Week 10

A-Level Computer Science

Homework bundle for this week

本周作业合集

exercises.lol

Contents 目录

A-Level Computer Science Name: _____ Score: _____

1. The stored-program (Von Neumann) idea means that:

   ☐ a single memory holds both the program instructions and the data

   ☐ programs are stored only in the CPU

   ☐ data and instructions use separate memories

   ☐ the program cannot be changed

2. Select **all** the factors that affect CPU performance.

   ☐ clock speed

   ☐ number of cores

   ☐ cache size

   ☐ the colour of the case

   Tick every correct option.

3. Assembly language is:

   ☐ a readable form of machine code, with one instruction per machine instruction

   ☐ a high-level language like Python

   ☐ the same as binary data

   ☐ only used for web pages

4. The 8-bit value ‘00001011‘ (11) is shifted left by 1 (‘LSL #1‘). What is the new denary value?

   _____

5. The ALU is responsible for:

   ☐ arithmetic and logic operations

   ☐ decoding instructions

   ☐ storing the next instruction’s address

   ☐ generating the clock pulses

6. A multi-core CPU is especially helpful for:

   ☐ running several threads/tasks at the same time (parallel work)

   ☐ a single task that cannot be split up

   ☐ reducing the clock speed

   ☐ storing more files

7. What does pass 1 of a two-pass assembler do?

   ☐ builds a symbol table recording each label's address

   ☐ runs the program

   ☐ generates all the machine code

   ☐ optimises the code

8. Shifting an unsigned number left by 3 places multiplies it by what number?

   _____

9. The _____ unit decodes each instruction and sends the control signals that carry it out.

   _____

10. _____ memory is a small, fast store next to the processor that holds recently used instructions and data, so fewer slow trips to RAM are needed.

    _____

4. Processor Fundamentals

Answers · 答案

A-Level Computer Science

1. **a single memory holds both the program instructions and the data**

   One memory holds instructions and data together; changing the stored program changes what the computer does, with no rewiring.

2. **clock speed; number of cores; cache size**

   Clock speed, cores and cache (plus word size, RAM, storage and bus width) all affect performance. The case colour does not.

3. **a readable form of machine code, with one instruction per machine instruction**

   Assembly uses mnemonics and maps one-to-one to machine code; an assembler translates it.

4. **22**

   A left shift by 1 multiplies by 2: $11 \times 2 = 22$ (‘00010110‘).

5. **arithmetic and logic operations**

   The Arithmetic and Logic Unit performs calculations and logic/comparisons. Decoding is the control unit's job.

6. **running several threads/tasks at the same time (parallel work)**

   Extra cores run tasks in parallel. A purely single-threaded job benefits more from higher per-core speed.

7. **builds a symbol table recording each label's address**

   Pass 1 records where each label is (the symbol table); pass 2 then generates code, using the table to resolve label references.

8. **8**

   Shifting by n places multiplies by 2^n, so by 3 places is $2^3 = 8$.

9. **control**

   The control unit decodes and signals; the ALU does the arithmetic and logic; the clock keeps them in step.

10. **Cache**

    More cache means more of the working set sits next to the processor, which is the reason it appears in every performance comparison.

A-LEVEL COMPUTER SCIENCE • EXERCISE SHEET

4.1 Central Processing Unit (CPU) Architecture

Name: _____ Class: _____ Date: _____ **Total: 36 marks**

KEY WORDS register 寄存器 • von neumann 冯 · 诺依曼 • accumulator 累加器 • control unit 控制单元 • memory data register 内存数据寄存器

Objective

Build the skills to answer exam questions on **CPU architecture** — the Von Neumann model, the registers, the buses, and the fetch–execute cycle.

You must be able to:

- explain the **Von Neumann** 冯 · 诺依曼 model and the **stored-program** 存储程序 concept
- state the role of the **registers**, the **ALU** 算术逻辑单元, the **control unit** 控制单元 and the clock
- explain the **address**, **data** and **control buses**
- describe the stages of the **fetch–execute cycle** and the purpose of **interrupts** 中断

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.

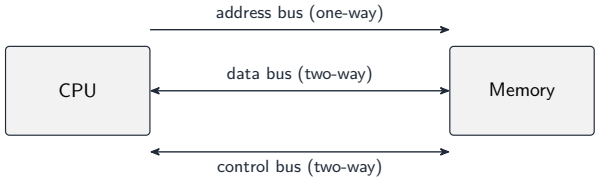
■ Von Neumann model and registers

In the **Von Neumann** model one memory holds **both program and data** (the **stored-program** idea), and the CPU runs the instructions one at a time, in order, unless a branch changes the flow. **Registers** 寄存器 are tiny, very fast stores inside the CPU. Each special-purpose register has a fixed job:

Register	Holds
PC (program counter)	address of the **next** instruction
MAR (memory address register)	the address being read/written
MDR (memory data register)	the data going to/from memory
CIR (current instruction register)	the instruction being decoded
ACC (accumulator)	the value the ALU is working on
IX (index register)	a value added to a base address (indexed addressing)

The **ALU** does arithmetic and logic; the **control unit** decodes instructions and sends control signals; the **clock** keeps everything in step.

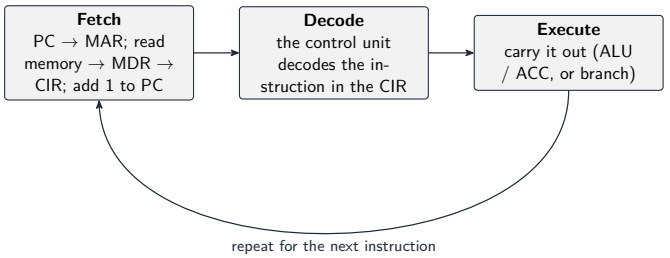
■ The three buses



The **address bus** is **one-way** (the CPU only ever *sends* an address to memory); the **data** and **control** buses are **two-way**. The **width** of a bus matters: a wider **data bus** carries more bits per transfer (faster), and a wider **address bus** can address **more** memory locations (each extra line doubles the number of addresses).

■ The fetch–execute cycle

Every instruction goes through the same three stages, then the cycle repeats:



An **interrupt** is a signal that makes the CPU pause the current program, run a

service routine, then return — so urgent events are handled quickly.

2 Practice

Now apply the methods above.

2.1 State what the **stored-program concept** means. [1]

2.2 State the role of the **PC** and the **MAR**. [2]

2.3 State the difference between a **general-purpose** and a **special-purpose** register. [2]

2.4 State which bus is **one-way**, and explain why. [2]

3 Exam-style questions

3.1 Describe the stages of the **fetch–execute cycle**. [4]

3.2 (a) State the role of the **ALU** and the **control unit**. [2]

(b) State **two** factors that affect the performance of a CPU. [2]

3.3 Explain the purpose of an **interrupt**. [2]

3.4 During the fetch stage, describe the roles of the **MAR** and the **MDR**. [2]

3.5 A computer is redesigned with a **wider address bus**. Explain the effect this has on the computer. [2]

3.6 A processor has four cores, a clock speed of 3.2 GHz and a small amount of cache memory.

(a) **Complete** the table by describing the role of each register. [4]

register	role
Program Counter (PC)	
Memory Address Register (MAR)	
Memory Data Register (MDR)	
Current Instruction Register (CIR)	

(b) **Explain** how increasing the **number of cores** can improve performance, and **give** one reason why four cores rarely make a program four times faster. [3]

(c) **Explain** how increasing the amount of **cache** improves performance. [2]

(d) **State** three differences between **Dynamic RAM** and **Static RAM**, and **identify** which is normally used for cache. [4]

(e) A second processor has the same clock speed but a **wider** data bus. **Explain** the effect on performance. [2]

Solutions

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

2.1

- both the **program (instructions) and the data** are held together in the **same memory**

2.2

- PC holds the address of the **next** instruction
- MAR holds the address currently being read from or written to memory

2.3

- a special-purpose register has a **fixed role** in the cycle (e.g. PC, MAR)
- a general-purpose register holds **temporary values** chosen by the program

2.4

- the **address bus** is one-way
- the CPU only ever **sends** an address to memory — memory never sends an address back

3.1

- **fetch:** the address in the PC is copied to the MAR; the instruction is read from memory into the MDR, then the CIR; the PC is increased by 1
- **decode:** the control unit decodes the instruction in the CIR
- **execute:** the instruction is carried out (ALU/ACC, or a branch)

3.2

(a) the ALU performs **arithmetic and logic** operations

- the control unit **decodes instructions and sends control signals**

(b) any two of: clock speed; number of cores; cache size; bus width

3.3

- an interrupt is a signal that makes the CPU **pause** the current program
- it runs a routine to handle an urgent event, then **returns** to where it left off

3.4

- the MAR holds the **address** of the instruction to fetch
- the MDR holds the **instruction/data** read back from that address

3.5

- a wider address bus has more lines, so it can carry **more distinct addresses**
- the CPU can then directly address **more memory** (each extra line doubles the addressable memory)

3.6

(a) **PC** — holds the address of the **next** instruction to be fetched

- **MAR** — holds the address currently being read from or written to
- **MDR** — holds the data or instruction just fetched from, or about to be written to, that address
- **CIR** — holds the instruction currently being decoded and executed

(b) each core can fetch and execute its own instructions, so several instructions run **at the same time**

- the program must be written to divide its work between cores, and parts of it are inherently sequential
- the cores also compete for the same memory and bus, so the gain is less than the core count suggests

(c) cache holds recently and frequently used instructions and data close to the processor

- it is far faster to access than main memory, so more cache means fewer slow main-memory fetches

(d) any three: DRAM stores each bit in a **capacitor** that must be **refreshed**, SRAM uses flip-flops and does not; SRAM is **faster**; SRAM is more expensive per bit and takes more space, so DRAM gives more capacity for the money

- **SRAM** is used for cache

(e) a wider data bus carries **more bits per transfer**

- each fetch moves more data and fewer transfers are needed for the same work, raising throughput without any change in clock speed

6 **(a)** The processor uses several registers, including the Accumulator (ACC) and the Current Instruction Register (CIR).

Complete the table by describing the role of each register.

Register	Role
ACC	....
CIR	....

[2]

(b) Increasing the number of cores in a processor can affect the performance of a computer.

Describe the drawbacks of increasing the number of cores in a processor.

.....

.....

.....

..... [2]

(c) State **three** differences between Dynamic RAM (DRAM) and Static RAM (SRAM).

1 .....

.....

2 .....

.....

3 .....

.....

[3]

1 **(a)** The table has six statements about the Von Neumann model for a computer system. Three of the statements are incorrect.

Statement number	Statement
1	The Program Counter (PC) stores the next instruction to be fetched from memory.
2	The Arithmetic and Logic Unit (ALU) performs mathematical and logical operations.
3	The Control Unit (CU) sends signals to other components on the data bus.
4	The Memory Data Register (MDR) transfers data to the memory address stored in the Memory Address Register (MAR).
5	The MAR stores an address from memory.
6	The Accumulator (ACC) stores the result of calculations.

Complete the table by writing the **three** incorrect statement numbers and the corrected statements.

Incorrect statement number	Corrected statement
.....	....
.....	....
.....	....

[3]

(b) Registers that are used in the Fetch-Execute (F-E) cycle include the PC, MAR, MDR and the ACC.

Identify **one** other register and describe its role in the Fetch-Execute (F-E) cycle.

Register .....

Role .....

.....

[2]

(c) Explain how an interrupt from an input device will be detected and handled in the F-E cycle.

.....

.....

.....

.....

.....

.....

.....

..... [4]

2 A computer has a processor.

(a) The processor has a Control Unit (CU) and system clock.

(i) Explain how the CU and the system clock work together.

.....

.....

.....

..... [2]

(ii) The processor runs the Fetch-Execute (F-E) cycle.

Write the stages of the F-E cycle using register transfer notation.

.....

.....

.....

.....

.....

..... [4]

(b) The computer has cache memory.

Describe **one** benefit of the computer using cache memory.

.....

.....

.....

..... [2]

(c) The computer connects to a monitor using a High Definition Multimedia Interface (HDMI) cable that connects into an HDMI port.

Explain how HDMI provides connection to peripheral devices.

.....

.....

.....

..... [2]

3 A computer system has a dual-core Central Processing Unit (CPU).

(a) State the purpose of the system clock and the Control Unit (CU) in a CPU.

System clock .....

.....

CU .....

.....

[2]

(b) **(i)** The number of cores in the processor affects the performance of the computer system.

Identify **one other** feature of a processor that can affect the performance of a computer system **and** state why it affects the performance.

Feature .....

.....

Reason .....

.....

[2]

(ii) A solid state (flash) memory drive is automatically recognised by the computer when it is plugged into a port in the computer.

Identify an appropriate type of port to connect the solid state memory drive to the computer.

Explain how this port provides an automatic connection.

Port .....

Explanation .....

.....

.....

.....

[3]

(c) Identify **two** disadvantages of using Dynamic RAM (DRAM) instead of Static RAM (SRAM) in a computer system.

1 .....

.....

2 .....

.....

[2]

(d) The computer system is used to store data received from a temperature sensor every five seconds. The data is stored on an optical disc using an optical disc reader/writer.

(i) Describe the principal operation of an optical disc reader/writer.

.....

.....

.....

.....

.....

.....

.....

..... [4]

(ii) The computer uses a buffer when writing data to the optical disc.

Explain the use of a buffer when writing data to the optical disc.

.....

.....

.....

.....

.....

..... [3]

8 A computer designed using the Von Neumann model for a computer system contains general purpose registers and special purpose registers.

(a) Describe the purpose of the Status Register (SR).

.....

.....

.....

..... [2]

(b) Identify **two** differences between general purpose registers and special purpose registers.

1 .....

.....

.....

2 .....

.....

.....

[2]

3 A student has a computer.

(a) The computer is designed using the Von Neumann model for a computer system.

Complete the table by describing the purpose of each of the given registers.

Register	Purpose
Program Counter (PC)	....
Memory Address Register (MAR)	....
Memory Data Register (MDR)	....
Index Register (IX)	....

[4]

(b) The student needs to connect the computer to a monitor that has a screen resolution of 2560 × 1600 pixels. The monitor also has built-in speakers.

The computer has a Video Graphics Array (VGA) port and a High Definition Multimedia Interface (HDMI) port.

Explain the benefits of connecting the monitor to the computer using the HDMI port instead of the VGA port.

.....

.....

.....

.....

.....

.....

.....

..... [4]

(c) The computer has an Operating System (OS). One of the key management tasks of the OS is process management.

Describe the process management tasks performed by an OS.

.....

.....

.....

.....

.....

.....

.....

..... [4]

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.

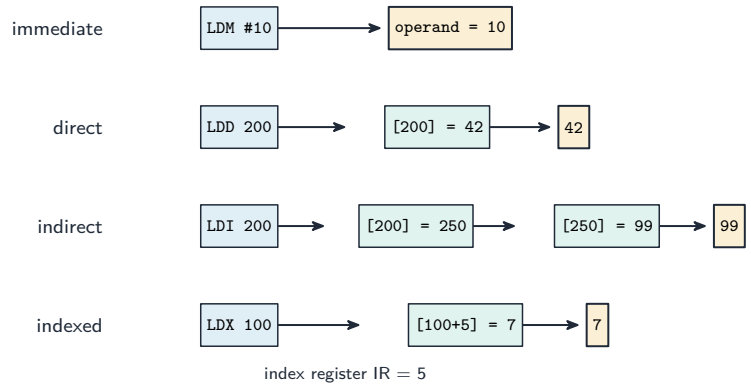
You must be able to:

- 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

4 The table shows part of the instruction set for a processor. The processor has one register, the Accumulator (ACC).

Instruction		**Explanation**
Opcode	**Operand**	
AND	#n / Bn / &n	Bitwise AND operation of the contents of the ACC with the operand
AND	<address>	Bitwise AND operation of the contents of the ACC with the contents of <address>
XOR	#n / Bn / &n	Bitwise XOR operation of the contents of the ACC with the operand
XOR	<address>	Bitwise XOR operation of the contents of the ACC with the contents of <address>
OR	#n / Bn / &n	Bitwise OR operation of the contents of the ACC with the operand
OR	<address>	Bitwise OR operation of the contents of the ACC with the contents of <address>
LSL	#n	Bits in ACC are shifted logically n places to the left. Zeros are introduced on the right-hand end.
LSR	#n	Bits in ACC are shifted logically n places to the right. Zeros are introduced on the left-hand end.
<address> can be an absolute or symbolic address # denotes a denary number, e.g. #127 B denotes a binary number, e.g. B10010001 & denotes a hexadecimal number, e.g. &4A		

(a) The ACC currently contains the following positive binary integer:

0	0	0	1	1	1	1	0

Write a bit manipulation instruction that uses a binary shift to change the contents of the ACC to:

0	1	1	1	1	0	0	0

Instruction ..... [1]

(b) The ACC currently contains the following positive binary integer:

1	1	1	0	0	0	1	1

Write the contents of the ACC after the instruction `XOR &12` is carried out.

[1]

(c) The ACC currently contains the following positive binary integer:

1	1	1	0	0	0	1	1

Write the contents of the ACC after the instruction `AND #63` is carried out.

[1]

(d) The ACC currently contains the following positive binary integer:

1	1	1	0	0	0	1	1

The current contents of memory are:

Address	**Data**
98	00100100
99	00110001
100	00110011
101	10100011
102	10101100

Write the contents of the ACC after the instruction `OR 100` is carried out.

[1]

3 **(a)** **(i)** State what is meant by relative addressing.

.....

.....

..... [1]

(ii) Registers such as the Accumulator (ACC) and the Index Register (IX) are used in the CPU.

Identify **two** special purpose registers used in the CPU. Do **not** include the ACC or IX in your answers.

1 .....

2 .....

[2]

(b) The following table shows part of the instruction set for a processor. The processor has two registers: the ACC and an IX.

Instruction		Explanation
Opcode	**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
<address> can be an absolute or symbolic address # denotes a denary number, e.g. #127		

The current contents of the main memory and the index register are shown.

Address	Instruction
98	8
99	16
100	3
101	98
102	32
IX	2

Write the contents of the ACC after each instruction is executed.

Instruction	Value in ACC
LDM #98	
LDI 101	
LDX 100	

[3]

(c) A student buys a new computer. The table shows the specifications of the old computer and the new computer.

Old computer	New computer
1.8 GHz dual core processor	2.3 GHz dual core processor
16 MB cache	32 MB cache

Explain why increasing the clock speed **and** increasing the cache memory will improve the performance of the computer.

Clock speed .....

.....

.....

.....

Cache memory .....

.....

.....

.....

[4]

6 The following table shows part of the instruction set for a processor. The processor has two registers: the Accumulator (ACC) and the Index Register (IX).

Instruction		Explanation
Opcode	**Operand**	
LDM	#n	Immediate addressing. Load the number n to the ACC
AND	#n / Bn / &n	Bitwise AND operation of the contents of the ACC with the operand
AND	<address>	Bitwise AND operation of the contents of the ACC with the contents of <address>
XOR	#n / Bn / &n	Bitwise XOR operation of the contents of the ACC with the operand
XOR	<address>	Bitwise XOR operation of the contents of the ACC with the contents of <address>
OR	#n / Bn / &n	Bitwise OR operation of the contents of the ACC with the operand
OR	<address>	Bitwise OR operation of the contents of the ACC with the contents of <address>
CMP	#n	Compare the contents of the ACC with number n
CMP	<address>	Compare the contents of the ACC with the contents of <address>
LSL	#n	Bits in the ACC are shifted logically n places to the left. Zeros are introduced on the right-hand end
LSR	#n	Bits in the ACC are shifted logically n places to the right. Zeros are introduced on the left-hand end
<address> can be an absolute or symbolic address # denotes a denary number, e.g. #127 B denotes a binary number, e.g. B10010001 & denotes a hexadecimal number, e.g. &4A		

The current contents of main memory are shown:

Address	Data
100	0000 0011
101	1010 1110
102	1100 1100
103	1111 1111
104	1100 1100

(a) Complete the table by writing the contents of the ACC after the execution of each instruction.

Current contents of the ACC	Instruction	Contents of the ACC after the execution of the instruction
0000 1111	AND 101	
0000 0000	LDM #100	
0000 0001	XOR &F1	
0001 0001	CMP 101	

[4]

(b) The Von Neumann model for a computer system uses registers.

Describe the role of the Memory Address Register (MAR) and Memory Data Register (MDR) in the fetch-execute (F-E) cycle.

.....

.....

.....

.....

.....

.....

.....

..... [4]

(c) Assembly language instructions are grouped.

Complete each statement by writing the name of the appropriate instruction group.

Loading data into the accumulator is an example of an instruction in the

..... group. Incrementing the index register is an example

of an instruction in the ..... group. Branching to another

address is an example of an instruction in the ..... group.

[3]

11 The table shows assembly language instructions for a processor that has one register, the Accumulator (ACC).

Label	Instruction		Explanation
	Opcode	Operand	
	LDM	#n	Load the number n to the ACC
	LDD	<address>	Load the contents of the location at the given address to ACC
	LDI	<address>	The address to be used is at the given address. Load the contents of this second address to the ACC.
	ADD	<address>	Add the contents of the given address to the ACC
	SUB	<address>	Subtract the contents of the given address from the ACC
	STO	<address>	Store the contents of the ACC at the given address
<label>:		<data>	Gives a symbolic address <label> to the memory location with the contents <data>
# denotes a denary number, e.g. #123 <label> can be used in place of <address>			

The current contents of memory are:

Address	Contents
150	26
300	86
420	150

Write **assembly language** code, using **only** the given instruction set to:

- store the contents of location 300 as labelled variable A
- store the contents of location 420 as labelled variable B
- add the value stored in the address contained in variable B to the value contained in variable A
- store the result in variable Answer.

Show the initialisation and values of the variables A, B and Answer in the table provided.

.....

.....

.....

.....

.....

.....

.....

.....

Label	Content

[7]

7 **(a)** The following table shows part of the instruction set for a processor. The processor has one register: the Accumulator (ACC).

Instruction		Explanation
Opcode	**Operand**	
LDD	<address>	Direct addressing. Load the contents of the location at the given address to ACC
LDM	#n	Immediate addressing. Load the number n to ACC
STO	<address>	Store the contents of ACC at the given address
ADD	#n/Bn/&n	Add the number n to the ACC
ADD	<address>	Add the contents of the given address to the ACC
INC	<register>	Add 1 to the contents of the register (ACC)
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
JMP	<address>	Jump to the given address
END		Return control to the operating system

ACC denotes Accumulator
<address> can be an absolute or a symbolic address
# denotes a denary number, e.g. #123
B denotes a binary number, e.g. B01001010
& denotes a hexadecimal number, e.g. &4A

The current contents of memory are:

Address	Instruction
10	12
11	11
12	10
13	22
14	22
...	
100	LDD 10
101	ADD 12
102	STO 11
103	CMP 14
104	JPE 107
105	INC ACC
106	JMP 101
107	STO 10
108	END

(i) Trace the program currently in memory using the following trace table.

Instruction address	ACC	Memory address				
		10	**11**	**12**	**13**	**14**
		12	11	10	22	22

[3]

(ii) State the effect of changing instruction `LDD 10` in address 100 to `LDM #10`

.....

.....

..... [1]

(iii) Identify **and** describe **one** mode of addressing **not** given in the table of instructions in part **(a)**.

Mode of addressing .....

Description .....

.....

..... [2]

(b) The table shows part of the instruction set for a processor. The processor has one register: the Accumulator (ACC).

Instruction		Explanation
Opcode	**Operand**	
AND	#n/Bn/&n	Bitwise AND operation of the contents of ACC with the operand
AND	<address>	Bitwise AND operation of the contents of ACC with the contents of <address>
XOR	#n/Bn/&n	Bitwise XOR operation of the contents of ACC with the operand
XOR	<address>	Bitwise XOR operation of the contents of ACC with the contents of <address>
OR	#n/Bn/&n	Bitwise OR operation of the contents of ACC with the operand
OR	<address>	Bitwise OR operation of the contents of ACC with the contents of <address>
<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		

(i) The ACC currently contains the following binary value.

1	1	1	1	0	0	0	0

Write the result after the instruction `OR B00001111` is run.

[1]

(ii) The ACC currently contains the following binary value.

0	0	0	1	1	1	0	1

Write the result after the instruction `XOR #30` is run.

[1]

5 The following table shows part of the instruction set for a processor. The processor has two registers: the Accumulator (ACC) and an Index Register (IX).

Instruction		Explanation
Opcode	**Operand**	
LDM	#n	Immediate addressing. Load the number n to ACC
STO	<address>	Store the contents of ACC at the given address
ADD	#n/Bn/&n	Add the number n to the ACC
ADD	<address>	Add the contents of the given address to the ACC
SUB	<address>	Subtract the contents of the given address from the ACC
SUB	#n/Bn/&n	Subtract the number n from the ACC
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
END		Return control to the operating system
ACC denotes Accumulator <address> can be an absolute or a symbolic address # denotes a denary number, e.g. #123 B denotes a binary number, e.g. B01001010 & denotes a hexadecimal number, e.g. &4A		

(a) The current contents of memory are:

Address	Instruction
50	47
51	48
52	49
53	50
...	
500	LDM #50
501	DEC ACC
502	CMP 51
503	JPE 505
504	JMP 501
505	STO 50
506	SUB #10
507	STO 51
508	END

(i) Trace the program currently in memory using the following trace table.

Instruction address	ACC	Memory address			
		50	**51**	**52**	**53**
		47	48	49	50

[3]

(ii) Complete the table by identifying **and** describing **two** modes of addressing that are **not** used in the program in part **(a)**.

Mode of addressing	Description
.....	.....
.....	.....
	.....
	.....
.....	.....
.....	.....
	.....
	.....

[4]

(b) The table shows part of the instruction set for a processor. The processor has one register: the Accumulator (ACC).

Instruction		Explanation
Opcode	**Operand**	
AND	#n/Bn/&n	Bitwise AND operation of the contents of ACC with the operand
AND	<address>	Bitwise AND operation of the contents of ACC with the contents of <address>
XOR	#n/Bn/&n	Bitwise XOR operation of the contents of ACC with the operand
XOR	<address>	Bitwise XOR operation of the contents of ACC with the contents of <address>
OR	#n/Bn/&n	Bitwise OR operation of the contents of ACC with the operand
OR	<address>	Bitwise OR operation of the contents of ACC with the contents of <address>
<address> can be an absolute or a symbolic address # denotes a denary number, e.g. #123 B denotes a binary number, e.g. B01001010 & denotes a hexadecimal number, e.g. &4A		

Write the bit manipulation instructions that can be used to set only the most significant bit to 1 in an 8-bit register.

The instructions need to work on a register that contains any 8-bit binary number.

.....

..... [2]

8 The following table shows part of the instruction set for a processor. The processor has two registers, the Accumulator (ACC) and the Index Register (IX).

Instruction		Explanation
Opcode	**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
ADD	#n/Bn/&n	Add the number n to the ACC
ADD	<address>	Add the contents of the given address to the ACC
SUB	#n/Bn/&n	Subtract the number n from the ACC
SUB	<address>	Subtract the contents of the given address from 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)
<address> can be an absolute or a symbolic address # denotes a denary number, e.g. #123 B denotes a binary number, e.g. B01001010 & denotes a hexadecimal number, e.g. &4A		

(a) The current contents of memory are shown:

Address	Data
19	25
20	23
21	2
22	4
23	15
24	50
25	22

The current contents of the ACC and IX are shown:

ACC	50
IX	20

Complete the table by writing the content of the ACC and the IX after each set of instructions has run.

	Instructions	ACC content	IX content
1	`LDM #19` `DEC ACC`		
2	`LDD 23` `ADD 19`		
3	`LDI 25` `INC ACC`		
4	`LDR #21` `LDX 2`		

[5]

(b) The instruction set also includes these bit manipulation instructions:

Instruction		Explanation
Opcode	**Operand**	
`AND`	`#n/Bn/&n`	Bitwise AND operation of the contents of ACC with the operand
`AND`	`<address>`	Bitwise AND operation of the contents of ACC with the contents of <address>
`XOR`	`#n/Bn/&n`	Bitwise XOR operation of the contents of ACC with the operand
`XOR`	`<address>`	Bitwise XOR operation of the contents of ACC with the contents of <address>
`OR`	`#n/Bn/&n`	Bitwise OR operation of the contents of ACC with the operand
`OR`	`<address>`	Bitwise OR operation of the contents of ACC with the contents of <address>
`LSL`	`#n`	Bits in ACC are shifted logically n places to the left. Zeros are introduced on the right-hand end
`LSR`	`#n`	Bits in ACC are shifted logically n places to the right. Zeros are introduced on the left-hand end.

<address> can be an absolute or a symbolic address
# denotes a denary number, e.g. #123
B denotes a binary number, e.g. B01001010
& denotes a hexadecimal number, e.g. &4A

The current content of the ACC is shown:

ACC	1	0	0	1	1	0	1	0

(i) The table has three sets of instructions. The binary number 10011010 is reloaded into the ACC before each set of instructions is run.

Complete the table by writing the content of the ACC after each set of instructions has run.

	Instructions	ACC content
1	`LSL #2`	
2	`ADD #5` `AND #30`	
3	`OR B11110010` `INC ACC`	

[3]

(ii) Explain how bit manipulation can be used to test whether the binary number stored in the ACC represents an odd denary number.

Write the bit manipulation instruction that will be used.

Explanation .....

.....

.....

.....

.....

Instruction .....

[3]

4 The following table shows part of the instruction set for a processor. The processor has two registers: the Accumulator (ACC) and an Index Register (IX).

Instruction		Explanation
Opcode	**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
ADD	#n/Bn/&n	Add the number n to the ACC
ADD	<address>	Add the contents of the given address to the ACC
SUB	#n/Bn/&n	Subtract the number n from the ACC
SUB	<address>	Subtract the contents of the given address from the ACC
INC	<register>	Add 1 to the contents of the register (ACC or IX)

<address> can be an absolute or a symbolic address
# denotes a denary number, e.g. #123
B denotes a binary number, e.g. B01001010
& denotes a hexadecimal number, e.g. &4A

(a) The current contents of memory are shown:

Address	Data
19	24
20	2
21	1
22	3
23	5
24	4
25	22

The current contents of the ACC and IX are shown:

ACC	12
IX	1

Complete the table by writing the content of the ACC after each program has run.

Program number	Code	ACC content
1	LDD 20 ADD #2	
2	LDX 22	
3	LDI 25 INC ACC SUB 22	
4	LDD 19 LDM #5 LDM #25	

[4]

(b) The processor includes these bit manipulation instructions:

Instruction		Explanation
Opcode	**Operand**	
AND	#n/Bn/&n	Bitwise AND operation of the contents of ACC with the operand
AND	<address>	Bitwise AND operation of the contents of ACC with the contents of <address>
XOR	#n/Bn/&n	Bitwise XOR operation of the contents of ACC with the operand
XOR	<address>	Bitwise XOR operation of the contents of ACC with the contents of <address>
OR	#n/Bn/&n	Bitwise OR operation of the contents of ACC with the operand
OR	<address>	Bitwise OR operation of the contents of ACC with the contents of <address>
<address> can be an absolute or a symbolic address # denotes a denary number, e.g. #123 B denotes a binary number, e.g. B01001010 & denotes a hexadecimal number, e.g. &4A		

The current contents of memory are shown:

Address	Data
30	01110101
31	11111111
32	00000000
33	11001100
34	10101010

The current content of the ACC is shown:

1	0	0	1	1	0	1	0

Complete the table by writing the content of the ACC after each program has run.

The binary number 10011010 is reloaded into the ACC before each program is run.

Program number	Code	ACC content
1	AND 31	
2	XOR B01001111	
3	OR #30	

[3]

A-LEVEL COMPUTER SCIENCE • EXERCISE SHEET

4.3 Bit manipulation

Name: _____ Class: _____ Date: _____ **Total: 33 marks**

KEY WORDS bit manipulation 位操作 • logical shift 逻辑移位 • mask 掩码 • cyclic shift 循环移位 • bit 位

Objective

Build the skills to answer exam questions on **bit manipulation** 位操作 — binary shifts and using bit masks to monitor and control a device.

You must be able to:

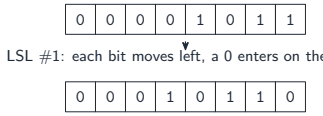
- perform **logical binary shifts** (left and right) and state their effect on the value
- use a **mask** 掩码 to **set**, **clear**, **toggle** and **test** a single bit
- write the **assembly instructions** that do this: `AND`, `OR`, `XOR`, `LSL`, `LSR`
- read the three operand forms: `#` denary, `B` binary, `&` hexadecimal

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.

■ Binary shifts

A **logical shift** moves every bit left or right, filling the new positions with 0.



set bit 2	clear bit 6	toggle bit 3
01001000	01001000	01001000
OR 00000100	AND 10111111	XOR 00001000
= 01001100	= 00001000	= 01000000

Bit masking with AND/OR isolates or sets chosen bits

- **left shift** by $n \rightarrow \times\ 2^n$ (for an unsigned number).

- **right shift** by $n \rightarrow$ **integer** $\div 2^n$.
- An **arithmetic** right shift keeps the **sign bit**, so a negative signed number stays negative.
- A **cyclic shift** 循环移位 (rotate) feeds the bit that falls off one end back in at the other end, so no bits are lost.

■ Bit masks (monitor and control)

A device often uses **one bit per signal** (bit n = LED n). Combine the register with a **mask** using a logic operation:

Goal	Operation	Mask (for bit n)
set bit n to 1	`R OR mask`	bit $n = 1$, rest 0
clear bit n to 0	`R AND mask`	bit $n = 0$, rest 1
toggle bit n	`R XOR mask`	bit $n = 1$, rest 0
test bit n	`R AND mask`, then check $\neq 0$	bit $n = 1$, rest 0

Example — turn on bit 0 of `00000100` without changing the rest: `00000100 OR 00000001 = 00000101`.

■ Writing it as an assembly instruction

The exam asks for the **instruction**, not just the mask. Every one of these works on the **accumulator**, `ACC`, and there is only one general-purpose register:

Instruction	What it does to `ACC`
`AND #n` / `AND Bn` / `AND &n`	bitwise AND of `ACC` with the operand
`AND <address>`	bitwise AND of `ACC` with the contents of that address
`OR` and `XOR`	the same two forms, with OR and XOR
`LSL #n`	shift `ACC` **left** n places, zeros in on the right
`LSR #n`	shift `ACC` **right** n places, zeros in on the left

The operand prefix says which base the number is written in:

- `#123` — a **denary** number;
- `B01001010` — a **binary** number;
- `&4A` — a **hexadecimal** number.

So "set the least significant bit of `ACC` to 1" is `OR B00000001` (or `OR #1`, or `OR &01` — all the same instruction with the operand written three ways). "Clear bit 3" is `AND B11110111`. "Test bit 3" is `AND B00001000`, then check whether `ACC` is zero.

A `<label>:` in front of an instruction names it so a jump can reach it, and a `<label>: <data>` line gives a symbolic address to a memory location holding that data.

2 Practice

Now apply the methods above.

2.1 Perform a **logical shift left by 1** on `00000101`, and give the denary value before and after. [2]

2.2 State the arithmetic effect of a **logical shift right by 1** on an unsigned number. [1]

2.3 Give the mask and the operation needed to **set bit 2** of a register to 1, leaving the other bits unchanged. [2]

2.4 A logical shift **left by 3** multiplies an unsigned number by what? [1]

3 Exam-style questions

3.1 (a) Perform a logical shift **left by 2** on `00000110`, and give the denary result. [2]

(b) State the effect this shift has on the value. [1]

3.2 An 8-bit register controls 8 LEDs (bit n = LED n).

(a) Give the mask and operation to turn **on only LED 0**, leaving the others unchanged. [2]

(b) Give the mask and operation to **test whether LED 3 is on**. [2]

3.3 Explain why an **arithmetic** right shift, not a logical one, is used when halving a **negative** signed number. [2]

3.4 State the difference between a **logical** shift and a **cyclic** shift. [2]

3.5 A microcontroller uses an 8-bit accumulator `ACC` to control a set of sensors and indicator lamps. Bit 0 is the least significant bit.

(a) `ACC` currently contains `01101100`. **Write** the single bit-manipulation instruction that **sets the least significant bit to 1** without changing any other bit, and **give** the resulting contents of `ACC`. [2]

(b) **Write** the single instruction that **clears bit 6 to 0**, leaving the other bits unchanged, using a **binary** operand. [2]

(c) **Write** the instruction that would **test whether bit 2 is set**, and **describe** how the program then decides the answer. [3]

(d) **Complete** the table by giving the contents of `ACC` after each instruction. Each row starts from the value shown in the row above. [3]

instruction	**contents of `ACC`**
starting value	`10011110`
`LSR #3`	
`XOR B00001111`	
`LSL #2`	

(e) The value `10011110` is now treated as a **two's complement** integer. **Show** the result of an **arithmetic** right shift of 3 places on it, and **explain** why the result differs from your answer in (d). [3]

(f) **Complete** the binary addition below, showing your working and any **overflow** bit. **State** whether the answer is valid if the values are unsigned 8-bit integers. [3]

`10110101` + `01011010`

Solutions

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

2.1

- logical shift left one place: insert 0 at the right
- `00000101` → **`00001010`**; denary 5 → **10**

2.2

- it **halves** the number (integer ÷2)

2.3

- bit 2 is the third bit from the right, so the mask is `00000100`
- operation **OR**: `R OR 00000100`

2.4

- a left shift of 3 places multiplies by $2^3 = $ **8**

3.1

(a) logical shift left two places

- `00000110` → **`00011000`**; denary **24**

(b) it **multiplies the value by 4**

3.2

(a) mask `00000001`, operation **OR** (`R OR 00000001`)

(b) mask `00001000`, operation **AND** (`R AND 00001000`)

- if the result is non-zero, LED 3 is on

3.3

- a logical shift puts a **0** into the top bit, which would make a negative number look positive
- an arithmetic shift **copies the sign bit**, so the number stays negative

3.4

- a logical shift fills the vacated end with **0** and the bit shifted off the other end is **lost**
- a cyclic shift feeds that bit **back in** at the other end, so no bits are lost

3.5

(a) `OR B00000001` (or `OR #1`, or `OR &01`)

- `ACC` becomes **`01101101`**

(b) `AND B10111111`

- the mask is 1 everywhere **except** bit 6, so every other bit survives the AND unchanged

(c) `AND B00000100`

- the AND leaves every other bit zero, so `ACC` is **non-zero** only if bit 2 was 1
- the program then branches on whether `ACC` is zero

(d) `LSR #3` on `10011110` → **`00010011`**

- `XOR B00001111` flips the low four bits → **`00011100`**
- `LSL #2` → **`01110000`**

(e) an arithmetic shift copies the **sign bit** into the vacated places

- `10011110` → **`11110011`**
- the logical shift in (d) put zeros in, turning a negative number positive; the arithmetic shift keeps it negative, which is what halving a negative number must do

(f)

```
  10110101
+ 01011010
----------
 100001111
```

- nine bits; the 8-bit answer is `00001111` with an **overflow** bit of 1
- as unsigned 8-bit integers the result is **not valid**: $181 + 90 = 271$, which will not fit in 8 bits

8 **(a)** The following table shows part of the instruction set for a processor. The processor has two registers: the Accumulator (ACC) and an Index Register (IX).

Instruction		Explanation
Opcode	**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
INC	<register>	Add 1 to the contents of the register (ACC or IX)
STO	<address>	Store the contents of ACC at the given address
ADD	#n/Bn/&n	Add the number n to the ACC
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>
JPE	<address>	Following a compare instruction, jump to <address> if the compare was True
END		Return control to the operating system

ACC denotes Accumulator
<address> can be an absolute or a symbolic address
# denotes a denary number, e.g. #123
B denotes a binary number, e.g. B01001010
& denotes a hexadecimal number, e.g. &4A

The current contents of memory are:

address	Instruction
80	10
81	8
82	80
83	81
...	
200	LDD 81
201	INC ACC
202	STO 83
203	LDI 82
204	CMP 83
205	JPE 209
206	LDD 83
207	ADD #10
208	JMP 210
209	DEC ACC
210	STO 81
211	END

Trace the program currently in memory using the following trace table.

Instruction address	ACC	Memory address			
		80	**81**	**82**	**83**
		10	8	80	81

[4]

(b) The table shows part of the instruction set for a processor. The processor has one register: the Accumulator (ACC).

Instruction		Explanation
Opcode	**Operand**	
`AND`	`#n/Bn/&n`	Bitwise AND operation of the contents of ACC with the operand
`AND`	`<address>`	Bitwise AND operation of the contents of ACC with the contents of <address>
`XOR`	`#n/Bn/&n`	Bitwise XOR operation of the contents of ACC with the operand
`XOR`	`<address>`	Bitwise XOR operation of the contents of ACC with the contents of <address>
`OR`	`#n/Bn/&n`	Bitwise OR operation of the contents of ACC with the operand
`OR`	`<address>`	Bitwise OR operation of the contents of ACC with the contents of <address>
`LSL`	`#n`	Bits in ACC are shifted logically n places to the left. Zeros are introduced on the right-hand end
`LSR`	`#n`	Bits in ACC are shifted logically n places to the right. Zeros are introduced on the left-hand end
<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		

(i) Write the bit manipulation instruction that can be used to set the least significant bit to 1 in an 8-bit register. All other bits must remain unchanged.

The instruction needs to work on a register that contains any 8-bit binary number.

..... [1]

(ii) The ACC currently contains the following binary value.

0	1	0	1	0	1	0	1

Write the result after the instruction `XOR &FE` is run.

[1]

(iii) The ACC currently contains the following binary value.

0	1	1	0	1	0	1	1

Write the result after the instruction `LSR #5` is run.

[1]

7 **(a)** A computer can perform logical and arithmetic shifts.

(i) Show the result of a logical left shift of 2 places on the two's complement binary integer 11001010

.....

..... [1]

(ii) Show the result of an arithmetic right shift of 3 places on the two's complement binary integer 10011110

.....

..... [1]

(b) Complete the following binary addition. Show your working. Include any overflow bit(s).

```
  1 1 1 1 0 1 0 1
+ 1 0 1 1 0 0 0 1
-----------------
```

[1]

(c) Subtract the binary number 00011110 from the binary number 01100100 using binary subtraction. Show your working.

[2]

7 The following table shows part of the instruction set for a processor. The processor has two registers, the Accumulator (ACC) and the Index Register (IX).

Instruction		**Explanation**
Opcode	**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
ADD	#n/Bn/&n	Add the number n to the ACC
ADD	<address>	Add the contents of the given address to the ACC
SUB	#n/Bn/&n	Subtract the number n from the ACC
SUB	<address>	Subtract the contents of the given address from 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)
<address> can be an absolute or a symbolic address # denotes a denary number, e.g. #123 B denotes a binary number, e.g. B01001010 & denotes a hexadecimal number, e.g. &4A		

(a) The current contents of memory are shown:

Address	Data
50	54
51	55
52	50
53	52
54	100
55	25
56	50

The current contents of the ACC and IX are shown:

ACC	50
IX	45

Complete the table by writing the content of the ACC after each program has run.

	Instructions	ACC content

1	`LDD 50`
	`ADD #4`
	`ADD 54`	

2	`LDI 53`
	`DEC ACC`
	`ADD 56`	

3	`LDM #55` `SUB #5`	

[3]

(b) The instruction set also includes these bit manipulation instructions:

Instruction		Explanation
Opcode	**Operand**	
AND	#n/Bn/&n	Bitwise AND operation of the contents of ACC with the operand
AND	<address>	Bitwise AND operation of the contents of ACC with the contents of <address>
XOR	#n/Bn/&n	Bitwise XOR operation of the contents of ACC with the operand
XOR	<address>	Bitwise XOR operation of the contents of ACC with the contents of <address>
OR	#n/Bn/&n	Bitwise OR operation of the contents of ACC with the operand
OR	<address>	Bitwise OR operation of the contents of ACC with the contents of <address>
<address> can be an absolute or a symbolic address # denotes a denary number, e.g. #123 B denotes a binary number, e.g. B01001010 & denotes a hexadecimal number, e.g. &4A		

Explain how bit manipulation can be used to clear the data in an 8-bit register.

Write the bit manipulation instruction that will be used.

Explanation .....

.....

.....

.....

.....

Instruction .....

[3]

5 The following table shows part of the instruction set for a processor. The processor has two registers: the Accumulator (ACC) and an Index Register (IX).

Instruction		Explanation
Opcode	**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
ADD	#n/Bn/&n	Add the number n to the ACC
ADD	<address>	Add the contents of the given address to the ACC
SUB	#n/Bn/&n	Subtract the number n from the ACC
SUB	<address>	Subtract the contents of the given address from the ACC
INC	<register>	Add 1 to the contents of the register (ACC or IX)

<address> can be an absolute or a symbolic address
# denotes a denary number, e.g. #123
B denotes a binary number, e.g. B01001010
& denotes a hexadecimal number, e.g. &4A

(a) The current contents of memory are shown:

Address	Data
10	1
11	3
12	5
13	11
14	10
15	16
16	12

The current contents of the ACC and IX are shown:

ACC	10
IX	0

Complete the table by writing the content of the ACC after each program has run.

Program number	Code	ACC content
1	LDI 15 SUB #1	
2	LDD 14 ADD 11	
3	LDM #11 ADD #3 SUB 16	
4	LDR #2 LDX 14 ADD #2	

[4]

(b) The processor includes these bit manipulation instructions:

Instruction		Explanation
Opcode	**Operand**	
AND	#n/Bn/&n	Bitwise AND operation of the contents of ACC with the operand
AND	<address>	Bitwise AND operation of the contents of ACC with the contents of <address>
XOR	#n/Bn/&n	Bitwise XOR operation of the contents of ACC with the operand
XOR	<address>	Bitwise XOR operation of the contents of ACC with the contents of <address>
OR	#n/Bn/&n	Bitwise OR operation of the contents of ACC with the operand
OR	<address>	Bitwise OR operation of the contents of ACC with the contents of <address>
<address> can be an absolute or a symbolic address # denotes a denary number, e.g. #123 B denotes a binary number, e.g. B01001010 & denotes a hexadecimal number, e.g. &4A		

The current contents of memory are shown:

Address	Data
25	11000110
26	11100001
27	10000001
28	11001101
29	00001111

The current content of the ACC is shown:

0	1	0	0	0	1	1	0

Complete the table by writing the content of the ACC after each program has run.

The binary number 01000110 is reloaded into the ACC before each program is run.

Program number	Code	ACC content
1	XOR 29	
2	AND #29	
3	OR B11111111	

[3]