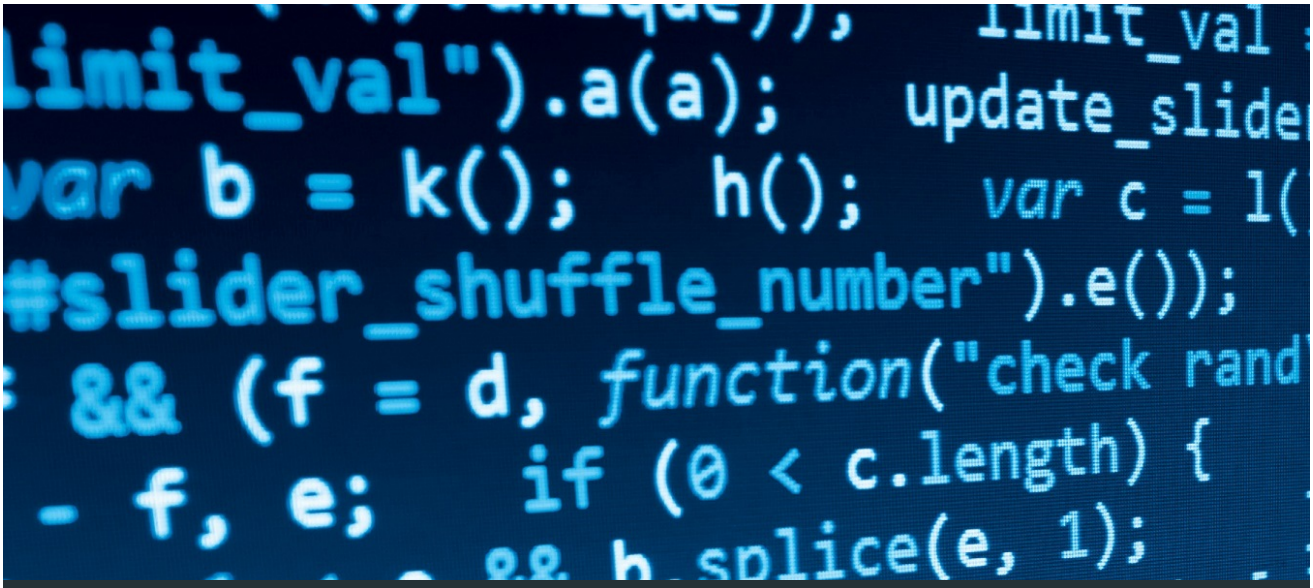
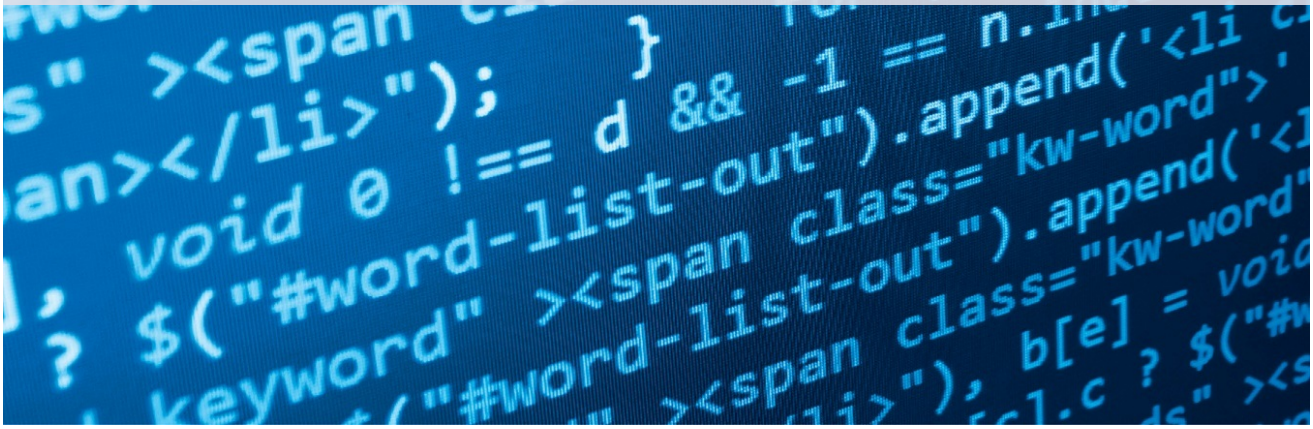


Chapter 6:
Assembly language programming

Learning objectives

By the end of this chapter you should be able to:

- show understanding of the relationship between assembly language and machine code
- describe the different stages of the assembly process for a two-pass assembler
- trace a given simple assembly language program
- show understanding that the set of instructions are grouped into instructions for:
  - data movement
  - input and output of data
  - arithmetic operations
  - unconditional and conditional jumps
  - comparisons
- show understanding of modes of addressing
- show understanding of and perform binary shifts.



6.01 Machine code instructions

We need to start with a few facts.

- The only language that the CPU recognises is machine code.
- Machine code consists of a sequence of instructions.
- An instruction contains an **opcode**.
- An instruction may not have an **operand** but up to three operands are possible.
- Different processors have different instruction sets associated with them.
- Different processors will have comparable instructions for the same operations, but the coding of the instructions will be different.

For a particular processor, the following must be defined for each 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 significant bits.

We will consider a simple system where there is either one or zero operands. This simple system will be assumed to have a 16-bit address bus width. Following on from the approach in Chapter 5 (Section 5.02), the system will have the accumulator as the only general purpose register.

The number of bits needed for the opcode depends on the number of different opcodes in the instruction set for the processor. The opcode can be structured with the first few bits defining the operation and the remaining bits associated with addressing. A sensible instruction format for our simple processor is shown in Figure 6.01.

Opcode			Operand
Operation	**Address mode**	**Register addressing**	
4 bits	2 bits	2 bits	16 bits

Figure 6.01 A simple instruction format

This has an eight-bit opcode consisting of four bits for the operation, two bits for the address mode (discussed in Section 6.05) and the remaining two bits for addressing registers. This allows 16 different operations each with one of four addressing modes. This opcode will occupy the most significant bits in the instruction. Because in some circumstances the operand will be a memory address it is sensible to allocate 16 bits for it. This is in keeping with the 16-bit address bus.

When an instruction arrives in the CPU the control unit checks the opcode to see what action it defines. This first step in the decode stage of the fetch–execute cycle can be described using the register transfer notation which was introduced in Chapter 5 (Section 5.07). However, a slight amendment is needed to the format. The following shows the transfer of bits 16 to 23, which represent the opcode, from the current instruction register to the control unit:

```
CU ← [CIR(23:16)]
```

6.02 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 is the most efficient type of program. However, writing a substantial program as a sequence of machine code instructions would take a very long time and there would be inevitably lots of errors along the way. 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**.

Using an assembly language, the programmer has the advantage of the coding being easier to write than it would have been in machine code. In addition, the use of the assembler allows a programmer to include some special features in an assembly language program. Examples of some of these are:

- comments
- symbolic names for constants
- labels for addresses
- macros
- directives.

A macro is a sequence of instructions that is to be used more than once in a program. A **directive** is an instruction to the assembler as to how it should construct the final executable machine code. This might be to direct how memory should be used or to define files or procedures that will be used.

Discussion Point:

Although writing a program in assembly language is much easier than using machine code, many would argue that its use is no longer justified. Can you investigate the arguments for and against?

6.03 Symbolic, relative and absolute addressing

When considering how an assembler would convert an assembly language program into machine code it is necessary to understand the difference between symbolic, relative and absolute addressing. To explain these, we can consider a simple assembly language program which totals single numbers input at the keyboard. Table 6.01 shows the program as it would be written using symbolic addressing together with an explanation of each instruction.

Assembly language program using symbolic addressing	Explanation of each instruction
`IN`	A single number is input at the keyboard and its ASCII code is stored in the accumulator
`SUB #48`	This subtraction converts the ASCII code into the binary code for the number (see Task 6.01)
`STO MAX`	The number in the accumulator is stored at the address labelled MAX:
`LDM #0`	Loads zero into the accumulator
`STO TOTAL`	The zero in the accumulator is stored at the address labelled TOTAL:
`STO COUNT`	The zero in the accumulator is stored at the address labelled COUNT:
`STRTLP:IN`	A single number is input at the keyboard and its ASCII code is stored in the accumulator
`SUB #48`	This subtraction converts the ASCII code into the binary code for the number
`ADD TOTAL`	Adds the value at address labelled TOTAL: to the value in the accumulator and stores the sum in the accumulator
`STO TOTAL`	The number in the accumulator is stored at the address labelled TOTAL:
`LDD COUNT`	Loads the value stored at address COUNT: into the accumulator
`INC ACC`	Adds 1 to the value in the accumulator
`CMP MAX`	Compares the value in the accumulator with the value stored at address MAX:
`JPN STRTLP`	If the compared values are not equal the program jumps to the instruction labelled STRTLP:
`END`	The execution of the program has finished
`MAX:`	A labelled address where a value can be stored
`TOTAL:`	A labelled address where a value can be stored
`COUNT:`	A labelled address where a value can be stored

Table 6.01 An assembly program using symbolic addressing with explanations

The convention has been followed that a label is written with a following colon which is ignored when the label is referenced. Note how the code is dominated by the use of the accumulator.

TASK 6.01

Check the ASCII coding table to see why the subtraction in Table 6.01 works.

The use of symbolic addressing allows a programmer to write some assembly language code without having to bother about where the code will be stored in memory when the program is run. However, it is possible to write assembly language code where the symbolic addressing is replaced by either relative addressing or absolute addressing. Table 6.02 shows the simple code from Table 6.01 converted to use these alternative approaches.

Assembly language program using relative addressing	Assembly language program using absolute addressing	
(0) IN	(200)	IN
(1) SUB #48	(201)	SUB #48
(2) STO [BR] + 15	(202)	STO 215
(3) LDM #0	(203)	LDM #0
(4) STO [BR] + 16	(204)	STO 216
(5) STO [BR] + 17	(205)	STO 217
(6) IN	(206)	IN
(7) SUB #48	(207)	SUB #48
(8) ADD [BR] + 16	(208)	ADD 216
(9) STO [BR] + 16	(209)	STO 216
(10) LDD [BR] + 17	(210)	LDD 217
(11) INC ACC	(211)	INC ACC
(12) CMP [BR] + 15	(212)	CMP 215
(13) JPN [BR] + 7	(213)	JPN 207
(14) END	(214)	END
(15)	(215)	
(16)	(216)	
(17)	(217)	

Table 6.02 A simple assembly language program using relative and absolute addressing

For the relative addressing example, the assumption is that a special-function base register BR contains the base address. The contents of this register can then be used as indicated by [BR]. Note that there are no labels for the code. The left-hand column is just for illustration identifying the offset from the base address which is the address of the first instruction in the program.

For the absolute address example there are again no labels for the code. The left-hand column is again just for illustration but this time identifying actual memory addresses. This has been coded with the understanding that the first instruction in the program is to be stored at memory address 200.

6.04 The assembly process for a two-pass assembler

For any assembler there are a number of things that have to be done with the assembly language code before any translation can be done. Some examples are:

- removal of comments
- replacement of a macro name used in an instruction by the list of instructions that constitute the macro definition
- removal and storage of directives to be acted upon later.

A two-pass assembler is designed to handle programs written in the style of the one illustrated in Table 6.01. This program contains forward references. Some of the instructions have a symbolic address for the operand where the location of the address is not known at that stage of the program. A two-pass assembler is needed so that in the first pass the location of the addresses for forward references can be identified.

To achieve this during the first pass the assembler uses a symbol table. The code is read line by line. When a symbolic address is met for the first time its name is entered into the symbol table. Alongside the name a corresponding address has to be added as soon as that can be identified. Table 6.03 shows a possible format for the symbol table that would be created for the program shown in Table 6.01.

Symbol	Offset
MAX	+15
TOTAL	+16
COUNT	+17
STRTLP	+7

Table 6.03 A completed symbol table for the assembly language program in Table 6.01

Note that the assembler has to count the instructions as it reads the code. Then when it encounters a label it can enter the offset value into the symbol table. In this example the first entry made in the offset column is the +7 for STRPLP.

For the second pass the Assembler uses the symbol table and a lookup table that contains the binary code for each opcode. This table would have an entry for every opcode in the set defined for the processor. Table 6.04 shows entries only for the instructions used in the simple program we are using as an example. Note that the binary codes are just suggestions of codes that might be used.

Opcode mnemonic	Opcode binary
IN	0001 0000
SUB	0110 0001
STO	0100 0100
LDM	0010 0001
ADD	0100 0101
LDD	0010 0101
INC	0101 0101
CMP	1000 0100
JPN	1010 0100
END	1111 1111

Table 6.04 An opcode lookup table

Provided that no errors have been identified, the output from the second pass will be a machine code program. For our example, this code is shown in Table 6.05 along with the original assembly code for comparison.

Machine code		Assembly code	
Opcode	Operand		
0001 0000			IN
0110 0001 0000 0000 0011 0000			SUB #48
0100 0100 0000 0000 0000 1111			STO MAX
0010 0001 0000 0000 0000 0000			LDM #0
0100 0100 0000 0000 0001 0000			STO TOTAL
0100 0100 0000 0000 0001 0001			STO COUNT
0001 0000		STRTLP:	IN
0110 0001 0000 0000 0011 0000			SUB #48
0100 0101 0000 0000 0001 0000			ADD TOTAL
0100 0100 0000 0000 0001 0000			STO TOTAL
0010 0101 0000 0000 0001 0001			LDD COUNT
0101 0101			INC ACC
1000 0100 0000 0000 0000 1111			CMP MAX
1010 0100 0000 0000 0000 0110			JPN STRTLP
1111 1111			END
0000 0000		MAX:	
0000 0000		TOTAL:	
0000 0000		COUNT:	

Table 6.05 Machine code created from assembly code

Some points to note are as follows.

- Most of the instructions have an operand which is a 16-bit binary number.
- Usually this represents an address but for the `SUB` and `LDM` instructions the operand is used as a value.
- There is no operand for the `IN` and `END` instructions.
- The `INC` instruction is a special case. There is an operand in the assembly language code but this just identifies a register. In the machine code the register is identified within the opcode so no operand is needed.
- The machine code has been coded with the first instruction occupying address zero.
- This code is not executable in this form but it is valid output from the assembler.
- Changes will be needed for the addresses when the program is loaded into memory ready for it to be executed.
- Three memory locations following the program code have been allocated a value zero to ensure that they are available for use by the program when it is executed.

6.05 Addressing modes

When an instruction requires a value to be loaded into a register there are different ways of identifying the value. Each one is known as an **addressing mode**. 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 6.06.

Addressing mode	Use of the operand
Immediate	The operand is the value to be used in the instruction; `SUB #48` is an example.
Direct	The operand is the address which holds the value to be used in the instruction; `ADD TOTAL` is an example.
Indirect	The operand is an address that holds the address which has the value to be used in the instruction.
Indexed	The operand is an address to which must be added the value currently in the index register (IX) to get the address which holds the value to be used in the instruction.

Table 6.06 Addressing modes

For immediate addressing there are three options for defining the value:

- #48 specifies the denary value 48
- #B00110000 specifies the binary equivalent
- #&30 specifies the hexadecimal equivalent

6.06 Assembly language instructions

We continue to consider a simple processor with a limited instruction set. The examples described here do not correspond directly to those found in the assembly language for any specific processor. Individual instructions will have a match in more than one real-life set. The important point is that these examples are representative. In particular, there are examples of the most common categories of instruction.

Data movement

These types of instruction can involve loading data into a register or storing data in memory. Table 6.07 contains a few examples of the format of the instructions with explanations.

Instruction opcode	Instruction operand	Explanation
`LDM`	`#n`	Immediate addressing. Load the number n to ACC.
`LDR`	`#n`	Immediate addressing. Load the number n to IX.
`LDD`	`<address>`	Direct addressing. Load the contents 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.
`MOV`	`<register>`	Move the contents of the accumulator to the given register (IX).
`STO`	`<address>`	Store the contents of ACC at the given address.

Table 6.07 Some instruction formats for data movement

The important point to note is that the mnemonic defines the instruction type including which register is involved and, where appropriate, the addressing mode. It is important to read the mnemonic carefully! The instruction will have an actual address where `<address>` is shown, a register abbreviation where `<register>` is shown and a denary value for `n` where `#n` is shown. The explanations use ACC to indicate the accumulator. For explanations of `LDD`, `LDI` and `LDX`, refer back to Table 6.07.

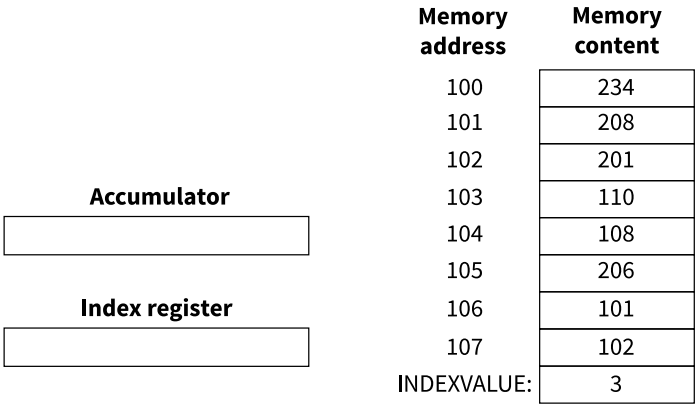


Figure 6.02 Example of some data stored in memory

The following shows some examples of the effect of an instruction or a sequence of instructions based on the memory content shown in Figure 6.02.

LDD 103	the value 110 is loaded into the accumulator
LDI 106	the value 208 from address 101 is loaded into the accumulator
STO 106	the value 208 is stored in address 106

LDD INDEXVALUE	the value 3 is loaded into the accumulator
MOV IX	the value 3 from the accumulator is loaded into the index register
LDX 102	the value 206 from address 105 is loaded into the accumulator

Input and output

There are two instructions provided for input or output. In each case the instruction has only an opcode; there is no operand.

- The instruction with opcode `IN` is used to store in the ACC the ASCII value of a character typed at the keyboard.
- The instruction with opcode `OUT` is used to display on the screen the character for which the ASCII code is stored in the ACC.

Comparisons and jumps

A program might need an unconditional jump or might need a jump if a condition is met. In the second case, a compare instruction is executed first. Table 6.08 shows the format for these types of instruction.

Instruction opcode	Instruction operand	Explanation
`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 the number n
`CMI`	`<address>`	Indirect addressing. The address to be used is at the given address. Compare the contents of ACC with the contents of this second address
`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

Table 6.08 Jump and compare instruction formats

Note that the comparison is restricted to asking if two values are equal.

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. This jump instruction does not cause an immediate jump. This is because a new value has to be supplied to the program counter so that the next instruction is fetched from this newly specified address. The incrementing of the program counter that took place automatically when the instruction was fetched is overwritten.

Arithmetic operations

There are no instructions for general-purpose multiplication or division. General-purpose addition and subtraction are catered for. Table 6.09 contains the instruction formats used for arithmetic operations.

Instruction opcode	Instruction operand	Explanation
`ADD`	`<address>`	Add the contents of the given address to the ACC
`ADD`	`#n`	Add the denary number n to the ACC
`SUB`	`<address>`	Subtract the contents of the given address from the ACC
`SUB`	`#n`	Subtract the denary number n 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)

Table 6.09 Instruction formats for arithmetic operations

Figure 6.03 shows a program to find out how many times 5 divides into 75.

The following should be noted concerning the program.

- The first three instructions initialise the count and the sum.
- The instruction in address 103 is the one that is returned to in each iteration of the loop; in the first iteration it is loading the value 0 into the accumulator when this value is already stored but this cannot be avoided.

Memory address	Memory content
100	LDD 200
101	STO 202
102	STO 203
103	LDD 202
104	INC ACC
105	STO 202
106	LDD 203
107	ADD 201
108	STO 203
109	CMP 204
110	JPN 103
111	LDD 202
112	OUT
113	END
200	0
201	5
202	
203	
204	75

Figure 6.03 A program to calculate the result of dividing 75 by 5

- The next three instructions are increasing the count by 1 and storing the new value.
- Instructions 106 to 108 add 5 to the sum.
- Instructions 109 and 110 check to see if the sum has reached 75 and if it has not the program begins the next iteration of the loop.
- Instructions 111 to 113 are only used when the sum has reached 75 which causes the value 15 stored for the count to be output.

Shift operations

There are two shift instructions available:

- `LSL  #n`

  where the bits in the accumulator are shifted logically n places to the left

- `LSR  #n`

  where the bits are shifted to the right.

In a **logical shift** no consideration is given as to what the binary code in the accumulator represents. Because a shift operation moves a bit from the accumulator into the carry bit in the status register this can be used to examine individual bits. For a left logical shift, the most significant bit is moved to the carry bit, the remaining bits are shifted left and a zero is entered for the least significant bit. For a right logical shift, it is the least significant bit that is moved to the carry bit and a zero is entered for the most significant bit.

If the accumulator content represents an unsigned integer, the left shift operation is a fast way to multiply by two. However, this only gives a correct result if the most significant bit is a zero. For an unsigned integer the right shift represents integer division by two. For example, consider:

00110001 (denary 49) gives if right shifted 00011000 (denary 24)

The remainder from the division can be found in the carry bit. Again, the division will not always give a correct result; continuing right shifts will eventually produce a zero for every bit. It should be apparent that a logical shift cannot be used for multiplication or division by two when a signed integer is stored. This is because the operation may produce a result where the sign of the number has changed.

As indicated earlier, only the two logical shifts are available for the simple processor considered here. However, in more complex processors there is likely to be a **cyclic shift** capability. Here a bit moves off one end into the carry bit then one step later moves in at the other end. All bit values in the original code are retained. Left and right **arithmetic shifts** are also likely to be available. These work in a similar way to logical shifts, but are provided for the multiplication or division of a signed integer by two. The sign bit is always retained following the shift.

Bitwise logic operation

The options for this are described in Table 6.10.

Instruction opcode	Instruction operand	Explanation
`AND`	`#Bn`	Bitwise AND operation of the contents of ACC with the binary number n
`AND`	`<address>`	Bitwise AND operation of the contents of ACC with the contents of <address>
`XOR`	`#Bn`	Bitwise XOR operation of the contents of ACC with the binary number n
`XOR`	`<address>`	Bitwise XOR operation of the contents of ACC with the contents of <address>
`OR`	`#Bn`	Bitwise OR operation of the contents of ACC with the binary number n
`OR`	`<address>`	Bitwise OR operation of the contents of ACC with the contents of <address>

Table 6.10 Bitwise logical operation instructions

The operand for a bitwise logic operation instruction is referred to as a mask because it can effectively cover some of the bits and only affect specific bits. Some examples of their use are given in Chapter 7 (Section 7.03).

6.07 Further consideration of assembly language instructions

Register transfer notation

Section 6.01 introduced an extension to register transfer notation. We can use this to describe the execution of an instruction. For example, the `LDD` instruction is described by:

```
ACC ← [[CIR(15:0)]]
```

The instruction is in the CIR and only the 16-bit address needs to be examined to identify the location of the data in memory. The contents of that location are transferred into the accumulator.

TASK 6.02

Use register transfer notation to describe the execution of an LDI instruction.

Computer arithmetic

In Chapter 1 (Section 1.03) we saw that computer arithmetic could lead to an incorrect answer if overflow occurred. In Chapter 5 (Section 5.02) we saw the possible uses of the Status Register. The following worked example illustrates how the values stored in the Status Register can identify a specific overflow condition.

The use of the following three flags is required:

- the carry flag, identified as C, which is set to 1 if there is a carry
- the negative flag, identified as N, which is set to 1 if a result is negative
- the overflow flag, identified as V, which is set to 1 if overflow is detected.

WORKED EXAMPLE 6.01

Using the status register during an arithmetic operation

1 Consider the addition of two positive values where the sum of the two produces an answer that is too large to be correctly identified with the limited number of bits used to represent the values. For example, Figure 6.04 shows what happens if we use an eight-bit binary integer representation and attempt to add denary 66 to denary 68.

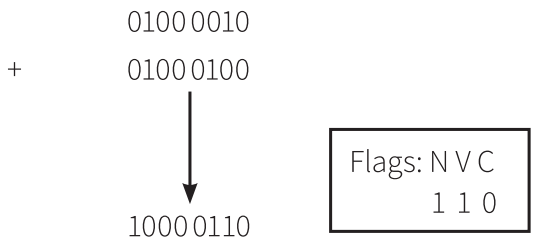


Figure 6.04 An attempted addition of denary 66 to denary 68

The answer produced is denary −122. Two positive numbers have been added to get a negative number. This impossibility is detected by the combination of the negative flag and the overflow flag being set to 1. The processor examines the flags, identifies the problem and generates an interrupt.

2 Consider using the same eight-bit binary integer representation but this time we add two negative numbers (−66 and −68 in denary). The result is shown in Figure 6.05.

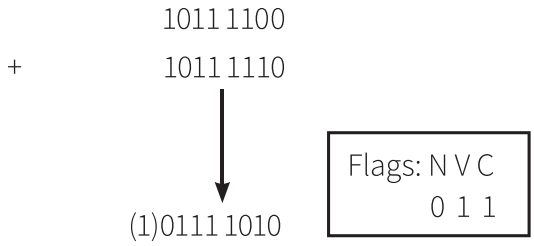


Figure 6.05 An attempted addition of denary −66 to denary −68

We get the answer +122. This impossibility is detected by the combination of the negative flag not being set and both the overflow and the carry flag being set to 1.

Extension Question 6.01

Carry out a comparable calculation for the addition in binary of −66 to +68. What do you think the processor should do with the carry bit?

Tracing an assembly language program

One way of checking to see if an assembly language program has errors is to carry out a dry (practice) run. The main feature of this will be to check how the contents of the accumulator change as the program runs. The following two worked examples illustrate the process.

WORKED EXAMPLE 6.02

Tracing an assembly language program

For this example the trace table needs a column for the accumulator, two for memory locations and one for the output.

The tracing is based on an initial user input of 15, a second input of 27 and a final input of 31.

The program is shown in Figure 6.06.

100	IN
101	STO 200
102	IN
103	STO 201
104	IN
105	ADD 200
106	STO 200
107	ADD 201
108	INC ACC
109	OUT
110	END

Figure 6.06 The assembly language program

The completed trace table is shown in Figure 6.07

Accumulator	Memory location 200	Memory location 201	Output
15			
	15		
27			
		27	
31			
46			
	46		

73			
74			
			74

Figure 6.07 The trace table showing the execution of the program

Note that in this presentation the decision has been made to use a new row in the trace table for each instruction in the program. This helps with checking. However, an alternative correct method is to enter a value in a column in the first available position. For example in the Memory location 200 column the first two rows could contain the 15 and 46. The other point to note is that if an instruction does not change an entry in a column it is not necessary to enter the value stored again. The trace table only needs to show activity; it does not have to record a complete set of values at each stage in the program execution.

WORKED EXAMPLE 6.03

Tracing an assembly language program

Some instructions for part of a program are contained in memory locations 100 upwards. Some 4-bit binary data values are stored in locations 200 upwards. For illustrative purposes the instructions are shown in assembly language form. At the start of a part of the program, the memory contents are as shown in Figure 6.08.

Address	Contents
100	LDD 200
101	INC ACC
102	ADD 201
103	CMP 202
104	JPE 106
105	DEC ACC
106	INC ACC
107	STO 203

Address	Contents
200	0001
201	0011
202	0101
203	

Figure 6.08 The contents of memory addresses before execution of the program begins

The completed trace table for this example is shown in Figure 6.09. Because the program contains a jump instruction it is necessary to record the values for the program counter as well as for the accumulator.

Program counter PC	Accumulator ACC	Memory location 203
100	1000	
101	0001	
102	0010	
103	0101	
104		
106		
107	0110	
108		0110

Figure 6.09 The contents of the program counter and accumulator during program execution

The entries in the table can be explained as follows.

- The first row shows the stored value before execution of this part of the program. There will be a value in the accumulator resulting from an earlier instruction.

- The second row shows the result of the execution of the instruction in location 100 which loads a value into ACC; this is followed by the PC being automatically incremented.
- The next two rows show the value being changed in the ACC by the instructions in 101 and 102 and the automatic incrementing of the PC each time.
- The fifth row has no new value in ACC because only a comparison is being done but there is an automatic increment of the PC.
- The sixth row shows a new value in the PC which has resulted from the execution of the jump instruction which tested for equality and found it to be True.
- The seventh row shows the result of the instruction in location 106 which has incremented the ACC.
- The final row shows the value stored in location 203.

Question 6.01

Can you follow through the changes in the trace table for Worked Example 6.03? Could it be possible for the program to change the content in one of the memory locations 100–107 during execution?

TASK 6.03

Without looking at the explanations provided, trace the assembly language program shown in Table 6.01. Use a value of 3 for MAX and then 7, 8 and 9 as input values.

Reflection Point:

There are several references in this chapter to the content in earlier chapters. Have you checked that you understand how the topics are related by revising the content in the earlier chapters?

Summary

- A machine code instruction consists of an opcode and an operand.
- An assembly language program contains assembly language instructions plus directives that provide information to the assembler.
- A two-pass assembler identifies relative addresses for symbolic addresses in the first pass.
- Processor addressing modes can be: immediate, direct, indirect or indexed.
- Assembly language instructions can be categorised as: data movement, input/output, compare and jump, arithmetic, shift and logical.

Exam-style Questions

1 Three instructions for a processor with an accumulator as the single general purpose register are:

`LDD <address>` for direct addressing

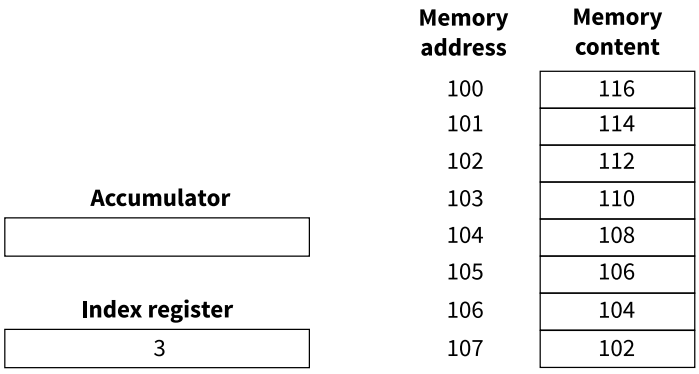
`LDI <address>` for indirect addressing

`LDX <address>` for indexed addressing

In the diagrams below, the instruction operands, the register content, memory addresses and the memory contents are all shown as denary values.

a Consider the instruction `LDD 103`.

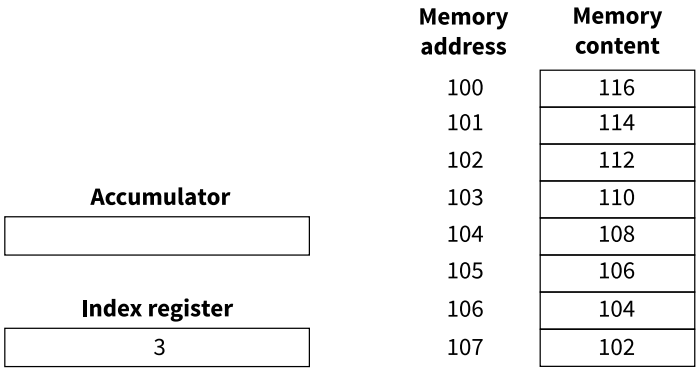
i Draw arrows on a copy of the diagram below to explain execution of the instruction. [2]



ii Give the contents of the accumulator as a denary value after execution of the instruction. [1]

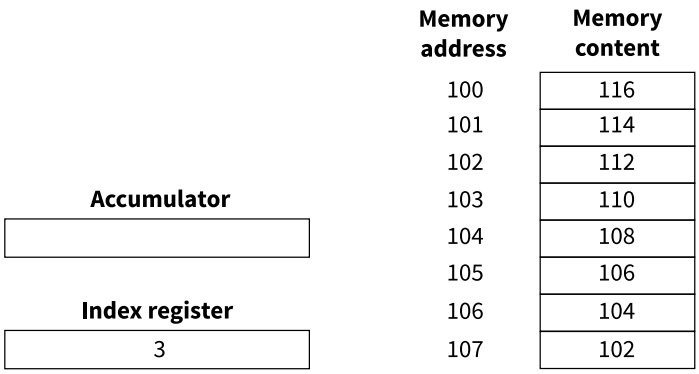
b Consider the instruction `LDI 107`.

i Draw arrows on a copy of the diagram below to explain execution of the instruction. [3]



ii Give the contents of the accumulator as a denary value after execution of the instruction. [1]

c **i** Draw arrows on a copy of the diagram below to explain the execution of the instruction `LDX 103`. [3]



ii Give the contents of the accumulator as a denary value after the execution. [1]

2 Every machine code instruction has an equivalent in assembly language. An assembly language

program will contain assembly language instructions. An assembly language program also contains components not directly transformed into machine code instructions when the program is assembled.

a Describe the use of three types of component of an assembly language program that are not intended to be directly transformed into machine code by the assembler. [6]

b Complete the trace table for the following assembly language program. Note that the LDI instruction uses indirect addressing. [6]

Assembly language program

Memory address	Memory content
100	LDD 201
101	INC ACC
102	STO 202
103	LDI 203
104	DEC ACC
105	STO 201
105	ADD 204
107	STO 201
108	END
201	10
202	0
203	204
204	5

Accumulator
0

Memory addresses			
201	**202**	**203**	**204**
10	0	204	5

3 Consider the following assembly language program:

```
<code>      IN
            STO CHARACTER
            IN
            SUB #48
START:      CMP #0
            JPN OUTPUT
            END
OUTPUT:     OUT
            DEC ACC
            JMP START
CHARACTER:
```

a Explain what the program takes as input. [4]

b Explain what the program outputs. [3]

c Complete the symbol table shown below which would be obtained from the first pass of a two-pass assembler. You can use denary numbers for addresses and you can assume that the first instruction is stored in address 0.

Label	Address

[4]

4 The table shows assembly language instructions for a processor which has one general purpose register, the Accumulator (ACC) and an index register (IX).

Instruction		**Explanation**
Op code	**Operand**	
LDD	<address>	Direct addressing. Load the contents of the given 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.
STO	<address>	Store contents of ACC at the given address.
ADD	<address>	Add the contents of the given address to 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).
CMP	<address>	Compare contents of ACC with contents of <address>.
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.
JMP	<address>	Jump to the given address.
OUT		Output to screen the character whose ASCII value is stored in ACC.
END		Return control to the operating system.

a The diagram shows the current contents of a section of main memory and the index register:

60	0011 0010
61	0101 1101
62	0000 0100
63	1111 1001
64	0101 0101
65	1101 1111
66	0000 1101
67	0100 1101
68	0100 0101
69	0100 0011
. . .	
1000	0110 1001

Index register:

0	0	0	0	1	0	0	0

i Show the contents of the Accumulator after the execution of the instruction:

LDX 60

Accumulator:

Show how you obtained your answer. [2]

ii Show the contents of the index register after the execution of the instruction:

DEC IX

Index register: | | | | | | | | | [1]

Cambridge International AS & A level Computer Science 9608 paper 11 Q9a June 2016