Learning aims

- Describe the differences between, and uses of, CISC and RISC processors

Keyword	Simplified definition
RISC (Reduced Instruction Set Computer)	CPU design with a **small set of simple instructions**. Each instruction is designed to execute very quickly (often one clock cycle). Relies on efficient use of registers.
CISC (Complex Instruction Set Computer)	CPU design with a **large set of complex instructions**, some of which can execute multi-step operations. Each instruction may take multiple cycles but can reduce the number of instructions per program.
Machine code	The **binary code (0s and 1s)** that a CPU directly understands and executes.
Assembly language	A low-level programming language that uses **mnemonics** instead of binary machine code, e.g. ADD, MOV.
Mnemonics	Short, human-readable codes used in assembly to represent machine instructions, e.g. SUB = subtract.
Instruction Set Architecture (ISA)	The complete set of instructions that a CPU can execute. Different for RISC and CISC designs.
Opcode	The part of a machine code instruction that specifies the operation (e.g., ADD, LOAD).
Operand	The part of a machine code instruction that specifies the data or memory location to be used.
Instruction length	RISC instructions are usually fixed length (simpler to decode); CISC instructions vary in length.
Clock Cycles	RISC aims for 1 instruction per cycle, CISC often takes multiple cycles per instruction.
Pipelining	Easier to implement in RISC because instructions are simple and uniform in size.
Compiler optimisation	RISC relies heavily on the compiler to break complex tasks into simple instructions efficiently.

A history lesson …

The first ever programs were written solely in **machine code** (0s and 1s).

Each instruction was entered by hand before being **executed**.

Operation code (**opcode**) tables were used to help with this, but it was still very time consuming.

To speed things up, programmers developed an **assembly language**.

This made the **opcodes** easier to read by using a **mnemonic** and an **operand** to form an instruction.

Each line of **assembly language** was equivalent to one line of **machine code**.

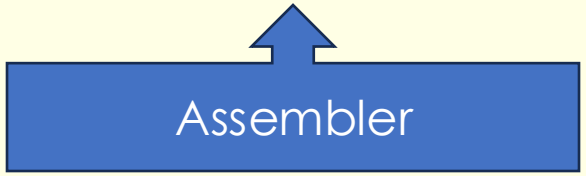
Assemblers were created to automatically translate the assembly language into machine code.

```
10001011010101011111100

10001011010001011111000

111010000

10001001010001011110100
```




```
LDA R1, A
LDA R2, B
MULT R1, R2
STO R1
A
```

Low-level languages

Assembly language and machine code are **low-level languages**.

Each line of assembly language is equivalent to one line of machine code.

The code is **specific** to the **CPU** that it is written for.

Assembly Language

LDA R1, A

Machine Code

```
10001011010101011111100
```

RISC

RISC, like what we have seen in Little man computer is a way to program a RISC architecture.

Using simple instructions, each taking one clock cycle, can be executed.

Thus the multiplication instruction **a = a * b.** might be written:

LDA R1, A
LDA R2, B
MULT R1, R2
STO R1
A

Mnemonic	Action
LDA	Loads a value from a memory address
STA	Stores a value in a memory address
ADD	Adds the value held in a memory address to the value held in the accumulator
SUB	Subtracts from the accumulator the value held in a memory address
MOV	Moves the contents of one memory address to another

Reduced Instruction Set Computers (RISC)

The opposite approach is adopted in the more modern RISC architecture.

Only simple, smaller instructions, each taking one clock cycle, can be executed.

Thus the multiplication instruction **a = a * b.** might be written:

LDA R1, A
LDA R2, B
MULT R1, R2
STO R1
A

Complex Instruction Set Computers (CISC)

Intel processors, for example use a CISC architecture to use with modern computers.

CISC uses a **large** instruction set, used to accomplish tasks in as **few lines** of assembly language as possible.

The processor hardware is capable of understanding and executing the **series of sub-tasks** that make up a **single instruction.**

Complex instructions are built into the **machine's hardware.**

CISC Example

To multiply two values held in different memory locations A and B, storing the result in A, a processor using several general purpose registers would load each of the values into a separate register, carry out the multiplication and then store the result back in A.

The assembly language instruction for a CISC processor might be written something like

MULT A, B

A CISC processor has in its instruction set a single instruction that will do the loading, multiplication and storing of the result.

The instruction is equivalent to the high level instruction:

a = a * b

Comparison

Example:
Multiply value in memory location "X" by value in memory location "Y"; store result back into location "X". Registers "A" and "B" are available.

CISC Assembly:

```
IMUL X, Y
```

RISC Assembly:

```
LOAD A, X
LOAD B, Y
PROD A, B
STORE X, A
```

Comparison

RISC	**CISC**
Smaller instruction set	Larger instruction set
Requires less complex hardware/ requires little cooling, minimising manufacture cost/ less transistors	Requires more complex hardware/ requires cooling, more expensive to manufacture/ more transistors
One clock cycle to execute an instruction	Multiple clock cycles to execute an instruction
Tends to use less energy	Tends to use more energy
Uses more RAM	Uses less RAM
Easier to pipeline	Difficult to pipeline
But compiler has to do more work to translate the code into machine code	Compiler has to do less work to translate the code into machine code
Fewer addressing modes (drawback)	More addressing modes
Applications requiring **high-speed processing** and **efficiency**, such as embedded systems and mobile devices, due to its **faster, simpler** instructions.	Preferred in general-purpose computers where **ease of programming** are important, as complex instructions can make the software simpler.

RISC back in fashion!

Mobile devices and Apple ARM processors are using RISC architectures instead of CISC

Why?

Easier to design (less transistors)
Smaller in Design
More power efficient
Cheaper to produce

But compiler has to do more work to translate the code into machine code