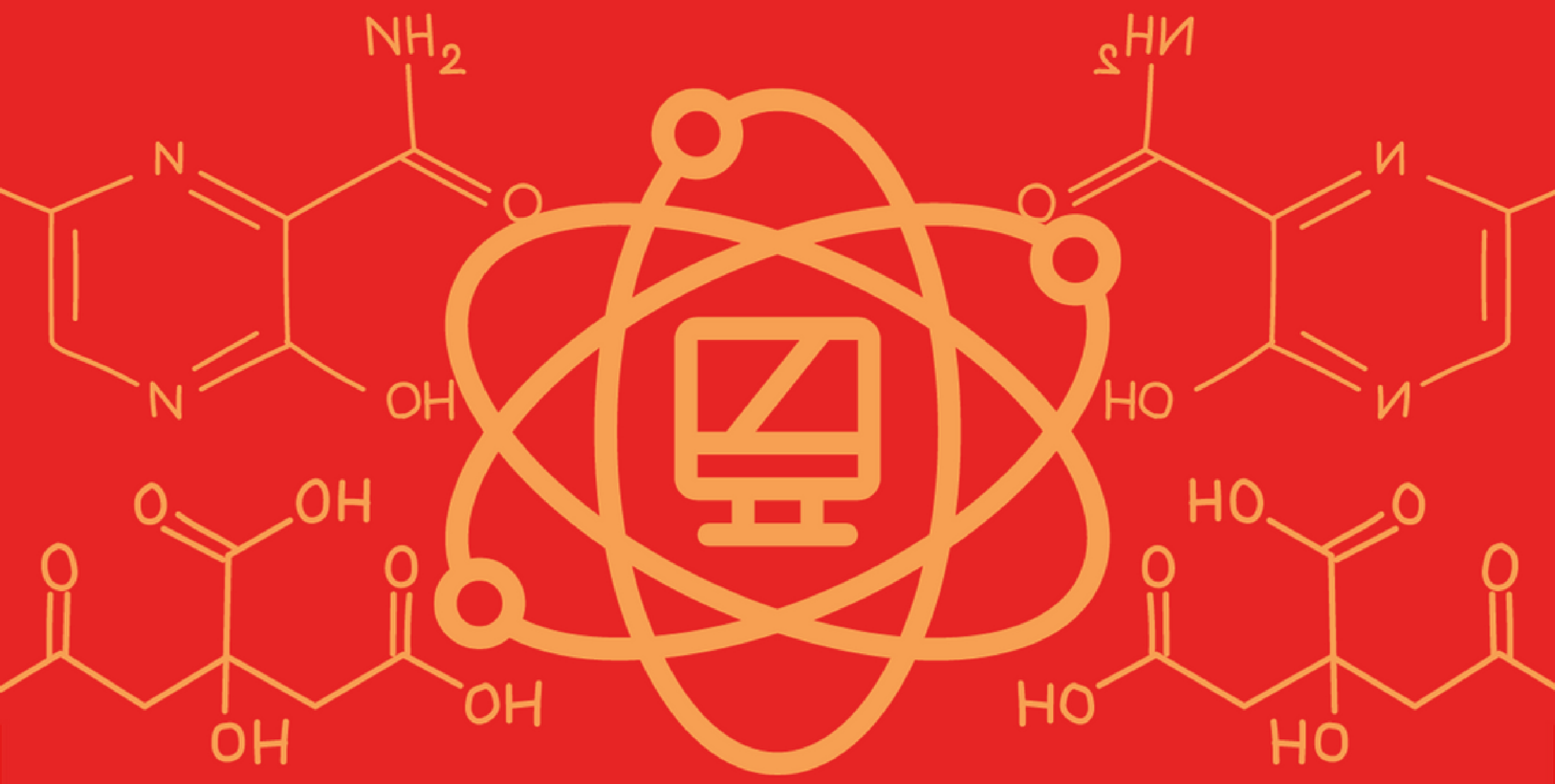
EXAM PAPERS PRACTICE

- Boost your performance and confidence with these topic-based exam questions
- Practice questions created by actual examiners and assessment experts
- Detailed mark scheme
- Suitable for all boards
- Designed to test your ability and thoroughly prepare you

Assembly Language



CIE AS & A Level

Computer Science Revision Notes 9618

Syllabus Content
4.2 Assembly language
show understanding of the relationship between assembly language and machine code, including symbolic and absolute addressing, directives and macros
describe the different stages of the assembly process for a 'two-pass' assembler for a given simple assembly language program
trace a given simple assembly language program

Machine code:–

Machine code, also known as machine language, is the elemental language of computers, comprising a long sequence of binary digital zeros and ones (bits).

Simple instructions that are executed directly by the CPU. Each instruction performs a very specific task, such as a **load**, a **jump**, or an **ALU operation** on a unit of data in a CPU register or memory.

Every program directly executed by a CPU is made up of a series of such instructions.

Machine code may be regarded as the lowest-level representation of a compiled or assembled computer program or as a primitive and hardware-dependent programming language.

While it is possible to write programs directly in machine code, it is **tedious and error prone** to manage individual bits and calculate numerical addresses and constants manually. For this reason machine code is almost never used to write programs.

Different processors have different instruction sets associated with them. Even if two different processors have the same instruction, the machine codes for them will be different but the structure of the code for an instruction will be similar for different processors.

Machine code instruction: A binary code with a defined number of bits that comprises an opcode and, most often, one operand.
For a particular processor, the following components are defined for an individual **machine code instruction:**

The total number of bits or bytes for the whole instruction
The number of bits that define the opcode
The number of operands that are defined in the remaining bits
Whether the opcode occupies the most significant or the least sigr:iificant bits.

Almost all practical programs today are written in higher-level languages or **assembly language**. The **source code** is then translated to **executable machine code** by utilities such as **interpreters, compilers, assemblers, and/or linkers.**

Assembly language

A programmer might wish to write a program where the actions taken by the processor are directly controlled. It is argued that this can produce optimum efficiency in a program.

However, writing a program as a sequence of machine code instructions would be a very time-consuming and error-prone process. The solution for this type of programming is to use assembly language. As well as having a uniquely defined machine code language each processor has its own assembly language.

The essence of assembly language is that for each machine code instruction there is an equivalent assembly language instruction which comprises:

a mnemonic (a symbolic abbreviation) for the opcode

a character representation for the operand.

If a program has been written in assembly language it has to be translated into machine code before it can be executed by the processor. The translation program is called an 'assembler'

The fact that an assembler is to be used allows a programmer to include some special features in an assembly language program. Examples of these are:
comments
symbolic names for constants
labels for addresses
macros
subroutines
directives
System calls.

The first three items on this list are there to directly assist the programmer in writing the program. Of these, comments are removed by the assembler and symbolic names and labels require a conversion to binary code by the assembler.

Macro: A macro or a subroutine contains a sequence of instructions that is to be used more than once in a program.

Directives: and system calls are instructions to the assembler as to how it should construct the final executable machine code. They can involve directing how memory should be used or defining files or procedures that will be used. They do not have to be converted into binary code.

Assembly Language:

Amongst others, the following instructions are important for all processors:

LDD - Loads the contents of the memory address or integer into the accumulator
ADD - Adds the contents of the memory address or integer to the accumulator
STO - Stores the contents of the accumulator into the addressed location

Assembly code is easy to read interpretation of machine code, there is a one to one matching; one line of assembly equals one line of machine code:
Machine code Assembly code
000000110101 = Store 53
Let's take a look at a quick coding example using assembly code.

LDM #23: *Loads the number 23 into the accumulator.*

ADD #42*: Adds the number 42 to the contents of the accumulator = 65.*
STO 34: *Saves the accumulator result to the memory address 34.*

The code above is the equivalent of saying x = 23 + 42 in VB language.

Addressing modes

When an instruction requires a value to be loaded into a register there are different ways of identifying the value.

These different ways are described as the 'addressing modes'. In Section 6.01, it was stated that, for our simple processor, two bits of the opcode in a machine code instruction would be used to define the addressing mode. This allows four different modes which are described in Table.

Addressing mode	Operand
Immediate	The value to be used in the instruction
Direct	An address which holds the value to be used in the instruction
Indirect	An address which holds the address which holds the value to be used in the instruction
Indexed	An address to which must be added what is currently in the index register (IX) to get the address which holds the value in the instruction

You might notice that some instructions use **"#"** and others don't
 # = number**, [No hash] =** address

Let's take a look at a quick example:

Instruction		Explanation
Op Code	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
STO	<address>	Store the contents of ACC at the given address
ADD	<address>	Add the contents of the given address to 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)
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
JPN	<address>	Following a compare instruction, jump to <address> if the compare was False
IN		Key in a character and store its ASCII value in ACC
OUT		Output to the screen the character whose ASCII value is stored in ACC
END		Return control to the operating system

All questions will assume there is only one general purpose register available **(Accumulator)**
 # ACC denotes **Accumulator**
IX denotes **Index Register**
# denotes **immediate addressing**
B denotes a binary number, **e.g. B01001010**
& denotes a **hexadecimal number, e.g. &4A**

Assembly code	Main memory start	Main memory end

Let's take a look at doing this without the hashes:

Assembly code	Main memory start		Main memory end	
	Address	Contents	Address	Contents
	10	9	10	9
	11	2	11	2
LDD 10 ADD12 STORE 12	12	7	12	<u>**16**</u>
	13	10	13	10
	14	12	14	12

This code loads the value stored in memory **location** 10 into the accumulator (9), then adds the value stored in memory **location** 12 (7), it then stores the result into memory location 12 (9 + 7 = 16).

Data movement:

These types of instruction can involve loading data into a register or storing data in memory.

Instruction opcode	Instruction operand	Explanation
LDM	#n	Immediate addressing loading *n* to ACC
LDR	#n	Immediate addressing loading *n* to IX
LDD	<address>	Direct addressing, loading to ACC
LDI	<address>	Indirect addressing, loading to ACC
LDX	<address>	Indexed addressing, loading to ACC
STO	<address>	Storing the contents of ACC

Arithmetic operation:

Instruction opcode	Instruction operand	Explanation
ADD	<address>	Add the address content to the content in the ACC
INC	<register>	Add 1 to the value stored in the specified register
DEC	<register>	Subtract 1 from the value stored in the specified register

Comparisons and jumps:

A program might require an unconditional jump or might only need a jump if a condition is met. In the latter case, a compare instruction is executed first and the result of the comparison is recorded by a flag in the status register.
The execution of the conditional jump instruction begins by checking whether or not the flag bit has been set.

Instruction opcode	Instruction operand	Explanation
JMP	<address>	Jump to the address specified
CMP	<address>	Compare the ACC content with the address content
CMP	#n	Compare the ACC content with *n*
JPE	<address>	Jump to the address if the result of the previous comparison was TRUE
JPN	<address>	Jump to the address if the result of the previous comparison was FALSE

Assemblers:

A computer program that translates programming code written in Assembly language to machine code is known as assemblers.
Assemblers can be **One-Pass Assembler** or **Two-Pass Assembler**

Two-Pass Assembler:

1st Pass

1. Data items are converted to their binary equivalent
2. Any directives are acted upon
3. Any symbolic addresses are added to the symbolic address table

2nd Pass

1. Forward references are resolved
2. Any symbolic address is replaced by an absolute address.

Two-Pass Assembler explanation:

Consider an assembler instruction like the following

JMP LATER

...

...
LATER:

This is known as a **forward reference**.

If the assembler is processing the file one line at a time, then it doesn't know where LATER is when it first encounters the jump instruction.
So, it doesn't know if the jump is a short jump, a near jump or a far jump. There is a large difference amongst these instructions.
They are 2, 3, and 5 bytes long respectively.
The assembler would have to guess how far away the instruction is in order to generate the correct instruction.
If the assembler guesses wrong, then the addresses for all other labels later in the program would be wrong, and the code would have to be regenerated. Or, the assembler could always choose the worst case.
But this would mean generating inefficiency in the program, since all jumps would be considered far jumps and would be 5 bytes long, where actually most jumps are short jumps, which are only 2 bytes long.

So how it solves the problem by Two-Pass Assembler?
Answer:
Scan the code twice.
The first time (One-Pass) , just count how long the machine code instructions will be, just to find out the addresses of all the labels.
Also, create a table that has a list of all the addresses and where they will be in the program. This table is known as the **symbol table.**
On the second scan (Second-Pass), generate the machine code, and use the symbol table to determine how far away jump labels are, and to generate the most efficient instruction.

This is known as a **two-pass assembler**. Each pass scans the program,

The first pass generates the symbol table and the second pass generates the machine code. I have created a listing of an assembler program that has the machine code listed, and the symbol table listed.

4.2.2 Stages of assembly

Before a program written in assembly language (**source code**) can be executed, it needs to be translated into machine code. The translation is performed by a program called an **assembler**. An assembler translates each assembly language instruction into a machine code instruction. An assembler also checks the syntax of the assembly language program to ensure that only opcodes from the appropriate machine code **instruction set** are used. This speeds up the development time, as some errors are identified during translation before the program is executed.

There are two types of assembler: single pass assemblers and two pass assemblers. A single pass assembler puts the machine code instructions straight into the computer memory to be executed. A two pass assembler produces an object program in machine code that can be stored, loaded then executed at a later stage. This requires the use of another program called a loader. Two pass assemblers need to scan the source program twice, so they can replace labels in the assembly program with memory addresses in the machine code program.

Label → Total; Memory address → 0140

`LDD Total`	`0140`
Assembly language mnemonics	**Machine code hexadecimal**

Pass 1

- Read the assembly language program one line at a time.
- Ignore anything not required, such as comments.
- Allocate a memory address for the line of code.
- Check the opcode is in the instruction set.
- Add any new labels to the symbol table with the address, if known.
- Place address of labelled instruction in the symbol table.

Pass 2

- Read the assembly language program one line at a time.
- Generate **object code**, including opcode and operand, from the symbol table generated in Pass 1.
- Save or execute the program.

The second pass is required as some labels may be referred to before their address is known. For example, `Found` is a forward reference for the `JPN` instruction.

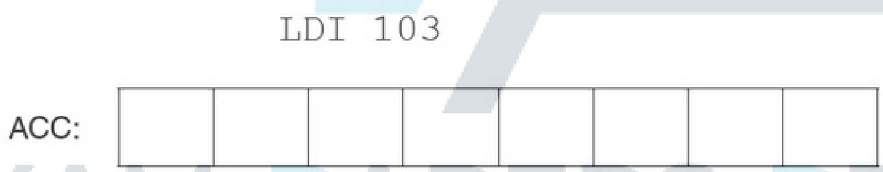
Label	Opcode	Operand
Notfound:	LDD	200
	CMP	#0
	JPN	Found
	JPE	Notfound
Found:	OUT	

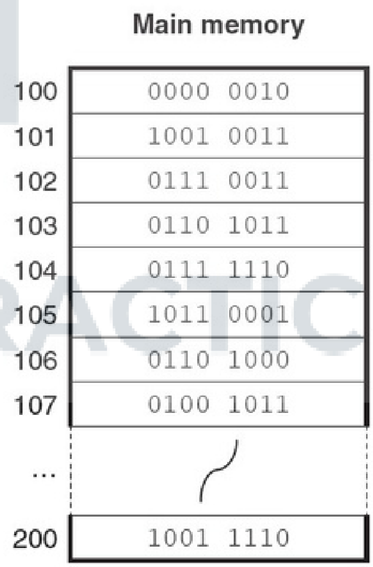
If the program is to be loaded at memory address 100, and each memory location contains 16 bits, the symbol table for this small section of program would look like this:

Label	Address
Notfound	100
Found	104

Exam Style Questions:

Indirect Addressing:





Answer:

0	1	0	0	1	0	1	1

- Memory address 103 contains the value 107
- So address 107 is the address from which to load the data

Indexed Addressing

LDX 800

Index Register:	0	0	0	0	1	0	0	1

Answer:

Accumulator:								

Accumulator:	1	1	0	0	0	0	1	0

- Index Register contains: **00001001 = 9**
- **800 + 9 = 809**

Exam-style Questions

Q. Complete the trace table below for the following assembly language program.

800	LDD 810
801	INC ACC
802	STO 812
803	LDD 811
804	ADD 812
805	STO 813
806	END
...	
810	28
811	41
812	0
813	0

Answer on next page

Answer:
- **LDD 810** (28 Loaded in ACC)
- **INC ACC** (Accumulator incremented with 28++1 = 29, 29 written in ACC)
- **STO 812** (29 Stored at Memory Location 812)
- **LDD 811** (Loaded contents of memory location 811 in ACC)
- **ADD 812** (Added 41 with 29, Contents of ACC added with memory loc 812)
- **STO 813** (Stored contents of ACC in 813 memory location)

ACC	Memory address			
	810	811	812	813
	28	41	0	0
28				
29				
			29	
41				
70				
				70

PastPaper Questions

2. Assemblers translate from assembly language to machine code. Some assemblers scan the assembly language program twice; these are referred to as two-pass assemblers.
The following table shows five activities performed by two-pass assemblers.
Write 1 or 2 to indicate whether the activity is carried out during the first pass or during the second pass.

Activity	First pass or second pass
any symbolic address is replaced by an absolute address	
any directives are acted upon	
any symbolic address is added to the symbolic address table	
data items are converted into their binary equivalent	
forward references are resolved	

Answer

2

Activity	First pass or second pass
any symbolic address is replaced by an absolute address	2
any directives are acted upon	1
any symbolic address is added to the symbolic address table	1
data items are converted into their binary equivalent	1
forward references are resolved	2

[5]