4 Processor Fundamentals – Part 2

Name: _____ Class: _____ Date: _____

Vocabulary

Write each word three times.

English	**Write it 3 times** (English)		
machine code			
Assembly language			
mnemonics			
assembler			
opcode			
operand			
addressing mode			
immediate addressing			
direct addressing			
indexed addressing			
logical shift			
mask			

Recall

In **Part 1** you met the Von Neumann CPU (ALU, control unit, registers like the PC and accumulator), the fetch–execute cycle, and the three buses. Part 2 looks at the language the CPU runs, how it finds its data, and working on single bits – the level a real processor

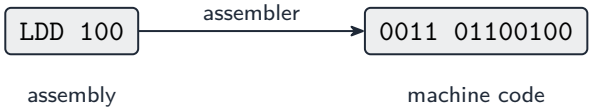
actually works at.

New ideas

Read these first. Each idea has a short worked example. Then do the Practice.

1 Machine code and assembly language

The CPU actually runs **machine code** – bit patterns. **Assembly language** is a readable form with one instruction each, written using short **mnemonics** like `LDD`, `ADD`, `JMP`. An **assembler** translates the assembly into machine code.

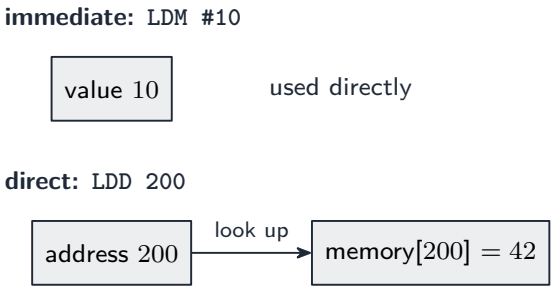


Worked example. The assembler turns the readable `ADD 100` into a pattern of bits: an **opcode** part that means "ADD", and an **operand** part that holds 100.

2 Addressing modes

The **addressing mode** says how the CPU finds the operand of an instruction:

- **immediate addressing** – the value is *in* the instruction (`LDM #10` loads the number 10);
- **direct addressing** – the instruction holds an *address*; the operand is the value stored there (`LDD 200`);
- **indexed addressing** – the address is a base plus an index register, used to step through an array.



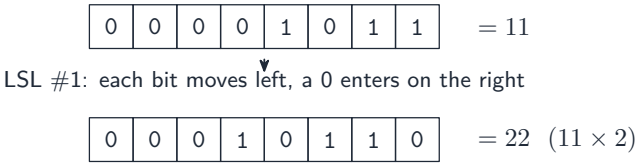
Worked example. Suppose memory location 10 holds the value 99. Then `LDM #10` loads 10 (the number written in the instruction), but `LDD 10` loads 99 (the value found at address 10) – same number, very different result.

Watch out: the `#` changes everything. `LDM #10` uses the number 10; `LDD 10` uses whatever is *stored at* address 10.

3 Binary shifts and bit masks

A **logical shift** moves all the bits left or right, filling the empty places with 0:

- left by 1 → multiply by 2; right by 1 → integer divide by 2.



A **mask** lets you change one chosen bit: **OR** with a mask **sets** a bit, **AND** with a mask **clears** a bit, **XOR** with a mask **toggles** a bit.

Worked example. To set bit 0 of `01000000` without touching the rest, OR it with the mask `00000001`: the result is `01000001`.

Practice

4.1 State what an assembler does. [1]

4.2 State which bitwise operation (AND, OR or XOR with a mask) is used to **set** a chosen bit to 1. [1]

4.3 State what each instruction loads: (a) `LDM #10`, (b) `LDD 200`. [2]

4.4 Apply `LSL #1` (left shift by one) to `00001011`. Give the result and the arithmetic operation it performs. [2]

exercises.lol exercises.com.cn

4.5 Apply `LSR #1` (right shift by one) to `00011000`. Give the result and the operation it performs. [2]

4.6 Explain *why* a logical left shift by one is the same as multiplying by 2. [2]

4.7 Challenge. Show how to multiply 5 (`00000101`) by 4 using a logical shift. Give the shift used and the final byte. [2]

exercises.lol exercises.com.cn