Exercise walk-through

Assembly Language ▪ 4.2

eXercise

You must be able to

- explain the relationship between **assembly language** and **machine code** 机器码
- describe the stages of a **two-pass assembler**
- use the **modes of addressing** (immediate, direct, indirect, indexed)
- **trace** a simple assembly-language program

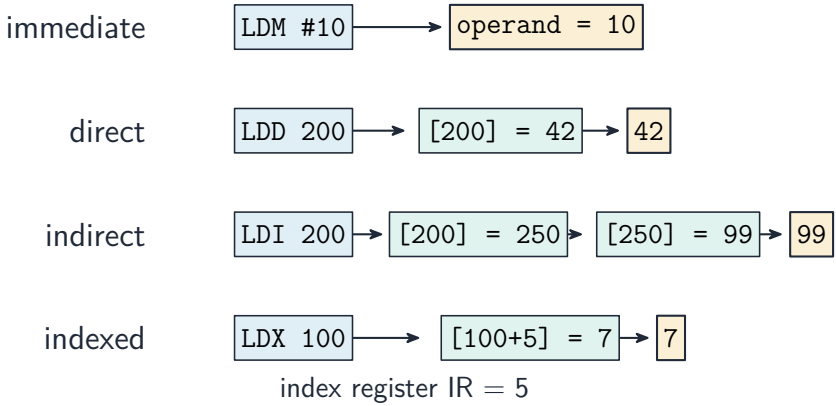
Method — Assembly, machine code and the assembler

The CPU runs **machine code** (binary). **Assembly language** is a readable form — one **mnemonic** 助记符 (like `LDD`, `ADD`) per machine instruction. An **assembler** translates it. A **two-pass assembler** reads the source twice:

- **Pass 1** builds a **symbol table** — it records the address of every **label** (e.g. `LOOP:`).
- **Pass 2** generates the code — and when an instruction uses a label (e.g. `JMP LOOP`), it looks the address up.

Two passes are needed for **forward references** — a jump to a label that is defined *later* in the code.

Method — Assembly, machine code and the assembler



Assembly instructions use addressing modes to locate their operands

Method — Modes of addressing

The addressing mode says **how the operand is found**:

Mode	The operand is...	Example
immediate	the value in the instruction	`LDM #10` → loads 10
direct	the value at the given address	`LDD 200` → loads contents of 200
indirect	at the address *stored* at the given address	`LDI 200`
indexed	at `address + index register (IX)`	`LDX 100`, IX = 5 → reads 105

Method — Common instructions

You will also meet these in traces (the exam gives the full list):

Mnemonic	What it does
`STO <addr>`	store the ACC into the address
`ADD` / `INC` / `DEC`	add to / add 1 to / subtract 1 from
`CMP <addr>`	compare the ACC with the contents of the address
`JMP` / `JPE` / `JPN`	jump always / jump if the last compare was **equal** / **not equal**
`IN` / `OUT`	input / output one character (its ASCII value)
`END`	stop the program

Method — Tracing a program

Make a table with a column for the **ACC** and for each variable, then step through, updating after each instruction. Trace this (start: `num1` $= 6$, `num2` $= 9$):

Instruction	**ACC**	**total**
`LDM #0`	0	
`STO total`	0	0
`LDD num1`	6	0
`ADD num2`	15	0
`STO total`	15	15

Method — Tracing a program

So `total` ends as **15**.

Practice 2.1 ▪ 2 marks

State the difference between **assembly language** and **machine code**.

Solution 2.1

Method: Assembly, machine code and the assembler

Worked steps

Machine code is **binary** that the CPU runs directly; assembly language is a **readable** version using mnemonics, with one instruction per machine instruction B1 B1.

Practice 2.2 ▪ 2 marks

State what each instruction loads into the ACC: (a) `LDM #20` (b) `LDD 20`.

Solution 2.2

Method: Modes of addressing

Worked steps

(a) the value **20** (immediate) B1. (b) the **contents of address 20** (direct) B1.

Practice 2.3 ▪ 2 marks

Explain why an assembler needs **two passes**.

Solution 2.3

Method: Assembly, machine code and the assembler

Worked steps

A jump can refer to a label defined **later** (a forward reference) B1; pass 1 records every label's address first, so pass 2 can fill it in B1.

Practice 2.4 ▪ 1 mark

`LDX 100` is executed when the index register IX = 4. State which address is read.

Solution 2.4

Method: Modes of addressing

Worked steps

Address **104** (100 + IX 4) B1.

Exam-style 3.1 ▪ 4 marks

(a) Describe what happens in **pass 1** and in **pass 2** of a two-pass assembler.
(b) State why two passes are needed.

Solution 3.1

Method: Assembly, machine code and the assembler

Worked steps

(a) **Pass 1:** scan the source and build the symbol table — record the address of each label; do not generate code yet B1 B1. **Pass 2:** translate each instruction to machine code, looking up label addresses in the symbol table B1 B1.
(b) To resolve **forward references** (a jump to a label defined later) B1.

Exam-style 3.2 ▪ 3 marks

Explain the difference between **immediate**, **direct** and **indirect** addressing.

Solution 3.2

Method: Modes of addressing

Worked steps

Immediate = the operand **is the value** in the instruction; direct = the operand is at the **address** given; indirect = the address given holds **another address**, which holds the data B1 B1 B1.

Exam-style 3.3 ▪ 3 marks

Trace this program (start: a = 10, b = 4) and complete the table.

Instruction	ACC	c
LDD a		
ADD b		
STO c		
END		

Solution 3.3

Method: Tracing a program

Worked steps

`LDD` a $\rightarrow$ ACC 10, c — ; `ADD` b $\rightarrow$ ACC 14, c — ; `STO` c $\rightarrow$ ACC 14, c 14; `END` $\rightarrow$ c **14** M1 A1 A1 *ACC steps, ACC 14, c 14.*

Exam-style 3.4 ▪ 1 mark

Give one reason a programmer might write part of a program in assembly language rather than a high-level language.

Solution 3.4

Method: Assembly, machine code and the assembler

Worked steps

e.g. to control hardware directly / for speed / for a small, tight routine B1.