Assembly language, Assemblers, & Translation softwares

Made for AS level at Musashi International School Tokyo

Outline

Prerequisites

To understand this topic, you should be familiar with:

- The basic architecture of a computer system (CPU, memory, I/O)
- The fetch-decode-execute cycle
- Binary and hexadecimal number systems
- Basic programming concepts (variables, instructions, control flow)

Definitions

Take some time to write definitions for the following terms:

HLL (High-level language)

LLL (Low-level language)

Assembly language

Machine code

Assembler

Definitions (answers)

HLL	High-level language: closer to natural language and portable; must be translated (compiled/interpreted) before execution.

Definitions (answers)

HLL	High-level language: closer to natural language and portable; must be translated (compiled/interpreted) before execution.
LLL	Low-level language: close to the hardware (e.g., machine code and assembly); architecture specific.

Definitions (answers cont.)

Assembly language	A low-level language that uses mnemonics and labels to represent the machine instructions of a specific CPU/instruction set.

Definitions (answers cont.)

Assembly language	A low-level language that uses mnemonics and labels to represent the machine instructions of a specific CPU/instruction set.
Machine code	The binary/hex-encoded instructions and operands executed directly by the CPU.

Definitions (answers cont.)

Assembly language	A low-level language that uses mnemonics and labels to represent the machine instructions of a specific CPU/instruction set.
Machine code	The binary/hex-encoded instructions and operands executed directly by the CPU.
Assembler	A translator that converts assembly language into machine code, resolving labels (symbols) into addresses.

Outline

CIE curriculum focus (9618 AS 4.2)

We will cover:

- Relationship between assembly language and machine code
- Why an assembler is needed (translation and label resolution)
- Instruction groups: data movement, I/O, arithmetic, compare, conditional/unconditional control
- Addressing modes: immediate, direct, indirect, indexed, relative
- Tracing using ACC/IX and a memory table
- Two-pass assembly: Pass 1 (symbol table + addresses), Pass 2 (machine code output)

Machine code (reminder)

Machine code is the **binary/hex-encoded instructions and operands executed directly by the CPU**. Here is an example of two machine code instructions:

1) `00100111 00001010`
2) `D2 13`

Machine code (reminder)

Machine code is the **binary/hex-encoded instructions and operands executed directly by the CPU**. Here is an example of two machine code instructions:

1) `00100111 00001010`
2) `D2 13`

What do these bits mean:

- First part is the **opcode** (which instruction to do). This is determined by the **CPU's instruction set**. It is hardware specific and **will never change** for that CPU.
- Second part is the **operand** (data, an address, or how to find the data)

Opcodes correspond to specific instructions, and each CPU has its own instruction set defining these opcodes.
Which means:

Remember: A specific program in machine code will only run on the CPU architecture it was designed for. A different CPU will need a different machine code program.

The **set** of opcodes and their meanings is called the **instruction set** of that CPU. This is what differs between CPUs. CPU manufacturers design their own instruction sets (e.g., x86, ARM, MIPS).

RISC vs CISC

There are two main types of instruction sets:

- **RISC (Reduced Instruction Set Computer)**: simple instructions, usually one machine instruction per assembly instruction; easier to learn and understand; examples: ARM, MIPS, CIE instruction set.
- **CISC (Complex Instruction Set Computer)**: more complex instructions that can do multiple operations; can be more efficient in terms of code size; examples: x86.

Remember: RISC uses simpler instructions for easier understanding, while CISC has complex instructions for efficiency. Both are modern and valid CPU designs.

Question: why don't we write machine code?

Look at this example again:

1) `00100111 00001010`
2) `D2 13`

– Can you make sense of what these instructions do?

Question: why don't we write machine code?

Look at this example again:

1) `00100111 00001010`
2) `D2 13`
   - Can you make sense of what these instructions do?
   - How readable are they for humans?

Question: why don't we write machine code?

Look at this example again:

1) `00100111 00001010`
2) `D2 13`

- Can you make sense of what these instructions do?
- How readable are they for humans?
- How easy is it to write without making mistakes?

Question: why don't we write machine code?

Look at this example again:

1) `00100111 00001010`
2) `D2 13`

- Can you make sense of what these instructions do?
- How readable are they for humans?
- How easy is it to write without making mistakes?
- How easy is it to debug and maintain later?

Question: why don't we write machine code?

Look at this example again:

1) `00100111 00001010`
2) `D2 13`
   - Can you make sense of what these instructions do?
   - How readable are they for humans?
   - How easy is it to write without making mistakes?
   - How easy is it to debug and maintain later?
   - What happens if you change to a different CPU/architecture?

Remember: Machine code is difficult for humans to read, write, and maintain, and it is not portable across different architectures.

Remember: Machine code is difficult for humans to read, write, and maintain, and it is not portable across different architectures.

To solve some of these problems, we use **assembly language**, which uses **mnemonics and labels** to represent machine instructions in a more human-readable form.

mnemonics	Short, memorable codes that represent machine instructions (e.g., `LDA` for "load accumulator").
labels	Symbolic names for memory addresses, making it easier to reference data and instructions.

Example: assembly language vs. machine code

Here is the assembly language that might be equivalent to the previous machine code instructions:

1) `00100111 00001010` $\longrightarrow$ `ADD 10`
2) `D2 13` $\longrightarrow$ `JMP 19`

Example: assembly language vs. machine code

Here is the assembly language that might be equivalent to the previous machine code instructions:

1) `00100111 00001010` $\longrightarrow$ `ADD 10`
2) `D2 13` $\longrightarrow$ `JMP 19`

- Would you hasard a guess what these assembly instructions do now ?
- How much easier is it to write, read, understand, and maintain?

Remember: The CPU only executes machine code. Assembly language is a human-readable way to write the same instructions, and an assembler translates between them.

Remember: The CPU only executes machine code. Assembly language is a human-readable way to write the same instructions, and an assembler translates between them.

- Machine code: **opcodes** + **operands** encoded in binary/hex
- Assembly: **mnemonics** + **operands/labels** that represent those opcodes
- Assembler: converts the assembly source into machine code so the CPU can run it

One assembly instruction ≈ one machine instruction

- In most cases, there is a near **one-to-one mapping**:
  - one assembly instruction (e.g., `ADD 10`)
  - corresponds to one machine instruction (one opcode + its operand)

One assembly instruction ≈ one machine instruction

- In most cases, there is a near **one-to-one mapping**:
  - one assembly instruction (e.g., `ADD 10`)
  - corresponds to one machine instruction (one opcode + its operand)
- This is why assembly is still **low-level**:
  - you control **registers** (ACC/IX) and **memory addresses** directly
  - you think in terms of individual CPU steps

Contrast: one HLL statement → many low-level steps

– A single high-level statement often needs **multiple** machine/assembly instructions.

High-level idea	**assembly-level steps**
`x = x + 10`	`LDD x`
	`ADD #10`
	`STO x`

Contrast: one HLL statement → many low-level steps

– A single high-level statement often needs **multiple** machine/assembly instructions.

High-level idea	**assembly-level steps**
`x = x + 10`	`LDD x`
	`ADD #10`
	`STO x`

– So HLL is usually **more productive**, but assembly gives **fine control**.

So where does the assembler fit?

– The CPU cannot execute mnemonics like `ADD` or labels like `LOOP`.

So where does the assembler fit?

- The CPU cannot execute mnemonics like `ADD` or labels like `LOOP`.
- The assembler converts your source program into machine code:

`assembly source` $\rightarrow$ `assembler` $\rightarrow$ `machine code (binary/hex)`

So where does the assembler fit?

- The CPU cannot execute mnemonics like `ADD` or labels like `LOOP`.
- The assembler converts your source program into machine code:

`assembly source` → `assembler` → `machine code (binary/hex)`

- It also resolves **labels** into real numeric **addresses**.

Quick check: relationship questions

– Why is assembly language described as **low-level**?

Quick check: relationship questions

- Why is assembly language described as **low-level**?
- Is machine code **portable** between different CPUs? Why/why not?

Quick check: relationship questions

- Why is assembly language described as **low-level**?
- Is machine code **portable** between different CPUs? Why/why not?
- What is the **main job** of an assembler?

Quick check: relationship questions

– Why is assembly language described as **low-level**?
– Is machine code **portable** between different CPUs? Why/why not?
– What is the **main job** of an assembler?
– Why do labels help humans but not the CPU?

Instruction set (for this course)

For CIE questions, we assume a **model CPU** and its **instruction set**, with specifications bellow. The instruction set is given to you during exams, so **you don't need to know it all from memory**. That being said, you still need to **understand** and **use** it.

- Only one general-purpose register: **ACC** (Accumulator)
- One index register: **IX**
- Instructions mostly:
  - load values into ACC / IX
  - do operations using ACC
  - store results back to memory

We will use these assumptions consistently when tracing programs.

Remember: In this topic, the CPU model is simple: ACC does most work, and IX is used for indexed addressing.

Operand formats (literals vs addresses)

In the questions, operands may be written with prefixes:

- `#n` means a **denary literal** (immediate value)
- `Bn...` means a **binary literal**
- `&n...` means a **hex literal**
- `<address>` means a **memory address** (or a **label** that stands for an address)

Operand formats (literals vs addresses)

In the questions, operands may be written with prefixes:

- `#n` means a **denary literal** (immediate value)
- `Bn...` means a **binary literal**
- `&n...` means a **hex literal**
- `<address>` means a **memory address** (or a **label** that stands for an address)

Examples:

- `LDM #10` loads the literal 10 into ACC
- `ADD B00001010` adds the literal 10 (binary) to ACC
- `SUB &0A` subtracts the literal 10 (hex) from ACC
- `LDD 200` loads `memory[200]` into ACC

These prefixes are a quick way to tell whether an operand is **data** or an **address**.

Remember: Prefixes like `#`, `B`, `&` mean “literal value”. No prefix means “use memory at this address/label”.

Before moving on, let's take some time to read the instruction set provided for this course. Try to familiarise yourself with the mnemonics and their purposes.

Do you see any logic or patterns in how the instructions are named or grouped?

Instruction groups (categories)

To trace and explain programs, it helps to group instructions by purpose:

– **Data movement** (load/store/move): `LDM`, `LDD`, `LDI`, `LDX`, `LDR`, `MOV`, `STO`

Instruction groups (categories)

To trace and explain programs, it helps to group instructions by purpose:

- **Data movement** (load/store/move): `LDM`, `LDD`, `LDI`, `LDX`, `LDR`, `MOV`, `STO`
- **Arithmetic**: `ADD`, `SUB`, `INC`, `DEC`

Instruction groups (categories)

To trace and explain programs, it helps to group instructions by purpose:

- **Data movement** (load/store/move): `LDM`, `LDD`, `LDI`, `LDX`, `LDR`, `MOV`, `STO`
- **Arithmetic**: `ADD`, `SUB`, `INC`, `DEC`
- **Bitwise / shifts**: `AND`, `OR`, `XOR`, `LSL`, `LSR`

Instruction groups (categories)

To trace and explain programs, it helps to group instructions by purpose:

- **Data movement** (load/store/move): `LDM`, `LDD`, `LDI`, `LDX`, `LDR`, `MOV`, `STO`
- **Arithmetic**: `ADD`, `SUB`, `INC`, `DEC`
- **Bitwise / shifts**: `AND`, `OR`, `XOR`, `LSL`, `LSR`
- **Compare + branch**: `CMP`, `CMI` then `JPE`/`JPN`

Instruction groups (categories)

To trace and explain programs, it helps to group instructions by purpose:

- **Data movement** (load/store/move): `LDM`, `LDD`, `LDI`, `LDX`, `LDR`, `MOV`, `STO`
- **Arithmetic**: `ADD`, `SUB`, `INC`, `DEC`
- **Bitwise / shifts**: `AND`, `OR`, `XOR`, `LSL`, `LSR`
- **Compare + branch**: `CMP`, `CMI` then `JPE/JPN`
- **Input/Output**: `IN`, `OUT`

Knowing the group tells you what to look for in a trace table.

Remember: Instruction groups help you predict what changes: ACC/IX, memory writes, output, or control flow.

Compare and conditional jump are a pair

In this model, `CMP/CMI` produce a compare result that the jump just after it uses.

- `CMP <address>` or `CMP #n`: compares ACC with a value
- `JPE <address>`: jump if compare was **True**
- `JPN <address>`: jump if compare was **False**

Compare and conditional jump are a pair

In this model, `CMP/CMI` produce a compare result that the jump just after it uses.

- `CMP <address>` or `CMP #n`: compares ACC with a value
- `JPE <address>`: jump if compare was **True**
- `JPN <address>`: jump if compare was **False**

Typical pattern:

```
CMP #0
JPE ZERO
...
ZERO:
...
```

Common mistakes to avoid

– Don't confuse `#n` (a literal) with `<address>` (a memory location).

Common mistakes to avoid

- Don't confuse `#n` (a literal) with `<address>` (a memory location).
- Don't confuse `LDI` (**indirect**) with `LDM #n` (**immediate**) or `LDX <address>` (**indexed**).

Common mistakes to avoid

- Don't confuse `#n` (a literal) with `<address>` (a memory location).
- Don't confuse `LDI` (**indirect**) with `LDM #n` (**immediate**) or `LDX <address>` (**indexed**).
- Always link `CMP/CMI` to the jump after it: `JPE/JPN` depends on the compare result.

Remember: Most tracing mistakes come from misreading the operand: literal vs address vs effective address.

Mini-drill: read the instruction

For each instruction, write: **(1)** what group it belongs to,**(2)** whether the operand is a **literal** or an **address**, and **(3)** whether the ACC, memory, or none is changed.

- `LDM #25`
- `LDD 200`
- `ADD &0A`
- `STO 250`
- `CMP #0`
- `JPE END`
- `OUT`

Mini-drill answers

- `LDM #25`: data movement; **literal**; changes **ACC**

Mini-drill answers

- `LDM #25`: data movement; **literal**; changes **ACC**
- `LDD 200`: data movement; **address**; changes **ACC**

Mini-drill answers

– `LDM #25`: data movement; **literal**; changes **ACC**
– `LDD 200`: data movement; **address**; changes **ACC**
– `ADD &0A`: arithmetic; **literal**; changes **ACC**

Mini-drill answers

- `LDM #25`: data movement; **literal**; changes **ACC**
- `LDD 200`: data movement; **address**; changes **ACC**
- `ADD &0A`: arithmetic; **literal**; changes **ACC**
- `STO 250`: data movement; **address**; changes **memory**

Mini-drill answers

- `LDM #25`: data movement; **literal**; changes **ACC**
- `LDD 200`: data movement; **address**; changes **ACC**
- `ADD &0A`: arithmetic; **literal**; changes **ACC**
- `STO 250`: data movement; **address**; changes **memory**
- `CMP #0`: compare + branch; **literal**; changes **none**

Mini-drill answers

- `LDM #25`: data movement; **literal**; changes **ACC**
- `LDD 200`: data movement; **address**; changes **ACC**
- `ADD &0A`: arithmetic; **literal**; changes **ACC**
- `STO 250`: data movement; **address**; changes **memory**
- `CMP #0`: compare + branch; **literal**; changes **none**
- `JPE END`: compare + branch / control flow; **address (label)**; changes **none**

Mini-drill answers

- `LDM #25`: data movement; **literal**; changes **ACC**
- `LDD 200`: data movement; **address**; changes **ACC**
- `ADD &0A`: arithmetic; **literal**; changes **ACC**
- `STO 250`: data movement; **address**; changes **memory**
- `CMP #0`: compare + branch; **literal**; changes **none**
- `JPE END`: compare + branch / control flow; **address (label)**; changes **none**
- `OUT`: input/output; **no operand**; changes **none** (but produces output)

Mini-drill: mnemonic patterns

Most mnemonics give you a clue about what they do.

- `LD_` usually means **load** into a register
- `ST_` usually means **store** from ACC into memory
- `IN/OUT` are **I**/**O**
- `CMP` means **compare** (used with `JPE/JPN`)

Remember: Use the instruction group and operand format to quickly predict what an instruction will do.

Addressing modes (how to read an operand)

When you see an instruction like `LDD 200`, always ask:

- What is written in the instruction (literal, address, or something else)?

Addressing modes (how to read an operand)

When you see an instruction like `LDD 200`, always ask:

– What is written in the instruction (literal, address, or something else)?

– What is the **effective address** (the real memory address used)?

Addressing modes (how to read an operand)

When you see an instruction like `LDD 200`, always ask:

- What is written in the instruction (literal, address, or something else)?
- What is the **effective address** (the real memory address used)?
- What value ends up in **ACC**?

Some modes use a direct address, some compute an address, and some follow a pointer.

Remember: Addressing mode tells you how to find the effective address (and therefore which value is used).

Immediate addressing

- Example: `LDM #10`
- Meaning: load the **literal** value 10 into ACC
- Effective address: **not used** (no memory lookup)

Immediate addressing

- Example: `LDM #10`
- Meaning: load the **literal** value 10 into ACC
- Effective address: **not used** (no memory lookup)

Result: ACC $\leftarrow$ 10

Immediate means the operand is the data.

Remember: Immediate addressing: ACC gets the literal value; memory is not read.

Direct addressing

- Example: `LDD 200`
- Meaning: use address 200 directly; load `memory[200]` into ACC
- Effective address: $EA = 200$

Direct addressing

- Example: `LDD 200`
- Meaning: use address 200 directly; load `memory[200]` into ACC
- Effective address: $EA = 200$

Example memory: `memory[200]   37`

Result: ACC $\leftarrow 37$

Direct means the operand is the address of the data.

Remember: Direct addressing: use the operand as the effective address.

Indirect addressing (`LDI`)

- Example: `LDI 300`
- Meaning: address 300 holds another address (a pointer)
- Step 1: pointer $\leftarrow$ `memory[300]`
- Step 2: ACC $\leftarrow$ `memory[pointer]`

Why do we need indirect addressing?

- Sometimes you don't know the real data address in advance.
- Instead, you store the **address** of the data in memory (a **pointer**).
- Then the instruction follows that pointer to reach the data.

Why do we need indirect addressing?

– Sometimes you don't know the real data address in advance.
– Instead, you store the **address** of the data in memory (a **pointer**).
– Then the instruction follows that pointer to reach the data.

Example: reading the first element of an array
Suppose an array starts at address 500, and the value 500 is stored at address 300:

`memory[300]`	`500`	(array start address)
`memory[500]`	`42`	(array[0])

Then `LDI 300` loads `memory[memory[300]]` = `memory[500]` = `42` into ACC.

Remember: Indirect addressing means: the operand points to an address that points to the data.

Indexed addressing (`LDX`)

- Example: `LDX 200`
- Meaning: effective address = `200 + IX`
- Then load `memory[200 + IX]` into ACC

Indexed addressing (`LDX`)

- Example: `LDX 200`
- Meaning: effective address = `200 + IX`
- Then load `memory[200 + IX]` into ACC

Example: if IX = 3 and `memory[203] = 99`, then `LDX 200` loads 99.

Indexed means you compute the effective address using the index register.

$$\boxed{\textbf{Remember: } \text{Indexed addressing: } EA = \text{base} + IX.}$$

Relative addressing (conceptual)

Some CPUs store jumps as an **offset** from the current instruction location.

- Direct jump (absolute): `JMP 120` means go to address 120
- Relative jump (offset): `JMP +3` means go forward 3 instructions/bytes

Relative addressing (conceptual)

Some CPUs store jumps as an **offset** from the current instruction location.

- Direct jump (absolute): `JMP 120` means go to address 120
- Relative jump (offset): `JMP +3` means go forward 3 instructions/bytes

In CIE questions, if they say **relative**, treat the operand as an **offset**, not an absolute address.

Relative addressing is about offsets.

Remember: Relative addressing: target = current location + offset.

Micro-drill: effective address and ACC

Assume IX = 4 and the memory table is:

`memory[200]`	`11`	`memory[210]`	`500`
`memory[204]`	`77`	`memory[500]`	`42`

For each instruction, write the effective address (if any) and the final ACC value:

- `LDM #10`
- `LDD 200`
- `LDI 210`
- `LDX 200`

Micro-drill answers

– `LDM #10`: EA = — ; ACC $\longleftarrow$ 10

Micro-drill answers

- `LDM #10`: EA = — ; ACC $\longleftarrow$ 10
- `LDD 200`: EA = 200 ; ACC $\longleftarrow$ `memory[200]` = 11

Micro-drill answers

- `LDM #10`: EA = — ; ACC $\longleftarrow$ 10
- `LDD 200`: EA = 200 ; ACC $\longleftarrow$ `memory[200]` = 11
- `LDI 210`: EA = `memory[210]` = 500 ;
  ACC $\longleftarrow$ `memory[500]` = 42

Micro-drill answers

- `LDM #10`: EA $=$ — ; ACC $\longleftarrow 10$
- `LDD 200`: EA $= 200$; ACC $\longleftarrow$ `memory[200]` $= 11$
- `LDI 210`: EA $=$ `memory[210]` $= 500$;
  ACC $\longleftarrow$ `memory[500]` $= 42$
- `LDX 200`: EA $= 200 + IX = 204$;
  ACC $\longleftarrow$ `memory[204]` $= 77$

Tracing a simple assembly program

In exam questions, you are often asked to **trace** a program step-by-step.

You are often given a trace table with these columns (not always all of them):

- Step / line number
- Instruction
- ACC
- IX
- Memory changes (address/value)
- Compare result/flag (if used)
- Output (if any)

Tracing exercise (swap two values)

Initial state:

- ACC $= 0$ IX $= 0$
- `memory[200] = 5` `memory[201] = 9` `memory[202] = 0`

Program:

```
LDD 200
STO 202
LDD 201
STO 200
LDD 202
STO 201
END
```

Task: complete a trace table (ACC/IX + memory changes).

Tracing exercise (table)

Step	Instruction	ACC	IX	Memory changes	CMP	OUT
1	`LDD 200`				–	–
2	`STO 202`				–	–
3	`LDD 201`				–	–
4	`STO 200`				–	–
5	`LDD 202`				–	–
6	`STO 201`				–	–
7	`END`				–	–

Tracing exercise (answers)

Step	Instruction	ACC	IX	Memory changes	CMP	OUT
1	`LDD 200`	5	0	–	–	–
2	`STO 202`	5	0	`memory[202] = 5`	–	–
3	`LDD 201`	9	0	–	–	–
4	`STO 200`	9	0	`memory[200] = 9`	–	–
5	`LDD 202`	5	0	–	–	–
6	`STO 201`	5	0	`memory[201] = 5`	–	–
7	`END`	5	0	–	–	–

Typical tracing pitfalls

- Confusing `#n` (literal) with `<address>` (memory reference).

Typical tracing pitfalls

- Confusing `#n` (literal) with `<address>` (memory reference).
- Forgetting that `STO <address>` writes ACC **into memory**.

Typical tracing pitfalls

- Confusing `#n` (literal) with `<address>` (memory reference).
- Forgetting that `STO <address>` writes ACC **into memory**.
- Treating `LDI` as “load immediate”(it is **load indirect** here).

Typical tracing pitfalls

- Confusing `#n` (literal) with `<address>` (memory reference).
- Forgetting that `STO <address>` writes ACC **into memory**.
- Treating `LDI` as “load immediate”(it is **load indirect** here).
- Forgetting to update IX after `LDX #n` using `INC IX`/`DEC IX` when writing loops.

If you avoid these, your trace tables become much more reliable.

Remember: Most tracing errors are operand mistakes or missing state updates (ACC/IX/memory).

Common program patterns to trace

- Swap two memory locations
- Sum an array (usually with indexed addressing)
- Read characters until a sentinel value (I/O + compare + jump)

Tracing questions repeat these patterns a lot.

Remember: Learn the tracing method once, then apply it to common patterns (swap, loop, array, sentinel).

Practice makes perfect 1

Initial state:

- ACC = 0 IX = 0
- `memory[300] = 3` (loop counter)
- `memory[301] = 0` (running total)

Trace the program and give the final value of `memory[301]`.

```
LOOP: LDD 301
ADD #2
STO 301
LDD 300
DEC ACC
STO 300
CMP #0
JPN LOOP
END
```

step	instruction	ACC	memory[300]	memory[301]
0	-	0	3	0
1	`LDD 301`	0	3	0
2	`ADD #2`	2	3	0
3	`STO 301`	2	3	2
4	`LDD 300`	3	3	2
5	`DEC ACC`	2	3	2
6	`STO 300`	2	2	2
7	`CMP #0`	2	2	2
8	`JPN LOOP`	2	2	2
9	`LDD 301`	2	2	2
10	`ADD #2`	4	2	2
11	`STO 301`	4	2	4
12	`LDD 300`	2	2	4
13	`DEC ACC`	1	2	4
14	`STO 300`	1	1	4
15	`CMP #0`	1	1	4
16	`JPN LOOP`	1	1	4
17	`LDD 301`	4	1	4
18	`ADD #2`	6	1	4
19	`STO 301`	6	1	6
20	`LDD 300`	1	1	6
21	`DEC ACC`	0	1	6
22	`STO 300`	0	0	6
23	`CMP #0`	0	0	6
24	`JPN LOOP`	0	0	6
25	`END`	0	0	6

Practice makes perfect 2

Initial state:

- ACC = 0 IX = 0
- Array: `memory[200]=4`, `memory[201]=1`, `memory[202]=7`, `memory[203]=2`
- `memory[210] = 0` (sum)
- `memory[211] = 4` (count)
- `memory[212] = 0` (temp)

Trace the program and give the final value of `memory[210]`.

```
LDR #0
LOOP: LDX 200
STO 212
LDD 210
ADD 212
STO 210
INC IX
LDD 211
DEC ACC
STO 211
CMP #0
JPN LOOP
END
```

Practice makes perfect 2 (answer trace table)

Each pass adds one array element into `memory[210]` and decrements `memory[211]`.

Step	Instruction	ACC	IX	[210]	[211]	[212]
1	`LDR #0`	0	0	0	4	0
2	`LDX 200`	4	0	0	4	0
3	`STO 212`	4	0	0	4	4
4	`LDD 210`	0	0	0	4	4
5	`ADD 212`	4	0	0	4	4
6	`STO 210`	4	0	4	4	4
7	`INC IX`	4	1	4	4	4
8	`LDD 211`	4	1	4	4	4
9	`DEC ACC`	3	1	4	4	4
10	`STO 211`	3	1	4	3	4
11	`CMP #0`	3	1	4	3	4
12	`JPN LOOP`	3	1	4	3	4
13	`LDX 200`	1	1	4	3	4
14	`STO 212`	1	1	4	3	1
15	`LDD 210`	4	1	4	3	1

Practice makes perfect 2 (answer trace table)

Step	Instruction	ACC	IX	[210]	[211]	[212]
16	`ADD 212`	5	1	4	3	1
17	`STO 210`	5	1	5	3	1
18	`INC IX`	5	2	5	3	1
19	`LDD 211`	3	2	5	3	1
20	`DEC ACC`	2	2	5	3	1
21	`STO 211`	2	2	5	2	1
22	`CMP #0`	2	2	5	2	1
23	`JPN LOOP`	2	2	5	2	1
24	`LDX 200`	7	2	5	2	1
25	`STO 212`	7	2	5	2	7
26	`LDD 210`	5	2	5	2	7
27	`ADD 212`	12	2	5	2	7
28	`STO 210`	12	2	12	2	7
29	`INC IX`	12	3	12	2	7
30	`LDD 211`	2	3	12	2	7
31	`DEC ACC`	1	3	12	2	7
32	`STO 211`	1	3	12	1	7
33	`CMP #0`	1	3	12	1	7
34	`JPN LOOP`	1	3	12	1	7

Practice makes perfect 2 (answer trace table)

Step	**Instruction**	**ACC**	**IX**	**[210]**	**[211]**	**[212]**
35	`LDX 200`	2	3	12	1	7
36	`STO 212`	2	3	12	1	2
37	`LDD 210`	12	3	12	1	2
38	`ADD 212`	14	3	12	1	2
39	`STO 210`	14	3	14	1	2
40	`INC IX`	14	4	14	1	2
41	`LDD 211`	1	4	14	1	2
42	`DEC ACC`	0	4	14	1	2
43	`STO 211`	0	4	14	0	2
44	`CMP #0`	0	4	14	0	2
45	`JPN LOOP`	0	4	14	0	2
46	`END`	0	4	14	0	2

Final: `memory[210] = 14`

Practice makes perfect 3

Initial state:

- ACC = 0   IX = 0
- `memory[300] = 310` (pointer to a pointer)
- `memory[310] = 500` (start address of array)
- Array with sentinel 0: `memory[500]=7`, `memory[501]=2`, `memory[502]=9`, `memory[503]=0`
- `memory[400] = 0` (sum)
- `memory[401] = 0` (temp)

```
LDI 300
MOV IX
LOOP: LDX 0
CMP #0
JPE DONE
STO 401
LDD 400
ADD 401
STO 400
INC IX
JMP LOOP
DONE: LDD 400
OUT

END
```

Practice makes perfect 3 (answer trace table)

`LDI 300` loads 500 into ACC, then `MOV IX` sets IX to 500.

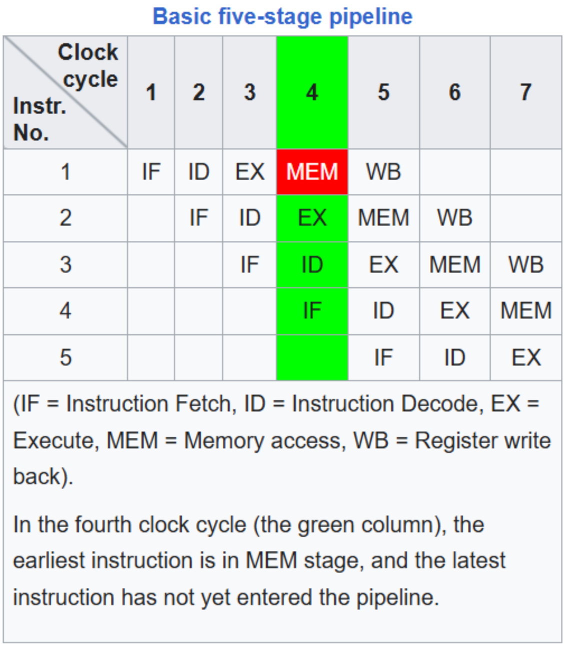
Pass	**IX**	**Value read (`LDX 0`)**	**memory[400] (sum)**	**JPE DONE?**	**Next IX**
Start	500	–	0	–	–
1	500	7	7	no	501
2	501	2	9	no	502
3	502	9	18	no	503
4	503	0	18	yes	(stop)

Final: `memory[400] = 18`. `OUT` outputs the ASCII character with code 18.

Outline

Instruction pipelining

- Modern CPUs use **pipelining** to improve performance.
- Different stages of instruction execution are overlapped.
- While one instruction is being executed, the next instruction is being decoded, and the one after that is being fetched.



Basic five-stage pipeline

Clock cycle / Instr. No.	1	2	3	4	5	6	7
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX

(IF = Instruction Fetch, ID = Instruction Decode, EX = Execute, MEM = Memory access, WB = Register write back).

In the fourth clock cycle (the green column), the earliest instruction is in MEM stage, and the latest instruction has not yet entered the pipeline.

Remember: Pipelining increases instruction throughput (more instructions per unit time) but does not reduce the latency of a single instruction.

Remember: Past exams have asked about broad descriptions of pipelining and its benefits, or asked to fill a pipeline diagram as above.

Outline

Two-pass assembler: why do we need two passes?

We use a **two-pass assembler** when:

- we allow **labels** (symbolic addresses), and
- we allow **forward references** (jumping to a label defined later).

Two-pass assembler: why do we need two passes?

We use a **two-pass assembler** when:

- we allow **labels** (symbolic addresses), and
- we allow **forward references** (jumping to a label defined later).

So the assembler needs one scan to learn addresses, and another scan to generate final machine code.

Labels are great for humans, but the CPU needs numeric addresses.

Remember: Pass 1 builds the symbol table; Pass 2 uses it to output machine code.

Pass 1: symbol table + address assignment

What the assembler does in Pass 1:

- scans the program line-by-line
- keeps a **location counter** (the address of the current instruction)
- when it sees a **label**, it records it:
  - label name $\rightarrow$ address

Pass 1: symbol table + address assignment

What the assembler does in Pass 1:

- scans the program line-by-line
- keeps a **location counter** (the address of the current instruction)
- when it sees a **label**, it records it:
  - label name → address

Output of Pass 1:

- a **symbol table**
- often an annotated listing (each line with its address)

Pass 1 is about **finding addresses**, not generating final machine code.

Remember: Pass 1 assigns addresses and records labels in the symbol table.

Pass 2: translation + resolution

What the assembler does in Pass 2:

- scans the program again
- translates each mnemonic + addressing mode into an **opcode**
- replaces label operands with **numeric addresses** using the symbol table
- outputs the final machine code

Pass 2 is the pass that produces the output that the CPU can run.

Remember: Pass 2 converts mnemonics to opcodes and labels to numeric addresses.

Forward reference example (what two passes solve)

Consider this program (assume the first instruction is at address 0 and each instruction is 1 address):

```
        LDM #0
        JMP END_LABEL
        LDM #99
END_LABEL:
        ADD #1
        END
```

Forward reference example (what two passes solve)

Consider this program (assume the first instruction is at address 0 and each instruction is 1 address):

```
        LDM #0
        JMP END_LABEL
        LDM #99
END_LABEL:
        ADD #1
        END
```

– END_LABEL is a **forward reference**: it is used before it is defined.

– Pass 1 finds END_LABEL's address; Pass 2 can then encode the earlier JMP.

Mini-task: build the symbol table (Pass 1)

Assume address 0, and each instruction is 1 address.

Address	**Label**	**Opcode**	**Operand**
0	–	LDM	#0
1	–	JMP	END_LABEL
2	–	LDM	#99
3	END_LABEL	ADD	#1
4	–	END	–

Mini-task: build the symbol table (Pass 1)

Assume address 0, and each instruction is 1 address.

Address	**Label**	**Opcode**	**Operand**
0	–	LDM	#0
1	–	JMP	END_LABEL
2	–	LDM	#99
3	END_LABEL	ADD	#1
4	–	END	–

Fill the symbol table: END_LABEL $\rightarrow$?

Mini-task answers (Pass 1 + Pass 2 resolution)

– Symbol table: `END_LABEL` $\rightarrow 3$

Mini-task answers (Pass 1 + Pass 2 resolution)

- Symbol table: `END_LABEL` $\rightarrow 3$
- Resolved (Pass 2): `JMP END_LABEL` becomes `JMP 3`

Address	Instruction	Resolved operand	Output format
0	`LDM #0`	`#0`	`OP operand`
1	`JMP END_LABEL`	`3`	`OP operand`
2	`LDM #99`	`#99`	`OP operand`
3	`ADD #1`	`#1`	`OP operand`
4	`END`	–	`OP`

Outline

Bit manipulation (assembly focus)

WE already studied:

- bitwise operations and masking (AND/OR/XOR)
- testing/setting/clearing bits using masks
- binary shifts (logical / arithmetic / cyclic)

Bit manipulation (assembly focus)

WE already studied:

- bitwise operations and masking (AND/OR/XOR)
- testing/setting/clearing bits using masks
- binary shifts (logical / arithmetic / cyclic)

Now, let's look at what those ideas look like in **assembly**:

- bitwise/shift instructions act on **ACC** (result stored back in **ACC**)
- the mask can be a **literal** (e.g. `B01000000`) or loaded from memory
- this is commonly used for **monitor/control** patterns (device/status bits)

Remember: In assembly, masking is: load into ACC $\rightarrow$ AND/OR/XOR $\rightarrow$ store back if needed.

Bit manipulation: quick review questions

– What does `AND mask` typically do in a masking task?

Bit manipulation: quick review questions

- What does `AND mask` typically do in a masking task?
- What is the difference between using `OR mask` and `XOR mask` on a bit?

Bit manipulation: quick review questions

– What does `AND mask` typically do in a masking task?
– What is the difference between using `OR mask` and `XOR mask` on a bit?
– How would you **test** whether bit 0 (LSB) is set, using only `AND` and `CMP`?

Bit manipulation: quick review questions

- What does `AND mask` typically do in a masking task?
- What is the difference between using `OR mask` and `XOR mask` on a bit?
- How would you **test** whether bit 0 (LSB) is set, using only `AND` and `CMP`?
- If ACC is `B00010110`, what is ACC after `LSR #1`?

Bit manipulation: quick review questions

- What does `AND mask` typically do in a masking task?
- What is the difference between using `OR mask` and `XOR mask` on a bit?
- How would you **test** whether bit 0 (LSB) is set, using only `AND` and `CMP`?
- If ACC is `B00010110`, what is ACC after `LSR #1`?
- If ACC is `B00010110`, what is ACC after `LSL #2`?

Check your notes for full details (bit positions, truth tables, and shift behaviour).

Practice: test and set a bit (masking)

Initial state:

- ACC = 0   IX = 0
- `memory[600] = B10110110`

Task:

- The program tests bit 0 of `memory[600]`.
- If bit 0 is 1, it sets bit 7. If bit 0 is 0, it clears bit 7.
- Trace the key steps and give the final value of `memory[600]`.

```
LDD 600       LDD 600       ZERO:         DONE:
AND B00000001 OR  B10000000 LDD 600       END
CMP #0        STO 600       AND B01111111
JPE ZERO      JMP DONE      STO 600
```

Practice answers (masking)

– `memory[600] = B10110110` so bit 0 is **0**.

Practice answers (masking)

- `memory[600] = B10110110` so bit 0 is **0**.
- After `AND B00000001`, ACC becomes `B00000000`.

Practice answers (masking)

– `memory[600] = B10110110` so bit 0 is **0**.
– After `AND B00000001`, ACC becomes `B00000000`.
– `CMP #0` is true, so `JPE ZERO` is taken.

Practice answers (masking)

- `memory[600] = B10110110` so bit 0 is **0**.
- After `AND B00000001`, ACC becomes `B00000000`.
- `CMP #0` is true, so `JPE ZERO` is taken.
- `AND B01111111` clears bit 7, so `B10110110` becomes `B00110110`.

Practice answers (masking)

- `memory[600] = B10110110` so bit 0 is **0**.
- After `AND B00000001`, ACC becomes `B00000000`.
- `CMP #0` is true, so `JPE ZERO` is taken.
- `AND B01111111` clears bit 7, so `B10110110` becomes `B00110110`.
- Final: `memory[600] = B00110110` (denary 54)

Outline

Language translators (quick recap)

– **Assembler**: translates **assembly** → **machine code**

Language translators (quick recap)

- **Assembler**: translates **assembly** $\rightarrow$ **machine code**
- **Compiler**: translates a whole **high-level language** program **before** it runs

Language translators (quick recap)

– **Assembler**: translates **assembly** $\rightarrow$ **machine code**
– **Compiler**: translates a whole **high-level language** program **before** it runs
– **Interpreter**: translates and executes a high-level language program **line-by-line** / statement-by-statement

Remember: CPU runs machine code; translators convert code into a runnable form.

Compiler vs interpreter (compare)

Compiler	**Interpreter**
Translates the whole program first	Translates while running (line-by-line)
Usually faster execution	Usually slower execution
Many errors found before running	Many errors appear at runtime
Produces platform-specific output (often)	Often portable source, needs interpreter

Compiler vs interpreter (compare)

Compiler	**Interpreter**
Translates the whole program first	Translates while running (line-by-line)
Usually faster execution	Usually slower execution
Many errors found before running	Many errors appear at runtime
Produces platform-specific output (often)	Often portable source, needs interpreter

Remember: Compiler: translate then run. Interpreter: translate as you run.

Hybrid translation (real systems)

Many languages use a mix of both approaches.

- Example: **Java** compiles source code to **bytecode**
- Bytecode is then run by a virtual machine (often interpreted and/or JIT-compiled)

IDE features to recognise (exam list)

An IDE is more than a text editor. Common features include:

- context-sensitive prompts / autocomplete
- syntax highlighting + linting
- dynamic checks / live error detection
- pretty-print / formatting / folding (collapse code blocks)
- debugger: breakpoints, single stepping, watch expressions/variables

Remember: IDE features help you write code faster and debug more reliably.

Practice: translators + IDE

– Why is an **assembler** needed for assembly programs?

Practice: translators + IDE

- Why is an **assembler** needed for assembly programs?
- Give one benefit and one drawback of a **compiler**.

Practice: translators + IDE

- Why is an **assembler** needed for assembly programs?
- Give one benefit and one drawback of a **compiler**.
- Give one benefit and one drawback of an **interpreter**.

Practice: translators + IDE

- Why is an **assembler** needed for assembly programs?
- Give one benefit and one drawback of a **compiler**.
- Give one benefit and one drawback of an **interpreter**.
- Name **three** IDE features (from the syllabus list) that help with debugging or error detection.

Practice answers (translators + IDE)

– Assembler needed because CPU runs **machine code**, not mnemonics/labels; assembler translates and resolves labels.

Practice answers (translators + IDE)

– Assembler needed because CPU runs **machine code**, not mnemonics/labels; assembler translates and resolves labels.
– Compiler (benefit): faster execution; many errors found before running. (drawback): compile step; platform-specific output.

Practice answers (translators + IDE)

- Assembler needed because CPU runs **machine code**, not mnemonics/labels; assembler translates and resolves labels.
- Compiler (benefit): faster execution; many errors found before running. (drawback): compile step; platform-specific output.
- Interpreter (benefit): easier debugging/rapid iteration; portable source. (drawback): slower; errors often appear at runtime.

Practice answers (translators + IDE)

- Assembler needed because CPU runs **machine code**, not mnemonics/labels; assembler translates and resolves labels.
- Compiler (benefit): faster execution; many errors found before running. (drawback): compile step; platform-specific output.
- Interpreter (benefit): easier debugging/rapid iteration; portable source. (drawback): slower; errors often appear at runtime.
- IDE features (examples): syntax highlighting/linting/live errors; debugger (breakpoints, single stepping, watch).