

**BASIC MICROPROCESSOR APPLICATION
SYSTEM MA02/1**

EXPERIMENT MANUAL

MP126/A

CONTENTS

INTRODUCTION

SECTION 1

COMPUTER ARITHMETIC AND NUMBER REPRESENTATION	Pages
Chapter 1.1 Introduction to Digital Technology.	1 - 9
Chapter 1.2 Binary Arithmetic.	9 - 18
Chapter 1.3 Computer Arithmetic.	19 - 25
Chapter 1.4 ASCII Codes and Parity.	26 - 29
Chapter 1.5 Octal and Hexadecimal Number Systems.	30 - 35

SECTION 2

INTRODUCTION TO MICROCOMPUTERS

Chapter 2.1 Introduction to the Microprocessor.	1 - 9
Chapter 2.2 Microcomputer System Elements.	10 - 17
Chapter 2.3 Introduction to Programming.	18 - 22
Chapter 2.4 The Instruction Set.	23 - 29

SECTION 3

PROGRAMMING THE EMMA MICROCOMPUTER

Chapter 3.1 Introduction to Emma.	1 - 7
Chapter 3.2 Program Entry and Execution.	8 - 15
Chapter 3.3 Using 6502 Instructions.	16 - 25
Chapter 3.4 Writing a Program.	26 - 41
Chapter 3.5 Arithmetic Operations.	42 - 53
Chapter 3.6 Logical Operations.	54 - 58
Chapter 3.7 Sub-routines.	59 - 62
Chapter 3.8 Stack Processing.	63 - 65
Chapter 3.9 Software Delays.	66 - 73
Chapter 3.10 Interrupts.	74 - 87
Chapter 3.11 Using the VIA.	88 - 108
Chapter 3.12 Program Debugging.	109 - 112
Chapter 3.13 Using the Cassette Interface.	113 - 114

SECTION 4

Pages

APPLICATIONS HARDWARE

Chapter 4.1	Using the Application Hardware.	1 - 15
Chapter 4.2	Further Application Programs.	16 - 18

SECTION 5

APPENDICES

1	Conversion Tables Standard Coding Sheet	1 - 3
2	Application Modules	1 - 11
3	Emma Monitor Sub-Routines.	1 - 9
4	Emma Memory Map.	1 - 4
5	Oscilloscopes.	1 - 6
6	Instruction Set.	1 - 19
7	Microcomputer Glossary.	1 - 26
8	Solutions to Questions.	1 - 8

SECTION 1

COMPUTER ARITHMETIC AND NUMBER REPRESENTATION

Chapter 1.1	Introduction to Digital Technology.	Page 1
Chapter 1.2	Binary Arithmetic	Page 9
Chapter 1.3	Computer Arithmetic	Page 19
Chapter 1.4	ASCII Codes and Parity	Page 26
Chapter 1.5	Octal and Hexadecimal Number Systems	Page 30

INTRODUCTION

The Microprocessor Application Manual provides a course of study based around the L.J. ELECTRONICS Basic Microprocessor Application System MA02 which comprises the following items of equipment:

MS1	Emma Microcomputer.
MS3	A/D Converter.
MS5	Switch Pad.
MS6	Buffered Loudspeaker.
MS7	Strain Gauge.
MS12	I/O Port Monitor.
PS4	System Power 90.
HM203	Dual Trace Oscilloscope.
H235	x1 Voltage Probe Kit (2 off).
CS1	Set of 36 x 4mm Leads.
CS2	Emma Cassette Interface - DIN Plug Lead.

Also required but not part of the set.

CR1	Cassette Recorder - with AUX DIN socket: tape counter and 2 x C12 cassettes.
-----	------------------------------------------------------------------------------

This manual considers the EMMA Microcomputer and selected Application Hardware Modules to provide an understanding of the basic principles and practice of microprocessors: especially with "Control" applications in mind. It is primarily intended to introduce EMMA (The Microcomputer) which forms the central element of the L.J. ELECTRONICS Microprocessor Teaching System.

A basic understanding is gained through graded examples and exercises using low-level programming techniques - both machine code and assembly language. To explore the considerable capabilities of the microprocessor more fully: a selection of peripheral items are introduced: and used in conjunction with EMMA: to form an integrated working control system.

The oscilloscope: while not necessarily forming an essential part of a particular application: is used to permit the examination of waveforms and so serves as an early introduction to system fault diagnosis. It also enhances the understanding of system operation.

Programs of even the most ingenious and meticulous programmers do not always function the first time. To cater for such possibilities the program debug feature of the EMMA Microcomputer is introduced.

To expedite the saving of programs: the use of a Cassette recorder is considered. Program "dumping" is an important aspect and this facility utilises the EMMA on-board cassette interface.

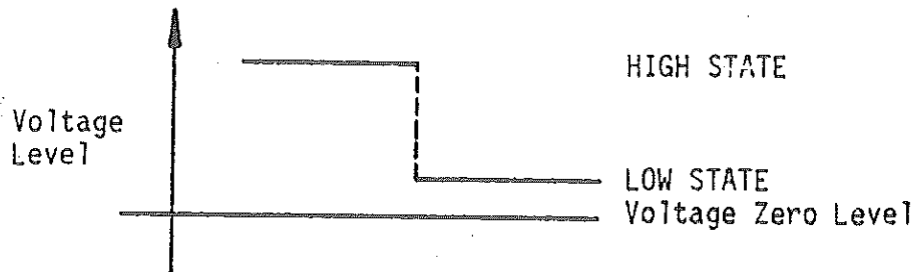
Numerous examples are used and exercises set to facilitate the learning process. To obtain maximum benefit it is strongly suggested that these be systematically and methodically worked through. Solutions to the exercises are published in a separate manual: reference number MP101. However a set of self-test questions are provided at the end of each Chapter with solutions given in Appendix 9.

Having completed the course of study detailed in this manual L.J. ELECTRONICS provide a wide range of equipment for more advanced studies: these include:

- More complex Application Hardware Modules.
- VISA expansion unit - providing ASSEMBLY and BASIC language programming.
- Memory Expansion/EPROM Programming module.
- Floppy Disc System.
- Test and Fault Diagnosis equipment.
- Robotics equipment.

LOGIC STATES

The inputs and outputs of digital devices are represented by two discrete voltage levels or states which are often referred to as either "high" or "low" with respect to each other.



We also refer to the high state as a "logical one" (1) and the low state as a "logical zero" (0). Since these devices are used to implement logical functions, we use the term "Binary Logic" to describe them. This is the technology of information processing.

ELECTRONIC REPRESENTATION OF INFORMATION

Information (data) is encoded using a binary format for processing purposes, the smallest unit being the binary digit or **Bit**. A number of bits are grouped together and the resulting sequence of 1's and 0's made to represent a unique piece of digitally or binary encoded data. A typical bit sequence would be

1010 1110

8 bits are used to encode the data item.

For machine processing the number of bits grouped together is usually fixed and such a group is termed a **word**. A **word** can therefore represent any fixed length of bits but it is normal to find standard word lengths of 8, 16 and 32 bits, especially in computing systems. We also define a word length of 8-bits as a **byte** and a **nibble** as being a group of 4-bits. The term byte is used extensively in the field of micro-processing.

MICROCOMPUTER DATA REPRESENTATION

The way in which data is represented in computer systems depends upon who or what is going to use the encoded information. We have already seen that "machines" require data presented in a binary format, however to the user of such machines this presentation is by no means convenient. We therefore encode data according to whether it will be used internally (within the machine) or externally (by the machine programmer). We will deal with these two aspects separately:

MICROCOMPUTER INTERNAL DATA REPRESENTATION

Computer systems operate to a strict sequential list of instructions called a **Program**. This is written by a Computer Programmer and designed to manipulate data. The program will be stored within the system memory and will perform operations upon data either stored within the system or obtainable as required from system peripherals*.

*See Glossary of Terms

Both the program, and the data upon which it operates, will need to be encoded into a binary format for use by the machine. We can, however, identify two distinct data groups which will require representation - Numeric Data and Alphanumeric Data. We will briefly outline these and then look at them in some detail.

Numeric Data is used to quantify things, for example, how many of these or those do we have. We then try to process this data by means of arithmetic or logical operations.

Computer systems excel at this type of processing; they are able to perform data processing operations both quickly and accurately. However, numbers must be represented on computer based systems in such a way that the arithmetic can be performed both easily and accurately.

We will examine the way in which numbers that require arithmetic operations are represented and how the arithmetic is performed later.

Alphanumeric Data comprises characters such as all the letters of the alphabet (both upper and lower case), numeric symbols, 0 through to 9, together with additional special symbols, such as question marks, quotes and brackets. We can easily accommodate these within 7-bits which allow for 128 possible codes. As you may imagine, standard codes have been devised but by far the most common for use in microprocessors is termed ASCII (pronounced Ask-ee) Code. ASCII stands for "American Standard Code for Information Interchange".

We will examine ASCII Code in some detail later.

MICROCOMPUTER EXTERNAL DATA REPRESENTATION

Much of the data input to a computer system is generated by human beings, also much of its output is frequently used by them. We can identify three formats which are suitable for computer user presentation - Binary, Octal or Hexadecimal and Symbolic.

We will deal with these in detail later.

Now let us consider in detail the whole problem of data representation and its manipulation.

DATA REPRESENTATION AND MANIPULATION

Various representations are in use and different methods of manipulation are available to us. All have advantages and disadvantages. In general, the designer is faced with a compromise situation and hence no two designs are exactly the same. However, it is our intention, throughout the rest of this section to provide you, the reader, with a few tools which will help you better understand the technology of microprocessors and their application.

Direct Binary is used to encode integers or whole numbers.

Let's consider a sequence of 8-bits or a single byte. Each bit is numbered 0 through to 7.

$B_7 \ B_6 \ B_5 \ B_4 \ B_3 \ B_2 \ B_1 \ B_0$

In any number system the digit position carries a "weighting" and the binary system is no exception to this. For example, the eighth bit (bit 7) has a greater significance than the first bit (bit 0). We refer to these particular bits as the most significant bit (MSB) and least significant bit (LSB), respectively. The weighting represented by the bit position is evaluated by raising the number system base (2 for binary) to a power indicated by the bit position, e.g.:

BIT POSITION	BIT WEIGHTING	DENARY EQUIVALENT
0	2^0	1
1	2^1	2
2	2^2	4
3	2^3	8

If you evaluate the weighting of the remaining bit positions ($B_4 - B_7$) you should find them to be 16, 32, 64 and 128 respectively. We are now in a position to decode any direct binary number into its denary equivalent.

● EXAMPLE:

Decode the following single byte direct binary encoded number into its denary equivalent:

Binary Number 0110 1101

$\begin{array}{cccc} 1 & 1 & 1 & 1 \\ 4 & 3 & 2 & 1 \\ \hline 16 & 8 & 4 & 1 \end{array}$

We will solve by tabulation:

BIT POSITION	BINARY VALUE	DENARY WEIGHTING	DENARY VALUE
0 (LSB)	1	2^0	$1 \times 2^0 = 1$
1	0	2^1	$0 \times 2^1 = 0$
2	1	2^2	$1 \times 2^2 = 4$
3	1	2^3	$1 \times 2^3 = 8$
4	0	2^4	$0 \times 2^4 = 0$
5	1	2^5	$1 \times 2^5 = 32$
6	1	2^6	$1 \times 2^6 = 64$
7 (MSB)	0	2^7	$0 \times 2^7 = 0$
			<u>Total</u> <u>Denary</u> <u>109</u>

The denary equivalent of binary 0110 1101 is therefore 109.

● EXERCISE 1.1.1

Decode the following direct binary numbers into their denary equivalents:

- Q1. 0110 Q2. 1010 Q3. 1100
 Q4. 001 0011 Q5. 0011 0011 Q6. 1001 1111

● EXERCISE 1.2.2

Determine the maximum denary whole number that can be encoded by:

- Q1. A Nibble
 Q2. A Byte

We will now encode a denary whole number in direct binary format. An example is probably the best way of doing this.

● EXAMPLE:

Encode denary 124 into direct binary.

We proceed by dividing our denary number by 2 and then successively dividing resultant quotients until zero is obtained. For example:

DIVIDEND	DIVISOR	QUOTIENT	REMAINDER	BINARY DIGIT
124	2	62	0	0 (LSB)
62	2	31	0	0
31	2	15	1	1
15	2	7	1	1
7	2	3	1	1
3	2	1	1	1
1	2	0	1	1 (MSB)

You should notice that 7 bits were required and that the binary equivalent of 124 is 111 1100. Since machines operate on a fixed number of bits, denary 124 would be represented in a single byte as 0111 1100.

● EXERCISE 1.1.3

Encode the following denary whole numbers into their direct binary, single byte equivalent.

Q1. 72 Q2. 36 Q3. 116 Q4. 231

QUESTIONS

1. Define a bit.
2. Define a byte.
3. What is alphanumeric data?
4. What is the denary equivalent of bit 5 (B_5) in an eight binary word.
5. Convert the following direct binary numbers into their denary equivalents.
(a) 1101 (b) 0011 1001 (c) 1111 0000
6. Convert the following denary numbers into their direct binary equivalents.
(a) 38 (b) 120 (c) 201

OBJECTIVES OF THE CHAPTER

Having studied this chapter you should:

- Be able to perform simple binary addition and subtraction.
- Understand the concept of signed binary numbers.
- Be able to calculate the two's complement of a negative binary number.
- Be able to perform binary subtraction by two's complement addition.

BINARY ARITHMETIC

We will now address ourselves to performing simple computations on direct binary numbers.

Binary Arithmetic is carried out according to simple rules.

The rules for addition are:

0 plus 0 = 0] - ADDITION
0 plus 1 = 1	
1 plus 0 = 1	
1 plus 1 = 0 and CARRY 1	

● EXAMPLE:

Using these rules we will add together two 4-bit direct binary coded numbers:

```
      1001
Plus  0101
      1110 RESULT
          1 CARRIES
```

You may like to check this by using denary equivalents.

● FURTHER EXAMPLE:

Perform an addition on the following 8-bit direct binary coded numbers:

0100 1111 and 0101 1100

now:

```
      0100 1111
Plus  0101 1100
      1010 1011 RESULT
          1 11 1 CARRIES
```

Again check using denary equivalents. You will notice the method is exactly the same as when using denary numbers.

● EXERCISE 1.2.1

Compute the SUM of the following pairs of direct binary numbers:

- Q1. 0110 and 1001
- Q2. 0101 and 1001
- Q3. 0100 1001 and 1000 1000
- Q4. 0011 1000 and 0110 1111

As with addition, so with subtraction; the rules are:

0 minus 0 = 0	SUBTRACTION
1 minus 0 = 1	
1 minus 1 = 0	
0 minus 1 = 1 and BORROW 1	

The Borrow is taken from adjacent more significant column.

● EXAMPLE:

Subtract 0101 from 1001

1001	
Minus 0101	
0100	RESULT
1	BORROWS

Again you should check using denary equivalents.

● EXERCISE 1.2.2

Convert the following denary numbers to direct binary and then perform the subtraction:

Q1. (6 - 3) Q2. (12 - 5) Q3. (24 - 18)

Signed Binary allows us to represent both positive and negative numbers, that is numbers which have both **magnitude** and **sign**. You may already have noticed that the numbers we have been operating upon are both small (a maximum of 255 for denary numbers encoded in 8 bits of direct binary) and positively signed.

Let us consider the problem of "signing" our binary numbers. Clearly we have to encode our "sign" (+ or -) using the binary symbols 0 and 1 since these are the only ones available to us. Conventionally, we assign the

left-most bit to providing the "sign" and the rest of the bits to the "magnitude" of our encoded number. Thus in **signed binary** notation the format is:

sign MSB ----- LSB
 B₇ B₆ ---- B₀

If the sign bit is "0" then positive sign is denoted, and if the sign bit is "1" then negative sign is denoted. For example:

SIGNED BINARY	SIGN BIT	MAGNITUDE BITS	DENARY EQUIVALENT
0110 1001	0	110 1001	+ 105
1110 1001	1	110 1001	- 105

If you compute the values of signed binary 0111 1111 and 1111 1111 you will notice that our range of numbers is now from +127 through zero to -127 as against 0 through to 255 in direct binary.

● EXERCISE 1.2.3

Determine the denary representation of the following signed binary numbers:

- Q1. 1011 0000 Q2. 0011 1100
 Q3. 1110 0000 Q4. 0101 0011

● EXERCISE 1.2.4

Encode the following denary numbers in signed binary format:

- Q1. -63 Q2. +24
 Q3. +87 Q4. - 8

We sometimes use the term **sign-magnitude** for this representation and we will use it to perform simple addition on the denary numbers +6 and -4:

+6 is represented by 0000 0110
-4 is represented by 1000 0100
 1000 1010 RESULT

Decoded our Sum is -10, clearly this is incorrect and should be +2. We must look for another method of representation if computations of this nature are to take place.

The solution to our problem lies in a representation called two's complement.

Two's Complement provides correct results when performing binary arithmetic computations, but first let us consider an intermediate step - one's complement. In this representation, all POSITIVE whole numbers are represented in signed binary format, thus +4 is 0000 0100 as usual, however its complement (-4) is represented by 1111 1011 and not 1000 0100 as it is in signed binary.

We obtained the **one's complement** by changing all 1's to 0's and all the 0's to 1's for example:

+4 is represented by: 0000 0100
-4 is represented by: 1111 1011

You will notice that the left-most bit can still be reserved to indicate the sign (0 for positive and 1 for negative).

If we perform simply arithmetic processes of addition and subtraction using this method we will find that the method does not always work. However, this method is used in large computers together with suitable tests, and corrections are carried out if necessary. Two's complement has evolved from this representation - it does work and is used almost exclusively in microcomputer based systems.

In two's complement representation positive numbers are still represented as for signed binary, but negative numbers are encoded in two's complement notation. Let's consider the signed denary number +5. In two's complement +5 is represented as 0000 0101:

Its complement (-5) is obtained by first obtaining the one's complement and then adding 1 to it, for example:

+5 is	0000 0101	
one's complement is	1111 1010	
ADD '1' (to one's complement)	<u> 1</u>	
	1111 1011	RESULT

Where the RESULT is the two's complement of +5 or, we may say, the RESULT is -5 represented in two's complement notation.

To summarise:

+5 is represented by	0000 0101	
-5 is represented by	1111 1011	in two's complement notation

You should notice that the left-most bit is still 0 for positive and 1 for negative numbers.

We will now check our method and assure ourselves that it does indeed function correctly. To do this we will perform a number of arithmetic operations.

● Add +5 to +3:

+5	0000 0101	
(Add) +3	<u>0000 0011</u>	
	0000 1000	RESULT (+8)
	111	CARRIES

RESULT CORRECT

● Subtract +5 from +3:

We can do this in either of two ways:

a) Using rules of binary subtraction given on page 1.8

+3	0000 0011	
(Subtract) +5	<u>0000 0101</u>	
	1111 1110	RESULT
1	1111 1	BORROWS

The result is obviously negative since the bit (B_7) is set to 1. If we cannot easily recognise the magnitude (since it is in two's complement notation) as 2 we can perform a two's complement to find its positive equivalent.

For example:

	1111 1110	RESULT
	<u>0000 0001</u>	1's comp
Add 1	<u>1</u>	
	0000 0010	(+2)

The result 1111 1110 is thus -2 in two's complement notation.
The RESULT is CORRECT.

Now let's do the computation the second way. This time we will change the sign of the (+5) and ADD rather than subtract.

The +5 must now be two's complemented to give -5.

+5	0000 0101
1's complement	1111 1010
Add 1	<u>1</u>
2's complement	1111 1011

now for the computation:

+3	0000 0011	
(add) -5	<u>1111 1011</u>	
	1111 1110	RESULT
	11	CARRIES

The RESULT is the same as before.

YOU WILL SEE LATER THAT THIS IS AN IMPORTANT FINDING.

- Add -5 to +3:

Again we must represent the negative denary number (-5) in two's complement and perform an addition.

+5	0000 0101	
1's complement	1111 1010	
Add 1	<u> 1</u>	
2's complement	1111 1011	(-5)

now for the computation:

-5	1111 1011	
(add) +3	<u>0000 0011</u>	
	1111 1110	RESULT
	11	CARRIES

Clearly the RESULT is negative. Performing a two's complement on 1111 1110 to find the magnitude will indicate a value of -2.

The RESULT is CORRECT.

- Subtract -5 from +3

Again we will represent -5 in two's complement which (from previous example) is 1111 1011 and perform the subtraction:

+3	0000 0011	
(Subtract) -5	<u>1111 1011</u>	
	0000 1000	RESULT
1	1111	BORROWS

You will notice that to perform our computation we required an EXTERNAL BORROW; that is, we borrowed externally to our 8-bits. If we now ignore this borrow the RESULT (0000 1000 which is +8) is CORRECT.

Now let's perform the computation a different way. Instead of performing a subtraction we will perform a two's complement on (-5) and ADD.

-5 is	1111 1011	in two's complement notation
Complement	0000 0100	
Add 1	<u> 1</u>	
	0000 0101	2's complement of -5

Now ADD +3 (0000 0011) and the two's complement of -5 which is +5 (0000 0101):

+3	0000 0011	
(Add) +5	<u>0000 0101</u>	
	0000 1000	RESULT
	111	CARRIES

The RESULT is the same, i.e. +8 and is CORRECT.

● Add -5 to -3:

-5 is	1111 1011	
(Add) -3 is	<u>1111 1101</u>	
	1111 1000	RESULT
1	1111 1111	CARRIES

We have now generated an EXTERNAL CARRY. If we ignore this the result (1111 1000 which is -8) is CORRECT.

● Subtract -5 from -3:

Rather than subtract directly we will ADD the two's complement of -5 (+5) to -3.

-3 is	1111 1101	
(Add) +5 is	<u>0000 0101</u>	
	0000 0010	RESULT
	1111 1 1	CARRIES

Again we have generated an EXTERNAL CARRY and again if we ignore it the result is CORRECT.

You should now be convinced that representing negative numbers in two's complement does enable us to perform computations using the rules of binary arithmetic addition only and still get correct results in terms of both sign and magnitude.

Now let's have some practice!

● EXERCISE 1.2.5

Using normal binary arithmetic rules and two's complement to represent negative numbers, perform the following computations after having encoded each number into 8-bits.

- | | | |
|-----------------|------------------|------------------|
| Q1. $18 + 9$ | Q2. $6 + 2$ | Q3. $24 + 16$ |
| Q4. $23 - 18$ | Q5. $15 - 18$ | Q6. $-23 + 18$ |
| Q7. $21 - (+5)$ | Q8. $-16 - (-2)$ | Q9. $18 + (-24)$ |

In this chapter we have restricted ourselves to small numbers, in the next chapter we will investigate the consequences of using larger numbers for our computation.

QUESTIONS

1. Perform the following binary operations:

- (a) $1001 + 0101$
- (b) $0011 + 0101$
- (c) $1111 - 0101$
- (d) $1001 - 0111$

2. What is the range of values (in denary) for an 8-bit signed binary number.

3. What is the two's-complement of -38.

4. Perform the following subtraction using two's complement addition.
 $47 + (-22)$.

OBJECTIVES OF THIS CHAPTER

Having studied this chapter you should:

- Understand the use of CARRY AND OVERFLOW flags in microcomputers.
- Be able to determine whether or not these flags will be set as the result of a calculation.
- Understand the use of multiple precision arithmetic.

The numbers you have been using in previous exercises were deliberately chosen small. We must now ask ourselves what would happen if larger ones were used.

Let's try adding together +127 and say +6

```
+127  0111 1111
+   6  0000 0110
-----
      1000 0101  RESULT
```

The answer appears to be (-5). Clearly this is INCORRECT.

Have we discovered a flaw in two's complement notation arithmetic?

Not really. The problem lies in the fact that although the magnitude of the individual numbers could be encoded in seven bits the result of the computation required eight bits. The result was a **CARRY** between bit 6 and bit 7 (sign bit) which corrupted both the magnitude and the sign. You may be interested to treat all 8-bits as magnitude and decode as such. The result would then be correct (+133).

Technically an **OVERFLOW** has occurred due to the internal carry.
Problems of this nature must be taken care of in any computing system.

Let us recapitulate:

- So far all computations have been performed using a total of 8-bits to encode both sign and magnitude.
- A tacit assumption has been made that a ninth bit has been available for both external BORROWS and CARRIES, indeed we even ignored them when decoding our results.
- We have discovered that subtractions can be performed using COMPLEMENT and ADD methods. We therefore only need the ability to ADD and not both add and subtract

Carries and Overflows are an important concept and must be considered in some detail.

The majority of microprocessors use a fixed bit length of 8-bits (one byte). They also provide a separate bit for use of the external carry. We will adopt this format and perform a number of additions specifically to explore the concept of OVERFLOW.

Our computations are tabulated overleaf:

DENARY NUMBERS	BINARY NUMBERS			RESULT	BIT MOVEMENT		
	CARRY	SIGN	MAGNITUDE		C	S	B ₆
+ 3 + 5	0	0	000 0011	CORRECT	N O N E		
		0	000 0101				
		0	000 1000				
+127 + 1	0	0	111 1111	ERROR	• ← •		
		0	000 0001				
		1	000 0000				
		1	111 111				
+ 8 - 2	1	0	000 1000	CORRECT	• ← •		
		1	111 1110				
		0	000 0110				
		1	111				
+ 2 - 8	0	0	000 0010	CORRECT	N O N E		
		1	111 1000				
		1	111 1010				
- 2 - 8	1	1	111 1110	CORRECT	• ← •		
		1	111 1000				
		1	111 0110				
		1	111				
-128 - 1	1	1	000 0000	ERROR	• ← •		
		1	111 1111				
		0	111 1111				

Note

- Arithmetic Notation - Two's Complement
- Abbreviations: C = Carry Bit
S = Sign Bit
B₆ = Most significant Bit of Number
• ← • = Bit Movement

If we examine the table we will notice that ERRORS occurred when results exceeded +127 and -128. Obviously such results require more than our allotted 8-bits. OVERFLOW is said to have occurred.

If we now examine the bit movement, we will find that an OVERFLOW occurs only when there is a bit movement from

- B₆ → Sign Bit
OR
- Sign Bit → Carry Bit.

If both bit movements occur for the same computation OVERFLOW has not happened.

We have already stated that computation in microprocessors is performed in 8-bit registers with a 1-bit device for the external carry. An additional bit may also be provided to automatically indicate an overflow condition. We term these separate bits, STATUS FLAGS.

● EXERCISE 1.3.1

Let us assume we have 8-bit registers for computation purposes and two status flags, carry (c) and overflow (v). These flags will be set (contents made equal to logical '1') when carries or overflow occur. Perform the following computations and hence determine the status of the two flags and state whether overflow has occurred.

Q.1 1101 1001

Add 0111 1111

Q.2 0110 1010

Subtract 1010 1111

Q.3 0100 1111

Add 1100 0001

Q.4 1001 0000

Subtract 1101 0000

All our computations have been performed in a fixed number of bits. It is easier for a machine to perform arithmetic calculations if this is so rather than have a variable bit length. If the number of bits are fixed we refer to this as being FIXED FORMAT.

Fixed format does present problems in as far as the range of numbers that can be accommodated (without special provision) is also fixed. Using Two's Complement representation for negative numbers then the range for an 8-bit register is +127 and -128. Clearly this is insufficient for many applications. We will now address this problem.

Multiple Precision Arithmetic is a method which involves using a two or more byte format to accommodate the required magnitude. If two bytes were sufficient we would use a high byte and a low byte thereby effectively giving us a 16-bit DOUBLE-PRECISION format.

High byte Low byte
8-Bits 8-Bits

A number would be "stored" as:

NUMBER	FORMAT	
	High Byte	Low Byte
0	0000 0000	0000 0000
1	0000 0000	0000 0001
-1	1111 1111	1111 1111
32767	0111 1111	1111 1111

The sign bit is still the left most bit.

A computation would be performed as below:

NUMBER	HIGH BYTE	LOW BYTE	COMMENT
-1	1111 1111	1111 1111	
+32767	0111 1111	1111 1111	
+32766	0111 1111	1111 1110	RESULT
1	1111 1111	1111 111	CARRIES

Two other formats are also used. These are Binary Coded Decimal (BCD) and Floating point. However, we will not touch on these at this stage, only to say that BCD gives absolutely accurate results but uses a large amount of memory. It is normally used for accounting purposes. Floating point extends the capacity for handling large numbers beyond that of the fixed format. It also generally makes for ease of programming.

In general it would be true to say that these later two formats are more usefully employed in data processing rather than in control applications for which the microprocessor is admirably suitable.

In the next chapter we will investigate a means of encoding alphabetic as well as numeric characters into a digital format suitable for microcomputers.

QUESTIONS

1. When does an overflow occur.
2. Determine whether or not the CARRY and OVERFLOW flags will be set by the following calculations.

(a) $120 - 25$

(b) $83 + 52$

(c) $8 - 23$

3. Use double precision arithmetic to perform the following binary calculations.

(a) $32767 - 255$

OBJECTIVES OF THIS CHAPTER

Having studied this chapter you should:

- Be familiar with the use of ASCII codes to represent alphanumeric characters.
- Be able to deduce the ASCII code, for any particular character, from the table given.
- Understand the use of parity bits as an aid to error detection.

ASCII Codes we have already mentioned. They arise because we need to encode alphanumeric data or characters. 7-bits are sufficient for this purpose, but 8-bits will probably be available. This eighth bit is sometimes used as a **PARITY** bit and helps us to perform a check to see whether data has been corrupted during transmission especially when this takes place between a peripheral outstation and its centrally situated microcomputer. There are two methods of implementing a parity check. The parity bit (bit 7) may be set to make up an even sum of all bits, in which case it is referred to as **EVENS PARITY** or set to make up an odd sum to give **ODDS PARITY**.

● **EXAMPLE:**

1110 0111	EVENS PARITY
1110 0110	ODDS PARITY

The parity bit is the left most bit.

ASCII is a standard coding system and perhaps a table is the best way of expressing it.

ASCII CODES

					COLUMNS								
					0	1	2	3	4	5	6	7	
					B ₆	0	0	0	0	1	1	1	1
					B ₅	0	0	1	1	0	0	1	1
					B ₄	0	1	0	1	0	1	0	1
ROWS	0	0	0	0	0		SP	0		P		P	
	1	0	0	0	1		!	1	A	Q	a	q	
	2	0	0	1	0		"	2	B	R	b	r	
	3	0	0	1	1		£	3	C	S	c	s	
	4	0	1	0	0		\$	4	D	T	d	t	
	5	0	1	0	1		%	5	E	U	e	u	
	6	0	1	1	0		&	6	F	V	f	v	
	7	0	1	1	1		'	7	G	W	g	w	
	8	1	0	0	0		(8	H	X	h	x	
	9	1	0	0	1)	9	I	Y	i	y	
	10	1	0	1	0		*	:	J	Z	j	z	
	11	1	0	1	1		+	;	K		k		
	12	1	1	0	0		'		L		l		
	13	1	1	0	1		-	=	M		m	-	
	14	1	1	1	0		°		N		n		
	15	1	1	1	1		/	?	O	-	o	DEL	
					B ₃	B ₂	B ₁	B ₀					

If you now consider the table you should notice the following:

- The seven bit code (no parity) is represented by $B_0 - B_6$. Bit six being the most significant.
- The first four bits ($B_0 - B_3$) appear in the left-hand vertical column while the last three bits ($B_4 - B_7$) appear in the top three horizontal rows.
- The basic principle of the table is that it is conceived in binary form which makes it very suitable for processing information by computer.
- Direct binary codes 0000 0000 through to 0001 1111 (Denary 0 - 32) are devoted to control codes for data communication equipment. We have not specified these here.
- Bit zero through to bit three ($B_0 - B_3$) encode in direct binary the numerals 0 through to 9.

An example in finding an ASCII code for a given character is:

Find the ASCII code for the character, numeral 5.

Follow through the shading on the table. This will give us the ASCII code required. Reading both columns and rows will give:

COLUMN 3			ROW 5			
B_6	B_5	B_4	B_3	B_2	B_1	B_0
0	1	1	0	1	0	1

Therefore the ASCII Code for Numeral 5 is 011 0101.

If a **parity bit** is to be used then:

0011 0101 (EVEN)

1011 0101 (ODD)

● EXERCISE 1.4.1

Q1. Determine the ASCII codes for the following Alphanumeric Characters:

- | | | |
|------|------|------|
| a) 9 | b) 5 | c) Z |
| d) A | e) a | f) ? |

Q2. Assign the parity bit for each of the codes determined in Q1. for Even Parity and Odd Parity.

So far we have concerned ourselves with data representation within the machine. In the next chapter we will address ourselves to EXTERNAL representation.

QUESTIONS

1. What does ASCII stand for?

2. Determine the ASCII codes for:

(a) Q

(b) 3

(c) +

(d) v

3. Assign the odd parity bit for the codes of 2(a) and (b) and the even parity bit for 2(c) and (d).

OBJECTIVES OF THIS CHAPTER

Having studied this chapter you should:

- Be familiar with octal and hexadecimal number system.
- Be able to convert denary numbers to octal or hexadecimal representations and vice versa.
- Understand the use of the symbolic representation of data.

We have seen that data is stored within the machine using a sequence of bits (0's and 1's). These sequences can be output to devices termed light emitting diodes (LED's). These are illuminated to indicate a '1' and extinguished to indicate a '0'. There is some merit in having this representation available to the user for program debugging but is fairly time consuming and error prone. More convenient representations have been devised which we will consider next.

Octal and Hexadecimal are two systems fairly convenient to use.

They encode a number of bits into a unique code and are frequently used at the machine/human interface.

Three bits are encoded in the **OCTAL** system to form a symbol between 0 and 7.

The table below indicates:

BINARY	OCTAL
0000	
001	1
010	2
011	3
100	4
101	5
110	6
111	7

A sequence of 8-bits would be encoded as:

00 100 011

0 4 3

or 043_8 in Octal.

The subscript 8 denotes the base of the number system used.

The **Hexadecimal** system is now universally adopted for use in microprocessor systems since it uses only two symbols to encode 8-bits, that is. four bits per symbol. The symbols used in the hexadecimal representation are 0 - 9 and A - F, sixteen symbols in all permitting a number system to base 16.

An hexadecimal conversion table related to Decimal, Binary and Octal is given below:

DECIMAL	BINARY	HEX	OCTAL
0	0000	0	0
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
6	0110	6	6
7	0111	7	7
8	1000	8	10
9	1001	9	11
10	1010	A	12
11	1011	B	13
12	1100	C	14
13	1101	D	15
14	1110	E	16
15	1111	F	17

A sequence of 8-bits would be encoded as:

0100 1110

4 E

or $4E_{16}$ in hexadecimal.

Since the representation is universally accepted it is quite normal to omit the subscript 16. However, if confusion may result it is better to include it.

It is convenient to have a method of conversion rather than have to rely upon a "look-up" table. The following may help:

● Decimal to Octal

$85_{10} \Rightarrow \text{OCTAL}$

85 divided by 8 = 10 remainder 5

10 divided by 8 = 1 remainder 2

1 divided by 8 = 0 remainder 1

Thus 85_{10} (decimal) = 125_8 (Octal)

● Octal to Decimal

$137_8 \Rightarrow \text{DECIMAL}$

1 3 7

$$\begin{array}{rclcl} 7 \times 8^0 & = & 7 \times 1 & = & 7 \\ 3 \times 8^1 & = & 3 \times 8 & = & 24 \\ 1 \times 8^2 & = & 1 \times 64 & = & \underline{64} \\ & & & & \underline{95} \end{array}$$

Thus 137_8 (Octal) = 95 (decimal).

● Decimal to Hexadecimal

$90_{10} \Rightarrow \text{HEXADECIMAL}$

90 divided by 16 = 5 remainder A

5 divided by 16 = 0 remainder 5

Thus 90 (decimal) = 5A (Hexadecimal)

● Hexadecimal to Decimal

1AF → DECIMAL

1 A F

$$\begin{array}{rcl} F \times 16^0 & = & F \times 1 = 15 \\ A \times 16^1 & = & A \times 16 = 160 \\ 1 \times 16^2 & = & 1 \times 16^2 = \underline{256} \\ & & \underline{431} \end{array}$$

Thus $1AF_{16}$ (Hexadecimal) = 431_{10} (decimal)

● EXERCISE 1.5.1

Convert the following:

Q1. 125_{10} , 68_{10} , 25_{10} into:

- a) Straight Binary (8-bits)
- b) Octal
- c) Hexadecimal

Q2. -62_{10} , -94_{10} , -32_{10} , into:

- a) Two's Complement (8-bit)
- b) Octal
- c) Hexadecimal

Q3. 1BA, 39F, 21C, into:

- a) Decimal
- b) Binary

Symbolic is a representation which is extremely user friendly since data is presented in symbolic form. For example, alphabetic characters are presented as such rather than a sequence of binary digits displayed on LED's, or as coded in Octal or Hexadecimal.

A suitable means of display is required such as a Cathode Ray Tube (CRT) Monitor or Printer. On most single board micro-processors provision for such display methods may not be available in which case communication with the microcomputer will almost undoubtedly be in hexadecimal.

QUESTIONS

1. Convert the following denary numbers into octal:

(a) 22

(b) -17

2. Convert the following denary numbers into hexadecimal:

(a) 83

(b) -45

3. Convert the following hexadecimal numbers to denary:

(a) 13F

(b) 20C

SECTION 2

INTRODUCTION TO MICROCOMPUTERS

Chapter 2.1	Introduction to the Microprocessor	Page 1
Chapter 2.2	Microcomputer System Elements.	Page 10
Chapter 2.3	Introduction to Programming	Page 19
Chapter 2.4	The Instruction Set	Page 23

OBJECTIVES OF THIS CHAPTER

Having studied this chapter you should;

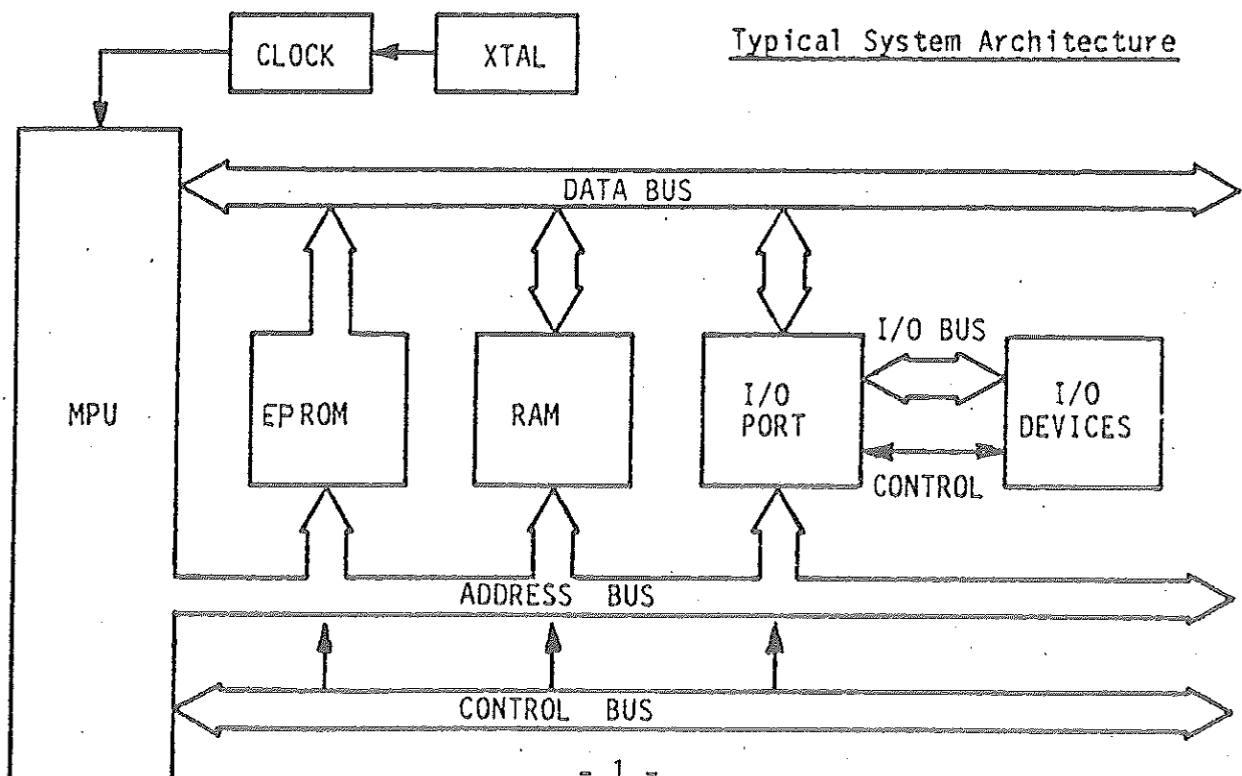
Understand how the microprocessor fits into the architecture of a microcomputer system.

Understand the function of the basic operational elements of a microprocessor.

Understand how the microprocessor communicates with other devices in the system.

SYSTEM ARCHITECTURE

We have already looked fairly deeply at the way data is represented both internally and externally to the microcomputer. In this chapter we are going to consider how the microprocessor fits into the architecture of a typical microprocessor system such as Emma.



With the exception of the I/O Devices each of the component parts shown above can be identified upon the **Emma** microcomputer board. We will first examine the microprocessor followed by the other system components in the next chapter.

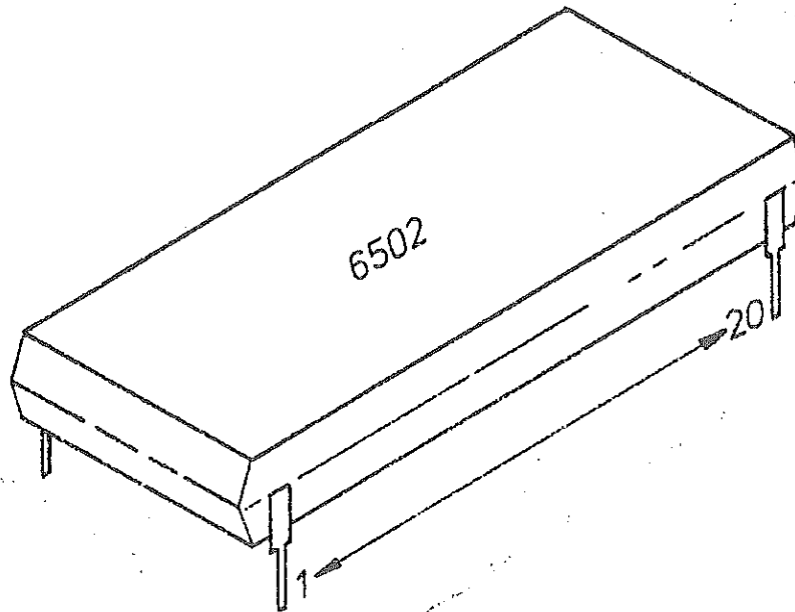
A TYPICAL MICROPROCESSOR

We are going to relate our typical microprocessor to that used in **Emma**. The various manufacturers of microprocessors configure their devices differently but the principles remain the same.

First we will pose the question "What is a microprocessor?" As a start it may be helpful if we could define what a **Process** is. One definition would be that a process is a series of operations which take place in a definite manner and a device capable of performing sequential operations in the manner defined would be referred to as a **Processor**. If the processor was physically very small as distinct to its complexity, we may well refer to it as a **Microprocessor**. In modern technological terms a microprocessor is all this. Additionally it is understood to be an **Electronic Device** contained within a small package known as an **Integrated Circuit (I.C.)**. Integrated circuits are not confined solely to microprocessors and many electronic circuits are packaged in this way.

These integrated circuits are then used as electronic circuit building blocks and when appropriately connected together form a larger electronic system. An example of a system configured in this way is the **Emma** microcomputer which you are using.

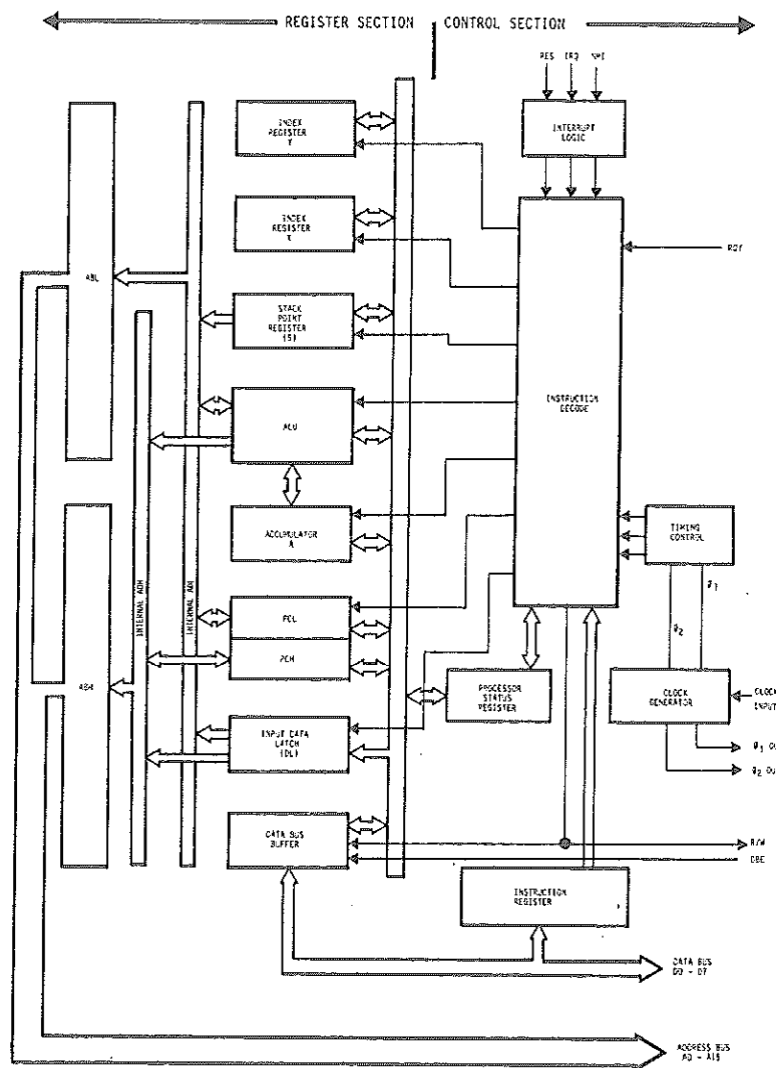
The appearance of a typical microprocessor is shown overleaf. You will easily be able to identify the one used in **Emma** by looking for the number 6502 marked on top of one of the larger IC's.



We also understand a microprocessor to be a device which performs arithmetic and logic operations on digitally encoded data. These operations are executed in sequence and as dictated by a **Program of Instructions** stored within **Memory**. The memory will form part of a complete system but will not necessarily be included in the microprocessor package.

The enormous power of the microprocessor lies in the speed at which it can execute these simple but humanly tedious operations and in the infinite variety of sequences which can be defined by the program of instructions.

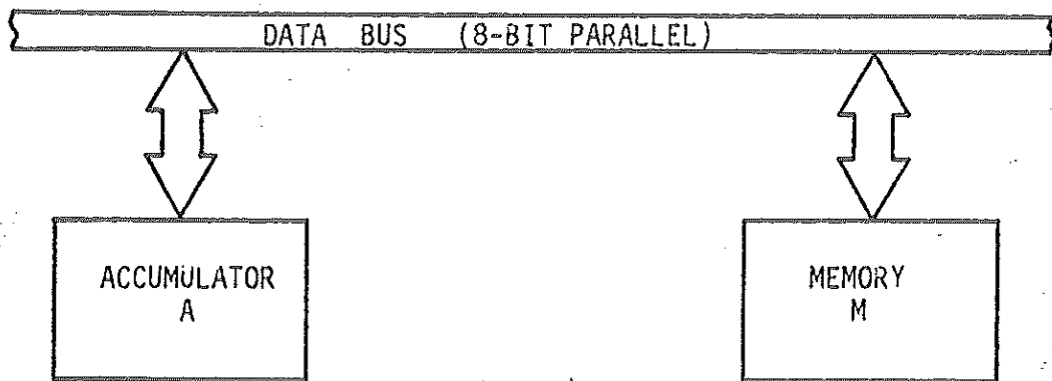
If we look at our system architecture: the microprocessor unit (MPU) appears as a single entity. Indeed it does comprise a single I.C. We can, however, identify, by function, a number of operational devices within the IC: although we cannot see them. It is these devices which we will now consider and a block diagram may help us to focus our thoughts.



INTERNAL ARCHITECTURE OF A MICROPROCESSOR The diagram is, in fact, that of the microprocessor used in Emma - the 6502, manufactured by Rockwell International.

The Accumulator (A) is a general purpose register which has two basic functions:

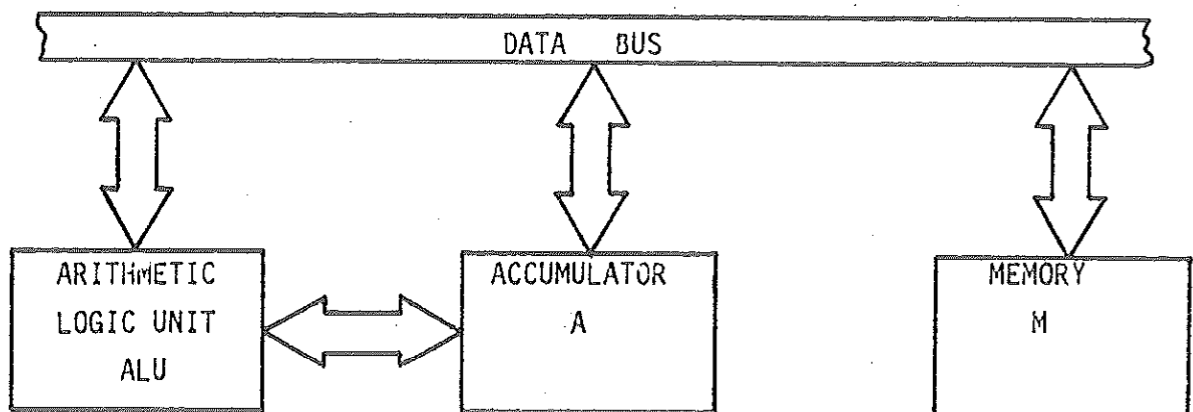
It acts as a primary storage element when data is moved from one memory location to another. Indeed all data transfers between memory locations will normally take place via the accumulator. These data transfers require a means to convey them and this is the function of the Data Bus. The diagram below indicates:



The accumulator also serves as a temporary storage element for intermediate results of arithmetic and logical operations. For example we may wish to perform an addition on two bytes of data. We can move the first byte into the accumulator, add to it the second byte, and leave the result in the accumulator.

To perform these functions the accumulator works closely with the Arithmetic Logic Unit (ALU).

The Arithmetic Logic Unit (ALU) performs simple arithmetic calculations and logical operations. This is the function of the ALU and, as can be seen from the diagram overleaf, data transfers can take place directly between the accumulator and the arithmetic logic unit.

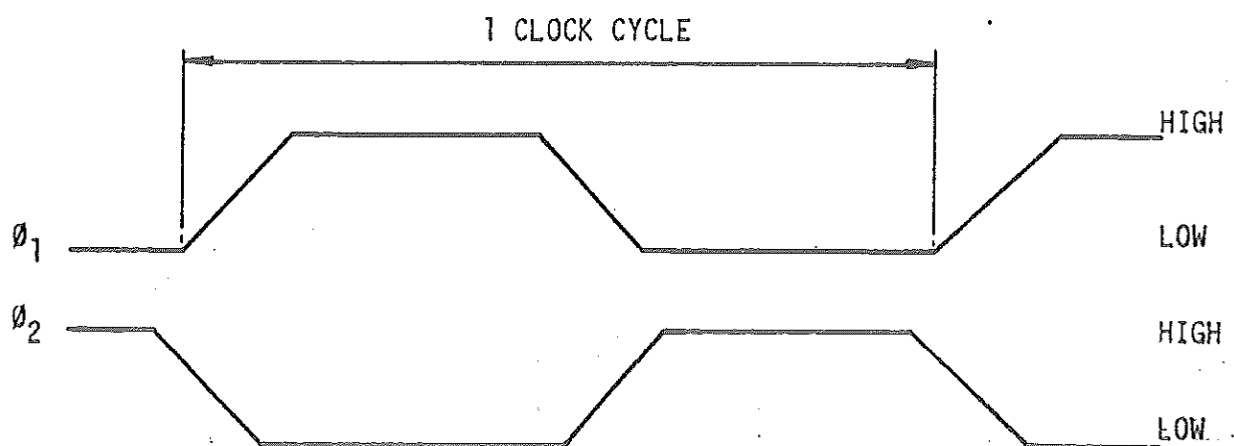


The arithmetic logic unit comprises many parts and performs all the necessary operations to implement two's complement arithmetic, logical operations such as ANDing two bytes of data and testing the results of these and also many more functions.

All these operations and many more are under the control of a Timing Control Unit.

The Control Unit (CU) keeps track of each specific cycle of operation which takes place within a data transfer or manipulation. Each instruction, for example when adding two bytes of data together, will require a number of cycles of operations to completely execute the requested instruction. It is the function of the timing control unit to provide these timing pulses. T0 will mark the first cycle at the beginning of each instruction, T1 the second cycle and Tn the last cycle to complete the instruction. The control unit derives its accurate timing from a system clock.

The Clock Generator is a device which produces two continuous waveforms whose frequency is accurately controlled by means of a quartz crystal (X1A1) which is external to the 6502 chip. The crystal maintains a frequency of 1MHz and the combination effectively acts as the 'heart beat' for the microcomputer system providing all timing reference points. We refer to the clock as being 'two phase' (ϕ_1 and ϕ_2) and non-overlapping. An exaggerated diagram of the waveforms is as shown below.



It will be noted that \emptyset_1 and \emptyset_2 are not at logic 1 (High) at the same time. That is they are not allowed to overlap. The events which take place within the microprocessor (or microcomputer system) depend upon whether \emptyset_1 or \emptyset_2 is at logic one.

The Instruction Decode/Instruction Register works closely with the Control Unit. The block diagram for the microprocessor can be seen to be split roughly into two parts; a 'control' side and a 'register' side. Data is processed in the register side and the sequence of operations performed for each instruction (such as ADD two bytes of data together) by the elements on the control side. However, the overall sequence of instructions that the microprocessor may be called upon to obey are dictated by the program of instructions. The data representing the program is latched into the INSTRUCTION REGISTER and then decoded by the INSTRUCTION DECODE circuitry, this, along with the timing control output, to generate **Control Signals** FOR EACH OF THE VARIOUS REGISTERS. These control signals may also be influenced by external incoming signals such as those produced by external devices interrupting the actual processing functions. These would appear via the **Interrupt Logic** Circuitry but others, such as the **Ready** (RDY) signal, may also hold up processing for various reasons.

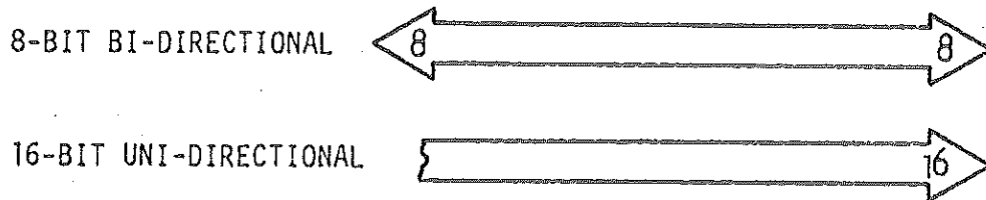
The Program Counter (PC) is simply a register which holds the address in memory of the next data item which the microprocessor will call for. The counter is incremented each time an instruction, or data upon which the instruction may operate, is fetched from the system memory.

The program counter may be modified, other than simply incremented, by events which may occur through execution of the program.

Index Registers (X) and (Y)/Stack Pointer (S) are simple 8bit latches which store data that is to be used in calculating addresses in data memory. We will discuss these in more detail when actually writing programs to run on Emma.

The Process Status Register (P) is a register which consists of eight individual latches. These may be set or cleared either under program control, or automatically by the microprocessor. The contents of these latches reflect the current **Status** of the microprocessor and may be inspected by the programmer under program control.

DATA AND ADDRESS BUSES are simply a group of electrically conducting paths grouped together by function. Data, in the form of a memory address or data to be operated upon, is pressed along these. The data bus is typically an 8bit parallel bus while the address bus is typically a 16bit parallel bus. It is normal to symbolise these buses as follows:



The data bus is normally bi-directional (data can pass in either direction) since data will be required to pass both into and out of the microprocessor.

The address bus is normally uni-directional (data passes in one direction only) since all addresses will be generated by the microprocessor.

Buffers are devices which "buffer" or isolate the microprocessor internal buses from the buses external to it. In the case of the 6502 they provide TTL compatible bus drivers delivering sufficient capacity for 1 standard TTL load and at least 130 pF.

The address bus buffers also provide latches which hold the addresses when used in accessing peripheral devices such as RAM, ROM and I/O.

The data bus has been seen to be bi-directional and as it connects together a number of devices all capable of feeding data on to it, conflict can exist. This is overcome by making the buffers enter a high impedance state (third state other than high and low) when not transferring data.

QUESTIONS

1. What is the accumulator and what is its purpose.
2. What is the ALU used for.
3. What is stored in the program counter?
4. Is the data bus unidirectional or bidirectional.
5. Why is it necessary to have a third: high impedance state other than logic '1' or '0') on some devices.

OBJECTIVES OF THIS CHAPTER

Having studied this chapter you should:

- Be familiar with the different types of memory device used in a microcomputer.
- Understand the function of the monitor program.
- Be familiar with the use of input/output ports for communication with external devices.
- Understand how the microcomputer execute a single instruction.

A TYPICAL MICROCOMPUTER SYSTEM

As we did with the microprocessor, we will relate our discussion to the **Emma** microcomputer. However, the devices used are common to microcomputers in general and hence generally applicable.

We have suggested that in general the microprocessing unit does not possess any program or data storage capacity. Also, it has limited capability to converse directly with the outside world in the shape of peripherals (printers, visual display units, control devices etc.). We will deal with each of these in turn.

Memory Devices are simply registers. We have already familiarised ourselves with the term 'register' as being a device capable of storing a binary word. If a number of registers are grouped together, each capable of being individually addressed and holding a single byte, we have the concept of a memory device. In practical terms these devices may contain 1024 (1K) or 2048 (2K) or more of memory locations. If the device is only capable of being READ (we can only look at the data already stored) it is known as a **Read Only Memory** or ROM.

If, on the other hand, we can WRITE (input data) into the device under program control as well as READ it, we refer to it as a **Random Access Memory** or RAM.

Both devices are, in fact, random access in the sense that we can access any desired memory location. We do not have to start at the first and work our way through until we arrive at the one we wish to either read (ROM) or read/write (RAM).

A simple ROM Device can only be written into once; there is no way in which the data stored can be erased. The **Eprom** however is a device which can be erased and then used again - it is both an erasable (E) and reprogrammable (P), ROM.

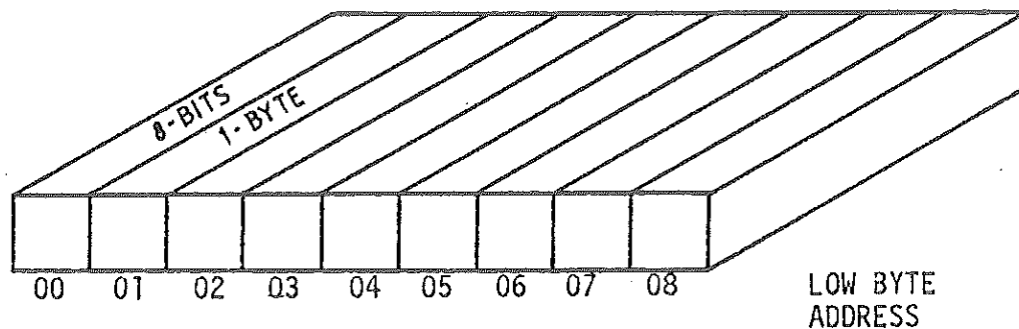
There is another feature of these devices which is of importance. ROM devices are non-volatile, that is when the supply is removed from the system the data contained within them remains; it is not lost. RAM devices lose their data upon loss of supply unless special provision is made to maintain the data, for example, provide a back-up battery. We would refer to such memory as **battery-backed RAM**. It should be noted that under conditions of 'back-up' the system cannot be used in an operational sense.

We may ask why use both ROM and RAM within a system. All microprocessors require a program which will operate automatically when the machine is switched on and/or when a reset button is pressed. This program is termed the **Monitor**. It is designed by the system manufacturer and must not be lost when the system supply is removed. It must be stored in non-volatile memory. Monitors have various features in common:

- They permit the system user to address any memory location and display its contents.
- They permit the user to modify the data stored in any addressed memory location.
- They permit the system user's program to take control of the microprocessor and to return control back to the monitor as required.

Various other features may be present. For example, the **Emma** monitor provides facilities for the loading or storing of programs from or to magnetic tape via a cassette tape recorder. It also provides a means of debugging user programs by allowing the operator to run a program and inspect automatically the content of important registers such as the accumulator and status register etc. In addition to all this, the **EMMA** monitor allows the programmer to make use of some of its program routines and hence simplify his program.

A schematic organisation of the individual memory functions is as shown in the diagram below:



Each location stores effectively 8-bits of encoded data and has a unique address encoded in hexadecimal for programming convenience.

A 2K EPROM would have 2048 locations starting at address 0000 and ending at address 07FF. We will return to the topic of memory addressing later.

INPUT/OUTPUT PORTS are referred to as I/O devices. Their function is to act as an input or output route for transferring data to or from microprocessor and application devices such as keypads, transistorised drive circuits for electromechanical devices (solenoids, stepper motors, relays etc.) and displays of various sorts.

I/O ports generally appear to the microprocessor as memory locations of 8-bits. Each bit can usually be set under program control to act as either an input port or an output port. Other features may well be available. For example the 6522 I/O port used in the **Emma** system comprises two 8-bit programmable I/O ports, two programmable timers and means whereby data can be input or output in a serial mode. It is a powerful device in its own right. We will look more deeply at I/O ports later.

Peripheral Device is a general term used to describe machines which are not physically part of a computer system, but may act in conjunction with it. They are used typically to display data either on a visual display unit or printer or store data on a disc or magnetic tape. The matrix printer and the cassette recorder which you will use with **Emma** are examples. Some of these devices may use an I/O port to communicate with the microprocessor (as when using the matrix printer) or a special interface as used for the cassette recorder. The display monitor used with the **Emma-Visa** combination uses a dedicated Cathode Ray Tube Controller (CRTC).

We have now completed a brief look at the various components which go to make up a microcomputer system. More complete details are to be found in the **Emma User and Technical Manuals** and you are encouraged to consult these.

We will now address ourselves to how these components operate within the concept of a microcomputer system.

MICROCOMPUTER SYSTEM OPERATION

We have already intimated that a microcomputer is capable of performing extremely complex functions by means of a series of simple operations. Control of the microcomputer system is largely the responsibility of the microprocessor. This causes the system to perform the desired operations by reading the first instruction of a program, and then 'executing' that instruction. The microprocessor will then 'fetch' the next instruction from memory, decode (interpret) it and then execute it. It will repeat these 'fetch/execute' cycles for each instruction until the program is terminated. Let us look at this in more detail.

A program instruction will usually, but not always, comprise two parts:



- The term **Op. Code** is short for 'Operation Code'. It is a single byte which specifies the type of operation which is to be performed. These codes are determined by the manufacturer of the microprocessor and are listed in the **Instruction Set** (see Appendix 7) for the particular microprocessor in use.
- The **Operand** may comprise either one or two bytes. If it is a single byte it may be data which is to be directly operated upon or it may specify the address of the data which is to be operated upon. Where two bytes are used for the operand, they specify an absolute address for data.

Let us consider a particular instruction which occurs frequently in any program of instructions:

Load Accumulator with Memory (LDA).

The instruction performs the operation - Transfer the contents of a specified memory into the accumulator. We can specify this operation symbolically as (M)-- A; where () means "contents of", and -- means "transfer to".

The specified Op. Code for this instruction is AD(HEX) (encoded in hexadecimal).

Let us assume the following:

- The data we wish to transfer into the accumulator is FF(HEX).
- The data is located at absolute address 0360(HEX).

- The program is to commence at absolute address 0020(HEX).

Our program is to be stored in RAM and can be tabulated as:

ADDRESS (HEX)	CONTENTS (HEX)	COMMENTS (BINARY)
0020	AD	1010 1101 OP. Code
0021	60	0110 0000 low byte of address
0022	03	0000 0011 high byte of address
0023	OP. Code of next instruction	

0360	FF	1111 1111 data to be transferred

Let us now see how the microcomputer would operate to effect this instruction. Under Monitor control, we would start the running of the program by pressing an appropriate control pushbutton or key. This would be detected and the user asked to provide the start address of the program to be run. Once accepted, system control would automatically be passed from the Monitor to the User program. We can best describe how our first instruction is executed by means of the table overleaf.

TIMING CONTROL CYCLE (t_n)	DATA ON ADDRESS	DATA ON DATA BUS	COMMENTS
T_0	P.C.	O.P. Code	Fetch OP. Code.
T_1	PC + 1	ADL	. Decode OP Code. . Fetch low order address byte.
T_2	PC + 2	ADH	Fetch high order address byte.
T_3	ADH, ADL	DATA	Fetch Data.
T_0	PC + 3	NEXT OP. CODE	. LOAD Accumulator with Data. . FETCH next op. code..

All instructions performed by the processor can be tabulated in this way and are in fact done so. You will find them under the heading 'Summary of Single Cycle Executions' in the **Emma Technical Manual**.

The important points to notice from the table are:

- The instruction takes a definite known time to execute.
- The data appearing on both the address and data buses for any clock cycle is known.

QUESTIONS

1. What do the following abbreviations stand for?
 - (a) RAM
 - (b) ROM
 - (c) EPROM
2. Which of the above are suitable for:
 - (a) Writing to?
 - (b) Random access?
3. How does the microcomputer access I/O ports?
4. What is an Op Code?
5. What is meant by (M) \rightarrow A?
6. What is the next operation performed by the processor after the completion of an instruction.

OBJECTIVES OF THIS CHAPTER

Having studied this chapter you should:

- Understand the basic difference between high level and low level programming language.
- Be able to construct a flow chart to perform a simple task.

INTRODUCTION

We have already defined a program as a series of instructions which the microprocessor can execute and hence perform some task. It is the format of such instructions to which we will now address ourselves.

PROGRAMMING LEVELS

The operations which we require the microprocessor to perform are expressed using binary codes. If we write our program directly in a binary format we are said to be writing a 'Machine Code Program'. These programs are at the lowest level possible, are extremely friendly to the machine (do not need machine translation) and require a minimum amount of memory. The timing of any instruction can also be accurately determined. They do, however, require of the programmer an intimate knowledge of the particular microprocessor for which he is writing the program. High level programs overcome this problem but at the premium of vastly increased memory requirements and an inability to predict accurately instruction timing. They are, however, extremely friendly to the programmer.

Programs are also written in 'Languages' and we frequently refer to high and low level languages. We will tabulate a few of the more frequently used languages.

LOW LEVEL LANGUAGES	INPUT
MACHINE CODE ASSEMBLE LANGUAGE	BINARY, HEXADECIMAL MNEMONICS

HIGH LEVEL LANGUAGES	APPLICATIONS
BASIC FORTRAN ALGOL COBOL	ALL PURPOSE. BEGINNERS COMPLEX MATHEMATICAL OPERATIONS ENGINEERING ORIENTATED BUSINESS ORIENTATED

All the High Level Languages are written in English-like statements. For example in **Basic** the single statement:

LET P = T + W

would be machine interpreted as add the value of W to the value of T and assign P to the result. In a low-level language this would involve a program of numerous instructions. However, before the machine could actually understand this statement and perform the necessary function it would have to translate it using a separate program which would be either a Compiler or an Interpreter. Either of these would have to be resident within the machine and obviously require memory space - this is just one of the overheads of high-level language use.

PROGRAM STRUCTURE

Obviously most tasks performed by human beings can be performed in many different ways. Also, there is not always one particular way which is 'best'. Programming is no exception to this. However, some thought to the structure of a program will prevent much wasted time when it is being given its first trial run.

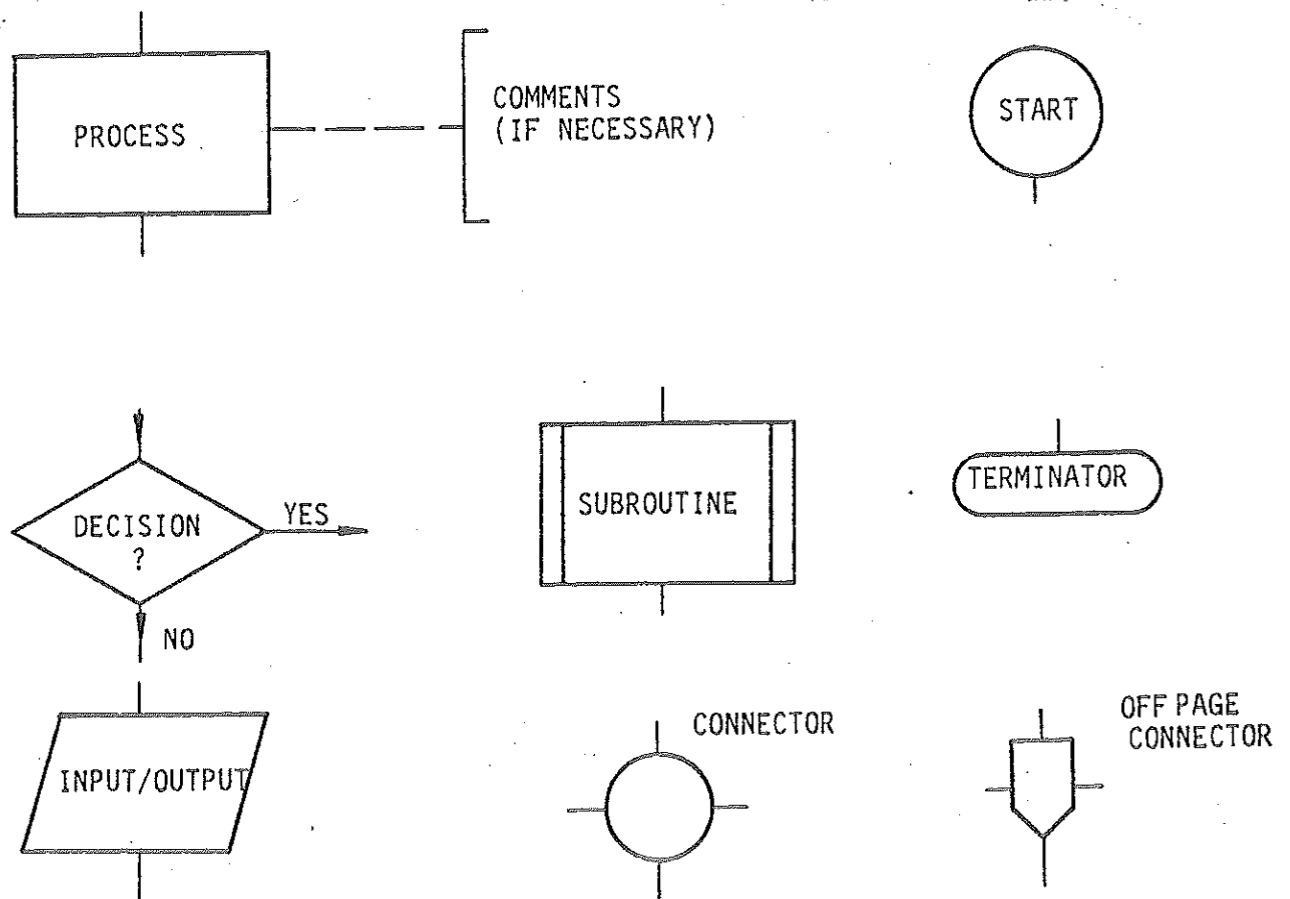
A well structured program will be one where the total task is broken down into a number of smaller self-contained tasks. The number of entry and exit points within the sub-task should be small: ideally one entry one exit. The nearer we get to this ideal, the easier it will be to isolate a particular part of a program and 'debug' it when it has failed to operate or perform in the way that we expected that it should.

To help us to this end a good programmer will construct a FLOW CHART.

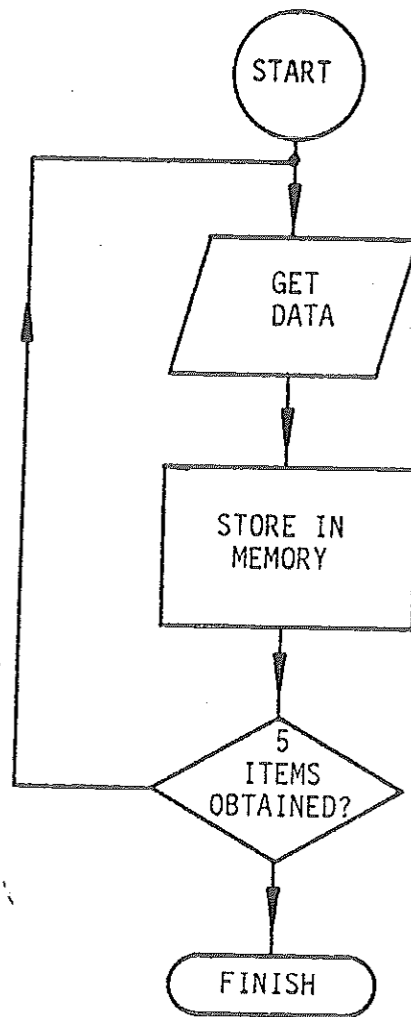
THE FLOW CHART

The Flow-Chart represents a program in schematic form. It has no reference to the assemble language of a particular machine or higher-level language. It simply helps to break large programs down into small modules which can be more easily worked upon.

Recognised symbols are used for flow-chart construction, frequently drawn by means of a template. The most common are outlined below:



A typical example of fetching say five items from a peripheral device and storing in memory would be:



The diagram should be self-explanatory. After each item is 'got' a check is made to see whether the correct number has been fetched: if not, the **'Get'** routine is repeated by looping back in our chart.

Before we can actually write a program we need to have some knowledge of the machine **Instruction Set**. We will consider this using that for the 6502 since EMMA is based upon this microprocessor.

QUESTIONS

1. What are the disadvantages of writing programs in machine code.
2. Should a high level or low level language be used if the timing of the program is critical.

Having studied this chapter you should:

- Have some familiarity with the manufacturer's instruction set for the processor we are going to use.
- Be aware that instructions can have several hexadecimal codes each relating to different addressing mode.
- Understand that some bits of the status register may be changed when an instruction is executed.

The instruction set for EMMA is to be found in Appendix 7, at the rear of this manual. However, for convenience, we will reproduce part of an instruction that we will be using to form our first simple program.

LDA Load accumulator with memory

LDA

N	Z	C	I	D	V
✓	✓	✗	✗	✗	✗

Addressing Mode	Assembly Language Form	Op Code	No Bytes	No Cycles
Absolute	LDA Oper	AD	3	4

We should notice the following:

- The operation is clearly stated as:

"Load accumulator with memory"

It is also symbolised by: $M \rightarrow A$

which means: "Transfer the contents of memory M to the Accumulator A".

- A Mnemonic is given:

LDA

meaning: "Load Accumulator".

- An addressing mode is given:

Absolute

This indicates that the data is to be found at the absolute address specified by the two bytes following the Op. Code.

- The format of the Assemble Language instruction is given:

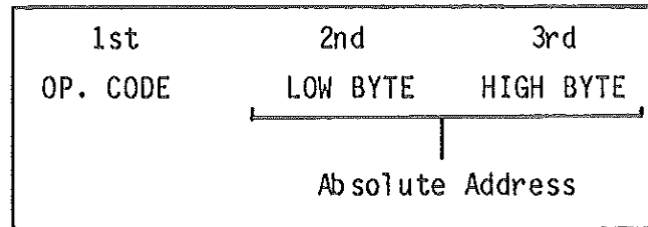
LDA Oper

where LDA specifies the Operation to be performed and is to be followed by an operand. In our case the operand is a two byte absolute address.

- a. A code is given in hexadecimal for the mnemonic LDA. This is

AD

- b. The total number of bytes are specified which make up the instruction i.e.



- c. The total number of machine cycles which the instruction will require are specified. This is
4 cycles.

Note: EMMA operates at a frequency of 1 MHz (1,000,000 cycles/sec).
The time taken to execute this particular instruction is therefore:

$$4 \times \frac{1}{1,000,000} = 4 \mu \text{ Sec}$$

An incredibly short time !!

- We have already mentioned the use of 'flags'. For example, when we did our Two's Complement arithmetic in Chapter 1.2 we found the need to detect a register 'overflow'. We provided a separate bit and set it to '1' if an overflow occurred. Our instruction set indicates the effect of the instruction upon the flags. There are effectively 6 individual flags which form the **Status Register**.

These are assigned as below:

- N - Negative Result
- Z - Zero Result
- C - Carry
- I - Interrupt Disable
- D - Decimal Mode
- V - Overflow

A tick (/) beneath a flag designation indicates that the instruction when executed may change the flag.

A dash (-) indicates that the instruction has no effect upon the flag.

A complete summary of the notation used prefaces the instruction set in Appendix 7.

Now let's see how we can employ this instruction and others in a simple program.

A SIMPLE PROGRAM EXAMPLE

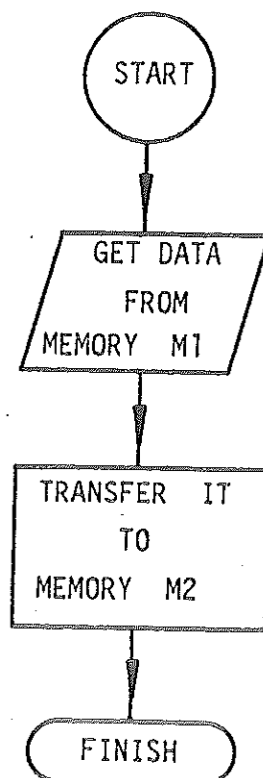
We will first specify the task, construct a flow chart and then write a program.

- TASK

Transfer the contents of memory M_1 to memory M_2 .

- FLOW CHART

This is so simple that a skilled programmer would not normally take the trouble! In fact, you may like to try this yourself first. The flow chart may look like this.



You will notice that this flow chart only breaks the task down, it does not specify what the data is or what the memories are. It could apply to almost any simple task not necessarily to be performed on a microcomputer.

● PROGRAM

We will write our program in machine code and therefore a commitment to a specific machine must be made. We will write the program to run on EMMA.

We have previously stated that all data movements must be made through the accumulator. We will therefore execute our task by:

- LOADING the data in memory M1 into the accumulator A.
- STORING the data transferred from M1 into memory M2.

Our program, using mnemonics, is:

LDA M1
STA M2

If we now assign addresses for M1 and M2 we get:

LDA 0080
STA 0081

where 0080 and 0081 are two absolute addresses with the high byte (00) specified before the low byte (80).

Note: It is normal to refer to addresses high byte first followed by low byte e.g. 0080 and 0081. However, when placing in memory as an **Operand**, the machine requires that we reverse this order.

Assigning addresses to our program we get:

0020	LDA	0080
0023	STA	0081
0026		next instruction op. code

Look up the instruction set (Appendix 7) and see if you can determine what addressing mode we have used for the store instruction.

If you turn to Appendix 1 you will also find a blank programming sheet. Reproduce this and in future use it to record your programs.

Our program (using a standard programming sheet) would look like this:

STANDARD PROGRAMMING FORM

PROGRAMMER: You and me

PROGRAM TITLE: 1st Program

HEXADECIMAL				SYMBOLIC ASSEMBLER INSTRUCTIONS			
ADDR	1	2	3	LABEL	MNEM	OPERAND	COMMENTS
0020	AD	80	00		LDA	M1	Address of M1 is 0080
0023	8D	81	00		STA	M2	Address of M2 is 0081
0026	4C	26	00		JMP	0026	Terminates Program
0080	?						Single byte of unspecified data to be transferred to memory 0081
0081							

You will notice that we have terminated our program using a **Jump** instruction.

If you do not understand everything we have just done **Do Not Worry** You will soon confidently be writing programs and **Running** them on EMMA.

Now let's get down to something more exciting! We are going to introduce you to EMMA.

QUESTIONS

1. Use the Instruction Set (Appendix 7) to determine the Op Codes for the following mnemonic instructions:
 - (a) LDA Zero Page
 - (b) BEQ
 - (c) PHA
 - (d) STA Absolute
2. If an instruction requires three machine cycles how long does it take to execute.
3. What is meant by a tick (✓) beneath one of the status flags for a particular instruction in the Instruction Set.

SECTION 3

PROGRAMMING THE EMMA MICROCOMPUTER

Chapter 3.1	Introduction to Emma	Page 1
Chapter 3.2	Program Entry and Execution	Page 8
Chapter 3.3	Using 6502 Instructions	Page 16
Chapter 3.4	Writing a Program	Page 26
Chapter 3.5	Arithmetic Operation	Page 42
Chapter 3.6	Logical Operations	Page 54
Chapter 3.7	Sub-Routines	Page 59
Chapter 3.8	Stack Processing	Page 63
Chapter 3.9	Software Delays	Page 66
Chapter 3.10	Interrupts	Page 74
Chapter 3.11	Using the VIA	Page 88
Chapter 3.12	Program Debugging	Page 109
Chapter 3.13	Using the Cassette Interface	Page 113

OBJECTIVES OF THIS CHAPTER

Having studied this chapter you should:

- Be familiar with general layout and features of the **EMMA** microcomputer.
- Understand the memory map of the **EMMA** microcomputer system.

GENERAL DESCRIPTION

EMMA is a fully assembled and tested microcomputer requiring only a 5V., 700 mA., regulated d.c. supply to commence computing.

The system is built around the powerful 6502 microprocessor, having a crystal controlled clock operating at 1 MHz. Operation is provided by a monitor program. The monitor is stored in a 2716 EPROM and the user program in 2K of RAM provided by 1 x 6116 integrated circuit.

- An important feature is the Input/Output port (I/O Port) facilities. These are provided by the 6522 Versatile Interface Adapter (VIA) which includes among other features, two 8-bit programmable I/O ports. Keyboard/Display interface is provided by the 6821 integrated circuit.

Of the total addressing capability only a limited amount is decoded. The required address decode is selected by links placed in an i.c. Header. Since other decode arrangements are possible it is important that the correct Header is used. The arrangement is shown below for the unexpanded EMMA II:



The cassette interface provides for rapid retention of programs on a standard cassette recorder. Connection to the recorder is made via the DIN input/output socket, or external microphone/earphone connectors.

The layout of the EMMA microcomputer is given on page 3.

Communication with **EMMA** is through the keyboard/display interface although you will soon make use of the I/O port which is brought out on 4mm sockets on the left-hand side of the microcomputer board. These sockets are designated PA0-PA7 and PB0-PB7 and provide the input/output connections to I/O Ports A and B respectively. Also available are 4mm sockets for interrupt facilities - these are used extensively in work associated with Application Modules.

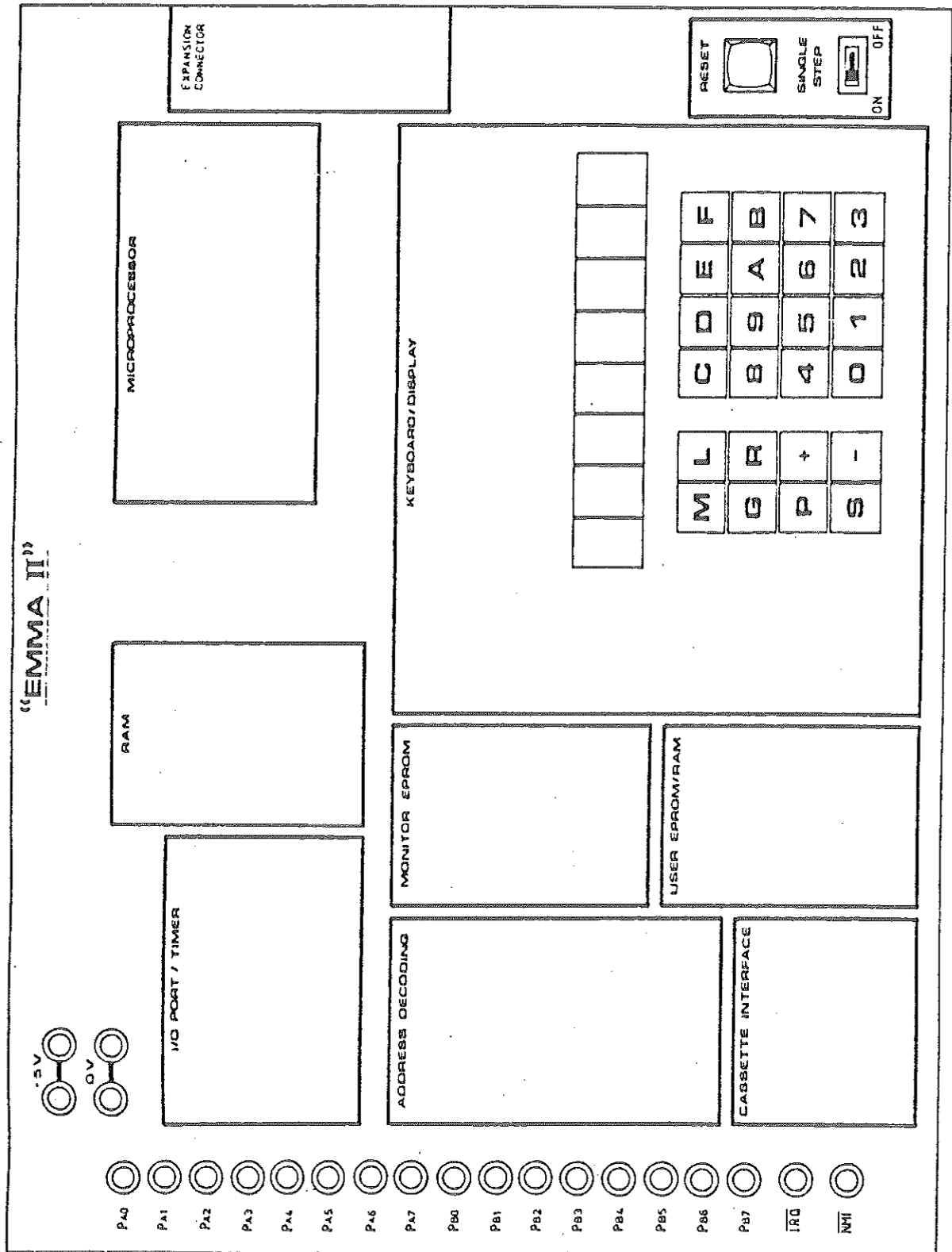
Power supply to the board is made via 4mm sockets, two sockets being provided for the +5V connection and two for the 0V connection. The provision of two sockets for each line facilitate the looping of supplies to other system items.

Access to all bus lines and various control signals is made through 0.1" printed circuit board (p.c.b.) pins. Again these will be used extensively during Fault Simulation and Diagnosis.

A 3-way 0.1" p.c.b. plug gives access to the cassette interface.

The single step facility considered in Chapter 3.12 of this manual must be switched to the OFF position for normal operation.

"EMMA II"



THE MEMORY MAP

You may remember we mentioned that certain locations, namely 0000 to 001F were reserved. There are other locations too which are not directly available to the user. This is common to all microcomputers. To help us find our way through the memory system we have what is termed a **Memory Map**.

A Memory Map is a listing of addresses which define the boundaries of the memory address space occupied by a program or series of programs. An unexpanded **EMMA** Memory Map is included in the Appendix.

USER MEMORY

Our major concern with the Memory Map at this point is to see what memory locations are available to us to store our programs. You should observe that available memory space is:

$$\begin{array}{l} 0020 \\ 00FF \end{array} \left. \vphantom{\begin{array}{l} 0020 \\ 00FF \end{array}} \right\} = 224 \text{ Bytes for user}$$
$$\begin{array}{l} 0200 \\ 03FF \end{array} \left. \vphantom{\begin{array}{l} 0200 \\ 03FF \end{array}} \right\} = 512 \text{ Bytes for user}$$
$$\begin{array}{l} 0C00 \\ 0FFF \end{array} \left. \vphantom{\begin{array}{l} 0C00 \\ 0FFF \end{array}} \right\} = 1024 \text{ Bytes for user}$$

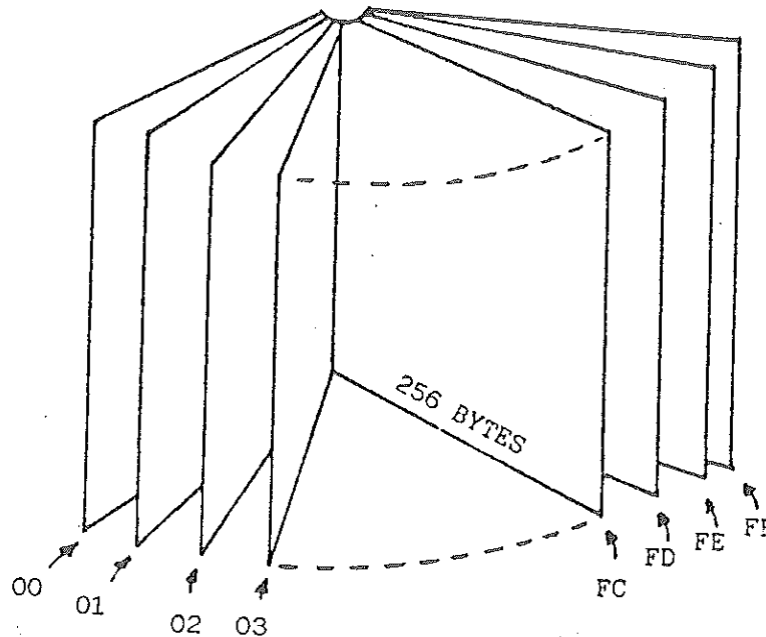
4 Bytes used in conjunction with useful sub-routines 001C - 001F

Now is an opportune moment for us to look at the two bytes which constitute an address:

THE CONCEPT OF PAGING

In common with most 8-bit microprocessors, EMMA has an address capability of 64K (65,536) memory locations. These are organised into **Pages** where a page is 256 consecutive memory locations. With this size of page, there are a possible 256 pages in 64K of memory.

Schematically we would show these as:



Each location would have a two-byte unique address, e.g.

HIGH BYTE	LOW BYTE
-----------	----------

where the high byte is the **Page Reference Number** and the low byte is the **Location on Page**.

Hence the address 0020(HEX) is:

20(HEX) location on page 00.

Page 00 and page 01 have instructions which are special to those pages.

ZERO PAGE

Zero page may be seen as comprising a set of working registers upon which any instruction will be executed in a shorter time than if any other page had been used. Since the time saving in executing a single instruction can be as much as 33.33% it is worthwhile reserving zero page for essential data that needs to be retrieved at high speed.

THE STACK

The stack is designated by the microprocessor as page one. Special instructions exist which operate only on the stack and serve to transfer and retrieve data 'pushed' onto and 'pulled' from the stack. None of these instructions specify a particular location on the stack - the location of the last data item pushed to the stack is 'remembered' in a register termed the STACK POINTER. We will meet the stack and it's pointer again during programming exercises.

THE STATUS REGISTER

We have already referred to the status register as comprising a number of 'flags' which inform the programmer of the status of the machine after each instruction is executed. The programmer has control over some of these flags, he can set or reset them as required by the logic of his program.

QUESTIONS

1. How much RAM is included in the system?
2. How much RAM is available for user programs?
3. How many 'pages' are contained in 1K of memory?
4. What are the advantages of using Zero page instructions?
5. What is page 01 of the EMMA memory used for?

OBJECTIVES OF THIS CHAPTER

- Having studied this chapter you should:
- Understand the functions of the **EMMA** control keys.
- Be able to modify the data stored at an address location.
- Be able to load and run a simple program.

EMMA KEY FUNCTIONS

The **EMMA** keyboard/display unit provides all the necessary controls for **EMMA** except for the Address Decode Patching header and the Reset Pushbutton.

The keyboard is split into two well defined key groupings:

- Hexadecimal Keys - These are the matrix of sixteen keys marked 0 - 9 and A - F. They are used to input all data e.g. user programs, data tables.
- Control Keys - These are the matrix of eight keys marked, M, G, P, S, L, R, + and -. These keys have well defined functions as below:

Program Entry - Programs are entered into **EMMA** using hexadecimal codes. Control of program entry is by means of the following keys:

- **M Key** - used to select the memory mode. For example the seven segment display may show either a memory address or a memory address and its contents. Depression of M will alternate these modes.

- + (plus) and - (minus) - used to increment or decrement the current program entry address. For example if the current address being displayed is 0020, pressing the + (plus) key will increment this to 0021. The use of these keys greatly facilitates program entry and subsequent checking before running program.

Program Run - The G key causes the program to **Run** (be executed). In practice having entered a program, G will be pressed and the start address of the program then keyed in. Pressing G again will cause the program to be executed.

Program Debug - Key R provides a single step feature and key P a means of inserting a forced break into a user program. Both these keys are essentially for program debug and are discussed more fully in Chapter 3.12 (Program Debug).

Program Dump - EMMA provides a feature for dumping programs onto magnetic tape using a conventional cassette tape recorder. Keys S and L are used to control this feature, S being for store onto tape, while L loads the microcomputer memory from tape. Both are discussed in Chapter 3.13 (Using the cassette interface).

These key functions are designated by the microcomputer monitor program. They may, however, be redesignated under user program control without destroying their normal function.

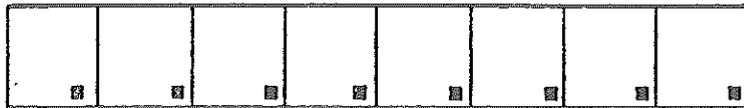
We will now follow a switch-on routine and familiarise ourselves with the actual **EMMA** Keyboard/Display.

EMMA SWITCH-ON

Connect **EMMA** to the 5V, 3 ampere supply outlet of System Power 90, carefully ensuring that the polarity is correct.

Switch on the supply.

Press the **Reset** button (bottom right-hand of the microcomputer board) and notice that the display shows eight decimal points:



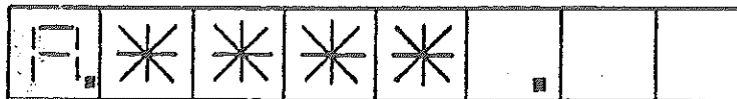
DISPLAY

This indicates that the monitor program is running and the microcomputer is ready to accept information from the keyboard.

It is also reasonable to assume that the microcomputer system is operating correctly.

FAMILIARISATION

Press the control key 'M' (memory key). The display will now indicate:



where * * * * is a 4 digit hexadecimal address between the values 0000 and FFFF.

This address can be modified by depressing the desired sequence of hexadecimal coding keys.

e.g. press 0 0 2 0

The display now shows:

A. 0 0 2 0

This is the address of the lowest user memory in **EMMA**.

Now press the 'M' key again. The display will now show the data stored at the location indicated by illuminating the two right-most 7-segment displays. Example, if the address part of the display shows A.0020 then pressing 'M' will cause the display to show:

A.0020 . * *

where * * are any two hexadecimal digits. Pressing any of the hexadecimal coding keys will modify this data.

Now press keys A, B, C, D, E and F and observe that some characters on the display are in capitals and some in lower case:

A b c d E and F

Note: the letter **b** and the number **6** are similar and care must be taken not to misinterpret these two characters.

We refer to the two parts of the display (as used above) as the **Address Field** and **Data Field** respectively.

Example:

A	0	0	2	0	.	4	8
---	---	---	---	---	---	---	---



Address Field

Data Field

Memory Modification

Now perform the following operation:

OPERATION	DISPLAY	COMMENTS
Press Reset	Monitor running
Press control key M	A. * * * *	Address field only illuminated and showing a random address.
Press Hex Keys 0020	A.0020 .	Address field indicates address high byte (00) address low byte (20)
Press Hex Keys 0238	A.0238 .	Address modified to 0238
Press M	A.0238 . * *	Address field <u>and</u> data field illuminated. Data field shows random data stored at location 0238.
Press Hex Keys 45	A.0238 .45	Data field shows data (45) input to location 0238.

The function of the control keys plus (+) and minus (-) can now be explored. These increment (increase by one) and decrement (decrease by one) the displayed address field when in the data mode. They provide a convenient way of sequentially moving through a series of addresses when entering a program or simply checking a program already entered without having to continually use the M control key.

We will now try entering an actual program.

PROGRAM LOADING

We have already written a simple program which is designed to transfer the contents of one memory location to another. We will repeat this for convenience.

PROGRAM		BYTES			COMMENTS
ADDRESS	1	2	3		
0020	AD	80	00	(M1) → A	
0023	8D	81	00	(A) → M2	
0026	4C	26	00	Terminates Program	

DATA		BYTES		COMMENTS
ADDRESS	1	2	3	
0080	FF			Data (FF) to be transferred
0081	00			Data (00) will be over written

You may notice that we are going to load memory location 0080 and 0081 with FF and 00 respectively. We have deliberately put 00 in 0081 so that we will positively know that FF has been transferred.

The program once entered into the machine would appear schematically in sequential locations as:

AD	80	00	8D	81	00	4C	26	00	FF	00	CODED MEMORIES
0	0	0	0	0	0	0	0	0	0	0	MSB
0	0	0	0	0	0	0	0	0	0	0	ADDRESSES
2	2	2	2	2	2	2	2	2	8	8	
0	1	2	3	4	5	6	7	8	0	1	LSB

Now **Load** this program!

- RESET EMMA - Press RESET
- Obtain Address Field - Press M key
- Set 1st Address (0020) - Press 0, 0, 2 and 0.
- Obtain Data Field - Press M key
- Modify Data Field - Set to AD
- Increment Program - Press + key
- Modify Data Field - Set to 80
- Increment Program - Press + key

Continue until program is entered.

Now **Load** Data.

- Obtain Address Field - Press M
- Set Address (0080) - Press 0, 0, 8 and 0

Etc. Continue for 0081

You should now have entered the whole of the program and the data. It is prudent to check this by looking at each loaded location. You may do this by incrementing or decrementing through the program and data memory.

Now let's RUN the program!

PROGRAM RUN

Press control key G (G stands for GO) - the display should now show:

□ * * * *

where * * * * is some random address.

Using the hexadecimal coding keys, modify this random address to the start address of the program e.g. 0020

Display:

0 0 2 0 0

Press the go key, G, again.

You will notice that the display has gone blank. To check whether the program has indeed run successfully we now need to inspect the memory location into which we transferred the contents of location 0080. Remember - contents of 0080 was FF and should have been transferred to 0081.

Symbolically:

(0080) \longrightarrow 0081

where () indicates contents of
and \longrightarrow transfer to.

Inspect the memory contents, press **Reset** followed by M, key in address 0081, press M again and data field of display should indicate FF.

Now inspect location 0080. You will notice that it still contains FF. Transferring contents of a memory does not destroy it.

You may not understand what you have been doing, but you should have familiarity with **EMMA** and its Keyboard/Display. In the next chapter we will look at program writing in more detail. In the meanwhile you may feel sufficiently confident to load other data and transfer to different locations. TRY !!

OBJECTIVES OF THIS CHAPTER

Having studied this chapter you should:

Understand the various addressing modes which are available.

Understand how to calculate the data for a relative branch instruction.

Understand the function of each of the flags in the processor status register (P register).

INTRODUCTION

In this chapter we will concentrate on the 6502 instruction set, paying special attention to the different addressing modes available for some instructions.

Firstly we will classify the instructions into functional groups.

CLASSIFICATION OF INSTRUCTIONS**Data Transfer Instructions**

LDA	Load Accumulator with Memory	STA	Store Accumulator in Memory.
LDX	Load X Register with Memory	STX	Store X in Memory
LDY	Load Y Register with Memory	STY	Store Y Register in Memory
TAX	Transfer Accumulator to X Register	TXA	Transfer X Register to Accumulator
TAY	Transfer Accumulator to Y Register	TYA	Transfer Y Register to Accumulator

Arithmetic Operation Instructions

ADC	Add Memory to Accumulator with Carry	SBC	Subtract Memory from Accumulator with Borrow
-----	--------------------------------------	-----	----------------------------------------------

Logic Operation Instructions

AND	AND Memory with Accumulator	ORA	OR Memory with Accumulator
EOR	EXCLUSIVE-OR Memory with Accumulator		

Shift And Modify Instructions

DEC	Decrement Memory by One	INC	Increment Memory by One
DEX	Decrement X Register by One	INX	Increment X Register by One
DEY	Decrement Y Register by One	INY	Increment Y Register by One.
ASL	Shift Left One Bit	LSR	Shift Right One Bit
ROL	Rotate Left One Bit	ROR	Rotate Right One Bit

Test Instructions

CMP	Compare Memory and Accumulator	CPX	Compare Memory and X Register
BIT	Test Bits in Memory with Accumulator	CPY	Compare Memory and Y Register

Branch Instructions

BCC	Branch on Carry Clear	BCS	Branch on Carry Set
BEQ	Branch on Result Zero	BNE	Branch on Result Not Zero
BMI	Branch on Result Minus	BPL	Branch on Result plus
BVC	Branch on Overflow Clear	BVS	Branch on Overflow Set

Modify Processor Status Register Instructions

CLC	Clear Carry Flag	SEC	Set Carry Flag
CLD	Clear Decimal Mode	SED	Set Decimal Mode
CLI	Clear Interrupt Flag	SEI	Set Interrupt Flag
CLV	Clear Overflow Flag		

Jump Instructions

JMP	Jump to New Location	RTS	Return from Sub-routine
JSR	Jump Sub-routine	RTI	Return from Interrupt Routine
BRK	Jump to Interrupt Routine		

Stack Operation Instructions

PHA	Push Accumulator on Stack	PLA	Pull Accumulator from Stack
PHP	Push P Register on Stack	PLP	Pull P Register from Stack
TXS	Transfer X Register to Stack Pointer	TSX	Transfer Stack Pointer to X Register

Do Nothing Instruction

NOP No Operation

There are other ways in which we can describe an instruction, for example:

- | | |
|----------------------|-------------------------------|
| • Mnemonic | - LDA |
| • Logical Expression | - $M \rightarrow A$ |
| • Operation Code | - AD |
| • Spoken Language | - Load the accumulator A with |
| Statement | a data byte from memory M. |

INSTRUCTION ADDRESSING MODES

Each instruction also has an Addressing Mode. The 6502 can perform 56 different operations, some of which can be executed in as many as eight different ways so producing 150 variations.

These addressing modes can be summarised as:

● Implied

Implied addressing uses a single byte instruction which operates on registers whose address is implied by the particular OP. Code used. These registers are those internal to the microprocessor - index registers, status register, stack pointer and external to the microprocessor (in memory) - the stack and interrupt vector locations.

● Immediate

All instructions in immediate addressing mode are two bytes long. The first is the OP. Code and the second specifies a constant or literal which is to be loaded into an internal register or external memory location.

● Absolute

You have already met this type of addressing mode. Instructions require three bytes of which the second two specify the location of the operand.

● Zero Page

Requires two bytes. Zero page is implied in Op. Code and therefore not specified implicitly. Only location on zero page is required.

● Relative

Requires two bytes. Instructions using this addressing mode are of the Branch type. They cause the microprocessor to branch to another part of the user program rather than execute the next instruction in sequence. The branch is taken upon the result of a test performed on the condition of flags within the status register. The second byte specifies the extent of the 'branch' that is the amount of program displacement and its direction relative to the address of the Op. Code of the instruction following the branch instruction.

Example:

BRANCH VALUE	PROGRAM ADDRESS	PROGRAM (BYTES)	
		1	2
	XXXX	OP.CODE	04
	(XXXX) +2	next sequential Op. Code	
+1	+3		
+2	+4		
+3	+5		
+4	+6	Op. code of Instruction executed if Test demands a 'branch'.	

The 'branch' illustrated is a 'forward' branch. Reverse branches can also be made.

Example:

BRANCH VALUE	PROGRAM ADDRESS	PROGRAM (BYTES)	
		1	2
-4	-4	Op. Code of Instruction executed if test demands branch	
	-3		
	-2		
	(XXXX) -1		
-3	XXXX	Op. Code FC next sequential Op. Code.	
-2	(XXXX) +2		
-1			

Note: Branch relative values are specified in Two's Complement, thus FC is 'branch back' four places.

If the branch is not taken the next sequential instruction is executed.

● Indexed

The 6502 is equipped with two index register, X and Y. The contents of an index register is added to a base address specified in the address field to modify that address.

Example:

PROGRAM ADDRESS	PROGRAM (BYTES)		
	1	2	3
XXXX	Op. Code	60	00
		└──────────┘ BASE ADDRESS	
(0060)+X	DATA		

Remember: Address low bytes precede high bytes in address field.

The method of addressing enables data tables to be sequentially accessed by performing increment (or decrement) operations on the index registers.

● Indirect

The concept of indirect addressing enables the address field following an Op. Code to specify an address of an address of data. An example will best indicate:

PROGRAM ADDRESS	PROGRAM (BYTES)		
	1	2	3
XXXX	Op. Code	60	02
0260	80 (low byte)		
0261	03 (high byte)		
0380	DATA		

The example indicates a pure indirect address mode and is only applicable in the 6502 to the JUMP instruction. However, two other modes of indirect addressing are possible but operating off zero page only and not an absolute address.

● Indexed X, indirect

This mode adds the contents of the index register X to the zero page address specified immediately following the Op. Code.

PROGRAM ADDRESS	PROGRAM (BYTES)		
	1	2	3
XXXX	Op. code	60	
00(60+X)	9A (low byte)		
00(61+X)	03 (high byte)		
039A	DATA		

The mode is only applicable to the X-Register and is shown logically in the instruction set as: (Indirect, X).

● Indirect, indexed Y

This mode adds the contents of the index register Y to the data base address.

Example:

PROGRAM ADDRESS	PROGRAM (BYTES)		
	1	2	3
XXXX	Op. code	60	
0060	9A (low byte)		
0061	03 (high byte)		
(039A) +Y	DATA		

This mode is only applicable to the Y-Register and is shown logically in the instruction set as (Indirect), Y.

STATUS FLAGS

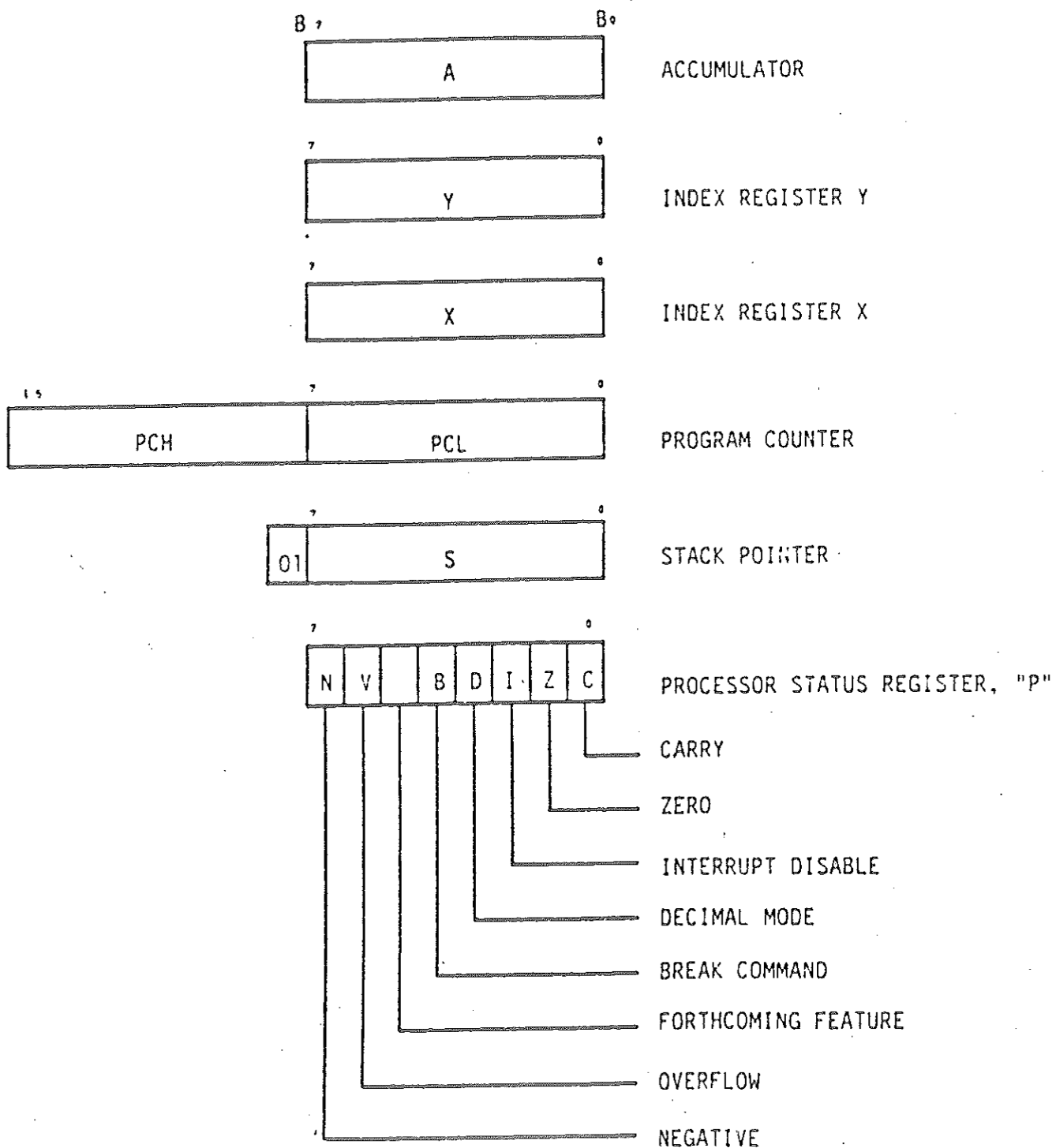
We mentioned in Chapter 3 that the Status register comprised a number of flags some of which are set or reset by the result of operations involving the arithmetic unit. The testing of these flags is an important part of any programming task. Below is a brief description of each flag:

The Status flags are:

- N - set if the most recent operation performed in the arithmetic unit gave a negative result.
- V - this flag indicates when the 7-bit result of a signed number arithmetic operation overflows.
- B - this is a break command flag. It is set by the microprocessor when an interrupt is caused by a break command.
- D - when this flag is set any arithmetic operations will be performed in binary coded decimal. With this flag cleared the arithmetic unit operates in true binary
- I - is the interrupt disable flag. When this flag is set the IRQ input will not interrupt the microprocessor.
- Z - set if the operation in the arithmetic unit gave a zero result.
- C - is a carry input to the arithmetic unit. If set, it will apply a '1' to the least significant bit of the arithmetic operation. An overflow from an 8-bit arithmetic operation sets the carry flag.

A PROGRAMMING MODEL

Finally we present a programming model of all the internal registers of the microprocessor.



QUESTIONS

1. How many bytes does a zero page instruction require.
2. How many bytes does an absolute instruction require.
3. Use the manufacturer's instruction set (see Appendix7) to obtain the hexadecimal codes for the following instructions:
 - (a) LDA Zero Page
 - (b) STA Absolute
 - (c) STX Absolute
 - (d) BEQ
 - (e) ADC Absolute, X
4. If there is a branch instruction at 0030 what data must be inserted at 0031 to cause the program to jump to:
 - (a) 0040
 - (b) 0020if the branch is taken.
5. What is the function of the following page in the status register.
 - (a) N
 - (b) V
 - (c) Z

OBJECTIVES OF THIS CHAPTER

Having studied this chapter you should:

- Be able to write a simple program for the EMMA microcomputer to transfer a block of data from one area of memory to another.
- Understand the concept of indexed addressing and its uses.
- Be able to calculate the data for positive and negative relative branch instructions.

INTRODUCTION

We will now construct a few programs which will use the more commonly used instructions and addressing modes.

The object of a program is to perform a task. A good programmer will attempt to write his program so that not only is the task performed but that it is done efficiently in terms of program execution time and memory requirement. He will also keep in mind the structure of the program so that program testing is more easily carried out. Finally he will maintain good documentation in terms of flow-charts and annotated programs which make for easier future modification.

We will now construct a series of simple programs. For each we will:

- state the task
- construct the flow-chart
- write the program

Our first program will use just three instructions: LDA, STA and JMP

PROGRAM - DATA MOVE

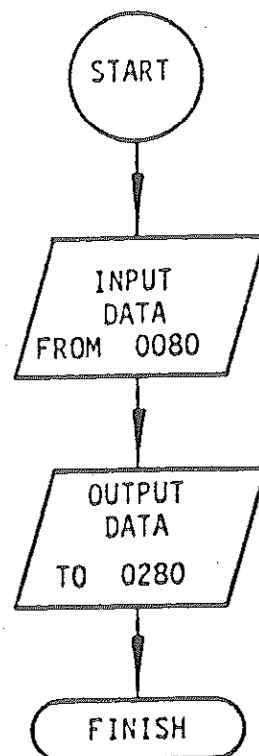
● TASK

Move a single byte of data from memory location 0080 to memory location 0280. Logically:

(0080) \longrightarrow 0280

Note: the brackets mean "the contents of".

● FLOW-CHART



● PROGRAM

This demands three basic operations to effect the transfer of data.

1. LOAD data into accumulator from memory
(0080) \longrightarrow A
2. STORE data in accumulator in memory
(A) \longrightarrow 0280
3. TERMINATE program.

We will now look at each of these steps in turn:

1. The data is on ZERO PAGE.

00	80
Page	Location

Now ask yourself if a LOAD instruction is available which operates directly on Zero Page (Consult Instruction Set - Appendix).

The Instruction Set should reveal the following:

Addressing Mode	-	Zero Page
Mnemonic	-	LDA
Op. Code	-	A5
Number of Bytes	-	2

We can write the instruction in **either**:

- Symbolic Machine Code -

OPERATION (Mnemonic)	OPERAND (Address)
LDA	80

where 80 is the Zero Page memory location.

- Hexadecimal Machine Code -

OP. CODE (Hexadecimal)	OPERAND (Address)
A5	80

2. Now let's look for a STROBE instruction.

The Instruction Set will reveal:

Addressing Mode - Absolute
Mnemonic - STA
Op. Code - 8D
Number of Bytes - 3

We cannot use zero page addressing mode since the data is stored on page 02.

Again we can write the instruction in two ways.

• Symbolic Machine Code -

OPERATION (MNEMONIC)	OPERAND (ADDRESS HIGH BYTE) (ADDRESS LOW BYTE)	
STA	02	80

• Hexadecimal Machine Code -

OPERATION	OPERRAND (ADDRESS LOW BYTE) (ADDRESS HIGH BYTE)	
8D	80	02

Note the way the operand has been written. When writing addresses it is normal to write HIGH BYTE followed by LOW BYTE. However, we **Enter the Hexadecimal Codes** into the machine (6502) LOW BYTE first. If you follow this practice when using the **Standard Programming Form** you are less likely to make mistakes when entering your program using the hexadecimal keyboard.

Now let's look at terminating the Program. If you scan the Instruction Set you will not find an instruction which you can use directly for this purpose. The 6502 Instruction Set does not have an instruction such as **Stop**, **Halt** or indeed **Finish**.

First let's consider what would happen if we did not bother, after all our two instructions will effectively complete the task! Unless we tell the machine to stop processing when the transfer is complete it will continue to fetch, sequentially, data from memory. Unfortunately, this may be either another program or, as is more likely, rubbish (the memories will always have something in them; they cannot be 'empty').

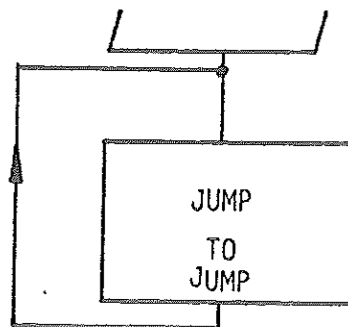
A simple way round the problem is to cause the machine to enter a 'program loop' out of which it can only get by the user pressing the RESET pushbutton. A JUMP instruction will do this:

The Instruction Set shows:

Addressing Mode	-	Absolute
Mnemonic	-	JMP
Op. Code	-	4C
Number of Bytes	-	3

We will use this instruction to jump back to itself.

Example:



Using Machine Code -

OPERATION (Mnemonic)	OPERAND (Address)
JMP	TERMINATE

In the Operand field we have used a **Label**. We can do this using symbolic notation. The label stands in place of a **Program Address**. We cannot enter the program address that we wish to jump back to because we have not yet allocated memory space for our program.

It is worthwhile mentioning here that the programmer invents his own labels for programming convenience. However, a simple guide to labelling is **DO NOT use labels that look like or contain Operation Code Mnemonics**.

We can now collate our instructions and enter them on a programming sheet.

Example:

HEXADECIMAL				SYMBOLIC ASSEMBLER INSTRUCTION			
ADDR	1	2	3	LABEL	MNEM	OPERAND	COMMENTS
0200	A5	80			LDA	80	
0202	8D	80	02		STA	02 80	
0205	4C	05	02	TERM	JMP	TERM	

You will notice that we have assigned memory addresses (shown shaded) to our program. This has also enabled us to assign a program address to the label TERM.

One final comment regarding the sequence in which the programming would normally be done.

- Construct Flow Chart. Annotate.
 - Write the Symbolic Assembler Instruction, using labels if necessary. Annotate.
 - Encode the hexadecimal machine code program.
 - Assign memory space to program and any associated data.
 - Assign addresses to any labels used.
-

● EXERCISE 3.4.1

Enter the program in **EMMA** and execute. Do not forget to enter appropriate data in memory locations 0080 and 0280. If you have any doubts regarding program entry procedure, go back and study Chapter 3.2 again.

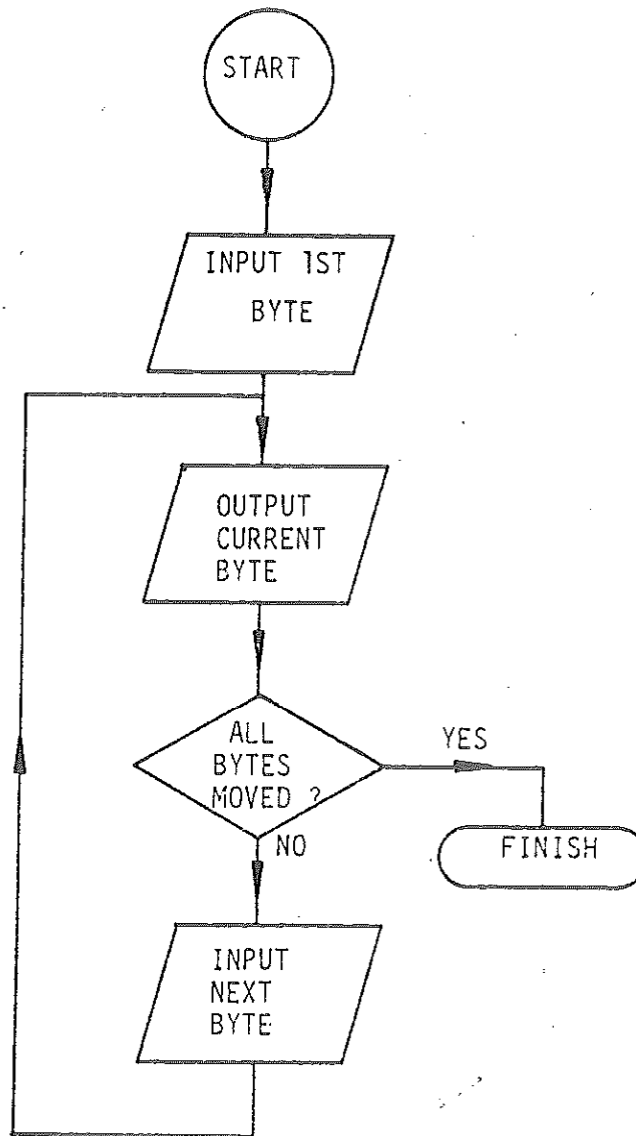
PROGRAM - DATA BLOCK MOVE, MK1

- TASK - Move a block of data bytes from and to the memory locations shown below:

MEMORY ADDRESS FROM	DATA	MEMORY ADDRESS TO
0020	01	02F0
0021	02	02F1
0022	03	02F2
0023	04	02F3
0024	05	02F4

Logically: (0020 - 0024) → 02F0 - 02F4

FLOW-CHART

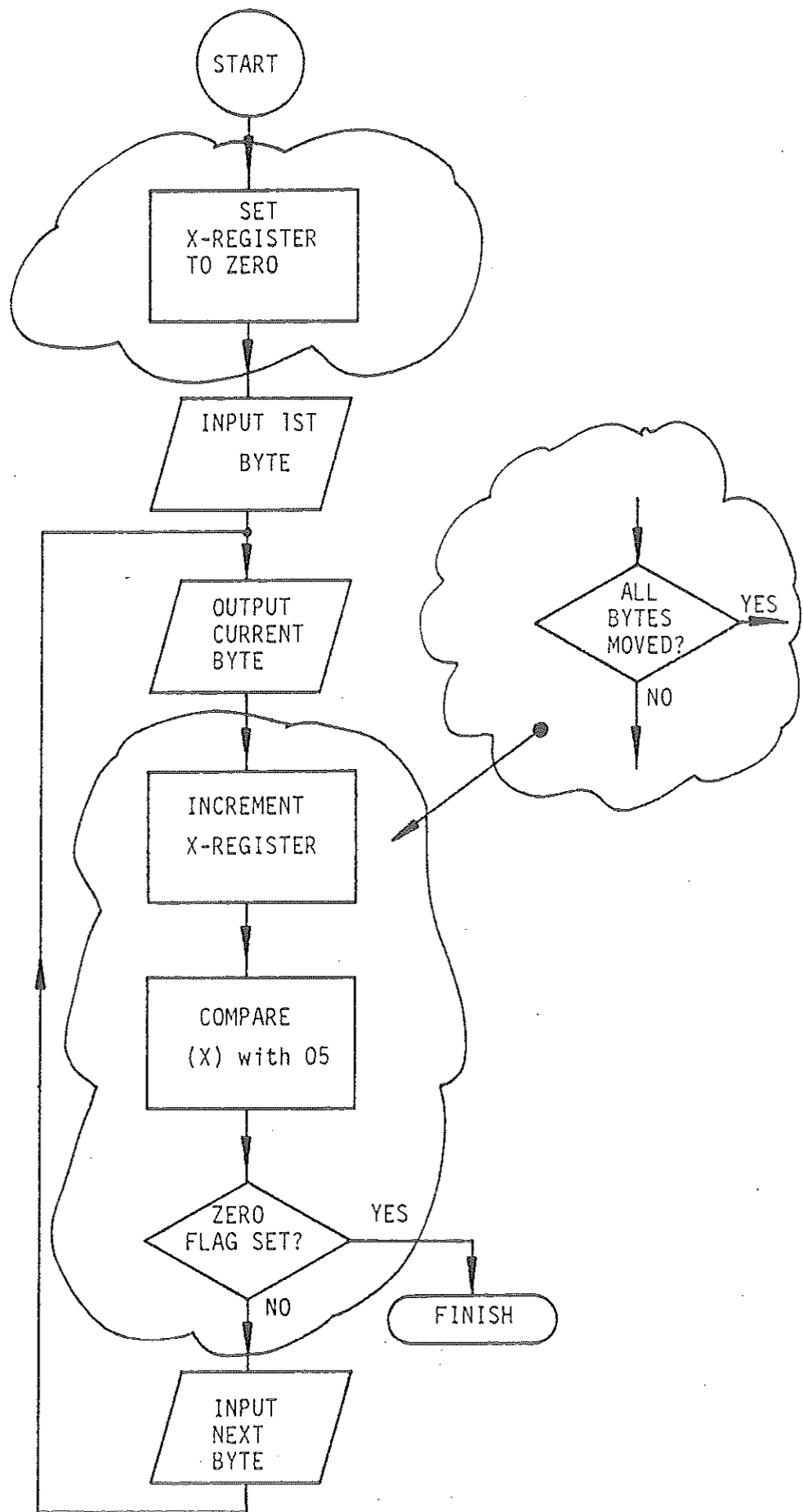


PROGRAM - The same three basic instructions are required as for Program 1, except that we now require to **Test** to see if the last byte moved was in fact the last byte in the data block. It is this part of the program that we shall concern ourselves with.

What we require is some mechanism for counting the bytes transferred.

A convenient way of doing this is by using the X-register, setting it to zero and then incrementing by one each time a byte is transferred. If we compare the X-register contents with the number of bytes to be transferred, and use this test to determine whether to terminate the program or input another byte, we will have achieved our objective.

We will redraw our flow-chart to help us decide what instructions to use.



Now to find suitable instructions.

- Set X-register to zero.

LDX# 00 (Immediate)

- Increment X-register

INX (Implied)

- Compare X-register contents with 05

CPX# 05 (Immediate)

- Branch if zero flag set

BEQ BRANCH OFFSET (Relative)

- Input data

LDA OPER, X (Zero page Indexed)

- Output Data

STA OPER, X (Absolute Indexed)

The LDA and STA instructions are INDEXED by value in X-register. The difference between the 2-byte and the 3-byte instructions is that data to be transferred is on ZERO PAGE. A zero page instruction can be used.

We will now collate these instructions and others necessary to complete our task.

SYMBOLIC ASSEMBLER INSTRUCTIONS			
LABEL	MNEM	OPERAND	COMMENTS
	LDX#	00	Set byte count
	LDA	20,X	20 is location of 1st byte
TRANS	STA	02 F0,X	Byte Transfer
	INX		Increment byte count
	CPX#	05	Bytes
	BEQ	FIN	Transferred?
	LDA	20,X	Next byte
	JMP	TRANS	Go to TRANS
FIN	JMP	FIN	END of PROGRAM

We will now encode these into hexadecimal machine code and enter on a coding sheet. We will also assign our program to memory space 0200 onwards.

PROGRAMMER: USER

PROGRAM TITLE: DATA BLOCK MOVE, MK1

HEXADECIMAL				SYMBOLIC ASSEMBLER INSTRUCTIONS			
ADDR	1	2	3	LABEL	MNEM	OPERAND	COMMENTS
0020	01						
0021	02						
0022	03						
0023	04						
0024	05						
0200	A2	00			LDX#	00	Initialise Data Table pointer
0202	B5	20			LDA	20,X	
0204	9D	F0	02	TRANS	STA	02 F0,X	
0207	E8				INX		Move up data table
0208	E0	05			CPX#	05	table end?
020A	F0	05			BEQ	FIN	
021C	B5	20			LDA	20,X	Get next Byte
021E	4C	04	02		JMP	TRANS	Go to transfer next byte
0221	4C	21	02	FIN	JMP	FIN	Terminate
02F0							
02F1							
02F2							
02F3							
02F4							

LABELS

0204 assigned to TRANS

0221 assigned to FIN

● EXERCISE 3.4.2

Enter the program and associated data and prove program performs required task.

REVIEW

We have now used the following addressing modes:

- Immediate
- Zero page
- Zero page, Indexed
- Absolute
- Absolute, Indexed
- Implied
- Relative

We have also used the following classifications of instructions:

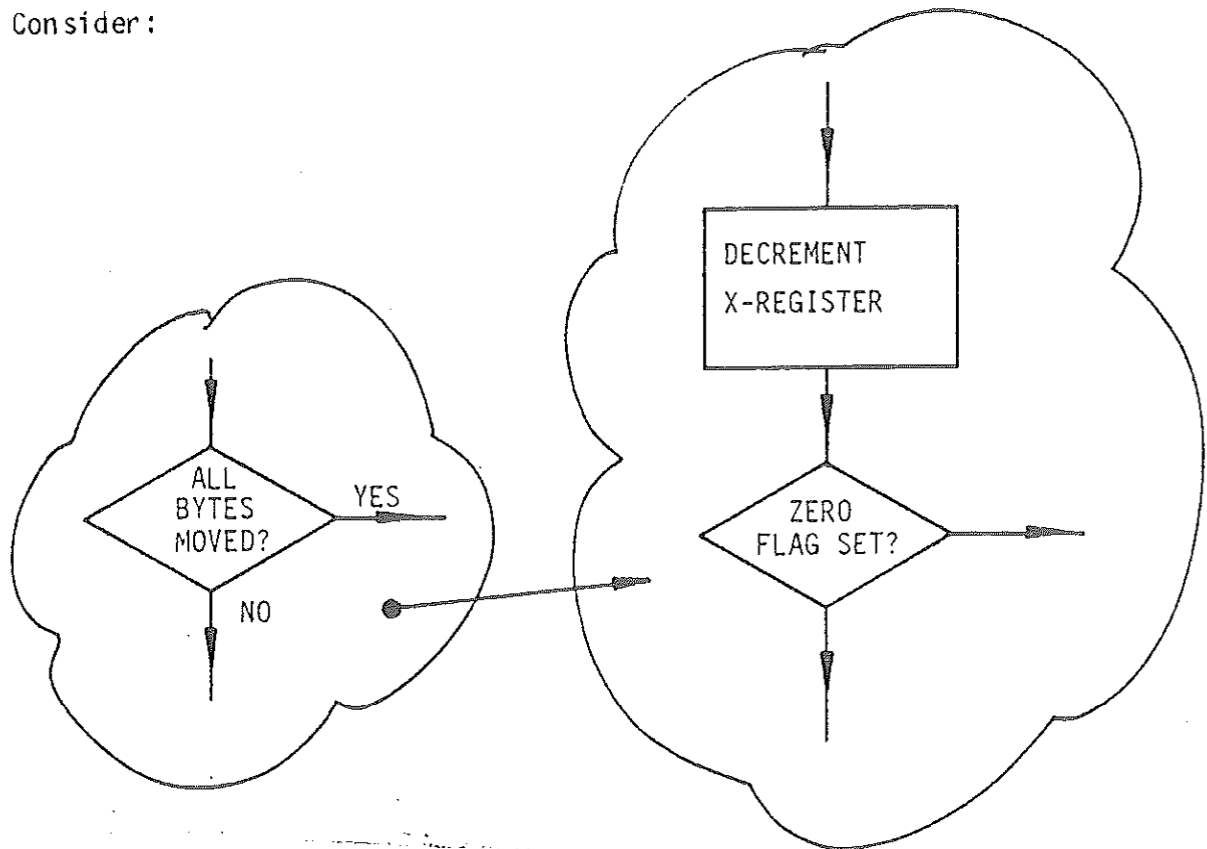
- | | |
|-----------------------|-----|
| • Data Transfer | LDA |
| • Shift and Modify | INX |
| • Test Instructions | CMP |
| • Branch Instructions | BEQ |
| • Jump Instructions | JMP |

You should now aim to check all the coding we have used and thoroughly understand the addressing modes and why we used them before going any further.

PROGRAM - DATA BLOCK MOVE, MKII

Both the task and initial flow-chart are the same as for **Program - Data Block Move, MKI**. The difference is in the way we implement the 'All Bytes Moved?' part of the program.

Consider:



You should notice that there is no compare instruction. To implement this we will load the X-register so that the last data byte is transferred first and then descend the table by decrementing the X-register. A disassembled listing of the required modifications compared to the original is:

ORIGINAL PROGRAM	MODIFIED PROGRAM
LDX# 00	LDX# 05
INX	DEX
CPX# 05	
BEQ FIN	BEQ FIN

You will notice that the compare instruction has been made redundant making the program more efficient - less time to execute; less program storage space required.

● EXERCISE 3.4.3

Re-write completely: Program - Data Block Move, MKI, incorporating the modifications of:

Program - Data Block Move, MKII.

Encode the program load EMMA and RUN.

The technique used above is a very useful one for sequentially accessing data tables.

● EXERCISE 3.4.4

Modify: Program - Data Block Move, MKII, to extend the length of the data table to 10 items.

PROGRAM - DATA BLOCK MOVE, MKIII.

We can further modify our Data Block Move Program to make it even more efficient. Here is a listing.

```
        LDX# 05
NEXT    LDA 20,X
        STA 02F0,X
        DEX
        TXA NEXT
FIN     JMP FIN
```

The program is based upon the original data table of 5 items.

Two important points should be observed:

- The branch is now backwards not forwards.
This has enabled us to remove the 'jump back for the next byte' instruction.
- The LDA 20,X instruction at 021B could have been redundant if instead of jumping back to 0204 we had gone further to 0202.

- Having written a program it does pay to check for redundant instructions or more efficient ways of implementation.

Before asking you to encode these latest modifications let us refresh your memory on relative branching. The relative branch is specified as a Two's Complement hexadecimal number and hence a sign is incorporated:

- Negative sign - Branch back
- Positive sign - Branch forward.

We need to branch back relative to the current contents of the program counter after the relative branch has been fetched. For our present program this would mean branching back eight places. Now -8 (Decimal) is equal to F8 (HEX) therefore the branch instruction should be BEQ F8.

● EXERCISE 3.4.5

Encode the program: Data Block Move, MKIII
Load EMMA and RUN. Check!!

In our next chapter we will look at a few more programming techniques.

QUESTIONS

1. The contents of which memory location are loaded into the accumulator by the instruction LDA 20, X if the X register = 8?
2. What is the location of the next instruction to be executed if the following branch instruction is taken.

Address

0210 BEQ F3

0212

3. What is meant by:

(0220 - 0224) \Rightarrow 0300 - 0304

4. If the data associated with a branch instruction is negative, is the branch forwards or backwards.
5. What is the hexadecimal equivalent of 10 (decimal)?

OBJECTIVES OF THIS CHAPTER

Having studied this chapter you should:

- Understand the arithmetic instruction available on the 6502 microcomputer.
- Be able to write programs to perform addition or subtraction in binary or decimal modes.
- Be able to write programs to perform double precision addition or subtraction in binary or decimal modes..

INTRODUCTION

In the previous chapter we introduced you to a few programming instructions and most of the addressing modes. We also devised a program to sequentially access a data table. We are now going to extend our knowledge by looking at a few more operations and programming techniques.

ARITHMETIC OPERATIONS

The 6502 has two arithmetic operation instructions - Add with carry, ADC and Subtract with carry, SBC. However, it can perform its arithmetic operations in Binary Coded Decimal (BCD) or in true Binary. The actual method used is determined by the **D-Flag** in the status register:

- D - Set (made equal to '1') then BCD.
- D - Reset (made equal to '0') then True Binary

We must also take account of the **Carry Flag**. When performing single precision arithmetic operations:

- Addition - Reset Carry Flag
- Subtraction - Set Carry Flag

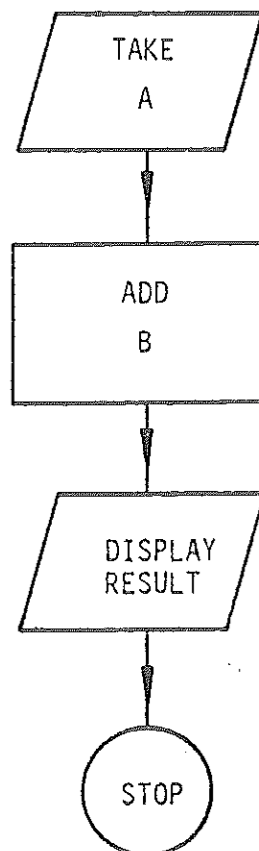
We will discuss the reasons for this later.

SINGLE PRECISION ADDITION

● TASK

Add +9(Base 10) to +15(Base 10) and express result in B.C.D.

● FLOW CHART



● PROGRAM

a) Instruction selection.

The instruction we are looking for is an ADD. The one we have available is the ADD with CARRY (ADC). Since we are performing single precision arithmetic we must CLEAR (reset) the carry flag.

required to SET the decimal mode flag. We must instruct the microprocessor to perform these operations BEFORE the actual arithmetic takes place.

Now:	SED	Set decimal mode
	CLC	Clear carry flag

You will notice that neither of these two operations are suggested by the flow-chart. **We constructed the flow-chart in a general way and hence applicable to any machine.** We could have written it at a different level and made it particular to the 6502 in which case we would have included them.

A second point to notice is the order in which these two instructions have been written - it is wise to set/clear flags required for a particular operation immediately prior to carrying out the operation.

Now let's carry on to do the arithmetic:

LDA	A	LOAD A into Accumulator
ADC	B	ADD B to Accumulator
STA	M	STORE A + B in memory M

We will terminate our program at this stage rather than display result.

JMP	FINISH
-----	--------

We will now allocate memory space to our program and associated data:

Program	- start at location 0200
Data	- A in location 0050
	B in location 0051
	M in location 0052

Now let's transfer all this to a Standard Programming Form.

HEXADECIMAL				SYMBOLIC ASSEMBLER INSTRUCTIONS			
ADDR	1	2	3	LABEL	MNEM	OPERAND	COMMENTS
0200					SED		
					CLC		
					LDA	A	
					ADC	B	
					STA	M	
				FINISH	JMP	FINISH	
0050	A						
0051	B						
0052	M						

You will notice that we have not committed ourselves to any particular addressing mode and that we have made extensive use of labels. In the second stage which follows we select our addressing modes:

b) Addressing Modes

MNEMONIC	COMMENTS	ADDRESS MODE
SED	Only one mode available	IMPLIED
CLC		
LDA	Eight modes available but since data (A) is on zero page and no indexing required.	ZERO PAGE
ADC	As for LDA	ZERO PAGE
STA	Similar comments to LDA	ZERO PAGE
JMP	No requirement for indirect addressing	ABSOLUTE

c) Instruction Code Allocation

We will allocate our codes and enter, for purposes of clarity only, on another programming sheet.

HEXADECIMAL				SYMBOLIC ASSEMBLER INSTRUCTIONS			
ADDR	1	2	3	LABEL	MNEM	OPERAND	COMMENTS
0200	F8				SED		
	18				CLC		
	A5	50			LDA	A	
	65	51			ADC	B	
	85	52			STA	M	
	4C			FINISH	JMP	FINISH	
0050	A						
0051	B						
0052	M						

d) Program Address and Label Allocation

This is the final stage and again for clarity we use a separate program sheet.

HEXADECIMAL				SYMBOLIC ASSEMBLER INSTRUCTIONS			
ADDR	1	2	3	LABEL	MNEM	OPERAND	COMMENTS
0200	F8				SED		
0201	18				CLC		
0202	A5	50			LDA	0050	
0204	65	51			ADC	0051	
0206	85	52			STA	0052	
0208	4C	08	02		JMP	0208	
0050	09						
0051	15						
0052	*						* Result (24) in BCD

We have used three separate programming sheets to complete our program. In practice we would forget the first stage and combine stages b), c) and d).

● EXERCISE 3.5.1.

Enter the program and RUN. Inspect memory location 0052 and CHECK result.

Our task is only partly complete since we set out to DISPLAY the result and not just leave it in a memory location. So, let's complete our **Task**:

The **EMMA** Monitor Program has a part program built into it which will display the contents of the accumulator. Such programs we refer to as **Subroutines**. All we need to do is to specify the start address of appropriate subroutines. These are:

- FE60 - this **converts** the contents of the accumulator into seven segment codes.
- FE0C - this **displays** the converted contents of the accumulator on the two-most right hand seven segment displays on the **EMMA** keyboard/display unit.

We will discuss the finer details of subroutines later; let's just use the facility to complete our task.

Since the subroutines display the contents of the accumulator we can delete the STA 0052 instruction and modify our program accordingly also calling for our subroutines:

HEXADECIMAL				SYMBOLIC ASSEMBLER INSTRUCTIONS			
ADDR	1	2	3	LABEL	MNEM	OPERAND	COMMENTS
0200	F8				SED		
0201	18				CLC		
2	A5	50			LDA	0050	
4	65	51			ADC	0051	
6	20	60	FE		JSR	FE60	Convert subroutine
9	20	0C	FE		JSR	FE0C	Display subroutine
0050	09						
0051	15						

You should notice that we have deleted:

- The STA instruction and its associate memory location.
If we had wished to save the result as well as display it we would have left both these.
- We have deleted the program 'terminate' instruction. We do not need this since the display sub-routine (FE0C) continuously loops back upon itself to 'refresh' the display. To get out of this loop you will have to press the machine **Reset**

● EXERCISE 3.5.2

Experiment by changing the two numbers to be added together. Remember you are in Decimal Mode.

● EXERCISE 3.5.3

Modify the program to clear the decimal mode flag and again experiment with various numbers. Remember the numbers will now be entered and displayed in hexadecimal.

● EXERCISE 3.5.4

What is the maximum result in both modes above that can be displayed without error?

● EXERCISE 3.5.5.

Modify the program to perform single precision subtraction of two numbers.

You should have discovered that relatively small numbers only can be handled in single precision arithmetic. If larger numbers are to be manipulated we use multiple precision, that is, we use two or more bytes to represent our number.

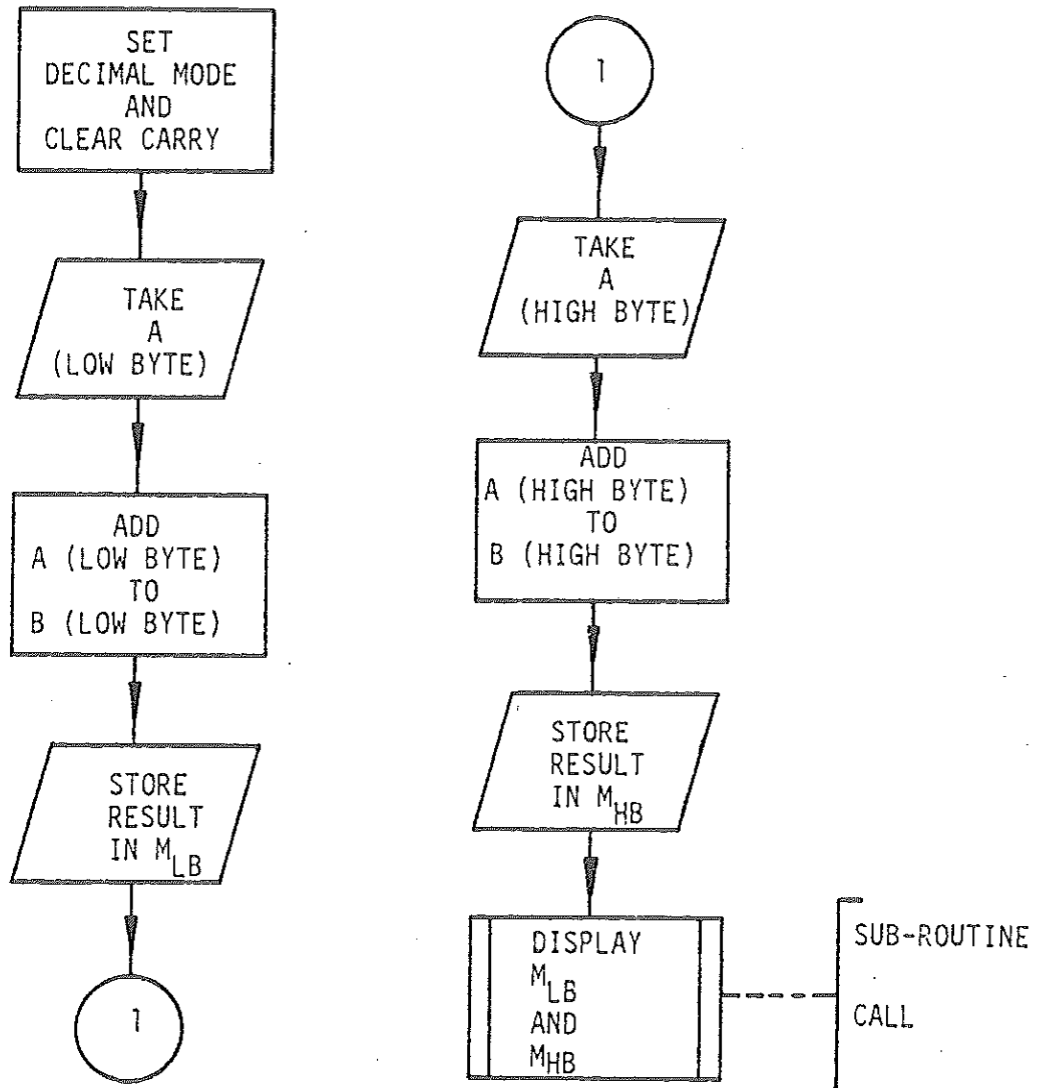
MULTIPLE PRECISION ADDITION

● TASK

Add +782(Base 10) to 324(Base 10) and display result in B.C.D.

FLOW CHART

An identical flow-chart to that drawn for Single Precision Addition could be used. However, we will enlarge on this:



We will now write out the complete program:

HEXADECIMAL				SYMBOLIC ASSEMBLER INSTRUCTIONS			
ADDR	1	2	3	LABEL	MNEM	OPERAND	COMMENTS
0200	F8				SED		
0201	18				CLC		
0202	A5	50			LDA	0050	(ALB)
0204	65	52			ADC	0052	(BLB)
0206	85	54			STA	0054	MLB
0208	A5	51			LDA	0051	(AHB)
020A	65	53			ADC	0053	(BHB)
020C	85	55			STA	0055	MHB
020E	A2	54			LDX#	54	INDEX for result
							Base Address (Low Byte)
0210	20	64	FE		JSR	FE64	Convert subroutine
0213	20	0C	FE		JSR	FE0C	Display subroutine
0050	82						ALB
0051	07		0782				AHB
0052	24						BLB
0053	03		0324				BHB
0054							RESULT LB
0055							RESULT HB

You should notice the following points:

- The addition has been performed in two parts - low bytes followed by high bytes
- The results are displayed from memory locations (0000+X) and (0000+X+1), low byte and high byte respectively.

- The seven segment display subroutine is different to that used in previous exercises which displayed the accumulator contents.
- The program clears the carry for the low byte addition but not for the high byte addition. Obviously any carry that is produced resulting from the low byte addition must be added in when the high byte addition is performed.
- The answer is presented prefaced by a . The seven segment code for . is stored in 0010 and is not cleared by either of the subroutines. You may like to try inserting the two instructions as shown below:

```

STA 0055
LDA# 00
STA 10
LDA# 54

```

} insertion

This modification will delete the . by storing 00 in location 0010.

Alternatively, the code ED will display S (for SUM). Details of the seven segment codes are to be found in Appendix 3.

You will find more details regarding these sub-routines in Appendix 3 and details of the coding required for the seven segment display in the **Emma User Manual**

● EXERCISE 3.5.6

Re-write the program to perform a Double Precision subtraction.

● EXERCISE 3.5.7

What are the largest results that can be obtained without error?

QUESTIONS

Use the programs we have written in this chapter to perform the following calculations.

1. $48(\text{Base } 16)$ $16(\text{Base } 16)$
2. $46(\text{Base } 16)$ $54(\text{Base } 16)$
3. $47(\text{Base } 16)$ $2D(\text{Base } 16)$
4. $99(\text{Base } 10)$ $43(\text{Base } 10)$
5. $3459(\text{Base } 10)$ $1078(\text{Base } 10)$

OBJECTIVES OF THIS CHAPTER

Having studied this chapter you should:

- Understand the logical operations available for the 6502 microprocessor.
- Be able to perform **AND OR** and exclusive **OR** operations on two 8bit binary numbers.

LOGICAL OPERATIONS

Logical operations can be demonstrated by means of a **Truth Table**. A truth table can be defined as a list of all the possible input combinations and the resulting output for each combination. For example, the truth table for a logical AND on two 1-bit inputs would be:

INPUT		OUTPUT
A	B	Z
0	0	0
0	1	0
1	0	0
1	1	1

The 6502 will perform this operation on two-8-bit inputs. It will also perform an **OR-operation** and an **Exclusive OR**. The truth tables are below:

OR:

INPUT		OUTPUT
A	B	Z
0	0	0
0	1	1
1	0	1
1	1	1

EXCLUSIVE OR:

INPUT		OUTPUT
A	B	Z
0	0	0
0	1	1
1	0	1
1	1	0

We will now demonstrate how these logical functions are performed.

LOGICAL AND

One of the most useful functions of the AND instruction is to 'mask out' selected bits of a byte. For example, let's assume that we wish to zero the four most significant bits of the byte 1011 1010 and leave the result in the accumulator. Clearly the result we would expect is 0000 1010.

The AND instruction is symbolised by:

$$A \wedge M \rightarrow A$$

where \wedge means 'logical AND'.

The following program will illustrate:

LDA# 1011 1010	clear M.S. Nibble
AND# 0000 1111	
JSR FE60	Display Subroutine
JSR FE0C	

● EXERCISE 3.6.1

Encode the two binary numbers (into hexadecimal). Use a coding sheet and complete the program for loading into **EMMA**. Note we have used immediate addressing for LDA and AND as signified by #.

LOGICAL OR

The logical OR allows us to set particular bits within a byte without changing any of the others bits.

The OR instruction is symbolised by:

$$A \vee M \rightarrow A$$

where \vee means 'logical OR'.

Suppose we wish to SET bit 7 of the byte 0101 1010 without changing bits 0 --- 6. The following program will illustrate:

LDA# 0101 1010	Set bit 7
ORA# 1000 0000	
JSR FE60	Display sub-routine
JSR FE0C	

● EXERCISE 3.6.2

Encode bytes to be 'OR'ed and complete Program and test.

LOGICAL EXCLUSIVE OR

The logical Exclusive OR can be used to compare bits or to complement bits.

The Exclusive OR is symbolised by:

$$A \vee M \rightarrow A$$

where \vee means 'logical Exclusive OR'.

The following programs will illustrate:

LDA# 1010 0111	Complement
EOR# 1111 1111	
JSR FE60	Display sub-routine
JSR FE0C	
LDA# 1010 0111	Compare
EOR# 1010 0111	
JSR FE60	Display sub-routine
JSR FE0C	

When using the EOR to compare two bytes the Z flag can be tested to decide result. Example

Z flag = 1 for NON Comparison

Z flag = 0 for COMPARISON.

QUESTIONS

Given the following binary numbers

A. = 1111 0000

B. = 0101 1100

C. = 0000 0110

D. = 1111 1111

Perform the following logical operations

1. A B

2. A C

3. D B

4. C B

OBJECTIVES OF THIS CHAPTER

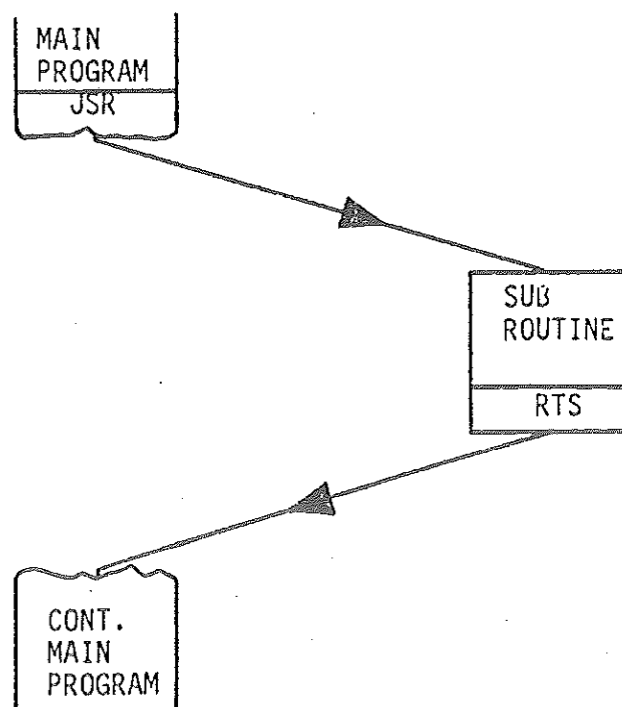
Having studied this chapter you should:

- Understand the use of subroutines as an aid to programming.
- Be aware that it may be necessary to save certain registers before calling a subroutine.

SUB-ROUTINES

We have already used sub-routines to encode a byte into a seven-segment display code and actually illuminate the display. These sub-routines proved rather useful and were built into the **EMMA Monitor**. By 'calling' for these, much program writing time was saved.

A sub-routine is therefore a short program which performs a specific function and one that we may wish to use many times during a program run. Schematically our subroutine can be seen as below.



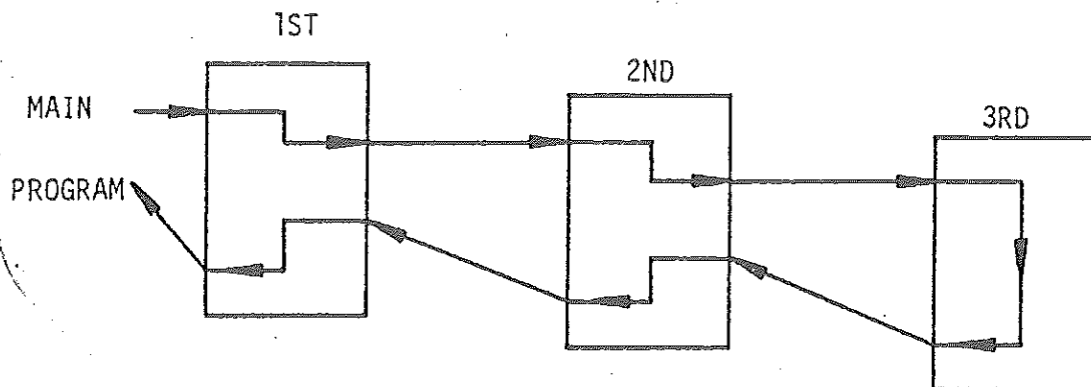
A SUBROUTINE CALL

A 'sub-routine is initiated by the instruction **Jump to Sub-routine (JSR)**: which effectively interrupts the main program and saves the address to which the microprocessor should return after the sub-routine.

To return from the subroutine a **Return from Sub-Routine (RTS)** must be used.

The advantages of sub-routine are that since the same routine can be called numerous times during the execution of a main program a significant saving in memory can be realised. Also the sub-routine can form a part of a library of such routines which a program designer can integrate into his program: thus saving considerably on design time.

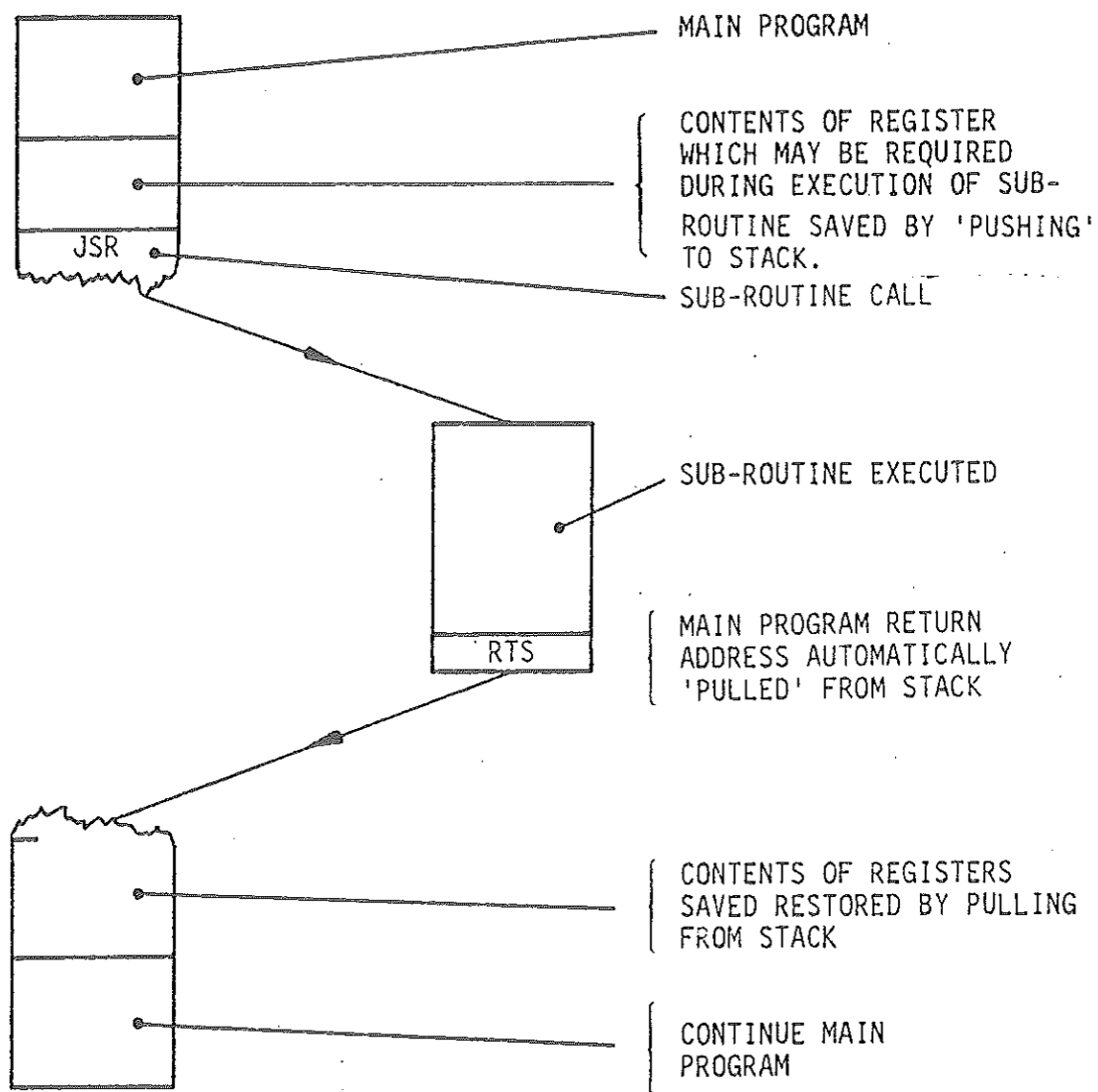
Sub-routine can also be 'called' within Sub-routine; the diagram illustrates:



NESTED SUBROUTINE CALLS

There are precautions that we need to take when implementing Sub-routine calls. For example: the sub-routine being called may use some of the microprocessor special purpose registers such as the Index Registers (X and Y) and alter the Status register. If the program originating the sub-routine call is also using these then their contents must be saved before actually calling the sub-routine. The contents of affected registers could be stored in some memory, but special instructions are available to 'push' the contents of these registers onto the **Stack** and retrieve them by 'pulling' from the stack upon completion of the sub-routine

Schematically this can be illustrated as below:



In the next chapter we will consider the stack in more detail.

QUESTIONS

1. What is the JSR instruction used for?
2. What is the RTS instruction used for?
3. What is the hex code for:
 - (a) JSR
 - (b) RTS

OBJECTIVES OF THIS CHAPTER

Having studied this chapter you should:

- Understand the data storage facility provided by the stack.
- Understand the use of the stack pointer.

STACK PROCESSING

All the programs so far written have used known memory locations. You may also have noticed that page 01 has been avoided although: in fact: it has been used automatically when sub-routines have been called. Why avoid page 01? Well: page 01 is dedicated to what is known as the **Stack** and associated with it is a special purpose register called the **Stack Pointer**.

The facility provided by the stack allows the storage of data in memory where the precise memory location is not known. What will be known: however: is the order in which the data has been stored. This type of programming is termed **re-entrant coding** and is often used in servicing interrupts.

The stack pointer is used by the program to access data placed on the stack. It uses the **push down** stack concept: that is the last data put on the stack is the first taken off. The data is always stored in sequential locations and the stack pointer is always pointing towards the next 'empty' memory location. In recalling data from the stack: the stack pointer is simply incremented repeatedly and automatically.

In the 6502 microprocessor chip as used in **EMMA**, the stack pointer is an 8-bit register with the most significant address byte set at 01. Thus the stack uses page 01, with the stack pointer resetting to FF. The method allows the whole of page 01 to form a stack of 256 locations.

Various instructions use the stack, such as JSR - Jump to Sub-Routine. When a JSR is implemented, the return address in the main program is stored on the stack and the start address of the sub-routine written into the program control counter, for example:

Program Control Counter	Memory Contents
0300	JSR Op. Code
0301	20 } Sub-Routine
0302	00 } at 0020
0303	XX
Stack Pointer	Stack Contents
01FF	03 } Return address
01FE	02 } 0302
01FD	XX

The JSR instruction takes 6 machine cycles. During the return from sub-routine instruction, the program counter has restored to it 0302 and is then incremented to 0303 before the return from sub-routine is completed.

Other instructions which involve the stack allow the programmer to push the accumulator (or Status Register) on to, or pull from, the stack, or transfer data between X-Register and stack or stack and X-Register.

The instructions allow temporary storage of data before servicing a jump to sub-routine or interrupt service routine.

Note: the status register is automatically saved on the stack when servicing an interrupt.(a point covered in the next section on interrupts) but not when implementing a Jump to Sub-routine.

● EXERCISE 3.8.1

Write a program which will load the accumulator with 80(HEX): load X-register with 60(HEX) transfer X-Reg to Stack Pointer and then 'push' the accumulator and 'transfer' the X-register to the Stack. Terminate your program. Reset and examine the contents of 0160 and 015F. Are they what you would expect?

● EXERCISE 3.8.2

Extend your program to change the contents of the various registers (for verification purposes) and then restore the original values by pulling the stack. How can you examine these registers to verify that the stack has indeed been 'pulled'?

QUESTIONS

1. How many memory locations are available on the stack?
2. What is the stack pointer?
3. Examine the 6502 manufacturer's instruction set and determine the instructions which:
 - (a) Are used for 'pushing' data on to the stack.
 - (b) Are used for 'pulling' data from the stack.

OBJECTIVES OF THIS CHAPTER

Having studied this chapter you should;

- Understand how to use instructions to "waste time" within the program.
- Be able to write a program for a single or double nested loop time delay.

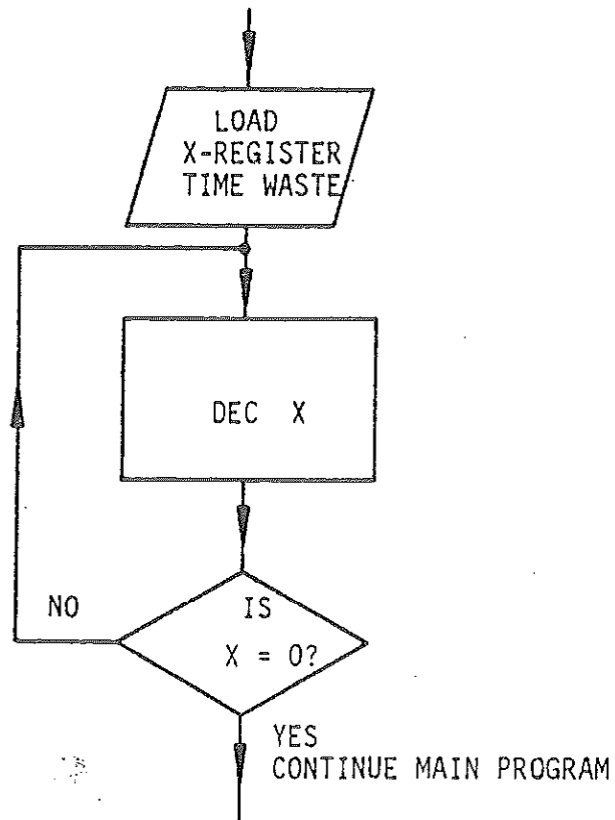
SOFTWARE TIME DELAYS

Time delays can be implemented by means of software (programs) or hardware (integrated circuits). Frequently we require the microprocessor to output signals at precise intervals of time and this will invariably mean that 'time' has to be wasted by the microprocessor. We will explore a technique which we will be using a lot in future chapters, that is, the 'Time Waste Routine'.

Let us consider the flow chart given overleaf:

Time waste routine Flow Chart:

FROM MAIN PROGRAM



The maximum value of time waste that we can obtain with this particular flow-chart is proportional to the maximum value that can be stored in the 8-bit x-register. This value is of course FF(HEX). (Actually 00(HEX) is the maximum value: since the 1st DEC will cause X-Reg. contents to be FF(HEX)).

A suitable program coding would be:

HEXADECIMAL				SYMBOLIC ASSEMBLER INSTRUCTIONS			
ADDR	1	2	3	LABEL	MNEM	OPERAND	COMMENTS
	A2	FF			LDX#	FF	
	CA			LOOP	DEX		
	D0	FD			BNE	LOOP	

We can also determine the exact time that this program will take from the time we start to execute the first instruction to the instant the last instruction is executed. We obtain the individual times for the execution of each instruction from the instruction set. If you care to check you will find these to be:

Instruction	No. of Cycles
LDX#	2
DEX	2
BNE	2 (or 3 if branch occurs to same page and is taken).

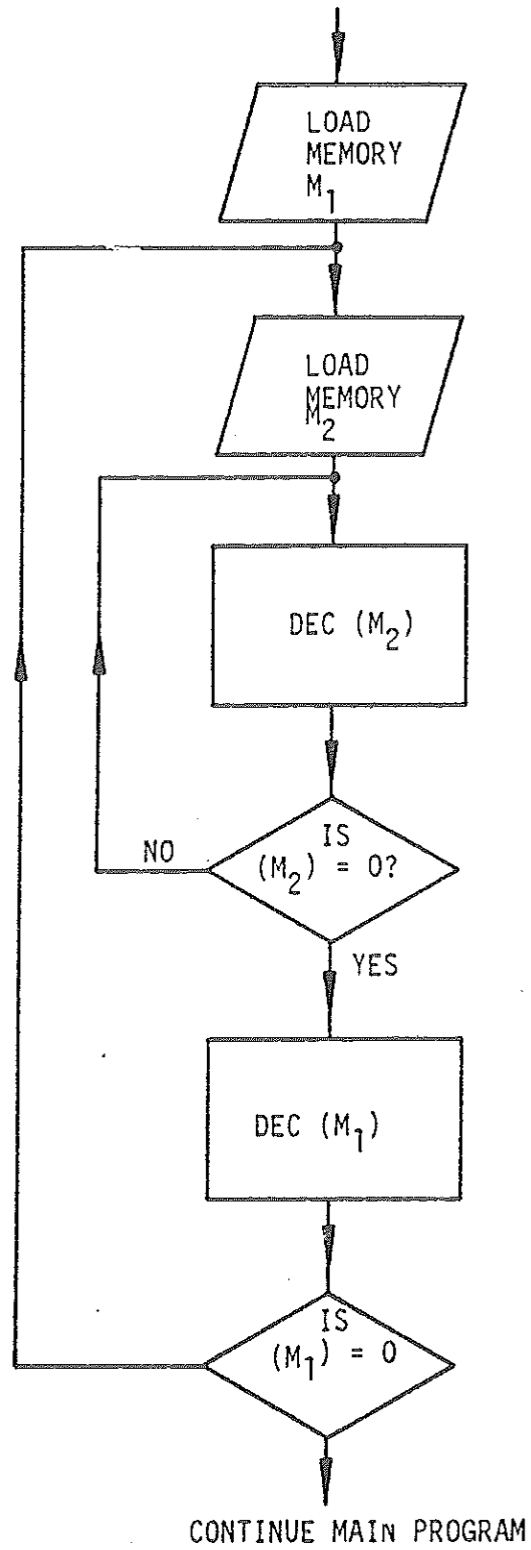
You will notice that we have quoted 'No. of cycles'. These refer to the number of 'clock' cycles taken by the microprocessor. Now: since the **EMMA** uses a 1 MegaHertz (MHz) clock frequency, the cycle or periodic time will be 1/frequency, that is one cycle is equivalent to one micro-second (μS).

We can now compute the time taken:

Instruction		Time (μS)		
LDX#		2		
Program executed when taking loop	DEX	1275	2	Branch taken 255 times
	BNE		3	
	DEX	2	last time through	
	BNE	2		
Total Time Taken		<u>1281</u>	μ Sec	
or 1.281 milli-sec (mS)				

Longer times can be achieved by means of 'nested loops'. The flow chart for a double nested loop time waste is given overleaf:

FROM MAIN PROGRAM



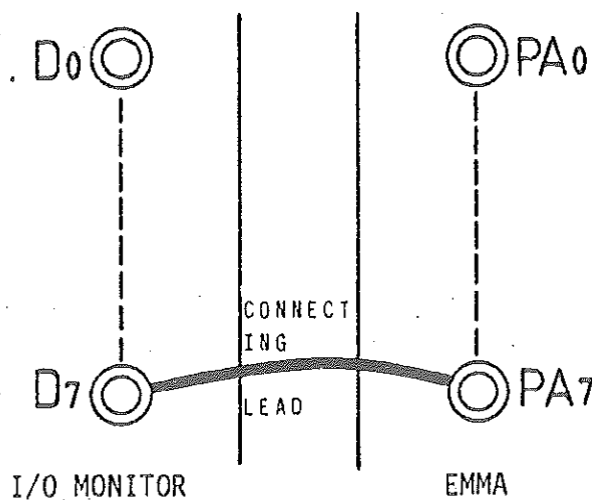
You may notice that instead of using the X-Register we have used memory locations. You will find instructions that will perform these decrement of memory.

● EXERCISE 3.9.1.

- a) Study carefully the flow-chart for the double nested loop and write a suitable program.
- b) Analyse your program and determine the maximum and minimum time delay that your program will permit.

In our next chapter we are going to investigate the I/O Ports in some detail. However, before we do this we will give you a simple routine without explanation that will enable us to demonstrate our time waste routines.

What we are going to do is cause a light emitting diode (LED) to flash at a regular rate and as determined by the time waste. We will need to connect to the I/O port 4-mm terminals marked PA0 - PA7 and down the left-hand side of the **EMMA** microcomputer: the I/O Port Monitor. The connections required are as below:



Apart from making sure you have connected a supply (link across from **ENMA**) to the I/O Port Monitor you need only to make the single connection shown between PA7 (ENMA) and D7 (I/O Port Monitor). You will also have to make sure you have put the I/O Port Monitor READ/WRITE slide switch into the READ position.

Now load the following program and RUN:

HEXADECIMAL				SYMBOLIC ASSEMBLER INSTRUCTIONS			
ADDR	1	2	3	LABEL	MNEM	OPERAND	COMMENTS
0200	A9	FF			LDA#	FF	Port Initialisation and incrementation.
0202	8D	03	09		STA	PA DDR	
0205	A9	00			LDA#	00	
0207	8D	01	09		STA	PA DR	
020A	EE	01	09	NEXT	INC	PA DR	
020D	A2	FF			LDX#	FF	Time waste.
020F	CA			LOOP	DEX		
0210	D0	FD			BNE	LOOP	
0212	4C	0A	02		JMP	NEXT	Repeat Port Inc.

You need not worry about the Initialisation part of the program: we will return to this later. What you should be aware of is that by altering the value placed in location 020E (shown shaded), the rate at which the LED flashes can be changed. Example: Change FF to 50. Does the flashing rate increase?

It is interesting to note that we can re-arrange this program to appear as a Main Program and the time waste as a Sub-Routine. Example:

HEXADECIMAL				SYMBOLIC ASSEMBLER INSTRUCTIONS			
ADDR	1	2	3	LABEL	MNEM	OPERAND	COMMENTS
0020	A2	FF			LDX#	FF	Time waste
0022	CA			LOOP	DEX		
0023	D0	FD			BNE	LOOP	Sub-Routine
0025	60				RTS		
0200	A9	FF			LDA#	FF	Main Program
0202	8D	03	09		STA	PA DDR	
0205	A9	00			LDA#	00	
0207	8D	01	09		STA	PA DR	
020A	EE	01	09		INC	PA DR	
020D	20	20	00		JSR	0020	
0210	4C	0A	02		JMP	020A	

You will notice that the Time Delay Sub-Routine has been put onto zero page with the Main Program still on page 02.

● EXERCISE 3.9.2.

Assuming you did program a double nested time waste routine; use this as a sub-routine to replace the time waste in the program above.

You may have already noticed that the time delays obtained at PA7 are considerably more than those calculated. If you move the connection from PA7 to PA6 you will find that the frequency has increased, in fact it has doubled. The time delays calculated were for the periodic time at PA0. We will come back to this in Chapter 3.11 when we will be making accurate measurements using the I/O Ports and an oscilloscope.

QUESTIONS

1. How many machine cycles are required to execute the instruction LDA absolute?
2. Calculate the time taken to execute the following program:

```
LDA# 65  
STA 20  
DEC 20  
BNE FC
```


OBJECTIVES OF THIS CHAPTER

Having studied this chapter you should:

- Understand the four different conditions which can interrupt the microprocessor.
- Understand the sequence of events which occurs when the processor is interrupted by an interrupt request ($\overline{\text{IRQ}}$) or a non-maskable interrupt ($\overline{\text{NMI}}$)
- Know where to store $\overline{\text{IRQ}}$ or $\overline{\text{NMI}}$ vectors.
- Understand the function of interrupt service routines.

INTERRUPTS

Interrupts is a concept which allows the program currently being run on the microprocessor to be interrupted (it's execution temporarily suspended); so that a more important task can be attended to.

There are four possible conditions which may interrupt the microprocessor: these are considered in turn.

INTERRUPT REQUEST ($\overline{\text{IRQ}}$)

This is a level sensitive input to the 6502. When a logical '0' is sensed at the $\overline{\text{IRQ}}$ pin (the bar — means enabled low), the processor will complete it's current operation. It will then 'read' the Interrupt Disable Flag (flag 1 in status register) and, if clear, will implement an interrupt sequence of operations. These are shown overleaf;

- Store Program Counter high byte (PCH) on the stack.
- Store Program counter low byte (PCL) on the stack.
- Store status register contents on the stack
- Load PCL from address FFFE.
- Load PCH from address FFFF.
- Set Interrupt Disable Flag (flag 1 in status register).
- Program execution sequences now continue from the memory VECTORS held at FFFF and FFFE.

If the Interrupt Disable Flag is set when the $\overline{\text{IRQ}}$ line goes low, the interrupt will be ignored.

If only one device were connected to the $\overline{\text{IRQ}}$ line it would be serviced by what is known as an 'interrupt service routine'. This routine would effectively be the final sequence of instructions above, namely "Program execution sequences now continue from the memory VECTOR held at FFFF and FFFE". However, it is more likely that a number of devices would be connected to the same $\overline{\text{IRQ}}$ line, each capable of bringing it low. A software routine would then have to determine which device had caused the interrupt before it could execute an appropriate set of instructions to service that particular device. Various methods are available not least of which is termed **Polling**

This is a technique whereby all devices connected to the $\overline{\text{IRQ}}$ line are 'polled' or interrogated to determine whether they are asking for service. Since two or more devices may be requesting an interrupt, the polling may be performed to some order of priority

Once an interrupt service routine is being executed it is possible for the programmer to allow a further interrupt to take place since the interrupt disable flag is under his control. In this way a number of interrupts may be in a state of being serviced at any one time.

At the end of an interrupt service routine the instruction, Return From Interrupt (RTI) must be used.

A further point to note is that if any other registers such as the accumulator, X or Y hold data at the instant of interrupt which needs to be remembered, these must be pushed to the stack at the start of the routine and pulled from the stack prior to RTI.

One last point, it is important that before executing the RTI, the **Device Interrupt Flag** which pulled the $\overline{\text{IRQ}}$ low is set since RTI will have the effect of clearing the Interrupt Disable Flag when the status register is restored, e.g., the flag must have been clear to allow the interrupt in the first place. If this action is not taken then a series of interrupts will be attempted although the device has, in fact, been serviced! An example of this is considered in Chapter 3.11 when using the VIA Timer 1.

NON MASKABLE INTERRUPT (NMI)

This is an edge sensitive input to the 6502. When a logical '1' to logical '0' transition takes place at the $\overline{\text{NMI}}$ pin, the microprocessor will complete it's current operation and set an internal flag such that no matter what state the interrupt disable flag is in, the microprocessor performs the interrupt sequence outlined under $\overline{\text{IRQ}}$. The only exception is that the memory vectors are taken from FFFB and FFFA.

The $\overline{\text{NMI}}$, through the way it has been implemented, has priority over the $\overline{\text{IRQ}}$ at all times.

It is possible to connect more than one device to the $\overline{\text{NMI}}$, but if a subsequent interrupt occurs while servicing the first, it will be ignored. Further, it will not be serviced when the initial service routine is completed. The implications are that multiple interrupt lines connected to the $\overline{\text{NMI}}$ require careful servicing.

BREAK COMMAND (BRK)

The break command is a software interrupt. It is primarily used to cause the microprocessor to go to a halt condition during program debugging but can equally be used in other useful ways.

The interrupt sequence is similar to the hardware interrupt $\overline{\text{IRQ}}$ except that it cannot be masked by the interrupt disable flag. Also when the break command is 'fetched' the Break Command Flag (flag B in status register) is set, this enables the programmer to check whether the interrupt was caused by the software BRK or the hardware $\overline{\text{IRQ}}$. Both use the same memory vectors FFFE and FFFF.

RESET ($\overline{\text{RES}}$)

This is an edge sensitive input to the 6502 when a logical '0' to logical '1' transition takes place at the $\overline{\text{RES}}$ pin, the microprocessor will begin a 'reset' sequence.

It is used to reset or start the microprocessor from a power down condition. With $\overline{\text{RES}}$ held low, read/write operations are inhibited. In the case of EMMA, $\overline{\text{RES}}$ is held high via a 4K7 resistor, it is pulled low when the reset pushbutton is depressed. Depression of the reset pushbutton and then it's release will start the reset sequence. After a system initialisation period of six clock cycles, the interrupt disable flag will be set and the microprocessor will load the program counter with the reset vectors at memory locations FFFC (low byte) and FFFD (high byte).

A software 'reset' can be used to terminate a program by jumping to the reset vectors (FEE0).

We have stated the mechanism for interrupts and where the various interrupt vectors are to be found. The vectors are effectively start addresses for the rest of the interrupt routines which you, the programmer, will decide. Let's take a look at the disassembled listing for the monitor (see Appendix Disassembled Listing). On page APDL/4 you will notice that locations FFFE and FFFF contain B0 and FF respectively. FFB0 are the $\overline{\text{IRQ}}$ vectors. If we now turn to page APDL/3 we will find:

```
FFB0  6C  JMP (FEOE)
```

the $\overline{\text{IRQ}}$ vectors point towards a Jump instruction and the op. code 6C indicates an 'indirect absolute' address mode. You may not be too familiar, as yet, with this mode of addressing, so let's see how it operates.

The 2-byte address code, following the operation code gives access to the low byte data address. Incrementing by 1, gives the high byte data address.

Pictorially:

MEMORY	MEMORY CONTENTS	COMMENTS
FFFE	B0	IRQ VECTORS
FFFF	FF	
FFB0	6C	OP. CODE
FFB1	FE	DATA ADDRESS LOW BYTE
FFB2	0E	DATA ADDRESS HIGH BYTE
OEFE	LB	LOW BYTE Start Address of
OEFF	HB	HIGH BYTE Interrupt Service Routine.
HBLB	OP. CODE	INTERRUPT SERVICE ROUTINE FIRST INSTRUCTION).

Locations OEFE (Low byte) and OEFF (High byte) must therefore be loaded with the start address of the **programmer's interrupt service routine**

● EXERCISE 3.10.1

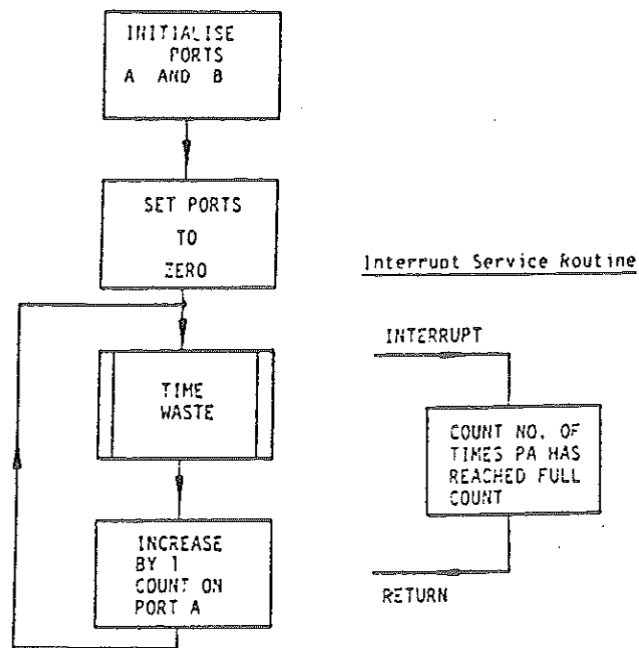
Determine the locations for the start address of a NMI. Hint, upon taking the NMI pin of the microprocessor low, the processor loads the program control counter from addresses FFFA and FFFB, low and high byte respectively.

PROGRAMMING INTERRUPT SERVICE ROUTINES

We will now design a program which will demonstrate the use of interrupt service routines.

Our program will repeatedly increment the VIA User Port A and the number of times a full count is achieved will be indicated at VIA User Port B. The program will include a time waste sub-routine so that Port A can be easily observed on the I/O Port Monitor.

● FLOW-CHART:



● PROGRAM

The program is in four parts:

- Main Program
- Interrupt Service Routine
- Time Waste Sub-routine
- Interrupt Vector Loading

We will look at each part separately:

Main Program

HEXADECIMAL				SYMBOLIC ASSEMBLER INSTRUCTIONS			
ADDR	1	2	3	LABEL	MNEM	OPERAND	COMMENTS
0200	A9	FF			LDA#	FF	Initialise Ports A and B to output.
0202	8D	02	09		STA	DDRB	
0205	8D	03	09		STA	DDRA	
0208	A9	00			LDA#	00	Set DRA and DRB to zero.
020A	8D	00	09		STA	DRB	
020D	8D	01	09		STA	DRA	
0210	EE	01	09	NEXT	INC	DRA	
0213	20	80	02		JSR	0280	Time waste Sub-routine
0216	4C	10	02		JMP	NEXT	Continue count

Interrupt Service Routine

HEXADECIMAL				SYMBOLIC ASSEMBLER INSTRUCTIONS			
ADDR	1	2	3	LABEL	MNEM	OPERAND	COMMENTS
0250	EE	00	09		INC	DRB	Increase by 1 count at Port B.
0253	40				RTI		Continue counting at Port A.

HEXADECIMAL				SYMBOLIC ASSEMBLER INSTRUCTIONS			
ADDR	1	2	3	LABEL	MNEM	OPERAND	COMMENTS
0280	A9	FF			LDA#	FF	
0282	8D	20	00		STA	0020	
0285	A9	FF		LOOP 2	LDA#	FF	
0287	8D	21	00		STA	0021	
028A	CE	21	00	LOOP 1	DEC	0021	
028D	D0	FB			BNE	LOOP 1	
028F	CE	20	00		DEC	0020	
0292	D0	F1			BNE	LOOP 2	
0294	60				RTS		

Interrupt Vector Loadings

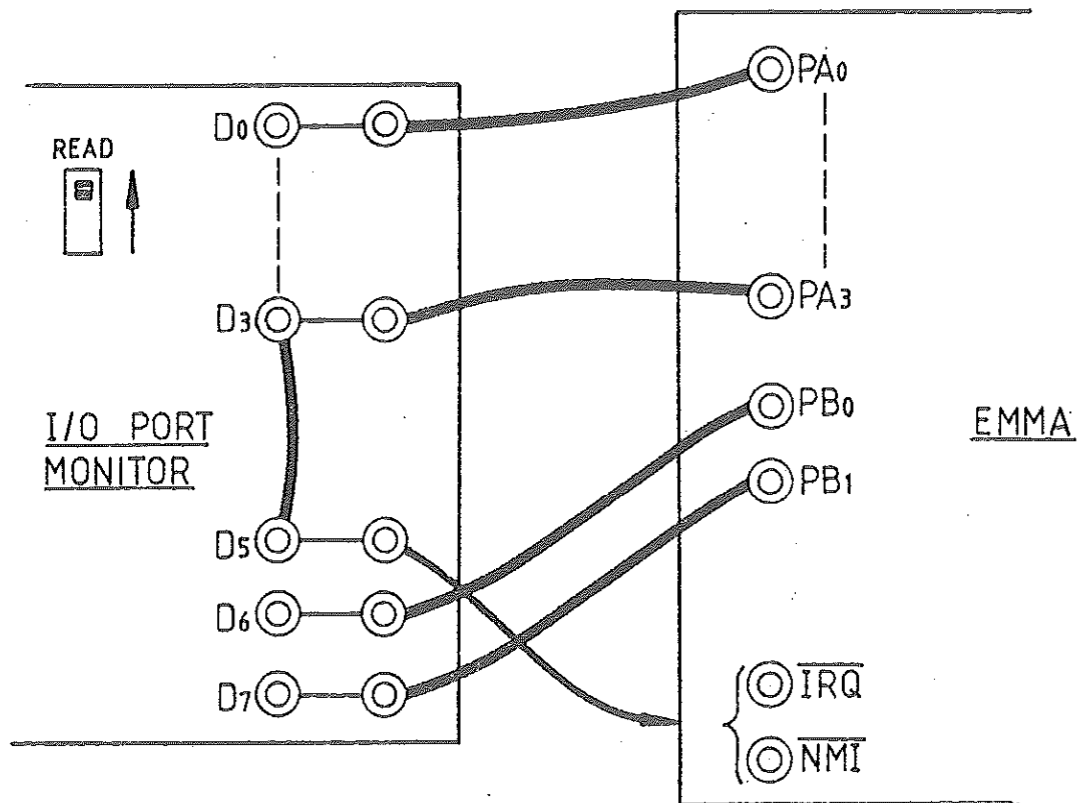
The interrupt vectors (start address of the interrupt service routines) are located at:

0EFC	low byte	NMI Vector
0EFD	high byte	
0EFE	low byte	IRQ Vector
0EFF	high byte	

These locations must be loaded with the start address (interrupt vectors) of the interrupt service routine (0250). Hence:

HEXADECIMAL				SYMBOLIC ASSEMBLER INSTRUCTIONS			
ADDR	1	2	3	LABEL	MNEM	OPERAND	COMMENTS
0EFC	50						$\overline{\text{NMI}}$ VECTORS
0EFD	02						
0EFE	50						$\overline{\text{IRQ}}$ VECTORS
0EFF	02						

We now require a connection diagram:



Don't forget the supply connections to both the I/O Port Monitor and **EMMA**

The arrangement is such that Port A will provide a binary count up to 16 (Denary) which will be indicated on the LED's of the I/O Port Monitor. D5 will also indicate the status of the Interrupt Line and D6 and D7 will indicate the number of interrupts. This connection will firstly be taken to the $\overline{\text{NMI}}$ and secondly to the $\overline{\text{IRQ}}$.

● EXERCISE 3.10.2.

- Connect the I/O Port Monitor to EWMA as shown in the diagram with interrupt connection to NMI.
- Load the whole of the program
- Run program from 0200 and observe the I/O Port Monitor.

We have already stated that the $\overline{\text{NMI}}$ interrupt is enabled low and on the negative edge (transition from high to low). You should observe this at the instant when Port B is incremented immediately following the reset of Port A to zero from a full count of 16 (denary).

We will now consider the Interrupt Request $\overline{\text{IRQ}}$. This signal differs from the $\overline{\text{NMI}}$ in that the $\overline{\text{NMI}}$ is negative edge enabled while the $\overline{\text{IRQ}}$ is level (low) enabled. We will keep to the same basic program except that some modifications will be necessary due to the difference in the interrupt enabling signals. We will present the programs and then discuss the modifications.

Main Program:

HEXADECIMAL				SYMBOLIC ASSEMBLER INSTRUCTIONS			
ADDR	1	2	3	LABEL	MNEM	OPERAND	COMMENTS
0200	A9	FF			LDA#	FF	Initialises Ports
0202	8D	02	09		STA	DDRB	A and B to
0205	8D	03	09		STA	DDRA	output.
0208	A9	00			LDA#	00	Resets Port B
020A	8D	00	09		STA	DRB	to zero.
020D	A9	10			LDA#	10	Sets Port A
020F	8D	01	09		STA	DRA	bit 4 to 'high'
0212	58				CLI	58	Enables $\overline{\text{IRQ}}$ flag
0213	EE	01	09	NEXT	INC	DRA	
0216	20	80	02		JSR	0280	Timewaste S.R.
021	4C	13	02		JMP	NEXT	

Interrupt Service Routine:

HEXADECIMAL				SYMBOLIC ASSEMBLER INSTRUCTIONS			
ADDR	1	2	3	LABEL	MNEM	OPERAND	COMMENTS
0250	A9	10			LDA#	10	Resets
0252	0D	01	09		ORA	DRA	Port A
0255	8D	01	09		STA	DRA	bit 4.
0258	EE	00	09		INC	DRB	
025B	40				RTI		

The Time Waste Sub-Routine and the Interrupt Vectors remain unchanged.

The program will require that the IRQ signal is obtained from Port A, bit 4 rather than bit 3 as in previous exercise.

● EXERCISE 3.10.3.

Change the interconnections between the I/O Port Monitor and EMMA, enter the new program (not forgetting the Time Waste and Interrupt Vectors if the machine has been switched off) and run the program.

You should observe that the effect is exactly the same as before.

Now let's consider the modifications to the program.

- If Port A is set originally to zero, an interrupt will be enabled immediately the Interrupt Flag in the microprocessor Status Register is cleared. Obviously this must be disallowed since we have not yet started counting! Bit 4 (connected to interrupt IRQ) is thus set high by storing 10 to DRA.
- When Port A is incremented it will now start counting at 10 (Hex) rather than 00 (Hex). When F (HEX) is reached (this will represent a denary count of 16 on Bits 0 - 3 at Port A), one further increment should break to interrupt (to increment Port B) and reset Bits 0 - 3 at Port A to zero. The actual output at Port A will go from 1F to 20, so causing the required interrupt (bit 4 goes low).
- Before leaving the interrupt routine we must set Bit 4 high otherwise upon return we will immediately break to interrupt and continue to do so without further counts at Port A taking place. The setting of bit 4 has been done by loading the accumulator with 10, performing a logical OR on the accumulator with the actual output of Port A, and storing the result back to Port A.

● EXAMPLE:

0001 0000 Port A being incremented
0001 1111

 next increment 16 \longrightarrow 0 (Denary)
0010 0000 on bits 0 - 3.
 Bit 4 goes low causing interrupt.

0010 0000 OR operation on Port A output
0001 0000 with 10 (HEX) sets Bit 4

0011 0000 Result of OR operation
 prevents recurring interrupts

● EXERCISE 3.10.4

Draw a neat timing diagram for each of the Bits PA0-7, PBO-1 and the IRQ signal. Label the break to interrupt point.

● EXERCISE 3.10.5.

Using the previous program for IRQ, explore the results of the following modifications separately:

- i) change contents of location 020E to 00
- ii) change contents of location 0212 to 78
- iii) replace interrupt service routine with that used during the NMI exercise.

Explain, in detail, why the changes gave the results they did.

QUESTIONS

1. What is 'Polling'.
2. Where are the \overline{IRQ} vectors stored?
3. Where are the \overline{NMI} vectors stored?
4. Which has the higher priority \overline{NMI} or \overline{IRQ} ?

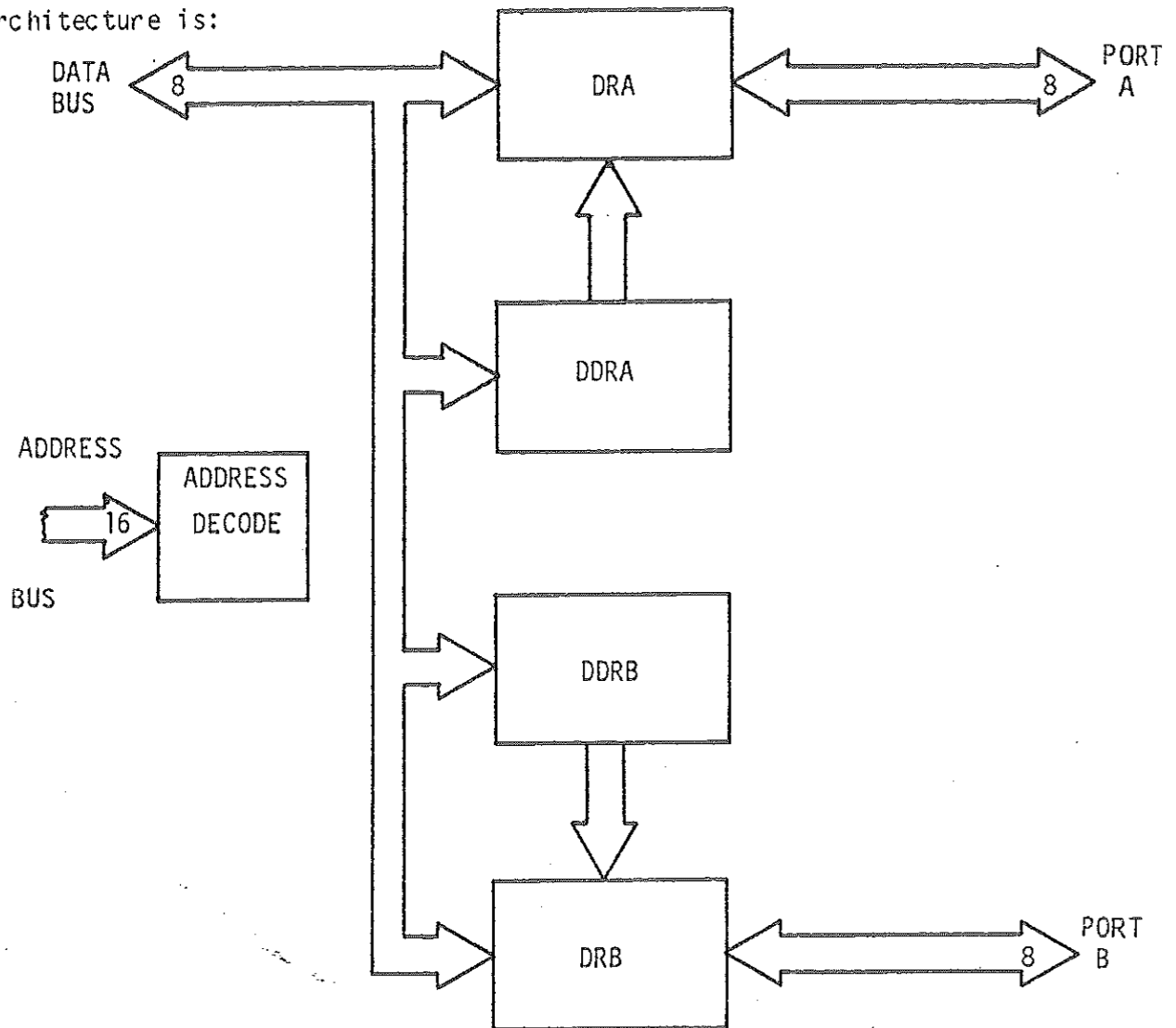
OBJECTIVES OF THIS CHAPTER

- Having studied this chapter you should:
- Understand the I/O port facilities offered by the VIA.
- Be able to program individual bits of the two 8bit ports to input or output as required.
- Understand the operation of the two 16bit timers incorporated in the VIA.
- Be able to program the timers to operate in any of the available modes.

INTRODUCTION

The I/O ports are contained in chip 6522 on the main **EMMA** board and is referred to as a Versatile Interface Adapter VIA. As its name implies, it is versatile in as far as it is capable of numerous functions. It is also an interface because it provides a means of connecting **EMMA** to the outside world i.e. it is physically connected between the 6502 (microprocessor chip) and external devices such as applications hardware modules.

The component parts of the I/O port which we will initially consider are the two ports designated Port A and Port B. Schematically the appropriate architecture is:



The VIA is far more comprehensive than depicted above but we will consider this subsequently.

The blocks in the diagram are fully addressable and are identifiable as below:

LABEL	DESIGNATION	ADDRESS
DRB	Port B: data Register	0900
DRA	Port A: data Register	0901
DDRB	Port B: data direction Register	0902
DDRA	Port A: data direction Register	0903

Each of these registers can be separately addressed and simply appear to the microprocessor as a memory location.

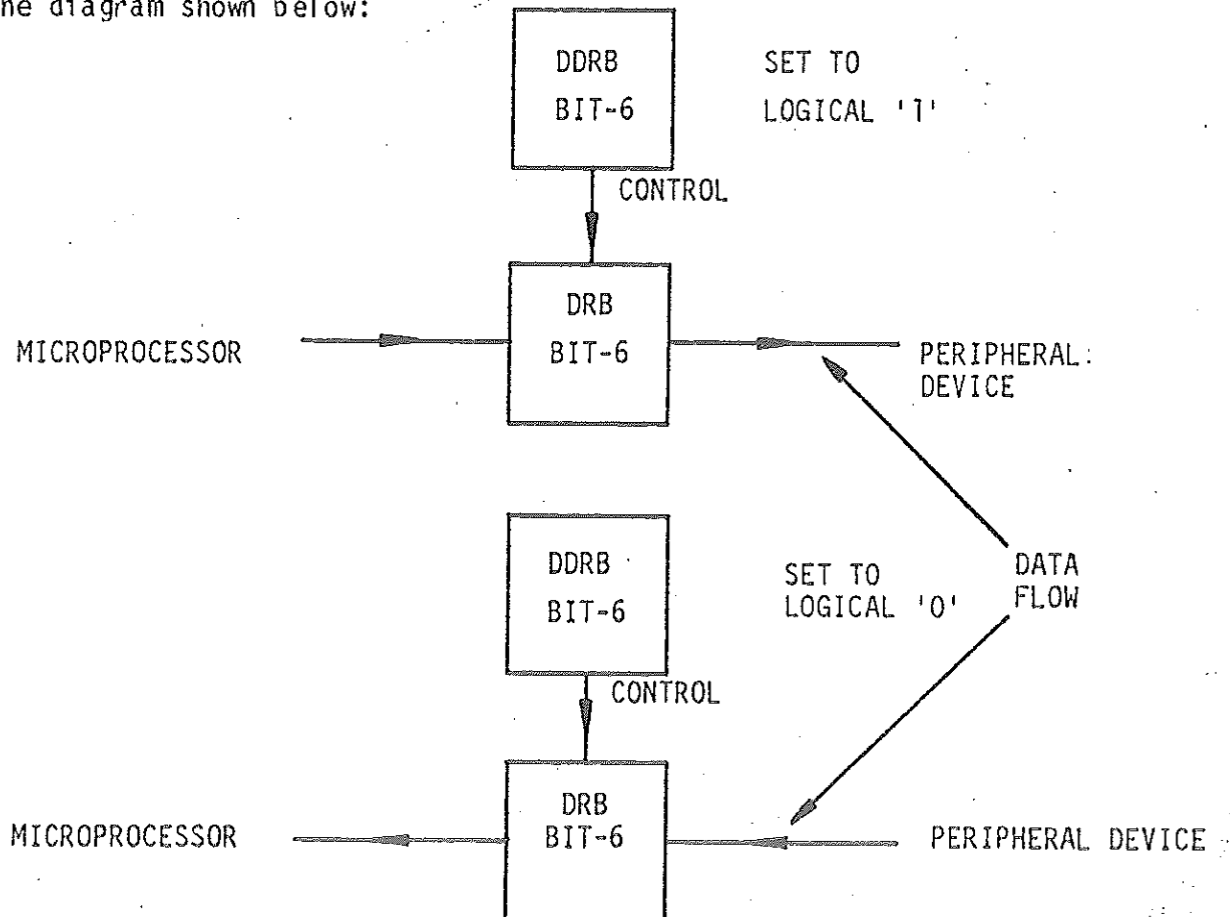
The data registers hold the data which is being transferred from the microprocessor to some outside peripheral device or from some outside peripheral device to the microprocessor; that is to say they are bi-directional. Needless to say data cannot pass through them in both directions simultaneously. They have to be set to operate in the required direction. An important feature of this particular VIA is that each of the eight bits of both ports can be directionally set independently. It is the function of the data direction registers to accomplish this.

The setting up of the ports is termed **initialisation** and must be done by the programmer before the port is used.

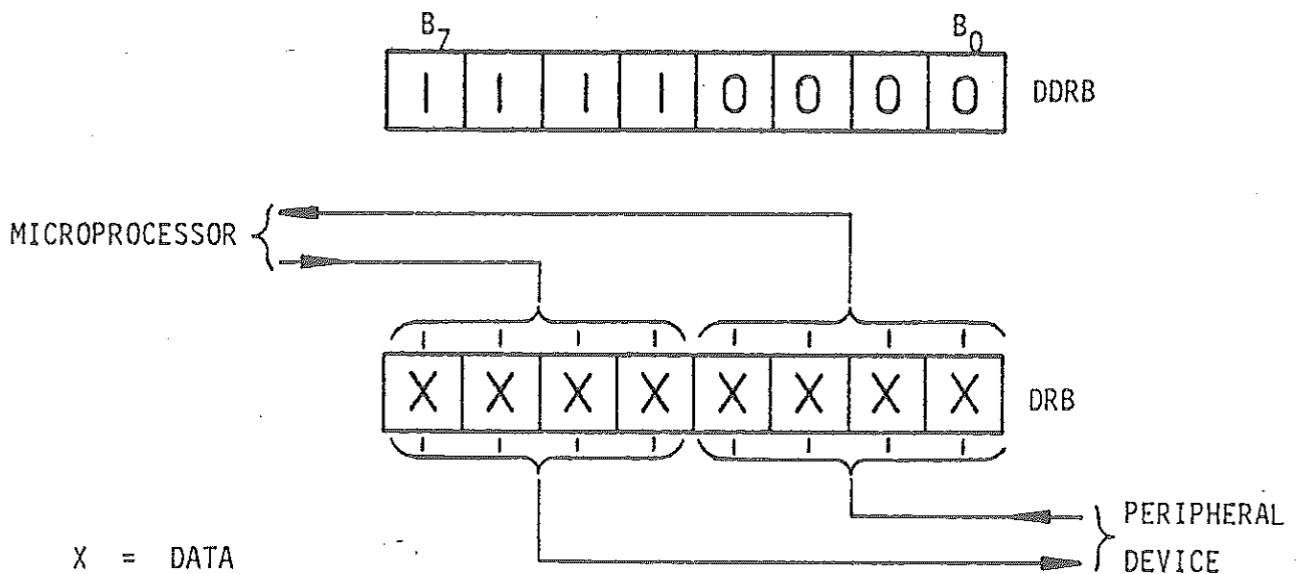
PORT INITIALISATION

The data direction registers are both eight bit registers with each bit being associated with a corresponding bit in the data register. For example: if bit 6 of data direction register Port B is set to logical one; then bit 6 of data register Port B will be set to 'output' while a logical zero will set it to 'input'.

The diagram shown below:

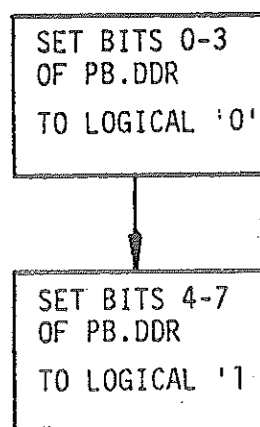


Considering all 8-bits: if we load Port B: data direction register with 1111 0000 then the Port B: data register will be set for inputs and outputs as shown below:



Obviously this is not the only combination that can be configured. It should be appreciated at this stage that these registers are cleared to logic '0' on RESET. This places all peripheral interface lines to the input state. They are also 'pulled-up' to logic '1'. This is a safety feature since most devices are 'enabled low'.

We will now write the program which will initialise Port B so that bits B0-B3 are configured as input and bits B4-B7 as output.



We can perform our setting of the data direction register simply by loading the DDRB with F0 and using the instructions:

```
LDA#    F0
STA     DDRB
```

where:

F sets bits B4 - B7 to 1's
0 clears bits B0 - B3 to 0's

Simply DDRB is the label for the address of Port B - Data Direction Register. In hexadecimal notation this is 0900 (see table on page 89).

● EXERCISE 3.11.1

Write I/O port initialisation routines to effect the following:

- a) PB bits 0 - 2 output, 3 - 7 input.
- b) PA bits 1, 2, 3 and 5 as output, rest as input.

Define any address labels used.

USING THE DATA REGISTERS

Once the ports have been initialised the data registers can be used. It is possible to 'latch' data being input but for our immediate purposes we will consider the Data Registers as being 'transparent', that is, any data appearing at the register in the correct direction (as determined by the DDR) will pass through it.

The control of any system configuration is the responsibility of the microprocessor and its program. The data registers therefore appear to the microprocessor as memory locations, data can be 'stored' to them or 'loaded' from them. The following examples will explain.

● EXAMPLE

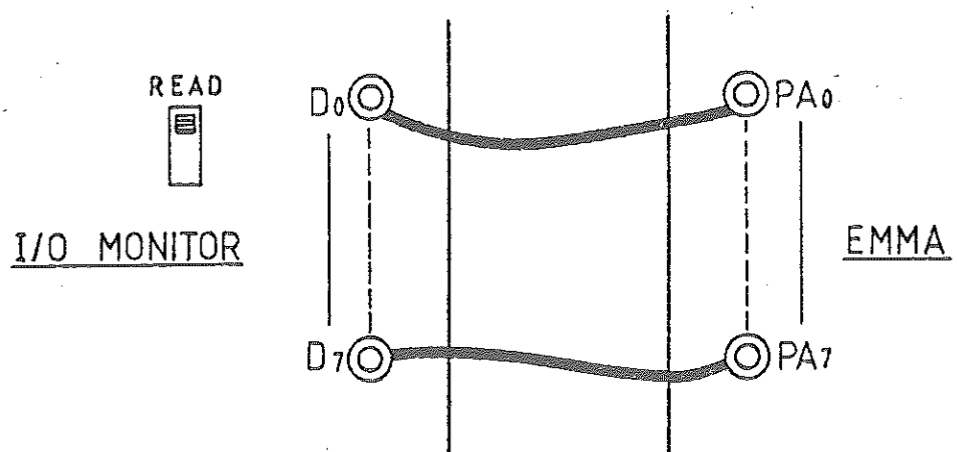
Transfer 8-bits from memory location 0020 via DRA to a peripheral device connected to Port A: bits 0 - 7.


	LDA#	FF	Initialises Port A
	STA	DDRA	ALL bits to output
	LDA	0020	Effects data
	STA	DRA	transfer
LABELS:	DDRA	0903	
	DRA	0901	


● EXERCISE 3.11.2

- Use a standard programming form and assemble the program above which transfers data from memory 0020 to Port A.
- Connect the I/O port monitor to Port A: and switch monitor to READ.

Connection Diagram:



Note: D0  means 'connect' ALL Ports D0, D1, D2 etc., up to and including D7

D7 

- c) Load your program into **EMMA**. Also load memory 0020 with any known data other than FF (can you see why?).
- d) RUN program and observe results

● EXERCISE 3.11.3

Write a program which **READS** data from a peripheral device and stores in memory 0020. Keep connection of I/O port monitor to Port A and switch its mode selector switch to **WRITE**.

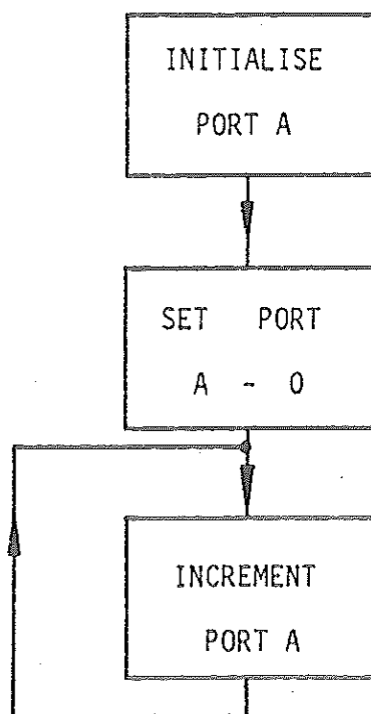
WAVEFORM GENERATION

The generation of waveforms is an important aspect of engineering system design. Frequently, waveshapes of various forms are required such as square wave, triangular wave, sawtooth and ramps generally. The microprocessor is eminently suitable for producing such.

We will now consider producing a square wave at each of the bits of one of the output ports. We can easily accomplish this by simply incrementing repetitively the output port.

FLOW CHART

Flow chart:



The program is:

The program is:

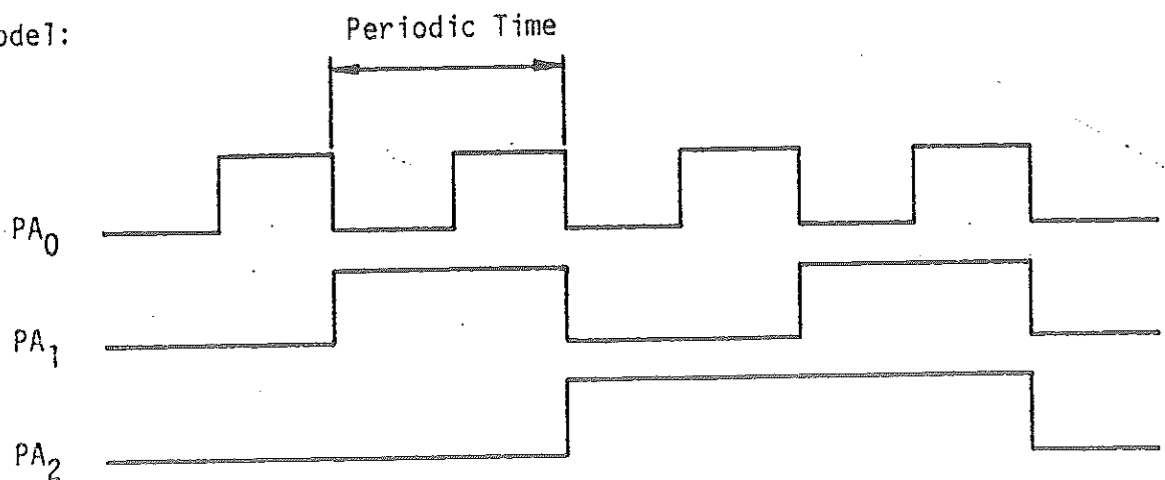
HEXADECIMAL				SYMBOLIC ASSEMBLER INSTRUCTIONS			
ADDR	1	2	3	LABEL	MNEM	OPERAND	COMMENTS
0200	A9	FF			LDA#	FF	Initialises Port A
0202	8D	03	09		STA	DDRA	to output
0205	A9	00			LDA#	00	Sets Port A
0207	8D	01	09		STA	DRA	to zero
020A	EE	01	09	REPT	INC	DRA	Repeatedly
020D	4C	0A	02		JMP	REPT	Increments Port A

You may now need to familiarise yourself with the oscilloscope before proceeding further. If this is so refer to the Appendix 5 (oscilloscopes).

● EXERCISE 3.11.4.

- Enter the program on EMMA to repeatedly increment the output port and observe each bit on the oscilloscope.
- Using the model below: record the waveform of all the bits relative to each other and clearly marking the periodic time for each.

Model:



CALCULATION OF PERIOD TIME

You should have observed that the periodic time doubles from one bit to the next; more significant one. You should also be in a position to actually predict or work out accurately the periodic times and hence frequencies obtained at each bit.

For example:

- Periodic Time (T) = $\frac{1}{f}$

where f = frequency in Hertz

= time for one complete cycle in seconds.

- The time can be predicted by calculating the number of machine cycles from the start of one increment through to the beginning of the next increment and multiplying by 2.

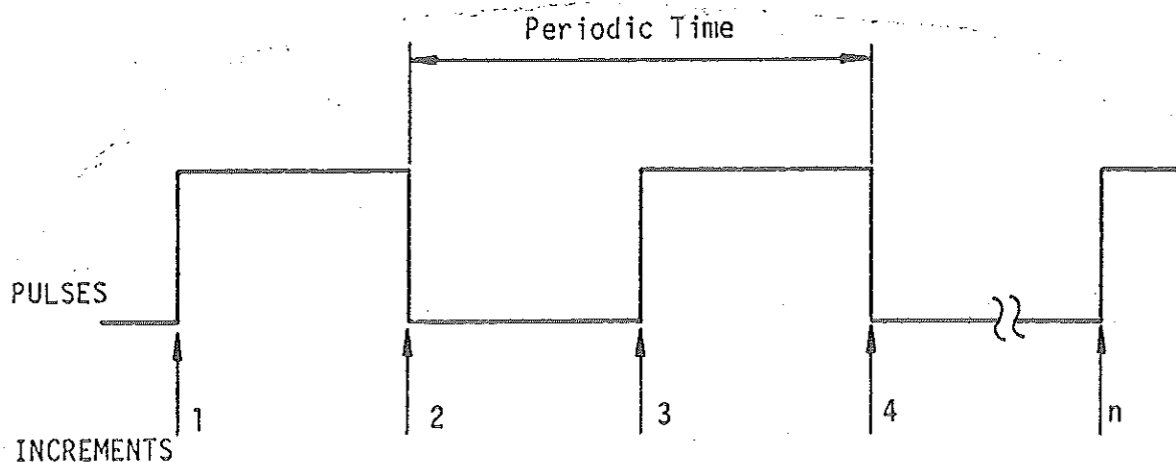
- EXAMPLE:

Program

INC (Absolute) 6 machine cycles

JMP (Absolute) 3 machine cycles.

Total machine number of machine cycles is 9 with 1 s/machine cycle.
Therefore there are 9 s between increments at Port A bit 0. Now considering the waveform generated at bit 0.



It should be observed that the periodic time (T) is twice the interval of time elapsing between two successive increments.

For bit₀: periodic time is therefore:

$$\begin{aligned} 2 \times 9 \mu s &= 18 \mu s \\ \text{and frequency} &= \frac{1}{18 \mu s} = 5.556 \times 10^4 \text{ Hertz} \end{aligned}$$

This is the highest 'software' frequency that we can generate. We can however generate lower frequencies by inserting time wastes between increments. In fact this is exactly what we did in Chapter 3.9. We can also obtain accurate frequencies by the use of the Instruction: **NO Operation (NOP)** as a means of fine 'adjustment' to a time waste routine.

● EXERCISE 3.11.5.

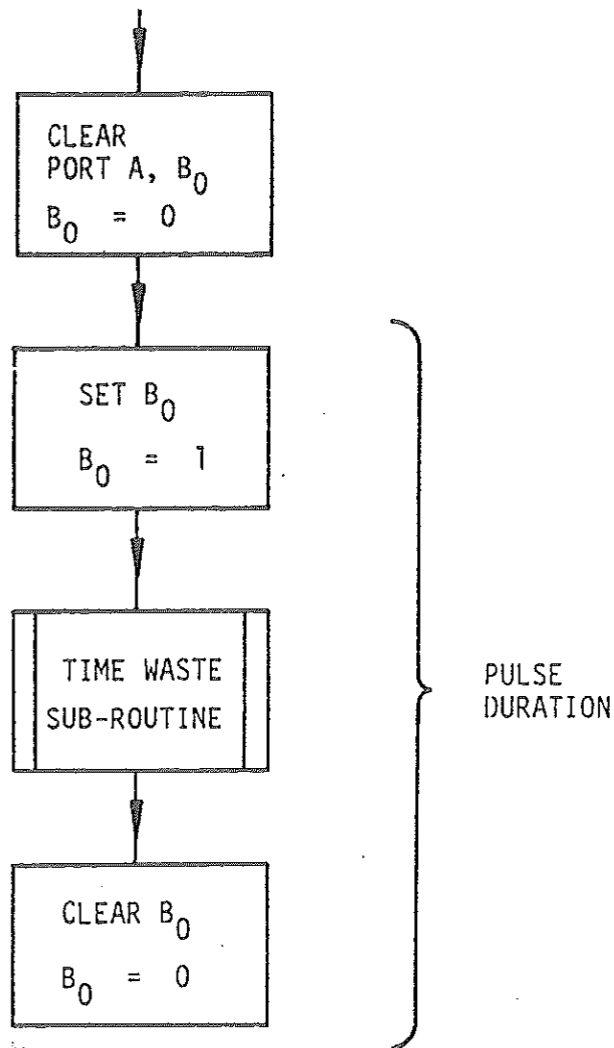
Use the NOP instruction only to modify the square wave generation program to produce a frequency of 33.334×10^3 Hertz.

● EXERCISE 3.11.6.

Design a program which will produce an output squarewave at Port A bit 0 of 1 KHz exactly.

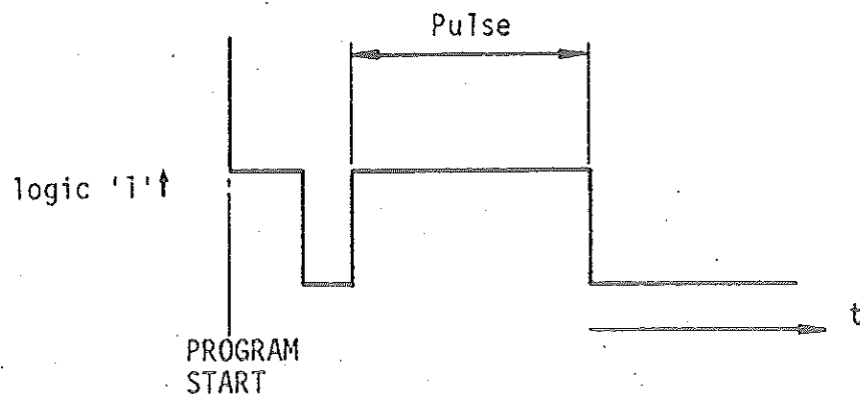
PULSE GENERATION

Frequently we require to generate a 'pulse' of known duration: as for example when we use the Analogue to Digital Converter (A/D). We can easily effect this simply by setting a particular bit on one of the ports and then timing an interval before clearing the set bit. The flow chart overleaf will effect this technique:



The flow chart assumes that we may be using the I/O port for any other data transfer at the time we need to 'pulse' bit 0. If this is not the case we can simply INCREMENT the port to SET B0 and then either DECREMENT or INCREMENT again to clear B0.

The effect of a program which has been written to this flow-chart will be to produce a pulse as below:



● EXERCISE 3.11.7

- a) Redraw the flow-chart such that the pulse occurs approximately 0.5 seconds after the start of the program execution and that the pulse lasts for a period of 2 milli-seconds approximately.
- b) Write the program to effect a) above and test using the I/O Port Monitor.

● EXERCISE 3.11.8

Modify the program so that the pulse is repeated every 0.5 seconds.

● EXERCISE 3.11.9

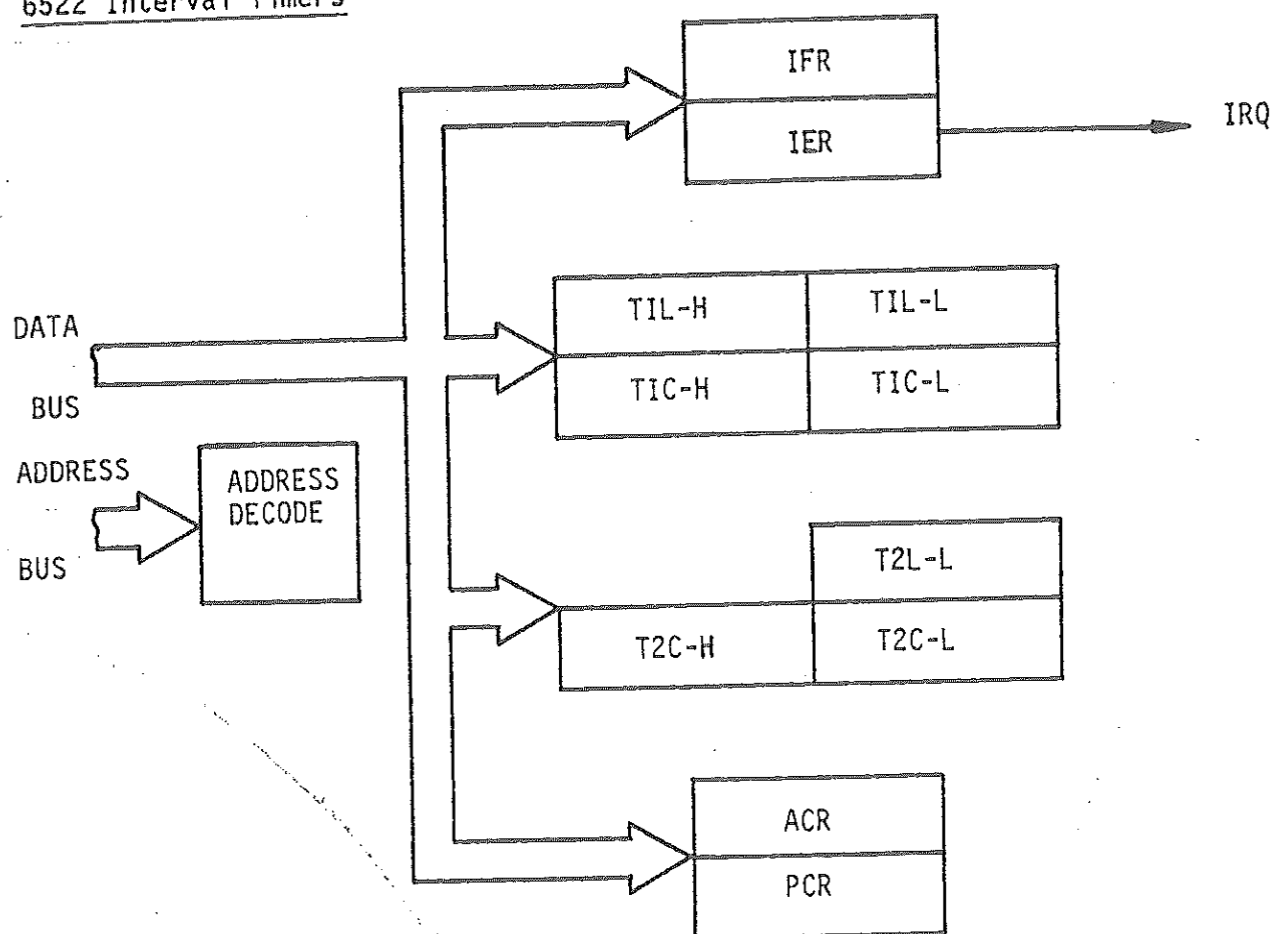
Change experimentally the time waste values to give a mark-space ratio of approximately 0.5:1 at a repetitive pulse rate of 2000 Hz approximately.

HARDWARE TIMERS

So far, time delays have been produced by software routines. This is not particularly an efficient way to use the microprocessor since during a software generated time waste the processor is unable to perform any other work. In control systems many such time wastes are required and hence the processor could be very inefficiently used. A better solution to this time waste/efficiency problem is to use hardware timers. These can be triggered off by the microprocessor and configured to interrupt the processor upon time out. In the meanwhile the microprocessor can be performing other useful functions.

The Versatile Interface Adapter (6522) has two timers which can be programmed to count out predetermined periods. They can be programmed to interrupt the microprocessor upon count out or the microprocessor can be programmed to read the timer at intervals and take appropriate action when the timer has timed out.

Simplified Architecture
6522 Interval Timers



Each of the blocks in the diagram are fully addressable and are identifiable as shown overleaf.

LABEL	DESIGNATION	ADDRESS
TIC-L	Timer 1 low-order latch (WRITE) Timer 1 low-order Counter (READ)	0904
TIC-H	Timer 1 High Order Counter (WRITE)	0905
TIL-L	Timer 1 low-order latch (WRITE)	0906
TIL-H	Timer 1 High-order latch (WRITE)	0907
T2C-L	Timer 2 low-order latch (WRITE) Timer 2 low-order Counter (READ)	0908
T2C-H	Timer 2 High-Order Counter (WRITE)	0909
ACR	Auxiliary Control Register	090B
IFR	Interrupt Flag Register	090D
IER	Interrupt Enable Register	090E
PCR	Peripheral Control Register	090C

You should observe that there is a slight difference between the two timers.

We will now look at each of the registers: considering timer 1 only.

AUXILIARY CONTROL REGISTER (ACR)

Two bits (bits 6 and 7) of the Auxiliary Control Register determine the four operating modes of Timer 1. Each mode will affect both the Interrupt flag Register (bit 6 in particular) and bit 7 of output Port B. The four modes are outlined in the table below and are selected when the appropriate code is written into bits 6 and 7 of ACR.

MODE	ACR 7	Bit No. 6	OPERATION	PORT B BIT 7
1	0	0	One-shot Mode. A single interrupt occurs upon time-out of Timer 1.	DISABLED
2	0	1	Free Run Mode. Upon time-out the counter is automatically reloaded and new time-out period begins. An interrupt occurs upon each time-out.	DISABLED
3	1	0	As for Mode 1	Goes low for duration of timed period.
4	1	1	As for Mode 2	Output inverted upon each time-out producing a square wave of equal mark/space ratio. Unless counter is re-loaded from latches creating a new time period.

INTERRUPT FLAG REGISTER (IFR)

Bit 6 of the Interrupt Flag Register is set upon time-out of Timer 1. This bit is cleared by either reading Timer 1: low-order Counter (TIC-L); by writing Timer 1: high-order counter (TIC-H) or by writing a '1' directly to the flag.

INTERRUPT ENABLE REGISTER (IER)

Bit 6 of the Interrupt Enable Register corresponds to bit 6 of the Interrupt Flag Register. A '1' in this bit will enable the interrupt while a '0' will disable. However: bits in this register are under program control as follows:

With bit 7 at '0': a '1' in bit 6 will CLEAR interrupt enable: while a '0' will leave unaffected.

With bit 7 at '1': a '1' in bit 6 will SET interrupt enable: while a '0' will leave unaffected.

Note: both IFR and IER are 8-bit registers and provide flags for other modes of operation of the VIA as well as for Timer 1.

LATCHES/COUNTERS

Two 8-bit latches designated low-order and high-order respectively are provided for Timer 1. Associated with these are two 8-bit counters: also designated low-order and high-order respectively. The latches are used to store data which is to be loaded into the counter. After loading: the counter is decremented at phase 2 (ϕ_2) clock rate (1 micro second). Upon reaching zero: an interrupt flag (bit 6 of IFR) is set and the IRQ line will go low if the interrupt is enabled (bit 6 of IER set to '1'). Further interrupts will be disabled by the timer unless programmed to automatically transfer the contents of the latches into the counter and begin to decrement again.

Now let's design a few programs which will enable us to apply these hardware timers.

PROGRAMMING HARDWARE TIMER 1

We will write a simple program which: with minor modifications: will demonstrate each of the modes of operation of the timers.

The program is effectively in four parts:

- Main Program
- Interrupt Service Routine
- Time Waste Sub-Routine
- Interrupt Vector loadings

a) Main Program

The main program is designed to initialise the microprocessor interrupt flags; the VIA interrupt flags; the VIA mode of operation and finally to set timer 1 time interval.

The program is:

HEXADECIMAL				SYMBOLIC ASSEMBLER INSTRUCTIONS			
ADDR	1	2	3	LABEL	MNEM	OPERAND	COMMENTS
0200	58				CLI		Enables Processor \overline{IRQ}
0201	A9	C0			LDA#	C0	} Enables VIA IRQ
0203	8D	0E	09		STA	IER	
0206	A9	*			LDA#	*	} Sets Mode of Timer 1 Operation
0208	8D	0B	09		STA	ACR	
020B	A9	FF			LDA#	FF	} Sets Timer 1 time interval
020D	8D	04	09		STA	TIC-L	
0210	A9	FF			LDA#	FF	
0212	8D	05	09		STA	TIC-H	
0215	4C	15	02	WAIT	JMP	WAIT	Program End

* Mode of operation Codes. (Enter C0 for the present).

Code	Operation	PB7
00	Single Shot	Disabled
40	Free Run	Disabled
80	Single Shot	Enabled
C0	Free Run	Enabled

b) Interrupt Service Routine

The interrupt service routine is designed to initialise Port A bits 0 - 6 to output; increment this port to a full count on bits 0 - 6; reset the VIA interrupt flag and then return from interrupt.

The program is:

HEXADECIMAL				SYMBOLIC ASSEMBLER INSTRUCTIONS			
ADDR	1	2	3	LABEL	MNEM	OPERAND	COMMENTS
0250	A9	FF			LDA#	FF	Initialises Port A:
0252	8D	03	09		STA	DDRA	bits 0-7 to output.
0255	A9	00			LDA#	00	Sets Port A
0257	8D	01	09		STA	DRA	to Zero.
025A	EE	01	09	REP	INC	DRA	
025D	20	80	02		JSR	0280	Time Waste S.R.
0260	A9	7F			LDA#	7F	Tests DRA for
0262	CD	01	09		CMP	DRA	full count.
0265	D0	F3			BNE	REP	
0267	A9	40			LDA#	40	Clears VIA
0269	8D	0D	09		STA	IFR	Timer 1 \overline{IRQ} flag.
026C	40				RTI		

c) Time Waste Sub-Routine

The time waste sub-routine is simply included to allow the I/O Port Monitor to be used to monitor Port A. bits 0 - 6. It is a straight forward double-loop routine which you will be familiar with.

The program is:

HEXADECIMAL				SYMBOLIC ASSEMBLER INSTRUCTIONS			
ADDR	1	2	3	LABEL	MNEM	OPERAND	COMMENTS
0280	A9	FF			LDA#	FF	
0282	85	20			STA	20	
0284	A9	FF		LOOP 2	LDA#	FF	
0286	85	21			STA	21	
0288	C6	21		LOOP 1	DEC	21	
028A	D0	FC			BNE	LOOP 1	
028C	C6	20			DEC	20	
028E	D0	F4			BNE	LOOP 2	
0290	60				RIS		

d) Interrupt Vector Loadings

The VIA Timer 1 causes an Interrupt request (\overline{IRQ}) at the VIA and hence the microprocessor itself (the VIA \overline{IRQ} output pin is connected on the printed circuit board. to the microprocessor \overline{IRQ} input pin). The microprocessor \overline{IRQ} breaks to memory locations FFFE and FFFF and hence to OEFE and OEFF (see section on Interrupts Chapter 3.10). Memory locations OEFE and OEFF must therefore be loaded with the interrupt vector's for the interrupt service routine. In this instance, the start address, and hence the vectors of the interrupt service routine are 0250. These must be loaded before the program is run.

Now let's enter these four components of our TIMER 1 program. But first let's decide on our mode of operation: (see * on Main Program).

- Mode 1 (Code 00)
The program will operate to increment Port A bits 0-6 to a single full count only.
- Mode 2 (Code 40)
The program will operate to increment Port A bits 0-6; reset Port A to zero and continue to increment. The microprocessor will continue this operation repeatedly.
- Mode 3 (Code 80)
The program will operate as in Mode 1 above, but bit 7 of Port B will also be enabled. PB7 will be pulsed low for the duration of the Timer 1 time interval (approx. 0.6 second).
- Mode 4 (Code C0)
The program will operate as in Mode 2 above but bit 7 of Port B will also be enabled. 7 will generate a continuous square wave.

● EXERCISE 3.11.10

Load into EMMA the four components of the program and run the program for each of the mode of operation codes. Observe results.

● EXERCISE 3.11.11

With the Timer 1 program entered and with mode 4 selected, change the main program such that the microprocessor \overline{IRQ} flag is disabled (change location 0200 from CLI. op. code. to SEI. op. code 78). Observe results. Note that a similar result can be obtained by disabling the interrupt at the VIA (load IER with 80)

Run the program in mode 1 but enter No Operations (op. code EA) at memory locations 0267-026B inclusive. This will show the importance of clearing a **Device Interrupt flag** before returning from interrupt. Although Timer 1 is in a single-shot mode, the VIA interrupt request line IRQ will only be cleared (taken high) if the counters are reloaded. Since mode 1 does not do this, the Timer 1 interrupt flag (bit 6 in IFR), must be cleared. If this is not done, the microprocessor will continue to be interrupted upon return from each interrupt although the timer itself is not producing interrupts.

The Timer 2 is slightly different from Timer 1. You will find technical details of this timer in the **EMMA TECHNICAL MANUAL**.

There are also other features belonging to the VIA which we have not yet mentioned. These cover important aspects concerning the control of peripheral devices. We will deal with these via a Technical Specification in the 'TECHNICAL MANUAL' associated with **EMMA**.

QUESTIONS

1. What is the address of Port B data register?
2. If we store 6F at location 0903, which bits of Port A are set to input.
3. Which do IER and IFR stand for?
4. What is the address of Timer 2 highorder counter (T2CH)?

OBJECTIVES OF THIS CHAPTER

Having studied this chapter you should:

- Be able to use the single-step facility to help identify program faults'.
- Be able to use the break-point facility to interrupt the microprocessor and enable the internal registers to be examined at a point in the program where a fault is suspected'.

INTRODUCTION

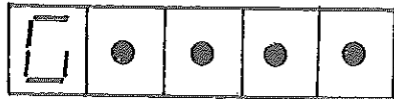
We have already suggested that even the best programmers' programs do not always operate the first time. If the program does not operate when RUN it is more likely to be your program than the hardware that is at fault! When this happens, as surely it will, you will appreciate the care (or lack of), that you took in firstly constructing your flow-chart and secondly providing adequate comments on your assembly listing. Assuming you have met these documentation requirements then you may proceed to use one or other of the debug features provided by the EMMA monitor'.

SINGLE STEP MODE

This causes the microprocessor to be interrupted after each instruction has been executed, enabling the user to inspect various internal registers. The contents of these registers indicate the result of the last operation performed by the microprocessor.

Once loaded the single step switch may be put to 'ON' and the following sequence performed:

- Press RESET
- Press **R** key.

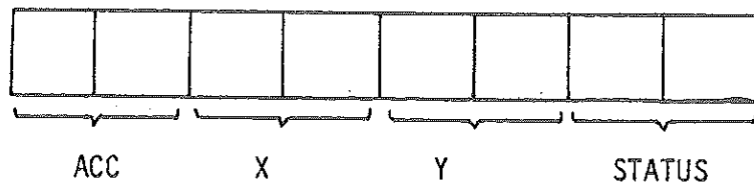


- Set START address
- Press **R** key. This will cause the first instruction to be executed.

The display will now show the contents of the:

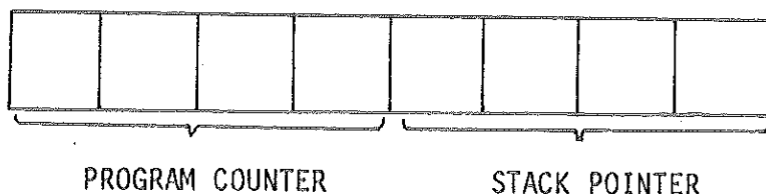
ACCUMULATOR, X-REGISTER, Y-REGISTER and the STATUS REGISTER.

Two hexadecimal digits will be devoted to each and displayed on the EMMA Keyboard/Display as follows:



- Press the **R** key.

The display will now show the contents of the **Program Counter** and the **Stack Pointer**. Four hexadecimal digits being devoted to each. The program counter gives the address of the next instruction to be executed.



If you now press **R** again the next instruction will be executed and the display will again show the current contents of ACC, X, Y and STATUS. Press **R** again to give P.C. and S.P.

Repeatedly pressing **R** will thus step the program through instruction by instruction, allowing you to inspect each of the processor registers after every instruction.

To return to the normal program RUN mode, the single step switch must be return to OFF. Press the **G** key twice and the program will run normally from its start address.

The single step mode can be a time consuming process especially if the program is large. The following procedure allows the program to be 'inspected' at set points in the program.

BREAK POINT MODE

Assuming the user suspects that problems are occurring in a particular part of his program.

A break point will interrupt the microprocessor, enabling him to inspect the state of the internal registers at any point within the program.

Now assume that we wish to insert a **Break Command** at a point in a program where it will jump to a sub-routine: e.g.

PROGRAM	00EA	XXXX
	00EC	20 20 00
	00EF	XXXX

We wish to insert a break command at location 00EC. Now the break command code is 00 and it is this that must replace 20 at 00EC.

Now:

- Press **P** Key
The display will read: P . * * * *
- Set P . 0 0 E C using hexadecimal keys.
- Press **P**. The display now shows that 00 (Break command code) has been loaded into location 00EC e.g. Display.
P 0 0 E C . 0 0

Note that pressing **P** again (twice) restores the data 20 (jump to sub-routine code) and pressing **P** again (twice) re-introduces the break command.

- With display showing P . 0 0 E C 0 0, press G and set display to:
h . * * * * where
* * * * is the start address of your program.

PRESS **G**.

The program will now RUN to address 00EC and break to the monitor sub-routine used for the single step mode. The display will now show:

Acc, X, Y and STATUS.

- Press **P**.
The display will now show:
PROGRAM COUNTER and STACK POINTER.
- Press **P** twice.
The break command is removed and the jump to sub-routine code is restored. The display indicates:
P . 00EC 1 2 0
- Press **R**.
The program continues to RUN from location 00EC.

OBJECTIVES OF THIS CHAPTER

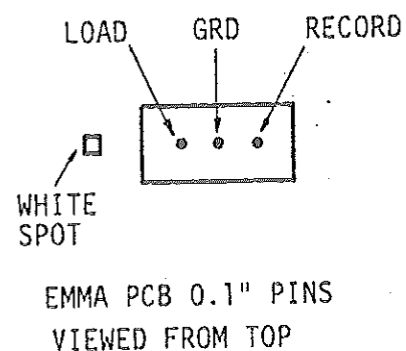
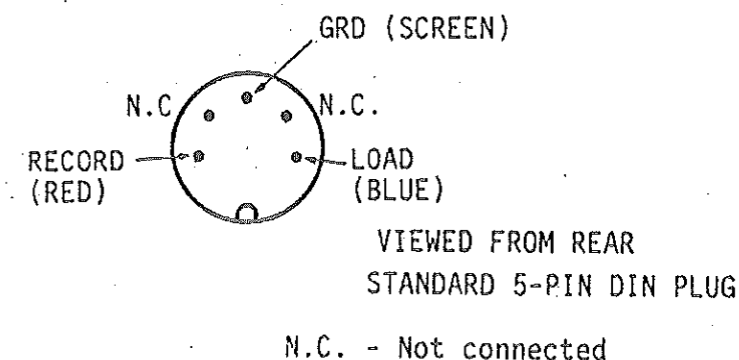
Having studied this chapter you should:

- Be able to save a program stored in memory, on cassette tape.
- Be able to load a program from a cassette into the microcomputer memory.

INTRODUCTION

A cassette interface is provided on the 'Emma' which allows the use of a normal commercial cassette recorder for the storage of programs. It is recommended that good quality tapes are used and that the tape head is kept clean with the occasional use of a head cleaning tape. Short duration tapes will be less likely to stretch with repeated use and for this reason C120 cassette tapes are not recommended.

Connection to the recorder is best facilitated by the hi-fi DIN connector. If the recorder does not have a DIN facility, then connection can be made via external microphone and ear-phone jack sockets. The cassette connector on the 'Emma' is a 3-way p.c.b. plug towards the bottom right of the cassette interface block. Connections are:



Note: no damage will be done if connections to input and output are initially reversed in trying the 3-way p.c.b. socket both ways round before obtaining correct operation.

PROGRAM SAVE ON CASSETTE

- Press save key **S**.
- Enter start address of program to be loaded.
- Press **S** again.
- Enter last program address +1. (this must include all program data. it will not matter if this address is: in fact, greater than last address +1).
- Press **S** again the display will ask for baud rate.
- Switch the recorder to RECORD.
- Select baud rate KEY 1 = 1200 Baud; KEY 0 = 300 Baud.
- A lead in tone is recorded followed by data.
- The display goes blank, until the finish address reappears, indicating that the program is loaded.
- Wait 3 to 4 seconds. During this time a 2.4 kHz tone is recorded.
- Press STOP.

Return to the start of the recording. Playing through the recording the 2.4kHz tone should be interspersed with a 1.2 kHz tone during the program store. It will be easier to find programs on tape if a recorder with a count facility is used.

PROGRAM LOAD FROM CASSETTE

The EMMA II interface is tolerant of the amplitude of the cassette output and it is generally acceptable to have the volume and tone controls set to approximately mid-position.

- RESET the microprocessor.
- Press **L** (Load Key).
- The display will ask for baud rate.
- Find the program lead-in tone: either with tape counter, or by listening for the 2.4 kHz tone.
- During the lead-in tone, select baud rate 0 = 300; 1 = 1200.
- When the program has loaded the row of dots will reappear on the display.
- Stop the recorder and run the program in the normal way.

SECTION 4

APPLICATION HARDWARE

Chapter 4.1	Using the Application Hardware	Page 1
Chapter 4.2	Further Application Program	Page 16

OBJECTIVES OF THIS CHAPTER

Having studied this chapter you should:

Understand the operation the applications hardware which form part of the Microprocessor Applications System MA02.

Be able to use these modules in conjunction with the Emma microcomputer.

INTRODUCTION

Emma is a microprocessor most suited to control applications, that is, the driving of output devices. The **Emma** microcomputer system range comprises numerous output devices which we term Application Hardware. You have already used some of these devices, for example, the I/O Port Monitor is such a device.

Specifications and circuit diagrams of the modules used are included in Appendix 2 (Application Modules).

In the following programs where a time waste routine is required we will use a standard time-waste sub-routine to be loaded at the top of page 03 and calling for programmer entered time wastes from locations 0020, 21 and 22. However, for convenience, there are instances where we will relocate or modify this.

TIME WASTE ROUTINE (STANDARD)

03D0	A5,20	LDA	20
	85,23	STA	23
03D4	A5,21	LDA	21
	85,24	STA	24
03D8	A5,22	LDA	22
	85,25	STA	25
03DC	C6,25	DEC	25
	D0,FC	BNE	03DC (FC)
	C6,24	DEC	24
	D0,F4	BNE	03D8 (F4)
	C6,23	DEC	23
	D0,EC	BNE	03D4 (EC)
03E8	60	RTS	

TIME WASTE (MEMORIES)

0020	} Programmer Set	◀ COARSE
21		◀ INTERMEDIATE
22		◀ FINE
23	} Decrementd	
24		
25		
	} Memories	

THE INPUT/OUTPUT (I/O) PORT MONITOR

The I/O Port Monitor allows the user to read from or write to, an 8-bit input/output port. You have already used the module to monitor the output ports of the 6522 VIA.

Each light emitting diode (LED) is fully buffered to permit operation from the logic levels output at the VIA.

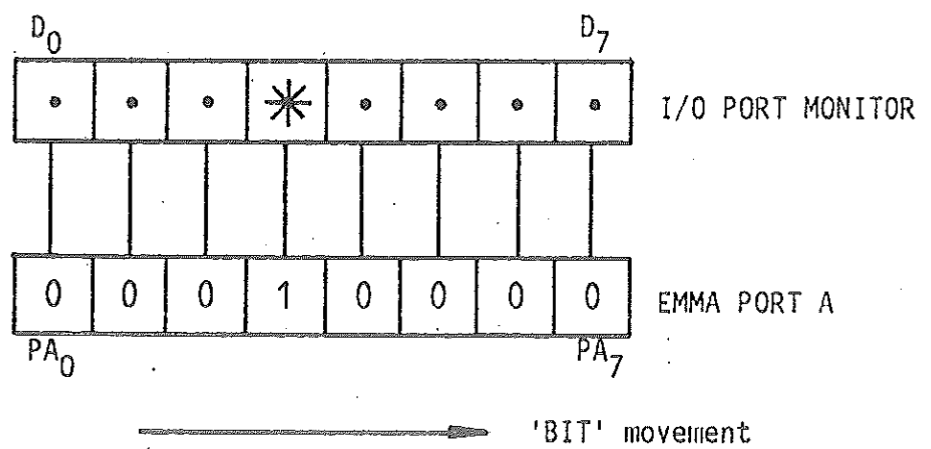
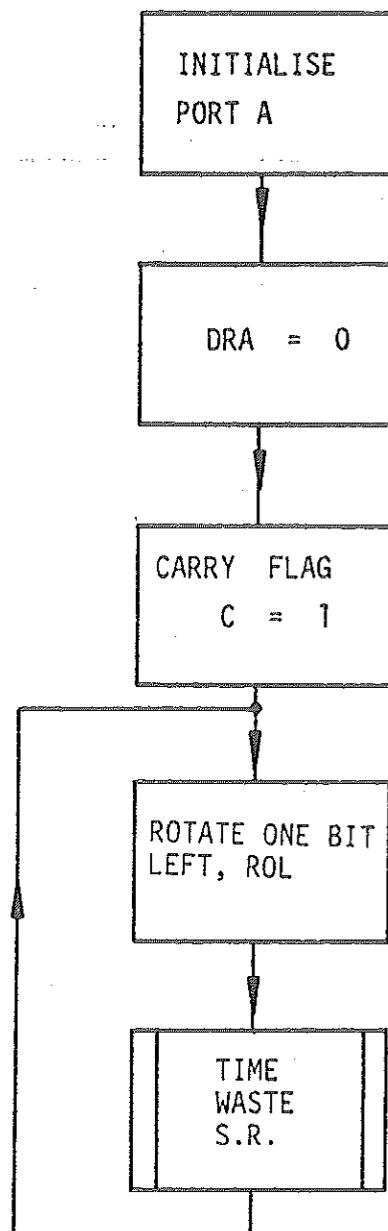
The monitor has a read/write selector switch. In the READ position the LED's indicate the logic levels present on the output port. In the WRITE position, the code set up on the switches opposite each LED drive socket (D_0-D_7) will be available for connection to input a port. The LED's will also indicate the code set.

PROGRAMMING THE I/O PORT MONITOR

The I/O Port has already been used by you in demonstrating various modes of operation of the microprocessor and the VIA. We think you will agree that you already have a fair understanding of the module - what you really need is some more programming experience. The following exercise should provide this:

● EXERCISE 4.1.1

Write a program such that the LED's of the I/O Port Monitor will be illuminated one LED at a time and so that they appear to move continuously from left to right. Example: first LED D_0 , followed by LED D_1 , D_2 , $D_3 \rightarrow D_7$ and back to D_0 continuously. You can approach the problem by using a data table, one entry for each 'pattern'. However, we suggest you consider the flow-chart given overleaf and use the instruction set to find suitable instructions to implement it.



THE SWITCH PAD MODULE

We will come back to the I/O Port Monitor when we have considered the Switch Pad Module.

The Switch Pad Module allows the user to key data to an I/O Port enabled to input and is simply 8 push-to-make switches, each switch being in series with a 470Ω register.

The inputs can be taken to voltage levels as required, depending on whether the outputs are required to be '1' or '0' when the switches are in the open position.

PROGRAMMING THE SWITCH PAD MODULE

As with the I/O Port Monitor, the Switch Pad is also very simple to use. Again we will pose a problem which will use the module and extend your programming expertise.

● EXERCISE 4.1.2

- Draw up a flow-chart which will read data presented by the Switch Pad to Port B and output the same data at Port A. Use the I/O Port Monitor to receive the data input from the Switch Pad.
- Design a program based upon your flow-chart to effect the data transfer required.

● EXERCISE 4.1.3.

- Draw a flow-chart which will read data presented by the Switch Pad to Port A, bits PA_0 - PA_3 and output the same data also to Port A but to bits PA_4 - PA_7 . Use the I/O Port to monitor output.
- Design your program, enter and test.

This exercise should indicate the versatility of a single output port to accommodate both inputs and outputs simultaneously.

You may consider, when designing your program, to use the new instruction 'Shift Left One Bit' (ASL).

THE BUFFERED LOUD-SPEAKER MODULE

The Buffered Loud-Speaker Module provides a means of obtaining an audible output from the square waves generated at any of the output port bits.

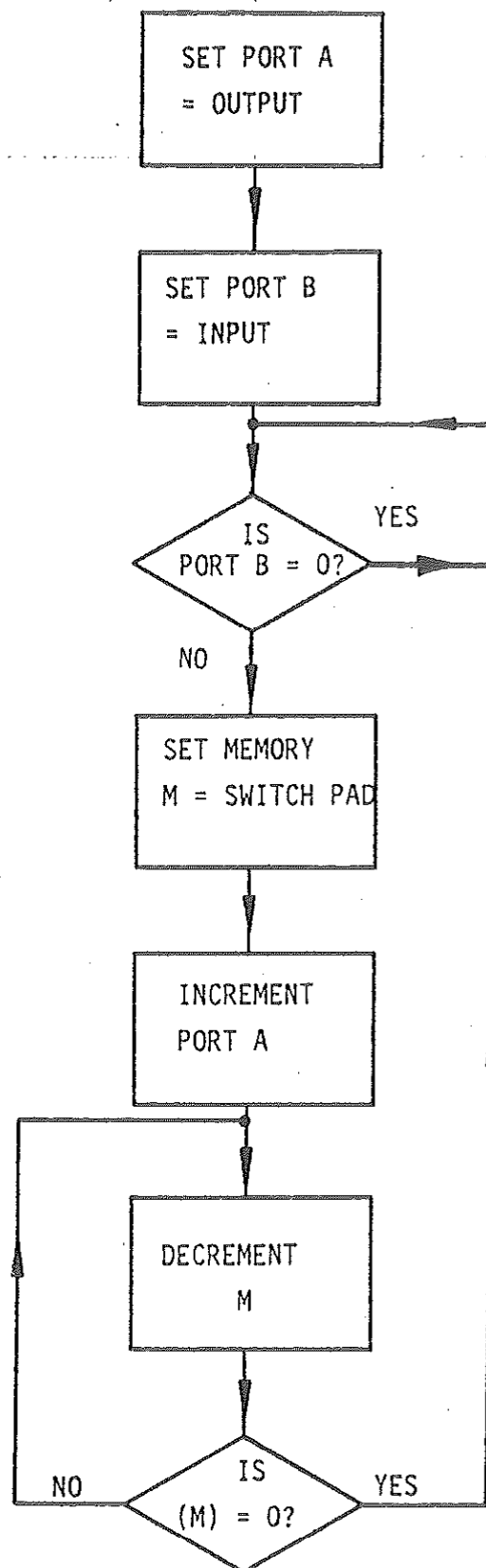
The programming technique introduced below provides a means of generating a sequence of steps (or delays between steps) that is proportional to some input data.

We will write a program to increment Port A, with a time delay between increments, that is proportional to data input via the switch pad. The switch pad will input to Port B.

The switch pad will be configured so that a logic '1' is output for a pushbutton depressed.

The buffered loud-speaker can be connected to any of the bits of Port A.

FLOW CHART:



PROGRAM:

HEXADECIMAL				SYMBOLIC ASSEMBLER INSTRUCTIONS			
ADDR	1	2	3	LABEL	MNEM	OPERAND	COMMENTS
					LDA#	FF	Set Port A = Output
					STA	DDRA	
					LDA#	00	Set Port B = Input
					STA	DDRB	
				WAIT	LDX	DRB	Wait for key(s) to be depressed
					BEQ	WAIT	
					INC	DRA	Pulse Loud-Speaker
				TONE	DEX		Produce interval between Loud-Speaker pulses
					BNE	TONE	
					WAIT		New Key(s)?

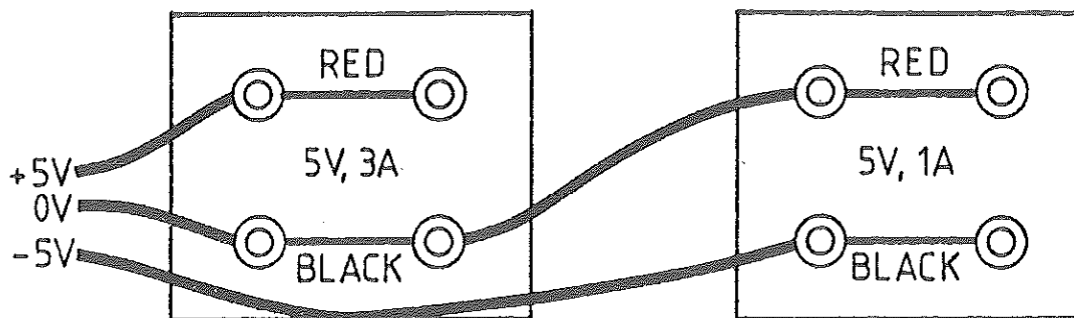
EXERCISE 4.1.4

- Convert the Symbolic Assembler Instructions into Hexadecimal Coding.
- Input program and execute. A suitably audible tone will be produced at PA₃.
- Does the 'organ' perform as you may have expected from the program?

THE STRAIN GAUGE MODULE

The Strain Gauge module gives an analogue output voltage proportional to the load placed on the weighing disc. It will be necessary to convert this voltage through the A/D Converter module for digital input to **Enna** user ports. We will also need to calibrate it.

Before using the strain gauge we must first connect the system power 90 to provide a plus or minus 5V supply. This connection will also be required for other modules used later. The diagram below explains



CALIBRATION of the strain gauge module is required simply because it is an analogue device. It may also require zeroing before use.

The following exercise goes through this procedure together with a method for calibration.

- EXERCISE 4.1.5
- Connect up the system power 90.
- Connect up the strain gauge module. Wait 10 minutes for warm up.
- Connect a suitable analogue or digital voltmeter to the output and '0' volt line. Zero output. Note: You may find the use of a digital instrument difficult to use for this type of measurement. Why?

- Progressively load the loading platform and plot a calibration curve of output voltage/input weight. File your curve for future reference.

NOTE: It is important that weights are placed on the loading platform centrally.

We will use the module as an application hardware module with **Ema** when we have considered the Analogue to Digital Converter.

THE ANALOGUE TO DIGITAL CONVERTER MODULE

The Analogue to Digital Converter Module is versatile in as far as it is capable of converting analogue signals to digital and digital to analogue. We therefore have to provide an input to set the device mode of operation e.g. $A \rightarrow D$ (usually written A/D) or $D \rightarrow A$ (usually written D/A). The input SELECT performs this function:

SELECT INPUT	MODE OF OPERATION
LOGIC '1'	A/D
LOGIC '0'	D/A

The architecture of the device requires that in the A/D mode, a counter is first set to zero and then incremented until it 'compares' with the analogue input.

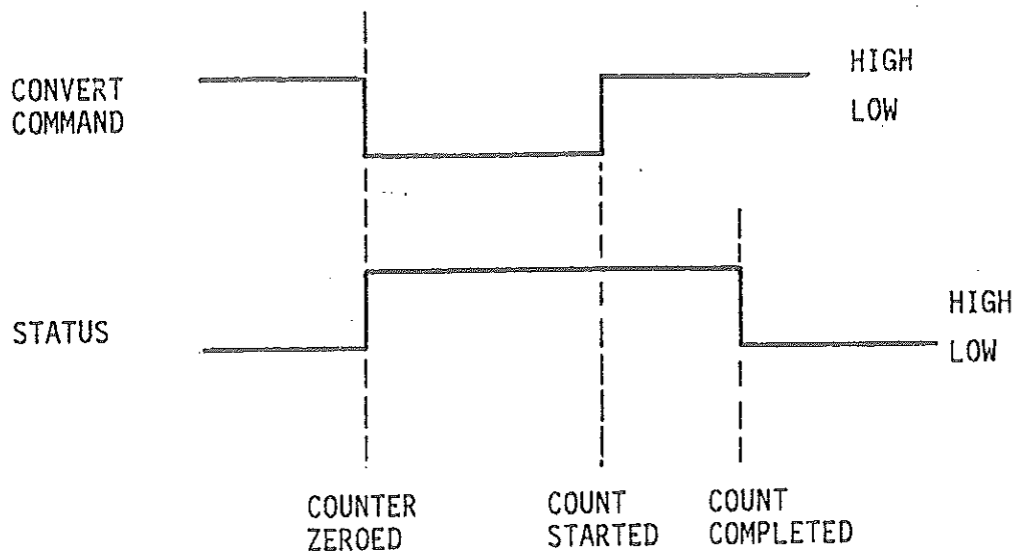
The CONVERT COMMAND input performs this function:

CONVERT COMMAND	OPERATION
goes 'low'	Resets counter to zero
goes 'high'	starts incrementing counter

The STATUS of the device is also given:

STATUS OUTPUT	DEVICE STATUS
goes 'high'	Counter set to zero
goes 'low'	Counter stopped at $D \approx A$

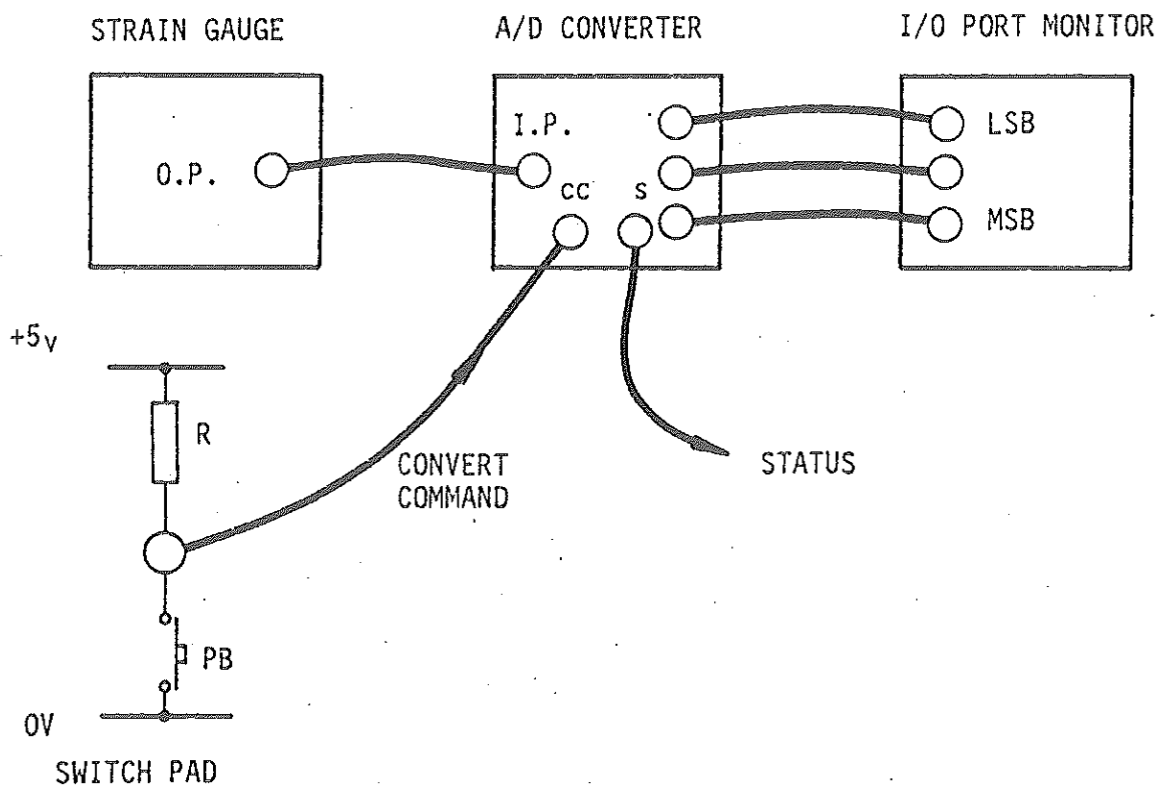
To summarise:



We must also **calibrate** the Analogue to Digital Converter. We will do this as an exercise.

● EXERCISE 4.1.6.

- Connect up the strain gauge, A/D and I/O Port Monitor as below:

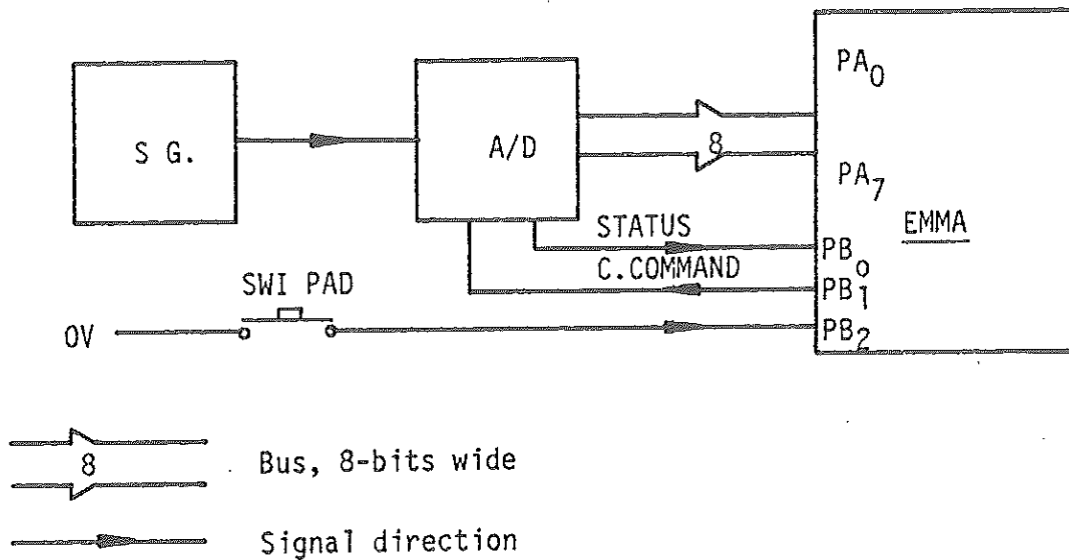


Depress swich and release to effect 'weigh'.

Following a similar procedure as for the strain gauge, calibrate the A/D, record and file for reference.

Having gained some familiarity with both the strain gauge and the A/D converter we will write a program in assembly language which will monitor the output of the strain gauge via the A/D converter and store it in a memory location in EMMA when a 'start weigh' command is given.

- Flow Chart - see page 13
- Program - see page 14
- Connection Diagram

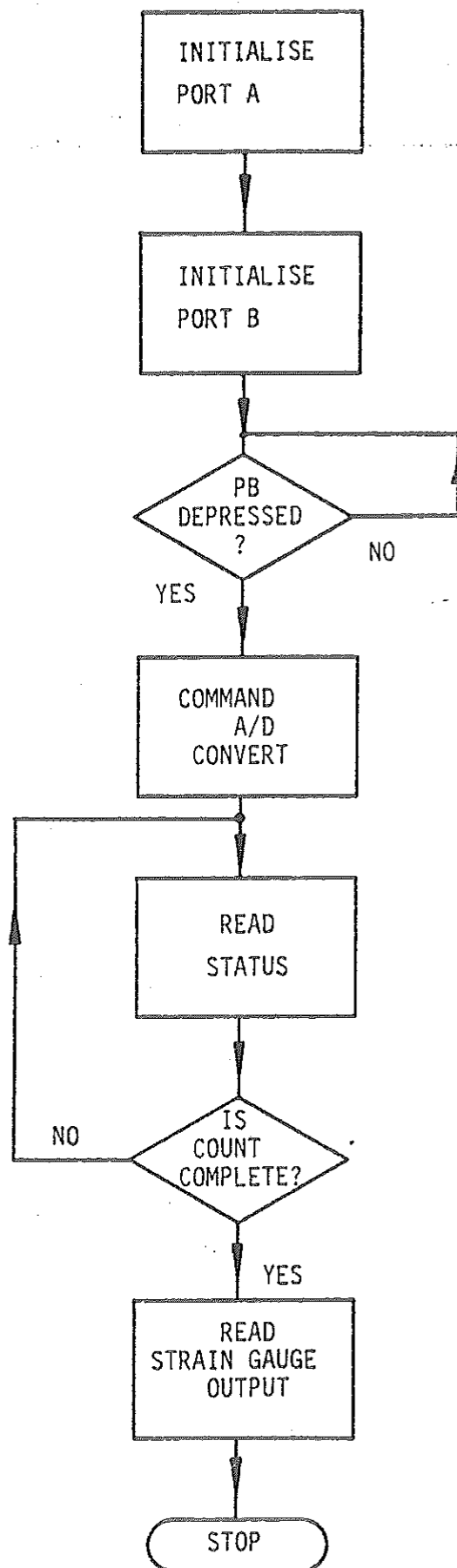


DEPRESS SW. PAD Key for 'weigh command'.

Comments:

The connection diagram indicates that bits PA₀-PA₇ are all set to INPUT, PB₀ and PB₂ are set to INPUT and PB₁ is set to OUTPUT. We are not interested in the other bits of Port B.

The **Convert Command** must be held low for a finite time to allow the counter to reset.



Again we will just do an assembler listing for you.

HEXADECIMAL				SYMBOLIC ASSEMBLER INSTRUCTIONS			
ADDR	1	2	3	LABEL	MNEM	OPERAND	COMMENTS
					LDA#	00	Sets Port A to input
					STA	DDRA	
					LDA#	02	Sets Port B B ₀ - INPUT B ₁ - OUTPUT B ₂ - INPUT
					STA	DDRB	
				MONSP	LDA	DRB	Monitors Switch Pad
					AND#	04	
					BNE	MONSP	
					LDA#	02	Takes convert command low for finite period
					STA	DRB	
					LDA#	00	
					STA	DRB	
					JSR	****	
					LDA#	02	
					STA	DRB	
				MONST	LDA	DRB	Monitors STATUS
					AND#	01	
					BNE	MONST	
					LDA	DRA	
					STA	MEMORY	
				FIN	JMP	FIN	

At the moment, we are producing the machine code program by 'hand'. Later you will be using an ASSEMBLER to do this work - assemblers have many features but, at this stage, you will appreciate that their use eliminates much tedious work and errors that may occur during this stage. Incidentally, the program we have written is referred to as a **Source Program**. You are going to convert this into an **Object Program** by 'hand assembly'.

- EXERCISE 4.1.7.
 - Produce an Object Program for EMMA using the Source Program given.
- Note: The program uses a time delay sub-routine to hold the convert command low for the finite period. You will need to write this delay routine and enter a suitable address.
- Run the program and check that the required 'weight' is stored in the memory location you have chosen.

Detailed specifications of each module are included in Appendix (Application Modules).

PROGRAMS USING I/O PORT MONITOR, D/A AND OSCILLOSCOPE

Exercise 4.2.1

Write a program to increment port A and then decrement port B repeatedly. Observe PA₇ and PB₇ on your dual beam oscilloscope.

Exercise 4.2.2

Write a program to output a table of nine values (00, 01, 02, 04, 08, 10, 20, 40, 80) to the I/O port monitor. Arrange the program so that the time delay between outputs is progressively increased and then decreased so sweeping the frequency at which the display sequence is generated.

Exercise 4.2.3

Write a program to produce a repeating 8-bit incrementing count at Port A. Connect to port A your A/D converter and observe, on the oscilloscope, that the D/A output is a sawtooth waveform.

Exercise 4.2.4

Write a program to produce a triangular waveform at the D/A output. Include a 'time waste' routine in each increment/decrement loop to enable the wave frequency to be varied.

Exercise 4.2.5

Write a program to produce a sinusoidal waveform at the D/A output. The program will be most easily accomplished by storing a look-up table of sine values.

● Exercise 4.2.6

Write a program to cause the A/D converter to perform a conversion every 0.5 seconds, reading the converted values into consecutive memory locations. The program is to cease conversions when 256 locations (1 page of memory) has been loaded. A variable resistor may be used to provide an analogue input which varies from 0V to +5V.

The converted values may be observed by inspection of the memory locations.

● Exercise 4.2.7.

Extend the program of 4.2.6 to cause the microcomputer to select the D/A function of the A/D converter module and output the stored digital values repeatedly. The output can be observed on an oscilloscope.

The combined programs of 4.2.6 and 4.2.7 provide a storage oscilloscope facility for an event occurring over 128 seconds. Reduction of delay between conversion will reduce the total time of the stored event.

PROGRAMS USING THE SWITCH PAD AND BUFFERED LOUDSPEAKER

● Exercise 4.2.8

Connect the Switch Pad Module to Port B PB_0 to PB_7 and connect the loudspeaker to any bit of Port A. Write a program to give a correct musical scale on operation of the individual switches.

The following frequencies give a suitable scale:

500	Hz
561	Hz
629	Hz
667	Hz
749	Hz
840	Hz
943	Hz
1000	Hz

● Exercise 4.2.9

Write a program to cause the computer to play a tune according to data stored by the user.

PROGRAMS USING THE STRAIN GAUGE

● Exercise 4.2.10

Connect the Strain Gauge, A/D Converter, Switch Pad and Emma Microcomputer as shown in Chapter 4.1 page 12. Write a program to read the A/D Converter output and display this on the microcomputers 7-segment displays.

Note: The Appendix (Useful Emma Monitor Sub-routines) provides information on sub-routines which may be of use.

● Exercise 4.2.11

Using the program above, weigh ten small coins or washers and calculate the average weight. Write a program to identify the number of these coins or washers placed on the platform and display this number on the 7-segment displays.

SECTION 5

APPENDICES

<u>TITLE</u>	<u>PAGE REF:</u>
Conversion Tables	
Standard Coding Sheet	1
Application Modules	2
Emma Monitor Sub-Routines.	3
Emma Memory Map.	4
Oscilloscopes.	5
Instruction Set.	6
Microcomputer Glossary.	7
Solutions to Questions.	8

APPENDIX 1

CONVERSION TABLES

STANDARD CODING SHEET

HEXADECIMAL CONVERSION TABLE

*	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
5	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
8	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
9	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
A	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
B	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
C	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
D	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
E	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
F	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255

* Hexadecimal values

RELATIVE BRANCH TABLES

FORWARD RELATIVE BRANCH

*	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
5	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	127	128

* Forward Relative Branch Values.

BACKWARD RELATIVE BRANCH

*	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
8	128	127	126	125	124	123	122	121	120	119	118	117	116	115	114	113
9	112	111	110	109	108	107	106	105	104	103	102	101	100	99	98	97
A	96	95	94	93	92	91	90	89	88	87	86	85	84	83	82	81
B	80	79	78	77	76	75	74	73	72	71	70	69	68	67	66	65
C	64	63	62	61	60	59	58	57	56	55	54	53	52	51	50	49
D	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33
E	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17
F	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1

* Backward Relative Branch Value

EMMA Program Sheet No:

Programmer:

Program Title:

[illegible]

VISA Program Sheet No.

Programmer:

Program Title:

Assembled/Disassembled

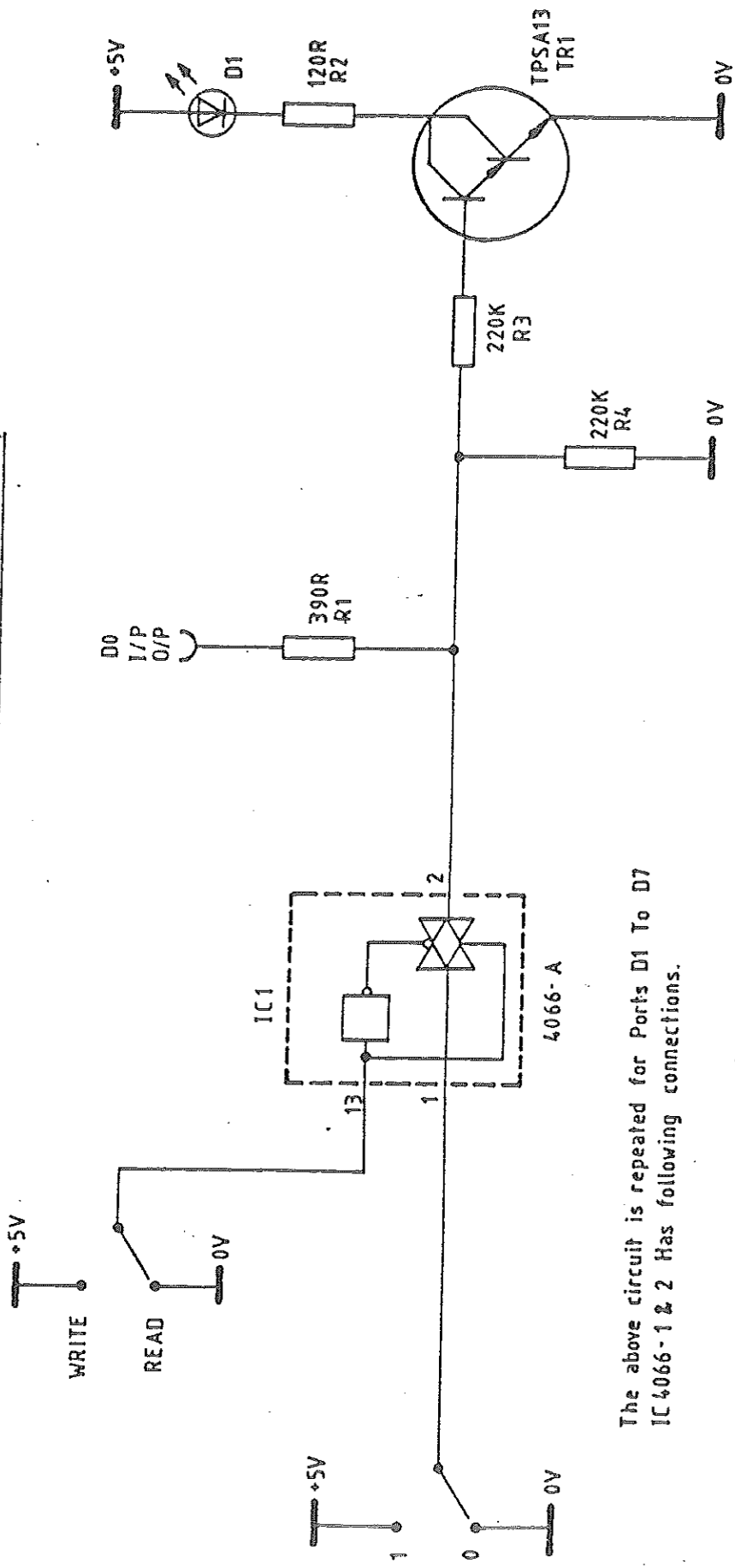
Address /Label	OP Code	MNEM	Operand /Label	Comments

APPENDIX 2

APPLICATION MODULES

This Appendix provides lay-out and schematic diagrams together with relevant data of application modules included in the Microprocessor Application System MA02.

I/O MONITOR CIRCUIT DIAGRAM (Port D0)



The above circuit is repeated for Ports D1 To D7
IC 4066-1 & 2 Has following connections.

IC1

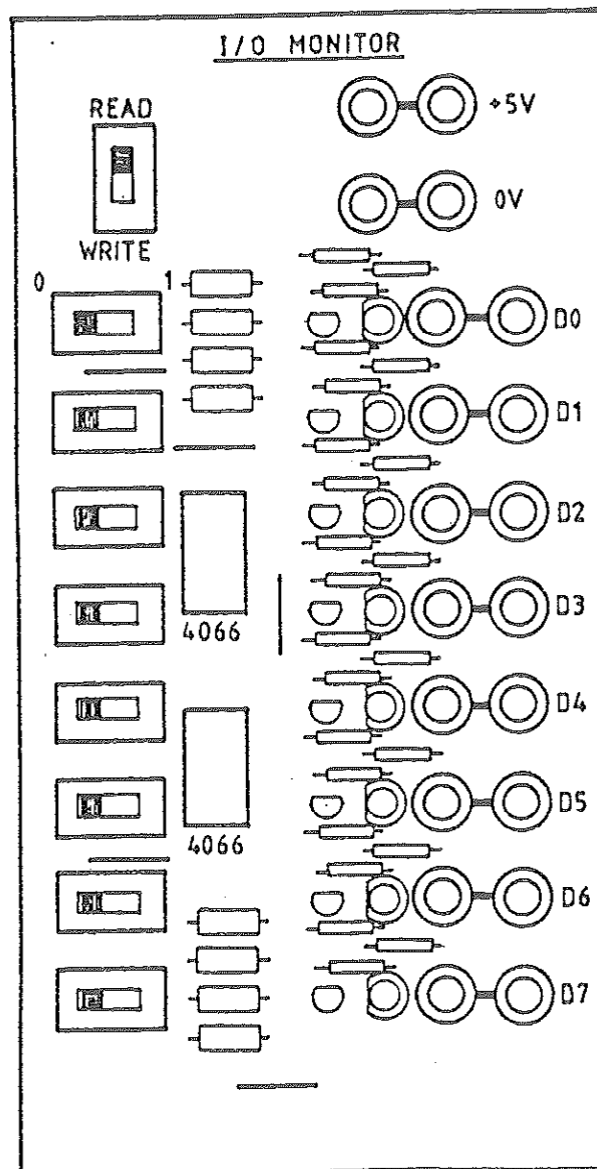
R/W	1-0	O/P	PORT
13	1	2	D0
5	4	3	D1
6	8	9	D3
12	11	10	D2

4066-1

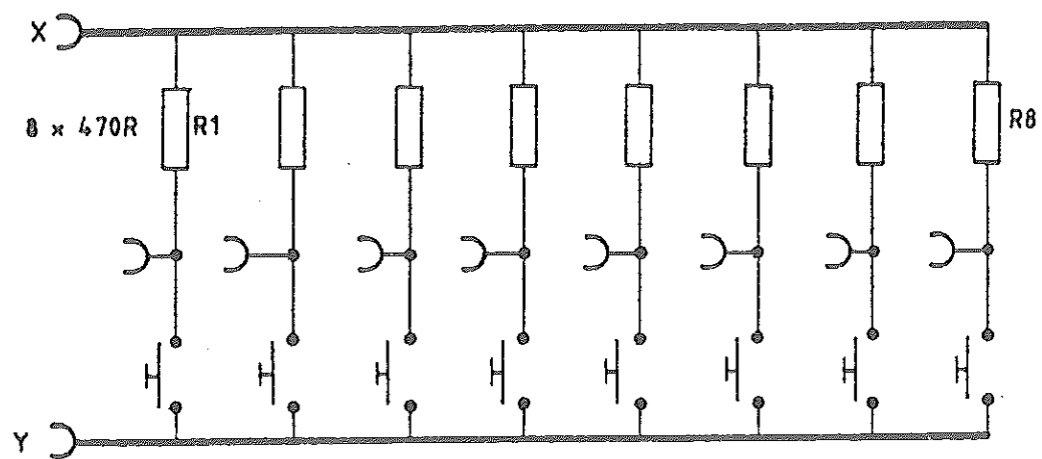
IC2

R/W	1-0	O/P	PORT
13	1	2	D4
5	4	3	D5
6	8	9	D7
12	11	10	D6

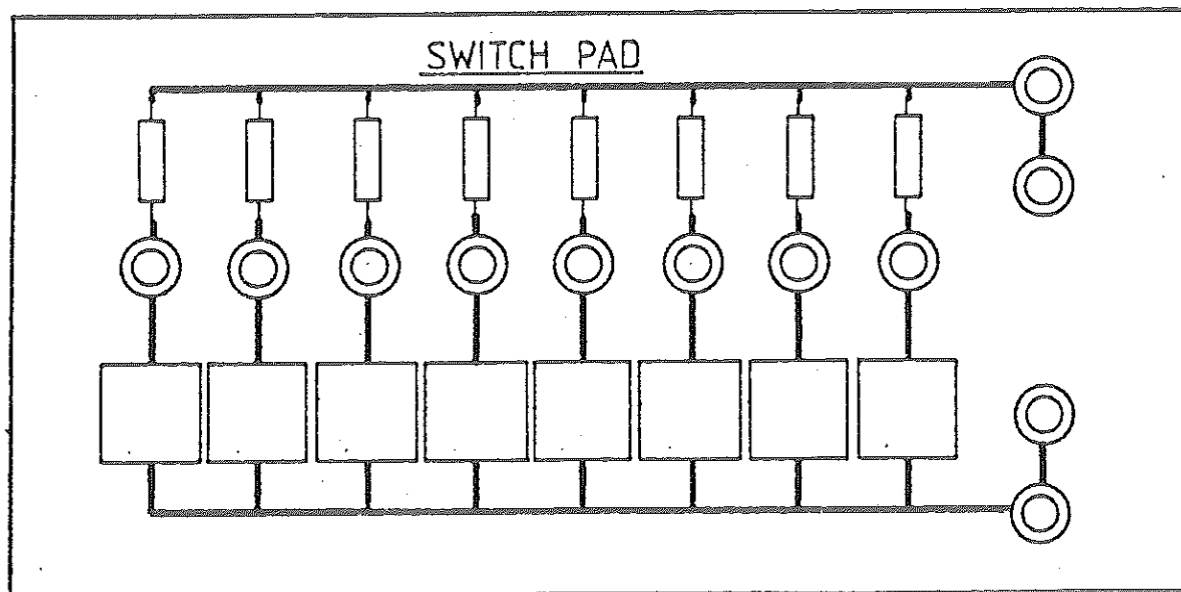
4066-2



I/O MONITOR LAYOUT



Switch Pad Circuit Diagram.



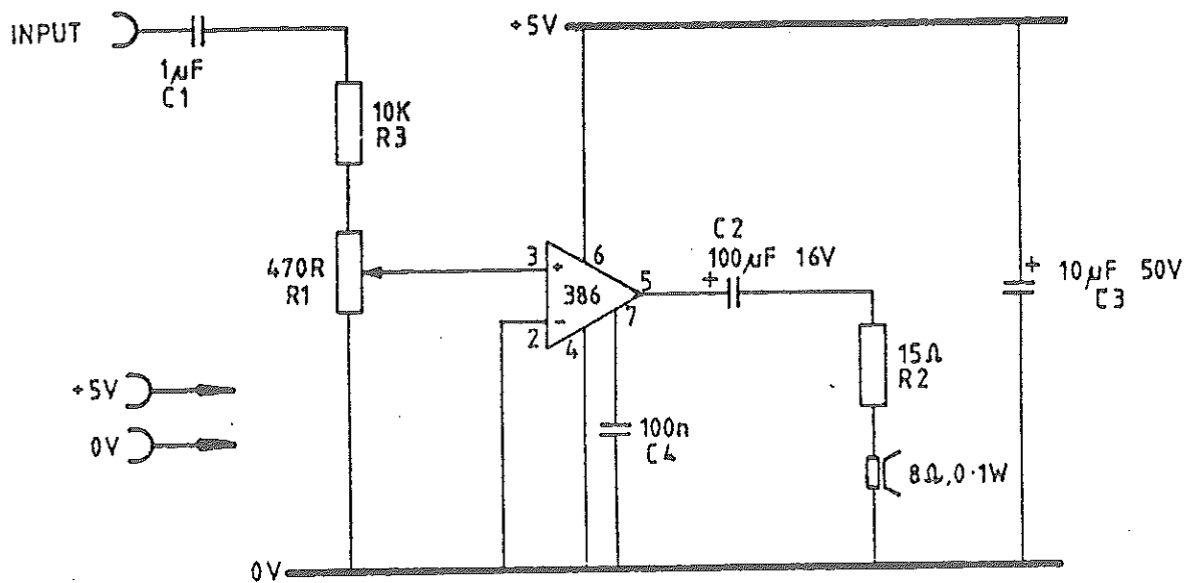
BUFFERED LOUDSPEAKER

This module incorporates a 1½", 0.2W., Loudspeaker. The buffer amplifier is designed to operate from a logic or sinusoidal input.

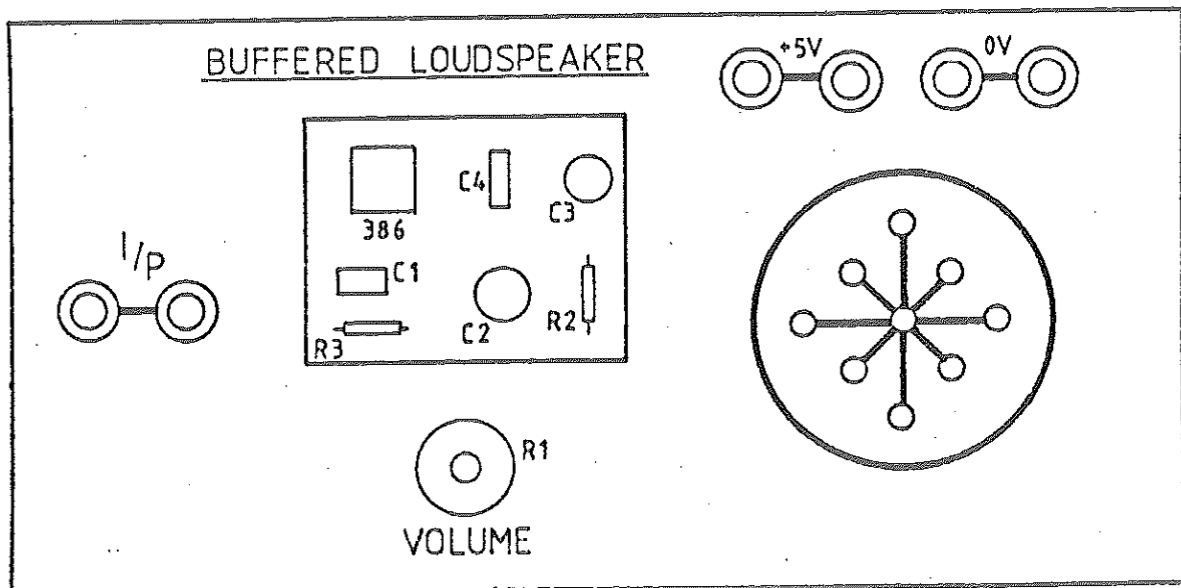
Specification:

Logic (assume 5V logic) - current drawn from '1' 50 A.
current sunk into '0' 400 A.

Sine - input resistance 400 k
power gain 45dB
bandwidth 50Hz - 20kHz
current drawn from 5B supply 200mA



BUFFERED LOUDSPEAKER CIRCUIT

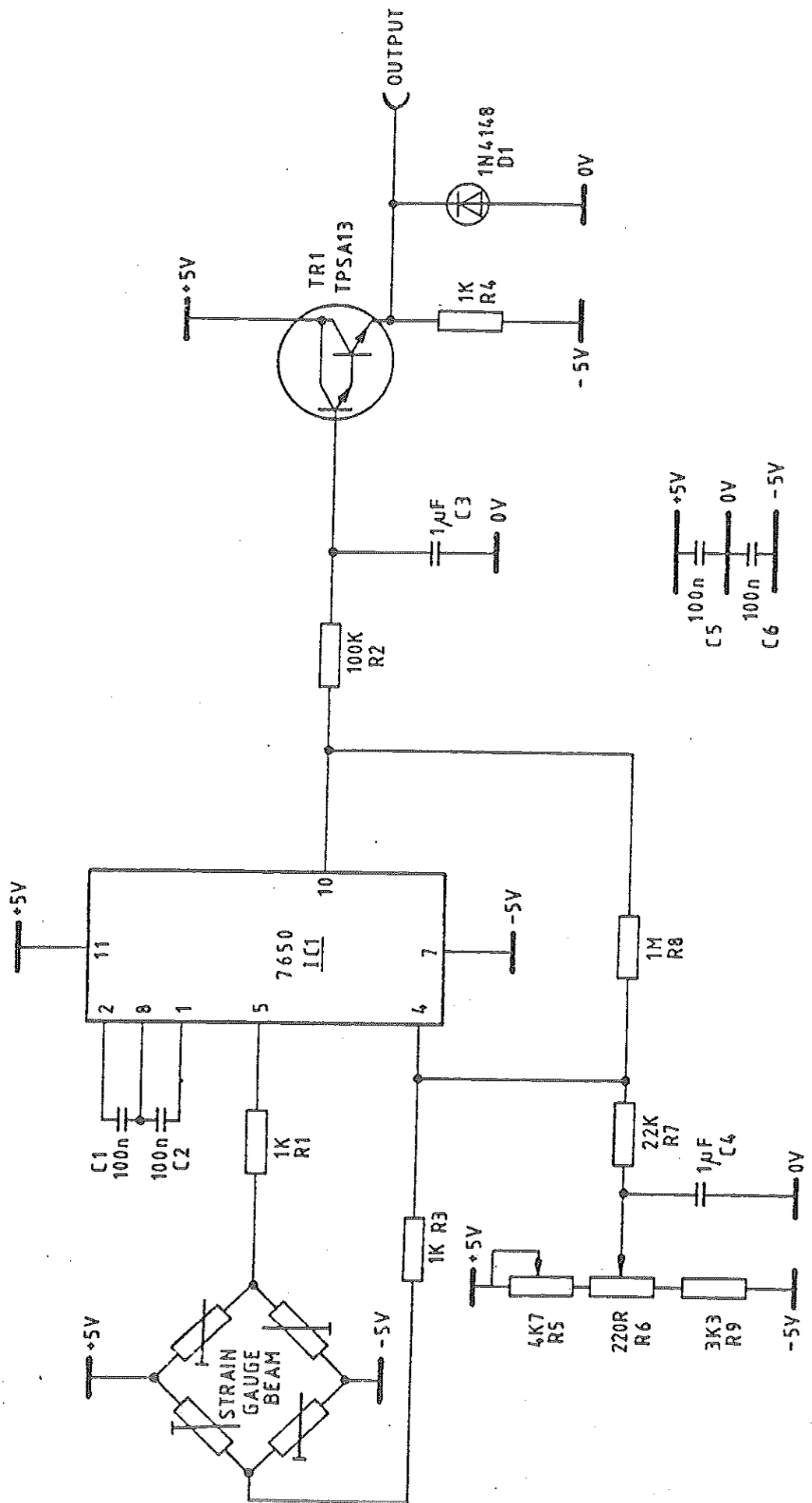


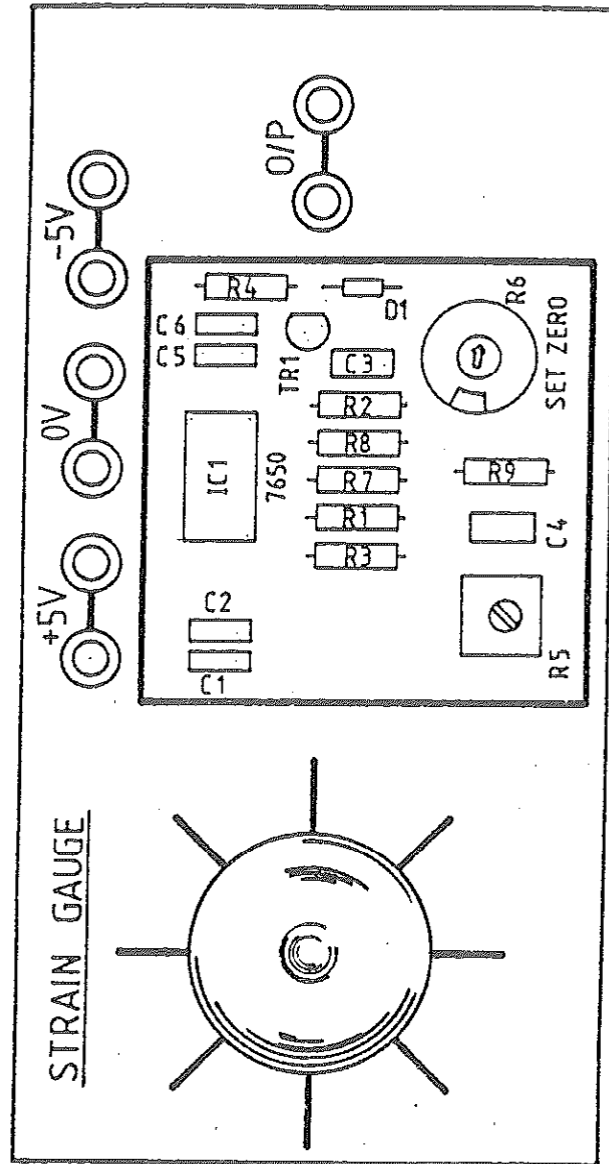
STRAIN GAUGE SPECIFICATION

The Strain Gauge module is based on a cantilever beam with four thin-film deposited strain gauges. Bonded to this beam is a 1" aluminium disc which is used as a weighing platform.

The strain gauge bridge output is then amplified through a 7600 commutating auto-zero amplifier to give approximately 250mV change in output voltage for a 10p piece placed on the weighing platform.

STRAIN GAUGE CIRCUIT DIAGRAM



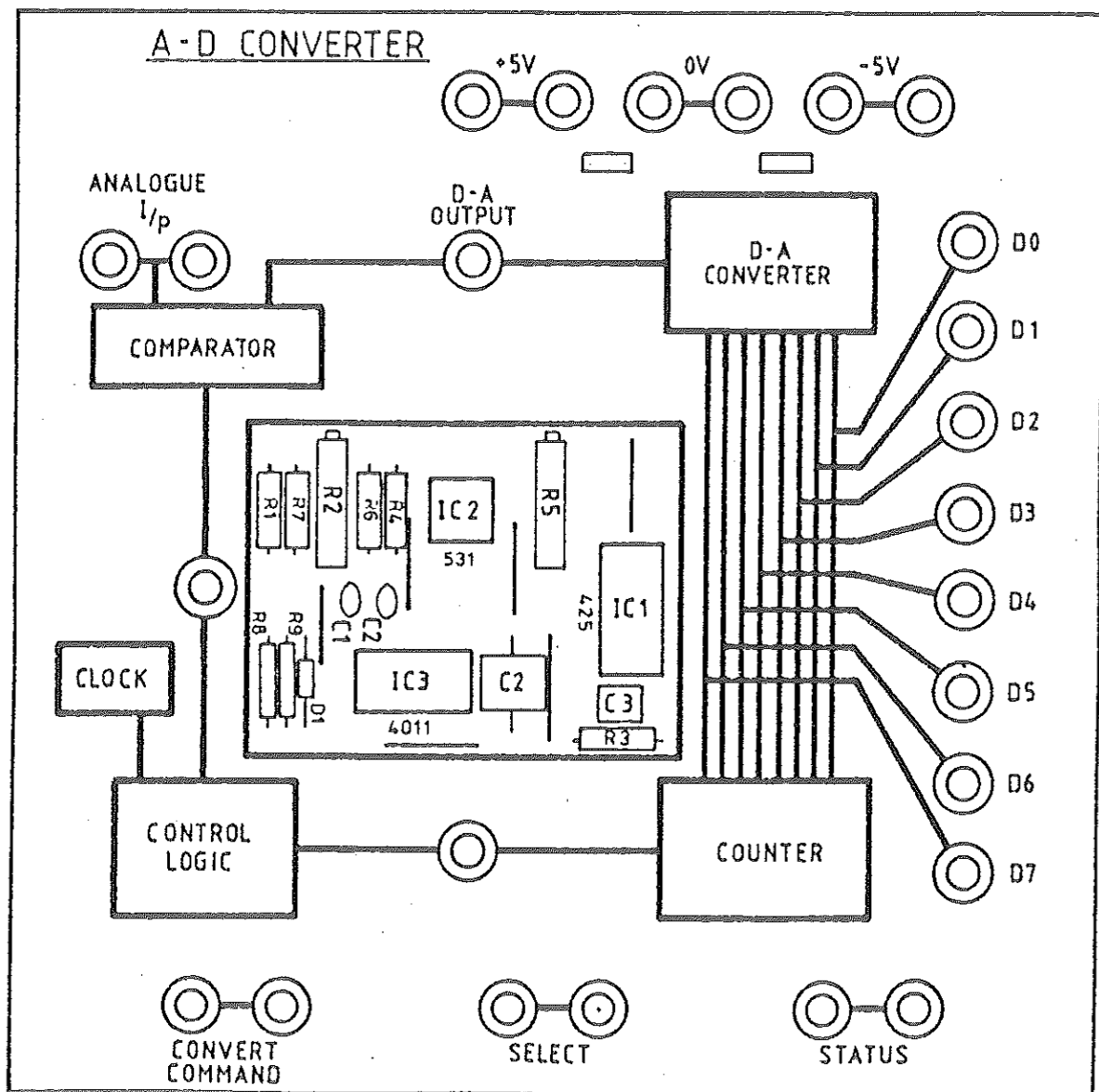


STRAIN GAUGE LAYOUT

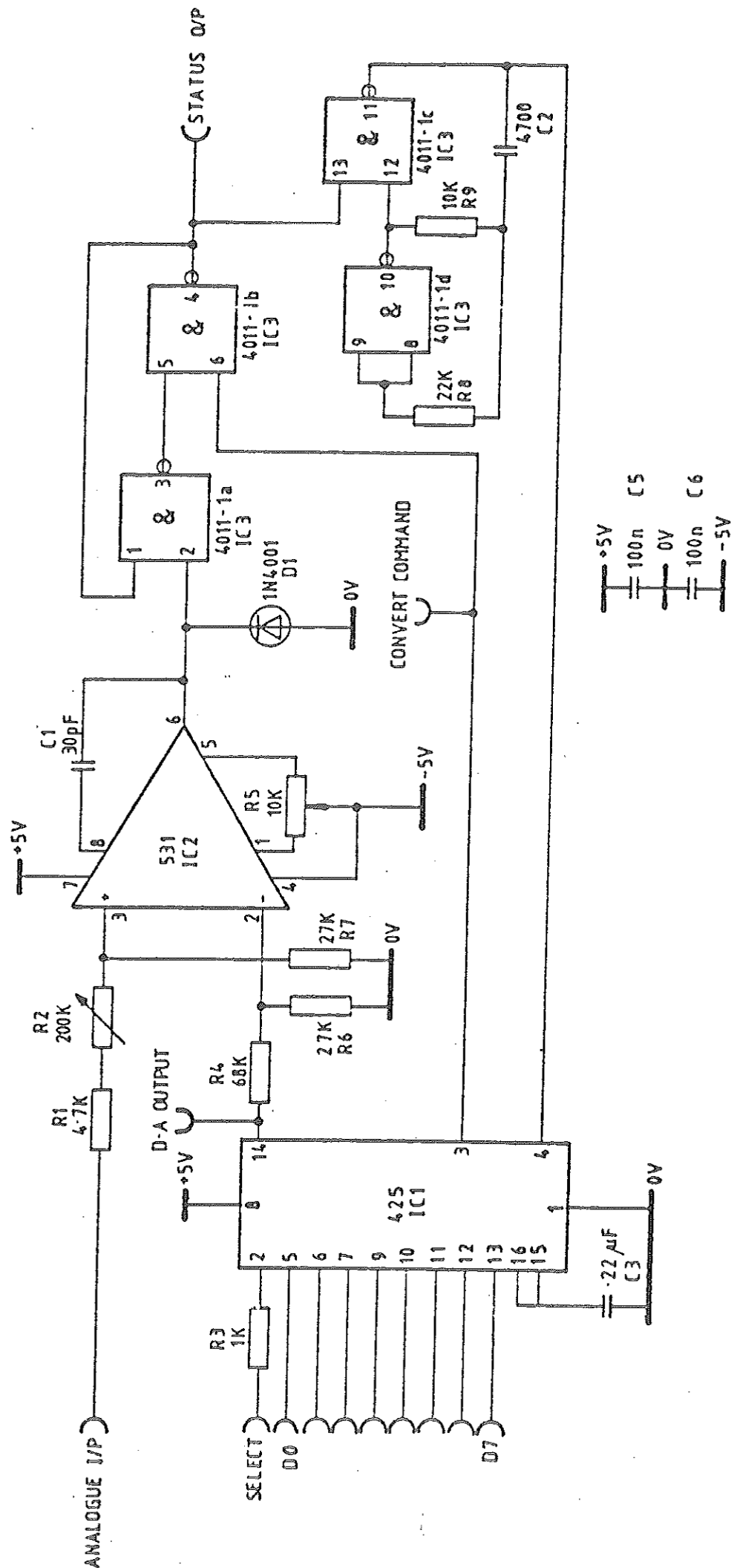
A/D Converter

Setting-Up

- Provide +0V analogue signal to input and adjust preset (Right-hand potentiometer) to give 00_{HEX} on I/O Port Monitor.
- Provide +5V analogue signal to input and adjust preset (Left-hand potentiometer) to give FF_{HEX} on I/O Port Monitor.
- Repeat zero and sensitivity settings until both are correct.



A/D CONVERTER CIRCUIT DIAGRAM



APPENDIX 3

EMMA MONITOR SUB-ROUTINES

USER ROUTINES

Included in the monitor program of EMMA II are sub-routines that perform functions to assist the programmer. This section of the manual will explain how to use these routines.

● HEXADECIMAL TO DECIMAL/DECIMAL TO HEXADECIMAL

Routine Start Address = D900.

A mode signal 'H' will appear indicating that the routine is in HEX DEC mode. The 'H' is asking for a HEX number up to 4 digits to be entered. After a HEX number has been entered. Pressing **P** "PROCESS", will change the state of the display to decimal and display 'D' in the mode character location and the decimal equivalent of the HEX number on the right of the display.

e.g.

H	F5
D	245

● BRANCH OFFSET CALCULATOR

Routine Start Address = D980

This sub-routine does branch calculations for the programmer.

Example:

Run the program from D980; the mode digit displays 'F' asking for a "FROM" address to be entered.

e.g. Enter 0045 as "FROM" address by pressing **P**
'Process' the mode character 't' will appear asking for a "TO" address.

For this example enter 0060.

Pressing **P** again and the mode character will display a '0' meaning that it is in the offset mode and displaying a two digit offset, '19' in this example.

If the branch is out of range, the mode character will show 'E' meaning ERROR.

In both of these cases pressing (Re-Run) will re-run the program.

i.e. Return to the "FROM" situation for the next calculation. By pressing the monitor program runs and a reset situation exists.

● RELOCATOR

Routine Start Address = D9C9

This program is designed to move blocks of program to new address locations. To use this program run from "D9C9". The mode digit means "F" asking for start address of block to be moved for an example enter 0245 as start address. Pressing "PROCESS" the mode digit will display 't' indicating the need for an end address +1 of the block to be moved, for this example enter 0260.

Pressing key again the command will be 'd' asking for a destination address, for the example enter 0800.

By pressing the the program will now transfer data from 0245 - 025F to a new address 0800; and return to the monitor program and a row of dots will appear on the display.

● CHECKSUM ROUTINE

Routine Start Address = DA50

A checksum is done by adding a group of data bytes (e.g. a program) together and forming a total to be used for comparison; for example after transmission etc a bit could be corrupted hence the program would have a different checksum. To use this routine:

Go from DA50

The mode character displays 'F' asking for an address of start of block to be checksummed.

Enter address.

Press the mode displays 'n' asking for the number of bytes.
Enter number of bytes

Press **P**

The mode character will display 'C' indicating a checksum and the checksum appears on the right of the display.

By pressing '0' the program returns to the monitor and a reset condition.

USEFUL SUBROUTINES

● DATA INSERT ROUTINE

Routine Start Address = D805

This routine can be used in a program to ask the user to insert data to be used as a function of the program. To utilise the program the 'Y' register needs to be set to the number of characters of the data. The start address of the sub-routine is D805. For example enter the small program.

0200	A0	05		LDY	#05	Number of characters
0202	20	05	D8	JSR	D805	'Insert' subroutine
0205	4C	E0	FE	JMP	FEE0	Reset.

Run the program from 0200.

The display will show G 0200.

Now you can enter data up to 5 HEX characters. The displayed characters will shift left as this is being loaded. For this example enter 01234. By pressing PROCESS key **P** the program is executed. The data has been converted into a 4 byte code and stored at 001C - 001F by examination.

1C = 34
1D = 12
1E = 00
1F = 00

From the example during the routine 'X' is used and restored on exit 'Y' is used and restored on exit.

● DISPLAY 8

Routine Start Address = D849

This routine transfers data from 001C - 001F to the display buffer in seven segment form. Zero suppression is also performed. For example load locations 001C - 001F with the following:

001C	67
001D	45
001E	23
001F	01

Run from D849 and the data will be displayed

1	2	3	4	5	6	7
---	---	---	---	---	---	---

● DISPLAY

Routine Start Address = D84D

This routine is similar to 'DISPLAY 8', the only difference is the data transferred for display is from 1C - 1E. Digit 1 of the display is cleared and digit 0 is loaded from 1B. 1B can be loaded to give a symbol or letter as a code of information. For example; to display length 'L' at 7500 load:

001B	38	Seven segment code for 'L'
001C	00	
001D	75	Data (7500)
001E	00	

Run from D84D

The display will read L 7500

● MULTIPLY BY 10

Routine Start Address = D89D

This routine multiplies the 3 byte number in 0A - 0C by 10_{based 10} and stores the result in HEX back in 0A - 0C. Care must be taken to ensure the monitor program does not corrupt the answer. (i.e. if the monitor uses 0A - 0C)

● **MULTIPLY BY 16**

Routine Start Address = D8B3

This routine multiplies the 3 byte number in 0A - 0C by 16_{10} and stores the result at 0A - 0C. This routine starts at D8B3. By modifying the program above at:

	0211	20	90	D8	JSR	D890
To	0211	20	B3	D8	JSR	D8B3

The result of the calculation will be displayed.

● **DISPLAY MEMORY CONTENTS**

Routine Start Address = FE0C

Displays seven segment codes stored at:

0010 - 0017

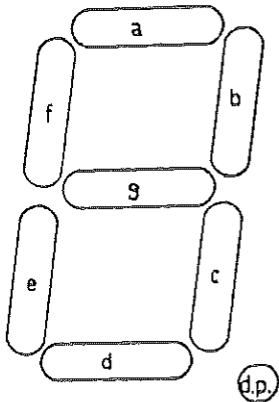
If the contents of 000E is less than 80 the display is scanned once and returns from the sub-routine. If the contents of 000E is greater than 80 the display is continuously scanned.

To show how this operation is done this sub-routine can be used:

HEXADECIMAL				SYMBOLIC		COMMENTS
ADDR	1	2	3	LABEL	MNEM	OPERAND
0300	A9	80			LDA	#80
0302	85	0E			STA	0E
0304	A2	00			LDX	#00
0306	B5	20		*	LDA	20,X
0308	95	10			STA	10,X
030A	E8				INX	
030B	E0	08			CPX	#80
030D	D0	F7			BNE	*
030F	20	0C	FE		JSR	FE0C
0312	8D	00	02		STA	0200
0315	4C	E0	FE		JMP	FE00

The program loads the contents of 0020 to 0027 into 0010 to 0017 and then displays them.

The code required to illuminate each segment are shown below:

	<u>SEGMENT</u>	<u>CODE</u>
	a	01
	b	02
	c	04
	d	08
	e	10
	f	20
	g	40
	d.p.	80

WRITE 4 CHARACTERS

Routine Start Address = FE00

Displays contents of $(0000 + X - 3)$ $(0000 + X - 2)$ $(0000 + X - 1)$ and $(0000 + X)$, from left to right, on the display.

For example: Load 01 into location 002D, 23 into location 002E, 45 into location 002F, 67 into location 0030.

Load the following simple program starting at location 0300:-

0300	A2,30	LDX#30
0302	20,00,FE	JSR FE00

Run the program from 0300, i.e. press 'G', 0300, 'G'.

The display will show the contents of locations 002D to 0030 inclusive. This subroutine drops through to subroutine FE0C to display the information.

NOTE: The subroutine uses the accumulator, X register and Y register, so if information in these registers is needed, then it must be stored elsewhere before entering the subroutine.

DISPLAY MEMORY CONTENTS

Routine Start Address = FE5E

Converts the contents of the indirect address formed from locations (0000 + X) and (0000 + X + 1) into seven segment codes, and stores them in locations 0016 and 0017. Jumping to subroutine FEOC will cause the information to be displayed on the right hand side of the display.

Load the following program starting at 0300:-

0300	A2,00	LDA#00	
0302	8A	TXA	
0303	95,10	* STA 10,X	clears
0305	E8	INX	locations
0306	E0,08	CPX#08	0010-0017
0308	D0,F9	BNE *	
030A	A9,80	LDA#80	
030C	85,0E	STA 0E	
030E	A2,20	LDX#20	
0310	20,5E,FE	JSR FE5E	
0313	20,0C,FE	JSR FEOC	

Load 0020 with 30 and 0021 with 00. Load location 30 with AB.

Run the program from 0300, the display will show B on the right-hand two digits. The subroutine starting at FE5E uses the numbers stored in 0020 and 0021 (X = 20) to form an indirect address (in this case 0030) and then converts the contents of this location into seven segment codes and stores them at 0016 and 0017. Jumping to the subroutine at FEOC displays these characters.

The subroutine uses the accumulator and Y register.

DISPLAY ACCUMULATOR CONTENTS

Routine Start Address = FE60

Converts the contents of the accumulator into seven segment codes and stores them in locations 0016 and 0017. To display the contents of these locations, jump to subroutine FEOC on returning from the subroutine at FE60.

To show how these subroutines can be used, load the following program:-

0300	A2,00	LDX#00	
0302	8A	TXA	
0303	95,10	* STA 10,X	clears
0305	E8	INX	locations
0306	E0,08	CPX#08	0010-0017
0308	D0,F9	BNE *	
030A	A9,80	LDA#80	
030C	85,0E	STA 0E	
030E	A5,30	LDA 0030	
0310	20,60,FE	JSR FE60	
0313	20,0C,FE	JSR FEOC	

In this program the contents of 0030 are loaded into the accumulator. On jumping to subroutine FE60, this information is converted into seven segment codes and stored at 0016 and 0017. Jumping to subroutine FEOC displays this information on the right of the display.

The subroutine uses the accumulator and Y register.

DISPLAY 4 CHARACTERS

Routine Start Address = FE64

Converts the contents of locations $(0000 + X + 1)$ and $(0000 + X)$ into seven segment codes and stores them at locations 0011, 0012, 0013 and 0014 respectively. Jumping to subroutine FEOC will display this information on the left hand centre of the display.

To show how this subroutine can be used, load the following program starting at 0300:-

0300	A2,00	LDX#00
0302	8A	TXA
0303	95,10	* STA 10,X
0305	E8	INX
0306	E0,08	CPX#08
0308	D0,F9	BNE *
030A	A9,80	LDA#80
030C	85,0E	STA 0E
030E	A2,30	LDX#30
0310	20,64,FE	JSR FE64
0313	20,0C,FE	JSR FEOC

Load location 0030 with 23 and 0031 with 01. Run the program and 0123 will be displayed.

The program displays the contents of locations 0030 and 0031 by first jumping to the subroutine at FE64. This converts the information into seven segment codes and stores these at location 0011, 0012, 0013 and 0014. The program then jumps to the subroutine at FEOC to display the information.

The subroutine uses the accumulator and Y register.

READ HEXADECIMAL KEYS

Routine Start Address = FE88

Shifts data entered on the keyboard hexadecimal keys into memory locations (0000 + X + 1) and (0000 + X), the subroutine then jumps to the subroutines starting at addresses FE64 and FE0C to display the information in the second, third, fourth and fifth digit positions (from the left side of the display). When a command key is pressed the subroutine is exited with the value of the command key stored in the accumulator and at address 000D. Also on exiting the subroutine, the information which has been entered will be stored in locations (0000 + X + 1) and (0000 + X), in this case 0031 and 0030.

Load the following program:-

0300	A2,00	LDX#00
	8A	TXA
	95,10	* STA 10,X
	E8	INX
	E0,08	CPX#08
	D0,F9	BNE *
	A2,30	LDX#30
	95,00	STA 00,X
	95,01	STA 01,X
	20,88,FE	JSR FE88
	4C,13,03	** JMP **

Run the program from 0300 and enter hexadecimal information via the keyboard. Notice that the information enters the display. Press a command key and note that the display goes blank. Press RESET and then examine memory locations 0030 and 0031, these will contain the last four digits entered via the keyboard.

The subroutine uses the accumulator and the Y register.

Key values are:

M 10	L 14	0 00	4 04	8 08	C 0C
G 11	R 15	1 01	5 05	9 09	D 0D
P 12	+ 16	2 02	6 06	A 0A	E 0E
S 13	- 17	3 03	7 07	B 0B	F 0F

OUTPUT DATA THROUGH THE CASSETTE INTERFACE (TO TAPE)

Routine Start Address = FEB1

This subroutine takes an 8-bit parallel code from the accumulator and outputs this byte as a serial code through the cassette interface.

The following program illustrates the use of subroutine FEB1:-

0300	A2,00	LDX#00
0302	BD,00,02 *	LDA 0200,X
0305	20,B1,FE	JSR FEB1
0308	E8	INX
0309	D0,F7	BNE *
030B	4C,E0,FE	JMP FEE0

Press reset before running the program and the program will output all of page 02 through the cassette interface.

If required, the following program can be run to store AA and 55 in alternate locations on page 02. The program given above can then be used to store this data on tape.

0350	A2,00	LDX#00
0352	A9,AA	LDA#AA
0354	85,20	STA 20
0356	A5,20	LDA 20
0358	49,FF	EOR#FF
035A	85,20	STA 20
035C	9D,00,02	STA 0200,X
035F	E8	INX
0360	D0,F4	BNE F4
0362	4C,E0,FE	JMP FEE0

The subroutine at address FEB1 uses the accumulator and Y register.

INPUT DATA THROUGH THE CASSETTE INTERFACE (FROM TAPE)

Routine Start Address = FE00

Loads a byte from tape into the accumulator. In loading this byte, the subroutine takes a serial 8-bit code from the cassette interface and converts it to a parallel code, placing this parallel byte in the accumulator.

The following program illustrates the use of this subroutine:-

0300	A2,00	LDX#00
0302	20,DD,FE	JSR FEDD
0305	9D,00,02	STA 0200,X
0308	E8	INX
0309	D0,F7	BNE F7
030B	4C,FE,E0	JMP FEE0

This program loads 256 bytes of data from the tape and stores it on page 02.

If the program given in the TO TAPE routine is run first, then the FROM TAPE routine can be used to read the data back from the tape.

The subroutine uses the accumulator and Y register.

APPENDIX 4

EMMA MEMORY MAP

The memory map for an unexpanded EMMA is as follows:

0000	Reserved by the EMMA/VISA Monitor; used in conjunction with devices such as the Eprom Programmer, Matrix Printer etc.
001F	
0020	
00FF	Available as user RAM.
0100	
01FF	Designated by the 6502 as the system stack.
0200	
03FF	Available as user RAM.
0900	Decoded for use by the EMMA II input/output ports.
09FF	
0A00	Decoded for use by the keyboard/display interface.
0AFF	
0C00	User RAM 0EFC-0EFF interrupt vectors. 0E80-0EFF in use when VISA connected.
0FFF	
D800	EMMA II Monitor; cassette routines, useful sub-routines.
DOFF	
F000	Available for user EPROM expansion.
F7FF	
FE00	EMMA II monitor program.
FFFF	

DETAILED SYSTEM MEMORY MAPS

The following sheets expand on the system memory map.

EMMA/VISA USER I/O PORT (VIA)

The VIA has four Register Select inputs which decode to 16 internal registers uniquely addressable by the User. The System Address Decode Hardware places these Registers at memory locations as below:

ADDRESS	REGISTER DESIG.	WRITE	DESCRIPTION	READ
09x0	ORB/IRB	Output Register "B"		Input Register "B"
09x1	ORA/IRA	Output Register "A"		Input Register "A"
09x2	DDRB	Data Direction Register "B"		
09x3	DDRA	Data Direction Register "A"		
09x4	T1C-L	T1 Low-Order Latches		T1 Low-Order Counter
09x5	T1C-H	T1 High-Order Counter		
09x6	T1L-L	T1 Low-Order Latches		
09x7	T1L-H	T1 High-Order Latches		
09x8	T2C-L	T2 Low-Order Latches		T2 Low-Order Counter
09x9	T2C-H	T2 High-Order Counter		
09xA	SR	Shift Register		
09xB	ACR	Auxiliary Control Register		
09xC	PCR	Peripheral Control Register		
09xD	IFR	Interrupt Flag Register		
09xE	IER	Interrupt Enable Register		
09xF	ORA/IRA	Same as 09x0 except no "Handshake"		

EMMA KEYBOARD/DISPLAY

ADDRESS LINES	CONTROL REGISTER BIT		
	CRA BIT 2	CRB BIT 2	
0 A 0 0	1	X	Peripheral Register A.
0 A 0 0	0	X	Data Direction Register A.
0 A 0 1	X	X	Control Register A.
0 A 0 2	X	1	Peripheral Register B.
0 A 0 2	X	0	Data Direction Register B.
0 A 0 3	X	X	Control Register B.
X = EITHER 0 or 1			

The table shown gives the addresses of the various registers in the 6821.

HEXADECIMAL 7-SEGMENT CODE CONVERSION FONT

Memory location FFEA-FFF9 contain seven-segment codes equivalent to hexadecimal 0-F consecutively.

NON-MASKABLE INTERRUPT VECTORS

Processor fetches NMI Vectors from memory locations FFFA-FFFB. Jump to (FFAD) that is 0EFC and 0EFD (EMMA).

INTERRUPT REQUEST VECTORS

Processor fetches IRQ Vectors from memory locations FFFF-FFFF. Jump to (FFB0) that is 0EFE and 0EFF.

SYSTEM RESET VECTORS

Processor fetches RES Vectors from memory locations FFFC-FFFF. Jump to FEED (EMMA) and F800 (VISA).

APPENDIX 5

OSCILLOSCOPES

INTRODUCTION

An oscilloscope will probably be your main test instrument and as such should be treated with respect. Also, take the trouble to read the manufacturer's manual which will set out the operational details particular to the oscilloscope you may be using. This appendix will consider oscilloscopes in more general terms.

The front panel of the oscilloscope is generally divided into sections:

Tube Control comprises an ON/FF switch, beam intensity and focus control. You may also find a trim control to adjust the horizontal deflection relative to the X-axis of the graticule.

Input Control comprises two sub-sections (dual trace) each of which will possess:

- Sensitivity selector switch - this allows a suitable deflection in the Y-direction to be obtained for a range of inputs.
- Signal coupling selector - allows selection of A.C. or D.C. input and also provides a means of grounding the oscilloscope input without grounding the input signal.
- Trace position potentiometer - allows the trace to be moved vertically upwards or downwards on the screen.
- An input connection, usually provided by a BNC socket.

Finally, a means of trace control will be provided which is common to both inputs:

- Input selection - usually a switch which permits selection of either a single or dual beam.
- Input Trigger Selection - allows the trigger control to operate from either input as selected.
- Display mode selection - when used in the dual mode selection of beam sweep mode must be made. In the 'alternative' position first one input is swept across the screen followed by the second. The alternation is continuously repeated.

In the chop mode, the beam is switched or 'chopped' at a high frequency (50-100 KHz) between inputs and for a single sweep of the screen. In general, the alternative mode would be used for high frequency signals (sweep speeds > 1 ms/division) and the 'chop' mode for low frequency signals.

Time Base Control is quite comprehensive and comprising:

- Sweep time selector - a rotary switch which sets the rate at which the beam is swept across the screen. Associated with this is a means of varying the sweep rate of any given setting. Normally this control would be set to its calibrate position when the sweep times as selected should be those actually obtained.
- Trigger Source Selector - usually a slide switch which allows the following trigger sources:
 - a) Internal - the trigger source is taken from the selected signal vertical amplifier. This would be the normal mode of operation.
 - b) Line - this presents a useful trigger source for 'looking at' small signals which are related in frequency to the A.C. lower line frequency. The trigger source is taken from the oscilloscope power transformer.

c) External - used when the start of the sweep must be related to some external event. Associated with this mode will be a BNC socket to input the external trigger signal.

- Trigger level and Slope - slope control allows the trigger to be effected on a rising slope (+) or falling slope (-). Trigger level can be associated with this in that the level to which the signal must rise or fall can be set.
- Time base Magnification - usually a switch which expands the horizontal deflection by a set factor, commonly times five (x5).
- Trace Position - allows the trace to be moved horizontally across the screen.
- X-Y Control - converts one of the vertical amplifiers into a horizontal amplifier thus allowing two input signals to drive the beam in the vertical and horizontal directions respectively.

Finally, some means of calibration of vertical amplifiers may be provided. This usually takes the form of a square wave form of at some fixed voltage level and frequency.

OBTAINING A TRACE

You will probably find that upon switching the oscilloscope ON, no trace appears. There may be numerous reasons for this, but simply 'twiddling the knobs' at random is not a particularly good approach to 'finding a trace'. A much more logical approach should be used which will prove less frustrating.

- Set sensitivity selector switch to maximum input voltage. This will ensure that any further lower setting is a deliberate action on your part and may prevent damage to the oscilloscope.
- Set signal coupling selector to 'ground' position - this ensures that stray inputs at the oscilloscope input cannot affect the beam of deflection.

- Trace position potentiometer - rotate to control position. This will ensure that trace will appear at the centre of the screen.
- Sweep time selector - set to approximately middle of range.
- Trigger source selector - set to 'internal'.
- Trigger level control - set to 'automatic'.
- Intensity control - set to mid-position.

You are now in a position to switch ON. A few seconds should be allowed before appearance of trace sine although most of the circuitry is solid state the cathode ray tube relies upon thermionic emission for its operation. You should now have a trace which you can adjust for both intensity and focus to obtain a good sharp trace across the screen.

USE OF PROBES

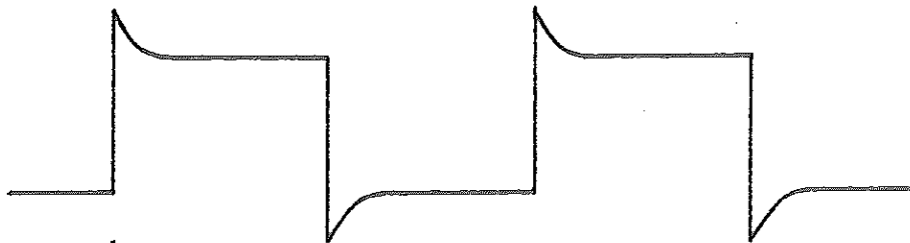
Probes are more than connecting leads between equipment under test and the oscilloscope. They allow the best possible signal to be fed to the oscilloscope and also present the least load to the circuitry being tested. The most common probes in use are X1, X10 and X100 passive probes. The X10 (times one ten) probe, for example, indicates that the signal appearing at the oscilloscope input is 1/10 that appearing at the probe tip, i.e., the signal at the point in the circuit being tested is attenuated by the probe. This means that the volts/cm selected on the sensitivity switch must be multiplied by 10 when computing the actual voltage measured.

We have already stated that the probe provides the better signal as distinct from using a 'piece of wire'. However, it must be adjusted to the oscilloscope to which it is being connected.

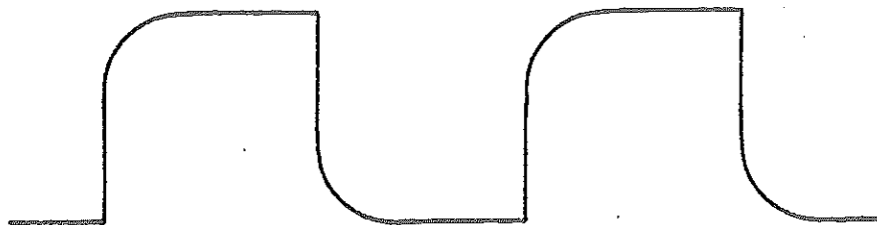
Note: The HAMEG X1 probe is matched to the HAMEG 203 oscilloscope and no adjustment is provided.

Where adjustment is provided, the object is to compensate the probe to give the best frequency response when connected to the particular oscilloscope in use. The method is as follows:

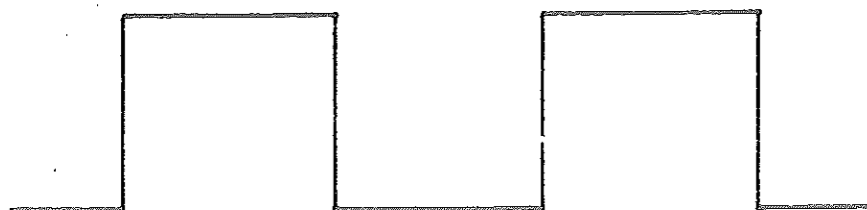
- Connect the probe outlet to the particular channel that it will be used with and the probe tip to the oscilloscope calibration output.
- Set the vertical volts/cm and the horizontal time/cm controls so that the square waveform generated by the calibration output fills most of the screen.
- Use a 'trimmer' tool to adjust the probe such that the corners of the displayed square wave are the best obtainable. Typical waveforms are as shown below:



a) too much gain at higher frequencies



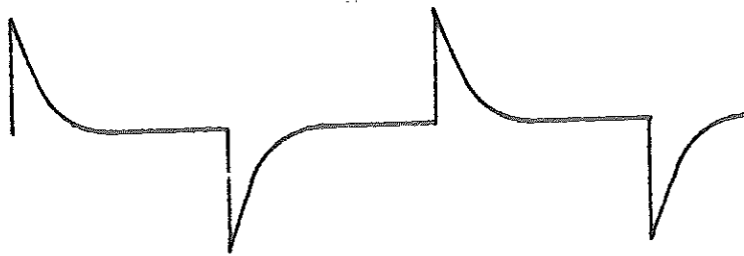
b) attenuation at higher frequencies



c) flat frequency response.

of the above c) is the waveform required

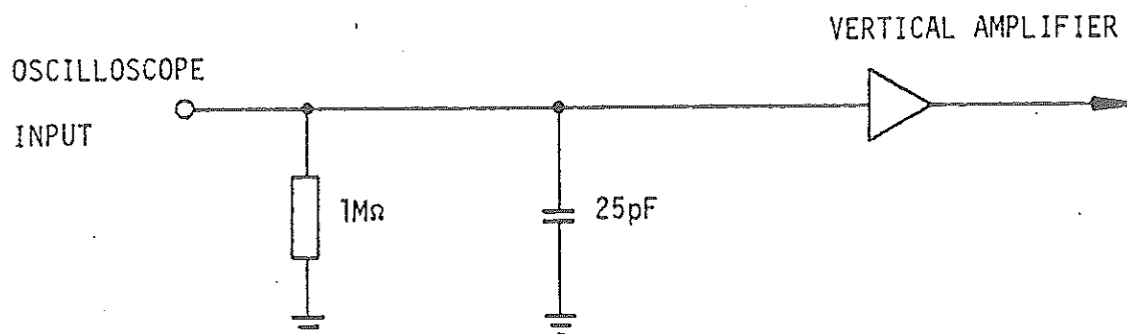
You may find that the actual response is:



d)

This suggests that the resistive element forming part of the probe is open circuit, in which case the probe differentiates the input to give the display shown.

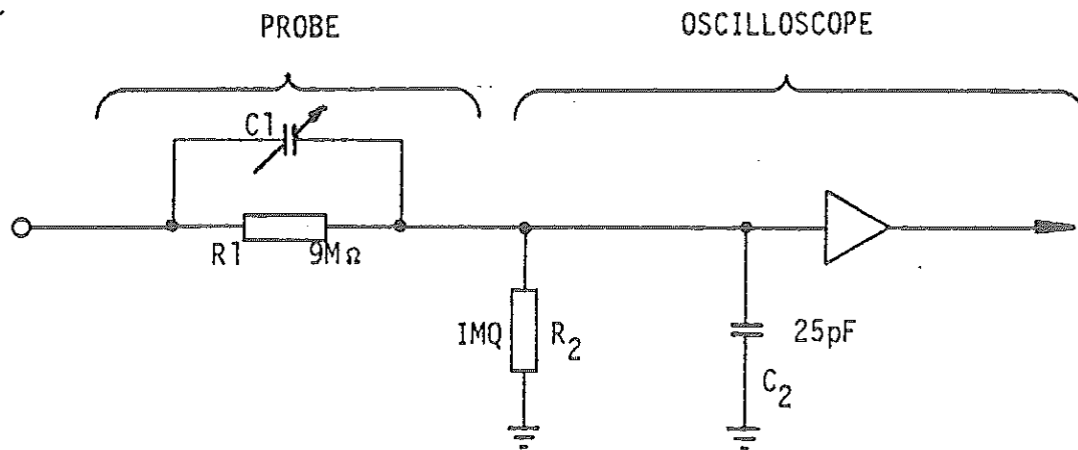
Now lets look at another use for the probe. The input impedance of a typical mid-frequency oscilloscope can be represented as a parallel resistor/capacitor combination, e.g.



If the input of the oscilloscope is connected to a logic circuit the 1M resistor may present very little problem but the capacitance may load the circuit under test such that changes in timing may result in malfunctioning.

A X10 probe can alleviate this problem.

The probe plus oscilloscope can be represented as the diagram below:



In matching the probe to the oscilloscope what you in fact did was to trim C_1 such that time constants $C_1 R_1 = C_2 R_2$.

Now, at low frequencies (say D.C.) the combination presents $10 M$ to the test piece and the oscilloscope sees only $1/10$ of the actual signal measured. As the frequency increases the oscilloscope still sees $1/10$ of the measured signal and will do so for all frequencies. However, as frequency increases the impedance of the combination will decrease from its D.C. value but its capacitive loading will always be less than approximately $2.3pF$

APPENDIX 6

6502 INSTRUCTION SET

6502 INSTRUCTION CODES

The following notation applies to this summary:

A	Accumulator
X,Y	Index Registers
M	Memory
P	Processor Status Register
S	Stack Pointer
✓	Change
-	No Change
+	Add
∧	Logical AND
-	Subtract
⊕	Logical Exclusive Or
↑	Transfer from Stack
↓	Transfer to Stack
→	Transfer to
→	Transfer to
V	Logical OR
PC	Program counter.
PCH	Program Counter High
PCL	Program Counter Low
OPER	Operand
#	Immediate Addressing Mode

ADC**Add memory to accumulator with carry****ADC**Operation: $A + M + C \rightarrow A, C$

N	Z	C	I	D	V
✓	✓	✓	-	-	✓

Addressing Mode	Assembly Language Form	OP CODE	No.of Bytes	No.of Cycles
Immediate	ADC # Oper	69	2	2
Zero Page	ADC Oper	65	2	3
Zero Page, X	ADC Oper, X	75	2	4
Absolute	ADC Oper	6D	3	4
Absolute, X	ADC Oper, X	7D	3	4*
Absolute, Y	ADC Oper, Y	79	3	4*
(Indirect, X)	ADC (Oper, X)	61	2	6
(Indirect), Y	ADC (Oper), Y	71	2	5*

* Add 1 if page boundary is crossed.

AND**"AND" memory with accumulator****AND**

Logical AND to the accumulator

Operation: $A \wedge M \rightarrow A$

N	Z	C	I	D	V
✓	✓	-	-	-	-

Addressing Mode	Assembly Language Form	OP CODE	No.of Bytes	No.of Cycles
Immediate	AND # Oper	29	2	2
Zero Page	AND Oper	25	2	3
Zero Page, X	AND Oper, X	35	2	4
Absolute	AND Oper	2D	3	4
Absolute, X	AND Oper, X	3D	3	4*
Absolute, Y	AND Oper, Y	39	3	4*
(Indirect, X)	AND (Oper, X)	21	2	6
(Indirect), Y	AND (Oper), Y	31	2	5*

* Add 1 if page boundary is crossed.

ASL**ASL Shift Left One Bit (Memory or Accumulator)****ASL**Operation: $C \leftarrow \begin{array}{|c|c|c|c|c|c|c|} \hline 7 & 6 & 5 & 4 & 3 & 2 & 1 \\ \hline \end{array} \leftarrow 0$

N	Z	C	I	D	V
✓	✓	✓	-	-	-

Addressing Mode	Assembly Language Form	Op CODE	No.of Bytes	No.of Cycles
Accumulator	ASL A	0A	1	2
Zero Page	ASL Oper	06	2	5
Zero Page, X	ASL Oper, X	16	2	6
Absolute	ASL Oper	0E	3	6
Absolute, X	ASL Oper, X	1E	3	7

BCC**BCC Branch on Carry Clear****BCC**

Operation: Branch on C = 0

 N Z C I D V
 - - - - -

Addressing Mode	Assembly Language Form	OP CODE	No.of Bytes	No.of Cycles
Relative	BCC Oper	90	2	2*

* Add 1 if branch occurs to same page.

* Add 2 if branch occurs to different page.

BCS**BCS Branch on Carry Set****BCS**

Operation: Branch on C = 1

 N Z C I D V
 - - - - -

Addressing Mode	Assembly Language Form	OP CODE	No.of Bytes	No.of Cycles
Relative	BCS Oper	B0	2	2*

* Add 1 if branch occurs to same page.

* Add 2 if branch occurs to different page.

BEQ**BEQ Branch on Result Zero****BEQ**

Operation: Branch on Z = 1

 N Z C I D V
 - - - - -

Addressing Mode	Assembly Language Form	OP CODE	No.of Bytes	No.of Cycles
Relative	BEQ Oper	F0	2	2*

* Add 1 if branch occurs to same page.

* Add 2 if branch occurs to different page.

BIT**BIT Test Bits in Memory with Accumulator****BIT**

Operation: A ← M, M7 → N, M6 → V

Bit 6 and 7 are transferred to the status register.

If the result of A ← M is zero then Z = 1, otherwise

Z = 0

 N Z C I D V
 M7 ✓ - - - M6

Addressing Mode	Assembly Language Form	OP CODE	No.of Bytes	No.of Cycles
Zero Page Absolute	BIT Oper	24	2	3
	BIT Oper	2C	3	4

BMI**BMI Branch on result minus****BMI**

Operation: branch on N = 1

N Z C I D V
- - - - -

Addressing Mode	Assembly Language Form	OP CODE	No.of Bytes	No.of Cycles
Relative	BMI Oper	30	2	2*

- * Add 1 if branch occurs to same page.
- * Add 2 if branch occurs to different page.

BNE**BNE Branch on result not zero****BNE**

Operation: Branch on Z = 0

N Z C I D V
- - - - -

Addressing Mode	Assembly Language Form	OP CODE	No.of Bytes	No.of Cycles
Relative	BNE Oper	00	2	2*

- * Add 1 if branch occurs to same page.
- * Add 2 if branch occurs to different page.

BPL**BPL Branch on result plus****BPL**

Operation: Branch on N = 0

N Z C I D V
- - - - -

Addressing Mode	Assembly Language Form	OP CODE	No.of Bytes	No.of Cycles
Relative	BPL Oper	10	2	2*

- * Add 1 if branch occurs to same page.
- * Add 2 if branch occurs to different page.

BRK**BRK Force Break****BRK**

Operation: Forced Interrupts PC + 2 ↓ P ↓

N Z C I D V
- - - 1 - -

Addressing Mode	Assembly Language Form	Op CODE	No.of Bytes	No.of Cycles
Implied	BRK	00	1	7

- * A BRK command cannot be masked by setting I.

BVC**BVC Branch on Overflow Clear****BVC**

Operation: Branch on V = 0

 N Z C I D V
 - - - - -

Addressing Mode	Assembly Language Form	OP CODE	No.of Bytes	No.of Cycles
Relative	BVC Oper	50	2	2*

* Add 1 if branch occurs to same page.

* Add 2 if branch occurs to different page.

BVS**BVS Branch on Overflow Set****BVS**

Operation: Branch on V = 1

 N Z C I D V
 - - - - -

Addressing Mode	Assembly Language Form	OP CODE	No.of Bytes	No.of Cycles
Relative	BVS Oper	70	2	2*

* Add 1 if branch occurs to same page.

* Add 2 if branch occurs to different page.

CLC**CLC Clear Carry Flag****CLC**

Operation: 0 → C

 N Z C I D V
 - - 0 - - -

Addressing Mode	Assembly Language Form	OP CODE	No.of Bytes	No.of Cycles
Implied	CLC	18	1	2

CLD**CLD Clear Decimal Mode****CLD**

Operation: 0 → D

 N Z C I D V
 - - - - 0 -

Addressing Mode	Assembly Language Form	OP CODE	No.of Bytes	No.of Cycles
Implied	CLD	D8	1	2

CLI**CLI Clear Interrupt Disable Bit****CLI**Operation: $\emptyset \rightarrow I$

N	Z	C	I	D	V
-	-	-	\emptyset	-	-

Addressing Mode	Assembly Language Form	OP CODE	No.of Bytes	No.of Cycles
Implied	CLI	58	1	2

CLV**CLV Clear Overflow Flag****CLV**Operation: $\emptyset \rightarrow V$

N	Z	C	I	D	V
-	-	-	-	-	\emptyset

Addressing Mode	Assembly Language Form	OP CODE	No.of Bytes	No.of Cycles
Implied	CLV	B8	1	2

CMP**CMP Compare Memory and Accumulator****CMP**Operation: $A - M$

N	Z	C	I	D	V
✓	✓	✓	-	-	-

Addressing Mode	Assembly Language Form	OP CODE	No.of Bytes	No.of Cycles
Immediate	CMP #Oper	C9	2	2
Zero Page	CMP Oper	C5	2	3
Zero Page, X	CMP Oper	D5	2	4
Absolute	CMP Oper	CD	3	4
Absolute, X	CMP Oper, X	DD	3	4*
Absolute, Y	CMP Oper, Y	D9	3	4*
(Indirect, X)	CMP (Oper, X)	C1	2	6
(Indirect), Y	CMP (Oper), Y	D1	2	5*

* Add 1 if page boundary is crossed.

CPX**CPX Compare Memory and Index X****CPX**Operation: $X - M$

N	Z	C	I	D	V
✓	✓	✓	-	-	-

Addressing Mode	Assembly Language Form	OP CODE	No.of Bytes	No.of Cycles
Immediate	CPX #Oper	E0	2	2
Zero Page	CPX Oper	E4	2	3
Absolute	CPX Oper	EC	3	4

CPY

CPY Compare Memory and Index Y

CPY

Operation: Y - M

 N Z C I D V
 ✓ ✓ ✓ - - -

Addressing Mode	Assembly Language Form	OP CODE	No.of Bytes	No.of Cycles
Immediate	CPY #Oper	C0	2	2
Zero Page	CPY Oper	C4	2	3
AAbsolute	CPY Oper	CC	3	4

DEC

DEC Decrement Memory by One

DEC

Operation: M - 1 → M

 N Z C I D V
 ✓ ✓ - - - -

Addressing Mode	Assembly Language Form	OP CODE	No.of Bytes	No.of Cycles
Zero Page	DEC Oper	C6	2	5
Zero Page, X	DEC Oper, X	D6	2	6
Absolute	DEC Oper	CE	3	6
Absolute, X	DEC Oper, X	DE	3	7

DEX

DEX Decrement Index X by One

DEX

Operation: X - 1 → X

 N Z C I D V
 ✓ ✓ - - - -

Addressing Mode	Assembly Language Form	OP CODE	No.of Bytes	No.of Cycles
Implied	DEX	CA	1	2

DEY

DEY Decrement Index Y by One

DEY

Operation: Y - 1 → Y

 N Z C I D V
 ✓ ✓ - - - -

Addressing Mode	Assembly Language Form	OP CODE	No.of Bytes	No.of Cycles
Implied	DEY	88	1	2

EOR**EOR "Exclusive-Or" Memory with Accumulator****EOR**Operation: $A \oplus M \rightarrow A$

N	Z	C	I	D	V
✓	✓	-	-	-	-

Addressing Mode	Assembly Language Form	OP CODE	No.of Bytes	No.of Cycles
Immediate	EOR #Oper	49	2	2
Zero Page	EOR Oper	45	2	3
Zero Page, X	EOR Oper, X	55	2	4
Absolute	EOR Oper	4D	3	4
Absolute, X	EOR Oper, X	5D	3	4*
Absolute, Y	EOR Oper, Y	59	3	4*
(Indirect, X)	EOR (Oper, X)	41	2	6
(Indirect),Y	EOR (Oper), Y	51	2	5*

* Add 1 if page boundary is crossed.

INC**INC Increment Memory by One****INC**Operation: $M + 1 \rightarrow M$

N	Z	C	I	D	V
✓	✓	-	-	-	-

Addressing Mode	Assembly Language Form	OP CODE	No.of Bytes	No.of Cycles
Zero Page	INC Oper	E6	2	5
Zero Page, X	INC Oper, X	F6	2	6
Absolute	INC Oper	EE	3	6
Absolute, X	INC Oper, X	FE	3	7

INX**INX Increment Index X by One****INX**Operation: $X + 1 \rightarrow X$

N	Z	C	I	D	V
✓	✓	-	-	-	-

Addressing Mode	Assembly Language Form	OP CODE	No.of Bytes	No.of Cycles
Implied	INX	E8	1	2

INY**INY Increment Index Y by One****INY**Operation: $Y + 1 \rightarrow Y$

N	Z	C	I	D	V
✓	✓	-	-	-	-

Addressing Mode	Assembly Language Form	OP CODE	No.of Bytes	No.of Cycles
Implied	INY	C8	1	2

JMP**JMP Jump to New Location****JMP**

Operation: $(PC + 1) \rightarrow PCL$
 $(PC + 2) \rightarrow PCH$

N Z C I D V
 - - - - -

Addressing Mode	Assembly Language Form	OP CODE	No.of Bytes	No.of Cycles
Absolute Indirect	JMP Oper	4C	3	3
	JMP (Oper)	6C	3	5

JSR**JSR Jump to New Location Saving Return Address****JSR**

Operation: $PC + 2 \downarrow$, $(PC + 1) \rightarrow PCL$
 $(PC + 2) \rightarrow PCH$

N Z C I D V
 - - - - -

Addressing Mode	Assembly Language Form	OP CODE	No.of Bytes	No.of Cycles
Absolute	JSR Oper	20	3	6

LDA**LDA Load Accumulator with Memory****LDA**

Operation: $M \rightarrow A$

N Z C I D V
 ✓ ✓ - - -

Addressing Mode	Assembly Language Form	OP CODE	No.of Bytes	No.of Cycles
Immediate	LDA #Oper	A9	2	2
Zero Page	LDA Oper	A5	2	3
Zero Page, X	LDA Oper, X	B5	2	4
Absolute	LDA Oper	AD	3	4
Absolute, X	LDA Oper, X	BD	3	4*
Absolute, Y	LDA Oper, Y	B9	3	4*
(Indirect, X)	LDA (Oper, X)	A1	2	6
(Indirect), Y	LDA (Oper), Y	B1	2	5*

Add 1 if page boundary is crossed.

LDX**LDX Load Index X With Memory****LDX**

Operation: $M \rightarrow X$

N Z C I D V
 ✓ ✓ - - -

Addressing Mode	Assembly Language Form	OP CODE	No.of Bytes	No.of Cycles
Immediate	LDX #Oper	A2	2	2
Zero Page	LDX Oper	A6	2	3
Zero Page, Y	LDX Oper, Y	B6	2	4
Absolute	LDX Oper	AE	3	4
Absolute, Y	LDX Oper, Y	BE	3	4*

* Add 1 if page boundary is crossed

LDY
Operation: M → Y

LDY Load Index Y With Memory

LDY
N Z C I D V
✓ ✓ - - - -

Addressing Mode	Assembly Language Form	OP CODE	No.of Bytes	No.of Cycles
Immediate	LDY #Oper	A0	2	2
Zero Page	LDY Oper	A4	2	3
Zero Page, X	LDY Oper, X	B4	2	4
Absolute	LDY Oper	AC	3	4
Absolute, X	LDY Oper, X	BC	3	4*

* Add 1 when page boundary is crossed

LSR LSR Shift Right One Bit (memory or accumulator)

Operation: 0 ←

7	6	5	4	3	2	1	0
---	---	---	---	---	---	---	---

 ← C

N Z C I D V
0 ✓ ✓ - - -

Addressing Mode	Assembly Language Form	OP CODE	No.of Bytes	No.of Cycles
Accumulaator	LSR A	4A	1	2
Zero Page	LSR Oper	46	2	5
Zero Page, X	LSR Oper, X	56	2	6
Absolute	LSR Oper	4E	3	6
Absolute, X	LSR Oper, X	5E	3	7

NOP NOP No Operation

Operation: No Operation (2 cycles)

N Z C I D V
- - - - -

Addressing Mode	Assembly Language Form	OP CODE	No.of Bytes	No.of Cycles
Implied	NOP	EA	1	2

ORA ORA "OR" Memory With Accumulator

Operation: A V M → A

N Z C I D V
✓ ✓ - - - -

Addressing Mode	Assembly Language Form	OP CODE	No.of Bytes	No.of Cycles
Immediate	ORA #Oper	09	2	2
Zero Page	ORA Oper	05	2	3
Zero Page, X	ORA Oper, X	15	2	4
Absolute	ORA Oper	0D	3	4
Absolute, X	ORA Oper, X	1D	3	4*
Absolute, Y	ORA Oper, Y	19	3	4*
(Indirect, X)	ORA (Oper, X)	01	2	6
(Indirect), Y	ORA (Oper), Y	11	2	5*

PHA
Operation: A ↓

PHA Push Accumulator on Stack

N Z C I D V
- - - - -

PHA

Addressing Mode	Assembly Language Form	OP CODE	No.of Bytes	No.of Cycles
Implied	PHA	48	1	3

PHP
Operation: P ↓

PHP Push Processor Status on Stack

N Z C I D V
- - - - -

PHP

Addressing Mode	Assembly Language Form	OP CODE	No.of Bytes	No.of Cycles
Implied	PHP	08	1	3

PLA
Operation: A ↑

PLA Pull Accumulator from Stack

N Z C I D V
✓✓ - - -

PLA

Addressing Mode	Assembly Language Form	OP CODE	No.of Bytes	No.of Cycles
Implied	PLA	68	1	4

PLP
Operation: P ↑

PLP Pull Processor Status from Stack

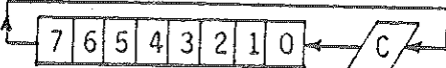
N Z C I D V
From Stack

PLP

Addressing Mode	Assembly Form	OP CODE	No.of Bytes	No.of Cycles
Implied	PLP	28	1	4

ROL **ROL Rotate One Bit Left (memory or accumulator)**

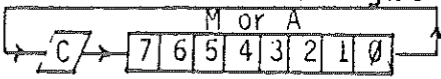
ROL

Operation: 

N Z C I D V
✓✓✓ - - -

Addressing Mode	Assembly Language Form	OP CODE	No.of Bytes	No.of Cycles
Accumulator	ROL A	2A	1	2
Zero Page	ROL Oper	26	2	5
Zero Page, X	ROL Oper, X	36	2	6
Absolute	ROL Oper	2E	3	6
Absolute, X	ROL Oper, X	3E	3	7

ROR ROR Rotate One Bit Right (memory or accumulator) ROR

Operation:  N Z C I D V
✓ ✓ ✓ - - -

Addressing Mode	Assembly Language Form	OP CODE	No.of Bytes	No.of Cycles
Accumulator	ROR A	6A	1	2
Zero Page	ROR Oper	66	2	5
Zero Page, X	ROR Oper, X	76	2	6
AAbsolute	ROR Oper	6E	3	6
Absolute, X	ROR Oper, X	7E	3	7

RTI RTI Return from Interrupt RTI

Operation: P↑ PC↑ N Z C I D V
From Stack

Addressing Mode	Assembly Language Form	OP CODE	No.of Bytes	No.of Cycles
Implied	RTI	40	1	6

RTS RTS Return from Sub-Routine RTS

Operation: PC↑, PC + 1 → PC N Z C I D V
- - - - -

Addressing Mode	Assembly Language Form	OP CODE	No.of Bytes	No.of Cycles
Implied	RTS	60	1	6

SBC SBC Subtract Memory from Accumulator with Borrow SBC

Operation: A - M - \bar{C} → A N Z C I D V
Note: \bar{C} = Borrow ✓ ✓ ✓ - -

Addressing Mode	Assembly Language Form	OP CODE	No.of Bytes	No.of Cycles
Immediate	SBC #Oper	E9	2	2
Zero Page	SBC Oper	E5	2	3
Zero Page, X	SBC Oper, X	F5	2	4
Absolute	SBC Oper	ED	3	4
Absolute, X	SBC Oper, X	FD	3	4*
Absolute, Y	SBC Oper, Y	F9	3	4*
(Indirect, X)	SBC (Oper, X)	E1	2	6
(Indirect), Y	SBC (Oper), Y	F1	2	5*

* Add 1 when page boundary is crossed.

SEC

SEC Set Carry Flag

SEC

Operation: $1 \rightarrow C$

N Z C I D V
- - 1 - - -

Addressing Mode	Assembly Language Form	OP CODE	No. of Bytes	No. of Cycles
Implied	SEC	38	1	2

SED

SED Set Decimal Mode

SED

Operation: $1 \rightarrow D$

N Z C I D V
- - - - 1 -

Addressing Mode	Assembly Language Form	OP CODE	No. of Bytes	No. of Cycles
Implied	SED	F8	1	2

SEI

SEI Set Interrupt Disable Status

SEI

Operation: $1 \rightarrow I$

N Z C I D V
- - - 1 - -

Addressing Mode	Assembly Language Form	OP CODE	No. of Bytes	No. of Cycles
Implied	SEI	78	1	2

STA

STA Store Accumulator in Memory

STA

Operation: $A \rightarrow M$

N Z C I D V
- - - - - -

Addressing Mode	Assembly Language Form	OP CODE	No. of Bytes	No. of Cycles
Zero Page	STA Oper	85	2	3
Zero Page, X	STA Oper, X	95	2	4
Absolute	STA Oper	8D	3	4
Absolute, X	STA Oper, X	9D	3	5
Absolute, Y	STA Oper, Y	99	3	5
(Indirect, X)	STA (Oper, X)	81	2	6
(Indirect), Y	STA (Oper), Y	91	2	6

STX
Operation: $X \rightarrow M$

STX Store Index X in Memory

STX
N Z C I D V
- - - - -

Addressing Mode	Assembly Language Form	OP CODE	No.of Bytes	No.of Cycles
Zero Page	STX Oper	86	2	3
Zero Page, Y	STX Oper, Y	96	2	4
Absolute	STX Oper	8E	3	4

STY
Operation: $Y \rightarrow M$

STY Store Index Y in Memory

STY
N Z C I D V
- - - - -

Addressing Mode	Assembly Language Form	OP CODE	No.of Bytes	No.of Cycles
Zero Page	STY Oper	84	2	3
Zero Page, X	STY Oper, X	94	2	4
Absolute	STY Oper	8C	3	4

TAX
Operation: $A \rightarrow X$

TAX Transfer Accumulator to Index X

TAX
N Z C I D V
✓ ✓ - - - -

Addressing Mode	Assembly Language Form	OP CODE	No.of Bytes	No.of Cycles
Implied	TAX	AA	1	2

TAY
Operation: $A \rightarrow Y$

TAY Transfer Accumulator to Index Y

TAY
N Z C I D V
✓ ✓ - - - -

Addressing Mode	Assembly Language Form	OP CODE	No.of Bytes	No.of Cycles
Implied	TAY	A8	1	2

TSX
Operation: $S \rightarrow X$

TSX Transfer Stack Pointer to Index X

TSX
N Z C I D V
✓ ✓ - - - -

Addressing Mode	Assembly Language Form	OP CODE	No.of Bytes	No.of Cycles
Implied	TSX	BA	1	2

TXA

TXA Transfer Index X to Accumulator

TXA

Operation: $X \rightarrow A$

N Z C I D V
 ✓ ✓ - - - -

Addressing Mode	Assembly Language Form	OP CODE	No.of Bytes	No.of Cycles
Implied	TXA	8A	1	2

TXS

TXS Transfer Index X to Stack Pointer

TXS

Operation: $X \rightarrow S$

N Z C I D V
 - - - - -

Addressing Mode	Assembly Language Form	OP CODE	No.of Bytes	No.of Cycles
Implied	TXS	9A	1	2

TYA

TYA Transfer Index Y to Accumulator

TYA

Operation: $Y \rightarrow A$

N Z C I D V
 ✓ ✓ - - - -

Addressing Mode	Assembly Language Form	OP CODE	No.of Bytes	No.of Cycles
Implied	TYA	98	1	2

	ACCUMULATOR	IMMEDIATE	ZERO PAGE	ZERO PAGE, X	ZERO PAGE, Y	ABSOLUTE	ABSOLUTE, X	ABSOLUTE, Y	IMPLIED	RELATIVE	(INDIRECT, X)	(INDIRECT, Y)	ABSOLUTE INDIRECT
ADC	0A	69	65	75		6D	7D	79			61	71	
AND		29	25	35		2D	3D	39			21	31	
ASL			06	16		0E	1E						
BCC										90			
BCS										B0			
BEQ										F0			
BIT			24			2C							
BMI										30			
BNE										D0			
BPL										10			
BRK	4A								00				
BVC										50			
BVS										70			
CLC									18				
CLD									D8				
CLI									58				
CLV									B8				
CMP		C9	C5	D5		CD	DD	D9			C1	D1	
CPX		E0	E4			EC							
CPY		C0	C4	D6		CC	DE						
DEC	2A		C6			CE							
DEX									CA				
DEY									88				
EOR		49	45	55		4D	5D	59			41	51	
INC			E6	F6		EE	FE						
INX									E8				
INY									C8				
JMP						4C							6C
JSR						20							
LDA		A9	A5	B5	B6	AD	BD	B9			A1	B1	
LDX	6A	A2	A6	B4		AE	BE	BE					
LDY		A0	A4	B4		AC	BC						
LSR			46	56		4E	5E						
NOP									EA				
ORA		09	05	15		0D	1D	19			01	11	
PHA									48				
PHP									08				
PLA									68				
PLP									28				
ROL			26	36		2E	3E						
ROR	2A		66	76		6E	7E						
RTI									40				
RTS									60				
SBC		E9	E5	F5		ED	FD	F9			E1	F1	
SEC													
SED									38				
SEI									F8				
STA			85	95		8D	9D	99			81	91	
STX			86		96	8E							
STY			84	94		8C							

TAX TAY TYA TSX TXA TXS		ACCUMULATOR	
		IMMEDIATE	
		ZERO PAGE	
		ZERO PAGE, X	
		ZERO PAGE, Y	
		ABSOLUTE	
		ABSOLUTE, X	
		ABSOLUTE, Y	
	AA A8 98 8A 8A 9A	IMPLIED	
		RELATIVE	
		(INDIRECT, X)	
		(INDIRECT, Y)	
		ABSOLUTE INDIRECT	

NUMERICAL LISTING

00 - BRK	30 - BMI	60 - RTS
01 - ORA - (Indirect,X)	31 - AND - (Indirect,X)	61 - ADC - (Indirect,X)
02 - Future Expansion	32 - Future Expansion	62 - Future Expansion
03 - Future Expansion	33 - Future Expansion	63 - Future Expansion
04 - Future Expansion	34 - Future Expansion	64 - Future Expansion
05 - ORA - Zero Page	35 - AND - Zero Page,X	65 - ADC - Zero Page
06 - ASL - Zero Page	36 - ROL - Zero Page,X	66 - ROR - Zero Page
07 - Future Expansion	37 - Future Expansion	67 - Future Expansion
08 - PHP	38 - SEC	68 - PLA
09 - ORA - Immediate	39 - AND - Absolute,Y	69 - ADC - Immediate
0A - ASL - Accumulator	3A - Future Expansion	6A - ROR - Accumulator
0B - Future Expansion	3B - Future Expansion	6B - Future Expansion
0C - Future Expansion	3C - Future Expansion	6C - JMP - Indirect
0D - ORA - Absolute	3D - AND - Absolute,X	6D - ADC - Absolute
0E - ASL - Absolute	3E - ROL - Absolute,X	6E - ROR - Absolute
0F - Future Expansion	3F - Future Expansion	6F - Future Expansion
10 - BPL	40 - RTI	70 - BVS
11 - ORA - (Indirect),Y	41 - EOR - (Indirect,X)	71 - ADC - (Indirect),Y
12 - Future Expansion	42 - Future Expansion	72 - Future Expansion
13 - Future Expansion	43 - Future Expansion	73 - Future Expansion
14 - Future Expansion	44 - Future Expansion	74 - Future Expansion
15 - ORA - Zero Page,X	45 - EOR - Zero Page	75 - ADC - Zero Page,X
16 - ASL - Zero Page,X	46 - LSR - Zero Page	76 - ROR - Zero Page,X
17 - Future Expansion	47 - Future Expansion	77 - Future Expansion
18 - CLC	48 - PHA	78 - SEI
19 - ORA - Absolute,Y	49 - EOR - Immediate	79 - ADC - Absolute,Y
1A - Future Expansion	4A - LSR - Accumulator	7A - Future Expansion
1B - Future Expansion	4B - Future Expansion	7B - Future Expansion
1C - Future Expansion	4C - JMP - Absolute	7C - Future Expansion
1D - ORA - Absolute,X	4D - EOR - Absolute	7D - ADC - Absolute,X
1E - ASL - Absolute,X	4E - LSR - Absolute	7E - ROR - Absolute,X
1F - Future Expansion	4F - Future Expansion	7F - Future Expansion
20 - JSR	50 - BVC	80 - Future Expansion
21 - AND - (Indirect,X)	51 - EOR - (Indirect),X	81 - STA - (Indirect,X)
22 - Future Expansion	52 - Future Expansion	82 - Future Expansion
23 - Future Expansion	53 - Future Expansion	83 - Future Expansion
24 - BIT - Zero Page	54 - Future Expansion	84 - STY - Zero Page
25 - AND - Zero Page	55 - EOR - Zero Page,X	85 - STA - Zero Page
26 - ROL - Zero Page	56 - LSR - Zero Page,X	86 - STX - Zero Page
27 - Future Expansion	57 - Future Expansion	87 - Future Expansion
28 - PLP	58 - CLI	88 - DEY
29 - AND - Immediate	59 - EOR - Absolute,Y	89 - Future Expansion
2A - ROL - Accumulator	5A - Future Expansion	8A - TXA
2B - Future Expansion	5B - Future Expansion	8B - Future Expansion
2C - BIT - Absolute	5C - Future Expansion	8C - STY - Absolute
2D - AND - Absolute	5D - EOR - Absolute,X	8D - STA - Absolute
2E - ROL - Absolute	5E - LSR - Absolute,X	8E - STX - Absolute
2F - Future Expansion	5F - Future Expansion	8F - Future Expansion

90 - BCC	C0 - CPY - Immediate	F0 - BEQ
91 - STA - (Indirect),Y	C1 - CMP - (Indirect,X)	F1 - SBC - (Indirect),
92 - Future Expansion	C2 - Future Expansion	F2 - Future Expansion
93 - Future Expansion	C3 - future Expansion	F3 - Future Expansion
94 - STY - Zero Page,X	C4 - CPY - Zero Page	F4 - Future Expansion
95 - STA - Zero Page,X	C5 - CMP - Zero Page	F5 - SBC - Zero Page,X
96 - STX - Zero Page,Y	C6 - DEC - Zero Page	F6 - INC - Zero Page,X
97 - Future Expansion	C7 - Future Expansion	F7 - Future Expansion
98 - TYA	C8 - INY	F8 - SED
99 - STA - Absolute,Y	C9 - CMP - Immediate	F9 - SBC - Absolute,Y
9A - TXS	CA - DEX	FA - Future Expansion
9B - Future Expansion	CB - Future Expansion	FB - Future Expansion
9C - Future Expansion	CC - CPY - Absolute	FC - Future Expansion
9D - STA - Absolute,X	CD - CMP - Absolute	FD - SBC - Absolute,X
9E - Future Expansion	CE - DEC - Absolute	FE - INC - Absolute,Y
9F - Future Expansion	CF - Future Expansion	FF - Future Expansion
A0 - LDY - Immediate	D0 - BNE	
A1 - LDA - (Indirect,X)	D1 - CMP - (Indirect),Y	
A2 - LDX - Immediate	D2 - Future Expansion	
A3 - Future Expansion	D3 - Future Expansion	
A4 - LDY - Zero Page	D4 - Future Expansion	
A5 - LDA - Zero Page	D5 - CMP - Zero Page,X	
A6 - LDX - Zero Page	D6 - DEC - Zero Page,X	
A7 - Future Expansion	D7 - Future Expansion	
A8 - TAY	D8 - CLD	
A9 - LDA - Immediate	D9 - CMP - Absolute,Y	
AA - TAX	DA - Future Expansion	
AB - Future Expansion	DB - Future Expansion	
AC - LDY - Absolute	DC - Future Expansion	
AD - LDA - Absolute	DD - CMP - Absolute,X	
AE - LDX - Absolute	DE - DEC - Absolute,X	
AF - Future Expansion	DF - Future Expansion	
B0 - BCS	E0 - CPX - Immediate	
B1 - LDA - (Indirect),Y	E1 - SBC - (Indirect,X)	
B2 - Future Expansion	E2 - Future Expansion	
B3 - Future Expansion	E3 - Future Expansion	
B4 - LDY - Zero Page,X	E4 - CPX - Zero Page	
B5 - LDA - Zero Page,X	E5 - SBC - Zero Page	
B6 - LDX - Zero Page,Y	E6 - INC - Zero Page	
B7 - Future Expansion	E7 - Future Expansion	
B8 - CLV	E8 - INX	
B9 - LDA - Absolute,Y	E9 - SBC - Immediate	
BA - TSX	EA - NOP	
BB - Future Expansion	EB - Future Expansion	
BC - LDY - Absolute,X	EC - CPX - Absolute	
BD - LDA - Absolute,X	ED - SBC - Absolute	
BE - LDX - Absolute,Y	EE - INC - Absolute	
BF - Future Expansion	EF - Future Expansion	

MICROPROCESSOR INSTRUCTION SET ALPHABETICAL SEQUENCE

ADC Add Memory to Accumulator with Carry
AND "AND" Memory with Accumulator
ASL Shift Left One Bit (Memory to Accumulator)
BCC Branch on Carry Clear
BCS Branch on Carry Set
BEQ Branch on Result Zero
BIT Test Bits in Memory with Accumulator
BMI Branch on Result Minus
BNE Branch on Result not Zero
BPL Branch on Result Plus
BRK Force Break
BVC Branch on Overflow Clear
BVS Branch on Overflow Set
CLC Clear Carry Flag
CLD Clear Decimal Mode
CLI Clear Interrupt Disable Bit
CLV Clear Overflow Flag
CMP Compare Memory and Accumulator
CPX Compare Memory and Index X
CPY Compare Memory and Index Y
DEC Decrement Memory by One
DEX Decrement Index X by One
DEY Decrement Index Y by One
EOR "Exclusive Or" Memory with Accumulator
INC Increment Memory by One
INX Increment Index X by One
INY Increment Index Y by One
JMP Jump to New Location
JSR Jump to New Location Saving Return Address
LDA Load Accumulator with Memory
LDX Load Index X with Memory
LDY Load Index Y with Memory
LSR Shift Right One Bit (Memory or Accumulator)
NOP No Operation
ORA "OR" Memory with Accumulator
PHA Push Accumulator to Stack
PHP Push Processor Status to Stack
PLA Pull Accumulator from Stack
PLP Pull Processor Status from Stack
ROL Rotate One Bit Left (Memory or Accumulator)
ROR Rotate One Bit Right (Memory or Accumulator)
RTI Return from Interrupt
RTS Return from Subroutine
SBC Subtract Memory from Accumulator with Borrow
SEC Set Carry Flag
SED Set Decimal Mode
SEI Set Interrupt Disable Status
STA Store Accumulator in Memory
STX Store Index X in Memory
STY Store Index Y in Memory
TAX Transfer Accumulator to Index X
TAY Transfer Accumulator to Index Y
TSX Transfer Stack Pointer to Index X
TXA Transfer Index X to Accumulator
TXS Transfer Index X to Stack Pointer
TYA Transfer Index Y to Accumulator

MICROPROCESSOR INSTRUCTION SET - ALPHABETIC SEQUENCE

INSTRUCTION ADDRESSING MODES AND RELATED EXECUTION TIMES (in clock cycles)													
	ACCUMULATOR	IMMEDIATE	ZERO PAGE	ZERO PAGE, X	ZERO PAGE, Y	ABSOLUTE	ABSOLUTE, X	ABSOLUTE, Y	IMPLIED	RELATIVE	(INDIRECT, X)	(INDIRECT, Y)	ABSOLUTE INDIRECT
ADC	2	2	3	4		4	4*	4*			6	5*	
AND		2	3	4		4	4*	4*			6	5*	
ASL			5	6		6	7						
BCC										2**			
BCS										2**			
BEQ										2**			
BIT			3			4							
BMI										2**			
BNE										2**			
BPL										2**			
BRK													
BVC										2**			
BVS										2**			
CLC									2				
CLD	2								2				
CLI									2				
CLV									2				
CMP		2	3	4		4	4*	4*			6	5*	
CPX		2	3			4							
CPY		2	3			4							
DEC			5	6		6	7						
DEX									2				
DEY									2				
EOR		2	3	4		4	4*	4*			6	5*	
INC			5	6		6	7						
INX									2				
INY									2				
JMP	2					3							5
JSR						6							
LDA		2	3	4		4	4*	4*			6	5*	
LDX		2	3		4	4		4*					
LDY		2	3	4		4	4*						
LSR			5	6		6	7						
NOP									2				
ORA		2	3	4		4	4*	4*			6	5*	
PHA									3				
PHP									3				
PLA									4				
PLP									4				

* Add one cycle if indexing across page boundary.

** Add one cycle if branch is taken. Add one if branching operation crosses page boundary.

INSTRUCTION ADDRESSING MODES AND RELATED EXECUTION TIMES (in clock cycles)													
	ACCUMULATOR	IMMEDIATE	ZERO PAGE	ZERO PAGE, X	ZERO PAGE, Y	ABSOLUTE	ABSOLUTE, X	ABSOLUTE, Y	IMPLIED	RELATIVE	(INDIRECT, X)	(INDIRECT, Y)	ABSOLUTE INDIRECT
ROL	2		5	6		6	7						
ROR	2		5	6		6	7						
RTI									6				
RTS									6				
SBC		2	3	4		4	4*	4*			6	5*	
SEC									2				
SED									2				
SEI									2				
STA			3	4		4	5	6			6	6	
STX			3		4	4							
STY			3	4		4							
TAX									2				
TAY									2				
TSX									2				
TXA									2				
TXS									2				
TYA									2				

* Add one cycle if indexing across page boundary.

** Add one cycle if branch is taken. Add one additional if branching operation crosses page boundary.

APPENDIX 7

MICROCOMPUTER GLOSSARY

A

Absolute Address

The unique address of a memory location as distinct from a relative address which may not be determined until required by the program execution.

Access Time

The time interval between when a stable memory address is set up on an address bus to when the data in that memory appears stable on the data bus.

Accumulator

A register associated with the arithmetic/logic unit of a microprocessor. It is used as an intermediate data memory during data movements and arithmetic/logical operations.

Address

A unique identifier given to a memory location or a device register which allows access to that memory/register only by a microprocessor.

Address Field

The portion of a microcomputer instruction which specifies either the address of the operand or data necessary to derive that address.

Addressing Modes

See memory addressing modes.

Algorithm

A statement of procedure used to solve a problem or perform a task. Algorithms are implemented on microcomputers by a series of instructions.

Arithmetic-Logic Unit (ALU)

A part of a microprocessor identified by function which executes arithmetic and logical operations.

Architecture

The organisational structure of a microcomputing system. Generally, applied specifically to the microprocessor or input/output interface devices.

ASCII Code

The acronym for American Standard Code for Information Interchange. Used extensively for data transmission.

Assembler

A computer program which changes a symbolic assembly language source program into an executable object (binary coded) program.

Assembly Language

A human-orientated symbolic mnemonic source language used by a microcomputer programmer to encode programs.

Assembly Listing

A printed listing of a machine code program assembled by an assembler and from an assembly language program.

B

Basic

A programming language. Acronym for Beginner's All-Purpose Symbolic Instruction Code.

Baud

A unit of serial data transmission rate. It approximates to the number of bits transmitted per second but includes 'start' and 'stop' bits.

Benchmark

A point of reference. A program written in a given language may be used as a benchmark or means whereby the relative performance of two or more systems can be compared.

Bi-Directional

Generally refers to bus structures where data or control signals can be transmitted in both directions. As opposed to uni-directional.

Binary Number System

A number system whose base is 2 and expresses all its quantities using two symbols only e.g. 0 and 1.

Bi-Stable Latch

A simple flip-flop which can be enabled so as to store a logical one or logical zero. Commonly used in memory or register circuits to store individual bits.

Bit

A binary digit. The smallest unit of data in a microcomputer system.

Bit Parallel

A means whereby all bits of a group of bits may be simultaneously transferred from one point to another using a group (bus) of wires. One wire for each bit.

Bit Serial

A means whereby a group of bits are transferred from one point to another, bit by bit and using a single wire. The transmission takes place to a strict format.

Block

A data group of consecutive words, bytes, characters or bits which are usually handled as a single entity.

Block Diagram

A chart which outlines the functional relationship between identifiable parts of the hardware of a system

Branch Instruction

An instruction which, when executed, may cause a program to branch (jump) to an instruction other than the next sequential one normally executed. The branch may be conditional upon the status of a register or unconditional. Synonymous with Jump.

Break Point

A point in a program at which the program execution can be halted to allow inspection of registers, input of data or generally check program operation. Frequently used in program debug routines.

Bus

A common communication path between two or more devices within a system. Usually defined by function.

Bus Driver

A device which amplifies a bus signal to ensure a sufficiently strong signal is received at destination.

Byte

A set of binary digits (Bits), usually eight, which are operated upon as a single entity. A computer word may comprise a number of bytes.

C

Carry

The transfer of a value from one lower order bit position to the next higher order bit position when the number system base is exceeded at the lower order bit position.

Character

A symbol belonging to a set of alphabetic, numeric and/or graphic symbols. Each character is represented in the computer memory by a unique binary code.

Chip

An integrated electronic circuit.

Clock

A device which generates regular timing pulses to which all operations of a system can be synchronised.

Cobol

A programming language. Acronym for Common Business - Oriented Language. Characterised by a high degree of machine independence.

Code

A system of characters which may be used to represent information and capable of being understood by a microcomputer.

Compiler

A program used to convert another program written in a high level language such as COBOL into an assembly or machine language program.

CMOS

Abbreviated for "Complementary Metal Oxide Semi-Conductor". These semi-conductor devices are characterised by low power dissipation, moderate speed of operation and high density of integration.

Conditional Jump

See Branch Instruction

Configuration

The arrangement of hardware and/or software making up a system.

Control Character

A character whose occurrence may initiate, modify or halt an operation.

Counter

A device such as a register or a memory location which can be used to record a number of events. Counters are usually incremented, decremented, set to a value or cleared.

Crosstalk

Undesirable electrical signals between adjacent channels of a data transmission system.

Crowbar

A protective circuit used on power supply systems to reduce the output voltage to ground when an over voltage condition occurs.

Current Loop

A communication method whereby the presence or absence of a current is used to represent transmitted data. 20 milli-amperes is typical.

Cycle Time

The time period required by a micro-processor to read from or write to a system memory. It is often used to measure a microprocessor performance.

D

Daisy Chain

A method of propagating signals along a bus system. Devices not requesting a daisy-chained signal respond by passing the signal on. The first device requesting the signal responds by breaking the signal continuity. The system permits device priorities according to electrical position along the chain.

Data Acquisition

The function of obtaining data from external sources, converting into binary format for processing by a microcomputer system.

Data Buffer Register

A temporary storage device which allows transmission of data between devices capable of handling data at different rates.

Data Pointer

A register holding the memory address of data to be used by an instruction. The register 'points' to the memory location of data.

Data Reduction

The process of transforming masses of raw text or experimental data into a useful or more readily understood form.

Debug

To eliminate programming mistakes.

Debug Programs

Programs which help a programmer to find errors in his programs.

Decoder

A logic device which converts data from one number system into another.

Decrement

To reduce a numerical constant by one.

Device

A unit of hardware capable of performing a given function.

Device Register

An addressable register used to store status, control information or data for transfer to or from a device.

Diagnostic Programs

Programs which check the operation of specific system hardware for correct operation.

Diode Transistor Logic (DTL)

A family of integrated circuit logic using both diodes and transistors. They are characterised by medium speed, low power dissipation, high drive capability and low cost.

Disc Storage

A method of bulk storage of data and programs onto a disc of magnetic material. It is characterised by the ability to randomly access portions of selected data.

Dual-in-line (DIL)

Most common method of packaging integrated circuits. Circuit leads or pins extend symmetrically outwards and downwards from opposite ends of the rectangular package.

Dynamic Memory

A type of semi-conductor memory in which the presence or absence of a capacitive charge represents the state of a binary storage element. The charge must be periodically refreshed.

E

Editor

A program which manipulates text material and so facilitates the preparation of source programs.

Enable

A single condition which permits a specific processing event to take place.

Encode

A process whereby individual characters are represented in a coded format. Usually consisting of binary numbers.

Exclusive OR function

A logical operation in which the result is logically true only when one input is true and false when both inputs are true or false.

Execute

To run a program or perform a specific computer instruction.

External Device

A unit of processing equipment external to that of the microprocessing unit.

F

Fetch

The action of obtaining an instruction from a stored program and decoding that instruction. Also refers to that portion of a computer's instruction cycle when that action is performed.

Firmware

A computer program (software) that is implemented in hardware, such as a ROM.

Fixed-point Arithmetic

Arithmetic in which the decimal point that separates the integer and fractional portions of numerical expressions is either explicitly stated for all expressions or is fixed with respect to the first or last digit of each expression.

Flag

Usually a single binary bit used to give an indication of state or condition. Flags can be implemented by both software or hardware.

Flip-Flop

A logical device which has two stable states.

Floating-point Arithmetic

Arithmetic in which the decimal point is defined as a power of ten and all exponents are equalised prior to any operations.

Flow-chart

A graphical representation of the processing steps performed by a computer program or a sequence of logical steps performed by hardware.

Format

Orderly structured arrangement of data elements (bits, character, bytes or fields) which form a larger entity.

Fortran

A computer high-level language. Science oriented. Derived from FORMula TRANslator.

Fusible Link

A type of programmable read-only memory integrated circuit whose memory is formed by 'burning' open by means of a current to form a logical '0'.

G

Gate

A logic element which has two or more inputs and one output. The output state is dependent upon the logic states of the inputs. It is generally described by means of a truth table.

General Register

An internal addressable register which is used for temporary storage. Example the accumulator, index register.

H

Handshaking

The sequencing of signals for communication between asynchronous system devices.

Hard copy

A printed output as opposed to a volatile display on a Video terminal.

Hardware

Physical equipment forming a computer system.

Hard-wire logic

A group of logic circuits permanently interconnected to perform a specific function.

Hexadecimal

A number system to base sixteen. Used to encode four binary digits.

High Level Language

A computer language which uses symbols conveniently read by the programmer. Examples BASIC, FORTRAN, COBOL.

I

Immediate Addressing

A method of addressing an instruction in which the operand is inherent in the instruction or in the memory location immediately following.

Immediate Data

Data which follows immediately an instruction in memory.

Indexed Addressing

A method of addressing in which the address part of an instruction is modified by the contents of an index register.

Index Register

A register which contains a quantity which may be used to modify memory address.

Indirect addressing

A means of addressing in which the address of the operand is specified by an auxiliary register or memory location specified by the instruction rather than by bits in the instruction itself.

Input/Output (I/O)

General term for the equipment used to communicate with a computer CPU; or the data involved in that communication.

Integrated Circuit (IC)

A solid-state microcircuit consisting of interconnected active and passive semiconductor devices diffused into a single silicon chip.

Instruction

A set of bits that defines a computer operation, and is a basic command understood by the CPU. It may move data, do arithmetic and logic functions, control I/O devices, or make decisions as to which instruction to execute next.

Instruction cycle.

The process of fetching an instruction from memory and executing it.

Instruction Length

The number of words needed to store an instruction. It is one word in most computers, but some will use multiple words to form one instruction. Multiple-word instructions have different instruction execution times depending on the length of the instruction.

Instruction set.

The set of general-purpose instructions available with a given computer. In general, different machines have different instruction sets.

Instruction time

The time required to fetch an instruction from memory and then execute it.

Interpreter

A program which fetches and executes 'instructions' (pseudo-instructions) written in a higher-level language. The higher-level language program is a pseudo-program. Contrast with compiler.

Interrupt latency

The delay between an interrupt request and acknowledgement of the request.

Interrupt request

A signal to the computer that temporarily suspends the normal sequence of a routine and transfers control to a special routine. Operation can be resumed from this point later. Ability to handle interrupts is very useful in communication applications where it allows the microprocessor to service many channels.

Interrupt mask

A mechanism which allows the program to specify whether or not interrupt requests will be accepted.

Interrupt service routine

A routine (program) to properly store away on the stack the present status of the machine in order to respond to an interrupt request; perform the 'real work' required by the interrupt; restore the saved status of the machine; and then resume the operation of the interrupted program.

Interrupt vector

Typically, two memory locations assigned to an interrupting device and containing the starting address and processor status word for its service routine.

I/O Interface

The control electronics required to interface an I/O device to a computer CPU. The power and use of a CPU is very closely associated with the range of I/O devices which can be connected to it. One cannot usually simply plug them into the CPU. The I/O control electronics will do the matchmaking. The complexity and cost of the control electronics are very much determined by both the hardware and software I/O architecture of the CPU.

I/O Port

A connection to a CPU which is configured (or programmed) to provide a data path between the CPU and the external devices, such as keyboard, display, reader, etc. An I/O port of a microprocessor may be an input port or an output port, or it may be bi-directional.

J

Jump

A departure from the normal one-step incrementing of the program counter. By forcing a new value (address) into the program counter the next instruction can be fetched from an arbitrary location (either further ahead or back).

L

Large-scale integration (LSI)

High density integrated circuits for complex logic functions. LSI circuits can range up to several thousand transistor type circuits per mm^2 of silicon chip.

Library

A collection of standard or frequently used routines and subroutines.

Loop

A self-contained series of instructions in which the last instruction can cause repetition of the series until a terminal condition is reached. Branch instructions are used to test the conditions in the loop to determine if the loop should be continued or terminated.

M

Machine cycle

The basic CPU cycle. In one machine cycle an address may be sent to memory and one word (data or instruction) read or written, or, in one machine cycle a fetched instruction can be executed.

Machine Language

The numeric form of specifying instructions ready for loading into memory and execution by the machine. This is the lowest-level language in which to write programs. The value of every bit of every instruction in the program must be specified (e.g. by giving a string of binary, octal, or hexadecimal digits for the contents of each word in memory).

Macro (macroinstruction)

A symbolic source language statement which is expanded by the assembler into one or more machine language instructions, relieving the programmer of having to write out frequently occurring instruction sequences.

Medium-scale integration (MSI)

A medium-density integrated circuit, containing logic functions more complex than small-scale integration but less complex than large-scale integration. Most 4-bit counters, latches, and data multiplexers are considered MSI devices.

Memory

That part of a computer which holds data and instructions. Each instruction or data is assigned a unique address which is used by the CPU when fetching or storing the information.

Memory address register

The CPU register which holds the address of the memory location being accessed.

Memory addressing modes

The method of specifying the memory location of an operand. Common addressing modes are: direct, immediate, relative, indexed, and indirect. These modes are important factors in program efficiency.

Memory cycle

The operations required for addressing, reading, writing and/or reading and writing data in memory.

Memory Map

A listing of addresses or symbolic representations of addresses which define the boundaries of the memory address space occupied by a program or a series of programs. Memory maps can be produced by a highlevel language such as FORTRAN.

Microcomputer

A computer whose processing unit is a microprocessor. A microcomputer is an entire system with microprocessor, memory and input-output controllers.

Microprocessor

A single LSI circuit which performs the functions of a CPU. Some characteristics of a microprocessor include small size, inclusion in a single integrated circuit or a set of integrated circuits, and low cost.

Mnemonic code

Computer instructions written in brief, easy-to-learn, symbolic or abbreviated form. Mnemonic code is also recognisable by the assembly program. For example, ADD, SUB, CLR and MOV are mnemonic codes for instructions which will be executed as machine code.

Monitor

A program, typically part of a larger operating system, which provides a uniform method of program timing, scheduling, and handling of input/output tasks.

Monolithic integrated circuit

An electronic circuit formed within a single small chip of crystalline semiconductor material, usually silicon. Typically, the chip is contained in a plastic or ceramic package.

Electrical connections to package leads are made by fine wire which is welded to metal pads on the chip and to the package leads.

N

Nesting

A programming technique in which a segment of a larger program is executed iteratively (looping) until a specific data condition is detected, or until a predetermined number of interactions has been performed. The nesting level is the number of times nesting can be repeated.

Nibble

A sequence of 4 bits operated upon as a unit. Also see byte.

Nonvolatile memory

A type of computer system memory offering preservation of data storage during power loss or system shutdown. Magnetic core read/write memory systems are typically nonvolatile, and, therefore, do not require reloading to restore programs and data when system power is applied.

O

Object program

The binary form of a source program produced by an assembler or a compiler. The object program is composed of machine-coded instructions that the computer can execute.

Online system

A system of I/O devices in which the operation of such devices is under the control of the CPU and in which information reflecting current activity is introduced into the data processing or controlling system as soon as it occurs.

Operand

Any of the quantities arising out of or resulting from the execution of a computer instruction. An operand can be an argument, a result of computation, a constant, a parameter, the address of any of these quantities, or the next instruction to be executed.

Operating System

A structured set of software routine whose function is to control the execution sequence of programs running on a computer, supervise the input/output activities of these programs, and support the development of new programs through such functions as assembly, compilation, editing and debugging.

Operation code (op-code)

That part of a computer instruction word which designates the function performed by a given instruction. For example the op-code for the arithmetic instruction Addition (with carry) is ADC.

Overflow

A condition occurring in a computer when the results of a mathematical operation produce a result which has a magnitude exceeding the capacity of the computer's word size.

P

Page

A natural grouping of memory locations by higher-order address bits. In an 8-bit microprocessor, $2^8 = 256$ consecutive bytes often may constitute a page. Then words on the same page differ in the lower-order 8 address bits.

Parity check

A method of checking the correctness of binary data after that data has been transferred from or to storage. An additional bit, called the parity bit, is appended to the binary word or character to be transferred. The parity bit is the single-digit sum of all the binary digits in the word or character, and its logical state can be assigned to represent either an even or an odd number of 1's making up the binary word. Parity is checked in the same manner in which it is generated.

Peripheral device

A general term designating various kinds of machine which operate in combination or conjunction with a computer but are not physically part of the computer. Peripheral devices typically display computer data, store data from the computer and return the data to the computer on demand, prepare data for human use, or acquire data from a source and convert it to a form usable by a computer. Peripheral devices include printers, keyboards, graphic display terminals, paper-tape reader/punches, analog-digital converters, discs and tape drives.

Pointer

Registers in the CPU which contain memory addresses. See program counter and data pointer.

Polling

A process in which a number of peripheral devices, remote stations, or nodes in a computer network are interrogated one at a time to determine if service is required.

Port

An input or output route for transferring data or information to or from a system.

Position-independent code (PIC)

Machine-coded programs using only relative addressing, permitting the programs to reside in any portion of system memory.

Power-fail circuit

A logic circuit that protects an operating program if primary power fails. A typical power-fail circuit informs the computer when power failure is imminent, initiating a routine that saves all volatile data. After power has been restored, the circuit initiates a routine that restores the data and restarts computer operation.

Priority

The sequence in which various entries and tasks are processed or peripheral devices are serviced. Priorities are based on analysis of codes associated with an entry or task, or the positional assignment of a peripheral device within a group of devices.

Processor status work (PSW)

A special-purpose CPU register which contains the status of the most recent instruction execution result, trap bit, and interrupt priority.

Program

A complete sequence of computer instructions necessary to solve a specific problem, perform a specific action, or respond to external stimuli in a prescribed manner.

Program Counter

A CPU register which specifies the address of the next instruction to be fetched and executed. Normally it is incremented automatically each time an instruction is fetched.

Programmable read-only memory (PROM)

A read-only memory which can be programmed after manufacture by external equipment. PROMs are generally integrated circuits, with each memory cell connected to assert a logic 1. The fusible link connecting a cell can be disconnected (burned open) to produce a logic 0.

Push-down stacks

Dedicated consecutive temporary storage registers in a computer, sometimes part of system memory structured so that the data items retrieved are the most recent items stored on the stack.

R

Random-access memory (RAM)

A computer memory structured so that the time required to access any data item stored in memory is the same as for any other item.

Read-write cycle

The sequence of operations required to read and write (restore) memory data.

Read Time

A computation or process by a computer using inputs derived from time-initiated events; the output resulting from the computation or processing can have an effect on and/or predict trends concerning those events.

Real-time clock

A timing device used by a computer to derive elapsed time between events and to control processing of time-initiated event data.

Re-entrant code

The instructions forming a single copy of a program or subroutine which is shared by two or more programs, as opposed to the conventional method of embedding a copy of a subroutine within each program. Characteristically, re-entrant routines are composed completely of instructions and constants which are not subject to modification during execution.

Register

A temporary storage unit which can be implemented as a hardware device or as a software structure and used to store data for manipulation and/or processing reference. Typically, a register consists of a single computer word or a portion of a word.

Relative address

An address of a machine instruction which is referred to as origin address. For example, consider the relative address 15 which is translated into the absolute address origin $R + 15$ where R is typically, the contents of the PC register. Relative addressing allows the generation of position-independent code.

Relocatable

Object programs that can reside in any part of system memory. The actual starting address is established at load time by adding a relocation offset to the starting address. Relocatable code is typically composed of position-independent code.

Resident software

Assembler and editor programs incorporated with a prototyping system to aid in user program writing and development; see software.

Response time

The time between the initiation of an operation from a computer terminal and the receipt of results at the terminal. Response time includes transmission of data to the computer processing, file, access and transmission of results to the terminal.

ROM read-only memory (fixed memory)

Is any type of memory which cannot be readily re-written; ROM requires a masking operation during production to permanently record program or data patterns in it. The information is stored on a permanent basis and used repetitively. Such storage is useful for programs or tables of data that remain fixed and is usually randomly accessible.

Routine

Usually refers to a subprogram, i.e. the task performed by the routine is less complex. A program may include routines; see program.

Run time

The time required to complete a single, continuous execution of an object program.

S

Scratch-pad memory

RAM or registers which are used to store temporary intermediate results (data) or memory addresses (pointers).

Semi-conductor memory

A memory with storage elements formed by integrated semiconductor devices, as opposed to a memory composed of ferrite cores.

Semiconductor read/write memories are characterised by low-cost, wide speed ranges, and data volatility. Semiconductor read-only memories are non-volatile.

Serial I/O

A method of data transfer between a computer and a peripheral device in which data is transmitted for input to the computer (or output to the device) bit by bit over a single circuit.

Serial memory (serial access memory)

Any type of memory in which the time required to read from or write into the memory is dependent on the location in the memory. This type of memory has to wait while non-desired memory locations are accessed. Examples are paper tape, disc, magnetic tape, CCD etc. In a random-access memory, access time is constant.

Service routine

A set of instructions to perform a programmed operation, typically in response to an interrupt.

Shift register

A register in which binary data bits are moved as a contiguous group a prescribed number of positions to the right or to the left.

Single-operand instruction

An instruction containing a reference to one register, memory location, or device.

Small-scale integration

The earliest form of integrated circuit technology, a typical SSI circuit contains 1-4 logic circuits.

Software

Programs which control the operation of computer hardware. Operating systems, executives, monitors, compilers, editors, utility routines and user programs are considered software.

Software documentation

Program listing and/or technical manuals describing the operation and use of programs.

Sort

A function performed by a program, usually part of a utility package; items in a data file are arranged or re-arranged in a logical sequence designated by a key word or field in each item in the file.

Source program

A program, in either hard copy or stored form, written in language (source language) other than machine language which requires translation by the assembler, compiler, or interpreter program.

Stack pointer

The counter, or register, used to address a stack in the memory; see stack.

Stand-alone system

A microcomputer software development system which runs on a microcomputer without connection to another computer or a time sharing system. This system includes an assembler, editor, and debugging aids. It may include some of the features of a prototyping kit.

Static memory

A type of semiconductor read/write random-access memory which does not require periodic refresh cycles.

Subroutine

A subprogram (group of instructions) reached from more than one place in a main program. The process of passing control from the main program to a subroutine is a subroutine call, and the mechanism is a subroutine linkage. Often data or data addresses are made available by the main program to the subroutine. The process of returning control from subroutine to main program is subroutine return. The linkage automatically returns control to the original position in the main program or to another subroutine: see nesting.

Synchronous operation

Use of a common timing source (clock) to time circuits or data transfer operations (contrast with asynchronous operation).

Syntax

Formal structure. The rules governing sentence structure in a language, or statement structure in a language such as assembly language or FORTRAN.

T

Truth Table

A listing which presents all possible input and output states of a logic function.

V

Variable

A symbol or mnemonic whose value changes from the execution of one program to another or during execution of a single program.

Vector

See interrupt vector

Volatile memory

Refers to a read/write memory whose content is irretrievably lost when operating power is removed. Virtually all types of read/write semiconductor memories are volatile.

W

Word

A set of binary bits handled by the computer as the primary unit of information. The length of a computer word is determined by the hardware design. Typically, each system memory location contains one word.

Word length

The number of bits in the computer word. The longer the word length, the greater the precision (number of significant digits). In general, the longer the word length, the richer the instruction set, and the more varied the addressing mode.

Write

The process of storing data in a memory.

APPENDIX 8

SOLUTIONS TO QUESTIONS

SOLUTIONS TO QUESTIONS

CHAPTER 1.1

1. A binary digit. The smallest unit of data in a microcomputer system.
2. A group of binary digits (bits), usually eight, which are operated on as a single entity. A computer word may comprise a number of bytes.
3. Data comprising alphabetic and numeric characters.
4. 32
5. (a) 13, (b) 57, (c) 240
6. (a) 0010 0110 (b) 01111000 (c) 11001001

CHAPTER 1.2

1. (a) 1110, (b) 1000, (c) 1010, (d) 0010
2. +127 to -127
3. 11011010
4. 1 00011001 = 25

CHAPTER 1.3

1. An overflow occurs during a calculation if there is a carry from bit 6 to the sign bit or from the sign bit to the carry bit.

2. (a)

```
+120    01111000
- 25    11100111
        1 01011111
```

v = 0 c = 1 NO OVERFLOW

(b)

```
+83     01010011
+52     00110100
        0 10000111
```

v = 1 c = 0 OVERFLOW

(c)

```
+ 8     00001000
-23     11101001
        0 11110001
```

v = 0 c = 0 NO OVERFLOW

3. +32767 0111 1111 1111 1111
 -255 1111 1111 0000 0001
 1 0111 1111 0000 0000

= 32512

CHAPTER 1.4

1. American Standard Code for Information Interchange.

2. (a) 1010001 (b) 0110011
 (c) 0101011 (d) 1110110

3. (a) Odd parity bit = 0
 (b) Odd parity bit = 1
 (c) Even parity bit = 0
 (d) Even parity bit = 1

CHAPTER 1.5

1. (a) 26_8 (b) 357_8
2. (a) 53_{16} (b) $B3_{16}$
3. (a) 319_{10} (b) 524_{10}

CHAPTER 2.1

1. The accumulator is a general purpose register used for temporary storage of data during data transfer or during arithmetic or logical operations.
2. The Arithmetic Logic Unit (ALU) is used to perform arithmetic and logical operations.
3. The address of the next instruction.
4. Bidirectional.
5. Devices which can output data on to the data bus have a high impedancestate so that they do not corrupt data on the bus when it is being used by other devices.

CHAPTER 2.2

1. (a) Random Access Memory.
(b) Read Only Memory.
(c) Erasable Programmable Read Only Memory.
2. (a) RAM
(b) All three.
3. I/O port is allocated a unique address and is accessed in the same way as an ordinary memory location.

4. Operation codes (Op Codes) are determined by the manufacturer of the microprocessor and each consists of one byte which specifies the operation to be performed. There may be one or two further bytes associated with the op code specifying data or an address location.
5. Transfer the contents of memory location M into the accumulator.
6. The microprocessor fetches the next op code (instruction) to be executed.

CHAPTER 2.3

1.
 - (a) A code is used rather than a written statement.
 - (b) The programmer must have an intimate knowledge of the microprocessor for which the program is being written
 - (c) A lot of instructions are required to perform a simple task.
2. A low level language because we can accurately predict the timing of programs written in this way.

CHAPTER 2.4

1.
 - (a) A5
 - (b) F0
 - (c) 48
 - (d) 8D
2. 4 μs.
3. Execution of the instruction may change this flag in the status register.

CHAPTER 3.1

1. 2046 Bytes
2. 1764 Bytes
3. 4
4. Less memory space is required and instructions are executed in a shorter period of time.
5. The system stack.

CHAPTER 3.3

1. 2
2. 3
3.
 - (a) A9
 - (b) 8D
 - (c) 8E
 - (d) F0
 - (e) 7D
4.
 - (a) 0D
 - (b) ED
5.
 - (a) The result of the last operation in the arithmetic unit was negative.
 - (b) The result of the last operation overflowed.
 - (c) The result of the last operation was zero.

CHAPTER 3.4

1. 0028
2. 0205

3. Transfer the contents of locations 0224 - 0224 to location 0300 - 0304.
4. Backwards.
5. F6.

CHAPTER 3.5

1. 32_{16}
2. $F2_{16}$ ($= -14_{10}$)
3. 74_{16}
4. 56
5. 2381

CHAPTER 3.6

1. 0101
2. 1111 0110
3. 1010 0011
4. 0101 1110

CHAPTER 3.7

1. Jump to sub-routine.
2. Return from sub-routine.
3. (a) 20
(b) 60

CHAPTER 3.8

1. 256 Bytes.
2. An 8-bit register which holds the address of the next 'empty' location in the stack.
3. (a) PHA , PHP
(b) PLA , PLP

CHAPTER 3.9

1. 4 Cycles.
2. 813 μ s.

CHAPTER 3.10

1. A technique by which each device, which can interrupt the processor, is interrogated to determine the interrupting device.
2. FFFF and FFFE
3. FFFB and FFFA
4. $\overline{\text{NMI}}$ has higher priority than $\overline{\text{IRQ}}$.

CHAPTER 3.11

1. 0900
2. PB_7 and PB_4
3. Interrupt Enable Register and Interrupt Flag Register.
4. 0909