

Overview of Computers

Computers: Data processors with internal structure made up of different parts which interact to carry out operations, produce input & produce output.

Analogue Computing

- Parameters **precisely** represented and manipulated by **continuously variable analogues** e.g. voltage, current.
- **Inaccurate:** Don't produce exact results because of issues like noise & general wear.

Digital Computing

- Parameters represented and manipulated using **discrete values**.
- **Imprecise** because it is quantized, a discrete value cannot be an infinite decimal.
- **Accurate:** Discrete nature means the value is always correct.

Importance of Understanding Architectures

- **Learning to Program:** Understanding architecture can help to solve problems.
- **Maximising Efficiency and Performance of Computers**
- **Avoiding Errors & Failures:** Understanding what could go wrong.
- **Recovering from Errors:** Identifying the problem & a solution.

History

Significant Analogue Computers

The Analytical Engine

- First Turing-complete general-purpose programmable computer.

MONIAC

- Modelled an economy using fluid logic.

Digital Computers

1st Generation – Valves & Vacuum Tubes (40s)

Colossus

- First programmable digital computer.
- Used for code breaking.

ENIAC

- First Turing-complete general-purpose programmable digital computer.

Problems with 1st Generation Computers

- Expensive.
- Large.
- Machine language only.

2nd Generation – Transistors (50s)

Transistors: Small devices used to transfer electronic signals across a resistor.

Advantages of 2nd Generation Computers

- Faster.
- More compact.
- More reliable.

3rd Generation – Integrated Circuits (60s)

Integrated Circuits: A set of circuits on a chip of semiconductor material.

Properties

- 10s-100s of transistors per chip.
- More compact.

4th Generation – Large Scale Integration (70s)

Integrated circuits could fit tens of thousands of transistors which enabled **microprocessors** and **microcontrollers**.

5th Generation – Very Large Scale Integration (80s)

Integrated circuits could fit billions of transistors.

The Present

Since the 70s, chips have been developed to fit more transistors allowing processors to be much smaller and prompting the development of:

- Operating systems.
- Networking.
- Mobile & ubiquitous computing.

Moore's Law: Computing power, relative to component cost, doubles approximately every two years.

The physical properties of silicone make this impossible to maintain. Computing power increase has been predicted to slow down many times, but **creative engineering** and **new technologies** have prevented it from doing so.

Von Neumann Architecture (Uniprocessor)

Programmers need to understand how their program is executed but lots of information about execution isn't useful in writing programs.

Von Neumann Architecture abstracts from the details of particular uniprocessor systems, revealing their common structure.

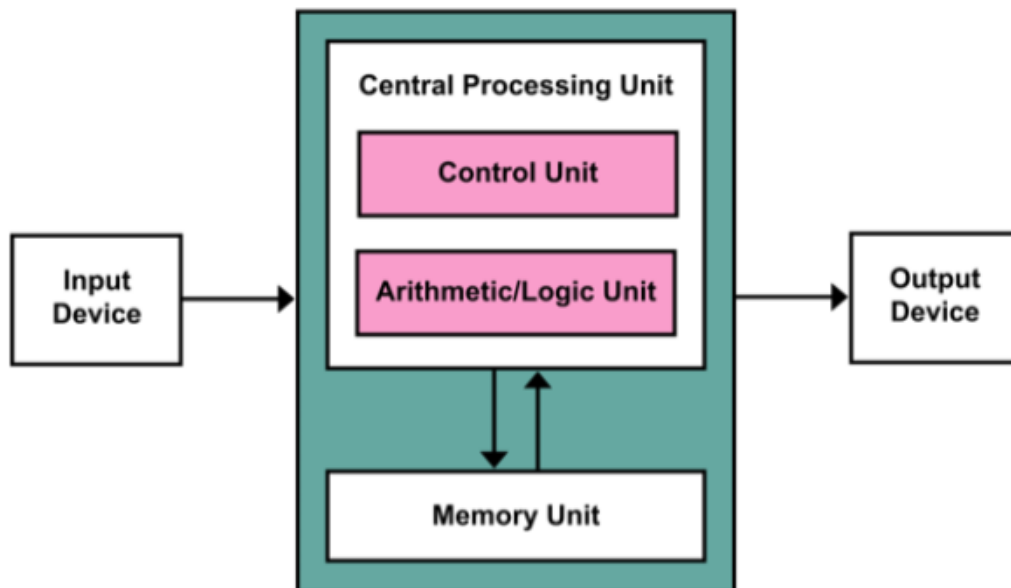
- Enables further abstractions which allow the same program to be run on a variety of machines.
- All uniprocessor architectures use it.
- It has five components, I/O are two of them.

Characteristics

- Data stored discreetly in exact form for an indefinite period.
- Instructions and data stored in the same form and place.
- Can process data using ALU.
- Automation enabled by CU.
- Can communicate via I/O devices.

The Central Processing Unit (CPU)

Acts as the computer's **control center** by executing the instructions of a computer program. **Comprised of a CU & an ALU.**



Components

Arithmetic Logic Unit (ALU)

Carries out computational operations. It needs:

- An operation specification.
- Operands to operate upon.

It contains registers called **accumulators/data registers** for holding data being processed.

Control Unit (CU)

Supplies operations and operands in a sequence to other components which **enables autonomy**.

- Determines the operation to be performed by supplying control data to the ALU.
- Selects the correct operands and supplies them at the right time.

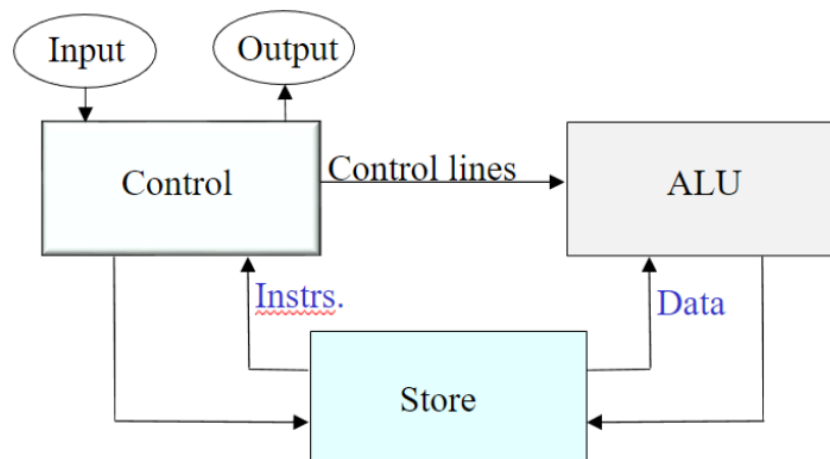
Random Access Memory (RAM)

Often called the **store or DRAM**. Contains data & instructions currently in use.

- Data is stored or retrieved in response to control signals from the CU.
- Fast RAM and a fast CU can keep the ALU in continuous operation.
- The CU retrieves instructions from the store to be used in ALU operations.
- It is volatile meaning that information stored inside is lost when it is turned off.

I/O Devices

These enable communication between the machine and the outside world.



Cache

Extremely **fast volatile** memory built into the processor which stores the most **frequently used data and instructions** so that the CPU can access it faster. Also known as **SRAM**.

This data may be a copy of data stored elsewhere or produced from an earlier calculation.

There are three levels of cache:

Level 1	Level 2	Level 3
<ul style="list-style-type: none">• Closest to the processor.• Very small capacity.• Very fast transfer rate.• Holds the most frequently used information.	<ul style="list-style-type: none">• Off-chip.• Larger capacity than L1.• Slower transfer rate.• Holds information used less frequently.	<i>Like level 2 in that it's slower and larger than level 1.</i>

Registers

Data stores on the CPU, with the fastest access possible, which store values used in ongoing computations.

Registers in the CPU

- Accumulator
- Current Instruction Register
- Program Counter
- Memory Address Register
- Memory Buffer Register

The Von Neumann Bottleneck

The **throughput of the data transfer between RAM and the CPU** limits the CPU's operating speed. It **takes longer to retrieve information from memory than it does to operate on it** because the data and instructions share a bus.

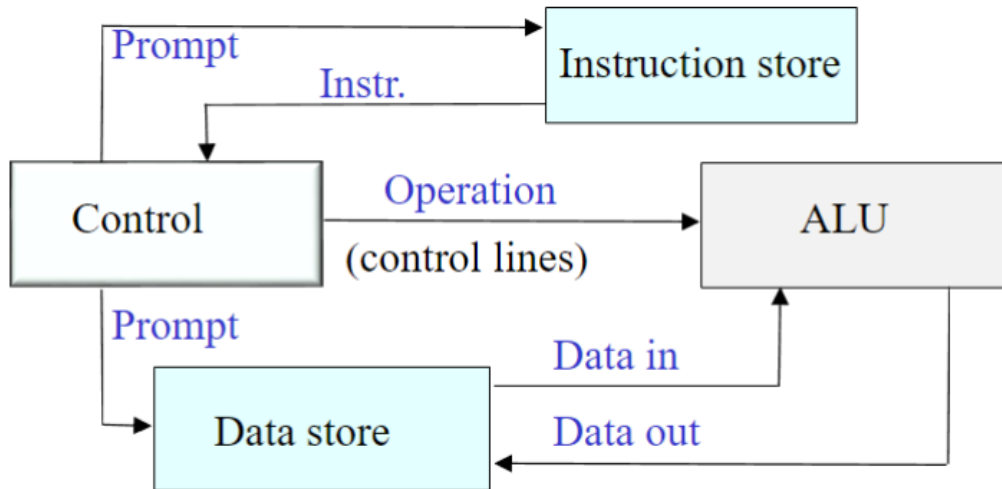
Solutions

- Alternative architectures (e.g. Harvard, distributed storage).
- Minimise use of main memory using cache and registers.

Locality of reference means this reduces the number of memory accesses.

Harvard Architecture (Uniprocessor)

Harvard architecture allows the CPU to access data and instructions simultaneously, **tackling the Von Neumann bottleneck**, by storing them separately. Increases processing speed at the cost of flexibility.



Use Cases

- **Special-purpose devices** like microcontrollers where instructions are stored in the ROM (Read Only Memory).
- **Sophisticated processors** which can exploit the parallel fetching of code and data.
- **Embedded systems:** A system with a dedicated function within a larger system. An embedded processor is one in an embedded system.

Possible Modifications

- Separate caching for code and data, but a single store, so the processor can function in either Harvard or Von Neumann mode.

Microcontrollers

A single integrated circuit containing an ALU, CU, I/O and memory.

- Good for embedded systems.

Software Bugs

An error, flaw or fault in a computer program or system that causes it to produce an incorrect or behave unexpectedly.

Parallel Architectures (Multiprocessor)

Multiprocessor:
Having multiple ALUs.

Parallel architectures carry out multiple operations simultaneously by using multiple ALUs.

Properties

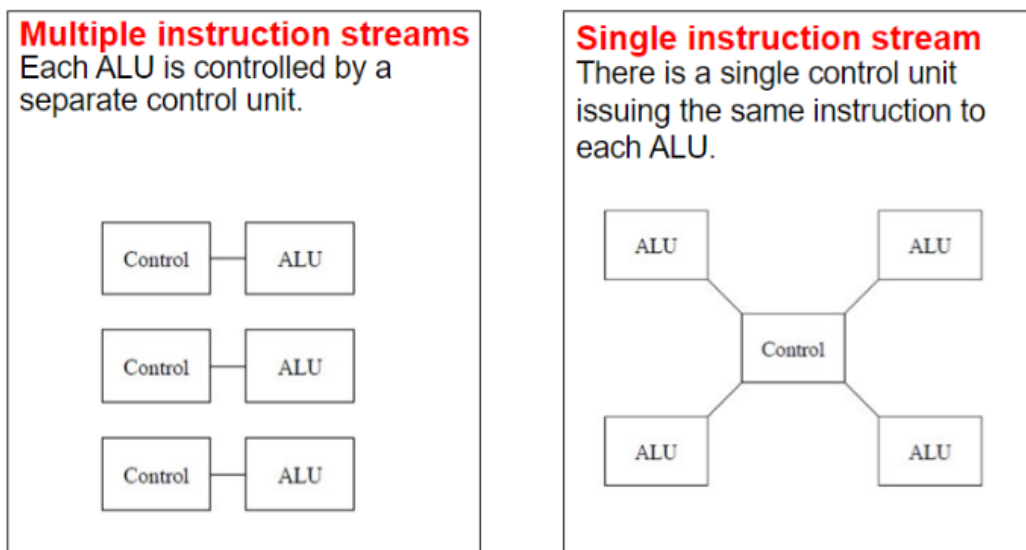
- Generally faster.
- Harder to understand, control & predict.

There is no dominant parallel architecture design. The design which you pick depends on whether each ALU has:

- its own control unit (**task level**).
- its own data storage (**data level**).

Parallelism at the Task Level

There are two possible **control architectures**:



Parallelism at the Data Level

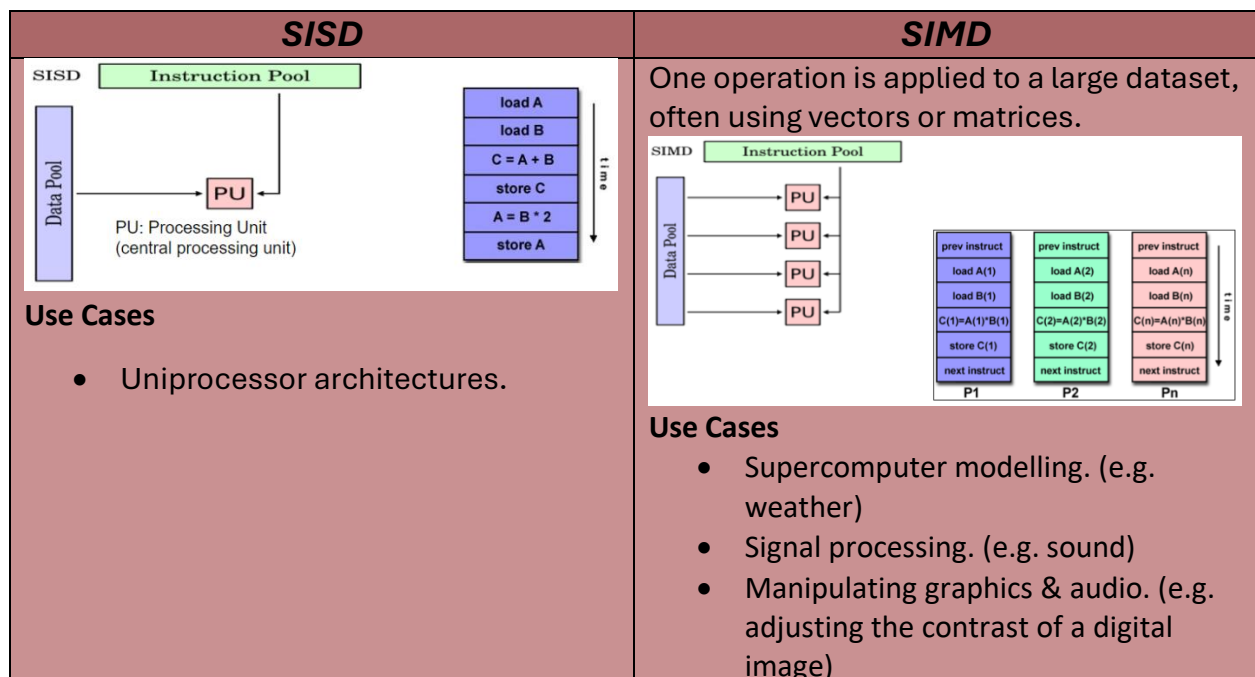
The ALUs may either:

- Operate on the same data stream.
- Each operate on a different data stream.

Flynn's Taxonomy

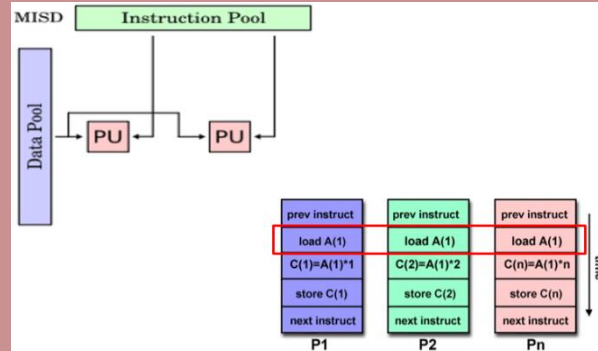
Taxonomy: The classification of something.

	Single Data Stream	Multiple Data Streams
Single Instruction	SISD	SIMD
Multiple Instruction	MISD	MIMD



MISD

Different operations are applied to the same data set.

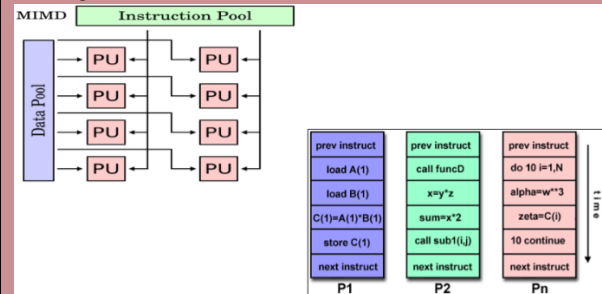


Use Cases

- Space shuttle flight control computers.

MIMD

Multiple processors each applying a different operation to a different data set. These processors are **asynchronous** and **independent**.

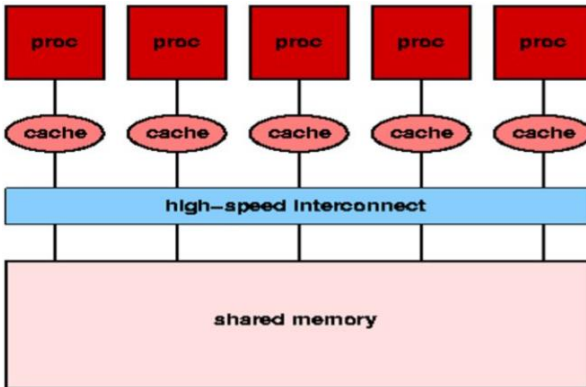


Use Cases

- Modern multi-core processors.

Parallelism and Memory

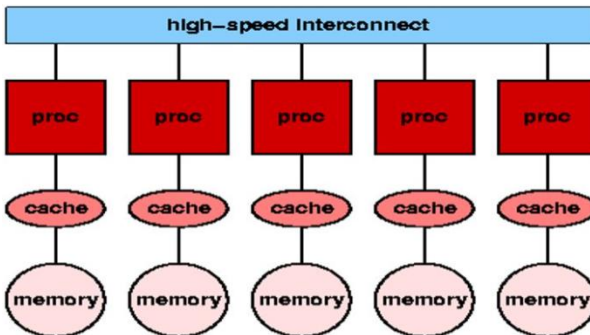
There are two main **memory architectures**. Both can be used in Von Neumann and Harvard architectures.



Shared Memory

All processors share the same memory.

- Leads to memory-to-CPU bottlenecks.
- Cache coherence becomes a problem.
- Doesn't scale well.



Distributed Memory

Each processor has its own memory.

- Easily scaled up for large numbers of CPUs.
- Communication between processors is done by **message passing** which is **indirect** and hence inefficient.
- **Distributing** (before computation) and then **reassembling** the data (after computation) is a complex task.

Other Memory Architectures

- **Virtual Shared Memory:** Distributed memory which seems shared to the CPUs.
- **Non-Uniform Memory Access:** Shared memory but some parts are faster for each CPU.

Shared Memory MIMD

Development of uniprocessor architectures is becoming less and less profitable. Hence, shared memory **MIMD is becoming dominant in general-purpose computing**.

- Multi-core processors are now standard for these machines.
- Allows applications which are intrinsically parallel to multi-task.

Multi-core Processor Architectures

Multi-core Processor: A special type of multiprocessor where all processors are on one chip.

Advantages

- Multitasking allows multiple programs to be run at once. (e.g. running antivirus while downloading software)

Disadvantages

- Difficult to design software for these architectures.

Amdahl's Law

Describes the **maximum speed change** gained by executing a program in parallel.

Amdahl's law states that if P is the proportion of a program that can be made parallel (i.e. benefit from parallelization), and $(1 - P)$ is the proportion that cannot be parallelized (remains sequential), then the maximum speedup that can be achieved by using N processors is:

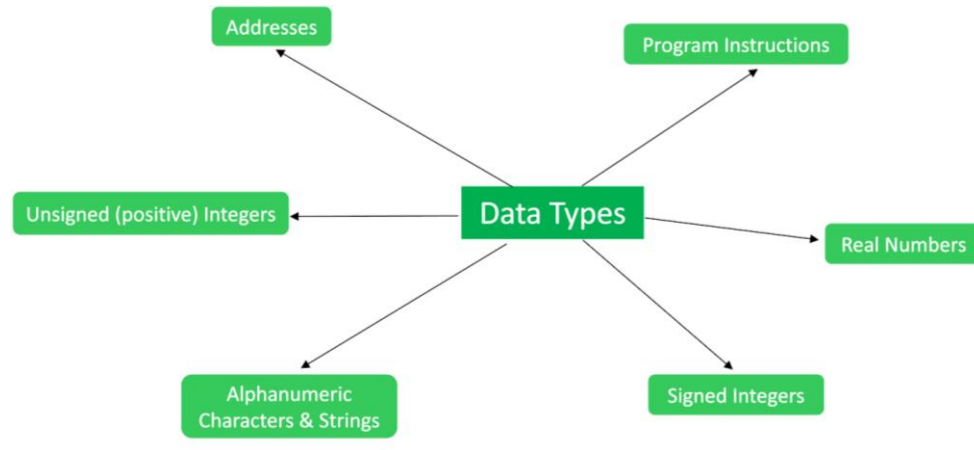
$$SpeedUp(P, N) = \frac{1}{(1 - P) + \frac{P}{N}}$$

As N tends to infinity, the maximum speedup tends to $1 / (1 - P)$.

The **more processors** you have, the **less speedup** you get with the addition of more processors. If a small proportion can be run in parallel, then the limit will be met relatively easily. If not, it will occur later.

Data Representation

Electronic computers are comprised of transistors which have two states, on or off, so we call them **two-state devices**. This is why we use the binary numeration system.



The exam is non-calculator, so you need to **memorise your powers of twos!**

$2^1 = 2$	$2^9 = 512$
$2^2 = 4$	$2^{10} = 1024$
$2^3 = 8$	$2^{11} = 2048$
$2^4 = 16$	$2^{12} = 4096$
$2^5 = 32$	$2^{13} = 8192$
$2^6 = 64$	$2^{14} = 16384$
$2^7 = 128$	$2^{15} = 32768$
$2^8 = 256$	$2^{16} = 65536$

Number Systems

Number systems are used to **represent and work with numbers**. There are many such systems used with computers.

- The base determines the cardinal number of standards (number of unique symbols).
- Each system has an agreed starting point denoted by a decimal point.

Properties of a Good Number System

- Few easy-to-remember symbols
- Unambiguous
- Easy to read & use.

Positional Notation

- **Place Value:** Increases from right to left in successive powers of the base starting at 0.
- **Value of a Digit:** Multiply the symbol (if it is not a number, its decimal equivalent) by its place value.
- **Value of a Number Comprised of Multiple Digits:** Add the values of all the digits.

Examples

1. $352 = (3 * 10^2) + (5 * 10^1) + (2 * 10^0)$
2. $179.32 = (1 * 10^2) + (7 * 10^1) + (9 * 10^0) + (3 * 10^{-1}) + (2 * 10^{-2})$

<i>The Decimal System</i>	<i>The Octal System</i>	<i>The Binary System</i>	<i>The Hexadecimal System</i>
<ul style="list-style-type: none"> • Base 10. • Digits 0-9. • A binary digit is called a bit. • 8 bits make a byte and 4 bits make a nibble. • A word of n bits can hold 2^n bit patterns. 	<ul style="list-style-type: none"> • Base 8. • Digits 0-7. 	<ul style="list-style-type: none"> • Base 2. • Symbols 0 & 1. 	<ul style="list-style-type: none"> • Base 16. • Symbols 0-F.

Converting to Decimal

Let b be the base of the number system and n be the length of the number.

$$x_1x_2x_3...x_n = (x_1 * b^{n-1}) + (x_2 * b^{n-2}) + (x_3 * b^{n-3}) + ... + (x_n * b^0)$$

Examples

Convert 134_8 to Decimal This is octal! (base 8) $= 1 * 8^2 + 3 * 8^1 + 4 * 8^0$ $= 64 + 24 + 4$ $= 92$	Convert $1FCB_{16}$ to Decimal This is hexadecimal! $= 1 * 16^3 + 15 * 16^2 + 12 * 16^1 + 11 * 16^0 = 2^8 + 2^6 + 2^5 + 2^3 + 1$ $= 4096 + 3840 + 192 + 11 = 8139$	Convert 101101001_2 to Decimal This is binary! $= 2^8 + 2^6 + 2^5 + 2^3 + 1$ $= 256 + 64 + 32 + 8 + 1$ $= 361_{10}$
---	---	---

Converting from Decimal to Binary

1. Write out the binary place values.
2. Going from left to right, if the number is greater than the place value, subtract it and put a 1 in that column.
3. The resulting number is your binary equivalent.

Converting from Decimal to Any Other Number System

1. Divide the decimal number by the base.
2. Multiply the number after the decimal by the base to get the remainder.
3. Repeat using the number before the point as the new decimal number.
4. Once you can't divide anymore, convert all remainders to the appropriate number system. (e.g. $11 = B$)
5. Reverse the result. This is your equivalent in the other number system.

Binary Addition

Similar to decimal addition except with a few extra rules.

1. $1+0=1$
2. $1+1=10$ (0 with an overflow 1)
3. $1+1+1=11$ (1 with an overflow 1)

Representing Data

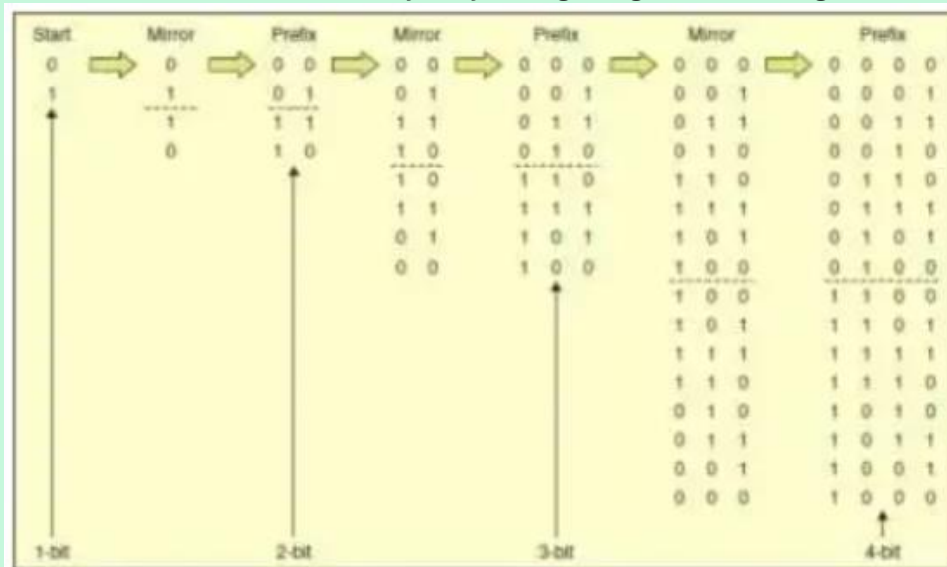
Unsigned Integers

Unsigned integers are positive integers.

Binary	Binary-coded Decimal	Binary-coded Sexagesimal
You can store 256 in a byte.	<p>Each digit is represented by its own 4-bit code. These codes are then compiled to form the full number.</p> <p><i>Converting 346 to BCD :</i></p> <p>3 = 0011, 4 = 0100, 6 = 0110</p> <p>$\therefore 346 = 001101000110$</p>	<p>The hour is represented by 4 bits. The minutes and seconds are represented by their own respective 6-bit codes. Good for time and angles.</p> <p><i>Converting 6 : 32 : 19 to BCS</i></p> <p>6 = 0110, 32 = 100000, 19 = 010011</p> <p>$\therefore 6 : 32 : 19 = 0110100000010011$</p>

Gray Codes / Reflected Binary Code

Successive values differ by only a single digit. Used in digital TV.



Signed Integers

Signed integers can be positive or negative integers. There are three major types of signed integers:

Sign & Magnitude	
The most-significant bit represents the sign of the integer (0 = positive, 1 = negative). The remaining bits represent its magnitude. <ul style="list-style-type: none">N bits can represent integers from $-(2^{N-1}-1)$ to $(2^{N-1}-1)$.	
Advantages	Disadvantages
<ul style="list-style-type: none">Easy to read.Symmetric about zero.	<ul style="list-style-type: none">Two representations of zero (+0 and -0).Can only represent 255 numbers in an 8-bit cell.Signs make arithmetic complex.

One's Complement	
The most-significant bit n is worth $-(2^n - 1)$. Positive integers have a leading 0 and negative integers have a leading 1. <ul style="list-style-type: none">You can represent the negative integer $-n$ by swapping the 1's and 0's in n.N bits can represent integers from $-(2^{N-1}-1)$ to $(2^{N-1}-1)$.	
Advantages	Disadvantages
<ul style="list-style-type: none">Easy arithmetic.	<ul style="list-style-type: none">Two representations of zero (+0 and -0).

Two's Complement	
The most-significant bit is worth -128 . Positive integers have a leading 0 and negative integers have a leading 1. <ul style="list-style-type: none">You can represent the negative integer $-n$ by swapping the 1's and 0's in n and adding 1.N bits can represent integers from $-(2^{N-1})$ to $(2^{N-1}-1)$.	
Advantages	Disadvantages
<ul style="list-style-type: none">Easy arithmetic.	<ul style="list-style-type: none">Not symmetric about zero.

- | | |
|---|--|
| <ul style="list-style-type: none">• One representation of zero. | |
|---|--|

Alphanumeric Characters

Each character has a **unique bit pattern**. Strings of characters are stored as a succession of bytes. In English, you need:

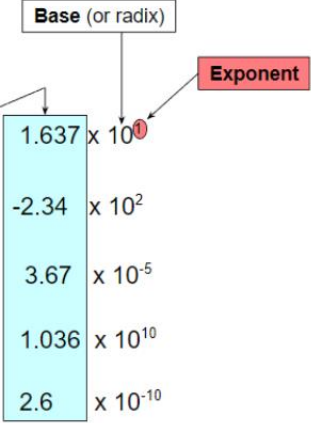
- 26 patterns for uppercase letters.
- 26 for the lowercase letters.
- 10 for the decimal digits.
- A handful for punctuation marks, asterisks and other graphic symbols.

Hence, fewer than 128-bit patterns are needed, so we can store individual character codes within a single byte.

Encoding Conventions

- **American Standard Code for Information Interchange (ASCII):** Uses 7 bits and is very common.
- **UNICODE:** Uses 32 bits and can represent over 2 billion symbols. Enables the use of a wider range of characters from other languages.
- **UTF-8:** Allows for variable-width character storage making UNICODE backwards compatible with ASCII and vice versa.

Real Numbers

Floating-point Notation	
<div><p>Mantissa (or significand) Normalized: between 1 and less than the base (zero is an exception) (could be positive or negative)</p><p>Base (or radix)</p><p>Exponent</p></div>	<p>Any non-integer number can be represented using floating-point notation.</p> <p><i>Advantages</i></p> <ul style="list-style-type: none">• Simple to generate, understand and manipulate.• Can represent a wide range of numbers in a compact way.• Accurate to the precision of the representation. <p><i>Disadvantages</i></p> <ul style="list-style-type: none">• Approximates numbers which could be represented exactly. For example, recurring decimals such as 1/3 or pi.

Normalising Floating-point Numbers

A floating point number of the form:

$$\pm m * R^{\pm x}$$

is normalised if:

$$1 \leq |m| < R$$

Excluding zero where $m = 0$ and $x = 0$.

Note: For a fixed radix, we can write numbers like 2.340×10^3 as 2.34E3.

Binary Floating-point Numbers

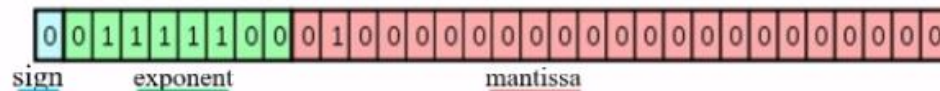
The radix is 2. Moving the point to the **right removes one** from the exponent and moving it to the **left adds one**.

IEE Standard for Binary Floating-point Arithmetic

The IEEE standard ensures that all computers use the same method which enables compatibility between devices. There are three basic formats:

- **Half-precision (16-bit):** 1 SB, 5 E, 10 M, Bias = 15
- **Single Precision (32-bit):** 1 SB, 8 E, 23 M, Bias = 127
- **Double Precision (64-bit):** 1 SB, 11 E, 52 M, Bias = 1023

Single Precision



- **Mantissa:** Denotes the value behind the decimal point. There is always a 1 before the decimal point so we don't need to include this but must account for it in conversion.
- **Exponent:** Denotes the number of places to shift the decimal point. A **bias** of 127 was added when we created it. This needs to be taken away to calculate the actual value of the exponent.
- **Sign:** Takes up 1 bit. (0 = positive, 1 = negative)

Converting from IEE Form to Decimal

1. Calculate the values of the individual components using the formulae:

Sign Bit: (0 = positive, 1 = negative)

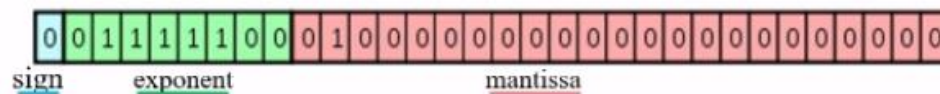
Exponent: (Value stored in the exponent) - Bias

Mantissa: $1 + (\text{sum of the actual mantissa})$

Sum of Mantissa: Add up the value of every bit which is 1. The value of each one is denoted as: 2^{-n} where n is the number of places the bit is to the right.

2. Use the formulae to convert it to a decimal real: $Number = \pm m * R^{\pm x}$

Example: Converting Single Precision to Decimal



Sign bit = 0 = Pos

Exponent = 01111100 = 124 \rightarrow 124 - 127 = -3

Mantissa = $1 + 2^{-2} = 1.25$

So, the value = $1.25 \times 2^{-3} = 0.15625$

Converting from IEE Form to Decimal Real

1. Calculate the value of the sign and the mantissa using the formulae:
Sign: 0 for positive, 1 for negative.
Mantissa: The first half is the binary equivalent of the number before the point.
2. Calculate the second half of the mantissa:
 - Starting with the value after the point, multiply it by 2.
 - The number before the point in the resulting number corresponds to the binary.
 - If this is 1, remove it and repeat.
3. Combine the two to produce the mantissa.
4. Normalise the mantissa. We do this by moving the point so that the leftmost 1 is before it. Every move to the left, adds one and a move to the right removes one from the exponent. Then, remove the 1 before the point because this is not stored.
5. Add the exponent bias.
6. Combine everything to produce your result: sign + exponent + mantissa. If any of the parts don't meet their full length, just add zeroes to the back.

Example: Converting Decimal to Single Precision

Convert 13.1875 into a single precision IEE floating-point number.

1. Sign = 0 and the first part of the mantissa = 13 = 1011.
2. The second part of the mantissa is 0.1875.
 $0.1875 \times 2 = 0.375$
 $0.375 \times 2 = 0.75$
 $0.75 \times 2 = 1.5$
 $0.5 \times 2 = 1$
3. Hence, 13.1875 = 1101.0011
4. 1101.0011 becomes 1.1010011 which requires moving the point three times to the left. Hence, the exponent becomes 3. Remember: the mantissa is 1010011
5. The bias of single precision is 127 so the exponent becomes $3 + 127 = 130 = 10000010$.
6. Result: 01000001010100110000000000000000

Exponent Special Cases

If the mantissa is zero:

- An exponent of 0 represents zero.
- An exponent of 255 represents infinity.

Operations

Addition

Illustrated in decimal for convenience:

$$(9.6 \times 10^2) + (6.6 \times 10^1)$$

- 1) **De-normalize** the smaller operand and adjust its exponent to equal that of the other operand:
 $(9.60 \times 10^2) + (0.66 \times 10^2)$
- 2) **Add** the mantissae and retain the common exponent:
 (10.26×10^2)
- 3) **Re-normalize** the mantissa and adjust the exponent if necessary:
 (1.03×10^3)

Multiplication

Illustrated in decimal for convenience:

$$(5.2 \times 10^{-4}) \times (2.6 \times 10^7)$$

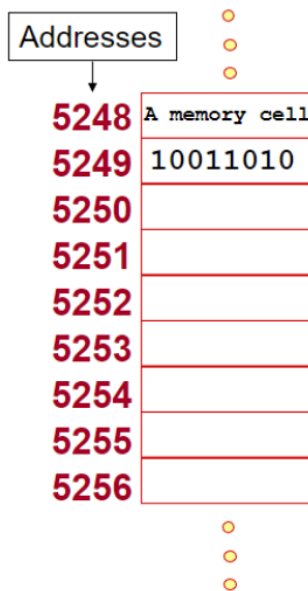
- 1) **Multiply** the mantissae and **add** the exponents. This will yield:
 (13.56×10^3)
- 2) **Renormalize** the mantissa and adjust the exponent if necessary:
 (1.356×10^4)

Storing Data

A bit is stored using a two-state device called a **transistor**. Bits are grouped into cells which collectively form a **store** which is a form of digital memory.

Store: A collection of cells.

Cells



A **cell** is the smallest addressable unit of a store.

Contents

- Contents are determined by the states of several bits.
- Represented as binary digits or constants. (e.g. *011101100* or *67*)
- Their length varies between machines but all cells in a store will have the same number of bits.
- In Von Neumann architecture, can contain instructions or data but you can't tell which by looking at them.

Addresses

- Each cell has a unique address numbered from 0 to (n-1) for a store of n cells.
- Addresses are used to locate a cell before accessing its contents.

Types of Memory Addresses

- **Sequential Access:** Access time relies on the address of the last cell accessed.
- **Random Access Memory (RAM):** Access time doesn't depend on the address.

Address Spaces

The **total number of addresses** associated with **physical** or **virtual locations** such as cells, I/O ports, IP addresses on a network, etc.

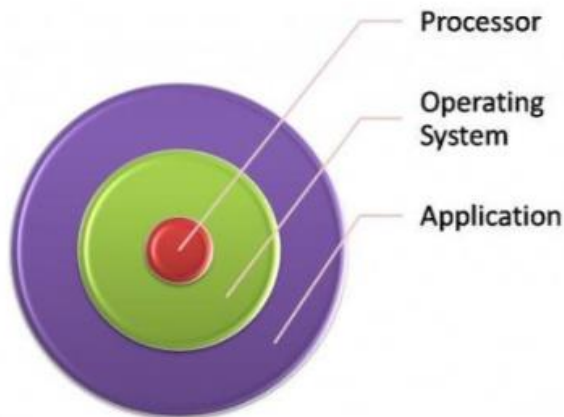
- One address can refer to different locations in different spaces.
- The address length determines the amount of memory that a given space has.

Virtual Memory

The temporary use of secondary storage as additional main memory. Used when the amount of memory needed to hold all running programs exceeds the amount of RAM. It frees up memory allowing unused data in RAM to be transferred to secondary storage.

When data in virtual memory is needed, it is swapped with other unused data in the RAM.

Components of Different Address Spaces Working Together



This diagram demonstrates how a **system's processing capabilities are often limited by its worst-performing component.**

E.g. a 64-bit app must have a 64-bit OS which needs a 64-bit CPU.

However, a 64-bit CPU doesn't need a 64-bit OS and apps. So, A 64-bit CPU can run 32-bit apps, but a 64-bit app can't be run on a 32-bit CPU.

Quantifying Memory

1024 (2^{10}) bytes = Kilobyte (Kbyte) (2^{10} bytes)

1024 (2^{10}) Kbytes = Megabyte (2^{20} bytes)

1024 (2^{10}) Mbytes = Gigabyte (2^{30} bytes)

1024 (2^{10}) Gbytes = Terabyte (2^{40} bytes)

Instructions

Von Neumann systems execute a series of instructions which **specify data-processing and control operations**.

$$\text{Instruction} = \text{Operation} + \text{Operands} + \text{Mode}$$

Instruction Set: A processors assembly language operations.

- Instruction sets should leverage **expressive power** and **compactness**.
- Operands are usually hexadecimal addresses (**h**) in memory which contain the actual operand. E.g. *Add 0Ah, 0Ch*
- There are many types: arithmetic, Boolean, movement of data, I/O & control.
- Other noteworthy instructions: Store, compare, jump & jump if (condition).

Representation

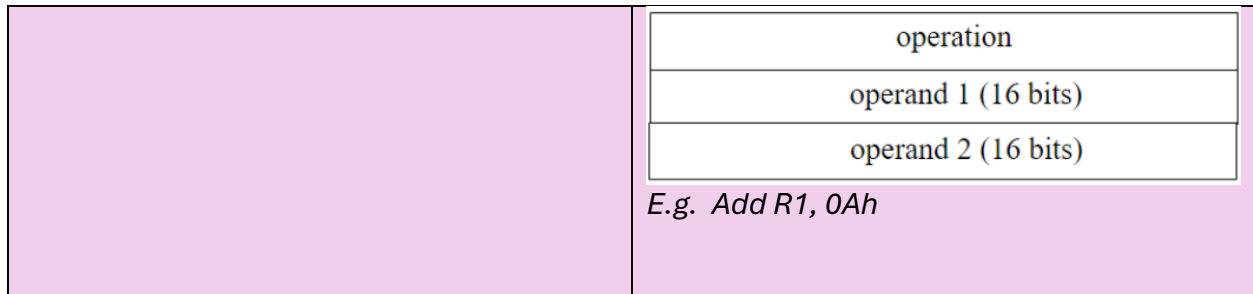
Instructions are usually stored like this. Sometimes, the operation is called the **control word** and the data following it the **extension word**.

Operation	Operand 1	Operand 2
2 bytes	2 bytes	2 bytes

Address Codes

The address code used for instructions depends on the number of accumulators.

One Address Codes	One and a Half Address Codes						
<p>Only one register which is always used as one operand of a two-operand operation. So, it doesn't need specifying. Assuming that:</p> <ul style="list-style-type: none">• Load & store exist. <i>E.g. Load 10</i> <table><tr><td>operation (2 bytes)</td></tr><tr><td>operand (2 bytes)</td></tr></table>	operation (2 bytes)	operand (2 bytes)	<p>Multiple registers and only one is referenced in each instruction. The register to use is referred to using 3 bits.</p> <table><tr><td>operation (13 bits)</td><td>data reg. (3 bits)</td></tr><tr><td colspan="2">operand (16 bits)</td></tr></table> <p><i>E.g. Load R3, 10</i></p>	operation (13 bits)	data reg. (3 bits)	operand (16 bits)	
operation (2 bytes)							
operand (2 bytes)							
operation (13 bits)	data reg. (3 bits)						
operand (16 bits)							
Zero Address Codes	Two Address Codes						
<p>No operands from memory.</p> <table><tr><td>operation (13 bits)</td><td>data reg. (3 bits)</td></tr></table> <p><i>E.g.</i> <i>Inc R1</i> <i>Halt</i></p>	operation (13 bits)	data reg. (3 bits)	<p>Both operands are an accumulator or a memory address.</p>				
operation (13 bits)	data reg. (3 bits)						



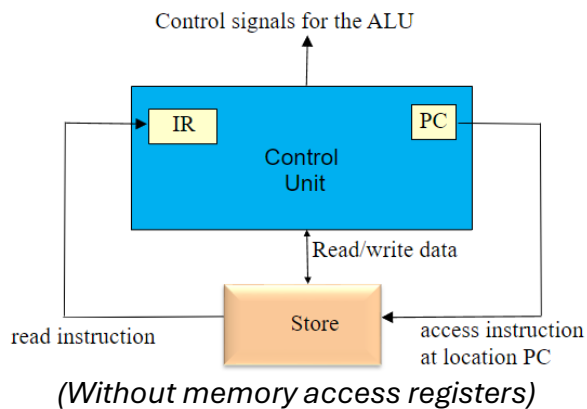
Handling Instruction Sequences

Storing successive instructions in successively higher-addressed addresses allows us to execute a program by stepping through the store addresses in order.

Program: An ordered sequence of instructions.

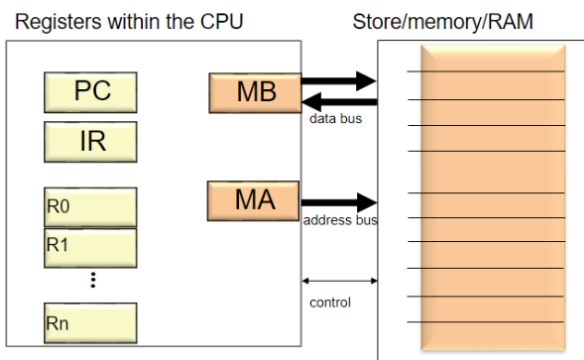
Control Unit Registers

Some are used to keep track of instruction sequence execution:



- **Program Counter (PC):** Stores the address of the next instruction to be executed. Increments after an instruction is fetched.
- **Instruction Register (IR):** Stores a copy of the instruction being executed.

Others are used to access memory:



- **Memory Address Register (MAR):** Stores the address that is being accessed.
- **Memory Buffer Register (MBR):** Stores the contents of the address being read or written to.

The Fetch-Decode-Execute Cycle

Used to execute instructions.

Fetch	Decode	Execute
<ol style="list-style-type: none">1. Address in the program counter (PC) is copied to the memory address register (MA).2. Contents of the address in the MA are copied to the memory buffer (MB).3. Contents of the MB are copied to the instruction register (IR) inside of the control unit (CU).4. The PC is incremented to the next instruction to be accessed (<i>Not necessarily to the next address because this could contain data.</i>)	<p>The instruction is decoded by the control unit into the operation to be carried out.</p>	<p>Highly dependent on what the operation is. For example, if it is to copy the next address to a data register or add the specified address's contents to the data register. We will do the later:</p> <ol style="list-style-type: none">1. MA is incremented.2. Contents of the address in the MA are copied to the MB. In this case, it's another address.3. Contents of the MB are copied to the MA.4. Contents of the address in the MA are copied to the MB.5. Contents of the MB are added to the contents of the data register specified in the operation.

Operand Modes

Specify how to treat the extension words. For example, in the MC68000 architecture “111000” means to treat the extension word as an address. There are a few types:

	Immediate Addressing	Direct Addressing	Indirect Addressing
Desc.	Treat as the data itself.	Treat as an address containing the data.	Treat as the address of an address containing the data.
Ex.	$D1 = D1 + \$101E$	$D1 = D1 + mem(101E)$	$D1 = D1 + mem(mem(101E))$

Instruction Set Design Philosophies

Complex Instruction Set Computer (CISC)	
Giving processors a higher number of more complex instructions.	
<i>Advantages</i>	<i>Disadvantages</i>
<ul style="list-style-type: none">• Fewer lines needed.• More Powerful Instructions: Meaning simpler compilers and higher performance.	<ul style="list-style-type: none">• Less Registers in Processor: More transistors needed to decode instructions.• Decoding is More Complex.

Reduced Instruction Set Computer (RISC)	
Reducing the number of instructions and making them simpler.	
<i>Advantages</i>	<i>Disadvantages</i>
<ul style="list-style-type: none">• Simpler Instructions• Simpler Decoding• Uniform Instruction Execution: Each only takes one clock cycle.	<ul style="list-style-type: none">• More Memory Needed: More code so more memory needed for instructions.• More Compiler Work: More difficult to convert high-level code into RISC statements.

Control Instructions

Usually instructions are executed sequentially. **Control instructions** change this order by branching or calling a procedure.

Why are They Needed?

- Some instructions need to be executed multiple times.
- All programs require some decision-making.

Branching

Branching is altering the contents of the program counter (PC) using a conditional or unconditional jump.

For example, *JNE* ___ means “Jump if not equal to”.

- One of the operands is the address of the first instruction to be executed if the branch is followed.

- **Conditional branches** may contain extra operands (e.g. *BRE R1, R2, 235 increments to address 235 if R1 and R2 are not equal.*) If the condition is met, they jump to the specified address. If not, the next instruction is executed as normal. They allow programmers to make **programming control structures** like loops.
- **Unconditional branches** contain an address which is jumped to by copying it to the PC. (e.g. *JMP ____*)

Condition Codes/Status Flags

Used in conditional branching to denote the state of the ALU:

CF - carry flag:	Set on bit carry; cleared otherwise
PF - parity flag:	Set if low-order eight bits of result contain an even number of "1" bits; cleared otherwise
ZF - zero flags:	Set if result is zero; cleared otherwise
SF - sign flag:	Set equal to high-order bit of result (0 if positive, 1 if negative)
OF - overflow flag:	Set if result is too large a positive number or too small a negative number (excluding sign bit) to fit in destination operand; cleared otherwise

Subroutines/Procedures

Enable task repetition by encapsulating a block of instructions in memory which can be called by branching when needed. When it is called:

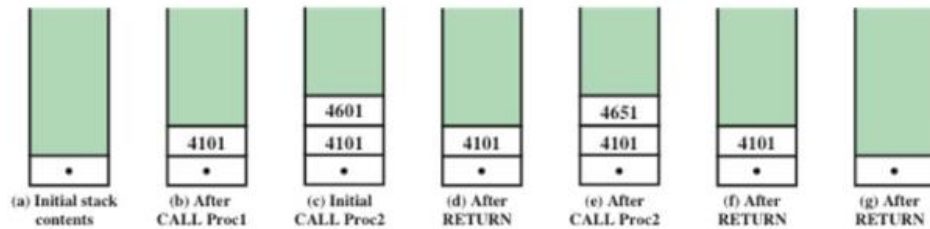
1. The main program is suspended.
2. The contents of the PC (the address after the call instruction) are pushed to the stack.
3. The PC is updated with the address of the subroutine.

When it ends, a return instruction is executed which pops the address after the call instruction from the stack and updates the PC with this.

Stacks

Operate on a **last-in, first-out** principle. They have the operations:

- **Push:** Add a data element to the top.
- **Pop:** Remove a data element from the top.



Stack Overflow: When the stack runs out of storage.

Implementation

- Contents stored using a block of successive memory locations.
- They start at high areas of memory and grow from high to low addresses.
- A register, called the stack pointer (SP), stores the address of the top of the stack.
- To push a value, decrement the stack pointer.
- To pop a value, increment the stack pointer.

Stack Frames

Stores extra information about each subroutine call such as:

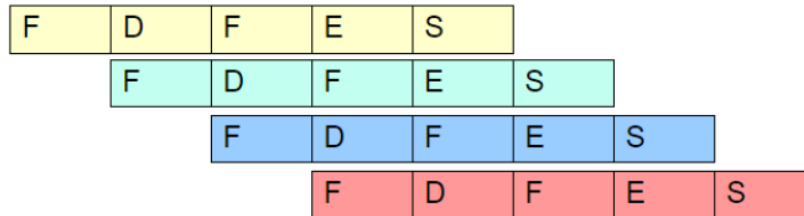
- The return address.
- Local variables
- Copies of registers modified by the subroutine so they can be restored.
- Argument variables passed on the stack.

Latency: Time taken to do a task.

Throughput: Tasks completed per unit time.

Pipelining

Most cycles involve a **f**etch, **d**ecode, maybe another **f**etch, an **e**xecute and a **s**toe – **FDFES**.



Pipelining is a technique where multiple instructions overlap during execution.

- Saves processing time, increasing efficiency.
- Improves the **throughput** but not the **latency** of cycles.

Problems

- **Instruction Dependencies:** If an instruction requires the result of the one before it, it must wait for it to be computed before executing.
- **Load Delays:** Fetching is slower than decoding or executing which can cause the pipeline to stall. Executing other instructions in the meanwhile can mitigate this problem.
- **Unequal Stage Times:** A particularly slow stage will slow proceeding stages down.
- **Branches:** Break the pipeline so try to avoid in code. Can be mitigated by predictive decoding.

Superscalar Architectures

A type of instruction parallelism where instructions are **executed in parallel if they are data independent**, the CPU checks this when it begins an instruction cycle.

This is achieved using a processor's **redundant functional units**, duplicates of some execution resource within each processor (e.g. ALUs). If an instruction is data independent, the CPU can execute them on these extra **redundant functional** units, allowing multiple instructions to be executed in parallel.

Input & Output

The **methods to transfer information between a computer and external devices**. Any **transfer of information beyond the CPU and main memory is I/O**. (e.g. *disk storage*) It connects a computer system to the outside world and is **slower than the CPU and RAM**. Different types of I/O devices have different operating speeds. If these don't match the speed of the I/O module, timing problems can occur. From fastest to slowest:

- Electronic (e.g. *other computers*)
- Electromechanical (e.g. *printers, disks*)
- Human (e.g. *keyboard, mouse*)

External Devices

Allow computers to exchange data with their external environment.

I/O Module: Links a computer to an peripheral device by exchanging control, status and data with it.

Peripheral: An external device connected to an I/O module.

There are three types of external devices:

- **Human Readable:** Communicates with user. (e.g. *monitors & printers*)
- **Machine Readable:** Communicates with equipment. (e.g. *disk & tape systems*)
- **Communication:** Communicates with remote devices such as another computer.

Functions of I/O Modules

The major functions of I/O modules fall into the following categories:

- **Control & Timing:** Controls the flow of traffic between internal resources and external devices.
- **Processor & Device Communication:** Command decoding, handling data, status reporting and address recognition.
- **Data Buffering:** Performs buffering to balance device and memory speeds.
- **Error Detection:** Detects and reports transmission errors.

Techniques for I/O

- **Programmed I/O:** See corresponding section.
- **Interrupt-driven I/O:** Processor issues I/O command, executes other instructions and is interrupted when it finishes. Data is transferred between the I/O and memory through the processor.

- **Direct Memory Access:** The I/O module and memory exchange data directly without processor involvement.

Evolution of I/O

I/O control became **increasingly processor independent**, improving performance. Eventually, the I/O module became a processor, giving it the name **I/O processor**.

1. The CPU directly controls peripherals.
2. I/O controller/module added which allows the CPU to use programmed I/O.
3. Interrupts were employed allowing the processor to resume execution while the peripheral is busy rather than waiting for its device flag to be raised.
4. I/O controller is given DMA meaning transfers only require CPU involvement at the start and end of a transfer.
5. I/O module is enhanced to become a processor with a specialised instruction set tailored for I/O.
6. I/O module is given its own local memory allowing many peripherals to be controlled with minimal CPU involvement.

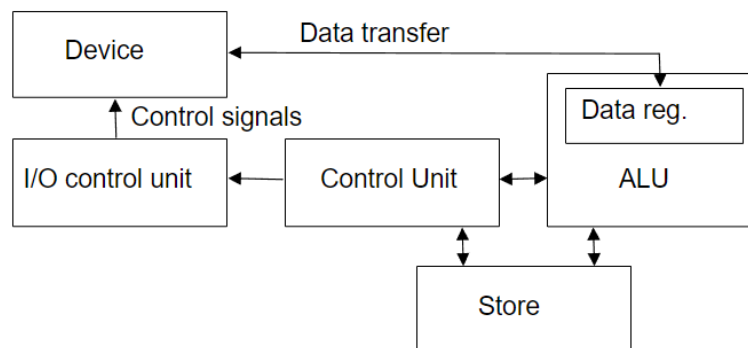
Programmed I/O

In programmed I/O, **programs are given direct control of I/O operations** meaning programmers write code to access data and transfer it to or from I/O devices. Data is exchanged between the memory and the I/O module through the processor.

- The processor must wait until an I/O operation is complete to execute a command.
- If the processor is faster than the I/O module, this is a waste of time.
- Often involves an electromechanical process

Controlling I/O Devices

An **I/O control unit**, attached to the control unit, oversees the input and output interactions between the CPU and peripheral devices.

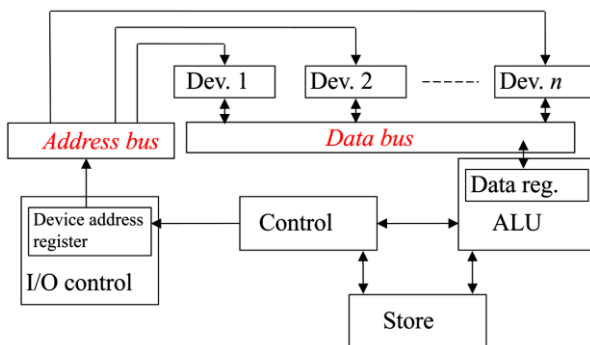


1. An instruction is fetched from the memory to the instruction register.
2. It is decoded by the control unit and turns out to be an I/O operation.
3. The instruction is sent from the CU to the I/O CU.
4. The I/O CU sends signals to the peripheral to set it into action.
5. Data is written or read from the device using the data registers in the ALU.

I/O Instruction Structure

I/O instructions are coded into a bit pattern with three parts:

- **I/O Operation Code:** The operation to be performed.
- **Direction of Transfer.**
- **Address of the Device:** Allows data to be placed on a data bus and be picked up by the target device.



Multiple I/O Devices

Multiple peripherals require the following supporting hardware:

- A **data bus** which transfers data between the data registers and the device's data buffers.
- An **address bus** which transfers address information to the peripherals, allowing them to be identified.

The I/O CU also has a **device address register** which stores the address of the device currently being communicated with.

Each peripheral has a **data register/buffer** which holds data passing into or out of it and can be used to identify the device.

Shared I/O Busses	
Advantages	Disadvantages
<ul style="list-style-type: none"> • Simple & Flexible: Provides a single I/O control system with a single connection to the CPU. 	<ul style="list-style-type: none"> • Cost of Building It • Shared Resources: Limited throughput may cause contention for data transmission. Could be mitigated by intelligent program design or additional hardware.

Device Flags

Each device has a **status flag** which is a one-bit register that indicates whether it is busy (0) or idle (1). The **device itself** unsets the status flag when it begins an I/O transfer and sets it when it finishes. The control unit tests this flag to check whether a device is ready for another data transfer.

Busy Wait Strategy

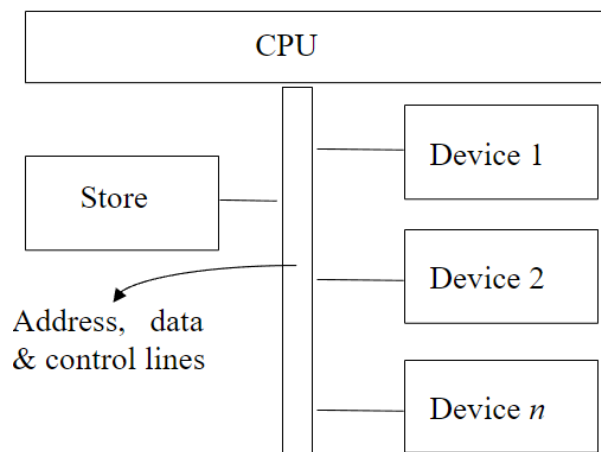
1. Load the next data to be sent to the peripheral into the I/O CU and data registers.
2. Issue a read status flag command to the I/O module.
3. Read the relevant status flag.
4. If the flag is not set, repeat step 3.
5. Read/write data from/to the device.

Polling Strategy

1. Load the next data to be sent to the peripheral into the I/O CU and data registers.
2. Issue a read status flag command to the I/O module.
3. Read the relevant status flag.
4. If the flag is not set, execute another operation for a fixed time and then repeat step 3.
5. Read/write data from/to the device.

<i>Advantages</i>	<i>Disadvantages</i>
<ul style="list-style-type: none">• Enables background computation while waiting for a busy peripheral.	<ul style="list-style-type: none">• Unresponsiveness: If the device flag becomes idle after just being polled, then data transfer must wait until the next poll.• Multiple Devices: Fast & slow devices may need to carry out I/O simultaneously.• Complexity: More difficult to program.

Memory-mapped I/O



Memory-mapped I/O **uses some of the addresses in the address space for peripheral addresses** rather than just store locations. So, the processor can operate I/O devices by loading and storing data as if it were doing so in the memory.

This means that **no special I/O instructions are needed in the instruction set.**

An address bus with 24 address lines can access 2^{24} addresses. In memory-mapped I/O, some of these addresses would be used to access peripherals. It works as follows:

- I/O data is transmitted using the data bus.
- Peripheral addresses are transmitted using the address bus.
- The direction of transfer, whether the operation is to read or write, is transferred using the control bus.

<i>Advantages</i>	<i>Disadvantages</i>
<ul style="list-style-type: none">• Simplifies I/O Control: Frees up physical space on the CPU.	<ul style="list-style-type: none">• Uses Address Space: Could be used for storage.• Bus Contention: I/O devices are in contention with the store for throughput.• Programs Harder to Understand: I/O and storage can be confused.

Interrupt Driven I/O

No time is wasted waiting for a peripheral because the processor is only interrupted when it is ready for I/O transfer.

1. When a peripheral is ready for data transfer, its I/O controller raises its status flag and sends an interrupt request (IRQ) to the CPU.
2. The processor finishes its current process.
3. If the interrupt request is accepted, the peripheral is identified.
4. The PC and program status flags are pushed to the control stack.

5. The PC's contents are replaced with the address of the first **interrupt service routine (IRS)** instruction stored in an **interrupt vector** in memory.
6. All registers are saved to the stack.
7. The peripheral is **serviced** meaning the **IRS is executed**.
8. The last instruction of the IRS restores the process state by popping from the stack.
9. The processor resumes the background process where it left off.
10. The I/O module sends another IRS when it finishes.

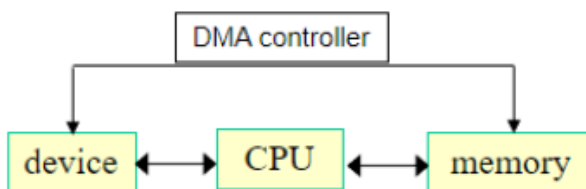
Multiple Devices

Each peripheral is given a priority using a **programmable interrupt controller**. When an IRQ is sent to the processor, it compares the peripheral's priority to the CPU's current operating priority and will only accept the IRQ if the peripheral's priority is higher.

- The stack handles nested interruptions.

<i>Advantages</i>	<i>Disadvantages</i>
<ul style="list-style-type: none"> • Wastes Less CPU Time • More Responsive • Easily Handles Multiple Devices: Using priorities. 	<ul style="list-style-type: none"> • Special Hardware Required: Programmable interrupt controller. • Context Switching Uses CPU Time: Means popping and pushing to the stack lots which takes up processing time and means devices wait longer to be serviced.

Direct Memory Access (DMA)



A DMA controller allows data transmission between the memory and peripherals to be independent of the processor.

The CPU starts the transfer, can do something else in the meanwhile and is interrupted when the transfer finishes.

<i>Advantages</i>
Direct memory access is useful where: <ul style="list-style-type: none"> • Large quantities of data are being transmitted. • The CPU cannot keep up with the rate of data transfer. • The CPU needs to perform work simultaneously as I/O transfer. • Many peripherals need to be controlled with minimal CPU involvement.


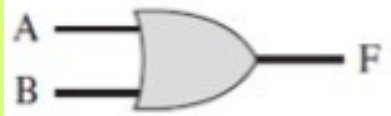
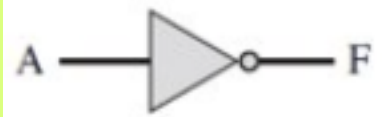

Boolean Logic & Circuits

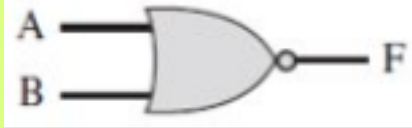

All operations, such as arithmetic, logic, data storage and control, are **implemented using electronic circuits**.

Types of Logic Circuits	
<i>Combinational</i>	<i>Sequential</i>
Combine their inputs in a way that is: <ul style="list-style-type: none"> • Static: Doesn't change over time. • Deterministic: Each input has a unique output. Used to implement "timeless" operations such as arithmetic.	Incorporate feedback from their inputs. They are: <ul style="list-style-type: none"> • Dynamic: Change over time. • Can be Non-deterministic: Several outputs for a given input. Used to implement "stateful" operations such as memory cells.

Boolean Algebra

Associating the high voltage and low voltage with 1 and 0 respectively, the ability to use Boolean variables and operators allows us to **describe circuits using Boolean logic and logic gates** which perform complex logical operations.

Name	Graphical Symbol	Algebraic Expression	Truth Table															
AND		$A \cdot B$	<table><tr><th>A</th><th>B</th><th>R</th></tr><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></table>	A	B	R	0	0	0	0	1	0	1	0	0	1	1	1
A	B	R																
0	0	0																
0	1	0																
1	0	0																
1	1	1																
OR		$A + B$	<table><tr><th>A</th><th>B</th><th>R</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	A	B	R	0	0	0	0	1	1	1	0	1	1	1	1
A	B	R																
0	0	0																
0	1	1																
1	0	1																
1	1	1																
NOT		$A' \text{ or } \bar{A}$	<table><tr><th>A</th><th>R</th></tr><tr><td>0</td><td>1</td></tr><tr><td>1</td><td>0</td></tr></table>	A	R	0	1	1	0									
A	R																	
0	1																	
1	0																	
NAND		$\overline{A \cdot B}$	<table><tr><th>A</th><th>B</th><th>R</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	A	B	R	0	0	1	0	1	1	1	0	1	1	1	0
A	B	R																
0	0	1																
0	1	1																
1	0	1																
1	1	0																

NOR		$\overline{A + B}$	<table><tr><th>A</th><th>B</th><th>R</th></tr><tr><td>0</td><td>0</td><td>1</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>0</td></tr></table>	A	B	R	0	0	1	0	1	0	1	0	0	1	1	0
A	B	R																
0	0	1																
0	1	0																
1	0	0																
1	1	0																
XOR		$A \oplus B$	<table><tr><th>A</th><th>B</th><th>R</th></tr><tr><td>0</td><td>0</td><td>1</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></table>	A	B	R	0	0	1	0	1	0	1	0	0	1	1	1
A	B	R																
0	0	1																
0	1	0																
1	0	0																
1	1	1																

Proving Equivalence by Algebra

Use equivalence laws to get from one formula to another. E.g.

$$A + A.B$$

$$\begin{aligned}
 &= A.1 + A.B \\
 &= A.(1 + B) \\
 &= A.1 \\
 &= A
 \end{aligned}$$

Identity: $A.1 = A$
Distributive: $A.(1+B) = A.1 + A.B$
Annihilator: $A+1 = 1$

Show that $A + A.B = A$
by truth table

A	B	A.B	A + (A.B)
0	0	0	0
0	1	0	0
1	0	0	1
1	1	1	1

Proving Equivalence by Truth Tables (Model Checking)

Use truth tables to show that two formulae produce the same outputs.

Useful Laws

- $A \oplus B = A\bar{B} + \bar{A}B$
- $A \text{ XNOR } B = \overline{A \oplus B} = A.B + \bar{A}.\bar{B}$

Simplifying Boolean Formulae

To **effectively implement a Boolean formula as a circuit**, we must find its simplest form.

Standard Sum of Products

The same as **disjunctive normal form**, disjunction of conjunction. Putting expressions in this form is the first step of simplification.

- AND operations joined by OR operations.
- All variables must appear in each term.

Every Boolean expression can be given in **disjunctive normal form**.

Example – Using Laws

1. Identify the terms that don't contain all variables.
2. Add the appropriate variable using the law:
3. Repeat.

$$\begin{aligned}A + B.C &= A.(B + B') + B.C && \boxed{A + A' = 1} \\&= A.B + A.B' + B.C \\&= A.B.(C + C') + A.B' + B.C \\&= A.B.C + A.B.C' + A.B' + B.C \\&= A.B.C + A.B.C' + A.B'.(C + C') + B.C \\&= A.B.C + A.B.C' + A.B'.C + A.B'.C' + B.C \\&= A.B.C + A.B.C' + A.B'.C + A.B'.C' + B.C.(A + A') \\&= \underline{A.B.C} + A.B.C' + A.B'.C + A.B'.C' + \underline{A.B.C} + A'.B.C \\&= A.B.C + A.B.C' + A.B'.C + A.B'.C' + A'.B.C\end{aligned}$$

Example – Using Truth Tables

1. Write out the truth tables for the expression.
2. For each row of the final expression, write the product of all literals.
3. Sum all these terms.

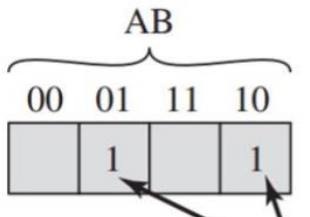
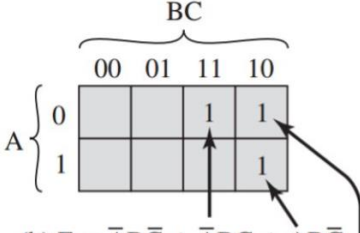
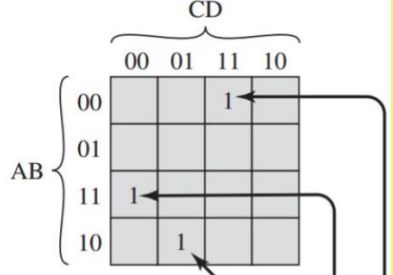
A	B	C	(B.C)	A+(B.C)
0	0	0	0	0
0	0	1	0	0
0	1	0	0	0
0	1	1	1	1
1	0	0	0	1
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

↓
(A'.B.C)+(A.B'.C')+(A.B'.C)+(A.B.C')+(A.B.C)

Karnaugh Maps

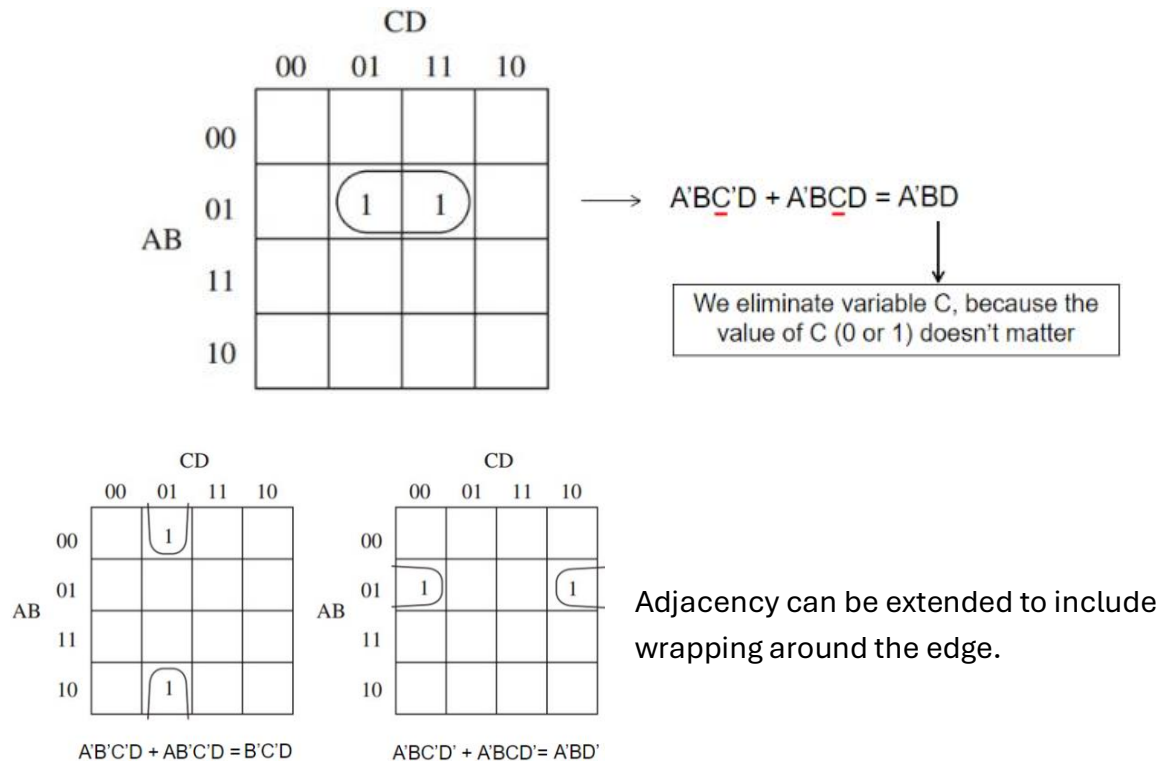
A method of simplifying standard sum of product expressions which **works well for up to four variables**. You write a 1 underneath the combinations which result in a positive formula and a 0 underneath those that don't. The order which you write them in is important!

Remember: This finds the simplest **disjunctive normal form** expression. Sometimes you can simplify it further!

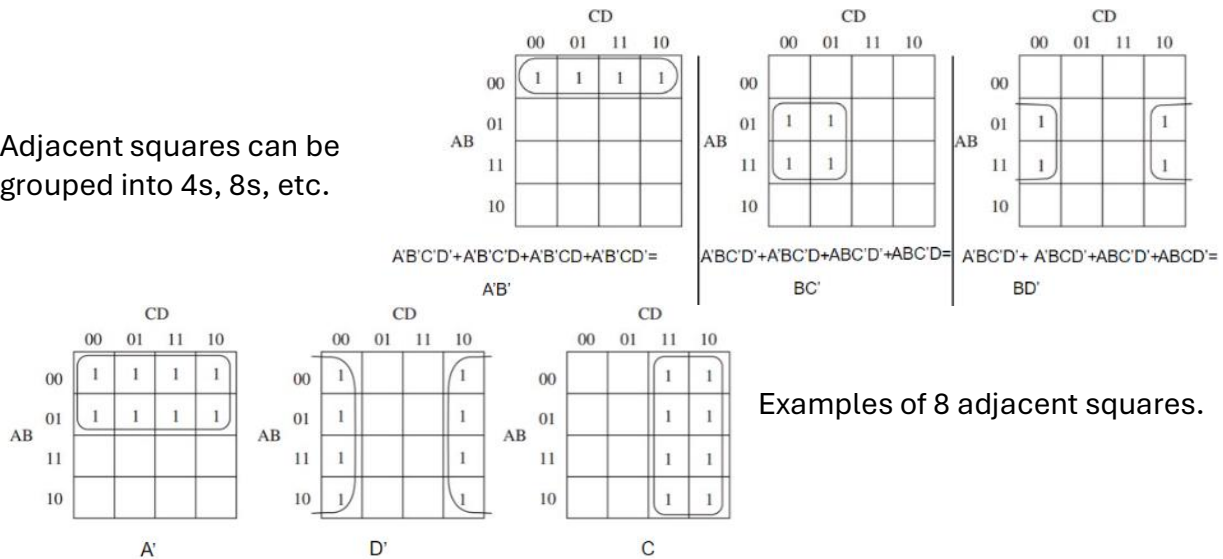
Two Variables	Three Variables	Four Variables
<p>example: $F = AB' + A'B$</p>  <p>(a) $F = A\bar{B} + \bar{A}B$</p>	<p>example: $F = A'BC' + A'BC + ABC'$</p>  <p>(b) $F = \bar{A}B\bar{C} + \bar{A}BC + AB\bar{C}$</p>	 <p>(c) $F = \bar{A}\bar{B}CD + \bar{A}B\bar{C}D + AB\bar{C}\bar{D}$</p>

Eliminating Terms

Any two adjacent squares with 1s differ by one variable. So, we can eliminate that variable.



Adjacent squares can be grouped into 4s, 8s, etc.

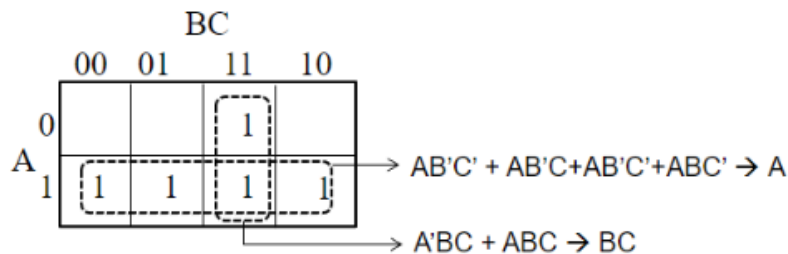


Example

Verify that the Boolean formula $F = A + B.C$ is in its simplest form.

1) We know that the **standard sum of products** form of this function is:
 $F = A.B.C + A.B.C' + A.B'.C + A.B'.C' + A'.B.C$

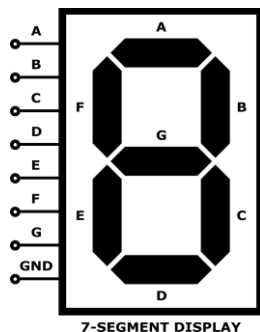
2) The Karnaugh map is:



3) The two areas give us: $F = A + B.C$

The Application of Formula Simplification

Certain combinations of values for variables never occur. Hence, we can consider them as either a 1 or a 0. We can mark these on a Karnaugh map with an **x**.

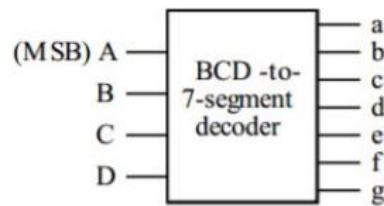


Example – Seven-segment Display

The number is given in **binary coded decimal (BCD)** meaning 4 bits represent one decimal.

- Segments are independent and use their own combinational logic.
- Can represent digits 0-9.

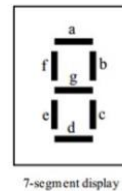
Each BCD sequence is mapped to a certain segment output using a decoder. You don't need to know how this works.



Decimal	Binary	Boolean
0	0000	$A'B'C'D'$
1	0001	$A'B'C'D$
2	0010	$A'B'C'D'$
3	0011	$A'B'C'D$
4	0100	$A'B'C'D'$
5	0101	$A'B'C'D$
6	0110	$A'B'C'D'$
7	0111	$A'B'C'D$
8	1000	$A'B'C'D'$
9	1001	$A'B'C'D$

Hence, any BCD values past 9 are “don't cares” because we don't care about their output.

Inputs				Outputs						
A	B	C	D	a	b	c	d	e	f	g
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	0	0	1	1
1	0	1	0	x	x	x	x	x	x	x
1	0	1	1	x	x	x	x	x	x	x
1	1	0	0	x	x	x	x	x	x	x
1	1	0	1	x	x	x	x	x	x	x
1	1	1	0	x	x	x	x	x	x	x
1	1	1	1	x	x	x	x	x	x	x



7-segment display

Don't
cares

Each of the seven outputs will have a Karnaugh map. We can use simplification to develop a formula for each segment.

Inputs				
A	B	C	D	a
0	0	0	0	1
0	0	0	1	0
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	x
1	0	1	1	x
1	1	0	0	x
1	1	0	1	x
1	1	1	0	x
1	1	1	1	x

Example: For segment (output) 'a'

		CD			
		00	01	11	10
AB	00	1	0	1	1
	01	0	1	1	1
	11	x	x	x	x
	10	1	1	x	x

AB \ CD	CD			
	00	01	11	10
00	1	0	1	1
01	0	1	1	1
11	x	x	x	x
10	1	1	x	x

K-map for segment a

$$a = A + C + BD + B'D'$$

Implementing Arithmetic

We can implement arithmetic operations on binary representations of unsigned & signed integers and floating-point numbers using **combinational Boolean circuits**.



Adders

1-Bit Half Adder

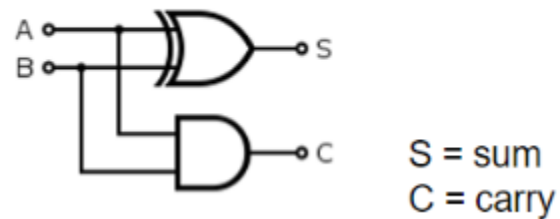
Writing out the different combinations using a truth table helps us to identify logic gates to use.

(a) Single-Bit Addition			
A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

XOR AND

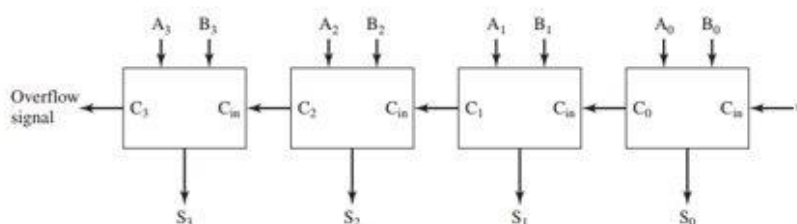
 out  out

1-bit half-adder



Multi-bit Adders / Full Adders

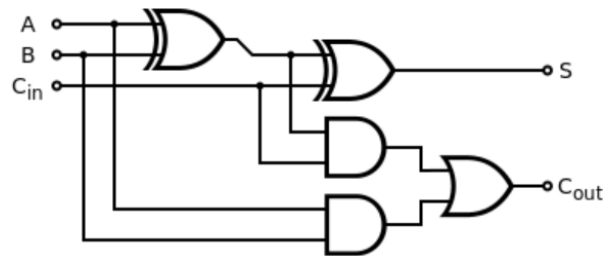
Chaining the carry bit from one adder to another allows us to chain them together. Hence, they each must take three inputs. This **enables us to add multi-bit numbers**. For example, a 4-bit adder would look like:



The output of a full adder is defined by the following formulas which are derived from the truth table.

$$\begin{aligned} \text{Sum} &= A \oplus B \oplus C \\ \text{Carry} &= A.B + C.(A \oplus B) \end{aligned}$$

(b) Addition with Carry Input				
C_{in}	A	B	Sum	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



You don't need to remember this. If asked to draw them, draw the black box-style diagram.

Addition – Guard Bits

A guard bit can be used to **detect overflow on addition operations**. It is added as the **most significant bit** of an addition operation. Hence, arithmetic on $n - bit$ operands is executed internally using $(n + 1) - bit$ operands.

- **Overflow occurs when the MSB and the second MSB are different!**

$$\begin{array}{r}
 00011 = 3 \\
 + 00101 = 5 \\
 \hline
 01000 = \text{overflow}
 \end{array}$$

Notice: the guard bit of the operands is copied from the most significant bit (msb)

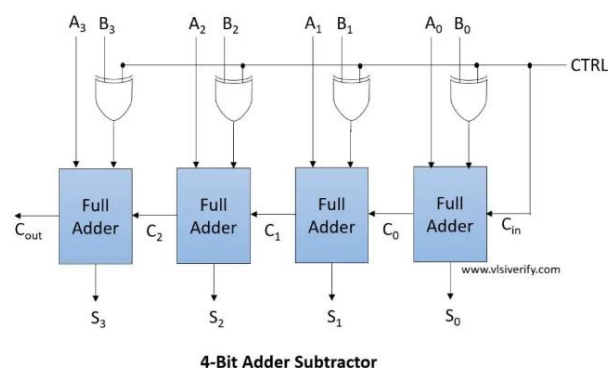
$$\begin{array}{r}
 11001 = -7 \\
 + 11010 = -6 \\
 \hline
 10011 = \text{overflow}
 \end{array}$$

Adder-Subtractors

$A - B = A + (-B)$. Hence, we can carry out subtraction by getting the two's complement of the B input. We can do this by:

- Adding a NOT gate to it, which gets the one's complement.
- Adding 1 to it, set the first carry to 1.

Adding an XOR to the input for B allows us to toggle addition and subtraction operations on and off. We call this circuit an **adder-subtractor**.



4-Bit Adder Subtractor

Multiplication of Positive Integers

We can carry out multiplication by just repeating addition. To carry out long multiplication, we can break it down like this:

$$010111 \times 001011 \text{ (decimal } 23 \times 11)$$

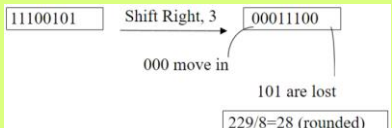
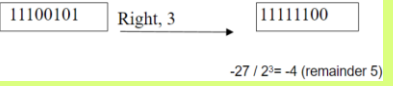
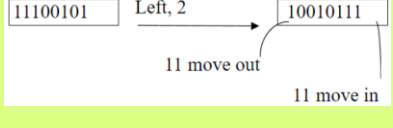
Becomes:

$$\begin{array}{r} 010111 \times 1000 \\ 010111 \times 10 \\ 010111 \times 1 \\ \hline 11111101 \end{array} = \begin{array}{r} 10111000 \\ 101110 \\ 10111 \\ \hline 11111101 \end{array}$$

Multiplying 2 n-bit integers can produce a 2n-bit integer.

Shift Operations

Shifts move bits left or right.

Logical Shifts	Arithmetic Shifts	Circular Shifts (Rotate)
<ul style="list-style-type: none"> Bits moved out are lost. Zeros fill vacated positions. Left shift = multiplication by 2^n. Right shift = division by 2^n. 	<ul style="list-style-type: none"> Like logical but copies of the sign bit fill vacated positions. Retains two's complement sign. Left shift = multiplication by 2^n. Right shift = division by 2^n. 	Bits moved out of one end are moved in the opposite end.
		

Bitwise Logical Operations

You can quickly carry out logical operations on bit patterns by executing them in parallel.

E.g. $A = 0110\ 1010$ An AND operation would generate 01000000 .

$B = 1101\ 0000$

Sequential Logic

Sequential logic systems are **dynamic**, meaning the **output of a logic gate may be fed back to the input of a logic gate earlier in the system**.

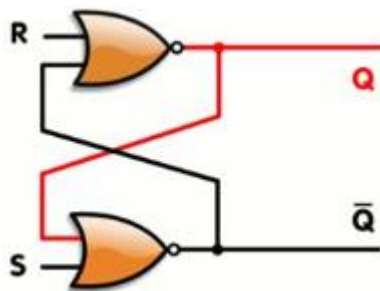
- Sequential systems have a **state** which outputs depend on.

Latches are **level-triggered** and flip-flops are **edge triggered**.

Set-Reset Latch

Different input combinations may result in a stable or unstable state.

- May be non-deterministic.



Functionality

It is comprised of two *NOR* gates connected dynamically.

- **Inputs:** Set & reset.
- **Outputs:** Q and \bar{Q} . We're primarily interested in Q .

- If you make $S = 1$ and $R = 0$, Q will always become 1 regardless of the current state.
- If you make $S = 0$ and $R = 1$, Q will always become 0 regardless of the current state.
- If $S = 1$ and $R = 1$, we get an inconsistent output.

(b) Simplified Characteristic Table		
S	R	Q_{n+1}
0	0	Q_n
0	1	0
1	0	1
1	1	—

Time & Sequence Impact

One gate relies on the other, so must wait for the computation of the first gate. Hence:

- There is a delay before it responds to an input.
- **SR Latches are Asynchronous:** Two separate gates cannot switch simultaneously.

Disadvantages

- **Asynchronous Transitions:** Circuits may have components which depend on the order of changes to inputs. SR latches being asynchronous can mess this up. Can be solved by flip-flops.
- **Certain Inputs Produce Non-Deterministic (Unstable) Outputs:** Can solve by adding gates to control inputs.

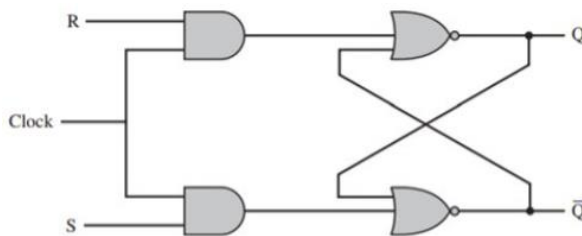
Inverted SR Latches

Can be achieved by swapping *NOR* gates with *NAND* gates. Sets when $S = 0$ and resets when $R = 0$.

Flip-Flops

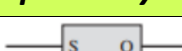
Make use of a **clock** with latches to **synchronise the state transitions across sequential circuits**.

- **Rising Edge:** *Low* \rightarrow *High*
- **Falling Edge:** *High* \rightarrow *Low*



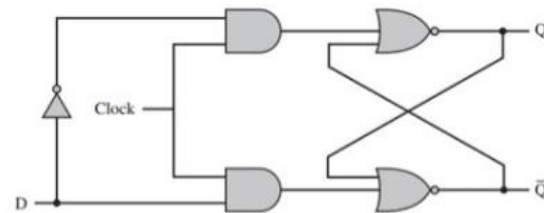
Gated/Clocked Set-Reset Latch

Only responds to the inputs when the clock signal is present.

Graphical Symbol	Truth Table															
	<table><tr><th>S</th><th>R</th><th>Q_{n+1}</th></tr><tr><td>0</td><td>0</td><td>Q_n</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>-</td></tr></table>	S	R	Q_{n+1}	0	0	Q_n	0	1	0	1	0	1	1	1	-
S	R	Q_{n+1}														
0	0	Q_n														
0	1	0														
1	0	1														
1	1	-														

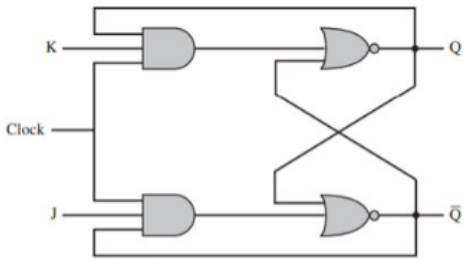
D-Type

An evolution of the SR latch which **prevents an unstable state from occurring**. It does this by ensuring the inputs are always different using a *NOT* gate.



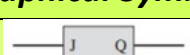
- The D stands for **data** or **delay** because it stores 1 bit of memory.
- It stores the last input.
- Delays any input provided by 1 clock pulse.

Graphical Symbol	Truth Table						
	<table> <tr> <th>D</th><th>Q_{n+1}</th></tr> <tr> <td>0</td><td>0</td></tr> <tr> <td>1</td><td>1</td></tr> </table>	D	Q_{n+1}	0	0	1	1
D	Q_{n+1}						
0	0						
1	1						



J-K Flip-Flop

The restricted combination (1,1) is used to **toggle** the output. It's a **universal flip-flop** meaning it can behave as an SR flip-flop or a D flip-flop.

Graphical Symbol	Truth Table															
	<table><tr><th>J</th><th>K</th><th>Q_{n+1}</th></tr><tr><td>0</td><td>0</td><td>Q_n</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>$\overline{Q_n}$</td></tr></table>	J	K	Q_{n+1}	0	0	Q_n	0	1	0	1	0	1	1	1	$\overline{Q_n}$
J	K	Q_{n+1}														
0	0	Q_n														
0	1	0														
1	0	1														
1	1	$\overline{Q_n}$														

Application of Flip-flops

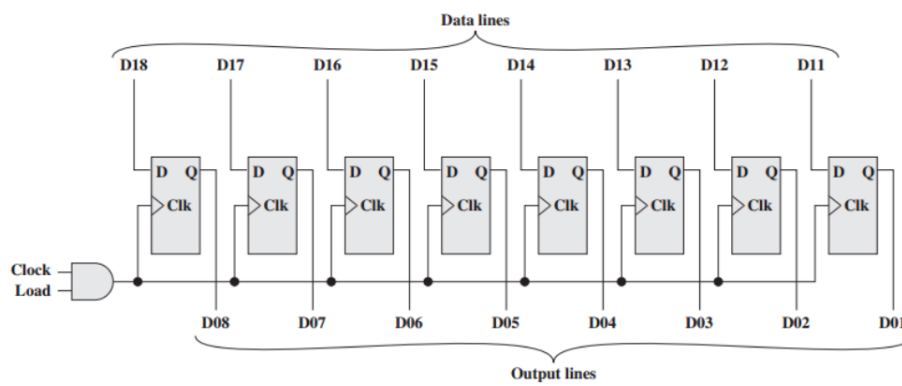
Registers

Registers are data stores on the CPU used to store one or more bits of data.

Parallel Registers

- Comprised of D-type flip flops which can be read from or written to simultaneously.
- Used to store data.
- Has a clock to synchronise writing.

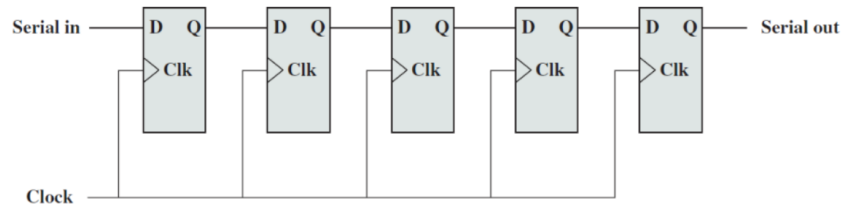
For example, an 8-bit parallel register would look like:



Shift Registers

- Used to transfer information serially.
- With each clock pulse, data is shifted right or left one position.
- Data is transferred out when it meets the bit at the end.

For example, a 5-bit shift register would look like:



Note: Connecting out with in makes a circular shift register.

Counters

- A register which can be incremented by 1.
- When the maximum is reached, the next increment sets the counter to 0.
- A counter of n flip-flops can count to $2^n - 1$.

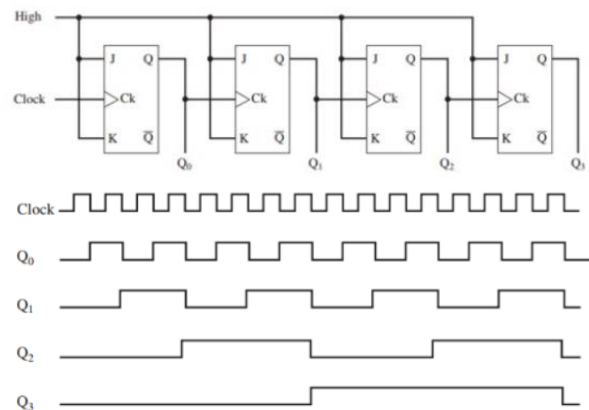
E.g. a program counter.

Ripple/Asynchronous Counter

The change begins at one end and *ripples* through to the other end.

- Asynchronous because the **first flip-flop is the only one which uses the actual clock pulse**.
- J and K are kept at a **constant high (1)**.
- The **delay is proportional to the length of the counter**.
- Edge-triggered means that the state changes with the falling edge of the clock pulse!

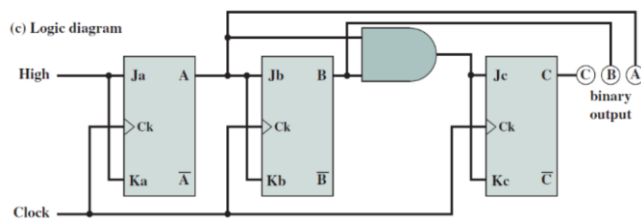
Q_1	Q_2	Q_3	Q_4
0	0	0	0
0	0	0	1
0	0	1	0
0	0	1	1
0	1	0	0
0	1	0	1
0	1	1	0
0	1	1	1



This ripple counter is edge-triggered.

Synchronous Counter

Unlike asynchronous counters, synchronous counters change all flip-flops at the same time.



J	K	Q_{n+1}
0	0	Q_n
0	1	0
1	0	1
1	1	$\overline{Q_n}$

Pointers & Memory Allocation

A pointer in C is a data item whose:

- **Value** is an **address** stored as an integer.
- **Type** describes the type of data located at that memory address.
- They enable us to pass data to functions by-reference.

We can use pointers to inspect or modify the location which they point to. We do this in C by **dereferencing the pointer**.

*i.e. if we have a pointer "a" then "*a" is the value within this pointer.*

Pass-by-Value

C always uses pass-by-value. When passing-by-value to a function:

- The value of the variable is passed.
- A copy of this value is created at a different memory address.
- Any modifications will be applied to this copy.
- When the function returns, the copy will pop meaning that the old value remains.

Pass-by-Reference

When passing-by-reference to a function:

- The address of the variable is passed.
- Any modifications will be applied to the real value.
- When the function returns, the original value will have changed.

This can be achieved in C **using pointers**. You must:

- Pass the address of the variables to the function.
- Initialise these addresses as pointers, meaning that C interprets them as the contents of these addresses.

Use of Pointers in C

Initialising Pointers

You must state:

- The **data type within the address** which the pointer is referring to.
- The **name** of the pointer.
- An **asterisk (*)** before the name to indicate that it is a pointer.
- The **address of the data** which it is pointing to.

E.g. `int *p = &n;`

```
int n = 10;
printf(&n);
>> 0x7ffee7d5f508
```

Referencing Memory Addresses

Using an ampersand before a variable gives the memory address of this variable. *E.g.*

Referencing Pointers

Referencing a pointer without the asterisk is referencing the address of its contents. *E.g.*

```
int n = 10;
int *p = &n;
printf(p);
>> 0x7ffee7d5f508

int n = 10;
int *p = &n;
printf(*p);
>> 10
```

Referencing a pointer with the asterisk is referencing its contents. Basically saying “**the value within this address**”. *E.g.*

```
int n = 10;
int *p = &n;
*p = 20;
printf(n);
>> 20
```

Indirectly Updating a Value

You can indirectly update a value by referring to the pointer.

Changing the Address of a Pointer

You can change the address of a pointer. Assuming an address is 4 bytes, the following operation would increase the pointer by 4 bytes. Hence, it would be pointing to a random piece of memory.

```
int a; // Declares an integer.
int *b; // Declares a pointer to an integer.
int **c; // Declares a pointer to a pointer to an integer.
```

```
int n = 10;
int *p = &n;
p = p + 1;
printf(*p)
>> Unknown
```

Pointers Pointing to Pointers

Pointers can point to pointers!

Arrays with Pointers

The array variable is **a pointer to the start of the array** which is stored in its own location in memory. So, when you create an array in C it:

- Allocates memory for each index of the array.
- Creates a pointer which points to the first index of the array.

Given this:

$$p[i] = *(p + i)$$

Structures

A structure in C is **a collection of one or more variables** of possibly different types under one identifier. You can refer to a variable in a struct which is a pointer using this special syntax:

$$(*s).member \equiv s \rightarrow member$$

Memory Allocation

Libraries in C allow us to dynamically manage memory. You don't need to know these specific functions for the exam. However, it is important to know that **whenever you**

allocate memory you must deallocate it to free it again otherwise it will continue to exist until the application quits.

Stack Variables

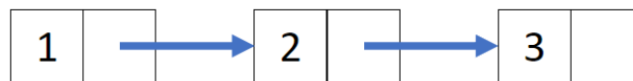
Once a stack variable falls out of scope, it is popped from the stack and new data is pushed to the stack meaning **the memory is reallocated to variables in other functions**.

Heap Variables

Heap variables are not automatically deallocated. Allocating memory space in C creates a pointer stored on the heap. Hence, this pointer and the corresponding memory space must be deallocated.

Linked Lists

A linked list consists of nodes where each node contains a **value** and a **pointer** to the next element of the list. They can be implemented in C using structures.



Creating a Node Create a node with a value and a null pointer.	Adding to the End of a Linked List Change the pointer at the end to point to the node being added.
Printing Print the value of the node and follow the pointer. Repeat until the pointer is null.	Adding After a Certain Position Loop through until it reaches the correct position or the end of the list. If it reaches the correct position, change the pointer before to point to it and change its pointer to the one after.
Removing a Node in a Certain Position Switch the pointer of the node before the point to the node after. Then deallocate the memory space given to the node being removed.	

Applications of Linked Lists

- **Browsers & Word Processors:** Each action creates a new node in a linked list allowing users to undo.
- **Navigation System:** Routes between locations.