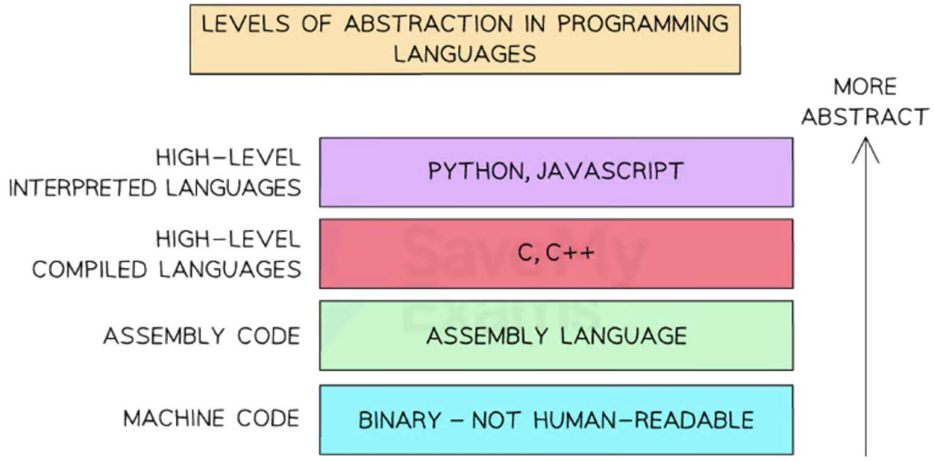
Machine code vs assembly

What is machine code?

- **Machine code** is a first-generation language
- Instructions are **directly executable by the processor**
- Written in **binary code**



Levels of Abstraction of Programming Languages

What is assembly?

- **Assembly code** is a second-generation language
- The code is written using **mnemonics**, abbreviated text commands such as **LDA (Load), STA(Store)**
- Using this language programmers can write **human-readable programs** that correspond **almost exactly to machine code**
- **One assembly language instruction** translates to **one machine code instruction**
- **Needs to be translated** into machine code for the computer to be able to execute it
- Each type of computer CPU has a **specific instruction set**

Instruction set

- An instruction set is a list of **all the commands** that can be **processed by a CPU**
- Each command has a **binary code** which is the **machine code**
- In assembly the binary code is written using mnemonics and split into an **opcode** and **operand**
- The **opcode** is the **part of an instruction** that tells the CPU what **operation** to perform It's short for **"operation code"**
  Example operations: *LDA* (load), *STA* (store), *ADD*, *INP*, *OUT*, *BRZ*, *BRA*

- The **operand** is the **data** or **memory address** the opcode will work with
- It gives the **extra detail** the CPU needs to complete the instruction
- Often used to specify a **memory address**, a **value**, or a **label**

Opcode	Operand	Explanation
LDM	#n	Immediate addressing. Load the number n to the accumulator (ACC).
LDD	<address>	Direct addressing. Load the contents of the location at the given address to ACC.
LDI	<address>	Indirect addressing. Use the value at the given address as a new address. Load contents to ACC.
LDX	<address>	Indexed addressing. Use <address> + index register to get the final address. Load to ACC.
LDR	#n	Immediate addressing. Load the number n to the index register (IX).
MOV	<register>	Move the contents of ACC to the given register (e.g. IX).
STO	<address>	Store the contents of ACC at the given memory address.
ADD	<address>	Add the contents of the given address to ACC.
SUB	<address>	Subtract the contents of the given address from ACC.
INC	<register>	Increment the contents of the register (ACC or IX) by 1.
DEC	<register>	Decrement the contents of the register (ACC or IX) by 1.
JMP	<address>	Unconditional jump to the given address.
CMP	<address>	Compare the contents of ACC with the contents of the given address.
CMP	#n	Compare the contents of ACC with the number n.

CMI	<address>	Indirect addressing. Compare ACC with the contents at the address stored in <address>.
JPE	<address>	Jump to <address> if the previous compare result was **true** (equal).
JPN	<address>	Jump to <address> if the previous compare result was **false** (not equal).
IN		Take input from the keyboard and store its ASCII value in ACC.
OUT		Output to the screen the character stored in ACC.
END		End the program and return control to the operating system

All questions will assume there is only one general purpose register available (Accumulator)

ACC denotes Accumulator IX denotes Index

Register

<address> can be an absolute or symbolic address # denotes a

denary number, e.g. #123

B denotes a binary number, e.g. B01001010 & denotes a

hexadecimal number, e.g. &4A

Two-pass assembler

What is a two-pass assembler?

- A two-pass assembler **translates assembly language into machine code** in two stages
- It produces an **object program** that can be **stored**, **loaded**, and **run later**
- To execute this program, another utility called a **loader** is needed
- The assembler scans the source code **twice**:
  - In the **first pass**, it identifies all **labels** and records their memory addresses
  - In the **second pass**, it replaces those labels with the correct **memory addresses** in the machine code
- This **ensures the final machine code is accurate** and ready to be executed

Example

- The following program loads the value of A (5), adds the value of B (3), stores the result (8) in C, then halts:

```
    LOAD A
    ADD B
    STORE
    C HALT
A:  DATA 5
B:  DATA 3
C:  DATA 0
```

- **Pass 1:**
  - The assembler reads each line and assigns memory addresses to labels (e.g. A, B, C), but **does not translate the instructions yet**

Address	Instruction	Label	Add to symbol table
00	LOAD A		
01	ADD B		
02	STORE C		
03	HALT		
04	DATA 5	A	**A** = 04
05	DATA 3	B	**B** = 05
06	DATA 0	C	**C** = 06

Symbol table created:

Label	Address
A	04
B	05
C	06

- **Pass 2:**
  - The assembler replaces labels with their corresponding memory addresses and outputs machine code
  - In this example we assume the following opcodes for simplicity:

Mnemonic	Opcode
LOAD	01
ADD	02
STORE	03
HALT	00
DATA	-

- Now generate the object (machine) code:

Address	Assembly	Machine code	Explanation
00	LOAD A	01 04	LOAD from address 04
01	ADD B	02 05	ADD value at address 05
02	STORE C	03 06	STORE into address 06
03	HALT	00 00	Stop program
04	DATA 5	00 05	Data value 5
05	DATA 3	00 03	Data value 3
06	DATA 0	00 00	Data value 0 (placeholder)

Final object program:

01 04
02 05
03 06
00 00
00 05
00 03
00 00

Categories of instructions

What are the categories of instructions?

Assembly language instructions can be grouped into:

Data Movement
Access the value and loading in the accumulator.

Input and Output of Data
Takes an input from user and output the characters of binary numbers.

Arithmetic Operations
Perform addition and subtraction

Unconditional and Conditional instructions
Move to another instruction (identify by labels)

Compare Instructions
Compare the result to another value.

For each category, a table containing the instruction (Opcode and Operand), and an explanation is given

Category	Opcode	Operand	Explanation
Data Movement	LOAD	A	Loads the value stored at memory location A into the accumulator.
	STORE	B	Stores the value from the accumulator into memory location B.
	MOV	A, B	Copies the value from register/memory A to register/memory B.
	CLEAR	ACC	Clears the accumulator (sets it to 0).
	DATA	5	Declares a constant or data value (e.g. DATA 5 stores the value 5).
Input and Output of Data	IN		Takes input from a user or input device and stores it in the accumulator.
	OUT		Outputs the value in the accumulator to a screen or output device.

Arithmetic Operations	ADD	A	Adds the value at memory location A to the accumulator.
	SUB	B	Subtracts the value at memory location B from the accumulator.
	INC		Increments the accumulator by 1.
	DEC		Decrements the accumulator by 1.
Unconditional and Conditional	JMP	LABEL	Unconditionally jumps to the instruction at LABEL.
	JZ	LABEL	Jumps to LABEL if the accumulator is zero.
	JNZ	LABEL	Jumps to LABEL if the accumulator is **not** zero.
	JC	LABEL	Jumps to LABEL if the carry flag is set.
	HLT		Stops the program (halts execution).
Compare Instructions	CMP	A	Compares the accumulator with the value at memory location A.
	TEST	A	Performs a logical AND between accumulator and A; sets flags, no output.

Addressing methods

What is an addressing method?

- An addressing method is a way in which **an instruction in assembly language or machine code** can **access data stored in memory**
- There are five main addressing methods:
  - **Immediate**
  - **Direct**
  - **Indirect**
  - **Indexed**

Concept of accumulator: Accumulator is a place where the results are stored.

IMMEDIATE ADDRESSING
The operand is the value to be used in the instruction.

Example:
LDM
Just load the value which is written next to the LDM instruction.

Opcode	Operand
LDM	#100

ACC
100

DIRECT ADDRESSING
The operand is the address which holds the value to be used in the instruction;
Memory will be involved.

Example:
LDD Address

Opcode	Operand
LDD	Address

Opcode	Operand
LDD	17

ACC
19

Memory	
Location/ Address	Content
15	25
16	35
17	19
18	55
19	30
29	22
30	15
31	7

Step-1 Go to the given address
Step-2 Load the value on that address in ACC

INDIRECT ADDRESSING

The operand is an address that holds the address which has the value to be used in the instruction. So two addresses will be involved.

Example:

Opcode	Operand
LDI	First Address

Opcode	Operand
LDI	17

ACC
30

Memory	
Location/ Address	Content
15	25
16	35
17	19
18	55
19	30
29	22
30	15
31	7

INDEX ADDRESSING

The operand is an address to which must be added the value currently in the index register (IX) to get the address which holds the value to be used in the instruction.

Example:

Opcode	Operand
LDX	Initial Address

Opcode	Operand
LDX	17

ACC
55

Memory	
Location/ Address	Content
15	25
16	35
17	19
18	55
19	30
29	22
30	15
31	7
IX	1

17+[IX]
=17+1
=18 (New Address)

Step-1 Add the initial address with IX and create a new address.
Step-2 Use the new address and load that value in accumulator.

Program tracing

What is a trace table?

- A **trace table** is a table used to **manually track the values of variables** as a program runs
- It helps to follow the flow of a program **line by line**, checking whether the logic works as expected
- Trace tables are used to:
  - **Understand how a program works**
  - **Debug errors** in the logic
  - **Predict outputs** before running the program
  - **Check variable values** at each step

Example

- An assembly program that loads the value from A (4) into the accumulator, adds the value from B (2), stores the result (6) into C, and halts

```
    LOAD A
    ADD B
    STORE
    C HALT
A:  DATA 4
B:  DATA 2
C:  DATA 0
```

- To trace the program, assumptions must be made:
  - **Accumulator (ACC)**: Used for calculations
  - Memory is addressed from 00 onwards
  - Simple instructions:
    - LOAD X → ACC = memory[X]
    - ADD X → ACC = ACC + memory[X]
    - STORE X → memory[X] = ACC
    - HALT → Stop

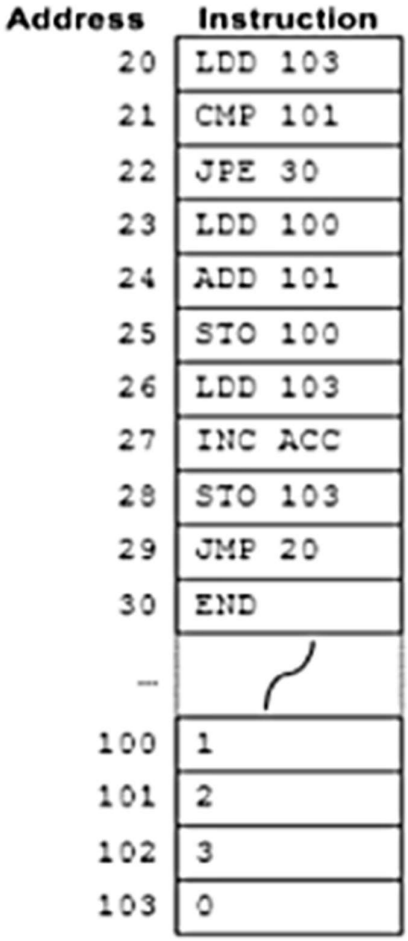
Trace table

Step	Instruction	ACC (Accumulator)	Memory[A]	Memory[B]	Memory[C]
0	(Start)	0	4	2	0
1	LOAD A	4	4	2	0
2	ADD B	6	4	2	0
3	STORE C	6	4	2	6
4	HALT	6	4	2	6

Final outcome:

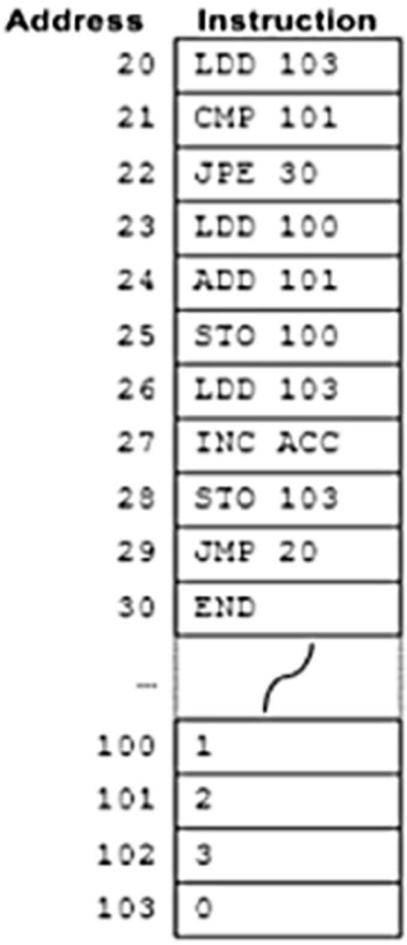
- **ACC = 6**
- **C = 6**
- The program has successfully added A and B, then stored the result in C

Instruction		Explanation
Op Code	Operand	
LDM	#n	Immediate addressing. Load the number n to ACC
LDD	<address>	Direct addressing. Load the contents of the location at the given address to ACC
LDI	<address>	Indirect addressing. The address to be used is at the given address. Load the contents of this second address to ACC
LDX	<address>	Indexed addressing. Form the address from <address> + the contents of the index register. Copy the contents of this calculated address to ACC
LDR	#n	Immediate addressing. Load the number n to IX
STO	<address>	Store the contents of ACC at the given address
ADD	<address>	Add the contents of the given address to the ACC
INC	<register>	Add 1 to the contents of the register (ACC or IX)
DEC	<register>	Subtract 1 from the contents of the register (ACC or IX)
JMP	<address>	Jump to the given address
CMP	<address>	Compare the contents of ACC with the contents of <address>
CMP	#n	Compare the contents of ACC with number n
JPE	<address>	Following a compare instruction, jump to <address> if the compare was True
JPN	<address>	Following a compare instruction, jump to <address> if the compare was False
IN		Key in a character and store its ASCII value in ACC
OUT		Output to the screen the character whose ASCII value is stored in ACC
END		Return control to the operating system

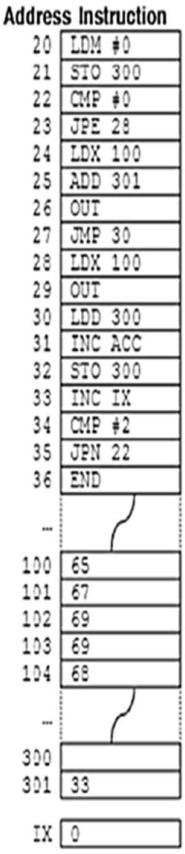


Instruction address	ACC	Memory address			
		100	101	102	103
		1	2	3	0
20	0				

Solution



Instruction address	ACC	Memory address			
		100	101	102	103
		1	2	3	0
20	0				
21					
22					
23	1				
24	3				
25		3			
26	0				
27	1				
28					1
29					
20	1				
21					
22					
23	3				
24	5				
25		5			
26	1				
27	2				
28					2
29					
20	2				
21					
22					
30					

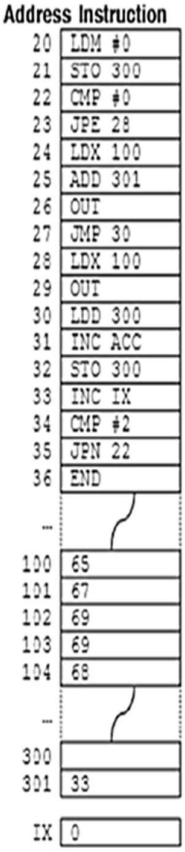


ASCII code table

ASCII Code	Character
65	A
66	B
67	C
68	D
69	E
97	a
98	b
99	c
100	d
101	e

Instruction address	ACC	Memory address							IX	OUTPUT
		100	101	102	103	104	300	301		
		65	67	69	69	68		33	0	
20	0									

Solution:



ASCII code table

ASCII Code	Character
65	A
66	B
67	C
68	D
69	E
97	a
98	b
99	c
100	d
101	e

Instruction address	ACC	Memory address							IX	OUTPUT
		100	101	102	103	104	300	301		
		65	67	69	69	68		33	0	
20	0									
21							0			
22										
23										
28	65									
29										A
30	0									
31	1									
32							1			
33									1	
34										
35										
22										
24	67									
25	100									
26										d
27										
30	1									
31	2									
32							2			
33									2	
34										
36										