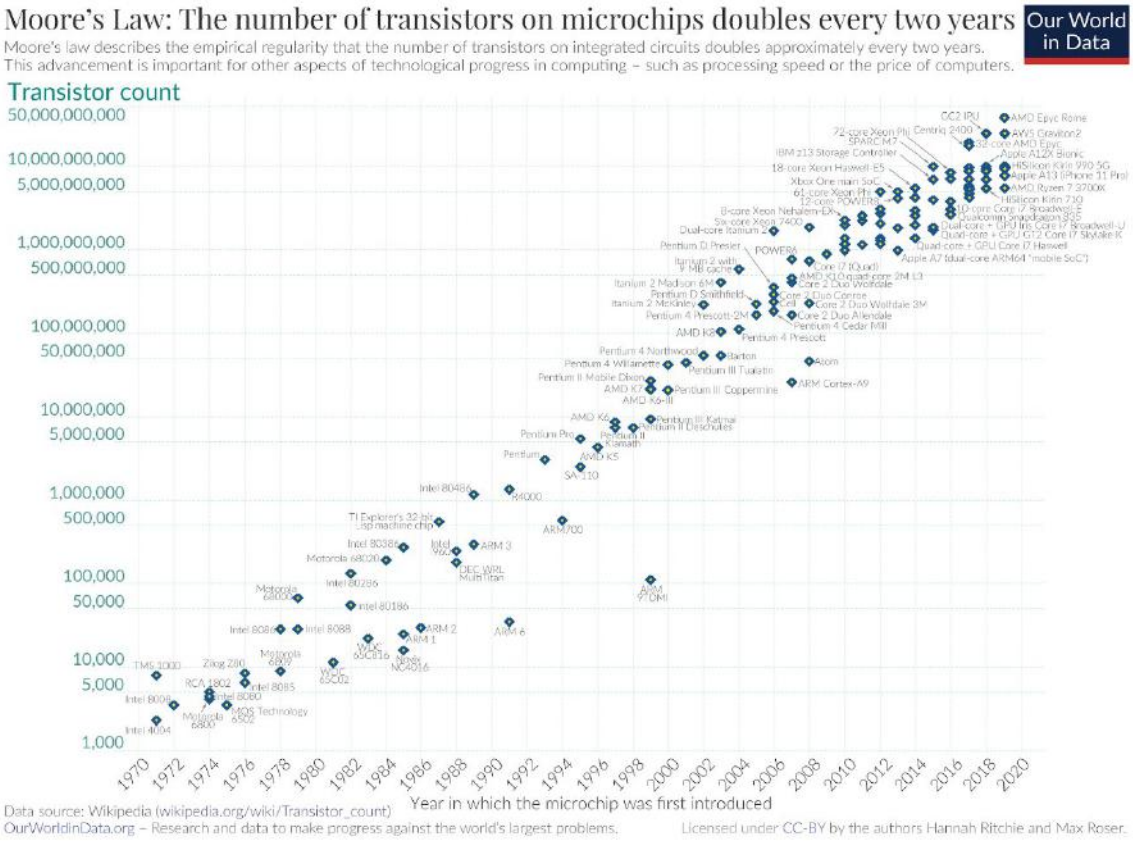


Integrated Circuits are so-called due to the way they integrate multiple electronic components into a single package. This carries significant advantages in terms of unit-cost and power consumption, although the cost advantages are somewhat offset by a very high design and tooling cost (cost of getting ready to manufacture the IC).

Integrated Circuits were a key advancement facilitating the development of computers as we know them now, and it was following their invention that machines which we would recognise as being close to modern computers began to appear.

This period also saw the beginning of the "Moore's Law Era". Moore's Law is named after Gordon Moore, co-founder (with Robert Noyce and others) of Fairchild Semiconductors, and later co-founder of Intel. The first version of the "law" (really more of a prediction, observation or projection) was based on an observation in 1965 that component density on microchips would double every year. In 1975 Moore revised this prediction, stating that component density would instead double every 2 years.

This observation has held remarkably consistent for almost half a century, and this increase in chip density is what has facilitated the extraordinary growth in computing power over the past several decades.



While the end of Moore's law has been incorrectly predicted for many years, there are some extremely solid physical barriers for engineers to overcome in the near future, most importantly the physical size of silicon atoms. Regardless of other developments, decreasing the physical size of a transistor below 1nm is probably impossible, due to the size of a silicon atom, while even reducing the manufacturing process below about 5nm may not be possible at scale.

Processor	Transistor Count	Year	Process	Comparable Objects	Electromagnetic Spectrum
Intel 4004	2300	1971	10 µm	Red Blood Cells	Infra Red
Motorola 68000	68000	1979	4 µm	Most bacteria	Infra Red
Intel 80486	1.18 million	1989	1 µm	E-coli	Visble Light 0.4–0.7 µm
Pentium III	9.5 million	1999	250 nm	Pollen, Viruses	Approaching UV light
Six Core i7/8	2 billion+	2011	32nm	Molecules	UV light
Exynos 8895	20 billion+	2017	10nm	50 atoms of Silicon	Soft X-rays are 1nm

Does this mean that computers will stop getting faster? Thankfully, no. Increasing the density of transistors is one way of increasing the speed of computers, but far from the only one. Apple's M1 chip[1], launched in late 2020, showed one way forward, with radically different microchip architecture enabling dramatically better performance with much lower energy consumption compared to Intel and AMD rivals. (The M1 does also benefit from a cutting edge 5nm manufacturing process, and 16 billion transistors).

Graphics chips show another route forward, with lots of smaller cores rather than a handful of large cores like a Central Processing Unit (CPU). These chips offer another route for tackling high performance computing, although they are also run subject to the same physical barriers around silicon atom size as CPUs

Finally there is the world of quantum computing, which promises to transform modern computing, at least in part by rendering current state-of-the-art encryption technologies almost useless. Quantum computing operates at the quantum level, using physical properties which seem completely counterintuitive the first time you encounter them. If you aren't baffled, bordering on angry, the first time you read about quantum computing, then you didn't properly understand quantum computing. (Luckily you don't need to understand quantum computing for the leaving cert exam, but it's worth being aware of as one possible path forward for certain types of computing).

[1] Architecture in this context means a very similar thing to architecture in the building context - where things are put relative to each other, how many of each thing we have, and how all those things will work together as a whole. From daringfireball.net/2020/11/the_m1_macs (not examinable, but interesting for context):

"First, an intriguing benchmark from David Smith, an engineer at Apple:

> Fun fact: retaining and releasing an NSObject takes ~30 nanoseconds on current gen Intel, and ~6.5 nanoseconds on an M1 … and ~14 nanoseconds on an M1 emulating an Intel.

Retaining and releasing an NSObject is a low-level operation that is foundational to Apple's programming frameworks. Just about everything is an object, and when an object is being used, software retains it, and when it's done being used, the software releases it. This is reference counting in a nutshell: every time an object is retained, its reference count increments. Every time that object is released, its reference count decrements. When it goes to zero, the system frees the object from memory, trusting that it is no longer in use. Retain and release are tiny actions that almost all software, on all Apple platforms, does all the time.

How this works and why it's so much faster on Apple Silicon than Intel is fascinating but beside the point. The point is that Apple's philosophical reliance on this model of software development is not typical in the broader world. This is the way Apple thinks software should work, dating back to the origins of NeXTstep in 1989. The Apple Silicon system architecture is designed to make these operations as fast as possible. It's not so much that Intel's x86 architecture is a bad fit for Apple's software frameworks, as that Apple Silicon is designed to be a bespoke fit for it.

1960s - ASCII - The language of language

Every symbol we type using the keyboard is known as a “character”. Just like everything else in computing, these characters are represented at a hardware level in binary (1s and 0s). It was recognised early on that we would need an agreed standard for what each sequence of 0s and 1s means. (If I think 0100 0001 represents ‘A’ and you think it represents ‘q’, or anything else that isn’t ‘A’, we’re going to run into serious compatibility issues!)

Work began on the standard in 1960, with the first edition published in 1963. This original standard, and the early amendments to it, described a 7-bit system with 127 different possible characters, to include upper- and lowercase letters, numbers, standard punctuation marks and 33 non-printable characters. Many of these non-printable characters have fallen out of use although some, notably including line endings and tabs, are still a vital part of our day-to-day computing.

ASCII (1977/1986)

	_0	_1	_2	_3	_4	_5	_6	_7	_8	_9	_A	_B	_C	_D	_E	_F
0_ 0	NUL 0000	SOH 0001	STX 0002	ETX 0003	EOT 0004	ENQ 0005	ACK 0006	BEL 0007	BS 0008	HT 0009	LF 000A	VT 000B	FF 000C	CR 000D	SO 000E	SI 000F
1_ 16	DLE 0010	DC1 0011	DC2 0012	DC3 0013	DC4 0014	NAK 0015	SYN 0016	ETB 0017	CAN 0018	EM 0019	SUB 001A	ESC 001B	FS 001C	GS 001D	RS 001E	US 001F
2_ 32	SP 0020	! 0021	" 0022	# 0023	$ 0024	% 0025	& 0026	' 0027	(0028	) 0029	* 002A	+ 002B	, 002C	- 002D	. 002E	/ 002F
3_ 48	0 0030	1 0031	2 0032	3 0033	4 0034	5 0035	6 0036	7 0037	8 0038	9 0039	: 003A	; 003B	< 003C	= 003D	> 003E	? 003F
4_ 64	@ 0040	A 0041	B 0042	C 0043	D 0044	E 0045	F 0046	G 0047	H 0048	I 0049	J 004A	K 004B	L 004C	M 004D	N 004E	O 004F
5_ 80	P 0050	Q 0051	R 0052	S 0053	T 0054	U 0055	V 0056	W 0057	X 0058	Y 0059	Z 005A	[005B	\ 005C	] 005D	^ 005E	_ 005F
6_ 96	` 0060	a 0061	b 0062	c 0063	d 0064	e 0065	f 0066	g 0067	h 0068	i 0069	j 006A	k 006B	l 006C	m 006D	n 006E	o 006F
7_ 112	p 0070	q 0071	r 0072	s 0073	t 0074	u 0075	v 0076	w 0077	x 0078	y 0079	z 007A	{ 007B	| 007C	} 007D	~ 007E	DEL 007F

Letter | Number | Punctuation | Symbol | Other | Undefined | Character changed from 1963 version and/or 1965 draft

The above table shows this 127 bit ASCII table in hexadecimal. Note that in the context of character sets we use 0x as a prefix to indicate a hexadecimal number, so 0x0f is equivalent to $0f_{16}$.

In the table above, character 0x41 is ‘A’, while 0x61 is ‘a’ (in the chart they are written as ‘0041’ and ‘0061’, and it is considered implied by context that they are in hex).

Exercise

Fill out hexadecimal, decimal and binary values of the following characters:

Character	Hexadecimal	Decimal	Binary
B	0x42	66	0100 0010
7			
$			
}			
b			
q			
W			
w			

Line endings

Line endings can cause issues in ASCII because different operating systems treat them differently (e.g. Microsoft Windows vs Apple OSX).

There are historically two characters primarily used at the end of a line of text:

- LF (Line Feed, 0x0a): also referred to as \n, or simply a new line character. The name refers to the character's role in older printers, where the paper would be advanced by a single line when this character occurred.
- CR (Carriage Return, 0x0d): also referred to as \r. In older printers this command would cause the "carriage" (the print head) to return to the start of the line.

Macs use the line feed (\n) as their new line character, whereas Windows uses both together (\r\n, also referred to as CRLF or CR+LF). If you open a file in Notepad++, or a similar text editor, you can see which kind of line ending is in place. In Notepad++ you can do this by selecting "View -> Show Symbols -> Show All Characters". The screenshot on the right shows a file created in Windows with the CRLF line endings.

Caesar Cypher

The Caesar Cypher is a very basic form of encryption, reportedly used by Julius Caesar to send secret messages. This form of encryption works by choosing a key, typically a small integer, and "shifting" each character by that number of places.

For example, let's choose a key of "3", and the message "Hello". We would "shift" each letter by three, and the output would be "Khoor". To do this in computer code we would first need to convert each letter to an ascii code (using the function "ord()"), add the number to the ascii code, and then convert back to a character (using the function "chr()").

The steps are shown below:

Original Letter	ASCII Code	ASCII Code + 3	Encoded Letter
H	72	75	K
e	101	104	h
l	108	111	o
l	108	111	o
o	111	114	r

Note that capitalisation is retained, as lowercase letters and uppercase letters are in a separate chunk of the table.

Try the example below - feel free to use the ASCII table above, bearing in mind that it is in hexadecimal, or find a decimal ASCII table online:

Original Letter	ASCII Code	ASCII Code + 3	Encoded Letter
I			
n			
s			
t			
i			

t			
u			
t			
e			

Now let's try again using eight as the key:

Original Letter	ASCII Code	ASCII Code + 8	Encoded Letter
I			
n			
s			
t			
i			
t			
u			
t			
e			

There is a sting in the tail here. If you pass beyond the end of the alphabet you will start outputting symbols, or perhaps nothing at all if you run into control characters! If you reach the end of the alphabet you need to loop back to the beginning.

When we write this in Python we will need some clever way of determining if we have passed beyond the end of the alphabet.

Challenge

Write a pseudocode algorithm describing how you would perform the shift above, looping back around to the start of the alphabet if you pass beyond z. Confirm that your method would work for both uppercase and lowercase letters.

Practice Questions:

Encode the following words with the specified key

1. Coding; key = 2
2. Microchip; key = 5
3. Zebra; key = 3
4. Warranty; key = 4

Other resources

As per usual Joel Spolsky has an excellent blog post from all the way back in 2003 on this subject, which is still pretty much all valid, available at taoc.ie/joel-charactersets. It is NOT all examinable, but it provides a richer context to our core notes on the topic.

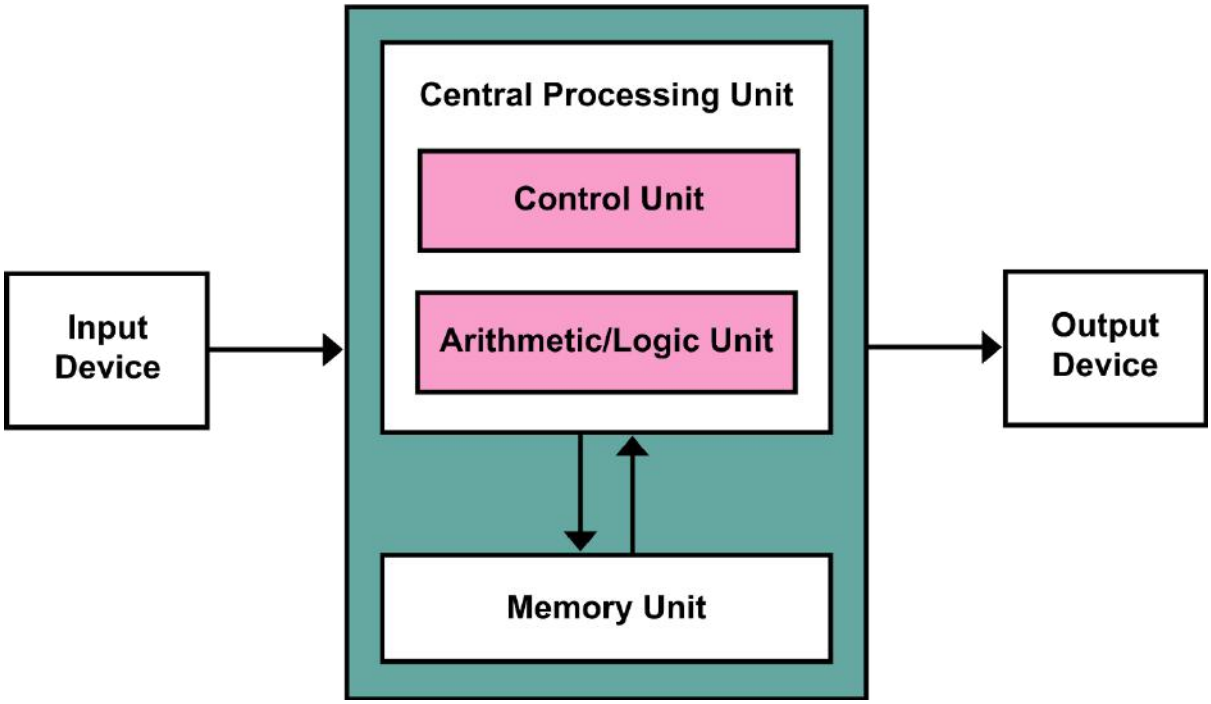
1970s - Modern Computer Architecture

Modern computer design developed quite quickly after the invention of integrated circuits in the late 1950s. By the early 1970s most of the elements discussed below had been developed in their early forms.

This (slightly cheesy and dated) video explains the key parts of a computer, and looks back over many elements we've already covered, connecting them with more modern implementations of the same ideas. http://taoc.ie/intel-computer-history (8:20)

Von Neumann Architecture

John Von Neumann was discussed in detail in a video earlier in the course. A Hungarian-American, who lived 1903-57, he made outstanding contributions to mathematics, physics, engineering and computer science. The basic von Neumann computer architecture, first described by him in 1945, remains at the core of computer architecture today.



The above image summarises the von Neumann architecture, which consists of four key component types:

- A central processing unit (CPU) which performs calculations.
- Memory, which stores incoming and outgoing data, and instructions to be executed.
- A *bus* connecting the CPU and memory - the bus is represented by arrows here, and is the physical connection between elements of the computer.
- Input and output devices, which facilitate data being received and transmitted by the computer.

CPU

A CPU (Central Processing Unit, also referred to as a processor or microprocessor) contains a **control unit**, **Arithmetic Logic Unit** (ALU) and **registers**, which are temporary storage locations.

The **control unit** directs the operations of the CPU, and is responsible for controlling the transfer of data and instructions between the other units of the computer.

The **ALU** performs arithmetic operations (addition, subtraction, multiplication and division, along with more complex operations completed by performing those simple operations multiple times) and logical operations (comparing and merging data, for example).

Registers are extremely fast, extremely expensive memory (we will discuss different memory types shortly).

Fetch-Decode-Execute; the Instruction Cycle

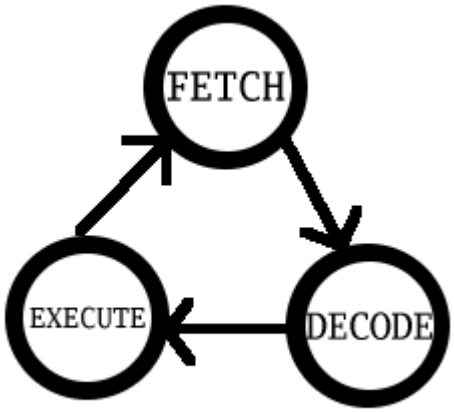
Also referred to as the fetch-execute cycle, this describes how processors process instructions. As the name suggests, this is a cycle: once the execute phase is complete the computer will loop back to the fetch phase.

The **Program Counter** is a special register which holds the memory address of the next instruction to be executed.

At the *fetch* stage of the cycle the processor will retrieve the next instruction to be executed, and update the program counter to point to the following instruction.

At the *decode* stage the instruction previously fetched will be interpreted, and various steps may be taken to prepare for the execute phase.

At the *execute* stage the processor will carry out the instruction, making use of its ALU if required to perform any arithmetic or logical operations, and storing the results in the appropriate location.



The clock is responsible for timing these operations and keeping everything in sync. You can think of it like a metronome, ensuring that all elements within the CPU operate together (in fact it's a stream of electrical pulses). The *clock speed* is the figure quoted in MHz (million operations per second) or GHz (billion operations per second) on your laptop, tablet, phone, etc - a CPU operating at 3GHz will perform the instruction cycle *3 billion times per second*.

In addition to clock speed, the number of **cores** in a computer will have a significant impact on the speed at which it can execute instructions. Each core is almost like its own CPU, with its own instruction cycle happening in parallel with all other cores. Cores may share the same clock, and some of the same memory, and if a program is optimised for multi-threading then having multiple cores can dramatically increase the speed of execution.

Note that some programs may not benefit at all from running on a multi-core CPU. Why not? Consider how long it takes to grow a beard, or produce a baby. If a beard takes a month to grow, can 30 people work together to produce a beard in a day. Can 9 women cooperate to produce a baby in a month?

The Pipeline - thinking of the CPU as a factory

The pipeline does not need to be understood in detail for the exam, but is another important element of CPU performance. The fetch-decode-execute process is a simplification of the reality of what happens within the CPU. In fact, the 3-stage cycle is often (but not always) broken into many more parts - for example the Intel "Prescott" family of processors from the early 2000s had a 31-stage pipeline!

Imagine the pipeline like a factory assembly line, where each stage can be split into multiple steps. (Eg "attach the wheels" can be split into four "attach one wheel" steps). In theory, increasing the number of steps can allow us to speed up the process.

There is a complicating factor for us though: **branching**. A simple if-statement can send us in two or more totally different directions, depending on how the condition is evaluated. In our car example, imagine the type of car we need to manufacture depends on the colour of the car in front of it on the assembly line. Now imagine that the paint isn't applied until the final step, and we have no way of knowing with certainty ahead of that step what colour will be used. We could wait until the car is complete before starting the next one, but in that case most of our factory is idle most of the time.

Instead of doing that we simply guess at the outcome, and modern processors will perform all sorts of tricks to guess what is coming next, to avoid downtime and ensure maximum useful work is performed.

Of course we can't always guess correctly, and if we do guess wrong we have a big problem! Not only is the second car in the queue now the incorrect car, but potentially so is everything behind it! We now need to throw out everything being worked on in our factory and start again.

In the case of the aforementioned Prescott processors, with their massively long pipelines, the clock speed was very high, making them very fast at tasks like video processing where there is little or no branching. On the other hand they were dramatically less good at gaming, where there is lots of branching. In spite of reaching clock speeds of 3.8GHz (fast even by today's standards), the Prescott processors were generally outperformed in gaming and other similar applications by the Athlon 64, which maxed out at 3.2GHz but with a 12 stage pipeline, instead of Prescott's 31 stage pipeline.

Memory Types

You may already be familiar with **main memory** or RAM ("Random Access Memory"), and with **secondary storage** (your hard disk drive (HDD) or Solid State Drive (SSD)). RAM is where we keep things we're using *right now*. It's in the order of 50,000 times faster than hard disk drives, and hundreds of times faster than an SSD.

RAM is, however, much more expensive than either a HDD or SSD, and it can only store data when it is powered on, so it's no good for long term storage. Because RAM is so expensive[1] we tend to have much less of it than we have secondary storage, so your computer will try to use it as intelligently as possible. It will attempt to ensure that the processor can always find what it needs in main memory, thus avoiding the delay of requesting data from secondary storage. If the CPU is waiting for data from secondary storage it will cause a big delay, and you as the user will probably notice a temporary slowdown as the CPU sits idle.

Imagine you want to paint a picture, but all of your paints are stored in a locker 500m away. You could go and get them one at a time, placing each colour back in the locker when you want a different colour, but you'd spend all your time walking back and forth and never make much progress on your painting.

Much more realistic is that you would collect the supplies you need and bring them all to the workplace in which you intend to paint. In computing terms, the locker is our *secondary storage* while the desk in our workplace is our *main memory* or *RAM*.

There are other types of memory within our computers as well. Broadly they all fit into a pyramid model, with small, fast and expensive memory at the top and big, slow and cheap memory at the bottom.

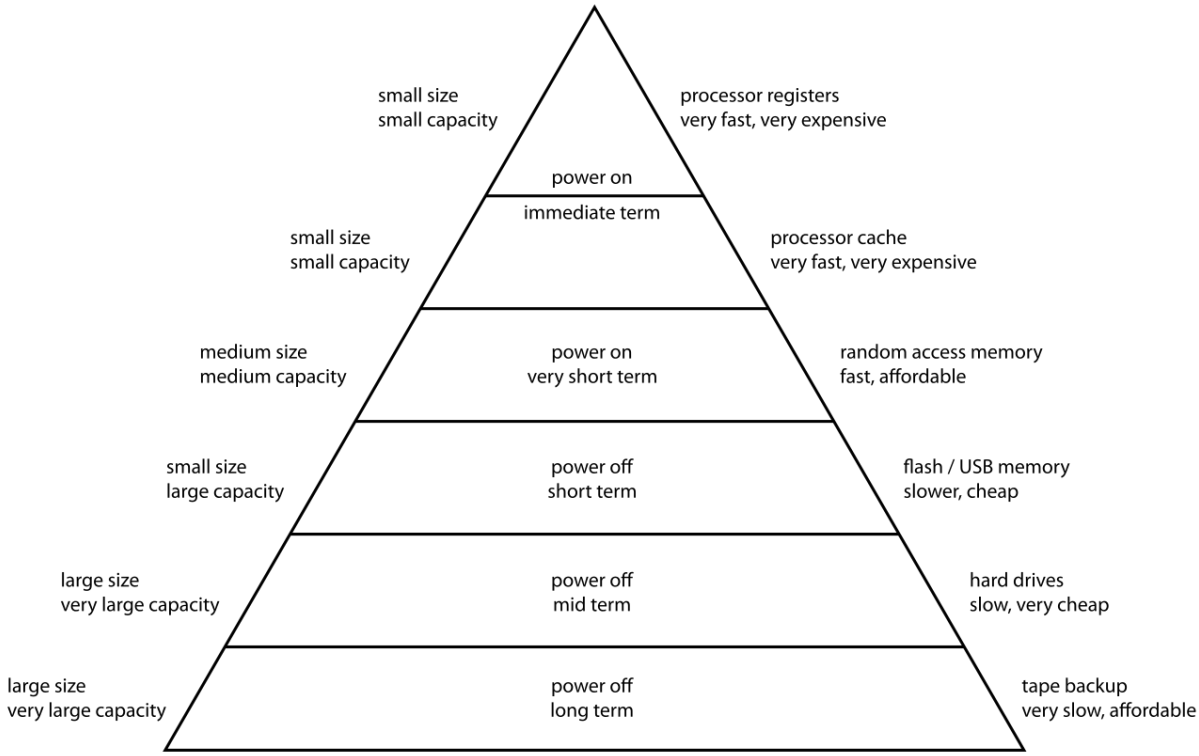
Within our CPU we have **registers**, memory physically located within the processor itself. Registers can be accessed extraordinarily quickly, but are very, very small (think in terms of bytes, rather than kB, MB or GB). There are also various layers of cache within the CPU, again generally increasing in size and decreasing in speed as you move further away from the physical CPU core[2].

Hard drives will often have their own cache as well, again intended to speed up access to data you are most likely to be using next. For example, a 1TB Seagate Barracuda desktop hard drive comes with 64MB of cache, while the 2TB model comes with 256MB of cache.

[1] Very roughly speaking, RAM is €4-6 per GB, compared with perhaps €0.12 per GB for SSD storage, and perhaps as little as €0.03 per GB for HDD storage. Tape storage is another order of magnitude cheaper, perhaps as little as €0.003 per GB or less.

[2] As one example, the original quad core Intel Core i7 launched with 64kB of level 1 cache and 256kB of level 2 cache per core, plus an 8MB pool of shared level 3 cache across all cores. The specifics of this are not examinable, but the general principle is.

Computer Memory Hierarchy



Further down the memory hierarchy we find tape backups, which are extremely slow but perhaps 10 times cheaper again than hard drives for a given capacity.

I/O Devices

Input/output devices, often referred to as I/O devices, allow us to interact with our computers. There are many examples, a few of which are listed below:

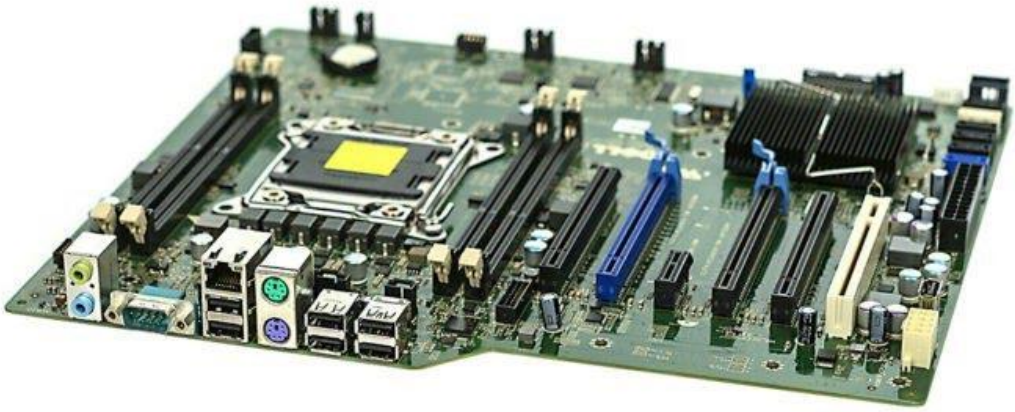
- **Input devices:**
  - Keyboard
  - Mouse
  - Touchscreen
- **Output devices:**
  - Monitor
  - Speakers
  - Printer

In the context of Von Neumann architecture I/O devices could also include secondary storage, which can serve as both an input and an output device. There are many other types of I/O devices.

Motherboard

The motherboard, also known as the “mainboard”, is a circuit board that all other components physically connect to. The CPU and RAM either slot into specific sockets on the motherboard, or in some cases are soldered directly to the motherboard.

Many components, including sound processing chips, network interface cards and more, are often now integrated into the motherboard, where they may previously have been separate expansion cards.



Power Supply Unit (PSU)

The PSU in a desktop computer is responsible for taking power from the mains supply (at 230V in Ireland) and converting it to voltages which can be used by the various components inside the computer (generally 12V, 5V and 3.3V supplies). When you plug a desktop computer into a wall you are physically connecting into the PSU, as can be seen in the pictured PSU (note that there would normally be an additional cover on the PSU - capacitors within a PSU can hold potentially deadly charges for long periods of time after the PSU has been disconnected, so treat any opened PSU with extreme caution!)

Electricity

Remember from previous notes that all data inside a computer is represented using ones and zeroes. We call this binary. The signals can equally be referred to as highs and lows, or on and off signals.

Of course what is *actually* moving around inside the computer is electricity, or more precisely electrons. **Electric current** is the flow of electrons (or other charged particles), and is measured in *amps*. You can imagine this as water moving through a pipe, where the volume of water passing through the pipe is the **current**.

Voltage is also called **potential difference**. In our water pipe analogy this is the water pressure.

Power is defined as **voltage x current**.

Resistance is the final key concept here, and refers to the extent to which the flow of current is restricted by the medium through which it is passing (ie the wire or other electrical component). In the water pipe analogy, imagine squeezing a flexible pipe to restrict the flow of water. You would expect the pressure on the far side of the restriction to be significantly lower than the original pressure, and this is exactly what happens in an electrical circuit as well.

Important electronic components:

- **Resistor:** a resistor is a simple component that impedes the flow of current. Some of the electricity passing through the resistor will be converted to heat.
- **Transistor:** previously discussed at length, a transistor is an electronic switch. In brief, the transistor has three terminals; the base; the collector; and the emitter. A small current applied at the base will unlock the flow of a much larger current from the collector to the emitter.
- **Capacitor:** essentially a high speed battery, capacitors charge and discharge extremely quickly. They are crucial components in almost every non-trivial electronic device, including touchscreens (modern phones and tablets use “capacitive” touchscreens, which are significantly more responsive than older resistive touchscreens, but can only sense conductive touches - hence touch screens not working when you’re wearing gloves!)

Image Credits

Page 3 - 1950 NPN General Electric Transistors - en.wikipedia.org (CC BY-SA 3.0)

Page 3 - Signpost at Chileka Airport, 1950 or mid 1960s - Society of Malawi, Historical and Scientific (CC BY-SA 4.0)

Page 12 - The Future of Jobs - Photo by Maximalfocus on Unsplash

Page 14 - Integrated Circuits - Photo by Vishnu Mohanan on Unsplash

Page 15 - Moore's Law - Ourworldindata.org

Page 17 - ASCII table - https://en.wikipedia.org/wiki/ASCII

Page 21 - Von Neumann architecture - Image by Kapooht from wikimedia.org (CC BY-SA 3.0)

Page 22 - Fetch Execute Cycle - nl.wikipedia.org (CC BY-SA 2.5)

Page 25 - Computer Memory Hierarchy - Public Domain

Page 26 - Motherboard By Marcin Wieclaw (pcsite.co.uk) - https://pcsite.co.uk/product/dell-precision-t3600-motherboard-8hpgt-5507/, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=93722010

Page 26 - Power Supply - Public Domain