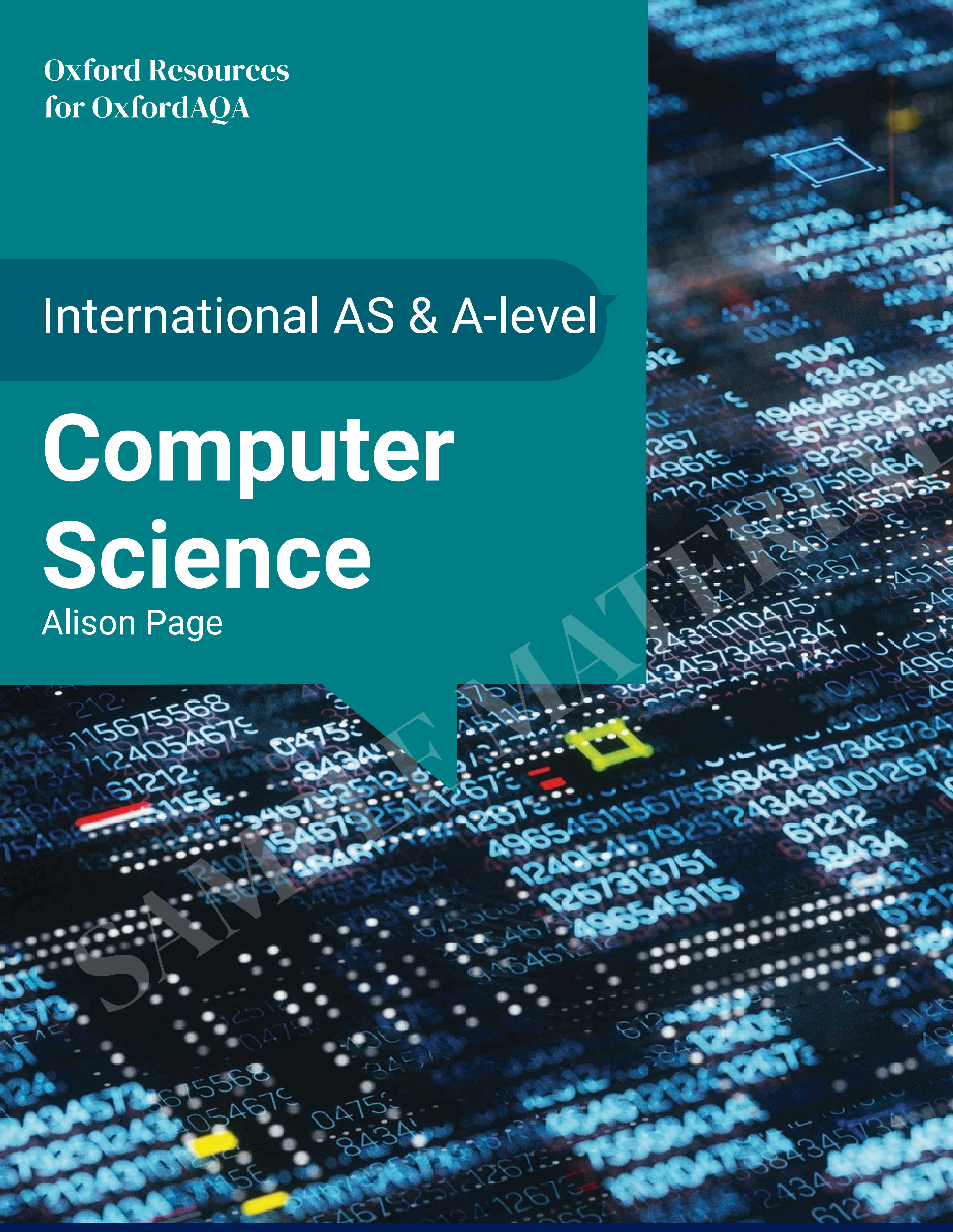
Oxford Resources
for OxfordAQA

International AS & A-level

Computer Science

Alison Page



OXFORD

Contents

To access the answers to all of the Test your understanding and Practice exam questions in this book, visit www.oxfordsecondary.com/oxfordaqa-compsci or scan the QR code.

Chapter 8

Unit 2

Chapter 8 Machine code and assembly language

8.1 Assembly language

Controlling the computer with instruction codes

During the Fetch-Execute cycle, the processor fetches instruction codes from RAM and then executes them (carries them out). A list of all the instruction codes that a computer understands is called the **instruction set** of that computer.

Instruction codes are binary numbers. Each binary number tells the computer to do a different action. Every type of processor has its own **processor instruction set** that works only for that processor. The same binary number can mean different things in different computers.

Objectives

You will be able to:

- understand what assem
  language is
- describe the format of a
  assembly language instruction
- use an address to define an operand.

Key terms

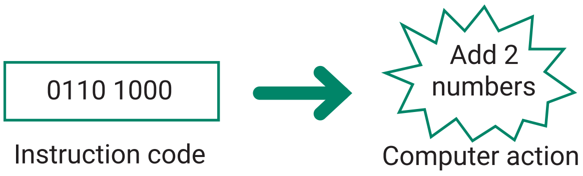
Instruction set: the binary number codes that stand for actions a processor can execute

Processor instruction set: the instruction set of a particular processor

Machine code: a low-level language consisting entirely of numbers, also known as executable code

Assembly language: a programming language that uses mnemonic word codes instead of binary instruction codes

Mnemonic: easy to remember



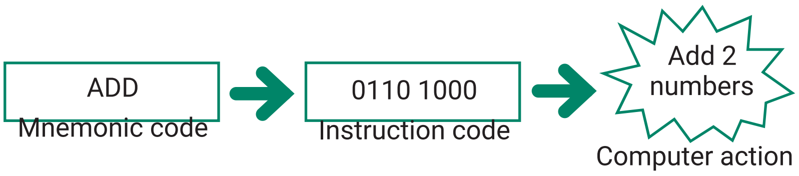
▲ Figure8.1:Amachine code instructionisdecodedand executed.

A program that is made up of binary instruction codes is called a **machine code** file. All software files are made of machine code. Programmers could write software by writing machine code. But this is rare, because it is hard for a human programmer to make a program using number codes. Instead, programs are almost always written in a programming language and then translated into machine code.

Mnemonic letter codes

The first programming language to be invented was **assembly language**. It is very similar to machine code. But instead of using binary number codes, it uses short words of three or four letters. These words are easily converted into number codes.

The words used in assembly language are called **mnemonic** codes. Mnemonic is pronounced 'nu-mon-ic'. It means 'easy to remember'. The mnemonic codes of assembly language exactly match the binary number codes of the instruction set.



◀ Figure 8.2: The mnemonic codes of assembly language can be converted directly into the binary numbers of machine code.

There are different versions of assembly language. Each one uses slightly different mnemonic codes. Exam questions will use a version called standard OxfordAQA assembly language.

Low-level languages

Machine code and assembly language are called **low-level languages**. These languages work directly with the registers and memory locations of the computer. The instructions you write are very similar to the actual codes the processor uses. This gives you very good control over the detailed working of the computer. But it can be hard to write programs in a low-level language.

Key terms

Low-level language: a language that matches closely to the structure of the computer system

Operator: the part of a command that tells the computer what process to carry out **Opcode:** the operator in an assembly language command

Operand: the part of an instruction that tells the computer what data to process **Destination register:** a numbered register used in an instruction, usually to store the result

Building skills 1

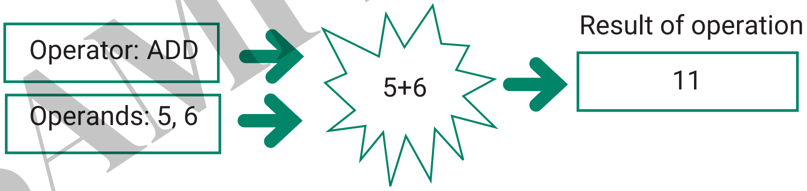
The assembly language used by OxfordAQA is based on the machine code of an ARM processor. Do online research to find out what an ARM processor is.

Operator and operands

In the examples we've seen so far, the instruction told the computer to add two numbers. In a real program, we must also tell the computer what numbers to add. Almost every instruction has two par ts.

- An **operator** is an instruction telling the computer what process to carry out. In assembly language this is shown using an **opcode**.
- The **operand(s)** are the data that has to be processed.

The operands are fetched at the same time as the instruction during the Fetch-Execute cycle.



▲ Figure 8.3: Each instruction consists of an operator and some operands.

The operation creates a new data value, called the result. The result has to be saved. The result is typically saved into a register. A register is a small area of memory inside the processor. The registers are numbered (for example, from 0 to 12). The register where the result is saved is called the **destination register**.

Addressing mode

You have seen that most instructions include operands. That is the data you want the computer to use in the operation. There are three ways to tell the computer what data to use.

- Immediate addressing: the operand is the actual data you want to use (like in Figure 8.3).
- Direct addressing: the operand is a register or address where the data is stored.
- Indirect addressing: the operand is a register, which holds another address where the data is stored (so it takes two steps to find the data).

Key term

Addressing mode: different ways of stating the operands in an assembly language instruction. Immediate addressing means providing the actual data. Direct addressing means giving the location where the data is stored

These three ways of stating the operand are called **addressing modes**.

Building skills 2

Do online research to find a web page that shows the list of mnemonic codes in OxfordAQA assembly language. You will learn all of these commands as you work through this unit.

Structure of an assembly language instruction

To recap, a typical assembly language instruction will have these par ts:

- opcode
- destination register
- addressing mode
- operands.

Not all instructions have exactly this structure. Some commands do not use operands or a destination register. You will learn more in the following sections.

Test your understanding

1. Why is it easier to write a program in assembly language than in machine code?
2. What is the purpose of the destination register?
3. Describe the difference between an operator and an operand.
4. Explain the three modes of addressing used in assembly language.

8.2 Commands in assembly language

Loading and storing data

Each register holds only one data value. When a program runs, the computer **loads** data from memory into the registers, makes some changes to the data, and then **stores** data back into memory.

In OxfordAQA assembly language there are 13 registers. Registers are numbered from R0 to R12. Memory locations are numbers without an R in front. In the exam, the question will tell you what registers and memory locations to use.

LOAD command

The command to load gets data from a memory location and puts it into a register. The command has this structure:

- the opcode LDR
- the destination register, Rd (where the data goes to)
- the memory location (where the data comes from).

You would write the command in this form. Notice the comma between the two operands.

```
LDR Rd, location
```

Here is an example. To load data from memory location 99 into register R0, you would give this command.

```
LDR R0, 99
```

Remember that when an operand identifies a place where the data is held, this is called direct addressing. That is the only type of addressing the load command can use.

Some commands can also use immediate addressing. There is more information about this towards the end of this lesson.

STORE command

The command to store data has this structure. It stores data from the destination register to a memory location.

```
STR Rd, location
```

Here is an example. This command stores the data from register R6 in memory location 23.

```
STR R6, 23
```

Once again, this is an example of direct addressing. That is the only kind of addressing the store command can use.

You can load and store data to a numbered register instead of a memory location. In this case, the second operand will start with an R. That tells you that it is a register. This is also an example of direct addressing.

Objectives

You will be able to:

- load and save data using assembly language commands
- carry out simple calc
 using assembly language instructions.

Key terms

Load: a value is copied from computer memory to the destination register (using direct addressing)

Store: a value is copied from the destination register to computer memory (using direct addressing)

Synoptic link

Registers are very small areas of memory inside the processor that can hold only one value (see section 7.1).

MOVE data between registers

The command MOV (short for **move**) copies data from one register to another. After the command is completed both registers hold the same data.

Key term

Move: a data value is copied into the destination register (using immediate or direct addressing)

```
MOV Rd, operand
```

Let's see the effect of this command. It can be helpful to draw the registers you need to use, and the data that they hold. In this example, R3 holds the data value 16.

R0	R1	R2	R3	R4
			16	

You may see a table like this ready-made in an exam question. Or the question might tell you in words: 'register 3 holds the data value 16'. Here is an example command.

```
MOV R0, R3
```

The command moves data into register R0. The data comes from register R3. After this command, the registers will look like this.

R0	R1	R2	R3	R4
16			16	

All these commands use direct addressing. The operands are locations, and the commands move data between locations. Remember that you will see an R in front of the operand if it is the number of a register. A number on its own is the address of a memory location.

Immediate addressing

Move operations can also use immediate addressing. That means you give the exact data value rather than a numbered location or register. A # symbol in front of the data value tells you it is immediate addressing. Here is an example.

```
MOV R6, #590
```

This command means 'move the data value 590 into register 6'.

Building skills 1

Write assembly language commands to carry out these operations:

- move the number 23 into register 2
- move data to register 0 from register 2
- store data from register 0 to memory location 99.

Carrying out calculations

We can add and subtract data values using assembly language instructions:

- ADD – add two operands and save the result in the destination register
- SUB – subtract one operand from another and save the result in the destination register.

These commands have this structure.

```
ADD Rd, operand 1, operand 2
SUB Rd, operand 1, operand 2
```

For example, this command adds the value in register 3 to the value in register 4 and stores the result in register 0.

```
ADD R0, R4, R3
```

We can use immediate addressing for the second operand. For example, this command subtracts the value 70 from the value in register 4. The result is put into register 6.

```
SUB R6, R4, #70
```

Building skills 2

Writeassemblylanguage commandsto carry out these operations:

- movethenumber 99 intoR1
- addthenumber 23 to thevalueinR1 and put the result in R0
- storethevalue in R0 to memorylocation 55.

Trace the registers

You have learned how to trace an algorithm written in pseudocode. You can also **trace** a program written in assembly language. You can trace the effect of each command on the registers. Some exam questions may ask you to do this. Here is a set of registers showing the starting values.

R0	R1	R2	R3	R4
7	8			

Here is an assembly language command.

```
ADD R2, R0, R1
```

This means 'add the values in R0 and R1, put the result in R2'. After running the command, the registers look like this.

R0	R1	R2	R3	R4
7	8	15		

Key term

Trace (assembly language): write down the values in the registers and numbered memory locations as you work through a program step by step

Synoptic link

You learned how to hand-trace an algorithm in section 3.5.

Test your understanding

The registers of a computer are empty, as shown in this diagram. Memory location 100 holds the data value 6.

R0	R1	R2	R3	R4

1. Show the contents of the registers after this command iscarried out.
   ```
   LDR R0, 100
   ```
2. Show how the contents of the registers change after this additional command is carried out.
   ```
   MOV R1, #99
   ```
3. Show how the contents of the registers change after this additional command is carried out.
   ```
   ADD R2, R1, R0
   ```
4. What value is stored in data location 100 after this command is carried out?
   ```
   STR R2, 100
   ```

8.3 Control program flow

Objectives

You will be able to:

- use conditional branching
- make loops and if struct
  using assembly language.

The program counter

An assembly language program is a series of commands. As the program runs, the program counter counts through the lines of the program one by one. Each line is carried out and then the program counter moves on to the next line.

You can control the program counter using assembly language commands.

- You can stop the program counter with the HALT command.
- You can make the program counter jump to a different place in the program. This is called **branching**.

This gives us control over the flow of the program.

Key terms

Branching: an assembly language command that makes the program jump to a new line

Label: an identifier used in the branch command, showing where to jump to

Branching

You can jump to a different line of the program with a branch command.

- Branch backwards: this can make the program go back and repeat some lines, like a loop in pseudocode.
- Branch forwards: this can make the program miss out some lines, like a selection structure in pseudocode.

Synoptic link

The program counter is a register used in the Fetch-Execute cycle (see section 7.2).

We put **labels** into the program to tell the program where to branch to. The branch command will make the program jump to where the label is.

For example, here is a program without branching. It adds 1 to the value in memory location 99.

```
MOV R0, #1 LDR
R1, 99 ADD R2,
R1, R0 STR R2,
99 HALT
```

Here is the same program with a branch backwards. The new lines are highlighted.

```
   MOV R0, #1
MYLABEL:
   LDR R1, 99
   ADD R2, R1, R0
   STR R2, 99
   B MYLABEL
   HALT
```

Now the program contains a label. In this example it is called 'MYLABEL'. But you can choose any identifier that seems suitable. When the program reaches the command

```
B MYLABEL
```

it branches back to where 'MYLABEL' is. This program will add 1 to the value in location 99. Then it will loop back and do it again. Can you see a problem with this program? There is no way to stop the loop! It will keep adding forever and never halt.

Key term

Conditional branching: a branch command that depends on the result of a comparison between two values

Conditional branching

Mostprograms that include branching use **conditional branching**. Thatmeans the branching is controlled by a logical comparison, usingarelational operator. The command CMP tells the computer to compare two values. For example:

```
CMP R2, #15
```

Thiscompares the value in R2 to the number 15. The result of the comparison is used in the next branch command. For example, BLT means'branch if less than':

```
BLT MYLABEL
```

Here is the program showing this change.

```
   MOV R0, #1
MYLABEL:
   LDR R1, 99
   ADD R2, R1, R0
   STR R2, 99
   CMP R2, #15
   BLT MYLABEL
   HALT
```

This branch command will happen if the value in R2 is less than 15. If R2 is not less than 15, the branch command will not happen. The program will halt.

Table 8.1 shows all the conditional branch commands that you can use.

▼ Table 8.1: Branching commands

Branch command	Meaning
BEQ	Branch if equal
BNE	Branch if not equal
BGT	Branch if greater than
BGE	Branch if greater than or equal to
BLT	Branch if less than
BLE	Branch if less than or equal to

Building skills 1

The example program branches if the value in R2 is less than 15. Rewrite the program so that it branches if the value in R2 is equal to 0.

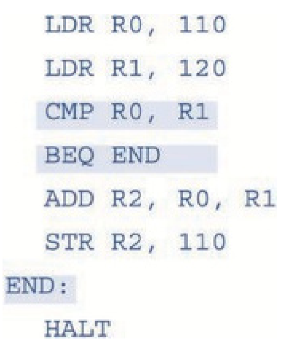
Use branching to make a selection structure

A branch backwards makes the program repeat. That makes a loop. Or you can branch forwards, jumping over some commands. That creates a selection structure (similar to if… else).

For example, this program loads two numbers, adds them together and saves the result to memory.

```
LDR   R0,   110
LDR   R1,   120
ADD R2, R0, R1
STR R2, 110
HALT
```

The next example shows the same program with a branch command added. The branch command skips to the label 'END' if the two values are equal. It misses out the commands to add the two values and store the result. The new lines are highlighted.



```
  LDR R0, 110
  LDR R1, 120
  CMP R0, R1
  BEQ END
  ADD R2, R0, R1
  STR R2, 110
END:
  HALT
```

Building skills 2

- Rewrite this program so that it skips to the end if the value in R1 is less than 0.
- Write an assembly language program that loads two values from memory locations 98 and 99, subtracts the smaller one from the larger one, and saves the result to memory location 100.

Test your understanding

1. What command stops a program?
2. How many operands follow the command CMP?
3. How is a label used in a branch command?
4. Which structure in a pseudocode program is similar to branching backwards in an assembly language program?

8.4 Understand and write programs

Variables, values and calculations

In high-level languages, we use variables. Variables are names given to data locations. Here is an example program written in pseudocode. The variables are called X and Y.

```
X = 40
Y = Y + X
Y = Y - 30
```

Now let's turn the program into assembly language. We will use registers and memory locations to store the values. The ones we have chosen are shown in this table.

Variable	Register	Memory location
X	R0	99
Y	R1	101

In the exam, they will tell you what registers and data locations to use.

Pseudocode	Discussion	Assembly language
`X = 40`	Use immediate addressing to move the value 40 into R0.	`MOV R0, #40`
	Store the value to memory location 99.	`STR R0, 99`
`Y = Y + X`	Use direct addressing to get the value from memory location 101 into register R1.	`LDR R1, 101`
	Add X and Y. The destination register is Y.	`ADD R1, R1, R0`
`Y = Y – 30`	Subtract 30 from Y using immediate addressing. The destination register is Y.	`SUB R1, R1, #30`
	Store Y back into memory location 101.	`STR R1, 101`
	End the program.	`HALT`

You can see that a single line of pseudocode often matches two lines of assembly language. One line loads or saves the value and another line processes the value.

Objectives

You will be able to:

- write an assembly la program to complete a task
- understand what an a language program does
- try out a program usin online simulator.

Building skills 1

Write an assembly language program to match the following pseudocode program. Load and save variables A and B using memory locations 81 and 82.

Use registers R0 and R1 in your code.

```
A = A – 7
B = A + A
```

Programs with branching

Many programs include loops and selection structures. To copy these structures in assembly language, you must use conditional branching. Here is an example requirement.

Two values are stored in memory locations 50 and 51. Find the largest of the two values and store it to location 55.

Let's go through the requirement and turn it into assembly language (see Table 8.2). We will use R0 and R1 to store the two values.

Table 8.2: The assembly language program to meet the example requirement

Requirement	Discussion	Assembly language

Two values are stored in memory locations 50 and 51.	Load the two values from the memory locations to the registers.	`LDR R0, 50`
		`LDR R1, 51`

Find the largest of the two values.	Compare the two values.	`CMP R0,R1`
	If the first value is greater go to the label 'first'.	`BGT first`
	Otherwise, go to the label 'second'.	`B second`

Store the largest value to location 55.	Store the first value and skip to the end.	`first:`
		`STR R0, 55`
		`B end`

	Store the second value.	`second:`
		`STR R1, 55`

	End the program.	`end:`
		`HALT`

Key terms

Assembly language simulator: a web page where you can enter an assembly language program and the program will run as if on a real computer

Understand or explain a program

You may be given a program in assembly language and asked to explain what it does or write pseudocode to match its actions. It is a good idea to trace the assembly language program and see what happens to the values in the registers as you work through the commands one by one.

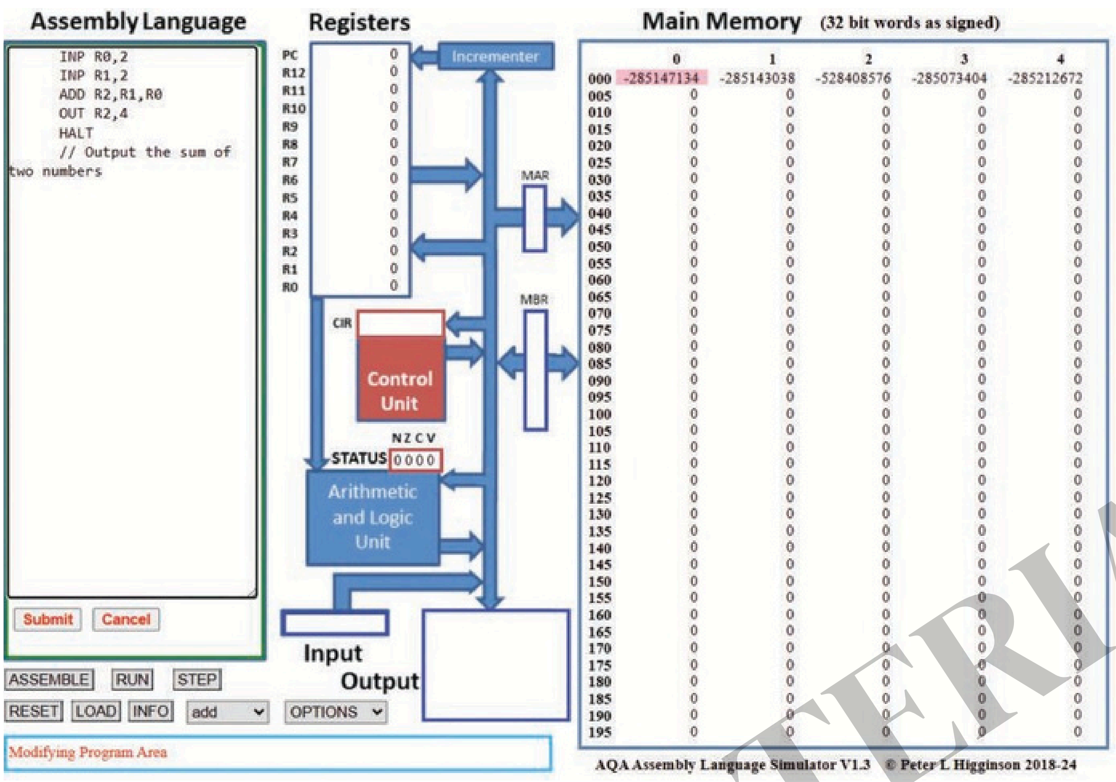
Building skills 2

Trace the program in Table 8.2. Show the contents of the registers at each line. Assume that the locations 50 and 51 hold the values 100 and 200.

Using a simulator

Your programs are written in standard OxfordAQA assembly language. This is not a real assembly language. You cannot run the programs on your computer. Instead, you can trace them by writing down the values in the registers after each line of the program.

There is another way to try out your programs. Clever and helpful people have created online **assembly language simulators**. An online simulator will run a program written in OxfordAQA assembly language as if it were inside a computer. Peter Higginson has made a very good simulator that you can use for free online at www.peterhigginson.co.uk/AQA



```
        INP R0,2
        INP R1,2
        ADD R2,R1,R0
        OUT R2,4
        HALT
        // Output the sum of
two numbers
```

PC	0
R12	0
R11	0
R10	0
R9	0
R8	0
R7	0
R6	0
R5	0
R4	0
R3	0
R2	0
R1	0
R0	0

	0	1	2	3	4
000	-285147134	-285143038	-528408576	-285073404	-285212672
005	0	0	0	0	0
010	0	0	0	0	0
015	0	0	0	0	0
020	0	0	0	0	0
025	0	0	0	0	0
030	0	0	0	0	0
035	0	0	0	0	0
040	0	0	0	0	0
045	0	0	0	0	0
050	0	0	0	0	0
055	0	0	0	0	0
060	0	0	0	0	0
065	0	0	0	0	0
070	0	0	0	0	0
075	0	0	0	0	0
080	0	0	0	0	0
085	0	0	0	0	0
090	0	0	0	0	0
095	0	0	0	0	0
100	0	0	0	0	0
105	0	0	0	0	0
110	0	0	0	0	0
115	0	0	0	0	0
120	0	0	0	0	0
125	0	0	0	0	0
130	0	0	0	0	0
135	0	0	0	0	0
140	0	0	0	0	0
145	0	0	0	0	0
150	0	0	0	0	0
155	0	0	0	0	0
160	0	0	0	0	0
165	0	0	0	0	0
170	0	0	0	0	0
175	0	0	0	0	0
180	0	0	0	0	0
185	0	0	0	0	0
190	0	0	0	0	0
195	0	0	0	0	0

▲ Figure 8.4: Peter Higginson's AQA assembly language simulator.

To use this simulator:

- copy all the assembly language commands into the box on the le of the screen and click the 'Submit' button that appears
- put any starting values into the numbered memory locations
- click on the 'run' button (or 'step' to go through the program line by line).

If this website is not available, carry out an online search for 'AQA assembly language simulator' to find other examples. Many simulators have minor limitations:

- in some simulators if you want to put starting values in the registers, you have to load them from memory locations
- the conditional branching commands BGE and BLE do not work in all simulators.

Building skills 3

Run the programs given in this section using an online simulator. If you have time, practice writing and running more assembly language programs.

Test your understanding

1. This pseudocode command assigns a value to a variable. You have to write assembly language to match the command. What type of addressing would you use?

```
    age = 26
```

2. The result of a program is stored in R5. Write an assembly language command to store this result to memory location 100.

3. A pseudocode program includes a structure that begins with the key word while. What assembly language technique would you use to match this structure?

4. Two values are stored in registers R1 and R6. If the two values are the same, the program must jump to the label 'start'. Write the assembly language commands to make this happen.

8.5 Bitwise operations

Working with binary values

Inside the computer, values are always stored in binary form. We do not usually think too much about the binary numbers. We just work with the values. But we can use assembly language commands to make changes to the 1s and 0s of binary numbers – the bits. These are called **bitwise operations**.

Binary shift

Shifting the bits in a binary number to the left or right multiplies or divides the number by a power of two. An assembly language command called **logical shift** will do this task. There are two logical shift commands:

Command	What it does	Explanation
LSL	Logical shift left	Multiplies a binary number by a power of two
LSR	Logical shift right	Divides a binary number by a power of two

For example, the binary number 00100100 is stored in register R0. That binary number has the value 36. This command will shift all the bits in R0 one place to the left. The destination register is R1.

```
LSL R1, R0, #1
```

After the command has run the registers look like this.

R0	R1	R2
00100100 (36)	01001000 (72)	

The bits have shifted one place to the left. This has multiplied the number by 2, giving a final value of 72.

Boolean operators

There is an assembly language command to match each Boolean operator.

Boolean operator	Output of the gate	Operator in assembly language
NOT	Changes the value of each bit to its opposite	*MVN*
AND	True only if both input values are True	*AND*
OR	True if either or both input values are True	*ORR*
XOR	True if one input value is True and the other is False	*EOR*

Objectives

You will be able to:

- use binary shift to multip
  divide binary numbers
- perform bitwise operatic
  using Boolean operators.

Key terms

Bitwise operation: an assembly language command that makes changes to the bits in a binary data value

Logical shift: an assembly language command that shifts all the bits in a byte to the right or left

Synoptic link

You used binary shift to multiply and divide binary numbers in section 5.3.

Building skills 1

Change the assembly language command so that it divides the same binary number by 4. Show the values in the registers after this command has run.

Synoptic link

You learned about Boolean logic gates in section 7.5.

The Boolean operators can be used to combine two binary numbers to make a new binary number. In bitwise operations, we use 1 to stand for True and 0 to stand for False.

In the following examples, we will use these two binary numbers held in registers R0 and R1.

R0	R1	R2
01001010	01101001	

NOT operator (MVN)

You have already learned that the move command ***MOV*** moves a value from one register to another. An alternative is the 'move negative' command ***MVN***. This moves the value, but also swaps the bits, so that every 1 becomes a 0, and every 0 becomes a 1.

Here is an example of a 'move negative' command.

```
MVN R2, R0
```

After the command has run, the registers look like this.

R0	R1	R2
01001010	01101001	10110101

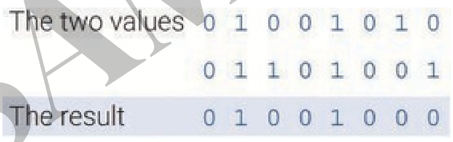
The MVN command works like the Boolean NOT gate – it reverses the value of each bit.

AND operator

Here is an example of an AND operation. The destination register is R2.

```
AND R2, R0, R1
```

To see how this command works, put one binary number above the other and compare the bits in each column. The number value doesn't matter. In columns where both bits are 1, the result is 1. There are only two places where both bits are 1. In all other cases the result is 0.



The two values	0	1	0	0	1	0	1	0
	0	1	1	0	1	0	0	1
The result	0	1	0	0	1	0	0	0

After this command has run, the registers look like this. The result is in R2.

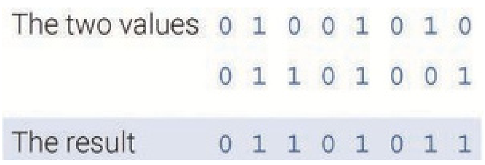
R0	R1	R2
01001010	01101001	01001000

OR operator (ORR)

In assembly language, the OR operator is written as***ORR***. Let's look at an example.

```
ORR R2, R0, R1
```

To see how this command works, put one binary number above the other. In columns where there is at least one 1, the result is 1. In all other columns the result is 0.



The two values	0	1	0	0	1	0	1	0
	0	1	1	0	1	0	0	1
The result	0	1	1	0	1	0	1	1

After this command has run, the registers look like this

R0	R1	R2
01001010	01101001	01101011

Key term

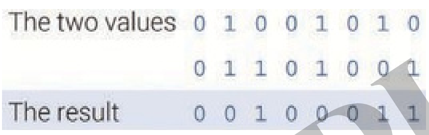
Exclusive-OR: a Boolean operator that returns True if the values of the two operands are different. The logic gate is called XOR. In AQA assembly language, it is written EOR

EXCLUSIVE-OR (EOR)

In assembly language, **exclusive-OR** is written as***EOR***. Let's look at an example.

```
EOR R2, R0, R1
```

In columns where the bits are different the result is 1. In all other cases, the result is 0.



The two values	0	1	0	0	1	0	1	0
	0	1	1	0	1	0	0	1
The result	0	0	1	0	0	0	1	1

After this command has run, the registers look like this.

R0	R1	R2
01001010	01101001	00100011

Building skills 2

Write an assembly language program to load two binary numbers from memory and combine them using the bitwise operators. Use a different destination register for the result of each operation.

Test your understanding

1. What is the mathematical effect of shifting the bits in a binary number to the left?
2. What assembly language command is equivalent in effect to the Boolean NOT operator?
3. What is the difference between the assembly language operators ***ORR*** and ***EOR***?
4. R0 holds 00001001 and R1 holds 00000001. What is the result of the following command?

   ```
   EOR R2, R0, R1
   ```

Chapter 8 Revision and exam practice

Revision checklist

Can you do the following?

Instruction format (see pages 176–178)

- ☐ Understand the term 'processor instruction set' and know that an instruction set is processor specific.
- ☐ Know that instructions consist of an opcode, an addressing mode and one or more operands (value, memory address or register).
- ☐ Know that the format of an instruction, in machine code or assembly language, may be dependent on the type of instruction.
- ☐ Understand and apply immediate, direct and indirect addressing modes.
- ☐ Know that in machine code instructions are expressed in binary and that in assembly language they are expressed as mnemonics.

Assembly language programming (see pages 179–190)

- ☐ Understand and apply the basic operations of:
  - load
  - add
  - subtract
  - store
  - branching (conditional and unconditional)
  - compare
  - logical bitwise operators (AND, OR, NOT, XOR)
  - logical
    - shift right
    - shift left
  - halt.
- ☐ Use the basic operations to write, trace and reason about assembly language programs using immediate, direct and indirect addressing modes.

Practice exam questions

Consider this assembly language program.

1.

```
    MOVR0,   #0
    MOVR3,   #0
    LDRR1,   98
    LDRR2, 99
loop:
    ADDR3,  R3,  R1
    ADDR0,  R0,  #1
    CMPR0,  R2  BLT
    loop STRR3, 98

       HALT
```

a) State the name of the addressing mode of the final operand in the instruction

`ADD R0, R0, #1` (1 mark)

b) Memory location 98 holds the number 11. Memory location 99 holds the number 2. Create a trace table to show how the contents of the memory locations and registers change when the program is executed. (4 marks)

c) State the purpose of the program. (2 marks)

d) A programmer considered using the single bitwise opcode ***LSL*** instead of this program. Contrast the two approaches and explain any advantages or limitations of each approach. (4 marks)

2. This algorithm compares two positive integers X and Y and subtracts the smaller from the larger.

```
IF X < Y THEN
Z = Y - X

ELSE
  Z = X - Y
ENDIF
```

a) Write an assembly language program, using the OxfordAQA assembly language instruction set, to implement this algorithm:

- load the values of X and Y from memory locations 55 and 56
- compare the values and subtract as shown in the algorithm
- store the result Z to memory location 99.

You may use any numbered registers to hold the values as the program runs. (6 marks)

b) The program has been written in assembly language. You could also write the program in machine code, or in a high-level language. Discuss the advantages and disadvantages of writing a program in assembly language compared to the alternatives. (4 marks)

The exam paper will include a list OxfordAQA assembly language. You will find this list on page 375. Good understanding of assembly language will help you to make effective use of the list. The list uses terms such as 'operand' and 'bitwise', so you need to remember and understand what these terms mean. The list will be a big help as you write assembly language code in the exam.

Oxford Resources for OxfordAQA

International AS & A-level

Computer Science

Written byanexpertin ComputerScience, this Student Bookprovides:

- Full alignmentwith the OxfordAQA International AS & A-level Computer Science (9645) specification for first teaching from September 2024
- Specialist knowledge and opportunities for skills development to build understanding and application of on-screen programming
- Globally relevant programming languages that continue those used at International GCSE
- Assessment opportunities with practice exam questions, exam tips and revision support throughout
- Key terms and specialist vocabulary clearly highlighted and explained

kerboodle

Digital Student Book available for use in class or at home on Kerboodle.

OXFORD UNIVERSITY PRESS

www.oup.com

How to get in touch:
web www.oxfordsecondary.com
email schools.enquiries.uk@oup.com
tel +44 (0)1536 452620

ISBN 978-1-382-06670-9
9 781382 066709