Do now! Find definitions for the following terms

Machine code	Assembly code	Mnemonics	Assembler
High level language	Translator	Complier	Interpreter

Learning Aims

- Describe the difference between **low level** and **high level programming languages**
- Understand the role of an **assembler**, **compiler** and **interpreter**
- Explain the difference between compilation and interpretation, and describe situations when both would be appropriate
- Explain why an **intermediate language** such as bytecode is produced as the final output by some compilers and how it is subsequently used

Assembly language	A language that replaces machine code with mnemonics and operands to make them easier to read/write.
Assembler	An assembler translates assembly language into machine code.
Compiler	A compiler creates an executable file for a program by translating a high-level language to machine-readable code.
Execute	To carry out the instructions for a computer program.
High-level language	A human-readable language written in formal, structured English.
Interpreter	An interpreter translates and executes code line by line. It translates the code into machine-readable code.
Low-level language	Quickly executed by a computer, written in either machine code or assembly.
Machine code	A program written using 1s and 0s. A computer can execute this directly.
Mnemonic	A code to help us remember something.
Operand	A piece of data that can be changed.
Translator	Executes the programs that programmers write in high-level languages.

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.

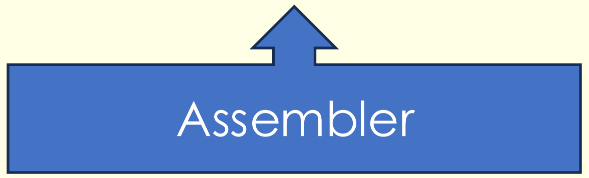
To speed things up, programmers developed an **assembly language**.

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.

```
100010110101010111111100

100010110100010111111000

111010000

100010010100010111110100
```



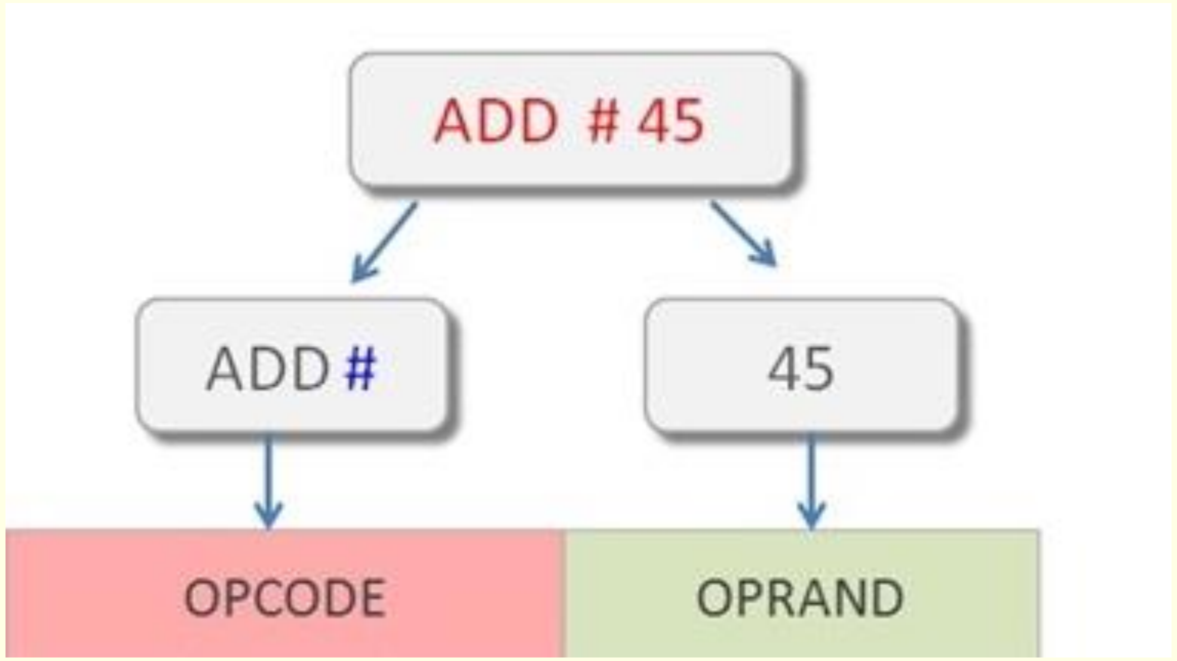
```
mov edx,DWORD PTR [rbp-0x4]

mov eax,DWORD PTR [rbp-0x8]

add eax,edx

mov DWORD PTR [rbp-0xc],eax
```

Opcode and Operand

The instruction is split into an **Opcode** and an **Operand**

Opcode is which operation to carry out.

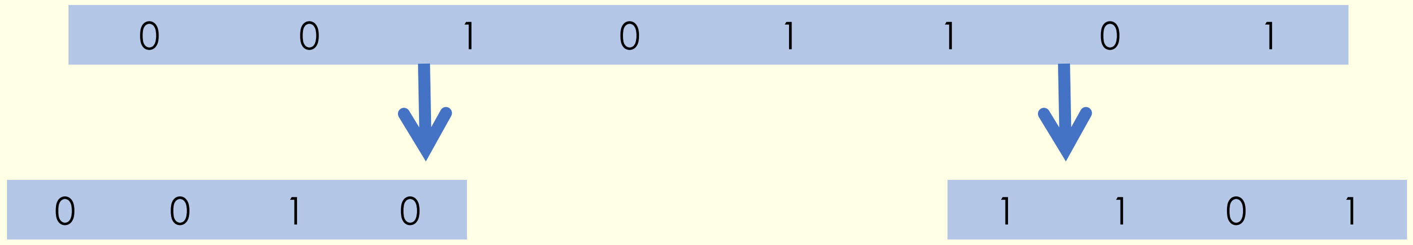
The **operand** specifies the data that needs to be acted on.



Opcode and operand

In a simple 8-bit instruction

1001 represents the instruction to add the value found in a memory location to the accumulator. If the following instruction is fetched:



Operator

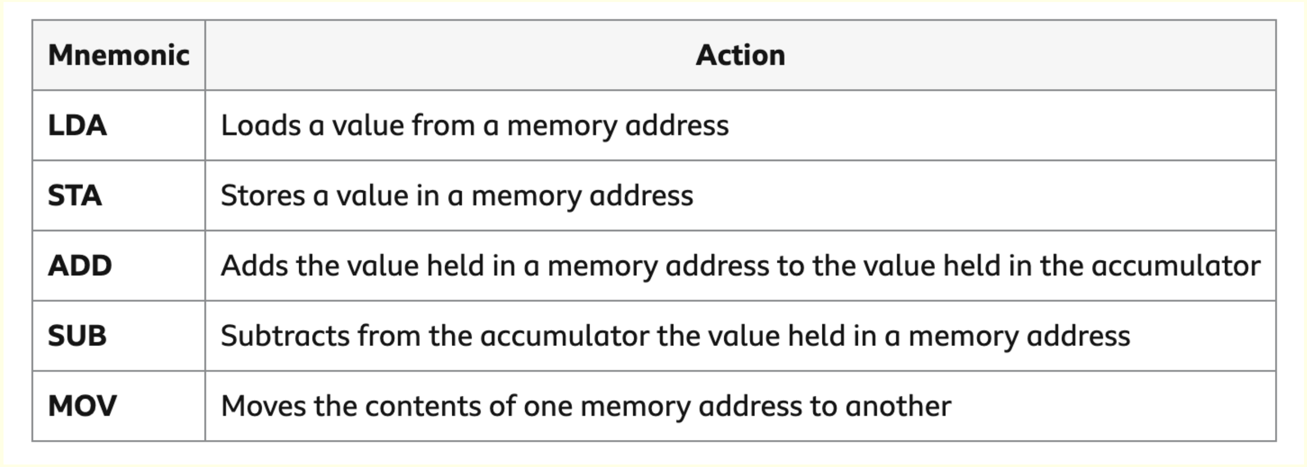
Add to the accumulator

Operand

The value found in memory location 1101

What is assembly language?

- Assembly language is a low-level language.
- They are microprocessor specific so they can **manipulate the hardware directly.**
- It is a bit easier to work with than machine code, because each instruction is written as a short, keyword called a **mnemonic.**
- Each mnemonic directly corresponds with a single machine code instruction.



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

- Mnemonics are much easier to understand and debug than machine code, giving programmers a simpler way of directly controlling a computer.
- But you have to decide and manage where data is stored in memory.
- Debugging can be more difficult than in high level languages.

Assembly code

Each instruction in assembly code equivalent to one machine code instruction. (**one-to-one conversion to machine code**)

MOV AX, 5 ; Load the value 5 into register AX

10111000 00000101 00000000

This demonstrates **one-to-one correspondence**, meaning each assembly instruction directly translates into a specific machine code instruction.

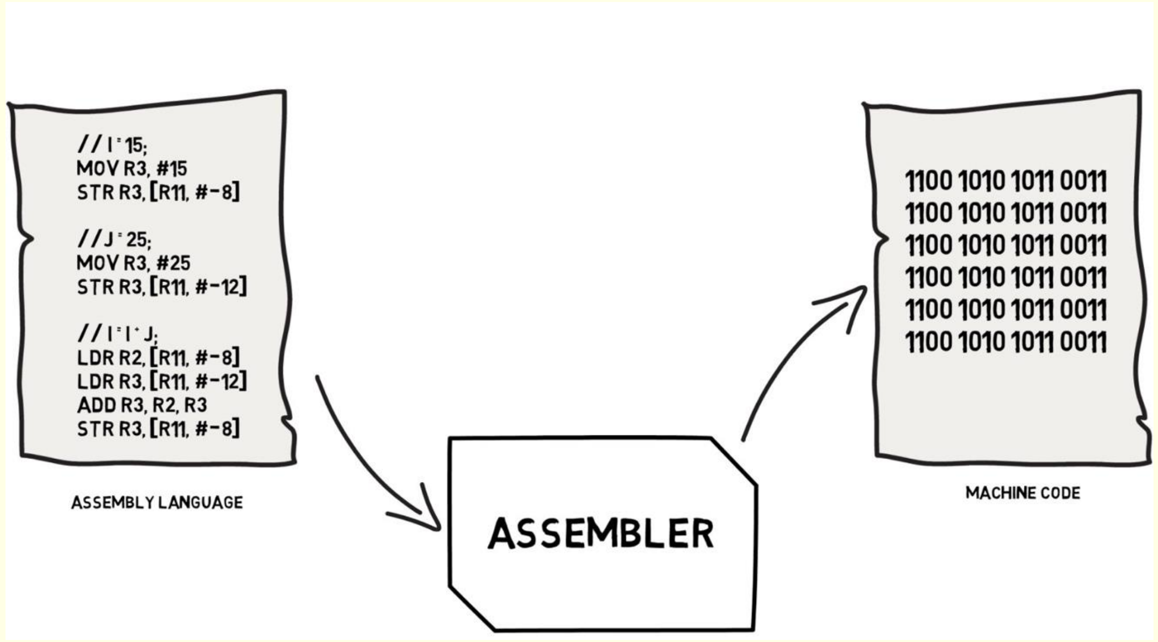
The machine code instructions that a particular computer can execute (the instruction set) are completely **dependent** on its hardware.

Each different type of processor will have a different instruction set and a different assembly code.

An **Assembler** to needed to translate assembly code into machine code

Assembler replaces each **mnemonic** with the appropriate binary machine code

•



```
//I=15;
MOV R3, #15
STR R3, [R11, #-8]

//J=25;
MOV R3, #25
STR R3, [R11, #-12]

//I=I+J;
LDR R2, [R11, #-8]
LDR R3, [R11, #-12]
ADD R3, R2, R3
STR R3, [R11, #-8]
```

```
1100 1010 1011 0011
1100 1010 1011 0011
1100 1010 1011 0011
1100 1010 1011 0011
1100 1010 1011 0011
1100 1010 1011 0011
```

The input to the assembler is called the **source code** and the output (machine code) the **object code.**

How an assembler work

- Reserves storage for instructions & data
- Replaces mnemonic opcodes by machine codes
- Replaces symbolic addresses by numeric addresses, for example When the assembler processes the code, it replaces the symbolic labels (like DATA) with actual memory addresses.
- Creates symbol table - is a data structure created by the assembler to map **symbolic labels** to **numeric memory addresses**.
- Checks syntax
- Error diagnostics

```
        INP       ; Read input into the accumulator
        ADD DATA  ; Add the value stored at DATA
        STA DATA  ; Store the result back in DATA
        HLT       ; Halt execution

DATA    DAT 10    ; Store initial value 10
```

```
        INP       ; Read input into the accumulator
       ADD 00010000; Add the value stored at 00010000
       STA 00010000; Store the result back in 00010000
       HLT        ; Halt execution
```

Symbol table

Symbol	**Address**
DATA	00010000

Why use assembly code?

- Quicker/more efficient to translate
- Makes more efficient use of the CPU // memory // system resources // where a system may have limited resources
- The programmer wants direct control over hardware/memory // to access machine specific functionality
- Code might be written for a specific architecture
- Compilers/interpreters may not be available

- To write software for **embedded systems** because there is no need for code to be used on other architecture and it keeps memory usage to a minimum.

- To write software for device drivers. **Device Drivers** are loaded into memory by the Operating System and used to control the operation of a Hardware Device e.g. Graphics Card Drivers, Printer Drivers.

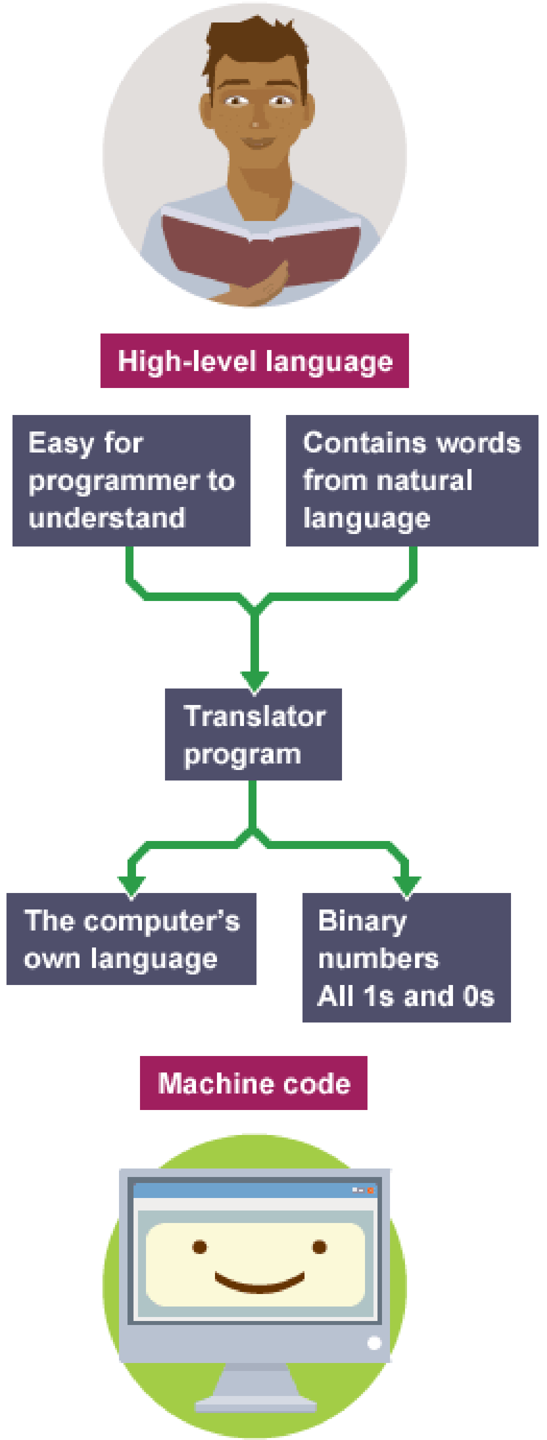
Drawbacks of Assembly Languages

- A very limited range of instructions is available. Every task, even the simplest, has to be built up from the smallest steps.
- You have to decide and manage where data is stored in memory
- Debugging can be more difficult than in high level languages.

What are high level languages?

- High level Language is easier to learn because it uses **English-like statements** which can be read by programmers.
- Example: Python uses 'print', ‘if’, 'input' and 'while' statements - all words from the English language - to form instructions.
- Faster to write programs
- Easier to debug programs
- Many types of high-level language exist and are in common use today, including: Python, Java, C++, C#, Visual Basic.NET, PHP and Ruby

- Translated using:
  – Compiler
  – Interpreter


High-level language
Easy for programmer to understand
Contains words from natural language
Translator program
The computer's own language
Binary numbers All 1s and 0s
Machine code

HLL and One to many

Python code

```
a = 5 + 10
```

$\longrightarrow$

Resulting Assembly Code

```
1. MOV R1, #5    ; Move constant 5 into register R1
2. MOV R2, #10   ; Move constant 10 into register R2
3. ADD R3, R1, R2 ; Add R1 and R2, store result in R3
4. MOV [a], R3   ; Store the result from R3 into the
memory location of a
```

One line of Python code results in **many lines of machine code** due to the steps necessary to perform the operation in hardware.

Comparison of types of programming languages

High -Level	**Low-level**
It uses **English-like statements** which can be easily read by programmers. Designed for quick programming. Can be translated for **multiple machine architectures.** Need to rely on compiler to optimise the code. HLL may produce multiple lines of machine code per line of code // one-to-many	Microprocessor/CPU/**Machine specific** and can **control the hardware directly.** Can be highly optimised to make **efficient use** of the hardware and execute more quickly. Each line of code is one instruction only Hard to read and learn. Only works for one type of machine architecture.

Translators for high level languages

Language translators are used to translate a language into machine code.

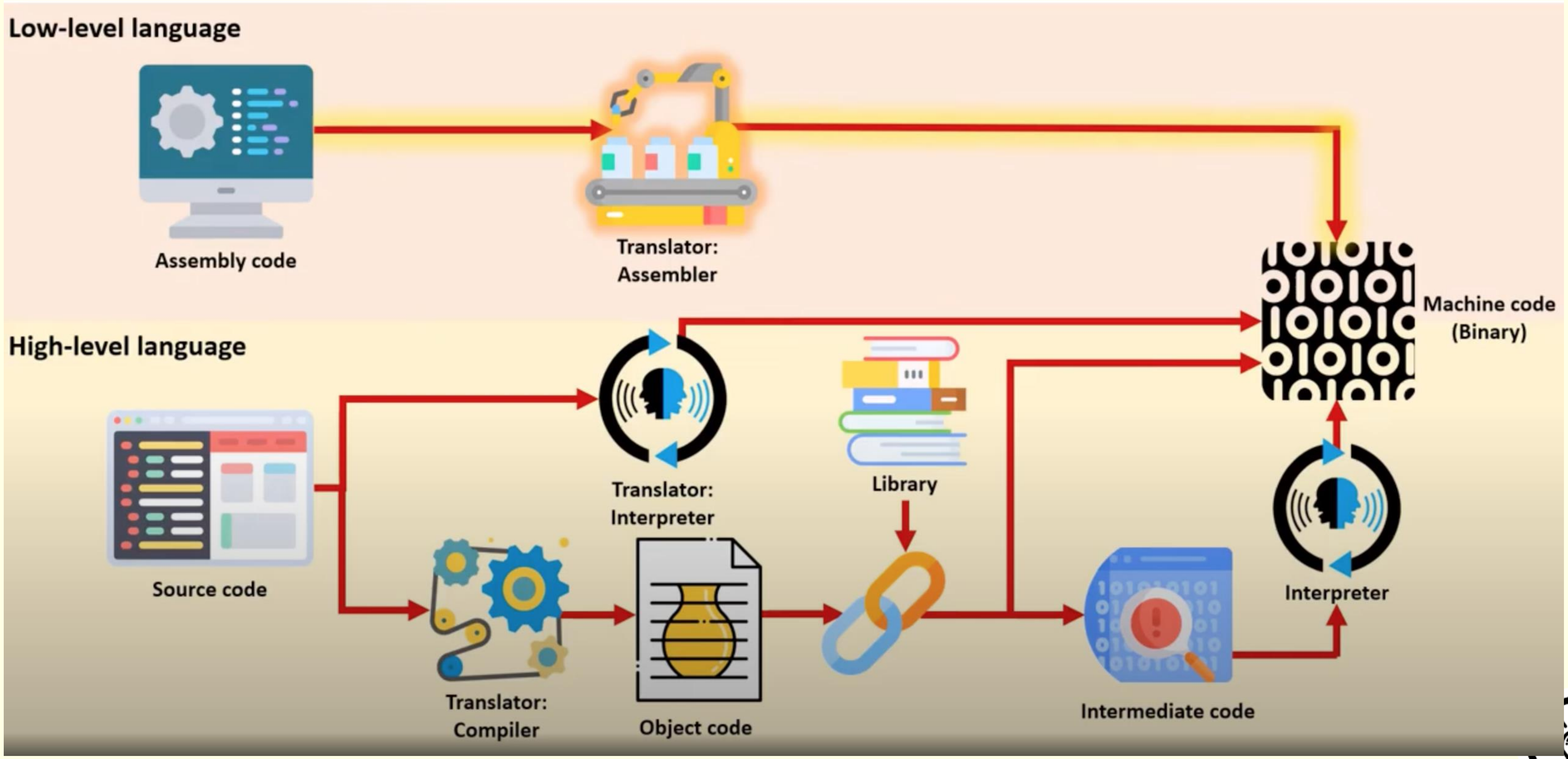
Complier

Translate **entire** source code all in one go into Machine Code and creates executable file (exe file) known as the **object code**

Interpreter

Translate and execute source code **Line by Line**

Processes associated with language translation



Intermediate code

Intermediate code is a **bridge** between high-level programming languages and machine code.

Instead of directly converting source code into machine code, a compiler first translates it into an **intermediate representation**.

Key Points:

- It is **not machine code** but **not human-readable** either so protects the source code from being copied
- It allows **portability**—the same code can run on different platforms.
- It is executed by a **virtual machine** (e.g., Java Virtual Machine for Java bytecode).
- It improves **efficiency**, as some optimisations can happen before final execution.

Examples of Intermediate Code:

- **Java Bytecode** – Runs on the **JVM** instead of directly on a CPU.
- **Python Bytecode (.pyc files)** – Executed by the **Python Virtual Machine**.
- **C Intermediate Representation (IR)** – Used for compiler optimisations before machine code is generated.

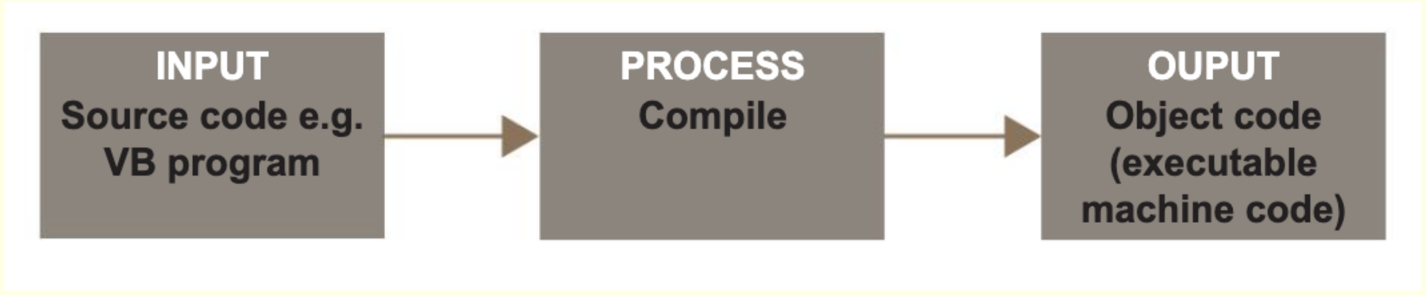
Compiler

The code written by the programmer, the **source code**, is input as data to the compiler, which scans through it several times, each time performing different checks and building up **tables of information** needed to produce the final **object code**.

Different hardware platforms will require different compilers, since the resulting object code will be hardware-specific.

For example, Windows and the Intel microprocessors comprise one platform, Apple and PowerPC processors another, so separate compilers are required for each.

The object code can then be saved and run whenever needed without the presence of the compiler.


INPUT
Source code e.g. VB program
PROCESS
Compile
OUPUT
Object code (executable machine code)

Interpreter

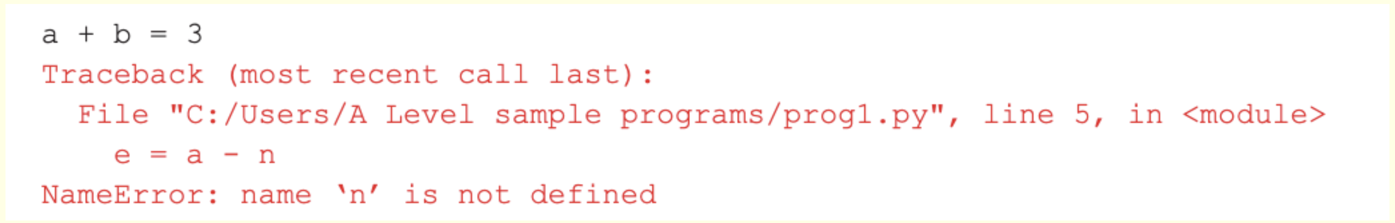
An interpreter is a different type of programming language translator.

Once the programmer has written and saved a program, and instructs the computer to run it, the interpreter looks at each line of the source program, analyses it and, if it contains no syntax errors, translates it into machine code and runs it.

For example, the following Python program contains an error at line 5.

```
1 a = 1
2 b = 2
3 c = a + b
4 print("a + b = ", c)
5 e = a - n
6 print("a - b = ", e)
7 print("goodbye")
```

When the program runs, it produces the following output:



```
a + b = 3
Traceback (most recent call last):
  File "C:/Users/A Level sample programs/prog1.py", line 5, in <module>
    e = a - n
NameError: name 'n' is not defined
```

Translators: Assemblers, compilers and interpreters

	Assembler	Compiler	Interpreter
Description	• Translates assembly language into machine code. • Takes basic commands and operations from assembly code and converts them into binary code that can be recognised by a specific type of processor. • The translation process is typically a one-to-one process from assembly code to machine code.	• Translates source code from high-level languages into object code and then machine code to be processed by the CPU. • The whole program is translated into machine code before it is run.	• Translates source code from high-level languages into machine code, ready to be processed by the CPU. • The program is translated line by line as the program is running.
Advantages	• Programs written in machine language can be replaced with mnemonics, which are easier to remember. • Memory-efficient. • Speed of execution is faster. • Hardware-oriented. • Requires fewer instructions to accomplish the same result.	• No need for translation at run-time. • Speed of execution is faster. • Code is usually optimised. • Original source code is kept secret.	• Easy to write source code, as the program will always run, stopping when it finds a syntax error. • Code does not need to be recompiled when code is changed. • It is easy to try out commands when the program has paused after finding an error – this makes interpreted languages very easy for beginner programmers to learn to write code.
Disadvantages	• Long programs written in such languages cannot be executed on small computers. • It takes lot of time to code or write the program, as it is more complex in nature. • Difficult to remember the syntax. • Lack of portability between computers of different makes.	• Source code is easier to write in a high-level language, but the program will not run with syntax errors, which can make it more difficult to write the code. • Code needs to be recompiled when the code is changed. • Designed for a specific type of processor.	• Translation software is required at run-time. • Speed of execution is slower. • Code is not optimised. • Source code is required.

Comparison of Compilers and Interpreters

Compilers	**Interpreters**
Translate **entire source** code all in one go into Machine Code and creates an executable file (exe file)	Translate and execute source code **Line by Line**
Object code is faster processing	Slower processing
Reports all errors at the end	Reports errors as they occur
Whole program must be loaded into memory	Memory is only needed as each line of code is run
Used at the end of development (ready for shipping)	Used for development (aide debugging)
Source code hidden so it protects the code from being copied.	Access source code
Machine dependant only run on a computer with a particular machine architecture.	Platform independent as executes the source code just needs a suitable interpreter

Recap

High-level languages

High-level programming languages are close to natural language spoken and written by humans, such as python.

Low-level languages

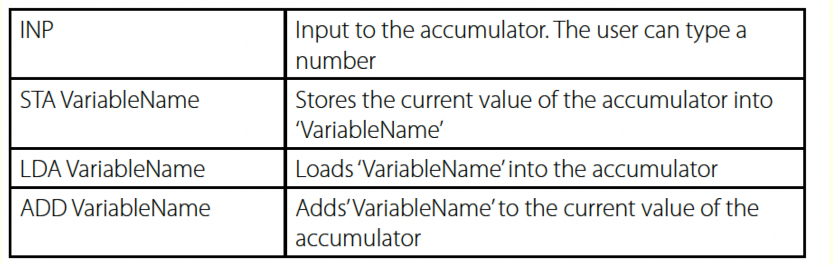
- *Low-level languages* are languages that sit close to the computer's *instruction set*. An instruction set is the set of instructions that the processor understands.
- Two types of low-level language are:
  - *machine code*
  - *assembly language*

Machine code

- Machine code is the set of instructions that a *CPU* understands directly and can act upon. A *program* written in machine code would consist of only 0s and 1s - *binary*. This is very difficult to write and *debug*. Even a very simple program could have thousands of 0s and 1s in it.

Assembly language

- Assembly language sits between machine code and *high-level language* in terms of ease of use. While high-level languages use *statements* to form instructions, assembly language uses *mnemonics* - short abbreviations. Each mnemonic directly corresponds with a machine code instruction. Here are some examples of mnemonics:

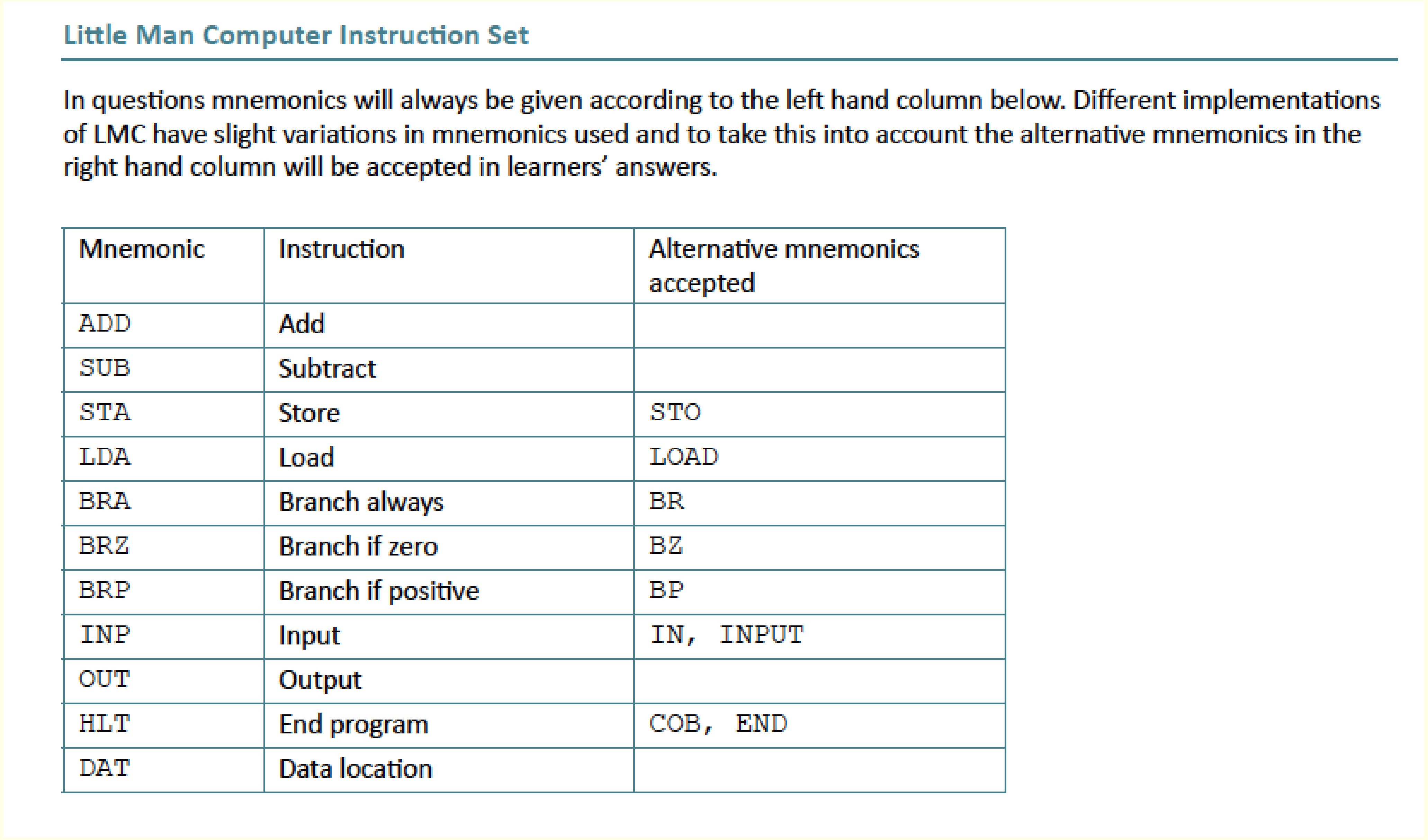


INP	Input to the accumulator. The user can type a number
STA VariableName	Stores the current value of the accumulator into 'VariableName'
LDA VariableName	Loads 'VariableName' into the accumulator
ADD VariableName	Adds'VariableName' to the current value of the accumulator

Learning Aims

- Know how a CPU processes instructions and data
- Be able to write and follow simple assembly language programs
- Be able to explain the role of specific CPU components

OCR



Little Man Computer Instruction Set

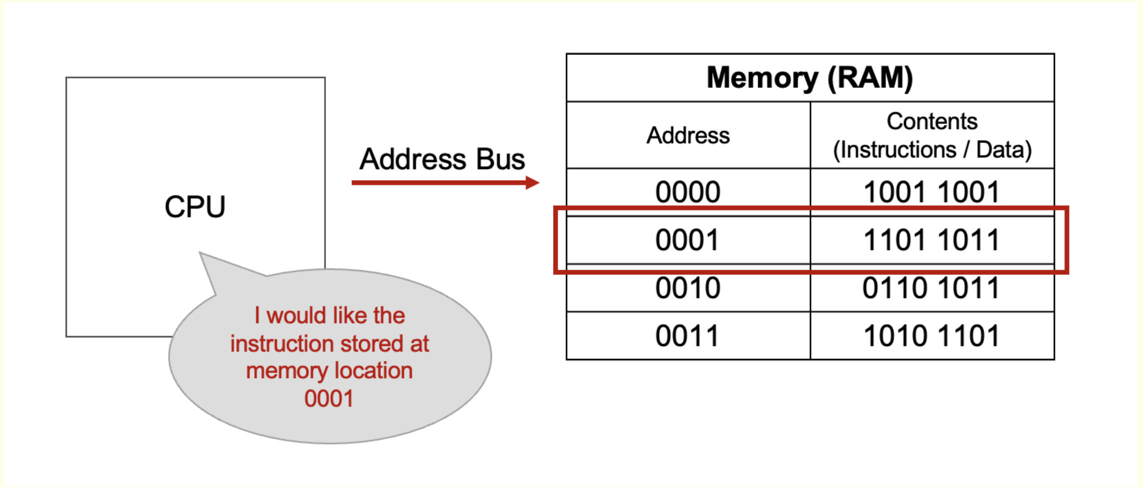
In questions mnemonics will always be given according to the left hand column below. Different implementations of LMC have slight variations in mnemonics used and to take this into account the alternative mnemonics in the right hand column will be accepted in learners' answers.

Mnemonic	Instruction	Alternative mnemonics accepted
`ADD`	Add	
`SUB`	Subtract	
`STA`	Store	`STO`
`LDA`	Load	`LOAD`
`BRA`	Branch always	`BR`
`BRZ`	Branch if zero	`BZ`
`BRP`	Branch if positive	`BP`
`INP`	Input	`IN, INPUT`
`OUT`	Output	
`HLT`	End program	`COB, END`
`DAT`	Data location	

Instructions and data

Instructions and data are stored in memory as binary.

When we write a program in a **high level language**, like python or use a **low level language**, known as assembly code, the programs need to be **translated** into binary.



Memory (RAM)	
Address	Contents (Instructions / Data)
0000	1001 1001
0001	1101 1011
0010	0110 1011
0011	1010 1101

Each processor architecture has a set of instructions that it can run.

This is known as the instruction set.

INPUT OUTPUT LOAD STORE ADD SUB HALT

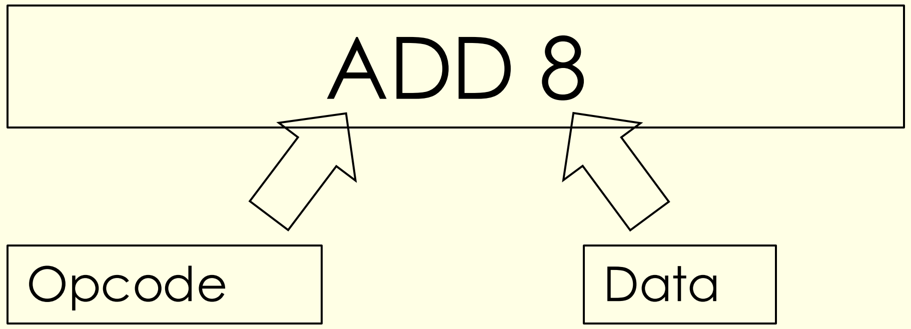
Task 3 Assembly code

- Assembly code uses text-based mnemonics to represent machine code instructions.
- It is considered a low-level language because it provides direct control over hardware.
- Requiring programmers to specify exact memory locations and instruction codes.

```
INP           Input first number
STA 10        Store it in memory location 10
INP           Input second number
ADD 10        Add the first number stored in location 10
OUT           Output the result
HLT           Halt the program
```

Opcodes and data

Instructions have two parts to them.



The opcode tells the CPU what to do. ADD tells the CPU to add two numbers together.

The data part of the instructions tells the CPU what to add. In this case 8.

The question is what do we add to? How does it work! What is this opcode? All shall be revealed!

Instruction Set

Each instruction corresponds to a **three-digit** machine code where XX is the memory address.

Mnemonic	**Opcode**	**Description**
INP	901	Input a value from the user into the accumulator
OUT	902	Output the value from the accumulator
LDA X	5XX	Load the value from memory location X into the accumulator
STA X	3XX	Store the value in the accumulator into memory location X
ADD X	1XX	Add the value at memory location X to the accumulator
SUB X	2XX	Subtract the value at memory location X from the accumulator
BR X	6XX	Unconditional branch (jump) to memory location X
BRZ X	7XX	Branch to memory location X if the accumulator is 0
BRP X	8XX	Branch to memory location X if the accumulator is 0 or positive
HLT	000	Halt the program (stop execution)
DAT	(None)	Define a data storage location (used for variables)

How to store a variable in LMC

In **Little Man Computer (LMC)**, variables are stored using the **DAT instruction**, which reserves a memory location for a value.

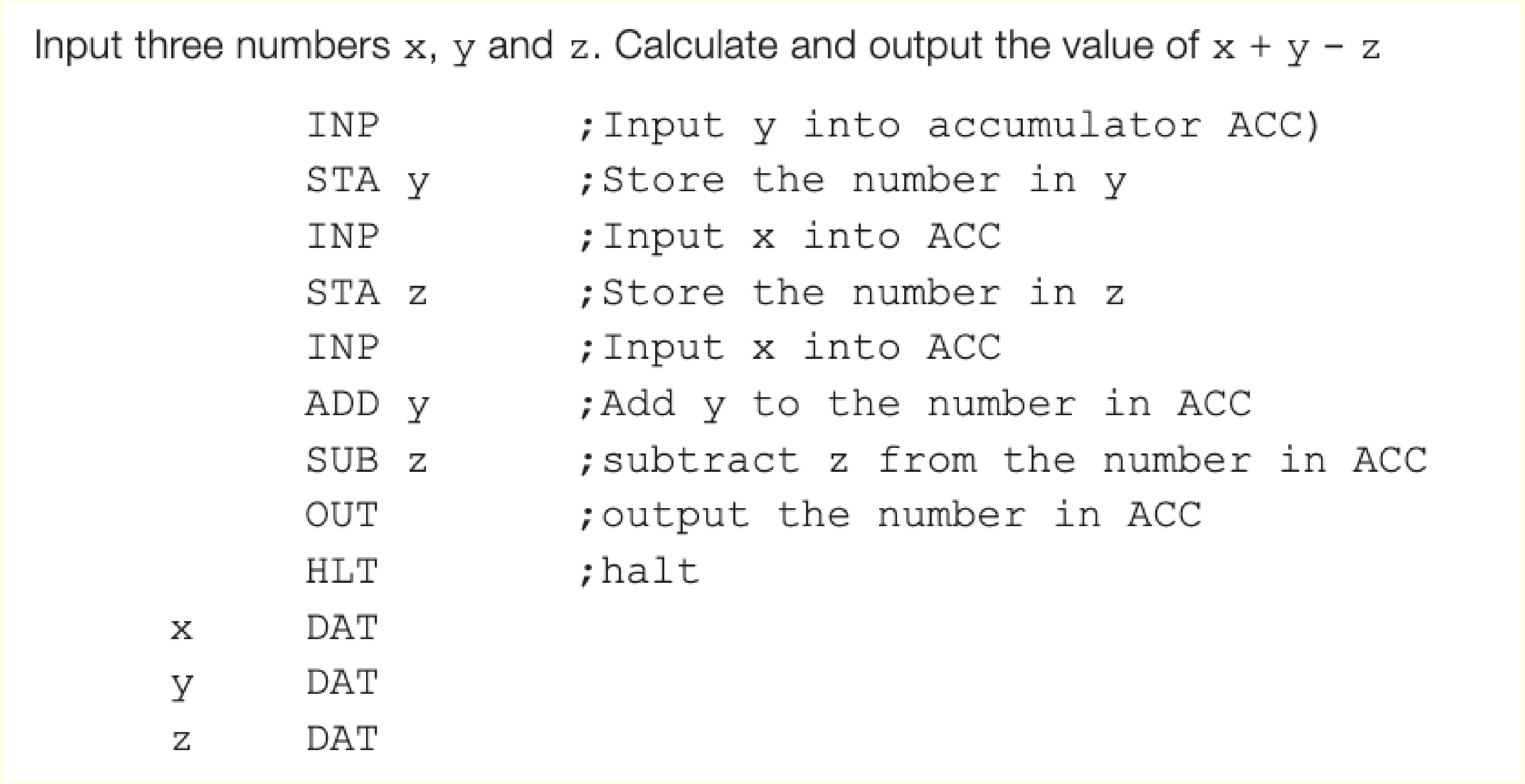
Here's how it works:

Storing a Variable

1. Use STA X to **store** a value in memory location X.
2. Use LDA X to **load** a value from memory location X into the accumulator.
3. Define the variable with DAT X, optionally initializing it.

```
INP              Take user input
STA NUM          Store input in memory location "NUM"
LDA NUM          Load the value back into the accumulator
OUT              Print the stored value
HLT              Halt the program

NUM    DAT       Reserve memory location for NUM (initially 0)
```

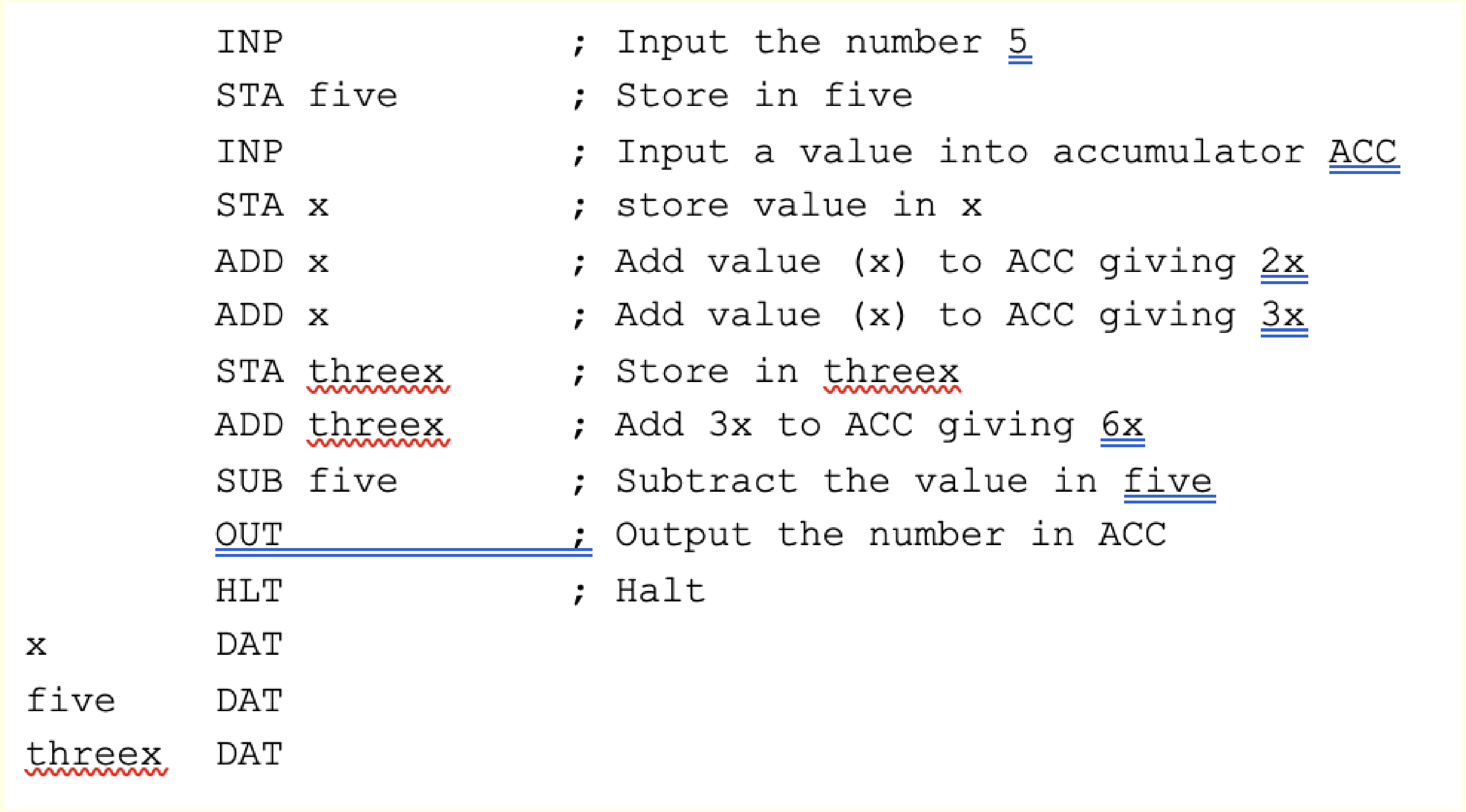
Data transfer and arithmetic operations



Input three numbers x, y and z. Calculate and output the value of x + y − z

```
          INP          ;Input y into accumulator ACC)
          STA y        ;Store the number in y
          INP          ;Input x into ACC
          STA z        ;Store the number in z
          INP          ;Input x into ACC
          ADD y        ;Add y to the number in ACC
          SUB z        ;subtract z from the number in ACC
          OUT          ;output the number in ACC
          HLT          ;halt
x         DAT
y         DAT
z         DAT
```

Example: Write an assembly code program to input a number x and calculate and output 6x – 5.



```
         INP               ; Input the number 5
         STA five          ; Store in five
         INP               ; Input a value into accumulator ACC
         STA x             ; store value in x
         ADD x             ; Add value (x) to ACC giving 2x
         ADD x             ; Add value (x) to ACC giving 3x
         STA threex        ; Store in threex
         ADD threex        ; Add 3x to ACC giving 6x
         SUB five          ; Subtract the value in five
         OUT               ; Output the number in ACC
         HLT               ; Halt
x        DAT
five     DAT
threex   DAT
```

Branch instructions

- The flow of the program can be altered using a conditional or unconditional branch instruction.

- The conditional branch instructions

BRP (Branch if positive)

BRZ (Branch if zero)

BRA - An **unconditional branch instruction** will cause a branch whatever the value held in the accumulator.

Selection in LMC

- In high-level languages selection takes place with the use of ***if..else*** and sometimes ***switch..case*** or equivalent.
- In LMC we use the branch instructions **BRP** and **BRZ**.

```
num1= input()
if num1>100 then
        print 100
  else
        print num1
  endif
```

```
INP                 ;Get num1 from user
SUB 99              ;Subtract 100 (stored at address 99)
BRP PRINT100        ;If num1 >= 100, jump to PRINT100
LDA 98              ;Load original num1 (stored at address 98)
OUT                 ;Print num1
HLT                 ;Stop the program

PRINT100      LDA 99       ; Load 100
              OUT          ;Print 100
              HLT          ;Stop the program


DAT 98        ;Memory location to store num1
DAT 99        ;Memory location with value 100 (constant)
```

In LMC we don't have access to operators such as > or <.
We do, however, know that if num1 is greater then 100 then 100 minus num1 will be negative.
We can use this to create a selection instruction.

Branching

- Compare the two numbers held in memory locations num1 and num2, and output the larger. If they are equal, either one can be output.

```
Inp
STA num2
Inp
STA num1
```

```
          LDA num1
          SUB num2
          BRP firstmax
          LDA num2
          OUT num2
          HALT
firstmax  LDA num1
          OUT num1
          HLT
```

Add variables
```
num1 DAT
num2 DAT
```

Iteration in LMC

- When we want to perform iteration (or looping) in a high-level language we usually have access to constructs such as ***for*** and ***while***.
- If we want a program that keeps asking the user for a number until they enter one under 100 it may look liked this:

```
num1=input("Enter a number less then 100")
while num1>100
    num1=input("Enter a number less then 100")
endwhile
```

```
loop        INP
            STA   Num1
            SUB   HundAndOne
            BRP   loop
            LDA   Num1
            OUT
            HLT
HundAndOne  DAT   101
Num1        DAT
```

As with selection, to perform iteration in LMC we use branches and labels.

We want to loop back to the top if the number entered is greater than 100.

To do this we, we subtract 101 from the number entered.

If the result is positive the number must be greater than 100. LMC treats zero as a positive number.

- This system does not have an instruction for **multiply** devise a solution using the **ADD** instruction

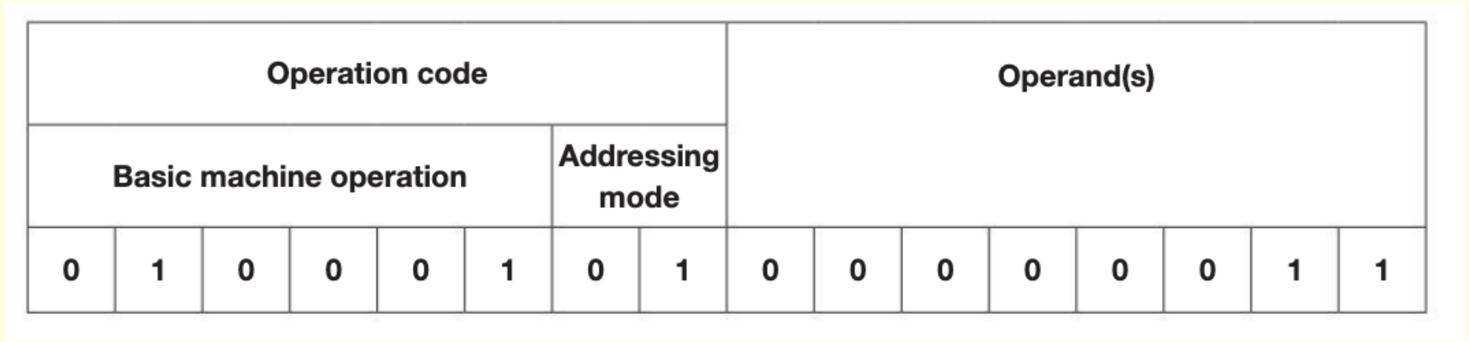
```
INP          Take input for N1 (multiplier)
STA N1          Store it in memory location N1

INP          Take input for N2 (multiplicand)
STA N2          Store it in memory location N2

LDA ZERO     Initialize result (0)
STA RESULT   Store the initial result as 0

LOOP:
      LDA RESULT   Load current result into the accumulator
      ADD N1       Add N1 to the result
      STA RESULT   Store the updated result

      LDA N2       Load N2 (the number of additions left)
      SUB ONE      Subtract 1 from N2
      STA N2       Store the updated N2

      BRZ DONE     If N2 is zero, exit loop

      BRA LOOP     Otherwise, continue the loop

DONE:
      LDA RESULT   Load the final result (N1 * N2)
      OUT          Output the result
      HLT          Halt the program

ZERO    DAT 000      Define 0 (for initializing result)
ONE     DAT 001      Define 1 (to decrement N2)
N1      DAT          Reserve memory for N1 (multiplier)
N2      DAT          Reserve memory for N2 (multiplicand)
RESULT  DAT          Reserve memory for the result
```

Learning Objectives

- Understand and apply immediate, direct, indirect and indexed addressing modes

Format of machine code instructions

The basic structure of a machine code instruction in a 16-bit word



Operation code								Operand(s)							
Basic machine operation						Addressing mode									
0	1	0	0	0	1	0	1	0	0	0	0	0	0	1	1

The LMC instruction set has only 11 instructions, and the imaginary machine has only 100 memory locations.

The maximum data value is 999, which can be held in 10 bits. Four bits would be enough to store the operation code, and 7 bits would be enough to store the operand.

A word size of 16 bits would be plenty big enough to hold an instruction or a data value.

Real computers

- In a real computer, there will be considerably more than 11 instructions in the instruction set.
- It will include, for example, multiply and divide in the arithmetic instructions, and shift instructions to shift bits left or right.
- There will also normally be up to 16 registers in which calculations can be carried out, rather than a single accumulator.

Addressing modes

Immediate Addressing (#)

# before a number means to use the **value itself**, not the address.

Example: ADD #4 means "Load the value 4 directly into the accumulator."

Direct Addressing (default, no symbol)

The operand refers directly to a memory address.

Example: LDA 10 means "Load the value at memory address 10."

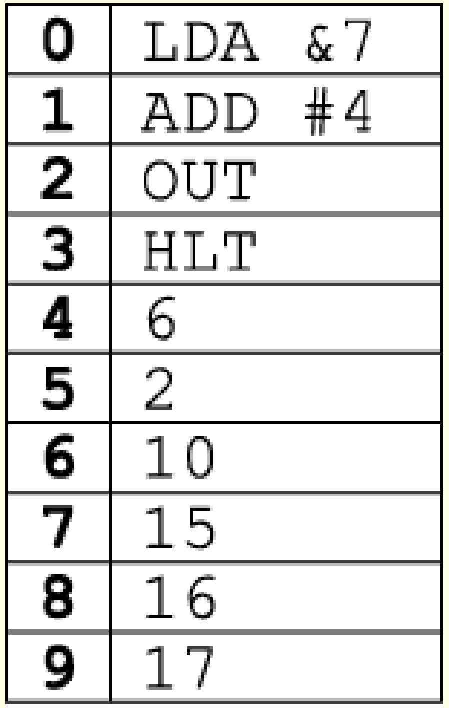
Indirect Addressing (&)

& means the operand is the address **of a memory location that contains another address**.

Example: LDA &7 means:

Go to memory address 7, read the number there 15

Then go to memory address 15 and load that value.



0	LDA &7
1	ADD #4
2	OUT
3	HLT
4	6
5	2
6	10
7	15
8	16
9	17

Fig. 3.1

Indexed Addressing

Indexed addressing uses a **base address + the value in an index register** to compute the final memory address. This allows for **array-like access** or **looping through data**.

For example:

- The instruction is **LDA 20,X**
- The index register X contains the value **0**

Then: The LMC loads the value from address **20 + 0 = 20**

So it accesses **memory address 20**.

After the operation, the index register **X increments by 1**.

How compilers work

Stages of compilation

Stages of compilation:

1. lexical analysis
2. syntax analysis
3. code generation and optimisation

Stage 1 Lexical Analysis

Step 1: **Lexer** scans the source code, letter by letter

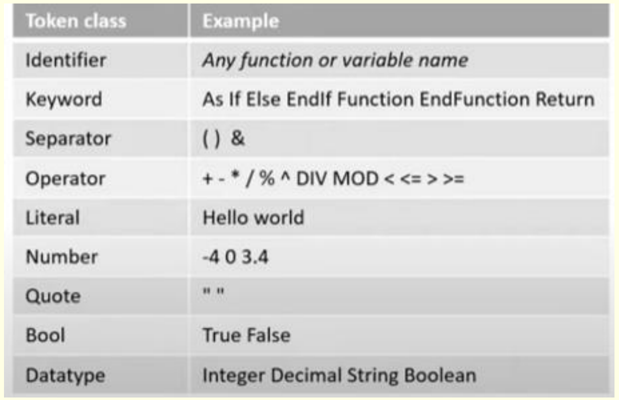
Step 2 Words/**lexeme** are identified when there is a white space, operator symbol or special symbol.

Step 3 The Lexeme is checked to see if it is a **valid token**

Step 4 The lexeme is then stored along with the token, for example,

[keyword: if] [Separator:(] [Identifier: x] [Operator:>] and added to a **symbols table.**

All keywords, constants and identifiers (e.g. variable names) used in the source code are replaced by '**tokens**'



Token class	Example
Identifier	*Any function or variable name*
Keyword	As If Else EndIf Function EndFunction Return
Separator	( ) &
Operator	+ - * / % ^ DIV MOD < <= > >=
Literal	Hello world
Number	-4 0 3.4
Quote	" "
Bool	True False
Datatype	Integer Decimal String Boolean

The symbol table

The symbol table plays a central role in the compilation process.

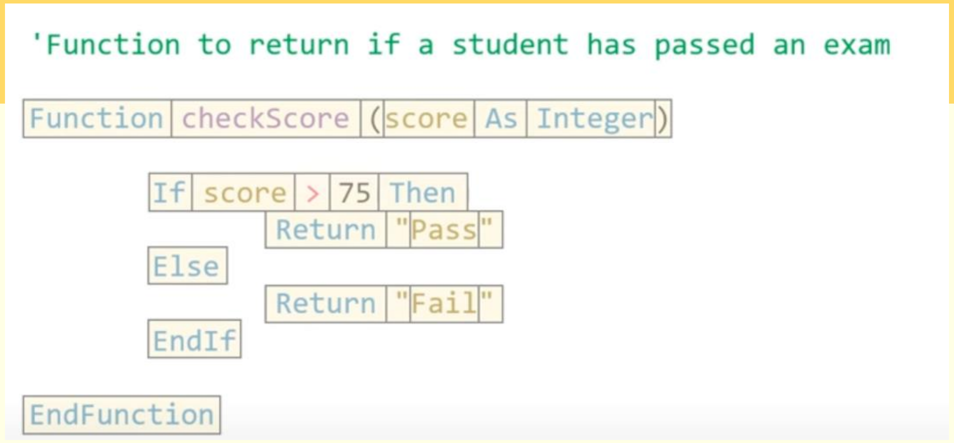
It will contain an entry for every keyword (reserved word) and identifier in the program.

The the table will show:

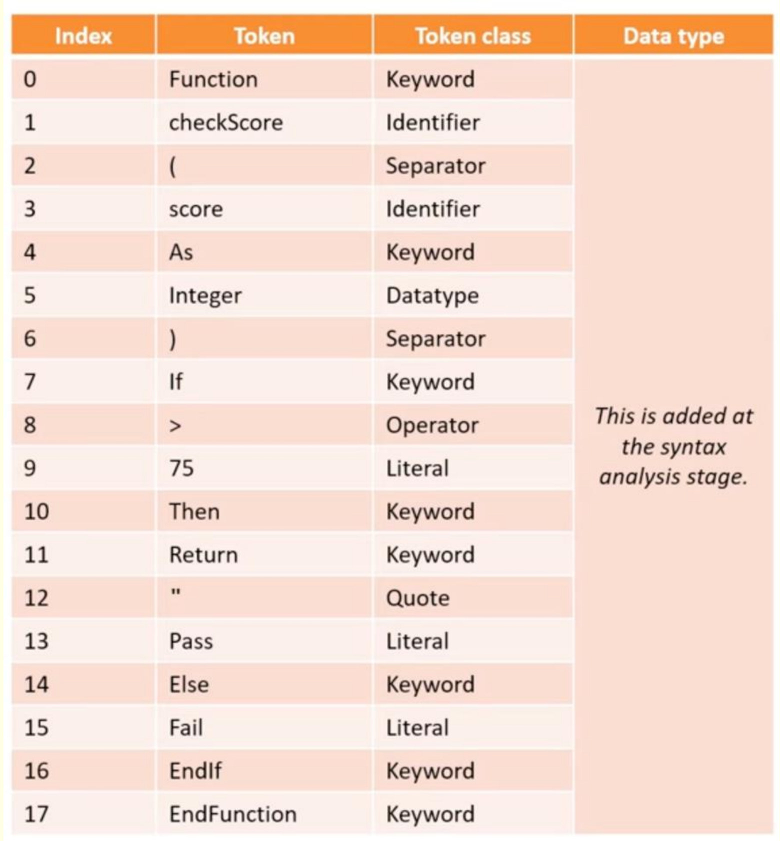
- **Identifier or keyword**
- **Kind of item** (variable, array, procedure, keyword etc.)
- **Type** of item (integer, real, char etc.) – added during the syntax stage

Also it will store:

- **Run-time address** of the item, or its value if it is a **constant**
- **Pointer** to accessing information (e.g. for an array, the bounds of the array, or for a procedure, information about each of the parameters).



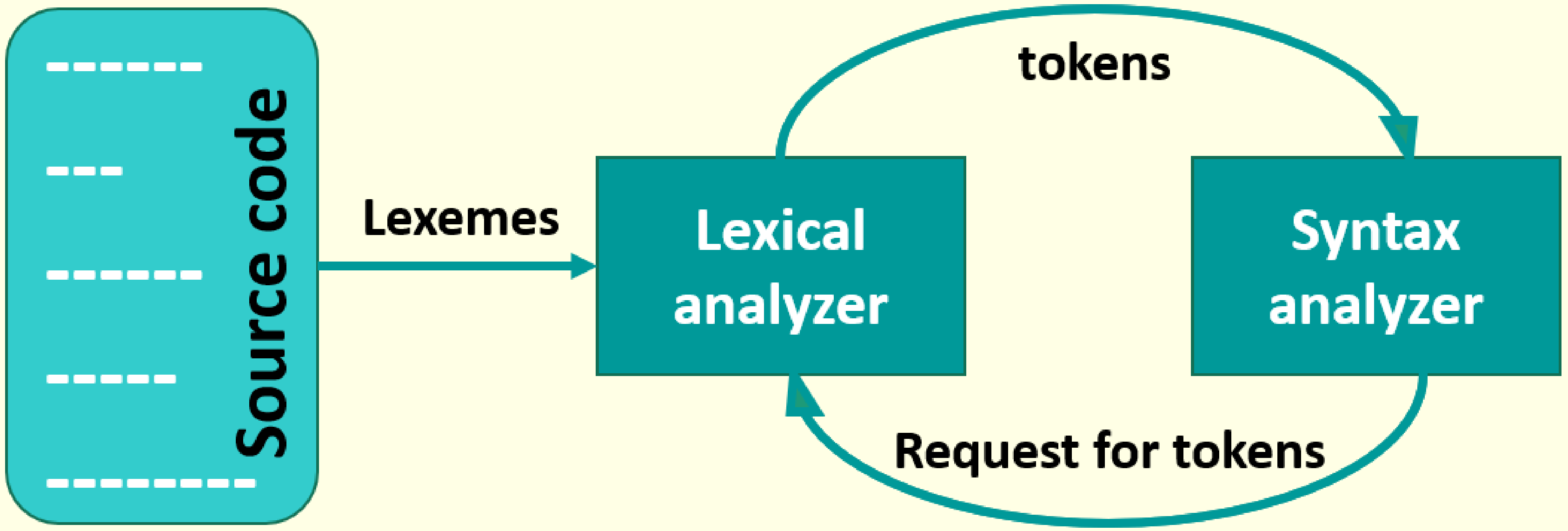
```
'Function to return if a student has passed an exam

Function checkScore (score As Integer)

        If score > 75 Then
                Return "Pass"
        Else
                Return "Fail"
        EndIf

EndFunction
```



Index	Token	Token class	Data type
0	Function	Keyword	*This is added at the syntax analysis stage.*
1	checkScore	Identifier	
2	(	Separator	
3	score	Identifier	
4	As	Keyword	
5	Integer	Datatype	
6	)	Separator	
7	If	Keyword	
8	>	Operator	
9	75	Literal	
10	Then	Keyword	
11	Return	Keyword	
12	"	Quote	
13	Pass	Literal	
14	Else	Keyword	
15	Fail	Literal	
16	EndIf	Keyword	
17	EndFunction	Keyword	

Stage 2 Syntax analysis

- This is alternatively known as **parsing**.
- This stage analyses the syntax of the statements to ensure they **conform to the rules of grammar for the computer language** in question.
- The purpose of syntax analysis or parsing is to check that we have a valid sequence of tokens.
- Tokens are a valid sequence of symbols, keywords, identifiers, etc.



Syntax Analysis

1. **Token stream** sent to the **syntax analyser**
2. Checks that the token stream follows the grammar rules of the language
3. Stacks will be used to check, for example, that brackets are correctly paired.
4. If fails Report errors
5. Reports diagnostics
6. Create **abstract syntax tree**
7. Update identifiers in the symbols table

Stage 3 Code generation and optimisation

This is the final phase of compilation creates **object code**

To make the program run faster/ code is more efficient
To make the program use fewer resources/less memory

The disadvantages of code optimisation are:

- it will increase compilation time, sometimes quite considerably
- it may sometimes produce unexpected results.

Example:

- Removes variables and subprograms that are not used
- Removing lines of code that are never accessed

Summary of the stages of compilation

Lexical analysis

- Comments and white space removed.
- Remaining code turned into a series of tokens.
- Symbol table is created to keep track of variables and subroutines.

Syntax analysis

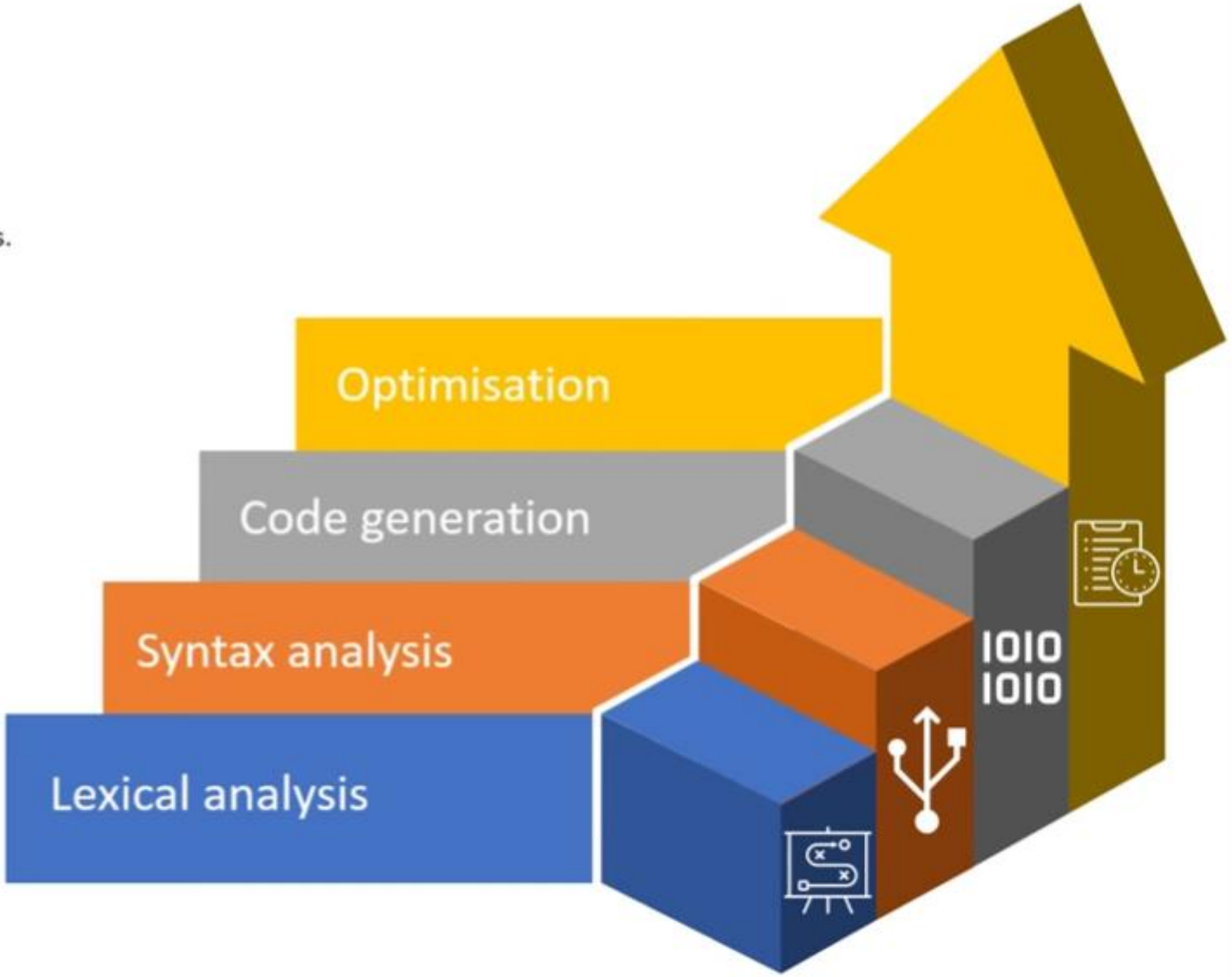
- Abstract syntax tree is built from tokens produced in the previous stages.
- Errors generated if any tokens break the rules of the language.

Code generation

- Abstract code tree converted to object code.
- Object code is the machine code produced before the final step (linker) is run.

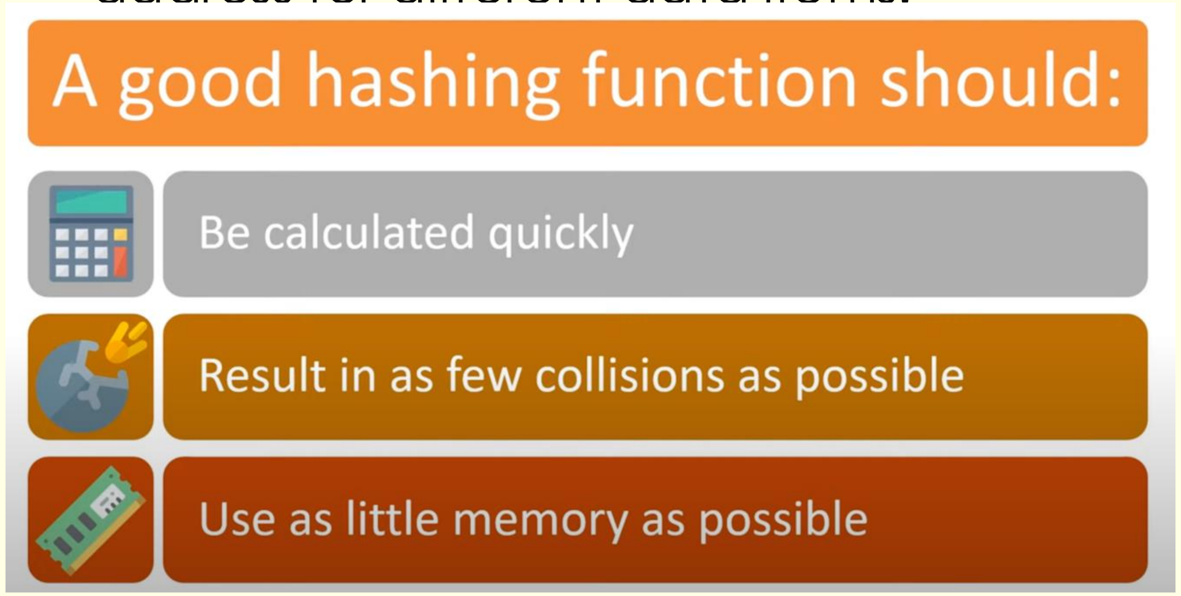
Optimisation

- Tweaks the code so it will run as quickly and use as little memory as possible.



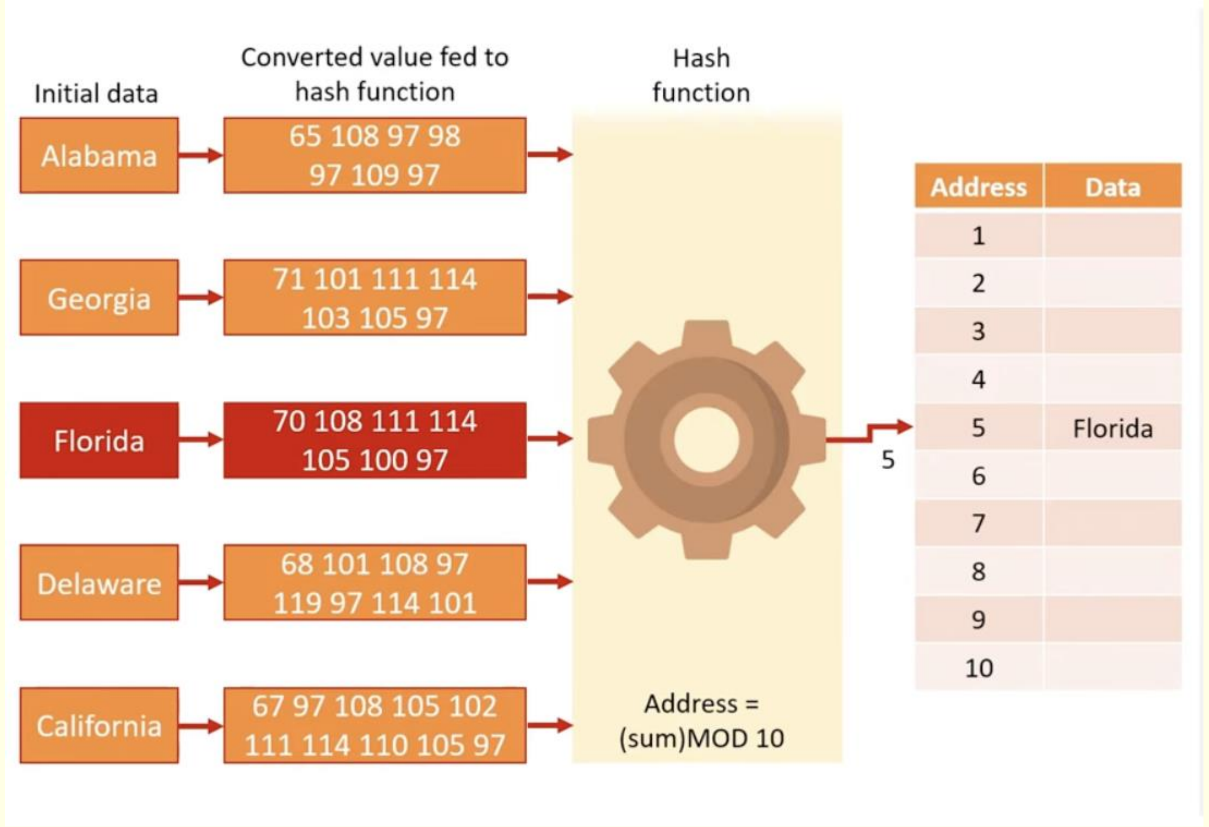
Hash Tables

- Hash Table is used to immediately find an item.
- A hashing function is used to calculate the position of item in a hash table
- Many different hashing functions are used.
- A hash function may course a collision – meaning the function generates the same address for different data items.



A good hashing function should:

- Be calculated quickly
- Result in as few collisions as possible
- Use as little memory as possible



Linker and Loader

What is a linker?

1 mark per bullet to max 2:

- Combines/links code/programs to files/software libraries…
- …to form a single executable file

- **Static linkers** combine code and libraries into one file
- **Dynamic linkers** link - add addresses to libraries

What is a loader?

1 mark per bullet to max 2:

- It is part of the operating system
- Loads an **executable file** (into memory)...
- …from secondary storage
- Loads the required **software libraries**

Use of libraries

Library programs are ready-compiled programs, grouped in software libraries, which can be loaded and run when required. In Windows these often have a .dll extension.

Most compiled languages have their own libraries of pre-written functions which can be invoked in a defined manner from within the user's program.

Advantages of library routines

These libraries can be imported into a user's program and have many advantages including:

- **They are tested and error-free**
- **They save the programmer time in "re-inventing the wheel" to write code themselves to perform common tasks**