A-LEVEL COMPUTER SCIENCE • EXERCISE SHEET

4.2 Assembly Language

Name: _____ Class: _____ Date: _____ **Total: 19 marks**

KEY WORDS assembly language 汇编语言 • assembler 汇编器 • machine code 机器码 • addressing mode 寻址方式 • index register 变址寄存器

Objective

Build the skills to answer exam questions on **assembly language** 汇编语言 — how it relates to machine code, the two-pass assembler, the modes of addressing, and tracing a program.

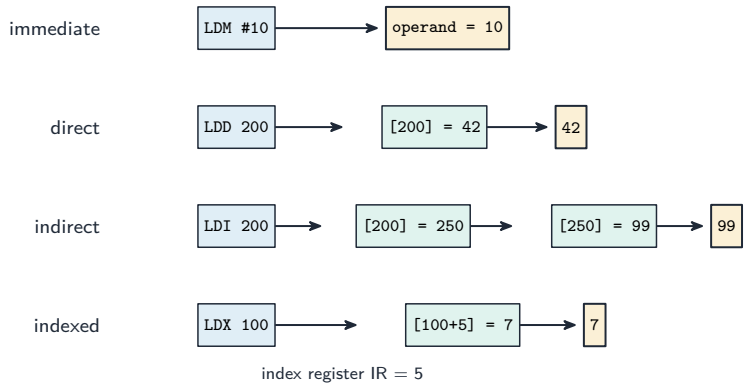
You must be able to:

- 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

1 Worked examples

Study these first. Each one shows the method for a question type used later — follow the steps and you can do the Practice and Exam-style questions yourself.

■ Assembly, machine code and the assembler



Assembly instructions use addressing modes to locate their operands

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.

■ 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

■ 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

■ 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

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

2 Practice

Now apply the methods above.

2.1 State the difference between **assembly language** and **machine code**. [2]

2.2 State what each instruction loads into the ACC: (a) `LDM #20` (b) `LDD 20`. [2]

2.3 Explain why an assembler needs **two passes**. [2]

2.4 `LDX 100` is executed when the index register IX = 4. State which address is read. [1]

3 Exam-style questions

3.1 (a) Describe what happens in **pass 1** and in **pass 2** of a two-pass assembler. [4]

(b) State why two passes are needed. [1]

3.2 Explain the difference between **immediate**, **direct** and **indirect** addressing. [3]

3.3 Trace this program (start: `a` = 10, `b` = 4) and complete the table. [3]

Instruction	ACC	c
`LDD a`		
`ADD b`		
`STO c`		
`END`		

3.4 Give one reason a programmer might write part of a program in assembly language rather than a high-level language. [1]

Solutions

Each answer shows the steps, then the **result**.

2.1

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

2.2

(a) the value **20** (immediate)

(b) the **contents of address 20** (direct)

2.3

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

2.4

- indexed: base 100 + IX 4 → address **104**

3.1

(a) **pass 1:** scan the source and build the symbol table — record the address of each label; do not generate code yet

- **pass 2:** translate each instruction to machine code, looking up label addresses in the symbol table

(b) to resolve **forward references** (a jump to a label defined later)

3.2

- 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

3.3

- start: $a = 10$, $b = 4$

```
Instruction   ACC    c
LDD a          10    -
ADD b          14    -
STO c          14    14
END            14    14
```

- ACC after `ADD` = **14**
- c after `STO` = **14**

3.4

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