

Processor fundamentals

A-Level Computer Science 9618 · Topic 4 revision notes

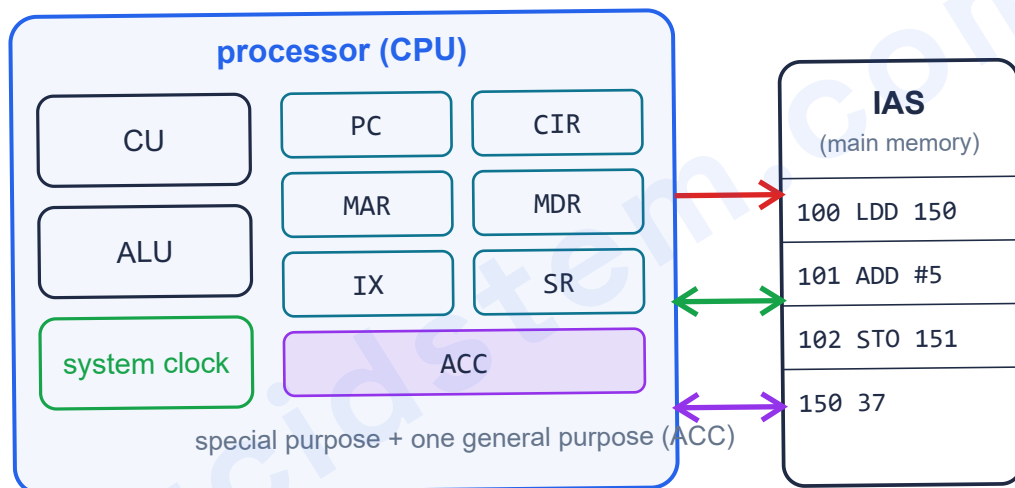
Everything from the 8 lessons of this topic on a few pages: the key idea of each lesson, the diagram to remember, the points to learn and the traps examiners set. Read it, cover it, test yourself.

- | | |
|---|--|
| 1. <u>The Von Neumann model and registers</u> | 5. <u>Assembly language and the two-pass assembler</u> |
| 2. <u>Buses, ports and processor performance</u> | 6. <u>Addressing modes</u> |
| 3. <u>The fetch-execute cycle in register transfer notation</u> | 7. <u>Tracing an assembly language program</u> |
| 4. <u>Interrupts</u> | 8. <u>Shifts, masks and bit manipulation</u> |

1 The Von Neumann model and registers

Syllabus 4.1

In the **Von Neumann model**, a computer has a single processor and one memory. The **stored program concept** means that program instructions and data are held **together in the same main memory**, and instructions are fetched from it and executed **one at a time, in sequence**. Inside the processor, the **control unit (CU)** decodes instructions and sends control signals, the **arithmetic and logic unit (ALU)** does calculations and comparisons, the **system clock** times every step, and **registers** hold the values in use. A **special purpose register** has one fixed job (PC, MAR, MDR, CIR, IX, status register); a **general purpose register** holds any data being processed (the accumulator, ACC).



stored program: instructions and data share one memory

Fig. 1: the processor's units and registers, linked by the buses to the immediate access store

Worked example: Name the register

Identify the register that (a) is incremented during every fetch, (b) has a flag set when an addition overflows, (c) is added to the operand of LDX.

1. (a) The register that points to the next instruction is incremented after each fetch: the program counter.
2. (b) Flags that record the outcome of an ALU operation are held in the status register.
3. (c) Indexed addressing adds the contents of the index register to the operand to form the address.

Answer: (a) PC (b) status register (c) IX

Where and what.

- ✓ **Addresses:** PC (next instruction), MAR (location being accessed now).
- ✓ **Contents:** MDR (data or instruction to or from memory), CIR (current instruction), ACC (data and results).
- ✓ **Helpers:** IX (added to an address), status register (flags such as carry, zero, negative, overflow).
- ✓ **Stored program:** instructions and data together in one memory, executed one at a time.

Watch out!

"Data and programs are stored in memory" is not enough to describe the stored program concept. Say they are stored together in the same memory and that instructions are fetched and executed one at a time.

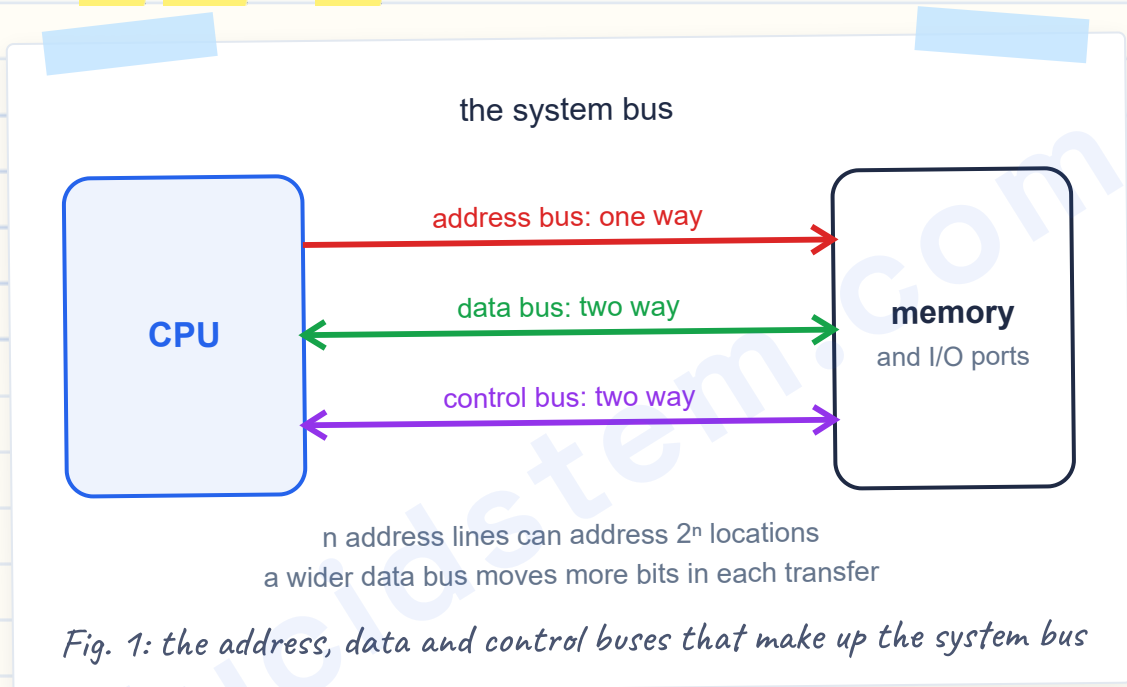
The control unit does not do calculations. It decodes and controls; the ALU calculates and compares.

General purpose does not mean "used for everything at once". A general purpose register holds whatever data the program is working on; a special purpose register always has the same fixed job.

2 Buses, ports and processor performance

Syllabus 4.1

The **address bus** carries addresses one way, from the processor to memory; with **n** address lines it can address **2^n** locations. The **data bus** carries data and instructions both ways; a wider data bus moves more bits in each transfer. The **control bus** carries signals such as read, write, clock and interrupt, both ways. Performance depends on the **processor type**, the **number of cores**, the **bus width**, the **clock speed** and the size of the **cache**. **Ports** such as **USB**, **HDMI** and **VGA** connect peripheral devices.



Worked example: Address bus width

A processor has a 24-bit address bus. How many memory locations can it address?

1. Each address line is one bit, so there are 2^{24} different addresses.
2. $2^{24} = 2^{10} \times 2^{10} \times 2^4 = 1024 \times 1024 \times 16 = 16\,777\,216$.
3. If every location holds one byte, that is 16 MiB of addressable memory.

Answer: 16 777 216 locations (16 MiB)

Name it, then explain it.

- ✓ **Address bus:** one way; n lines address 2^n locations.
- ✓ **Data bus:** two way; wider means more bits per transfer.
- ✓ **Performance:** processor type, cores, bus width, clock speed, cache; always give the reason.
- ✓ **Ports:** USB (serial, power, auto-detect), HDMI (digital video and audio), VGA (analogue video only).

Watch out!

Widening the address bus does not make each transfer bigger: it increases how many memory locations can be addressed. It is the data bus width that sets how many bits move per transfer.

Doubling the cores does not double the speed. Say that some tasks cannot be split up to run in parallel, and that time is spent coordinating the cores.

3 The fetch-execute cycle in register transfer notation

Syllabus 4.1

The **fetch-execute (F-E) cycle** is repeated for every instruction. In **register transfer notation**, **[X]** means the contents of X and **←** means is copied into. The fetch stage is: **MAR ← [PC]**; **PC ← [PC] + 1**; **MDR ← [[MAR]]**; **CIR ← [MDR]**. The control unit then **decodes** the instruction in the CIR and it is **executed**, which may use memory again through the MAR and MDR and the ALU.

the fetch stage in register transfer notation

MAR ← [PC]

address of next instruction

PC ← [PC] + 1

point to the following one

MDR ← [[MAR]]

contents of that address

CIR ← [MDR]

ready to decode

[X] means the contents of X; ← means is copied into

Fig. 1: the four register transfers of the fetch stage

Worked example: LDD 150

The PC holds 100. Address 100 holds the instruction LDD 150 and address 150 holds 37. Give the contents of the PC, MAR, MDR, CIR and ACC after the instruction has been executed.

1. **MAR ← [PC]**: the MAR holds 100.
2. **PC ← [PC] + 1**: the PC holds 101.
3. **MDR ← [[MAR]]**: the MDR holds the instruction LDD 150. **CIR ← [MDR]**: the CIR holds LDD 150.
4. Execute: the operand 150 is copied to the MAR, so the MAR holds 150. **MDR ← [[MAR]]**: the MDR holds 37.
5. **ACC ← [MDR]**: the ACC holds 37.

Answer: PC 101, MAR 150, MDR 37, CIR LDD 150, ACC 37

Four transfers, in order.

- ✓ $MAR \leftarrow [PC]$: the address of the next instruction.
- ✓ $PC \leftarrow [PC] + 1$: point to the following instruction.
- ✓ $MDR \leftarrow [[MAR]]$: double brackets, the contents of that address.
- ✓ $CIR \leftarrow [MDR]$: then decode, execute and check for interrupts.

Watch out!

$MDR \leftarrow [MAR]$ is wrong: it would copy the address itself. The instruction comes from the memory location the MAR points to, so it is $MDR \leftarrow [[MAR]]$.

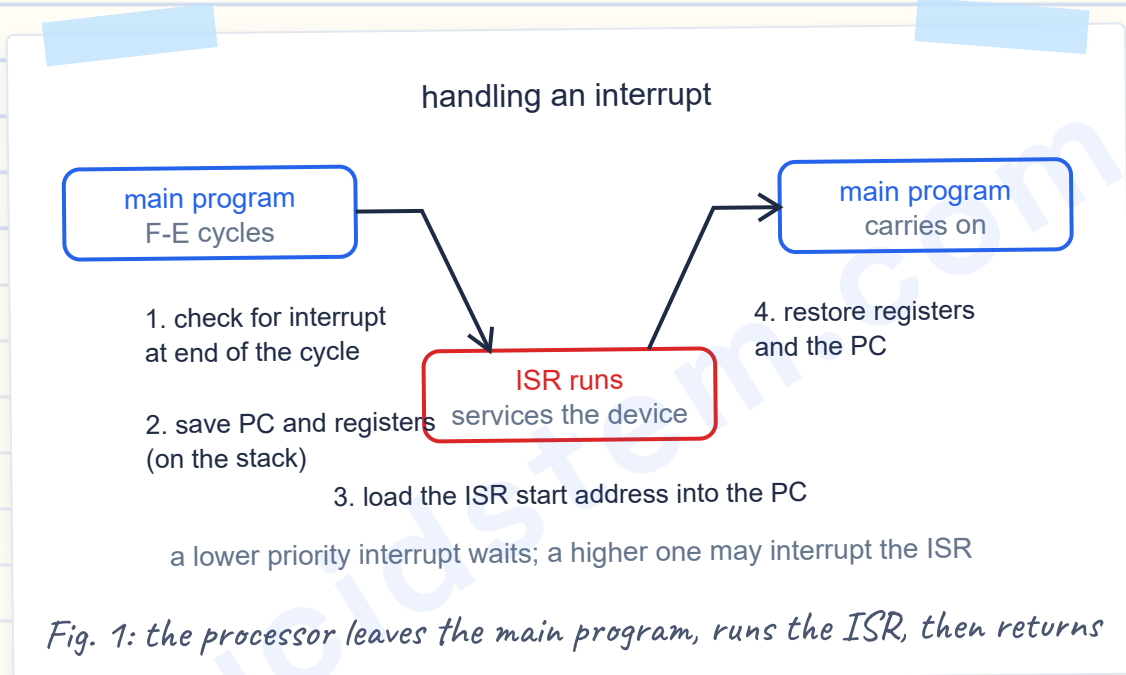
The PC is incremented in the fetch stage, not after the instruction has been executed. This is why a jump instruction simply overwrites the PC.

thelucidstem.com

4 Interrupts

Syllabus 4.1

An **interrupt** is a **signal** from a device or a program to the processor, showing that something **needs attention**. Interrupts are **checked at the end of each fetch-execute cycle**. If there is one with a **higher priority** than the current task, the processor **saves the contents of its registers** (including the PC) on the **stack**, loads the start address of the **interrupt service routine (ISR)** into the PC and runs it. When the ISR finishes, the saved values are **restored** and the interrupted task carries on from where it stopped.



Worked example: Order of servicing

A program is running. The printer sends a "finished" interrupt (priority 1, lowest). While its ISR runs, a key press interrupt (priority 2) arrives, and then a power failure interrupt (priority 5, highest) arrives during the key press ISR. Describe what happens.

1. At the end of a cycle the printer interrupt is detected. It has higher priority than the program, so the program's registers are saved on the stack and the printer ISR starts.
2. The key press (priority 2) is higher than the printer ISR (1), so at the end of a cycle the printer ISR's registers are saved and the key press ISR runs.
3. The power failure (priority 5) interrupts the key press ISR in the same way and is serviced at once, for example by saving data before the power goes.
4. If the power holds, each finished ISR restores the values from the stack in reverse order: key press ISR, then printer ISR, then the program.

Answer: Power failure first, then key press, then printer, then the program resumes

The handling sequence.

- ✓ **Detect:** check for interrupts at the end of each F-E cycle.
- ✓ **Priority:** service only if higher than the current task; otherwise it waits.
- ✓ **Save:** registers (including the PC) pushed onto the stack; ISR address into the PC.
- ✓ **Restore:** when the ISR ends, pop the registers and carry on where it stopped.

Watch out!

An interrupt is a signal to the processor. Do not describe it as "an error": many interrupts, such as a key press or a finished print job, are normal events.

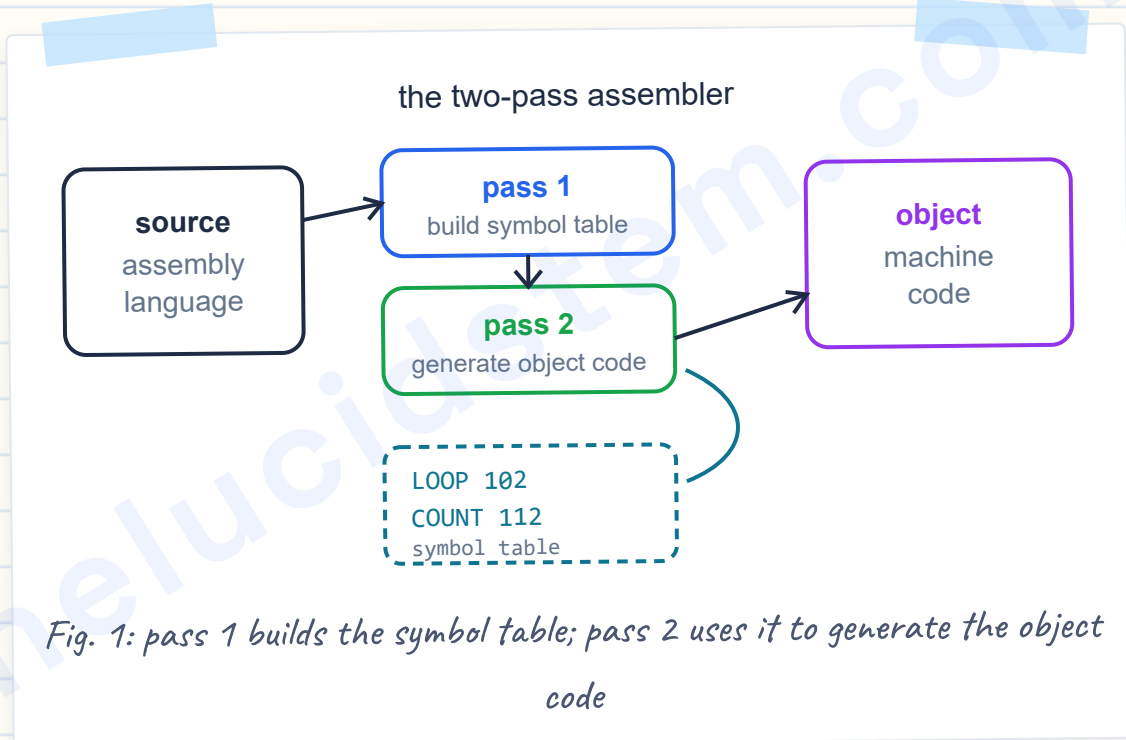
"The computer stops what it is doing" earns nothing. Say that the registers are saved (on the stack) so the task can be resumed later, and that the ISR is executed.

thelucidstem.com

5 Assembly language and the two-pass assembler

Syllabus 4.2

Machine code is the binary instructions a processor executes directly. **Assembly language** is a low-level language that uses **mnemonics** (such as LDD and ADD) for opcodes and **labels** for addresses; each assembly instruction usually translates into **one** machine code instruction, and both are specific to one processor's **instruction set**. An **assembler** translates it. A **two-pass assembler** needs two passes because of **forward references**: **pass 1** reads the code, allocates addresses and builds a **symbol table** of labels and their addresses; **pass 2** uses the symbol table and the **opcode table** to generate the **object code**.



Worked example: Build the symbol table

The program above is loaded starting at address 100. Complete the symbol table after pass 1, and give the object code (with absolute addresses) for the lines at 101 and 109.

1. Pass 1 counts addresses from 100: START is at 100 and LOOP is at 102.
2. At 101, STO TOTAL uses TOTAL before it is defined: a forward reference. TOTAL is added to the symbol table without an address.
3. The data labels are reached later: NUM at 111, COUNT at 112 and TOTAL at 113, which fills in the missing address.
4. Pass 2 replaces each label with its address: STO TOTAL becomes STO 113 and JPN LOOP becomes JPN 102.

Answer: START 100, LOOP 102, NUM 111, COUNT 112, TOTAL 113; STO 113 and JPN 102

Groups and passes.

- ✓ **Groups:** data movement, input and output, arithmetic, unconditional and conditional jumps, compare.
- ✓ **One-to-one:** one assembly instruction becomes one machine code instruction.
- ✓ **Pass 1:** allocate addresses, build the symbol table (including forward references).
- ✓ **Pass 2:** look up opcodes, replace labels with addresses, generate object code.

Watch out!

Assembly language is not machine code. It must still be translated, by an assembler, before the processor can execute it.

CMP only compares: it does not change the accumulator. It sets the result that the next JPE or JPN uses.

Do not say pass 1 "translates" the program. Pass 1 builds the symbol table; the object code is generated in pass 2.

6 Addressing modes

Syllabus 4.2

An **addressing mode** tells the processor how to use the operand to find the data.

Immediate: the operand **is** the value (LDM #n). **Direct**: the operand is the **address** of the value (LDD). **Indirect**: the operand is an address that holds **the address of the value** (LDI, CMI). **Indexed**: the address is the operand **plus the contents of the index register** (LDX). **Relative**: the address is an **offset** number of locations away from the address of the current instruction.

same operand, different modes (IX holds 2)

LDM #200

ACC = 200

LDD 200

ACC = 203

LDX 200

ACC = 7

LDI 200

ACC = 18

200 | 203

201 | 50

202 | 7

203 | 18

204 | 25

immediate: the operand is the value
indexed: $200 + IX = 202$

direct: the operand is the address
indirect: 200 holds 203, use 203

Fig. 1: one operand, 200, gives four different values in four different modes

Worked example: Four loads

Using the memory above, give the contents of the ACC after each instruction: LDM #200, LDD 200, LDX 200, LDI 200.

1. LDM #200 is immediate: the operand is the value, so ACC = 200.
2. LDD 200 is direct: load the contents of address 200, so ACC = 203.
3. LDX 200 is indexed: address = $200 + IX = 200 + 2 = 202$, which holds 7, so ACC = 7.
4. LDI 200 is indirect: address 200 holds 203, so use address 203, which holds 18, so ACC = 18.

Answer: 200, 203, 7, 18

One question per mode.

- ✓ **Immediate (#):** the operand is the value.
- ✓ **Direct:** go to the address once.
- ✓ **Indirect:** go to the address, read another address, go there.
- ✓ **Indexed:** add IX to the operand first.
Relative: offset from the current instruction.

Watch out!

The # is what makes LDM, LDR, ADD #n, SUB #n and CMP #n immediate. Without it, ADD S means "add the contents of address S", not "add S".

Indexed and indirect are easy to swap. Indexed adds IX to the operand; indirect looks up a second address in memory.

thelucidstem.com

7 Tracing an assembly language program

Syllabus 4.2

To **trace** a program, carry out **one instruction at a time** and record each change in a **trace table**, with a column for the **ACC**, any **memory addresses** that change, the **IX** if it is used, and the **output**. A **CMP** or **CMI** only sets the result of the comparison; the next **JPE** (jump if True) or **JPN** (jump if False) uses it. **JMP** always jumps. **OUT** outputs the **character** whose ASCII value is in the ACC.

a loop: JPN jumps back until the counter is 0

```
100 LDD 200
101 ADD 201
102 STO 200
103 LDD 202
104 DEC ACC
105 STO 202
106 CMP #0
107 JPN 100
108 LDD 200
109 END
```

200		0	total
201		6	number
202		3	counter

each pass adds 6 to the total
and takes 1 off the counter

a trace table records every change, one instruction at a time

Fig. 1: a program that multiplies 6 by 3 by repeated addition

Worked example: Output a stored string

Trace the program above and state the output.

1. LDR #0 sets IX to 0. LDX 200 loads the contents of 200 + 0, so ACC = 72.
2. CMP #0 is False, so JPE does not jump. OUT outputs H. INC IX makes IX = 1 and JMP returns to 101.
3. LDX 200 now uses 201, so ACC = 73: OUT outputs I and IX = 2. Next, 202 gives ACC = 33: OUT outputs ! and IX = 3.
4. LDX 200 uses 203, so ACC = 0. CMP #0 is True, so JPE jumps to 107 and the program ends.

Answer: HI! (and IX finishes at 3)

Trace like the CPU.

- ✓ **Start row:** copy every starting value first.
- ✓ **Record changes only:** LDD, ADD, SUB, INC, DEC change the ACC; STO changes memory.
- ✓ **Compare then jump:** JPE jumps if True, JPN if False, JMP always.
- ✓ **ASCII:** OUT and IN use character codes (48 = 0, 65 = A).

Watch out!

JPN jumps when the compare was False. Reading it as "jump if not zero" works here only because the compare is with #0; always check what was actually compared.

OUT does not print the number in the ACC. With 50 in the ACC, OUT outputs the character 2, not 50.

thelucidstem.com

8 Shifts, masks and bit manipulation

Syllabus 4.3

In a **logical shift**, bits move left or right and **zeros** fill the empty places (LSL #n, LSR #n). An **arithmetic shift** keeps the **sign bit**, so a two's complement number keeps its sign. In a **cyclic shift** (rotate), a bit leaving one end **re-enters at the other**. A **mask** is a bit pattern combined with the ACC: **AND** with a 1 in one place **tests** that bit (and AND with a 0 **clears** it), **OR** with a 1 **sets** a bit, and **XOR** with a 1 **flips** it, while all other bits are left unchanged.

masking the byte 01101010

test bit 3	AND 00001000	= 00001000 not zero, so bit 3 is 1
set bit 0	OR 00000001	= 01101011 1 forces the bit on
clear bit 6	AND 10111111	= 00101010 0 forces the bit off
toggle bit 7	XOR 10000000	= 11101010 1 flips the bit

bits are numbered 7 (left) to 0 (right)

Fig. 1: one byte, four masks: test, set, clear and toggle

Worked example: Two changes at once

A greenhouse controller byte at address CONTROL holds B01000101. Bit 4 switches the heater and bit 0 switches the fan. Turn the heater on and the fan off, leaving every other bit unchanged. Give the instructions and the final value in binary and hexadecimal.

1. LDD CONTROL loads 01000101 into the ACC.
2. To set bit 4, OR with B00010000: 01000101 OR 00010000 = 01010101.
3. To clear bit 0, AND with B11111110: 01010101 AND 11111110 = 01010100.
4. STO CONTROL stores the result. 01010100 = 0101 0100 = &54 (84 in denary).

Answer: LDD CONTROL, OR B00010000, AND B11111110, STO CONTROL; 01010100 = &54

Choose the right tool.

- ✓ **Logical:** zeros fill the gap; left multiplies, right divides (unsigned).
- ✓ **Arithmetic:** keeps the sign bit; **cyclic:** bits wrap round.
- ✓ **Test:** AND with 1 in that bit, then CMP #0. **Clear:** AND with 0 in that bit, 1s elsewhere.
- ✓ **Set:** OR with 1 in that bit. **Toggle:** XOR with 1 in that bit.

Watch out!

A logical right shift of a negative two's complement number does not halve it: the zero entering on the left makes it positive. Use an arithmetic shift to keep the sign. To clear a bit, the mask has a 0 in that bit and 1s everywhere else. A mask of 00000001 with AND clears every bit except bit 0.

thelucidstem.com