beginning machine code on the commodore 64

a simple introduction for beginners

david lawrence & mark england

beginning machine code on the commodore 64

a simple introduction for beginners

david lawrence & mark england

First published 1985 by:
Sunshine Books (an imprint of Scot Books Ltd.)
12–13 Little Newport Street
London WC2H 7PP

British Library Cataloguing in Publication Data
Lawrence, David, 1949–
Beginning machine code on the Commodore 64: a simple introduction for beginners
1. Commodore 64 (Computer)—Programming
2. Machine Codes (Electronic computers)
I. Title II. England, Mark
001.64'24 QA76.8.C64

ISBN 0 946408 73 4

Cover design by Grad Graphic Design Ltd.
Illustration by Stuart Hughes.
Typeset by Paragon Photoset, Aylesbury.
Printed in England by Short Run Press Ltd, Exeter.

CONTENTS

Contents in detail

Introduction

This is not a book for machine code buffs. One or two of them may look at it and smile at the naivety of some of the examples, they may even see a few explanations that interest them, but by and large they will pass by on their way to weightier tomes.

This is a book for people who do not understand machine code, but want to. In the past we have enjoyed the challenge of writing books of complex code to stretch the 64 to its limits. When we came to think about our next book together we began to perceive a different need.

We began to see that there were many, many people who were lingering on the fringes of machine code, eager to start but somehow unable to come to terms with the concepts and, more especially, the jargon. As we looked around the books available, we could understand why. Even those which set out to give simple explanations seemed to become quickly bogged down in the impenetrable language of the machine code programmer. It seems that people who write programs made up mostly of numbers are not always at their strongest when it comes to communicating with words.

All this seemed to us rather sad, because when all is said and done, though complex machine code programs can be written, machine code itself is extremely simple. It consists of a relatively small number of instructions which can be carried out employing a relatively small range of numbers. We decided to write a book to demonstrate just how easy machine code is.

The book was to have been a relatively simple task, because we both know the subject and already have written on it. It was to have been the kind of project that could be squeezed in between other commitments. In the end it has taken as much or more thought, and as much or more time than any of the books we have previously written.

We think the effort has been worthwhile, because we think that if you follow the book through you too will understand machine code. Not only that, you will have entered a great deal of simple machine code and tested almost every aspect of the behaviour of the 6510 chip which runs the Commodore 64. We do not claim that you will be able to write complex programs in machine code, but then there are plenty of books to teach you how to do that. Our hope is that you will no longer be afraid of it, which is much the more important step.

CHAPTER 1
A Plain Man's Guide to the 6510

The problem with machine code is that everything that is needed to make it work comes wrapped up in tiny black boxes with 40 legs, called chips. Because you can't see what is going on inside them, and because the results of their activities can be exceedingly complex, it is automatically assumed that they themselves must be almost impossible to understand and very difficult to give instructions to in the form of a program.

For someone who is setting out to design a chip, that is quite true. All kinds of technical considerations have to be taken into account as a chip is built up. Automatic functions of which the user is never aware may take months of design work and correction. Microscopic components must run in step with each other while working at unbelievable speeds.

But that is all quite irrelevant to someone who is starting out in machine code, the language in which a chip can be programmed. Just as you do not need to be able to design a car in order to drive it, so you do not need to be able to design a chip in order to be able to program it. And just as driving a car is simple once you get into the right habits and begin to get a feel for how things work together, so programming a chip is just a matter of getting a feel for the way a limited number of actions work together to produce a desired result.

If you are inclined to be wary of machine code programming, perhaps it would help to remind you just how limited are the actions that the 6510 chip which lies at the heart of the Commodore 64 can perform. Here is the impressive list:

(i) It has three storage places in which it can keep numbers in the range 0 to 255.
(ii) It has 7 storage places in which it can keep a record of simple yes/no decisions.
(iii) It can pick up values from memory and put them into one of its three storage places.
(iv) It can add together two numbers.
(v) It can multiply or divide by two.
(vi) It can understand 56 simple instructions to perform combinations of the above and can be told to find those instructions in the memory of the computer.

And that, basically, is it. Of course, the limited number of functions are dressed up in a variety of guises. And equally obviously, to achieve anything useful in programming the chip, relatively long lists of instructions are going to have to be written. That doesn't alter the fact that the basics are very simple, even crude, and that no-one need be daunted by them.

To conquer machine code, all that you need to do is to bear in mind the 6 functions listed above and begin to understand:

(i) How to move numbers in and out of the three storage places inside the chip.
(ii) How to use the 7 yes/no records to keep a check on what the chip is doing.
(iii) How to specify the places in memory where numbers are to be found or placed.
(iv) The variety of ways in which addition can be used — including using addition to perform subtraction (Oh yes you can!)
(v) How to multiply or divide merely by shifting the digits of numbers from side to side.
(vi) The names of the 56 instructions and how to persuade the chip to execute them in the order you want.

The Lawland 10000 chip

To help you in your task, we are going to begin this book, not with lists of machine code but with the description of a simple chip which bears some resemblance to the 6510 on which your 64 is based. Fortunately, however, the chip we are going to describe is even simpler than the 6510 and, to make things even easier, instead of working with binary numbers, which we shall examine later, it works in good old decimal, like the rest of us. The name of this wonder-chip is the Lawland 10000, designed by two giants of computing not unknown to the authors of this book. As yet there are no working examples of the chip, but at least it can be described in detail, and we can begin to learn some of the ways in which a chip can operate.

A working diagram of the internal architecture of the Lawland 10000 chip is shown in **Figure 1.1**.

Don't worry if at first you don't recognise all the parts. In this chapter we shall work through the chip's main functions and the need for everything will become a little clearer. To follow the chapter effectively, you might find it helpful to make some copies of the diagram, into which you can pencil numbers which are being moved around or manipulated. The diagram is designed so that for the main storage places you can write in

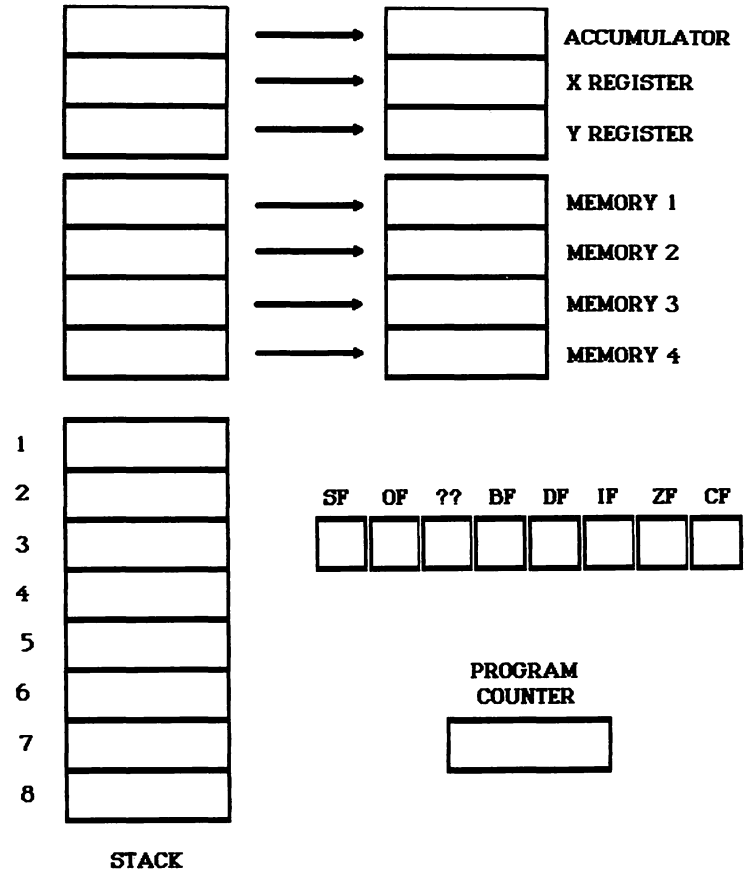

Figure 1.1

values, and then in the second set of spaces write the results of whatever operation you are performing.

The structure of the chip

Our imaginary chip has, as you can see, a very simple structure, even though we have added in some features that are not strictly part of the chip itself but part of the larger computer of which the chip is the heart.

a) The registers. These are memory locations within the chip itself, each of which is capable of storing a single number in the range 0–9999, in other words any number which can be made up of 4 digits or less. Any work which the chip does will normally be done on a number within one of these registers and any arithmetic will normally be done only on a number

contained in the register known as the 'accumulator'. So that the different abilities of the registers can be used flexibly, the chip will understand instructions to transfer numbers directly between the registers, though to move from the X register to the Y register, a number will have to be shuttled through the accumulator.

b) Memory. The majority of the numbers on which the chip will work are not held by the chip itself, but in other areas of the computer of which it forms a part. These areas are called memory because while the computer is switched on, anything which is placed into them will be 'remembered', and can be recalled. Memory is made up of numbered locations, each location being able to hold a number in the range 0–9999, like the registers in the chip. Because they are not part of the chip itself, the chip finds it more difficult to make use of the numbers stored in memory locations. Normally, in order to do anything with or to a number stored in the memory, the chip will copy that number into a register. The majority of the work of the chip will consist of taking numbers from the memory, doing something to them and then placing them back into the memory. Our imaginary chip is supplied with four memory locations to use.

c) Flags. Earlier we said that the 6510 chip is supplied with 7 locations which it can use to store simple yes/no answers on things which are currently happening within the chip. In fact it has eight, though one of them is not used. These are called 'flags' and the Lawland 10000 flags are represented by the eight small boxes on the right.

d) Program counter. Later in our explanation of the Lawland 10000 we shall see that it is capable of taking instructions from a program held in the memory. The program will be in the form of numbers, but these numbers will represent clearly identifiable orders to the chip. The job of the program counter is to record for the chip where the next instruction is to come from.

e) Stack. Every desk needs a notepad, the kind of place where messages can be recorded and then discarded, quick calculations done or facts noted down quickly before they are forgotten. For the Lawland 10000 this notepad is known as the stack. In fact the stack is a part of memory but it is used in a special way, rather like the spikes for papers which you see on some desks. When looking at the stack, the chip can only ever see what is on the top. If items are stored on the stack in the order 1, 2, 3, than only the 3 will be visible at the end of the process and to get the information back the chip will ask for the items in the order 3, 2, 1. Because the stack is a part of memory, all that can be stored on it is numbers in the range 0–10000.

Programming the Lawland 10000

Now that you have a grasp of the basic structure of the chip, we can get on with doing something. The rest of this chapter will be devoted to a series of simple exercises in programming our imaginary chip, and the kind of instructions it can understand. If you have been able to copy the diagram of the chip, you many find it helpful to work with the diagram plus a pencil and eraser.

Specifying a place in memory

To program the chip we have first to be able to tell it where it is to find the numbers it is to work on, i.e. their 'addresses' in memory. This might seem trivial when we only have four memory locations to begin with, but in fact it is probably the most difficult aspect of machine code, especially for beginners.

When we come to the 6510 chip itself we shall see that it is capable of understanding 13 different ways of specifying the address at which a number is to be found. In this simple introduction we shall consider only a few possibilities which will serve to give you some idea of the range of what can be meant by the term 'addressing mode'. In each case, the purpose is to specify a number upon which some action is to be performed, the only difference is where the chip will expect to look for that number.

a) Building the address into the command: this is the simplest form of instruction. The chip is, for instance, able to move a number from the X register to the accumulator. To make it do this, all we have to do is issue a single command — transfer X to accumulator. Since the chip knows all about its registers, it needs nothing else set out. Remember that numbers can be transferred from either the X or the Y registers to the accumulator, or vice-versa, but you cannot transfer between the X and Y registers directly.

b) Specifying the number in the command: here the command would take the form, 'load the number 123 into the accumulator'. Any of the three registers can be loaded in this way.

c) Specifying an address in memory at which a number can be found: suppose, for instance that memory location one contained the number 123. We would issue an instruction in the form 'load the X register with the contents of memory location one'. All three registers can be loaded in this way.

d) Taking one address in memory from another address: This one is more complicated. Let us suppose that we have the number 1 in memory location 4 and leave the 123 we used in the last section in memory location 1. We now issue an instruction of the form: 'Pick the address of a number from memory location 4 and then pick up the number at the address you have been given and load it into the Y register'. The result of this is that the chip looks first at memory location 4. From there it takes a number, which in our case happens to be one. It treats the 1 as an address and immediately takes the number from that address. The address it finds is, of course, 1, so it takes the contents of memory location 1, the number 123, and loads that number into the Y register.

e) Specifying a position in a block: suppose that someone were to give you directions to a house in an unfamiliar town. The directions run as follows: 'Go along this road until you come to the fifth turning on the right, it's the fourth house on the left along that road'. Nothing terribly difficult about that: you have been given a clearly recognisable point to start counting and then a number of houses to count from that point. The chip provides you with a method of addressing which is very similar. Try this for an example: the X register contains 1 and memory location 2 contains 234. We issue the instruction: 'Load into the accumulator the number which is found at an address equal to 1 plus the contents of the X register'. The result of adding the contents of the X register to 1 is 2, so the chip takes the 234 in memory location 2 and loads it into the accumulator.

These are all the addressing modes available on the Lawland 10000, and indeed though the 6510 has a few more you will recognise when we come to them that they are all based on what is given above.

Using the stack

The simplest way to see whether you understand what the stack is about is to set you a test. Imagine that the four memory locations 1–4 contain the numbers 1–4, but in the reverse order, so that location 1 contains four and so on. Your task is to use the accumulator and the stack to put the items into the right order, with 1 in memory location 1 and so on. You already know what the chip is capable of doing in terms of transferring numbers between the registers and memory, all you need to know in addition is that a number can be transferred from the accumulator to the stack and vice-versa. If each transfer between any two locations is one instruction, it should take you 16 instructions to reverse the order of the numbers in the memory.

The solution, if you need it, is to load each of the memory locations from 1 to 4 into the accumulator in turn, then place them on the stack. Then simply take them back from the stack in turn and replace them in

memory, but start placing them at location 4 and work downwards. The task is, of course, trivial and there are quicker ways of accomplishing it, but it is a useful example of the stack as a notepad and the way in which the stack reverses the order of items.

Why flags?

The eight little boxes on your chip diagram represent the flags which, as has already been pointed out, store yes/no decisions about the tasks the chip is undertaking. We shall not go into the details of all the flags here, since it would duplicate material which has to be included later in the book, but one or two illustrations may help to give you an idea of why flags are essential.

a) Numbers out of the range of the chip: Let's suppose that our chip has an instruction to add a number to what is in the accumulator. Such a command would clearly have a large number of uses in real life, but it does bring with it some problems. Our Lawland 10000 chip is limited to recording numbers made up of no more than 4 digits — anything else is literally impossible. So what happens if the number in the accumulator is 9999 and we tell the chip to add one to it. Faithful to every command, the chip adds one and comes up with the result '0000'. Of course, the answer should be 10000 but the chip can only cope with the first four digits, reading from the right — the result is a nonsense, and a nonsense which could have serious repercussions when it comes to more complex forms of arithmetic. This is where the flags, or at least one of them, comes in. The CF box stands for the 'carry flag' and it indicates whether, in the process of carrying out the last command, a number was created which was out of the range with which the chip can normally deal. In this case, if we had looked at the carry flag after carrying out the addition, it would have contained 1, meaning there was a carry, and we would immediately have known that the true result of the addition was not 0000 but 10000, and could have acted accordingly.

b) Testing for a zero result: Detecting a result of zero is an important part of programming. It might be that we use one of the registers to count a number of activities. We set up the number of times we want to repeat an action in one of the registers and then, each time the action is carried out, we subtract one from the total in the register. Eventually it reaches zero and the task has been executed the correct number of times. But how do we know that the register has reached zero? The answer again is a flag, this time ZF or the zero flag. This is designed so that if the result of the last operation carried out on any of the registers or memory locations resulted in a zero being created — our addition in the last example, for instance

would have set the zero flag to one to indicate 'yes, the result of that last action was a zero'.

We shall leave the other five flags (one of the eight is not used) for the moment, though later on you will find that they are vital parts of machine code programming. For the moment it is enough if you understand that a flag is an indicator which answers a specific question: if the answer is yes, then the flag will have the value 1, otherwise the flag will contain zero.

The form of machine code instructions

Up to now we have very loosely used phrases like 'we issue an instruction to the chip', while avoiding the very important question of *how* we issue such instructions.

The answer is that we use some memory to store a program. Machine code programs are made up of numbers, so programming in machine code consists of placing numbers into memory in the right order. In the tables spread throughout this book, you will find precise details of the numbers you will need to use to issue commands to a 6510 chip, but for the moment we shall imagine some simple instructions for the Lawland 10000 to perform some of the tasks we have just examined.

a) Transferring a number from memory to a register: Let's suppose that our task is to load the accumulator with the contents of the X register — what form will this instruction take? If you look back at the discussion of addressing modes you will find that strictly, we do not need an address for this so, in designing the Lawland 10000 we have built in a single command which tells the chip to take whatever is in the X register and place it in the accumulator. The chip is designed to recognise the number 1000 as being this specific command — one number, and one number only is required.

b) Transferring a specified number to the accumulator: The next task is to load the accumulator with the number 123. Clearly we cannot get away with a single number instruction here. The chip is going to need one number which it will recognise as the instruction to load the accumulator with a number (we have chosen 2000), and then another number which is the value to be loaded. To issue this instruction to the chip, we therefore send it two numbers: 2000,123. In designing the chip we will have had to ensure that when it comes across command 2000, it knows that it must expect another number following which is the number to be placed into the accumulator.

c) Loading a number from memory into the accumulator: A different challenge is presented by the task of loading the contents of memory location 1 into the accumulator. Once again, we are going to need to issue

the instruction in two parts, the first part telling the chip the kind of instruction and the second part giving the address from which a number is to be picked up. Notice something very important here. When we describe the task to be done as 'loading the accumulator', we use the same phrase in sections 2 and 3. In one we load a number directly and in the other we load a number from a memory location. Our brains are perfectly capable of understanding the distinction between the two, even though half the words are the same. The chip is not capable of making such fine distinctions, it needs to keep the two kinds of instructions entirely separate. To load a number from memory we need a separate instruction, 2001. To issue that instruction to the chip we send it: 2001, 1, which tells it to load the contents of memory location 1 into the accumulator.

These three imaginary simplified examples contain an important lesson. There is not necessarily any real link between the way an instruction is described in English and the way it is described in machine code. Many beginners become confused when a machine code book tells them that there is an instruction called 'load accumulator' and that it comes with a number of different 'addressing modes' or ways of specifying where the number to be loaded comes from. The fact is that all the instructions are different. If there are eight different ways of specifying the number to load into the accumulator then there will be eight different instructions. In this and other books you will find them described together under a table headed 'LDA' which is short for 'load accumulator' — this is only a convenience. To explain in detail each instruction and its addressing mode would require a book hundreds of pages long and costing more than you would probably want to pay. All machine code books work on the assumption that having explained to you all the different ways of specifying the location of a number, when they say that LDA comes with eight different addressing modes, you will understand that there are eight different instructions to load the accumulator.

The second lesson to be drawn from the examples is that there is no standard length to a machine code instruction. The number of memory locations which an instruction requires will depend on its addressing mode. The actual instruction itself will never, at least on the Lawland 10000 or the 6510, take more than a single memory location but there are also the byte(s) necessary to specify the number or address to be worked upon. In real machine code programming you will find that a whole instruction can have 1, 2, or 3 bytes. This is no problem for the chip itself, since it is designed to know exactly how many further numbers follow an instruction. The problem arises when the user sends the chip an instruction which requires two further numbers, but only one is sent. What will happen then is that the chip will insist that whatever comes next is part of the previous instruction, whether that was your intention or not.

The program and the program counter

So a machine code program is numbers stored in memory, and those numbers make up different instructions which can have different lengths. So far so good, but how are they a program? What is the difference between a random collection of numbers somewhere in the memory and a program which the chip will follow?

The answer is that you tell the chip where the program is, using the *program counter*, the register you see illustrated at the bottom right of the chip diagram. The program counter always contains an address, and that address is the position of the next instruction which the chip is going to execute. When for instance, you are in BASIC and you enter something like SYS 4567, the command to run a machine code program (don't try it unless you actually *have* a machine code program at 4567), all that happens is that the 6510 chip, which was previously executing the complex program known as the BASIC Interpreter, has the number 4567 loaded into its program counter and immediately sets about obeying whatever instructions it finds there. In terms of our chip model, it might work like this:

The instruction for 'transfer the contents of the X register to the accumulator' is 2500.

The instruction for 'store the contents of the accumulator in a specified address' is 3001, followed by another number specifying the address.

Set up the chip as follows:

123 in the X register
234 in the accumulator
2500 in memory location 1
3001 in memory location 2
4 in memory location 3
1 in the program counter.

Because the program counter points to memory location 1, the chip looks for an instruction there. It finds 2500, and so transfers the 123 in the X register to the accumulator. It now moves on to memory location 2. Here it finds 3001, which it knows is an instruction which requires one further number — it therefore picks up the 4 in location 3. With these two values having been picked up it knows that it has to store the contents of the accumulator in location 4, which finishes up containing 123.

If we had loaded 2 into the program counter to begin with, the transfer of the 123 to the accumulator would not have taken place, since the chip

would never have seen that instruction, and memory location 4 would have ended up as 234.

The program counter is therefore central to any machine code programming. Later in the book you will discover that the GOTO and GOSUB of BASIC are paralleled in machine code by instructions which place new values into the program counter, thus causing the chip to jump around within a program.

Machine code and assembly language

In everything that has gone before we have avoided one principal area of confusion that you may not even have noticed yet. We have talked at different times about the *names* given to individual machine code instructions and also about the *numbers* which are the instructions themselves as the chip understands them.

In the previous section we saw that a machine code program is nothing more than a series of numbers held in memory. Provided that the program counter points to the beginning of a series of valid machine code instructions, a program will be carried out.

In discussing the addressing modes we also said that newcomers to machine code are often confused by the fact that instructions are gathered into groups, such as the 'load the accumulator' instructions, even though every different method of loading the accumulator is an entirely different instruction.

None of this confusion need arise if we all programmed only in the form of machine code pure and simple — i.e. placing numbers into memory. The problem lies in the fact that while chips find it natural to communicate in the form of numbers, human beings on the whole do not.

For this reason, a separate language has been developed to help those who wish to program in machine code — assembly language.

Assembly language is sometimes called a form of shorthand for machine code but that can hardly be right since it takes far more space than the machine code itself. What assembly language *is*, is a way of writing and reading machine code in such a manner that it can be understood by anyone who wants to devote a little time and concentration to the task. It is a convenience, and nothing more. Chips do not understand assembly language, indeed thay are not expected to since whenever a program is written in assembly language it is first translated into machine code before the chip is asked to execute it.

We have already hinted at the form of some instruction in assembly language, but here is a little example to give you the flavour:

LDA#100 is an instruction to load the number 100 into the accumulator.

LDA 100 would load the contents of memory location 100 into the accumulator.

LDA 100,X would load into the accumulator the contents of the address arrived at by adding the contents of the X register to 100.

There is no need to puzzle too hard over these examples since such instructions will be dealt with in far greater detail later on. The important thing to note is that in assembly language the different instructions to load a number into the accumulator all have the same name LDA or LoaDAccumulator. Which of the different load accumulator instructions is meant will only be discovered by looking at the form of what follows the instruction. If you are going to spend any time working with machine code or assembly language then you will very quickly learn to recognise the forms in which the 11 different addressing modes are expressed.

Once you understand the difference between assembly language and machine code, you are faced with the choice of which you will work in. When planning a program it is almost universal to think and write in assembly language, but once the program is written on paper you have to decide how to enter it into your machine. One thing you can do is turn to a book like this one and use its tables to translate the assembly language instructions into machine code and place that into memory using something like the machine code loading program which you will find in Chapter 4 of this book. This is a long and laborious task and notoriously prone to errors which are almost impossible to detect — how do you set about debugging something which is simply a page of numbers?

Programming directly in machine code *can* be done, and programs presented in books and magazines will usually provide, alongside the assembly language version of a program, the numbers which make it up when translated into machine code. These can be entered directly and, provided that you check your entries carefully and the original worked, so should your copy.

Working directly in machine code is not, however, something that we would recommend to a beginner, for the simple reason that for most people it is not going to work. Errors are so frequent and so untraceable that most people who set out to program in machine code directly give up after a period and decide that it is all too difficult.

The proper course to take is to buy yourself a program known as an *Assembler/Disassembler.* The purpose of such a program is to allow you to enter a program in assembly language into your computer and only then to translate it into machine code — that is the *assembler* part of the package. The other side of the coin is the *disassembler* which will take a program in the memory of the computer and translate it into assembly language so that you read it more easily.

These are not the only advantages of an assembler. As you read on you will discover that there are many points at which programming in machine code directly will involve fussy calculations or counting, or where small changes in a program will involve great difficulty as the program is moved around slightly in the memory and all the important addresses to which the program points have to be changed. An assembler allows you to overcome such problems by, for instance giving names to memory locations and then using the name rather than the address in the program. Numerous other features, which may differ slightly from assembler to assembler, mean that the whole task of programming is immensely simplified.

You do not need an assembler package to use this book. It is written on the assumption that you do not have one, though instructions are also given for those who do. If you want to go on, however, and apply what you learn from this book in the real world of programming then an assembler will be an essential piece of equipment for you. At this point perhaps we should mention that our own assembler, the 'Mastercode Assembler', published by Sunshine in cassette form, was written largely with the inexperienced user in mind, and that the assembly language listings given in this book were all produced using it. At the same time, an assembler from any reputable software company should provide what you need.

CHAPTER 2
The Mysteries of Chip Arithmetic

In the last chapter we began to look in very crude terms at some of the capabilities of, and problems presented by, an imaginary chip called the Lawland 10000. In this chapter we shall stay with that technological wonder as we begin to look at the way in which a chip carries our arithmetic.

Perhaps it should be stressed at this point that there is no need to become bogged down in this chapter. Many people never get past the descriptions of the number system used by the chip, and thus never realise that for a large range of programming purposes, deep understanding is unnecessary. Even so, if you can begin to get a grasp of the way in which a chip looks at arithmetic, and why, you will probably save yourself some mistakes later on.

Because so many people seem to find the whole area confusing, we have decided in this book to adopt a new approach. You will already have gathered from the last chapter that our imaginary Lawland 10000 chip works with decimal numbers, the number system based on powers of ten which we use in everyday life. You probably also know that chips do not normally work in decimal, but in binary, the number system based on powers of two. Why that is, and what difficulties or opportunities it raises, we shall leave to a later chapter. In what follows here, we shall stick with decimal numbering in the hope that once you have grasped the principles in this familiar form you will have far less difficulty understanding what is going on in a real chip which uses binary numbers.

The range of the chip

In the last chapter we saw that the Lawland 10000 was capable of storing, in a single memory location, any number in the range 0–9999 *i.e.* a number which required no more than four digits. This is an important factor in assessing the capabilities of any chip since the limit imposed by the number of digits, whether 4, 8 or in the case of newer and more powerful chips 16 or 32 digits, is absolute. Everything that we want to do is going to have to be done within those digits and any handling of numbers with a larger number of digits can only be accomplished by using special techniques which divide those numbers up into sections. This

limitation will have important implications for the way the chip is designed and the way in which it is used, as we shall see when we come to put the chip to work at one of the tasks at which it is best, adding two numbers together.

Adding things up and the use of the carry flag

In looking at addition we shall not make any attempt to analyse what is going on 'mechanically' inside the chip, merely to note the way the addition proceeds, which is simply the process that we all learned at school, of adding two single digits and recording any carry.

Take the example of 4567+5678 — we can divide the process into four main stages:

(4)	(3)	(2)	(1)
4	5	6	7
+	+	+	+
5	6	7	8
+	+	+	
CARRY	CARRY	CARRY	

Step (1) involves the addition of 7 and 8 to produce 15, 5 of which is recorded and a 1 which is called a carry because it needs to be carried over into the next column to the left. Step (2) now comprises the addition of 6,7 and 1, to produce 14, of which four is recorded and one carried forward. Step (3), consists of the addition of 5,6 and 1 giving 2 to record and 1 to carry. The final step (4), involves adding 4,5 and 1. The result is 10 and, since there are no more columns to the left, the final result of the four steps is 10245

Two things are worth noting here. First, provided that we are only adding two numbers together the amount which has to be carried to the left can never be greater than 1 — 9+9 is the largest sum of two single digits, and that is 18. Carrying is therefore not a complex matter. The chip, which only ever adds two numbers at a time must simply be capable of recognising when a particular addition has resulted in a sum of more than 9 and then add 1 when adding together the next two figures to the left.

Secondly, it needs to be noted that something odd has happened at the left hand end of our number. We know what it is, since we took a brief

look at the problem in the last chapter. The number we have created is too large to be contained in 4 digits, and the extra digit at the left hand end of the number has been placed into the carry flag which exists just for that purpose. Later on in the chapter you will see that a wide range of arithmetic depends on the use of the carry flag to catch potentially lost digits.

BASIC program to demonstrate simple addition

```
10 PRINT"[CLEAR]"
20 INPUT "NUMBER~1~(0-9999):";N1
30 N1$ = RIGHT$("0000"+MID$(STR$(N1),2),4)
40 INPUT "[CD]NUMBER~2~(0-9999):";N2
50 N2$ = RIGHT$("0000"+MID$(STR$(N2),2),4)
60 BASE = 10
70 PRINT
80 FOR I = 1 TO 4
90 PRINT TAB(I*8-2) BASE "^" 4-I
100 PRINT "[CD][CD]" TAB(I*8+4) MID$(N1$,I,1)
110 PRINT TAB(I*8+4) MID$(N2$,I1)
120 PRINT TAB(I*8-2) "--------"
130 PRINT TAB(I*8-2) "[CD]--------"
140 PRINT "[CU][CU][CU][CU][CU][CU][CU][CU]
[CU]"
150 NEXT I
160 PRINT "[CD][CD][CD][CD][CD][CD]"
170 CARRY = 0
180 FOR I = 4 TO 1 STEP -1
190 T1 = ASC(MID$(N1$,I,1))-48
200 T2 = ASC(MID$(N2$,I,1))-48
210 PRINT "[CU][CU]"
220 SUM = T1+T2+CARRY
230 CARRY = 0
240 IF SUM>=BASE THEN SUM = SUM-BASE : CARRY
= 1
250 PRINT TAB(I*8+3) SUM
260 IF CARRY=1 THEN PRINT TAB(I*8-8) "[CD]CAR
RY[CU][CU]"
270 GET T$ : IF T$="" THEN 270
280 NEXT I
290 PRINT "[CD][CD][CD][CD]"
300 END
```

IMPORTANT NOTE on BASIC program formats:

The BASIC programs in this book have been processed to make them more readable. The major changes are:

a) Control characters, such as those for cursor movement, have been replaced by abbreviations in square brackets:

[CD] = CURSOR DOWN
[CU] = CURSOR UP
[CL] = CURSOR LEFT
[CR] = CURSOR RIGHT
[CLEAR] = CLEAR SCREEN
[RVS ON] = REVERSE ON

b) Graphics characters are replaced with codes as follows:

[^B] = Shifted B key
[C=B] = Commodore Key/B key simultaneously

c) Spaces inside quotes are replaced by the "~" symbol

The object of including BASIC programs such as this one in the book is not to engage in a great deal of commentary but simply to provide you with some tools to analyse procedures which may sometimes be a little difficult to visualise mentally. The current program accepts two unsigned numbers (i.e. they cannot be negative) and then, as you press the space bar, shows how the digits are added together, with any resulting carries. If you have any difficulty visualising how simple addition works, play with the program for a while until you have the rules clear in your mind.

Simple multiplication

We have noted that the Lawland 10000 is a very simple minded chip. Nevertheless, it is capable of a very crude form of multiplication or division — multiplication or division by 10. It does this by simply moving the digits of a number to the left or the right. If the start number were 1234, for instance, the carry flag and accumulator would look like this:

CF	ACCUMULATOR
0	1234

If the digits were shifted to the left, the result would be:

CF	ACCUMULATOR
1	2340

Division simply involves shifting the digits the other way:

CF	ACCUMULATOR
0	0123

What happens to the digit which is shifted off the end will depend on the kind of instruction being used, as you will see when you come to the Chapter 13.

Dealing with larger numbers

9999 is not, when all is said and done, a very large number, especially if it is to be the absolute limit of operations for a modern computer. Fortunately, though the Lawland 10000 is limited to four digit numbers, the programmer can easily make it handle larger values by splitting them up into sections. Using two sections, for instance, it would be possible to handle values of up to 99,999,999. The method for doing this, which will be discussed in greater detail for the real 6510, is to use two memory locations for each number. Each memory location, known as a byte, will hold one section of the number, one called the high byte and the other the low byte. The high byte is expressed in units which are one higher than the largest number that can be held by a single memory location. In the case of the Lawland 10000 the largest number which can be held in a single byte is 9,999, so the high byte will be in units of 10,000. As a practical example, if we wanted to store the number 12345678, we would place 5678 into the low byte, and the 1234 units of 10000 into the high byte. If we wanted to we could work with more than two bytes — three bytes would allow us to work with numbers up to 999,999,999,999.

Clearly, the techniques for working with such numbers are not going to be quite so simple as those for numbers which can be stored in a single byte. The problems, however, are not great, mainly revolving around ensuring that any carry from one byte is fed into the next byte up. In later chapters you will be shown in more detail how to work with two byte numbers on the 6510.

The problem of negative numbers

NOTE: You will gather from the length of this section that working with negative numbers is going to be difficult. If you find what follows hard,

perhaps we should put it in context by pointing out that many programmers seldom if ever work with negative numbers. The majority of programs can be made to work perfectly well without ever referring to negative numbers except for the calculation of addresses for what are known as *relative jumps*, a process which can be carried out according to simple rules and without any real understanding.

The reason that negative numbers cause us a problem is quite simply that to work with negative numbers involves subtraction and the Lawland 10000, in common with many chips including the 6510, is incapable of subtraction! Starting with the lowest digits of two numbers, it is programmed to be able to add them together, remembering any carry, then to move on to the next pair of digits, adding them, plus any carry from the previous pair and so on. The skills involved in subtraction, which seem so similar to our minds, are in fact quite different and quite beyond the chip.

To overcome this limitation we have to adopt a new system of numbering called 'ten's complement'. This reduces the range of numbers which the chip can store but it allows the chip to perform subtraction while all the time behaving as if only addition were going on.

Consider the following example:

Add together:

```
  0001
+9999
```

The answer:

```
 10000
```

But what would the answer be as far as the Lawland 10000 was concerned? It can only deal with four digits, so it places the fifth digit in the carry flag and records the main result as zero.
Now try this:

```
  0100
+9950
```

Real result 10050 — chip result 50 (discarding the fifth digit).
Compare those two sums with these:

```
  0001
 -0001

  0100
 -0050
```

Once again the results are zero and 50 respectively and the suspicion starts to creep in that something clever is going on. To find out what it might be, have another look at the second numbers in the first two sums, and then compare them with the second numbers in the second pair. If you think about it, it is obvious that 9999 and 9950 are simply 10000−1 and 10000−50 respectively — can such a simple dodge work and allow us to perform subtraction without anyone ever noticing.

The answer is yes. Simply taking a number away from 10000 and then adding it to a second number, produces a result as if the first number had been subtracted from the second. This raises the problem of when a number is negative. If you see the value 9950 being produced by a program, how do you interpret it. Is it plain 9950 or is it −50 in disguise? In fact, without a further dodge there is no way of telling.

What we must do is make a number announce as soon as we look at it whether it is positive or negative and the only way in which we can do this, given the fact that the Lawland 10000 can only store numbers, not minus signs or anything of the kind, is to do something to the number itself. In fact, what we shall do is to lay down a rule that any number with a 9 in the leftmost column is negative and any number with a zero is positive — no other digits will be allowed in the leftmost column. Remember that this does *not* mean that putting 9 in front of a number, say 123, turns it into its negative mirror image, ie −123. What it *does* mean is that when we see 9123 we shall regard it as 9123−10000, or −877.

Note that none of this really has anything to do with the chip itself. The beauty of our new numbering system is that the chip will behave perfectly normally and will produce answers just as if nothing had changed, it is simply that we shall regard the number as being positive or negative.

The disadvantages are:

1) that we have lost most of the potential range of our chip. We used to be able to store unsigned numbers from zero to 9999. With signed numbers we are limited to −1000 (10000–9000) to +999, only a fifth of the original range. In fact when we come to the real world of the 6510 chip we shall discover that there is no difference in range, it is simply that half the range has become negative.

2) more importantly, the programs that we write are going to have to be

that much more complex so as to be able to recognise and deal with the signs of numbers.

For these reasons it is most unlikely that we shall act as if numbers have positive or negative signs all the time. Most programming can get along quite well without negative numbers and so we shall be able to count up to 9999. On certain occasions, however, we shall accept the limitations and write our programs as if numbers were either positive or negative. Either way, the difference will be in our attitude to the answers and the way the programs are written — all the chip will ever do is add. Even if we build into the chip a special command to perform subtraction, all it will consist of is the translation of the number to be subtracted into ten's complement numbering before the two numbers are — you guessed it — added together.

This brief explanation is not meant to set you off on a major exploration of ten's complement arithmetic. If you were to start on such an exploration you would probably very quickly want to ask the question: 'How can we arrive at a negative number such as 9999, if the chip can't perform the sum 10000–1 to create it in the first place?' The answer to that is that you can't, not with a base of 10. When we move on to real chips, however, you will find that the parallel numbering system to what we have described above, called two's complement, *can* be created without any hint of subtraction.

Some peculiarities of negative numbers

So far, so good, but there is more to say if you are going to use negative numbers reliably. For the moment we shall simply raise some of the questions that arise, leaving it to later chapters to expand on the precise techniques which need to be employed to deal with them.

a) The format for negative numbers is always the same. Placing a 9 at the beginning of a number like 100 does not turn it into minus 100, but into minus 900 (1000–100). In other words the larger a negative number looks at first sight, the smaller it really is.

b) Different rules apply when it comes to any carry when adding together negative numbers. For instance, if we add 9001 and 9001 (minus 999 plus 999), the result is 18002, with the 1 going into the carry flag and the main result being 8002. According to our rules, this is not a negative number, so something has clearly gone wrong. The problem is that the main part of the number uses only the first three digits from the right and any carry into or out of the fourth digit can corrupt the format of the number so that we are no longer sure of the sign of the number. This is dealt with by the OF flag on your diagram, or *overflow flag*. The job of this flag is to record

whether any carry has been made into the fourth digit from the first three. In a later chapter you will see that the flag is used in conjunction with the carry flag to detect whether addition has resulted in the sign digit of a number being corrupted.

An additional facility is provided by the *sign flag* (SF on the diagram), which records at the end of any operation whether the result in the accumulator has a 9 in the fourth digit position. The flag works all the time, regardless of whether you are actually using negative numbers, and so is mostly irrelevant, but it can be very useful when adding and subtracting with signed numbers.

BASIC demonstration of addition employing signed numbers

```
10 PRINT"[CLEAR]"
20 INPUT "NUMBER~1~(-1000~TO~999):";N1
30 IF N1<0 THEN N1 = 10000+N1
40 N1$ = RIGHT$("0000"+MID$(STR$(N1),2),4)
50 INPUT "[CD]NUMBER~2~(-1000~TO~999):";N2
60 IF N2<0 THEN N2 = 10000+N2
70 N2$ = RIGHT$("0000"+MID$(STR$(N2),2),4)
80 BASE = 10
90 PRINT
100 FOR I = 1 TO 4
110 PRINT TAB(I*8-2)BASE "^" 4-I
120 PRINT "[CD][CD]" TAB(I*8+4) MID$(N1$,I,1)
130 PRINT TAB(I*8+4) MID$(N2$,I,1)
140 PRINT TAB(I*8-2) "--------"
150 PRINT TAB(I*8-2) "[CD]--------"
160 PRINT "[CU][CU][CU][CU][CU][CU][CU][CU]
[CU]"
170 NEXT I
180 PRINT "[CD][CD][CD][CD][CD][CD]"
190 CARRY = 0
200 FOR I = 4 TO 1 STEP -1
210 T1 = ASC(MID$(N1$,I,1))-48
220 T2 = ASC(MID$(N2$,I,1))-48
230 PRINT "[CU][CU]"
240 SUM = T1+2+CARRY
250 LASTCARRY = CARRY
260 CARRY = 0
270 IF SUM>=BASE THEN SUM = SUM-BASE : CARRY
= 1
280 OVERFLOW = LASTCARRY<>CARRY
290 PRINT TAB(I*8+3) SUM
```

```
300 IF CARRY=1 THEN PRINT TAB(I*8-8) "[CD]CAR
RY[CU][CU]"
310 GET T$ : IF T$="" THEN 310
320 NEXT I
330 PRINT "[CD][CD][CD][CD]"
340 IF OVERFLOW THEN PRINT "SIGN~DIGIT~IN~ERR
OR"
350 END
```

This second demonstration program brings the idea of 'offset' numbers to life by allowing you to specify negative numbers. The program automatically creates the offset number in the form in which it would be used on the Lawland 10000, and then demonstrates the process of addition as you press the space bar.

BASIC demonstration of subtraction employing signed numbers

```
10 PRINT"[CLEAR]"
20 INPUT "NUMBER~1~(-1000~TO~999):";N1
30 IF N1<0 THEN N1 = 10000+N1
40 N1$ = RIGHT$("0000"+MID$(STR$(N1),2),4)
50 INPUT "[CD]NUMBER~2~(-1000~TO~999):";N2
60 IF N2<0 THEN N2 = 10000+N2
70 N2$ = RIGHT$("0000"+MID$(STR$(N2),2),4)
80 BASE = 10
90 PRINT
100 FOR I = 1 TO 4
110 PRINT TAB(I*8-2) BSE "^" 4-I
120 PRINT "[CD][CD]" TAB(I*8+4) MID$(N1$,I,1)
130 PRINT TAB(I*8+4) MID$(N2$,I,1)
140 PRINT TAB(I*8-2) "--------"
150 PRINT TAB(I*8-2) "[CD]--------"
160 PRINT "[CU][CU][CU][CU][CU][CU][CU][CU]
[CU]"
170 NEXT I
180 PRINT "[CD][CD][CD][CD][CD][CD]"
190 CARRY = 1
200 FOR I = 4 TO 1 STEP -1
210 T1 = ASC(MID$(N1$,I,1))-48
220 T2 = ASC(MID$(N2$,I,1))-48
230 PRINT "[CU][CU]"
240 SUM = T1+(9-T2)+CARRY
250 LASTCARRY = CARRY
260 CARRY = 0
```

```
270 IF SUM>=BASE THEN SUM = SUM-BASE : CARRY
= 1
280 OVERFLOW = LASTCARRY<>CARRY
290 PRINT TAB(I*8+3) SUM
300 IF CARRY=1 THEN PRINT TAB(I*8-8) "[CD]CAR
RY[CU][CU]"
310 GET T$ : IF T$="" THEN 310
320 NEXT I
330 PRINT "[CD][CD][CD][CD]"
340 IF OVERFLOW THEN PRINT "SIGN~DIGIT~IN~ERR
OR"
350 END
```

The final program in this chapter allows you to input two signed numbers, either positive or negative, and then demonstrates the procedure for subtracting the second from the first as the space bar is pressed.

CHAPTER 3
Binary Numbers

Now is the time to move on to the real world. What has gone before should have helped to give you a picture of the workings of a chip and reminded you that those workings are by no means as complex as many people suppose. Now we hope to convince you that working with binary numbers is only a small hurdle to cross and, once crossed, you will be well on your way to being able to program the 6510 chip directly.

Just about everything that we have so far said about the Lawland 10000 is equally true of the 6510 except that the 6510 does not work with normal decimal numbers that we use in everyday life, numbers whose digits are based on powers of ten, but with binary numbers whose digits are based on powers of two.

Very little machine code programming actually needs you to be able to work with binary numbers, let alone be fluent with them. From time to time you will need to be able to translate a decimal number into binary so that you can see its shape, or translate a binary number into decimal but such translation doesn't have to be done instantly in your head. What matters more is that you understand what binary numbers are, since that understanding alone will unlock many of the secrets of the 6510 chip you want to program.

Numbers and bases

Before you can begin to use numbers a decision has to be taken about the 'base' of those numbers. Our normal decimal system works to a base of ten, that is to say that it has 10 units (0–9) and that once 9 has been passed, another digit has to be added. The value of any particular digit is determined by counting its distance from the right of the number, so that the number 1234 can also be described as:

(1*10↑3)+(2*10↑2)+(3*10↑1)+(4*10↑0)

It is also true, however, that *any* number (apart from 0 or 1) can be used as the base for a numbering system. The most commonly used are the binary (*base 2*), octal (*base 8*), duodecimal (*base 12*) and hexadecimal (*base 16*). Of these the binary system, with a base of 2 is universally used by computers because it fits exactly the way in which they work.

The computer and binary

Without getting technical, the chips that make up your computer are basically boxes crammed full of tiny on off switches. Different types of chip have slightly different ways of indicating whether a switch is on or off but it all boils down to the same thing. Anything that is done, and anything that is stored, has to be done by means of setting switches. The switches themselves are arranged into groups of eight, with each group being known as *byte*. These bytes correspond to what we have so far called memory locations, being a storage place in memory capable of holding a number, though the range of numbers which can be held is very different from that of our imaginary chip.

Here we have to go back to binary. Just as in decimal, the actual units that can be used range from 0 to 9 (1 less than 10), so in binary the units range from 0 to 1 (1 less than 2). Any number in binary has to be expressed in ones and zeros. And if you think about it, that is exactly how we have said the computer works — it consists of hundreds of thousands of tiny switches which can either be on (1) or off (0). Putting the computer and its bytes together with binary, it is not difficult to see that the computer works with numbers which run from zero to 8 binary digits, all set to 1, or the binary number 11111111, which is in fact 255 in decimal. There is, by the way, a special name given to the individual digits which make up a number. To save us having to say that 255 is made up of 8 binary digits, 'binary digit' is shortened to 'bit'. 255 is therefore the largest number which can be held in a byte made up of 8 bits.

The value of binary numbers

Just as earlier we described 1234 in terms of powers of ten, so we can describe a binary number in terms of powers of two. The eight binary digits, or bits, which a single byte of memory can hold represent, potentially:

(2↑7)+(2↑6)+(2↑5)+(2↑4)+(2↑3)+(2↑2)+(2↑1)+(2↑0)

or

128 + 64 + 32 + 16 + 8 + 4 + 2 + 1

Notice that this time we don't multiply each of the digits by anything. A binary digit can only ever be 1 or 0, so a digit either represents 1*2↑something or it represents zero.

Now all this sounds very forbidding when it is described in theory. In practice it very quickly becomes second nature to see that:

11 in binary represents 3 (2↑1 + 2↑0) in decimal
10000000 in binary represents 128 (2↑7) in decimal

To translate a number into binary manually, or by means of a computer program is a matter of dividing by successive powers of two. For example, to translate the number 173 into binary:

173/(2↑7)=1 remainder 45
45/(2↑6)=0 remainder 45
45/(2↑5)=1 remainder 13
13/(2↑4)=0 remainder 13
13/(2↑3)=1 remainder 5
5/(2↑2)=1 remainder 1
1/(2↑1)=0 remainder 1
1/(2↑0)=1 remainder 0

Thus 173 in binary is 10101101.

The same kind of calculation can be performed on any number and, if you do it regularly, you will probably find you are able to write down the binary representation of a number with only a little bit of mental calculation. Even so, in the early stages you might find the following table helpful, since it lays out the binary representation of all the decimal numbers between zero and 255, the range capable of being held in a single byte. All the binary numbers are given in eight digit form, the form in which they would be found when stored in a byte.

decimal	**binary**	**decimal**	**binary**
000	00000000	000	00000000
001	00000001	016	00010000
002	00000010	032	00100000
003	00000011	048	00110000
004	00000100	064	01000000
005	00000101	080	01010000
006	00000110	096	01100000
007	00000111	112	01110000
008	00001000	128	10000000
009	00001001	144	10010000
010	00001010	160	10100000
011	00001011	176	10110000
012	00001100	192	11000000
013	00001101	208	11010000
014	00001110	224	11100000
015	00001111	240	11110000

The way to use the table is as follows:

1) Take the number you want to translate into decimal and compare it with the column on the right. You are looking for the largest decimal number which is either less than or equal to your number.

2) Write down the first four digits of the binary number in the table which corresponds to the decimal number you have found.

3) Subtract the decimal number in the table from your number. The result will be a number under 16.

4) Go to the left hand column in the table and find your decimal remainder. The four digits on the right of the corresponding binary number are the remaining four digits you need.

Adding in binary

Some of the advantages of binary numbers for a simple minded beast like a computer can be seen very quickly when we come to perform some simple mathematics. Addition is a classic example, for in binary it is reduced to a set of straightforward rules. To see them in practice, try writing the following two binary numbers down:

1 1 0 1 0 1 1 0 (Decimal 214)
0 1 0 0 1 0 1 1 (Decimal 75)

If we want to add these two numbers together, the rules are:

Starting with the column on the right, start adding the corresponding columns in each number. Then

1) If there is no carry from the previous column, then
a) if there are two zeros then the equivalent column in the result is a zero.
b) if there is a one and a zero then the result is a one.
c) if there are two ones then the result is a zero, with a carry of one to the next column.

2) If there is a carry from the previous column, then
a) if there are two zeros then the equivalent column in the result is a one.
b) if there is a one and a zero then the result is zero with a carry to the next column of one.
c) if there are two ones then the result is one, with a carry of one to the next column.

Using these rules, the addition would go as follows:

```
  1 1 0 1 0 1 1 0
+
  0 1 0 0 1 0 1 1
  ---------------

                    1 (no carry)
                  0 (and carry)
                0 (and carry)
              0 (and carry)
            0 (and carry)
          1 (and carry)
        0 (and carry)
      0 (and carry)
    1 (no carry)
```

The result of the addition is 100100001, or 289 in decimal. Notice that our two eight digit numbers have produced a nine digit result. As with the imaginary chip, this extra digit, which cannot be held in a single byte, is recorded by the carry flag.

BASIC demonstration of unsigned addition in binary

```
10 PRINT"[CLEAR]"
20 INPUT "NUMBER^1^(0-1111):";N1
30 N1$ = RIGHT$("0000"+MID$(STR$(N1),2),4)
40 INPUT "[CD]NUMBER^2^(0-1111):";N2
50 N2$ = RIGHT$("0000"+MID$(STR$(N2),2),4)
60 BASE = 2
70 PRINT
80 FOR I = 1 TO 4
90 PRINT TAB(I*8-2) BASE "^" 4-I
100 PRINT "[CD][CD]" TAB(I*8+4) MID$(N1$,I,1)
110 PRINT TAB(I*8+4) MID$(N2$,I,)
120 PRINT TAB(I*8-2) "---------"
130 PRINT TAB(I*8-2) "[CD]---------"
140 PRINT "[CU][CU][CU][CU][CU][CU][CU][CU]
[CU]"
150 NEXT I
160 PRINT "[CD][CD][CD][CD][CD][CD]"
170 CARRY = 0
180 FOR I = 4 TO 1 STEP -1
190 T1 = ASC(MID$(N1$,I,1))-48
```

```
200 T2 = ASC(MID$(N2$,I,1))-48
210 PRINT "[CU][CU]"
220 SUM = T1+T2+CARRY
230 CARRY = 0
240 IF SUM>=BASE THEN SUM = SUM-BASE : CARRY
= 1
250 PRINT TAB(I*8+3) SUM
260 IF CARRY=1 THEN PRINT TAB(I*8-8) "[CD]CAR
RY[CU][CU]"
270 GET T$ : IF T$="" THEN 270
280 NEXT I
290 PRINT "[CD][CD][CD][CD]"
300 END
```

The program allows you to input two four digit binary numbers and to see the process of addition carried out as you press the space bar.

Subtraction in binary

Subtraction is equally simple, and we shall use the same two numbers to illustrate it — but it should be noted that what we are showing you a human being's way of subtracting. The chip will accomplish it in a different way as you will no doubt guess from the discussion of negative numbers in the last chapter.

To subtract, follow these rules:

Work from the rightmost column, and

1) If there is no borrow from the previous column, then
a) if subtracting zero from zero, or one from one, the result is zero.
b) if subtracting zero from one the result is one.
c) if subtracting one from zero the result if one with a borrow from the next column.

2) If there is a borrow from the previous column, then
a) if subtracting zero from zero the result is one and a borrow from the next column.
b) if subtracting zero form one the result is zero.
c) if subtracting one from zero the result is zero with a borrow from the next column.

Interestingly enough, if you subtract a larger number from a smaller using this method, you arrive at an offset number like those we examined in the

representation of negative numbers in the last chapter, and those you will find in the next section.

An example of subtraction using the two figures we previously added, would run as follows:

```
  1 1 0 1 0 1 1 0
-
  0 1 0 0 1 0 1 1
  ---------------
                    1  (and borrow)
                  1  (and borrow)
                0  (no borrow)
              1  (and borrow)
            0  (no borrow)
          0  (no borrow)
        0  (no borrow)
      1  (no borrow)
```

The result of the subtraction of the two numbers (214−75) is 10001011 (139 decimal).

BASIC demonstration of subtraction using unsigned binary numbers

```
10 PRINT"[CLEAR]"
20 INPUT "NUMBER^1^(0-1111):";N1
30 N1$ = RIGHT$("0000"+MID$(STR$(N1),2),4)
40 INPUT "[CD]NUMBER^2^(0-1111):";N2
50 N2$ = RIGHT$("0000"+MID$(STR$(N2),2),4)
60 BASE = 2
70 PRINT
80 FOR I = 1 TO 4
90 PRINT TAB(I*8-2) BASE "^" 4-I
100 PRINT "[CD][CD]" TAB(I*8+4) MID$(N1$,I,1)
110 PRINT TAB(I*8+4) MID$(N2$,I,)
120 PRINT TAB(I*8-2) "---------"
130 PRINT TAB(I*8-2) "[CD]---------"
140 PRINT "[CU][CU][CU][CU][CU][CU][CU][CU]
[CU]"
150 NEXT I
160 PRINT "[CD][CD][CD][CD][CD][CD]"
170 CARRY = 1
180 FOR I = 4 TO 1 STEP -1
190 T1 = ASC(MID$(N1$,I,1))-48
```

```
200 T2 = ASC(MID$(N2$,I,1))-48
210 PRINT "[CU][CU]"
220 SUM = T1+(1-T2)+CARRY
230 CARRY = 0
240 IF SUM>=BASE THEN SUM = SUM-BASE : CARRY
= 1
250 PRINT TAB(I*8+3) SUM
260 IF CARRY=1 THEN PRINT TAB(I*8-8) "[CD]CAR
RY[CU][CU]"
270 GET T$ : IF T$="" THEN 270
280 NEXT I
290 PRINT "[CD][CD][CD][CD]"
300 END
```

The program allows you to input two binary numbers and then illustrates the process of subtracting the second from the first as you press the space bar.

Negative numbers and binary

In the description of the Lawland 10000 we went through all kinds of evasions about the method by which the chip created numbers which were offset from 10000 while it wasn't able to subtract. When we come to binary numbering the whole question of these offset numbers becomes clearer.

As with decimal numbers, the way to represent a negative number is to place an indicator in the leftmost column of the number, with the rest of the number being an offset. Again as with decimal numbers, this reduces the maximum number which can be stored in a single byte, with the range becoming −128 to +127.

In the case of binary numbers, the value from which a number has to be subtracted in order to create a negative number, is 256, or one more than can be held in a single byte. If we wanted to create the number −1, then we subtract 1 from 256, leaving 255. The binary representation of 255 is 11111111, and if we were using unsigned numbers, 255 would be recognised as −1. However you do not have to go through all this mathematics in order to create a negative binary number, there is another simple method which is infallible. It is illustrated by the following example:

1) First take the equivalent positive number — so that if we want −97, we start with +97.

2) Translate the number into an eight digit binary number — 97 translates as 01100001.

3) Invert all the digits — 01100001 translates into 10011110.

4) Add 1 — in our case the result is 10011111.

5) The result is the negative representation of your number, or 159, which is 97 less than 256.

Once again it should be stressed that the chip does not recognise 10011111 as a negative number, it is simply that the kind of arithmetic operations which the chip can carry out can be made to produce results which would be sensible if the number were −97 rather than the 159 which the chip is actually working on. If you add 1 to 159 the result is 160 which makes sense. If we are regarding numbers as either positive or negative then that 160 will be looked upon as −96, which also makes sense since −97+1=−96. The chip will behave no differently just because you are treating numbers as having a plus or minus sign. The difference will be in the way you treat the results of the chip's calculations. This will be dealt with in more detail in Chapter 10.

BASIC demonstrations of addition and subtraction with signed binary numbers

```
10 PRINT"[CLEAR]"
20 INPUT "NUMBER~1~(-1000~TO~111):";N1
30 IF N1<0 THEN T = -N1 : GOSUB 360: N1 = T
40 N1$ = RIGHT$("0000"+MID$(STR$(N1),2),4)
50 INPUT "[CD]NUMBER~2~(-1000~TO~111):";N2
60 IF N2<0 THEN T = -N2 : GOSUB 360: N2 = T
70 N2$ = RIGHT$("0000"+MID$(STR$(N2),2),4)
80 BASE = 2
90 PRINT
100 FOR I =1 TO 4
110 PRINT TAB(I*8-2) BASE "^" 4-I
120 PRINT "[CD][CD]" TAB(I*8+4) MID$(N1$,I,1)
130 PRINT TAB(I*8+4) MID$(N2$,I,1)
140 PRINT TAB(I*8-2) "--------"
150 PRINT TAB(I*8-2) "[CD]--------"
160 PRINT "[CU][CU][CU][CU][CU][CU][CU][CU]
[CU]"
170 NEXT I
```

```
180 PRINT "[CD][CD][CD][CD][CD][CD]"
190 CARRY = 0
200 FOR I = 4 TO 1 STEP -1
210 T1 = ASC(MID$(N1$,I,1))-48
220 T2 = ASC(MID$(N2$,I,1))-48
230 PRINT "[CU][CU]"
240 SUM = T1+T2+CARRY
250 LASTCARRY = CARRY
260 CARRY = 0
270 IF SUM>=BASE THEN SUM = SUM-BASE : CARRY
= 1
280 OVERFLOW = LASTCARRY<>CARRY
290 PRINT TAB(I*8+3) SUM
300 IF CARRY=1 THEN PRINT TAB(I*8-8) "[CD]CAR
RY[CU][CU]"
310 GET T$ : IF T$="" THEN 310
320 NEXT I
330 PRINT "[CD][CD][CD][CD]"
340 IF OVEFLOW THEN PRINT "SIGN~DIGIT~IN~ERR
OR"
350 END
360 REM ****************************
370 REM 2'S COMP.
380 REM ****************************
390 T2 = 0
400 FOR I = 1 TO 4
410 T2 = T2/2+(1 AND T)*8
420 T = INT(T/10)
430 NEXT I
440 T2 = 16-T2
450 T = 0
460 FOR I = 1 TO 4
470 T = T/10+(T2 AND 1)*1000
480 T2 = INT(T2/2)
490 NEXT I
500 RETURN


10 PRINT"[CLEAR]"
20 INPUT "NUMBER~1~(-1000~TO~111):";N1
30 IF N1<0 THEN T = -N1 : GOSUB 360: N1 = T
40 N1$ = RIGHT$("0000"+MID$(STR$(N1),2),4)
50 INPUT "[CD]NUMBER~2~(-1000~TO~111):";N2
60 IF N2<0 THEN T = -N2 : GOSUB 360: N2 = T
```

```
70 N2$ = RIGHT$("0000"+MID$(STR$(N2),2),4)
80 BASE = 2
90 PRINT
100 FOR I =1 TO 4
110 PRINT TAB(I*8-2) BASE "^" 4-I
120 PRINT "[CD][CD]" TAB(I*8+4) MID$(N1$,I,1)
130 PRINT TAB(I*8+4) MID$(N2$,I,1)
140 PRINT TAB(I*8-2) "--------"
150 PRINT TAB(I*8-2) "[CD]--------"
160 PRINT "[CU][CU][CU][CU][CU][CU][CU][CU]
[CU]"
170 NEXT I
180 PRINT "[CD][CD][CD][CD][CD][CD]"
190 CARRY = 1
200 FOR I = 4 TO 1 STEP -1
210 T1 = ASC(MID$(N1$,I,1))-48
220 T2 = ASC(MID$(N2$,I,1))-48
230 "[CU][CU]"
240 SUM = T1+(1-T2)+CARRY
250 LASTCARRY = CARRY
260 CARRY = 0
270 IF SUM>=BASE THEN SUM = SUM-BASE : CARRY
= 1
280 OVERFLOW = LASTCARRY<>CARRY
290 PRINT TAB(I*8+3) SUM
300 IF CARRY=1 THEN PRINT TAB(I*8-8) "[CD]CAR
RY[CU][CU]"
310 GET T$ : IF T$="" THEN 310
320 NEXT I
330 PRINT "[CD][CD][CD][CD]"
340 IF OVERFLOW THEN PRINT "SIGN~DIGIT~IN~ERR
OR"
350 END
360 REM ****************************
370 REM 2'S COMP.
380 REM ****************************
390 T2 = 0
400 FOR I = 1 TO 4
410 T2 = T2/2+(1 AND T)*8
420 T = INT(T/10)
430 NEXT I
440 T2 = 16-T2
450 T = 0
460 FOR I = 1 TO 4
```

```
470 T = T/10+(T2 AND 1)*1000
480 T2 = INT(T2/2)
490 NEXT I
500 RETURN
```

The two programs allow signed binary numbers (simply use a minus sign to indicate a negative number) to be either added or subtracted. In the case of subtraction, the program automatically creates the correct offset form of the number.

Flags in binary

Once we have taken the plunge into binary, a great many things become clearer about the working of the chip. We shall not go into the details of all the flags at this point but it is worth noting that the flags are far simpler to visualise in binary than they were in decimal. The diagram of the chip shows that there are eight flags, even though one is not used. Eight flags, each of them capable of recording a single yes/no decision, might ring a bell with you. In fact the eight flags are simply the eight bits of a one byte register in the chip known as the *status register*. In the case of the flags, if a bit is *set* (switched to 1), then the flag is indicating a 'yes' answer to some question or other, while if it is *reset* (switched to 0) then the answer indicated is 'no'. It is wise to fix in your mind this slightly odd use of the words 'set' and 'reset' since you will find them widely used in books on machine code to refer to the state of individual bits, even though 'on' and 'off' might be more straightforward.

Multiplying and dividing in binary

Just as a decimal number can be multiplied or divided by its base of ten simply by shifting the digits to the left or right, so can a binary number. If you took the trouble to follow the discussion of multiplication and division in the last chapter there is nothing to add apart from two examples of the processes in action. To simplify, we shall use a binary number with only one digit set to 1. Any number would do so we shall choose 16 in decimal, which as an eight digit binary number is expressed as 00010000.

To multiply by two, we shift to the left, giving a result of 00100000, or 32 in decimal. If we were to perform this shift to the left on a number in the accumulator to the extent that the single 1 fell off the left hand end of the byte, the result would be zero, with the 1 stored in the carry flag.

Division is accomplished by shifting the digits in the opposite direction, to the right, and the result of one shift would be 00001000, or 8 in decimal. If the process were performed repeatedly on a number in the accumulator then depending on the command used (see Chapter 12) the 1 would either be placed into the carry flag or lost entirely.

CHAPTER 4
Using this Book

For a moment, we shall take a rest from the theory of a chip and, before moving on to the details of machine code, take a look at the methods which will be used throughout the rest of this book. The two main topics we would like you to be clear about are the hexadecimal numbering system and the Code Runner program which we recommend that you use for experimenting with the small machine code routine contained in the chapters which follow.

The BASIC Code Runner program

This book is liberally sprinkled with short machine code routines to illustrate the manner in which the various machine code instructions operate and work together. They are all designed to be run using a special BASIC program contained within this chapter. Why?

The answer lies in the problem of demonstrating *anything* in machine code. Most machine code programs, and certainly most machine code instructions, do nothing visible. Not only that, what they do they do very quickly — by the time you have blinked, the effects of the instructions have come and gone and your 64 has moved on to other tasks. Something has to be done to slow the whole thing down.

A second problem is that programs to demonstrate the different abilities of the chip are not always safe programs to run. Not that they can damage the 64, of course, but they could easily crash the system. Machine code programs have to be carefully constructed so that when they finish and control returns to BASIC, the system is in a sensible state. We are therefore faced with the ludicrous situation that to demonstrate a single instruction we might have to write half a page of machine code program to prevent the machine crashing once the demonstration has been made.

Thirdly, there is the question of experimenting. As far as possible, we wanted to give you machine code routines that you could play with. We wanted you to be able to alter the values that instructions were given to work on, without having to constantly rewrite programs or poke values into memory.

For all these reasons we decided to turn the job of running the machine code in this book to a program which is a mixture of BASIC and machine

code, and which we have called *Code Runner*. The job of Code Runner is to allow you to input a machine code program easily, to manipulate the values of a small number of memory locations on which the programs will operate and also to ensure that it is very difficult for you to crash the system.

Inputting a machine code program

The Code Runner program is designed to allow you to enter machine code programs in the form of lines of DATA in BASIC. This is a common method and, though we would not necessarily recommend it for long programs, it is by far the simplest way of entering pieces of code which are in many cases only four or five bytes long.

Whenever an example is given in the chapters that follow, it is given in two versions, first a listing in assembly language, on which the commentary is based, and then one or more lines of DATA to be written into the Code Runner program. The DATA is placed in a module starting at line 6030 and each time you enter some code you should start with a fresh version of Code Runner which has nothing between lines 6030 and 6999. The easiest method, of course, will be to store an 'empty' copy of Code Runner on tape or disc and load it afresh for each demonstration. If you are prepared to devote sufficient disk or tape space to it, each version of Code Runner, with its different DATA lines, can be saved under a different name so that in future, for any particular demonstration, all you need to do is load the version of code runner containing the right machine code and then run it.

Slowing down the execution of the code

The second advantage of the Code Runner program is that it allows you to see exactly what the machine code does. The way in which this is achieved is to use a piece of built in machine code to sample the contents of the chip registers and of certain memory locations immediately after the machine code has been executed and to display these on the screen. In other words, though the chip has long ago forgotten the piece of machine code, a snapshot of the effect of running it is left on the screen so that you can see exactly what happened.

Playing with values

In order to give you the opportunity to see the effect on machine code instructions of working on different values, most of the routines in the book work on a very limited number of locations in memory. In turn, the Code Runner program allows you to change the values contained in a

limited number of memory locations. It will probably not surprise you to learn that the limited number of locations are the same in each case. In most cases, when values are to be loaded into registers, or added together or whatever, you will be able to decide first of all what those values will be. The way in which this will be accomplished is that when you have added the machine code to Code Runner and RUN the program, it will first of all ask you whether you wish to change the value of any locations in memory. If you answer 'Y', you will be given the option to change any or all of either four locations in low, or zero-page memory, or in high memory — the difference between these two locations will be explained in subsequent chapters.

Once you have made your choice and changed any memory locations you desire, the machine code routine will be executed, using the values you have input.

Stopping system crashes

The final function of Code Runner is to allow you and us to get away with short machine code routines which would normally crash the system.

When the program is RUN and before it executes your machine code, it first takes a copy of all the important registers in the chip. Since the system is actually running, these registers must be at sensible values. As soon as your machine code has finished and the snapshot of its effect on the registers has been taken, the registers are restored to their original condition. This does not make it impossible for you to crash the system, but it does make it relatively difficult.

Example machine code routine format

```
 (A)      (B)          (C)
ADD.     DATA      SOURCE CODE
00                 10 PRT                  }
00                 20 ORG $C030            }  ——ASSEMBLER DIRECTIVES
C030               30 SYM                  }
C030               40 FINISH = $C013       ——ADDRESS
C030     A5FB      50 LDA $FB              }
C032     A2FC      60 LDX #$FC             }  ——MACHINE CODE ROUTINE
C034     9500      70 STA 0.X              }
C036     4C13C0    80 JMP FINISH           ——JUMP TO FINISH ROUTINE
C039               90 END

                   (D)
6000 REM#*****************************
6010 REM DATA FOR MACHINE CODE
6020 REM*****************************
6030 DATA A5,FB,A2,FC,95,00
```

(A) The column showing the address in memory at which the machine code instruction on that line will start.
(B) The hexadecimal values representing the machine code instructions.
(C) The assembly language form of the instructions. The first section of these consists of instructions to the Assembler program, telling it where to place the code in memory (ORG) and to print a list of variables at the end (SYM) though we have deleted the list. The machine code routine itself follows, and finally, in most cases, the FINISH routine is called to tidy up the memory.
(D) The DATA to place the machine code into the Code Runner program.

To sum up, the procedure for using Code Runner is as follows:

1) Call up a 'clean' version of the program with no DATA statements from 6000–6999.

2) Enter the machine code to be executed in the form of the DATA statements.

3) RUN the program and use the program options to set any memory locations to their desired values.

4) Code Runner will now execute your machine code and return the system to BASIC.

5) Examine the screen display, which should reveal the effects of running your machine code on the system.

The use of hexadecimal numbers

Before you go on to look at the Code Runner program itself, an explanation of why it is that all the DATA for the machine code routines and all the addresses in memory to which the machine code routines apply are all specified in the form of numbers known as *hexadecimal* — that is to say numbers which are expressed to the base 16.

Hexadecimal numbers, unlike decimal, have a range of 16 possible digits, 0–9, followed by A–F to account for the remaining 6. The lists of hexadecimal and decimal numbers match up as follows

D	H	D	H
0	0	32	20
1	1	48	30

2	2	64	40
3	3	240	F0
4	4	255	FF
5	5	256	100
6	6	512	200
7	7	768	300
8	8	1024	400
9	9	2048	800
10	A	4095	FFF
11	B	4096	1000
12	C	8192	2000
13	D	16384	4000
14	E	32768	8000
15	F	61440	F000
16	10	65535	FFFF

To translate from decimal to hexadecimal, all you have to do is to divide by powers of 16, and keep on doing it until what is left is less than sixteen. Take as an example the number 3781

1) We start by dividing it by 16↑2 (16↑3 is 4096 and therefore too big) with the result 14, remainder 197.

2) The first digit of the hexadecimal number is therefore E.

3) Dividing 197 by 16↑1 (or 16), we get 12 with a remainder of 5.

4) The second digit of the hexadecimal number is C

5) The final digit is divided by 16↑0 (or 1) and obviously the result is 5.

6) The final digit of the hexadecimal number is 5.

7) Putting the digits together, we find that the hexadecimal representation of the decimal number 3781 is EC5.

Why go to the trouble? The answer is that the hexadecimal system is a good compromise between what makes sense on a computer and what makes sense to a human being. At the level of a single byte, the whole of the range 0–255 can be expressed in two hexadecimal digits 00–FF. Not only that, each digit will apply to four bits within the byte — the first character will apply to the upper four bits and the second character to the lower four bits. Once you have begun to work with hexadecimal numbers for a while and become familiar with what the 16 possible units mean in

binary, you will very quickly start to see the shape of a binary number when you look at a hexadecimal one.

At a higher level, that of the whole machine, the computer works in units of 256, as we have already seen. Decimal numbering reveals very little about the significance of what may be important numbers from the point of view of the computer. The decimal number 32768 looks unexceptional — in hexadecimal it is expressed as 8000 and clearly announces that it is something significant. Most important borders between different activities within a computer lie at addresses which would mean very little in decimal but are clearly significant when expressed in hexadecimal.

It is for these reasons that machine code programmers almost universally work in hexadecimal and why we would recommend that you do, too. Of course, to begin with it will feel awkward and you may find yourself constantly referring to tables or to a calculator which is capable of translating for you, but you may be surprised at how quickly you adapt and you will never regret the effort.

The Code Runner program

Given below is a listing for the Code Runner Program on which the demonstrations in this book are based. As mentioned above, the program is a combination of BASIC and machine code. The commentary on the program will consist of a brief outline of the work of the BASIC lines since these are mostly self-explanatory. The commentary on the machine code is even more terse since to comment in detail would involve the discussion of techniques which are well beyond the scope of this book. Suffice it to say that the routines do the job. A full assembly language listing is given of the routines so that when your expertise has grown you can come back and analyse them for yourself.

The Code Runner program listing

```
1000 MEM = 0
1010 INPUT "[CLEAR][CD]DO~YOU~WISH~TO~USE~HIG
H~MEMORY~(Y/N)?~N[CL][CL][CL]";T$
1020 IF T$="Y" THEN MEM = 49664
1030 SIZE = 0
1040 INPUT "[CD]DO~YOU~WISH~TO~ALTER~MEMORY~?
~N[CL][CL][CL]";T$
1050 IF T$="Y" THEN GOSUB 4000
1060 PRINT "[CLEAR][CD][CD][CD][CD]~~~~~~~LOA
DING~MACHINE~CODE"
```

```
1070 IF A THEN 1220
1080 A = -1
1090 RESTORE
1100 FOR J = 49152 TO 49198
1110 READ H$
1120 GOSUB 2000
1130 POKE J,H
1140 NEXT J
1150 AD = 49200
1160 READ H$
1170 IF H$="END" THEN 1220
1180 GOSUB 2000
1190 POKE AD,H
1200 AD = AD+1
1210 GOTO 1160
1220 SYS 49152
1230 PRINT "[CLEAR][CD]~~~~~~~~REGISTERS"
1240 BASE = 16
1250 H = PEEK(49408) : GOSUB 3000
1260 PRINT "[CD][RVS ON]ACCUMULATOR~~~~~~~~~~
[RVS OFF]~" H$
1270 H = PEEK(49409) : GOSUB 3000
1280 PRINT "[CD][RVS ON]X~REGISTER~~~~~~~~~~~
[RVS OFF]~" H$
1290 H = PEEK(49410) : GOSUB 3000
1300 PRINT "[CD][RVS ON]Y~REGISTER~~~~~~~~~~~
[RVS OFF]~" H$
1310 H = PEEK(49412) : GOSUB 3000
1320 PRINT "[CD][RVS ON]STACK~POINTER~~~~~~~~
[RVS OFF]~" H$
1330 BASE = 2
1340 H = PEEK(49411) : GOSUB 3000
1350 PRINT SPC(21) "[CD]SO~BDIZC"
1360 PRINT "[RVS ON]STATUS~REGISTER~~~~~~
[RVS OFF]~" H$
1370 BASE = 16
1380 FOR I = 251+MEM TO 254+MEM
1390 SIZE = 4
1400 H = I : GOSUB 3000
1410 PRINT "[CD][RVS ON]MEMORY~LOCATION~" H$
;
1420 SIZE = 0
1430 H = PEEK(I) : GOSUB 3000
1440 PRINT "[RVS OFF]~" H$
```

```
1450 NEXT I
1460 END
2000 REM**********************************
2010 REM CONVERT HEX TO DEC
2020 REM**********************************
2030 H = 0
2040 FOR I = 1 TO LEN(H$)
2050 T = ASC(MID$(H$,I,1))-48
2060 IF T<10 THEN 2090
2070 T = T-7
2080 IF T<10 OR T>15 THEN PRINT "ERROR~IN~HEX
~BYTE" : END
2090 H = H*16+T
2100 NEXT I
2110 RETURN
3000 REM**********************************
3010 REM CONVERT DEC TO ANY BASE
3020 REM**********************************
3030 H$ = ""
3040 T = H-INT(H/BASE)*BASE
3050 IF T>9 THEN T = T+7
3060 H$ = CHR$(T+48)+H$
3070 H = INT(H/BASE)
3080 IF H>0 THEN 3040
3090 T = 2
3100 IF SIZE<>0 THEN T=SIZE
3110 IF BASE=2 THEN T = 8
3120 H$ = RIGHT$("00000000"+H$,T)
3130 RETURN
4000 REM**********************************
4010 REM ALTER MEMORY LOCATIONS
4020 REM**********************************
4030 BASE = 16
4040 FOR J = 251+MEM TO 254+MEM
4050 SIZE = 4
4060 H = J
4070 GOSUB 3000
4080 PRINT "[CD][CD][CD]LOCATION~" H$ "~NEW~V
ALUE~?~";
4090 SIZE = 0
4100 H = PEEK(J)
4110 GOSUB 3000
4120 PRINT H$ "[CL][CL][CL][CL]" ;
4130 INPUT H$
```

```
4140 GOSUB 2000
4150 POKE J,H
4160 NEXT J
4170 RETURN
5000 REM#################################
5010 REM DATA FOR REGISTER DUMP
5020 REM#################################
5030 DATA BA,8E,05,C1,CA,CA,CA,CA,CA,CA
5040 DATA 9A,08,68,8D,06,C1,4C,30,C0,8D
5050 DATA 00,C1,8E,01,C1,8C,02,C1,08,68
5060 DATA 8D,03,C1,BA,8E,04,C1,AE,05,C1
5070 DATA 9A,AD,06,C1,48,28,60
6000 REM#################################
6010 REM DATA FOR MACHINE CODE
6020 REM#################################
7000 REM#################################
7010 REM DATA TO RETURN TO HEX LOADER
7020 REM#################################
7030 DATA 4C,13,C0
7040 DATA END
```

Commentary
Lines 1000–1050: These lines allow the user to specify whether any of the locations on which the demonstration routines operate are to be altered. Some routines operate on four locations in high memory, beginning at 49403, but most work with four locations in the low memory at 251 onwards.

Lines 1026–1210: These lines perform the dual function of reading in the machine code for the built in machine code and then the code for whatever machine code routine is to be demonstrated. The values for the machine code are READ from the DATA statements and poked into memory to form a machine code program

The demonstration machine code must be entered in the hexadecimal number format and the lines which contain it must end with a DATA item reading 'END'.

Lines 1070 and 1080 ensure that if you start the program with GOTO 1000, the program does not bother to read and store the machine code it simply executes what has already been placed into the memory. Entering RUN, or changing the DATA, will clear the variable A and ensure that the machine code is reloaded into the memory.

Lines 1230–1450: These lines read the contents of the chip registers from locations in memory where the built in machine code routines have stored them, and then display them on the screen for your examination. The values will be presented in the hexadecimal format.

Lines 2000–2110: This subroutine takes a number in hexadecimal format and translates it into decimal so that it can be placed into the memory by the main part of the program, using POKE.

Lines 3000–3130: All the displays of the program are in hexadecimal, and the job of translating decimal numbers picked up from the memory into hexadecimal is performed by this module.

Lines 5000–5070: The DATA which will create the permanent machine code routines built into the program.

Lines 6000–6020: The markers for the space provided for the demonstration routines or your own routines.

Lines 7000–7040: These lines tag some extra machine code onto the end of whatever you enter from 6000 onwards. The effect is to ensure that if possible you return safely to BASIC without crashing the system.

Assembly language listings of built in machine code routines from Code Runner program

```
ADD.   DATA      SOURCE CODE
00                 10 PRT
00                 20 ORG $C000
C000               30 SYM
C000   BA          40 TSX
C001   8E05C1      50 STX TEMP1
C004   CA          60 DEX
C005   CA          70 DEX
C006   CA          80 DEX
C007   CA          90 DEX
C008   CA         100 DEX
C009   CA         110 DEX
C00A   9A         120 TXS
C00B   08         130 PHP
C00C   68         140 PLA
C00D   8D06C1     150 STA TEMP2
C010   4C30C0     160 JMP START
```

```
C013                170 ;
C013                180 FINISH
C013   8D00C1       190 STA ASTORE
C016   8E01C1       200 STX XSTORE
C019   8C02C1       210 STY YSTORE
C01C   08           220 PHP
C01D   68           230 PLA
C01E   8D03C1       240 STA PSTORE
C021   BA           250 TSX
C022   8E04C1       260 STX SSTORE
C025   AE05C1       270 LDX TEMP1
C028   9A           280 TXS
C029   AD06C1       290 LDA TEMP2
C02C   48           300 PHA
C02D   28           310 PLP
C02E   60           320 RTS
C02F                330 ;
C02F                340 ORG $C030
C030                350 START
C030   4C13C0       360 JMP FINISH
C033                370 ;
C033                380 ORG $C100
C100   00           390 ASTORE BYT 0
C101   00           400 XSTORE BYT 0
C102   00           410 YSTORE BYT 0
C103   00           420 PSTORE BYT 0
C104   00           430 SSTORE BYT 0
C105   00           440 TEMP1  BYT 0
C106   00           450 TEMP2  BYT 0
C107                460 END

TOTAL ERRORS IN FILE --- 0

FINISH              C013
START               C030
ASTORE              C100
XSTORE              C101
YSTORE              C102
PSTORE              C103
SSTORE              C104
TEMP1               C105
TEMP2               C106
TOTAL NUMBER OF SYMBOLS --- 9
```

Commentary
Lines 40–160: These lines store the present contents of the status register and the stack pointer, the two registers which are needed to return to BASIC safely. In case the stack contains values which system may be needed later, some dummy values are placed onto the stack so that any erroneous use of the top few bytes will not corrupt anything important.

Lines 180–320: The routine which reads the current state of all the registers and places their values into bytes of memory which the BASIC program can PEEK later when it creates the screen display of the state of the chip.

Lines 380–450: These lines are not really machine code but directions to the assembler to set aside a number of bytes of memory as storage places for the register values which the previous section will be reading.

Demonstration routines with an assembler
If you possess an assembler package then you have to decide whether you wish to work only with the Code Runner program, for the sake of simplicity, or whether you wish to try to perform the demonstrations using your assembler.

In order to be of any use in this respect, your assembler package must be able to:

1) Input an assembly language program and assemble it to memory.

2) Step through a machine code program using a *trace* function.

3) Set the value of individual memory locations by means of a machine code monitor facility.

Provided that both of these functions work well, it may be possible to follow the execution of a program through, instruction by instruction, seeing its effects on registers and so forth. Even so, some of the routines may cause difficulties since few assembler trace facilities are capable of handling every circumstance.

In general, though use of an assembler is strongly recommended for ordinary programming purposes, we would recommend, because of the specialised nature of the demonstrations, that for the sake of simplicity you stick to the use of the Code Runner program.

CHAPTER 5
An Introduction to Addressing

For the rest of the book we leave aside theory and turn to the real job of programming on the 6510 chip which runs the Commodore 64. But having said that, our first task is to go back and look again at a concept which was introduced in the first chapter — addressing, or the variety of methods by which the chip can be told to find a number on which it is going to perform an action.

Leave aside for the moment the question of the task to be performed, let us assume that we are going to do something to a value which is contained within memory. The byte which contains that value will have a place in memory somewhere between byte zero and byte 65535 (for an eight bit chip like the 6510), and that place is known as its *address*. So far so good. The point at which the confusion starts to arise is, 'How shall we describe that address to the chip?'

One straight answer would be to say that since the address is a number, the machine code instruction should simply contain the number. This is perfectly reasonable and, indeed, it is one of the ways in which an address can be described to the chip. But it is not the only way, by any means.

If it is left entirely up to the program to describe an address, we find that we are very limited in what we can achieve. Suppose, for instance, that we know the number we want to work on — do we really have first to place it into memory and then tell the chip where to find it? Wouldn't it make a great deal more sense to have a way of saying to the chip — forget about addresses, take this number from the program and work on it. Clearly, that would save some time and effort, so we'll add a new addressing mode, to allow us to write numbers we want to work with directly into our program.

Other questions quickly arise. Supposing that we want to use a table of values for some reason. Such tables, known as 'arrays' are used all the time in BASIC and it would be very difficult to imagine programming without their help. When an array is accessed in BASIC with a statement like:

```
X=ARRAY(12)
```

we rely on the BASIC Interpreter (the machine code program which runs

the BASIC language) to find the table of values called ARRAY and, having found the start of the table, to look up the 13th value (numbered from zero, remember). If there were no arrays, we would have to have a whole series of variables called, for instance, A1, A2, A3. . . . Programming in this way would become extremely cumbersome.

Clearly the usefulness of tables of values is not limited to the BASIC language — they represent a programming technique which every computer language needs. In machine code we cannot simply declare an array and then expect to be able to call it up by name, but what we can do is to set aside an area of memory and store a sequence of numbers in it. Then, ideally, we want to be able to point to the beginning of the table, and tell the chip 'I want the 10th value from the table that begins here'. This is clearly going to be faster and simpler than having to calculate the address of each position in the table, so we shall add a third kind of addressing.

Is that it now? Well no, not yet. So far we can specify a value in the program or we can specify an address in which a value is to be found. But what if we want to write a short program which will be executed several times, or a loop within a program, specifying a different address to find a value each time? Clearly we can't change the program every time — we need something along the lines of a BASIC variable, whose value changes as the program is run.

We have already seen that in machine code the equivalent to storing a value in a BASIC variable is to store it in a memory location. The upshot of all this is that if we want the same part of the program to refer to different addresses in memory at different times, we cannot specify the addresses ourselves, we must ask the program to look at a variable (*i.e.* a location in memory) and get an address from there. Another part of the program will have the task of changing the address from time to time — otherwise the whole exercise would be rather pointless.

Finally, when it comes to the use of instructions known as *jumps*, which are equivalent to BASIC's GOTO and GOSUB, yet another form of addressing might be useful. Because machine code programs can be placed almost anywhere in the memory of the computer, difficulties can be created by instructions which tell the chip to jump to the machine code instruction at such and such an address. If the program is moved to another part of memory, or if parts of the program are moved to make room for more instructions, these jump instructions can become meaningless. One way round this problem is to use another addressing mode known as *relative addressing* where, instead of specifying the exact address of the part of the program which is to be jumped to, the program is told something like 'jump to the instruction whose address is 100 bytes before the address of this instruction'. Now if the program is moved, provided that the two instructions are still the same distance apart, the

address will still make perfect sense.

So now we have five different ways of defining an address. It should be stressed that these are only vague and generalised descriptions. In real life, these five ways of describing an address are broken down into even more tightly defined methods and, before we go on to look at machine code instructions themselves, we shall look at the addressing modes in detail. Don't, however, read on until you can see *why* it is that the five different ways of describing an address are needed and how they operate.

Zero-page and whole memory addresses

So far, we have assumed that commands can be used to specify an address anywhere in the range 0–65535, even though the form in which the address is specified may differ. The reason for the particular limits on the range are to do with the fact that most addresses will be specified as a two-byte number of the kind we described at the end of Chapter 3.

There is, however, a second class of addressing modes which, while far more limited in their scope, are used as often as possible because of their speed. Speed is, of course, one of the constant goals of the machine code programmer and any calculation which the chip has to perform slows the program down. When an address, however it is specified, requires two bytes, then the chip is having to perform some internal calculations to deal with the two byte number involved. Some addresses, however, cut out this stage of the process because they only require one byte to specify them. Since one byte can hold any value in the range zero to 255, it is obvious that the only addresses which can be specified in a single byte, without the chip having to do extra calculations to arrive at the eventual position (as in relative addressing), are addresses between zero and 255.

These are known as *zero-page* addresses because it is conventional to regard the memory as being a collection of 256 byte blocks of memory called pages. To take account of the potential for speed which one-byte addresses provide, a whole new class of addressing modes is provided which allow the chip to dispense completely with the need to consider two bytes for an address. Unsurprisingly, these addressing modes are called zero-page modes. They are used extensively by experienced programmers, and the operating system of your Commodore 64 relies very heavily on them. If you look at a copy of the *Commodore 64 Programmers' Reference Guide* you will find that the first 256 bytes of memory are crammed with locations used by the 64 to store information it needs for its own purposes. This information could be stored anywhere in the memory — it is placed in the zero-page in order that the 64's internal activities should use as little time as possible.

The 6510 is provided with a whole range of instructions which require this zero-page form of addressing and they are known as zero-page

instructions. Sometimes they perform the same task as another command which can be directed towards any address in the memory, regardless of whether it is in the first 256 bytes — it is simply that the zero page form is quicker.

Note as always that where an instruction exists in two forms, what is meant is that despite any similarities between the work of the two instructions, they are entirely different when it comes to the command which must be given to the chip itself. You cannot stick a one-byte address on the end of a whole memory instruction and expect the chip to recognise that only one byte is necessary. If you use a whole memory form of a command then the chip will pick up the next two bytes for an address, even if that address happens to be in the zero page.

The form of the address

All of the addressing modes given above, except for one (immediate mode), require that an address, the position of a byte in memory, be specified. Not all machine code instructions actually require an address to be spelled out in the program, since some work on fixed locations such as the accumulator or one of the flags in the status register. When an address does have to be specified, it will be in one of two forms, according to whether the instruction is one that relates to the zero page or to the whole of the memory.

The address for a zero-page instruction

We have already noted that zero-page instructions are faster for the simple reason that any address in zero page can be specified in only one byte. Accordingly, a zero-page instruction will take the form (leaving aside extras such as those added in indexed instructions):

ZERO PAGE INSTRUCTION <ONE BYTE NUMBER> (specifying an address in the range 0–255)

The address for a whole memory instruction

Once we move outside the first 256 bytes of memory, a single byte is no longer enough to specify the address, and two bytes will need to be used. Using two bytes, the largest number which can be represented is 255*256 (the high byte) + 255 (the low byte or units), which equals 65535. This is quite enough, since the 6510 chip itself is only capable of recognising the presence of memory bytes numbered zero to 65535.

One important proviso has to be born in mind however, when entering an address into a program, and that is that the two bytes must be back-to-front. If the address to be specified were, for instance, 1024, which is equal to 4*256 + 1, we would logically expect the two bytes

necessary to specify the address in hexadecimal to be 04 followed by 01. *This is not the case.* For reasons which we shall not go into, the fact is that the address must be specified as 01 followed by 04. The rule is then:

When specifying a two byte address, the low byte (or units) comes first, the high byte second.

The strange reversal is a major source of confusion for those inexperienced in machine code and frequently catches even experienced programmers out. Other two byte values in your programs do not need to be reversed — indeed reversing them will make nonsense of them. Unless you are using an assembler program *(see below)* there is no way around this seeming illogicality, you must simply remember that if it is an address then the two bytes which make it up have to be reversed. When you run into inexplicable problems with your programs, it is wise to make one of your first checks on the form of the addresses the program contains. If you intended to refer to address 0401 and it appears as 0401 in the machine code program, then what you are actually referring to is address 0104.

The format for a whole memory instruction is, therefore:

WHOLE MEMORY INSTRUCTION <LOW BYTE> <HIGH BYTE>

Crossing the zero-page boundary

Clearly, if an address is outside the first 256 bytes of memory, then a single byte address cannot be used to specify it. The same is not true in reverse, however. If for some strange reason you do want to use a whole memory instruction to refer to something which is in the zero-page, then the address must be specified in the form:

INSTRUCTION <LOW BYTE> <HIGH BYTE (00)>

It is vital that these rules about the length of the address are adhered to if you are working directly in machine code rather than using an assembler. If you use a zero-page form of an instruction and then place a two byte address after it, the 6510 will take the first byte of the number as the address. The second byte will be regarded as if it is the beginning of another instruction. If, on the other hand, you use a whole-memory form of an instruction and follow it with a single byte address, the chip will take your one-byte address and the first byte of the following instruction. In both cases the chip will probably now be hopelessly out of synchronisation with the program as you wrote it and will start interpreting commands which you never intended to be there or jumping to unintended places.

Addresses with an assembler

One way around all of these problems is to use an Assembler program, which is designed to take instructions in assembly language and translate them into the pure numbers of machine code. A properly written assembler program will not allow you to make any of the mistakes outlined above. If you specify an action which has both a whole memory form and a zero page one, then the assembler will look at the address you have entered and choose the appropriate form. If it is possible to use the zero-page form, it always will, since that will be the quicker form of the command. If an instruction does not have a form appropriate to the type of address you have specified, the assembler will refuse to assemble the instruction and issue you with a warning that you are trying to do something impossible. Apart from ease of use, this protection against a very common form of error is one of the best reasons you should consider purchasing an assembler program if you do not already own one.

CHAPTER 6
Addressing Modes in Detail

Having looked at some of the theory behind the way addresses are specified, we now turn to the specifics, with an analysis of the 14 ways in which an address can be specified on the 6510 chip. In reading this chapter, however, it must be remembered that not every instruction can be used with every form of addressing. The commands which have the most addressing modes available, like LDA, have eight, while a number of commands have only one addressing mode.

Part 1: Non-memory forms

Not every number which needs to be handled will be specified in the form of an address in memory, they carry the address built in or specify the number directly.

Inherent addressing — the address built in to the command

The command TAX, for instance stands for 'Transfer the contents of the accumulator to the X register'. Since both of these registers are inside the chip, they do not have an address as such. When it encounters the TAX command the chip knows exactly where to get the number to be handled from and where to send it to — it is inherent in the command itself.

Commands which take the inherent addressing mode will stand alone, having no address or number following them.

Accumulator addressing — a built in reference to the accumulator

There are four commands, ASL, LSR, ROL and ROR (all of which will be explained in Chapter 12) which use a special form of inherent addressing called *accumulator addressing*. All of these commands perform a preset task on the contents of the accumulator, without anything having to be specified.

Once again, commands which take the accumulator addressing mode will stand alone.

Immediate Addressing — specifying a value in the program
When an instruction is found which uses immediate addressing, it will be in the form:

INSTRUCTION#<NUMBER>(in the range 0–255)

And on finding the instruction, like LDA (LoaD Accumulator) the chip will take the number following the command in the program and place it into the accumulator. In an assembly language program the number itself will be preceded by the '#' symbol, the hallmark of the immediate mode.

Part 2: Whole memory addressing

Absolute addressing — specifying a position in memory
Here the CPU is instructed to act on a value which is contained in a memory byte whose address is specified within the machine code program. The address is given and the CPU goes directly to that byte to find the number on which it is to work. The form of the instruction for absolute addressing is:

INSTRUCTION <TWO BYTE NUMBER> (in the range 0–65535)

Indexed Absolute Addressing — accessing part of a table
We have already discussed the concept behind indexed addressing, which allows the start of a table to be obtained and then a particular value within that table to be specified.

In real life you need first to note that it is a form of absolute addressing, by which is meant that the start position of the table is given directly in the program in the same way as an address is specified in normal absolute addressing mode. This, however, is only half the process, since the position of the desired byte within the table also has to be given. This index will be contained in one or other (depending on the exact instruction) of the 6510's X and Y registers. The format for indexed addressing is:

INSTRUCTION <TWO BYTE ADDRESS>,<NAME OF REGISTER>

Thus, if the two byte address were to $0401 (1025 decimal) and the register referred to were to be the X register, which at that moment contained the value 16, the byte to be acted upon would be $0401 + $10, or $0411.

Indirect addressing — finding an address from an address
We have already noted that there will be many cases in programming where we will not necessarily know the address which is to be worked upon until the program is running. In this case, we need to be able to look at a specified byte in the memory and to pick up from it not the number that we want, but the *address* at which we can find the value that we want. The format for an indirect addressing mode command is:

INSTRUCTION (<TWO BYTE ADDRESS>)

In an assembly language program the address would have brackets placed around it. This means 'take the contents of this address and use it for something'. If the bracketed number is in the position of an address then the brackets mean 'take the contents of this address and use them as an address'.

One thing to note about indirect addressing is that what must be pulled back from memory first of all is *two* bytes representing an address. The initial address in brackets must therefore point to the first of two bytes which must be in the usual low/high order of an address.

Relative Addressing — a position in regard to the current command
This is the form of addressing which is sometimes used with *jump* instructions — commands used to tell the chip to move to another part of the machine code program. Relative addresses are single byte numbers and they are different from all other forms in that they are regarded by the chip as signed numbers — *i.e.* they can be either positive or negative. In the chapter on binary arithmetic we saw that this means that instead of covering a range of zero to 255, the range which can be expressed becomes −128 to +127. As with any signed number, the most signficant bit will indicate whether the value is negative or not. Provided that you followed the description of signed numbers in earlier chapters, then relative addressing will hold no terrors for you. If you are not quite clear on how to use signed numbers, go back to the previous descriptions.

The format for an instruction in relative addressing mode is:

INSTRUCTION <SINGLE BYTE SIGNED NUMBER>

Part 3: Zero page modes

Absolute — specifying an address in the zero page
This is no different in effect from the non-zero page version. The zero-

page instruction specifies a single byte address at which the number to be held is found.

The format of a zero page absolute instruction is:

INSTRUCTION <SINGLE BYTE ADDRESS>

Zero page indexed — accessing a table in the zero page

Again, the difference from the non-zero form is that instead of a two-byte address for the start of the table, a single byte address is all that is necessary. Note however that the zero-page form of the indexed addressing mode requires that the eventual address, arrived at by adding the contents of the X or Y register (depending on the command) to the specified start address, must fall in the zero page.

The format for a zero page indexed command is:

INSTRUCTION <SINGLE BYTE ADDRESS>, <NAME OF REGISTER> (either X or Y)

Pre-indexed indirect — using a table to obtain an address

The essence of pre-indexed indirect addressing is that it is the indirect form of normal indexed addressing. That means that instead of taking a value from a position in a table, it takes a two byte address from a table and then picks up the number at that address. The register used for pre-indexed indirect addressing is always the X register.

The format for a pre-indexed indirect command is:

INSTRUCTION <SINGLE BYTE ADDRESS,X>

Post-indexed indirect — indexing an address previously obtained from the memory

The last and most unusual form of zero page addressing. Here a two byte address is picked up from a specified position in zero page memory, then the contents of the Y register are added to this address. This final form of the address is then used to pick up a value from the memory.

The format for a post-indexed indirect command is as follows:

INSTRUCTION <SINGLE BYTE ADDRESS>,Y

Conclusion

You now know the 11 possible addressing modes available on the 6510

chip — though some would say there are 13 since the indexed commands can both be used with X or Y registers. The formats you have been given above show what each of the addressing modes would look like in an assembly language program — if you want to work directly in machine code you will have to choose the addressing mode and command you need and look up the correct instruction form in the tables throughout this book. In addition, given below is a list of the assembly language formats so that in reading an assembly language program you can quickly refer to the table if you are unsure as to which addressing mode a particular format represents. In the list, an imaginary opcode called MEM is used in place of a real instruction, since there is no one command which is capable of using all the addressing modes:

MEM#<1 BYTE NUMBER> — IMMEDIATE

MEM <2 BYTE ADDRESS> — ABSOLUTE

MEM <1 BYTE ADDRESS> — ZERO PAGE (ABSOLUTE)

MEM — INHERENT

MEM A — ACCUMULATOR

MEM (<1 BYTE ADDRESS>,X) — PRE-INDEXED INDIRECT

MEM (<1 BYTE ADDRESS>),Y — POST-INDEXED INDIRECT

MEM <1 BYTE ADDRESS>,X — ZERO PAGE INDEXED, X

MEM <1 BYTE ADDRESS>,Y — ZERO PAGE INDEXED, Y

MEM <2 BYTE ADDRESS>,X — ABSOLUTE INDEXED, X

MEM <2 BYTE ADDRESS>,Y — ABSOLUTE INDEXED, Y

MEM <1 BYTE SIGNED NUMBER> — RELATIVE

MEM (<2 BYTE ADDRESS>) — INDIRECT

CHAPTER 7
Loading and Storing with A, X, and Y

With all the background out of the way it is time to turn to the actual machine code that is the purpose of the book. We shall start with one of the most easily understandable groups of instructions, those which load values into the chip registers and those which take values from registers and store them elsewhere.

The Load instructions — LDA, LDX and LDY

Function: To copy a value from a specified location into one or other of the CPU registers.

Comparing what we are doing with BASIC, loading a register is rather similar to setting the value of a variable with a statement such as:

LET X=10

In BASIC, the usual reason for storing a value in a variable is so that things can be done to or with that variable. The same is true of machine code. Most actions which can be carried out depend to some extent on the values which one or more of the registers contain and in most cases the result of a calculation will be left in one of the registers when the calculation has been completed. Sometimes the result will be an end in itself, sometimes it will be important for some future decision and will need to be stored in some place that is in less demand than one of the registers. We therefore need to know how to place a value into one of the registers and, subsequently, how to extract that value and place it into a more permanent storage place.

The different ways of loading a register

Registers can be loaded from three different sources:

1) From the program itself, or immediate addressing mode. BASIC equivalent:

LET X=<ACTUAL NUMBER>

2) With the contents of a specified byte in the memory OR 'direct addressing'. Equivalent to:

LET X=<CONTENTS OF ANOTHER VARIABLE>

3) With the contents of a byte which is at a numbered position in a block of memory, or indexed addressing. BASIC equivalent:

LET X=<ARRAY ELEMENT(10)>

Not every register can be loaded from every location with a single command, so combinations of commands will often need to be used to make full use of all the registers. The list of addressing modes available with the different load instructions is given in the table at the end of this section but to demonstrate some of them in action, load up the Code Runner program and enter the extra lines of data specified in the examples given below.

Example of loading a register direct from the program

```
ADD.   DATA       SOURCE CODE
00                   10 PRT
00                   20 ORG $C030
C030                 30 SYM
C030                 40 FINISH = $C013
C030   A2FB         50 LDX #$FB
C032   4C13C0       60 JMP FINISH
C035                 70 END
```

```
6000 REM#*********************************
6010 REM DATA FOR MACHINE CODE
6020 REM*********************************
6030 DATA A2,FB
```

Commentary
Lines 10–40 and 60–70: As you should remember from Chapter 4, these lines have nothing to do with the working machine code but are instructions to the assembler program which was used to place the machine code into the memory. In future examples, these lines will be ignored in any commentary.

Line 50: LOAD the X register with the hexadecimal value FB. The '#'

symbol informs the CPU that what is to be loaded is the value specified in the instruction. The '$' sign is the symbol conventionally used to indicate that a value which follows is expressed in hexadecimal. Comparing the format of the address with those given in the last chapter you will find that the instruction is using the immediate addressing mode.

If you are using the BASIC Code Runner program given in Chapter 4, then to see this instruction in action, add the four lines of BASIC to the program and then RUN it. Answer 'N' to the questions about using high memory or altering memory locations since these are irrelevant — the value to be loaded is contained within the program itself.

When the program has stopped, examining the display of registers will show that the hexadecimal value FB has been placed into the X register.

This straightforward loading of a value into a register can be performed on all three of the main registers by the instructions LDA, LDX and LDY, represented in hexadecimal by the codes A9, A2 (as in the current example) and A0. A complete list of the instructions and formats for loading the registers is given at the end of the chapter.

Example of loading from memory

```
ADD.   DATA       SOURCE CODE
00                   10 PRT
00                   20 ORG $C030
C030                 30 SYM
C030                 40 FINISH = $C013
C030   A4FB         50 LDY $FB
C032   4C13C0       60 JMP FINISH
C035                 70 END
```

```
6000 REM#********************************
6010 REM DATA FOR MACHINE CODE
6020 REM********************************
6030 DATA A4,FB
```

Commentary

Line 50: Here the format of the instruction shows that the zero-page addressing mode is being employed to copy a value from a location in memory (address $FB) into the Y register. Note the word 'copy', since the load instruction has no effect on the value contained in memory location $FB. Note the absence of the '#' symbol used in the first example, showing that here $FB is an address rather than a literal number.

Demo procedure: RUN the program, and on the second prompt, answer 'Y' to allow you to set the value of the four memory bytes in zero page. Set the value of byte $FB to some desired value between 0 and 255. When the machine code has executed, you should find that the Y register has been loaded with whatever value you placed in memory location $FB.

Example of loading from a position within a block of memory

```
ADD.   DATA       SOURCE CODE
00                  10 PRT
00                  20 ORG $C030
C030                30 SYM
C030                40 FINISH = $C013
C030   A2FB         50 LDX #$FB
C032   B500         60 LDA 0.X
C034   4C13C0       70 JMP FINISH
C037                80 END
```

```
6000 REM*********************************
6010 REM DATA FOR MACHINE CODE
6020 REM*********************************
6030 DATA A2,FB,B5,00
```

Commentary
In BASIC, it is very often useful to define a table of values within an array and then to use individual values from that table by using statements like:

LET TT=ARRAY (5)

In doing this, we take it for granted that the BASIC interpreter knows just where to find ARRAY and then how to identify the value in position five. No such facility is built into machine code. But it is possible to simulate a table by indexed addressing.

In indexed addressing, either the X or the Y register is first loaded with a value (you already know how to do that) and then the CPU is given a value (START) by the program which represents the start of a block of memory which is to be used as a table. The value in the X or Y register is used to determine which byte in the table beginning at START is to be referred to. Having decided which byte is intended, the contents of the byte are then loaded into one of the registers, depending on the precise form of the instruction.

Like many other forms of addressing, indexed addressing comes in two

major forms, namely zero-page addressing and a main memory parallel form. A typical sequence of commands to accomplish indexed addressing (this time in zero-page mode) is represented by the example.

Line 50: LOAD the X REGISTER with the hexadecimal value $FB — this is the index value.

Line 60: LOAD the accumulator with the contents of the byte which is numbered 'zero + the contents of the X register'. The zero is the start of the table. Note that the *Mastercode Assembler*, on which the listing was produced, uses a full stop rather than a comma to separate the 'X' at the end of the instruction.

Demo procedure: RUN the program and set the value of byte $FB to a value between 0 and 255. You should find, when the program is executed, that whatever value you have placed into byte $FB is now copied into the accumulator.

For your own experiments, try altering the value $FB, which is placed into the X register, to one of FC, FD, or FE — the numbers of the other three bytes of memory set aside for the demo program. On running the program again, you should find that the accumulator is loaded from whichever byte has the same number as that placed into the X register.

Example of loading the accumulator in absolute addressing mode

```
ADD.   DATA      SOURCE CODE
00                 10 PRT
00                 20 ORG $C030
C030               30 SYM
C030               40 FINISH = $C013
C030   ADFBC2      50 LDA $C2FB
C033   4C13C0      60 JMP FINISH
C036               70 END


6000 REM#********************************
6010 REM DATA FOR MACHINE CODE
6020 REM********************************
6030 DATA AD,FB,C2
```

Commentary

Line 50: Here a single instruction is all that is needed to pick up the value

stored in the high part of memory at $C2FB. This is one of the memory locations which the Code Runner program allows you to alter, so you can experiment with different values.

Example of loading the accumulator in absolute indexed addressing mode

```
ADD.   DATA      SOURCE CODE
00                 10 PRT
00                 20 ORG $C030
C030               30 SYM
C030               40 FINISH = $C013
C030   A2FB        50 LDX #$FB
C032   BD00C2      60 LDA $C200.X
C035   4C13C0      70 JMP FINISH
C038               80 END
```

```
6000 REM*********************************
6010 REM DATA FOR MACHINE CODE
6020 REM*********************************
6030 DATA A2,FB,BD,00,C2
```

Commentary
Line 50: The X register, which will be used as the index register, takes its values from $FB in the low memory, a byte you can set yourself.

Line 60: The accumulator is loaded with the contents of the memory location whose address is $C200 plus the value in the X register. For your own experiments, you might like to change $FB to a value which will make $C200 + X point to one of the high memory locations you can change, so that you can specify the value which is loaded into the accumulator.

Table of load instructions and available addressing modes
The addressing modes available for use with LDA are:

Zero page	: LDA $FF	: A5	: A5 FF
Absolute	: LDA $02FF	: AD	: AD FF 02
Immediate	: LDS#$FF	: A9	: A9 FF
Pre-indexed indirect	: LDA ($FF,X)	: A1	: A1 FF
Post indexed indirect	: LDA ($FF),Y	: B1	: B1 FF

Zero page indexed, X	: LDA $FF,X	: B5	: B5 FF
Absolute indexed, X	: LDA $02FF,X	: BD	: BD FF 02
Absolute indexed, Y	: LDA $02FF,Y	: B9	: B9 FF 02

The addressing modes available for use with LDX are:

Zero page	: LDX $FF	: A6	: A6 FF
Absolute	: LDX $02FF	: AE	: AE FF 02
Immediate	: LDX#$FF	: A2	: A2 FF
Zero page indexed, Y	: LDX $FF,Y	: B6	: B6 FF
Absolute indexed, Y	: LDX $02FF,Y	: BE	: BE FF 02

The addressing modes available for use with LDY are:

Zero page	: LDY $FF	: A4	: A4 FF
Absolute	: LDY $02FF	: AC	: AC FF 02
Immediate	: LDY#$FF	: A0	: A0 FF
Zero page indexed, X	: LDY $FF,X	: B4	: B4 FF
Absolute indexed, X	: LDY $02FF,X	: BC	: BC FF 02

Transferring between registers — the TXA, TAX, TYA, TAY, TXS and TSX instructions

Function: To copy a value from one specified CPU register to another.

Another method of placing a value into one of the three main CPU registers or the stack pointer is to use the transfer instructions. In assembly language, these instructions begin with 'T' followed by two letters representing the register a value is being copied (or transferred) from and the register it is being copied (or transferred) to. Thus:

TAX — 'transfer the contents of the accumulator to the X register'.
TXS — 'transfer the contents of the X register to the stack pointer'.
TYA — 'transfer the contents of the Y register to the accumulator'.

Example of transferring the contents of the accumulator to the X register

```
ADD.   DATA      SOURCE CODE
00                 10 PRT
00                 20 ORG $C030
C030               30 SYM
C030               40 FINISH = $C013
C030   A2FF        50 LDX #$FF
```

```
C032   A9AA        60 LDA #$AA
C034   AA          70 TAX
C035   4C13C0      80 JMP FINISH
C038               90 END

6000 REM**********************************
6010 REM DATA FOR MACHINE CODE
6020 REM**********************************
6030 DATA A2,FF,A9,AA,AA
```

Commentary
Lines 50–60: The accumulator is loaded with $AA and the X register with $FF. There is no particular reason for these values, it simply helps to distinguish between the two registers.

Line 70: A single command is all that is necessary to copy the contents of the accumulator into the X register.

Example of the use of TYA

```
ADD.   DATA      SOURCE CODE
00                 10 PRT
00                 20 ORG $C030
C030               30 SYM
C030               40 FINISH = $C013
C030   A0FF        50 LDY #$FF
C032   A9AA        60 LDA #$AA
C034   98          70 TYA
C035   4C13C0      80 JMP FINISH
C038               90 END

6000 REM**********************************
6010 REM DATA FOR MACHINE CODE
6020 REM**********************************
6030 DATA A0,FF,A9,AA,98
```

Commentary
The reverse of the procedure shown in the last example. Here the Y register and the accumulator are loaded with $AA and $FF respectively, then the value in the Y register transferred to the accumulator with a single command.

Table of transfer commands

Note: all the transfer commands use the single addressing mode 'inherent' — the instructions themselves specify all the locations involved and there is no need for any further address.

Transfer A to X	: TAX	: AA	: AA
Transfer X to A	: TXA	: 8A	: 8A
Transfer S to X	: TSX	: BA	: BA
Transfer X to S	: TXS	: 9A	: 9A
Transfer A to Y	: TAY	: A8	: A8
Transfer Y to A	: TYA	: 98	: 98

Loading values into the stack pointer, status register and program counter

Though the main registers to be used will always be the accumulator and the X and Y registers, it is wise to remember that there are three other registers into which values can be placed, though the effect of such action needs to be considered carefully before it is undertaken. When placing values into these registers there is no simple load command and we shall usually have to make use of combinations of commands.

Example of loading the stack pointer

```
ADD.   DATA      SOURCE CODE
00                 10 PRT
00                 20 ORG $C030
C030               30 SYM
C030               40 FINISH = $C013
C030   A6FB        50 LDX $FB
C032   9A          60 TXS
C033   4C13C0      70 JMP FINISH
C036               80 END
```

```
6000 REM*********************************
6010 REM DATA FOR MACHINE CODE
6020 REM*********************************
6030 DATA A6,FB,9A
```

Commentary
NB: The use of the stack itself will be discussed in a separate chapter.

There is no command to load the stack pointer directly from memory but as we have seen, there *is* a command to load a value from the X register. Since there is no difficulty in loading a value from memory into the X register, the stack pointer can be loaded in two stages by first loading the desired value into the X register and then transferring it to the stack pointer.

Line 50: LOAD the X register with the contents of the byte at $FB in zero page.

Line 60: Transfer the contents of the X register to the stack pointer.

Demo procedure: RUN the program and set the value contained by the byte $FB. The result of running the machine code will be that this value is placed into the X register and also into the stack pointer.

Example of loading the status register

```
ADD.   DATA       SOURCE CODE
00                  10 PRT
00                  20 ORG $C030
C030                30 SYM
C030                40 FINISH = $C013
C030   A5FB         50 LDA $FB
C032   48           60 PHA
C033   28           65 PLP
C034   4C13C0       70 JMP FINISH
C037                80 END
```

```
6000 REM#################################
6010 REM DATA FOR MACHINE CODE
6020 REM#################################
6030 DATA A5,FB,48,28
```

Commentary
Though there is a command to allow it to be done, there are very few occasions on which it would be desirable or necessary to load the status register as a whole. As we have already seen, in relation to our imaginary Lawland 10000 chip, the eight bits of the status register are not so much a single eight bit value but eight separate indicators (though only seven are used). Setting the whole of the register at once will have a major impact but it might be of use in resetting the whole system to some previous state

— as indeed the DEMO program itself does, first saving the contents of the status register and then, after the machine code has been executed, restoring the system to its original state, thus ensuring that any flags which might crash the system are reset to values which will allow BASIC to function.

Lines 50–60: The value contained in memory location $FB is loaded into the accumulator and then placed onto the stack using the PusH Accumulator instruction which will be explained more fully in Chapter 16. At this point all you need to know is that in machine code terms, 'push' means place a value onto the stack.

Line 65: The value placed on the stack is loaded into the status register using the PuL1 Processor status register. A 'pull' instruction is the opposite to a 'push' in that it takes the last value off the stack.

Demo procedure: RUN the program and place a chosen value into memory location $FB. When the machine code has been executed you should find that the display shows your chosen value in the status register — in fact it will have been there only for a moment and will then have been removed by the Code Runner program itself.

Example of loading the program counter

```
ADD.   DATA      SOURCE CODE
00                 10 PRT
00                 20 ORG $C030
C030               30 SYM
C030               40 FINISH = $C013
C030   4C13C0      50 JMP FINISH
C033               60 END
```

```
6000 REM**********************************
6010 REM DATA FOR MACHINE CODE
6020 REM**********************************
6030 DATA 4C,13,C0
```

Commentary
We are left with only one register which we do not know how to load — the program counter. It is sometimes said that there is no command to

load this register, but this is not the case. There is a whole set of instructions to load the program counter but they are not called load instructions, they are called *jump* instructions. Since the program counter has as its task to record the point at which the machine code program is to continue execution, loading a new value into it is equivalent to jumping to another part of the program.

Line 50: Rather than do anything fancy, we use the same jump which actually ends all the machine code demonstrations and is built into the DATA module at 7000 in the DEMO program. The effect of the JMP instruction is to send program execution to the little routine which tidies up the memory after the machine code.

Demo procedure: RUN the program. The important thing to note is that the picture of the program counter taken at the end of the machine code shows the address to which the jump was made — *i.e.* we loaded the program counter with $C013.

Storing the contents of registers

The STA, STX and STY instructions

We have now arrived at the stage where we can take a value from almost anywhere in the memory and place it into one or other of the 6510's registers. Later on, we shall look at the methods employed in *doing* something with what we have transferred. Before that, however, we shall explore the opposite side of the coin to loading the registers, namely, placing the contents of a register into a chosen memory location. As with the previous section, we shall also look at some of the different ways of addressing memory which are allowed when the transfer is made, but only some. The full list of the addressing modes available when storing the contents of a register will be found in the table at the end of the section.

Example of storing the contents of the accumulator

```
ADD.   DATA      SOURCE CODE
00                 10 PRT
00                 20 ORG $C030
C030               30 SYM
C030               40 FINISH = $C013
C030   A5FB        50 LDA $FB
C032   A2FC        60 LDX #$FC
```

```
C034  9500        70 STA 0,X
C036  4C13C0      80 JMP FINISH
C039              90 END
```

```
6000 REM*********************************
6010 REM DATA FOR MACHINE CODE
6020 REM*********************************
6030 DATA A5,FB,A2,FC,95,00
```

Commentary
Our first example in this section shows how the contents of the accumulator can be stored in zero page memory using the X register for indexed addressing. The assembly language instructions involved are:

Line 50: LoaD the Accumulator with the contents of memory byte $FB.

Line 60: LoaD the X register with the number $FC.

Line 70: STore the contents of the Accumulator at the address represented by zero plus the contents of the X register.

Demo procedure: RUN the program and set the contents of byte $FB to a value between 0 and 255. Now run the program. You should find that when the machine code has been executed, you are left with the same value in $FB, the accumulator and $FC. This is clearly because the contents of $FB were copied into the accumulator and from there copied into $FC. The contents of the X register should be $FC, indicating the byte in zero page at which the accumulator contents were stored.

Apart from the accumulator, you can also carry out this zero page indexed storage with the accumulator and Y register, using the X register to hold the index number, or with the X register, using the Y register to hold the index number. An example storing the X register might be:

```
LDX  $FB
LDY  #$FC
STX  0,Y
```

To demonstrate this sequence, load the following into the DEMO program:

```
ADD.  DATA      SOURCE CODE
00                10 PRT
```

```
00                    20 ORG $C030
C030                  30 SYM
C030                  40 FINISH = $C013
C030  A6FB            50 LDX $FB
C032  A0FC            60 LDY #$FC
C034  9600            70 STX 0,Y
C036  4C13C0          80 JMP FINISH
C039                  90 END
```

```
6000 REM**********************************
6010 REM DATA FOR MACHINE CODE
6020 REM**********************************
6030 DATA A6,FB,A0,FC,96,00
```

RUN the program and give a value to byte $FB. After the execution of the machine code, $FB, the X register and $FC should all contain the value which you entered. The Y register should contain the value $FC.

Storing the contents of the status register, stack pointer and program counter

While the three main registers are well catered for when it comes to storing their contents in memory, the other three registers require a little more thought and the use of several intermediate steps. We shall begin with the task of storing the status register.

Example of storing the contents of the status register

```
ADD.   DATA      SOURCE CODE
00                 10 PRT
00                 20 ORG $C030
C030               30 SYM
C030               40 FINISH = $C013
C030   08          50 PHP
C031   68          60 PLA
C032   85FC        70 STA $FC
C034   4C13C0      80 JMP FINISH
C037               90 END
```

```
6000 REM#********************************
6010 REM DATA FOR MACHINE CODE
6020 REM********************************
6030 DATA 08,68,85,FC
```

Commentary
Lines 50–60: Once again push and pull instructions which will be explained in Chapter 16 are used, first of all to place the contents of the status register onto the stack and then to recall them into the accumulator.

Line 70: Having transferred the contents to the accumulator, there is now no problem in transferring them to memory using a store instruction.

Example of storing the contents of the stack pointer

```
ADD.   DATA      SOURCE CODE
00                 10 PRT
00                 20 ORG $C030
C030               30 SYM
C030               40 FINISH = $C013
C030   BA          50 TSX
C031   86FC        60 STX $FC
C033   4C13C0      70 JMP FINISH
C036               80 END
```

```
6000 REM#********************************
6010 REM DATA FOR MACHINE CODE
6020 REM********************************
6030 DATA BA,86,FC
```

Commentary
We have already seen that the way to move the stack pointer to another register is to transfer it to the X register — from there it is a simple matter to transfer to a specified memory location.

Storing the contents of the program counter
The procedure for storing the contents of the program counter is a rather complex one and will be given in Chapter 16, by which time the instructions needed will all have been explained.

Table of store instructions together with their available addressing modes
The addressing modes available for use with STA are:

Zero page	: STA $FF	: 85	: 85 FF
Absolute	: STA $02FF	: 8D	: 8D FF 02
Pre-indexed indirect	: STA ($FF,X)	: 81	: 81 FF
Post indexed indirect	: STA ($FF),Y	: 91	: 91 FF
Zero page indexed, X	: STA $FF,X	: 95	: 95 FF
Absolute indexed, X	: STA $02FF,X	: 9D	: 9D FF 02
Absolute indexed, Y	: STA $02FF, Y	: 99	: 99 FF 02

The addressing modes available for use with STX are:

Zero page	: STX $FF	: 86	: 86 FF
Absolute	: STX $02FF	: 8E	: 8E FF 02
Zero page indexed, Y	: STX $FF,Y	: 96	: 96 FF

The addressing modes available for use with STY are:

Zero page	: STY $FF	: 84	: 84 FF
Absolute	: STY $02FF	: 8C	: 8C FF 02
Zero page indexed, X	: STY $FF,X	: 94	: 94 FF

CHAPTER 8
The Flags

Having started on the instructions necessary to write a machine code program we take a pause for a little very necessary explanation. Before we go on to consider a whole range of instructions we shall need to be clear on the function of the flags, which are crucial to many of the decisions a machine code program will make as it runs. Because flags are not so much important in themselves but in their effect on other instructions, we shall, in the course of the chapter, use one or two instructions with which you are not yet familiar. In each case we shall give a rough indication of the purpose of a command but shall avoid detail — the only alternative would be to explain commands without having dealt with the flags that are essential for their use — in other words you can't win.

What is a flag?

In the early chapters of this book, when looking at the imaginary Lawland 10000, we discovered that there were a collection of seven special locations capable of storing yes/no decisions in the form of either a 1 or a 0. While not looking at them in any detail, we did note that when arithmetic was being done, for instance, the crucial fact of whether a carry had been generated was stored in a flag. We also noted that each flag was a single bit in a special register within the CPU, called the status register. Like all the 6510 registers except the program counter, the status register is eight bits long, but in fact only seven of the bits are used. Each of the seven bits is used to record one yes/no decision or result of an action taken by the CPU. The seven flags are:

Bit 0 — Carry flag
Bit 1 — Zero flag
Bit 2 — Interrupt flag
Bit 3 — Decimal flag
Bit 4 — Break flag
Bit 5 — Not used (usually found set at 1)
Bit 6 — Overflow flag
Bit 7 — Sign flag

Each flag will be explained in turn but before you move on to the explanations, do be sure that you understand that each flag is the answer to a single question, with the answer 'yes' if the flag is at 1 and 'no' if the flag is at 0. This may seem a trivial thing to stress but so many machine code programmers do become confused as to whether, for instance, the zero flag is set when a zero is created or whether it is at zero in accordance with the zero that has been created. Far simpler, and far more reliable, to remember that the zero flag answers the question 'Has a zero been created?'. In the sections which follow on each flag, the questions that they answer are spelled out and will more than repay a little time spent memorising them.

The carry flag

Function: To record whether the last action involving the accumulator resulted in a value being created which was outside the range which can be held in eight bits. The question which the flag answers is therefore, 'Did the last operation carried out result in the creation of a number greater than 255 or less than zero?'

This is the flag which you will find yourself using more than any other. Indeed, there are a range of actions which the CPU carries out which are directly based on the value of the carry flag, without the user having to specify that it be taken into account. Its use in a program differs slightly according to whether signed or unsigned arithmetic is being used but its job is always to record whether an operation performed on the contents of the accumulator has resulted in, or should have resulted in, the creation of a value over 255.

Having said that, there are one or two instructions which operate on values contained in locations in memory, not in the accumulator, but are capable of setting the carry flag. The instructions, *shift* and *rotate*, will be dealt with in Chapter 12.

Note that the carry flag is affected by *every operation* involving the accumulator. If one operation involves a carry, then the carry flag will be set — if another operation follows which results in a value in the range 0–255, the carry flag will be cleared. Any use of the carry flag to test the result of a particular operation must therefore be carried out immediately, before any further use is made of the accumulator.

The carry flag and unsigned arithmetic

This is the simplest of the two usages. With unsigned numbers, the only operation which can lead to a value greater than 255 being created, when using the accumulator, is to add a number to that which is already stored in the accumulator. The format for the ADD command and examples of

its use will be given in a separate chapter. At this moment we simply assert that a series of commands such as:

LDA #200 (LOAD the ACCUMULATOR with 200)
ADC #200 (ADD 200 to ACCUMULATOR, recording any CARRY)

will set the carry flag if the value created in the accumulator *should be* over 255. Since the maximum value which the accumulator can hold is 255, or eight binary digits all set to one, the result created will be 256 less than the correct figure — the ninth bit, representing 256 in binary is simply lost. When this happens, the carry flag is set, thus becoming effectively a ninth bit for the accumulator, allowing values up to nine binary digits in length (up to 511 in decimal) to be created.

In the case of numbers made up of more than a single byte, the carry flag is an integral part of the process of adding the successive bytes together, allowing the results of each addition to carry over from the most significant digit of one byte to the least significant digit of the next most significant byte in the same way as is done automatically between digits within the same byte.

The carry flag and signed arithmetic

When using signed numbers the use of the carry flag is not quite as straightforward as with unsigned. Signed numbers, as we have already seen, use the most significant binary digit as an indicator of whether a number is negative or not. For this reason, the numbers which can be recorded in a single byte are in the range −128 to +127. Adding together two such numbers may result in the most significant binary digit being changed but it can never result in a number too large to be held in eight binary digits. Since any change in the sign bit is the province of another flag, the overflow flag, the carry flag has no role to play when single byte signed numbers are being added together in the accumulator.

Having said that, the carry flag often has a major role to play in signed arithmetic for the simple reason that machine code programs frequently make use of two-byte, and larger, signed numbers. To set up a two byte signed number, which can be in the range −32768 to 32767, we still use the most significant binary digit to record whether the number is positive or negative but only the most significant binary digit of the high byte. For instance, the number 511 can be recorded in two bytes as follows:

HIGH BYTE	LOW BYTE
00000001	11111111

Here you can see the most significant binary digit, the zero on the left,

indicating that the number is positive. The righthand binary digit of the high byte is set to one, indicating that the number contains 256. The low byte has every binary digit set, recording a value of 255 — not −127 since there is no need to have a sign bit on the second byte.

When we come to add together two signed values made up of more than one byte (eg 511+511), the high bytes behave just like single byte signed numbers and make no use of the carry flag. In adding together any bytes to the right of the high byte the carry flag *is* used, since what is being added together are two unsigned values.

Instructions which refer to the carry flag

In addition to instructions which alter the value of the accumulator, and therefore potentially set or clear the carry flag, there are a number of instructions which act either directly on it or act on the basis of its contents:

SEC (SEt Carry) — switches the carry flag to one, regardless of its present state.

CLC (CLear Carry) — switches the carry flag to zero, regardless of its present state.

BCS (Branch Carry Set) — program execution branches to a specified point, if the carry flag is set to one when this instruction is encountered.

BCC (Branch Carry Clear) — program execution branches to a specified point if the carry flag is clear when this instruction is encountered. For a fuller explanation and demonstration of these two branch instructions, see the chapter on program control.

Demonstration routine for SEC

```
ADD.   DATA      SOURCE CODE
00                 10 PRT
00                 20 ORG $C030
C030               30 SYM
C030               40 FINISH = $C013
C030   A900        50 LDA #00
C032   48          60 PHA
C033   28          70 PLP
C034   38          80 SEC
C035   4C13C0      90 JMP FINISH
```

```
C038                100 END


6000 REM#********************************
6010 REM DATA FOR MACHINE CODE
6020 REM********************************
6030 DATA A9,00,48,28,38
```

Commentary
Lines 50–70: We have briefly mentioned push and pull instructions when explaining how to transfer a value to the status register. These lines load zero into the accumulator and then transfer it to the status register, thus ensuring that the carry flag is clear.

Line 80: The single command SEC suffices to set the carry flag.

Demo procedure: RUN the program and answer 'N' to all the options and you should find that the carry flag, the rightmost digit in the status register, is set to one.

Demonstration routine for CLC

```
ADD.   DATA      SOURCE CODE
00                 10 PRT
00                 20 ORG $C030
C030               30 SYM
C030               40 FINISH = $C013
C030   A9FF        50 LDA #$FF
C032   48          60 PHA
C033   28          70 PLP
C034   18          80 CLC
C035   4C13C0      90 JMP FINISH
C038              100 END


6000 REM#********************************
6010 REM DATA FOR MACHINE CODE
6020 REM********************************
6030 DATA A9,FF,48,28,18
```

Commentary
Lines 50–70: As in the previous program except that here the value $FF, or 255 decimal, is placed into the status register. In binary, $FF is 11111111, so placing it in the status register ensures that the carry flag, along with all the others, is set.

Line 80: Having set the carry flag, the single instruction CLC clears it.

Demo procedure: RUN the program and answer 'N' to all the options. You should find that the carry flag has been set to zero.

In later chapters you will come across many examples of the use of the carry flag and we shall rely on those rather than give routines here which would involve instructions you have not yet examined.

The zero flag

Function: To record whether the last operation carried out resulted in zero being created in a register or memory location. In the event that zero is created the flag is set to one, otherwise to zero. Commands which do not affect the zero flag include, NOP (no operation), any form of the store command, flag clearing instructions, program control branches and calls to subroutines. The question which the zero flag answers is; 'Did the last operation carried out result in the creation of a zero?'

The zero flag is similar to the carry flag in that it reflects the result of the last operation to be carried out, and so must be sampled immediately if its contents are to make any sense.

In practical use the zero flag is used most often for mundane jobs like determining whether a task has been carried out a specified number of times by placing a value in one of the registers and subtracting one from it every time the task is carried out. If the zero flag is sampled every time one is subtracted it will indicate when the number of subtractions equals the number that was originally in the register. The contents of the zero flag are often used for program control to jump to another part of a program when a certain value reaches zero.

Once again, you will find many examples of the use of the zero flag later in the book when we have had time to examine a wider selection of commands. No demonstration routine is given because all that is required to set the zero flag is to load zero into the accumulator (LDA#$0) and all that is required to clear the flag is to load the accumulator with a number other than zero (LDA#$1).

Instructions which refer to the zero flag
BNE (Branch Non-Equal) — program execution branches to a specified point if the zero flag is at zero when this instruction is encountered.

BEQ (Branch EQual) — program execution branches to a specified point if the zero flag is at one when this instruction is encountered.

Fuller details of the use of these two instructions will be given in the chapter on program control.

The interrupt flag

Function: To disable or enable the CPU interrupts which allow essential systems tasks to be carried out during breaks in the execution of the current program. The question which the flag answers is, 'Are the interrupts disabled?'

This is a flag which you will seldom use unless you are intending to work on extremely technical applications. Briefly, the CPU has one pin which is used to receive requests to stop the task which it is currently engaged on. The reason for this is that the CPU has many tasks to perform in running the 64, many of which have to be performed at regular intervals. While running a BASIC program for you, for instance, the CPU must also ensure that the keyboard is being scanned for any input from the outside world. When the system needs one of these regular tasks performed, it sends a request to the CPU to stop what it is doing for the moment and do something else. The CPU will then normally take a careful record of the point it had reached in its current task, turn to the new request and carry it out, then resume the original task where it was left off. This process is, of course, so fast that the user is quite unaware that the BASIC program which is running is constantly being stopped.

Some tasks, however, are designed not to be interrupted, like timing loops which would be rendered meaningless if the CPU were to stop executing them. In such cases, the 6510 makes provision for interrupts to be disabled. The requests to carry out a task are still made but the CPU ignores them and carries on with the current job in hand. The disabling is carried out by setting the interrupt flag to one. Switching the interrupt flag to zero allows the interrupts to continue normally.

As we have said, the number of occasions on which you are likely to want to disable the interrupts is extremely limited. If you do wish to disable them, be sure that you know what you are doing. Once the interrupts are switched off, the CPU is effectively isolated from the outside world and your program may lock up if it is not properly designed to restore the interrupts at the relevant points.

Commands which refer to the interrupt flag

SEI (SET Interrupt) — sets the interrupt flag to one and disables the interrupts.

CLI (Clear Interrupt) — sets the interrupt flag to zero and enables the interrupts.

No demonstration routines are given at this point since to switch the interrupt flag to one and not switch it back before terminating the machine code routine would disable the 64 so that you would have to switch it off in order to get control back. However the machine code routines attached to the Demo program do contain a CLI instruction to ensure that when you return to BASIC, the interrupts are switched back on if you have inadvertently left them switched off.

The decimal flag

Function: To switch the chip into decimal mode, where each byte records two decimal digits and calculations are performed according to the normal rules of the decimal numbering system. The question which the decimal flag answers is, 'Is the chip in decimal mode?'

The family of chips of which the 6510 is a member is the only major group of 8 bit CPU chips to have the facility to directly carry out decimal arithmetic — albeit in a rather limited fashion. This capability will be discussed in greater detail when we come to the chapter on the arithmetic instructions.

The sole purpose of the decimal flag is to set the chip into this unusual mode.

Instructions which refer to the decimal flag

SED (SEt Decimal) — the decimal flag is set to one when this instruction is encountered, placing the chip into decimal mode.

CLD (CLear Decimal) — the decimal flag is set to zero when this flag is encountered, returning the chip to normal binary mode.

The normal functioning of the 64 cannot take place in decimal mode — it is useful only for the performance of specific programming tasks and is always deactivated before the CPU returns to a 'waiting' state.

Demonstration of SED

```
ADD.   DATA      SOURCE CODE
00                 10 PRT
00                 20 ORG $C030
C030               30 SYM
C030               40 FINISH = $C013
C030   A900        50 LDA #00
```

```
C032  48          60 PHA
C033  28          70 PLP
C034  F8          80 SED
C035  4C13C0      90 JMP FINISH
C038             100 END
```

```
6000 REM#********************************
6010 REM DATA FOR MACHINE CODE
6020 REM********************************
6030 DATA A9,00,48,28,F8
```

Commentary
The status register is loaded with zero to clear all the flags, then a single SED suffices to set the flag.

Demonstration of CLD

```
ADD.  DATA      SOURCE CODE
00                10 PRT
00                20 ORG $C030
C030              30 SYM
C030              40 FINISH = $C013
C030  A9FF        50 LDA #$FF
C032  48          60 PHA
C033  28          70 PLP
C034  D8          80 CLD
C035  4C13C0      90 JMP FINISH
C038             100 END
```

```
6000 REM#********************************
6010 REM DATA FOR MACHINE CODE
6020 REM********************************
6030 DATA A9,FF,48,28,D8
```

Commentary
The opposite of the previous routine in that the flags are all set to one and the decimal flag cleared with a single SED.

The break flag

Function: To indicate whether an interrupt request received by the CPU is generated by the hardware of the system (flag value zero) or by the BRK instruction in a machine code program being executed (flag value one). The question which the flag answers is; 'Was the last interrupt which occurred a break instruction included in a machine code program?'

Having briefly discussed interrupts in relation to the interrupt flag, we now need to make a distinction between interrupts which are generated by the system which surrounds the CPU and those which are generated by a machine code program. There is a physical difference between the two, since hardware interrupts, the kind we have been talking about up to now, are signified to the CPU by means of the pins which connect the CPU to the rest of the system, while software interrupts come in the form of a specific machine code instruction (BRK) contained within a program — the uses of BRK will be discussed in a later chapter.

The problem which the break flag is designed to deal with is that the CPU is not immediately capable of distinguishing between a software interrupt and a hardware one, it does not therefore know whether it is being asked to go away and scan the keyboard or to perform a subroutine dictated by the machine code program. The problem is solved by the break flag since it is possible for the machine code programmer to write a short routine which, when an interrupt is detected, samples the break flag to see whether the interrupt is created by the system or whether there is a routine specified by the programmer to be carried out.

Other than the BRK instruction itself, which we shall not examine at this point, there are no instructions which refer to the break flag.

The overflow flag

Function: to detect, when using signed numbers, that a carry has occurred from the main part of the number, the first seven binary digits, into the eighth digit, which is used to record the sign of the number, thus altering the sign of the number. The question of when the sign of a number has been wrongly changed by a carry is quite a complicated one and will be dealt with in Chapter 10 but the question which the overflow flag answers is; 'Did the last operation carried out result in a signed number being given an invalid sign?'

We have already discussed the use of the carry flag in recording when an operation on a number in the accumulator results in the creation of a value over 255, which is more than can be held in 8 binary digits. We also noted that when working with single byte signed numbers, where the eighth bit is used to record the sign of the number, the carry flag is of no use to us since the part of the byte which makes up the actual value only occupies the first seven binary digits.

Take the example of the addition of two numbers +127 and +127, for which the signed representation in binary would be:

```
01111111
+
01111111
```

with the zero in the leftmost position indicating that both numbers are positive. The CPU will treat the two numbers as it would any other, it has no special operating mode for signed numbers — they are a creation of the programmer. The two 127s will therefore be added together to produce 254, or 11111110 in binary. This is fine as far as unsigned numbers go but nonsense if we are working with signed numbers since the 1 in the leftmost position means that the number is negative. In fact, by the conventional method of reading a signed number, the value produced is −126. Clearly, something has gone wrong and equally clearly it is that a carry has been made from the seventh position into the eighth, which contains the sign bit.

When this occurs, the overflow flag is set to one to indicate both that the sign bit has been reversed when it should not have been, and that 128 has to be added to the number stored in the accumulator. The overflow flag therefore acts as an extra binary digit for the accumulator but instead of providing a ninth digit for an unsigned number, as in the case of the carry flag, it operates as an eighth bit for signed numbers, thus allowing values in the range −256 to 255 to be dealt with, provided of course that programmer has designed the program to take account of the overflow bit.

Instructions which refer to the overflow bit

BVC (Branch if oVerflow Clear) — program execution jumps to a specified point if the overflow flag is set to zero when this instruction is encountered.

BVS (Branch if oVerflow Set) — program execution jumps to a specified point if the overflow is set to one when this instruction is encountered.

CLV (CLear oVerflow) — set the overflow flag to zero.

This instruction is used almost exclusively in routines which are intended to handle signed numbers — it has no general use in programming — and you will find it dealt with in the chapters on signed arithmetic.

Demonstration of CLV

```
ADD.   DATA      SOURCE CODE
00                 10 PRT
00                 20 ORG $C030
C030               30 SYM
C030               40 FINISH = $C013
C030   A9FF        50 LDA #$FF
C032   48          60 PHA
C033   28          70 PLP
C034   B8          80 CLV
C035   4C13C0      90 JMP FINISH
C038              100 END
```

```
6000 REM#********************************
6010 REM DATA FOR MACHINE CODE
6020 REM********************************
6030 DATA A9,FF,48,28,B8
```

Commentary
Lines 50–70: These should be familiar to you from the last chapter since they were used there to demonstrate the manner in which a value is loaded into the status register. In this case, as in the previous example, $FF or 255 in decimal, is placed into the register, thus setting each of the bits.

Line 80: The single CLV instruction is sufficient to clear the overflow flag.

The sign flag

Function: To reflect the state of the leftmost binary digit of the value created by the last operation of the chip. It is affected by the same list of operations which was given for the zero flag above. The question which the sign flag answers is: 'Did the last operation create a value which could be regarded as negative, ie had bit eight of the value set?'

The sign flag is another aid in the use of signed arithmetic, though one with much wider applications in general programming techniques. Its purpose is simply to reflect what has happened to the leftmost binary digit of a value which has just been created. If the number created is negative, then the sign flag will be set to one, just as the leftmost bit of the number is set to one.

Compared with the overflow flag, however, the sign flag has a wide

range of uses. For instance, one simple method of counting the number of repetitions of an action is to set up the X register with one less than the value of the number of repetitions, then to decrement (subtract one) from the total each time the desired action is carried out. When the X register goes below zero to 255, the sign flag will be set and can be used to determine the flow of the program.

Instructions which refer to the sign flag
BPL (Branch on PLus) — program execution is sent to a specified point if the sign flag is set to zero when this instruction is encountered.

BMI (Branch on MInus) — program execution is sent to a specified point if the sign flag is set to one when this instruction is encountered.

Demonstrations of the use of these instructions will be given in the chapter on program control.

Demonstration of setting the sign flag

```
ADD.   DATA      SOURCE CODE
00                 10 PRT
00                 20 ORG $C030
C030               30 SYM
C030               40 FINISH = $C013
C030   A900        50 LDA #00
C032   48          60 PHA
C033   28          70 PLP
C034   A288        80 LDX #$88
C036   4C13C0      90 JMP FINISH
C039              100 END
```

```
6000 REM*********************************
6010 REM DATA FOR MACHINE CODE
6020 REM*********************************
6030 DATA A9,00,48,28,A2,88
```

Commentary
Lines 50–70: Once again a routine which should be familiar from the last chapter, its effect being to clear all the bits in the status register.

Line 80: The accumulator is loaded with the value $88, or 136 in decimal. In binary form this number is 10001000 and, since the leftmost bit, bit seven is set, the overflow flag must respond to this value being created in the accumulator.

Table of SET and CLEAR instructions:

Clear Carry	: CLC	: 18
Set Carry	: SEC	: 38
Clear Decimal	: CLD	: D8
Set Decimal	: SED	: F8
Clear Interrupt	: CLI	: 58
Set Interrupt	: SEI	: 78
Clear Overflow	: CLV	: B8

N.B. There is no 'set overflow' instruction.

Conclusion

Do not be concerned if at this point you have found this chapter uninteresting. Any introduction to machine code has to introduce some information before it can really be used. In later chapters you will find a host of commands and techniques which depend entirely on the flags which have just been described, and you will find yourself coming back to remind yourself what the flags record.

CHAPTER 9
Simple Arithmetic

Arithmetic must always be a large part of any book on machine code, not only because of its importance but also because the relatively cumbersome methods which machine code imposes require substantial explanation. In this chapter we shall make a start by taking a look at some simple arithmetic like the addition or subtraction of small numbers and the commands to accomplish them, leaving it to subsequent chapters to take us into more complex areas.

Commands covered in this chapter include, INC, DEC, INX, DEX, INY, DEY, ADC, SBC.

INCrement and DECrement

The simplest form of arithmetic that the 6510 chip is capable of performing is simply to add or subtract one form of a value in a register. This may not seem, at first sight, such a useful ability but it is in fact crucial to many techniques in machine code programming, such as the creation of loops like the BASIC FOR . . . NEXT loop to repeat a task a number of times. So important is the ability to add or subtract one, and so often is it used, that the 6510 chip has special commands built into it to achieve just those simple tasks, thus relieving the machine code programmer of the necessity to specify the number to be added or subtracted.

There are three forms of the command to add one or subtract one. The commands to add one to a value are known as *increment* instructions and they are:

INC to add one to the value in specified memory address
INX to add one to the value in X register
INY to add one to the value in Y register

The opposite form of these are the *decrement* instructions, which are:

DEC to subtract one from the value in specified memory address
DEX to subtract one from the value in X register
DEY to subtract one from the value in Y register

When using the instructions which refer to the X and Y registers, there is nothing to add, no address or value attached, they are what is known as single byte instructions, having all the information the CPU needs (ie the name of the register) built into them. Note that, annoyingly, there is no single instruction to increment or decrement a value in the accumulator. This can only be achieved by the use of the add and subtract instructions which will be discussed later. When incrementing or decrementing the value contained in a specified memory location (INC and DEC), an address will have be specified and several addressing modes are available for this — see the section later in this chapter.

Flags and the INC or DEC instructions

When using the increment or decrement instructions, it has to be remembered that they are distinct from the more complex forms of addition and subtraction. We shall see in a moment that the addition and subtraction instructions have built into them the ability to set and clear the carry flag. Increment and decrement have no effect on the carry flag though they will affect the sign and zero flags according to the rules described in the chapter on flags.

For this reason, the zero and sign flags are frequently used to detect the results of the use of the increment and decrement instructions. If an increment instruction is applied to a register containing the value 255, the register will move around to zero. This change will not, as mentioned, be detected by the carry flag, which will remain in whatever state it was before the increment was made, but it will be detected by the zero flag, since a zero has been created in the register which was incremented.

When a decrement is made to a register which currently stands at zero, the value in the register will become 255. The change in the opposite direction can be detected by the use of the sign flag, which reflects the state of the eighth bit, this having changed from zero when the whole byte was zero to one when the byte became 255.

Addressing modes for use with INC and DEC

While INX, INY, DEX and DEY need no address attached to them since they name the register they apply to, you cannot increment or decrement a value contained in a particular memory address without first of all specifying which memory address you intend to work on. To achieve this you have to attach to the INC or DEC instruction an address, and this can take several forms. We cannot, for reasons of space, discuss in detail the ideas behind the addressing modes since they have already been spelled out in the chapter on addressing. If you are not clear as to what is meant by the different modes of addressing, please go back and have another look at that chapter first.

In general the form of INC and DEC is the instruction followed by an address containing a value on which the increment or decrement is to be performed.

The addressing modes available for use with INC are:

Zero Page	: INC $FF	: E6	: E6 FF
Absolute	: INC $02FF	: EE	: EE FF 02
Zero Page indexed	: INC $FF,X	: F6	: F6 FF
Absolute indexed	: INC $02FF,X	: FE	: FE FF 02

Example of the use of INC

```
ADD.   DATA      SOURCE CODE
00                 10 PRT
00                 20 ORG $C030
C030               30 SYM
C030               40 FINISH = $C013
C030   E6FB        50 INC $FB
C032   4C13C0      60 JMP FINISH
C035               70 END
```

```
6000 REM*********************************
6010 REM DATA FOR MACHINE CODE
6020 REM*********************************
6030 DATA E6,FB
```

Commentary
Line 50: A single line is all that is necessary to show the effect of INC. Use the program options to set the value of memory location $FB to any desired value and then let the machine code run. Experimenting with different values in $FB will give you an idea of the effect of INC on the flags.

The addressing modes available for use with DEC are:

Zero Page	: DEC $FF	: C6	: C6 FF
Absolute	: DEC $02FF	: CE	: CE FF 02
Zero Page indexed	: DEC $FF,X	: D6	: D6 FF
Absolute indexed	: DEC $02FF,X	: DE	: DE FF 02

Example of the use of DEC

```
ADD.   DATA      SOURCE CODE
00                 10 PRT
00                 20 ORG $C030
C030               30 SYM
C030               40 FINISH = $C013
C030   C6FB        50 DEC $FB
C032   4C13C0      60 JMP FINISH
C035               70 END
```

```
6000 REM#********************************
6010 REM DATA FOR MACHINE CODE
6020 REM********************************
6030 DATA C6,FB
```

Commentary
Line 50: Once again, only a single line is necessary, and experimentation with the value in $FB will demonstrate the effect of DEC on flags.

Table of remaining increment and decrement instructions:

Increment X register	: INX	: E8
Decrement X register	: DEX	: CA
Increment Y register	: INY	: C8
Decrement Y register	: DEY	: 88

Addition with ADC

We turn now from the simplest form of addition, using the increment instructions, to the more general method which makes use of the ADC instruction, or 'ADd with Carry'. In using ADC we shall, in some ways, be more limited than in the use of the increment instructions since true arithmetic *can only be carried out on the value contained in the accumulator*. For this reason, ADC is seldom used on its own, but in conjunction with instructions which transfer values from memory to the accumulator so that they can be worked upon, then transfer the values back again to memory. We have already looked at the ways in which values can be transferred and so the examples given will include transfers to and from the accumulator.

The ADC instruction can be used with a number of addressing modes but its general effect is always to specify either a value or an address at

which a value can be found, and then to take that value and add it to the present contents of the accumulator. In addition the 'C' terminator on the instruction is a reminder that the carry flag forms an integral part of the addition. If the carry flag is set when the addition commences then an extra 1 will be added to the sum in the accumulator. Later on we shall see that this is essential to arithmetic involving numbers longer than a single byte. When it comes to adding together single byte numbers, however, it means that if the carry flag is set for some irrelevant reason before the addition is undertaken, it could result in an erroneous result. For this reason you will always find that for the purposes of normal single byte addition, the CLC instruction is always used to ensure that the carry flag is clear before executing the addition.

Flags which will be affected by the use of ADC are carry, zero, overflow and sign.

The addressing modes available for use with ADC are:

Zero page	: ADC $FF	: 65	: 65 FF
Absolute	: ADC $02FF	: 6D	: 6D FF 02
Immediate	: ADC #$FF	: 69	: 69 FF
Pre-indexed indirect	: ADC ($FF,X)	: 61	: 61 FF
Post indexed indirect	: ADC ($FF),Y	: 71	: 71 FF
Zero page indexed, X	: ADC $FF,X	: 75	: 75 FF
Absolute indexed, X	: ADC $02FF,X	: 7D	: 7D FF 02
Absolute indexed, Y	: ADC $02FF,Y	: 79	: 79 FF 02

Example of the use of ADC

```
ADD.   DATA      SOURCE CODE
00                 10 PRT
00                 20 ORG $C030
C030               30 SYM
C030               40 FINISH = $C013
C030   18          50 CLC
C031   A5FB        60 LDA $FB
C033   65FD        70 ADC $FD
C035   4C13C0      80 JMP FINISH
C038               90 END

6000 REM**********************************
6010 REM DATA FOR MACHINE CODE
6020 REM**********************************
6030 DATA 18,A5,FB,65,FD
```

Commentary
Line 50: The carry flag is first cleared, for the reasons explained above.
Lines 60–70: The contents of memory location $FB are loaded into the accumulator and then the contents of memory location $FD added. The result will depend upon the values you choose to place into the two locations.

Subtraction with SBC

The opposite of ADC is SBC (SuBtract with Carry) which subtracts a specified value from the contents of the directory. As with ADC, SBC is normally used in conjunction with instructions to transfer a value into the accumulator to be worked upon and then to transfer it out again.

The addressing modes with which SBC will function are the same as those for ADC and are listed below. The carry element of the instruction name refers to the fact that if the carry flag is *clear* at the commencement of the instruction, an extra 1 will be subtracted from the true result — note this reversal of the pattern shown in ADC very carefully. Once again, for this reason the carry flag is adjusted before single byte subtraction is carried out, though this time SEC is used to *set* the flag.

The flags affected by SBC are the same as those for ADC.
The addressing modes available for use with SBC are:

Zero page	: SBC $FF	: E5	: E5 FF
Absolute	: SBC $02FF	: ED	: ED FF 02
Immediate	: SBC #$FF	: E9	: E9 FF
Pre-indexed indirect	: SBC ($FF,X)	: E1	: E1 FF
Post indexed indirect	: SBC ($FF),Y	: F1	: F1 FF
Zero page indexed, X	: SBC $FF,X	: F5	: F5 FF
Absolute indexed, X	: SBC $02FF,X	: FD	: FD FF 02
Absolute indexed, Y	: SBC $02FF,Y	: F9	: F9 FF 02

Example of the use of SBC

```
ADD.   DATA      SOURCE CODE
00                 10 PRT
00                 20 ORG $C030
C030               30 SYM
C030               40 FINISH = $C013
C030   38          50 SEC
C031   A5FB        60 LDA $FB
C033   E5FD        70 SBC $FD
C035   4C13C0      80 JMP FINISH
C038               90 END
```

```
6000 REM#*******************************
6010 REM DATA FOR MACHINE CODE
6020 REM*******************************
6030 DATA 38,A5,FB,E5,FD
```

Commentary
Line 50: SEC is used to set the carry flag to ensure an accurate result.

Lines 60–70: The contents of memory location $FB are loaded into the accumulator and then the contents of memory location $FD subtracted. Experiment with different values in the two locations to see what the effect is on the result and on the flags.

Conclusion

At this point you may well have decided that the book has become distinctly less friendly. Gone are the pages of simple explanations in English of every in and out the chip's behaviour. Suddenly you are being asked to understand the concept of adding and subtracting in strange modes like pre-indexed indirect.

In fact, this chapter in the book is the one which will determine whether you are really going to understand machine code. It is one thing to explain all the concepts, to go through lists of addressing modes, to discuss the idea of addition and how the CPU's method of addition can be simulated on pieces of paper or to list in detail how each flag is to be interpreted. All that has been done in earlier parts of the book.

At some stage, however, you have to make the leap from knowing that, for instance:

a) indirect addressing means getting an address from a place in memory and then getting a value from the memory byte at that address,

b) LDA means place a value into the accumulator,

c) ADC means adding a value to what is already in the accumulator, and

d) the carry flag both affects and is affected by the size of the result of the addition,

to seeing the assembly language instructions:

```
LDA #$F0
ADC $01F0
```

and knowing that the number $F0 is being loaded into the accumulator, and then having added to it the contents of a memory byte whose address is pointed to by the contents of the two bytes starting at memory location $01F0.

In addition, this chapter gives you your first taste of something that so often confuses beginners in machine code, the fact that while an assembly language instruction makes clear by its format the instruction which is being used and the addressing mode (each as separate items), when working directly in machine code, a single code represents both instruction and addressing mode.

Only you can decide whether you are going to make the effort to familiarise yourself with the addressing modes and the relatively few instructions in 6510 machine code to the extent that your knowledge of the two begins to blend together, but unless you do you will never be able to tackle machine code with confidence. No book can do the work for you because to explain the addressing modes in full alongside each instruction which used them would require literally hundreds of extra pages and result in something closer to the Encyclopaedia Britannica than a beginner's handbook.

So before you go on, do ask yourself whether you feel confident that you understand the different addressing modes and the way they relate to an individual instruction like ADC or SBC. If not, go back to some of the earlier chapters or face being lost in the wilderness for a long time!

CHAPTER 10
Two-Byte Arithmetic

Having learned how to add together numbers which can be held in a single byte, or to subtract one such number from another, we turn to the slightly more complex topic of numbers which require two or more bytes. Notice that we say slightly more complex, because whether you realise it or not, you already know all the techniques necessary to add or subtract two byte numbers.

Adding two-byte numbers

Adding two-byte numbers is just like adding two-digit numbers in decimal. When adding two single digit numbers the only complication is that if there is a carry it has to be written in the leftmost column of the result. With two-digit numbers the carry has to be added into the second column of the sum. To labour the point, here are two examples:

```
  6                16
 +7               +17
 —                —
                   3
 13                1          ←CARRY
 ↑                 2
CARRY              —
                  33
```

The principle is exactly the same when it comes to adding in machine code, and we already know how to add in any carry, since it is done automatically when the ADC command is used.

To add two byte values, the procedure is:

1) Load the accumulator with the low byte of one of the two values.

2) Clear the carry flag, as at the start of any addition.

3) Use ADC to add the low byte of the second value.

4) Store the result somewhere.

5) Load the accumulator with the next byte of one of the two values.

6) Use ADC to add the next byte of the second value.

At the end of this procedure, the result is that you have a low byte stored somewhere, a high byte in the accumulator, and possibly a 1 in the carry flag which represents not 256 but 256↑2, or 65536, or 10000 in hexadecimal.

If you want to go beyond two byte numbers, though it is questionable whether a beginner in machine code would ever really need to, then it is simply a matter of repeating steps 4–6 until you run out of bytes in the two numbers. At the end of the process, if there is a 1 in the carry flag it represents 256 to the power of the number of bytes in each of the values being added.

Example of the addition of two 2-byte numbers

```
ADD.   DATA       SOURCE CODE
00                  10 PRT
00                  20 ORG $C030
C030                30 SYM
C030                40 FINISH = $C013
C030   18           50 CLC
C031   A5FB         60 LDA $FB
C033   65FD         70 ADC $FD
C035   AA           80 TAX
C036   A5FC         90 LDA $FC
C038   65FE        100 ADC $FE
C03A   4C13C0      110 JMP FINISH
C03D               120 END
```

```
6000 REM**********************************
6010 REM DATA FOR MACHINE CODE
6020 REM**********************************
6030 DATA 18,A5,FB,65,FD,AA,A5,FC
6040 DATA 65,FE
```

Commentary
Lines 50–70: The two low bytes of the numbers, which are contained in memory locations $FB and $FD respectively, are added together.

Line 80: Since more addition has to be done, the result is transferred for safe keeping to the X register.

Lines 90–100: The two high bytes of the numbers, which are contained in memory location $FC and $FE respectively, are now added together. Note that the carry flag is not cleared for this second addition since it contains the only evidence of whether there is a carry from the addition of the low bytes and is essential to a correct result. The result itself comprises a high byte which is contained in the accumulator and a low byte in the X register.

The overall result will depend upon the values you choose to place in memory locations $FB–$FE. Remember that the locations will be used by the program as follows:

$FB — low byte of value number 1
$FC — high byte of value number 1
$FD — low byte of value number 2
$FE — high byte of value number 2

If, for instance you were to place 1, 2, 3, and 4 in the locations, you would be adding together (1 + 256*2) and (3 + 256*4).

Subtracting two byte numbers

When it comes to subtraction things are just as simple, provided that you remember that the carry flag works in the *opposite way*, ie that when the carry flag is *clear* a borrow has to be made from the next byte to the left.

The procedure for two byte subtraction is as follows:

1) Set the carry flag, as in the case of any subtraction.

2) Load the accumulator with the low byte of the number to be subtracted from.

3) Use SBC to subtract the low byte of the other value.

4) Store the result somewhere.

5) Load the accumulator with the next byte of the number to be subtracted from.

6) Use SBC to subtract the next byte of the other number.

Note that while it does not matter in the case of ADC which of the two numbers the bytes to be placed in the accumulator are taken from — you can alternate from byte to byte if you really wanted to be perverse — with SBC however it *does* matter. Whatever you place into the accumulator will be what is subtracted *from*, so having started with the low byte of one of the two numbers in the accumulator, any subsequent bytes placed there must be from the same number.

At the end of the process you will have the low byte of the result stored somewhere and the high byte in the accumulator. If the carry flag is clear, this is an indication that there is a borrow, and the true result is 65536 less than the result recorded by the high and low bytes.

Once again, subtraction of numbers with more than two bytes simply involves repeating steps 4–6 until all the bytes are used up. If the carry flag is clear at the end, the apparent result requires the subtraction of 256 to the power of the number of bytes in each of the two values.

Example of two byte subtraction

```
ADD.   DATA      SOURCE CODE
00                 10 PRT
00                 20 ORG $C030
C030               30 SYM
C030               40 FINISH = $C013
C030   38          50 SEC
C031   A5FB        60 LDA $FB
C033   E5FD        70 SBC $FD
C035   AA          80 TAX
C036   A5FC        90 LDA $FC
C038   E5FE       100 SBC $FE
C03A   4C13C0     110 JMP FINISH
C03D              120 END
```

```
6000 REM#********************************
6010 REM DATA FOR MACHINE CODE
6020 REM********************************
6030 DATA 38,A5,FB,E5,FD,AA,A5,FC
6040 DATA E5,FE
```

Commentary

Lines 50–80: The subtraction of the low byte of one number from the low byte of the other, with the result being stored in the X register.

Lines 90–100: The subtraction of the two high bytes. Note once again that no adjustment is made to the carry flag for the high byte since its value indicates whether there is to be a borrow from the previous subtraction. The result comprises a high byte in the accumulator and a low byte in the X register.

CHAPTER 11
Signed Arithmetic

Having tackled the basics of addition and subtraction, we now turn to the rather more tricky topic of working with signed numbers, that is, numbers which can be either positive or negative. We have already looked at the ideas behind signed numbers once or twice in the opening chapters and in discussing the flags, so you will be aware that they are less simple to deal with than the straightforward unsigned numbers that we have been dealing with in the last two chapters.

The reason that things are not as simple is that the 6510 chip is not set up to recognise or deal with signed numbers. Such numbers are a 'convention', a decision by the programmer that certain numbers are going to be dealt with as if they are potentially positive or negative. When the chip comes across a number like 255, or 11111111 in binary, it is treated just as 255, it is only the programmer who is capable of intervening to say 'this number is not actually 255 but −1, since the eighth binary digit will be used to indicate whether the number is positive or negative and we shall work in two's complement notation'. In other words, work with signed numbers involves just that — work.

At this point, if you are interested in the ways in which signed numbers are manipulated, you have to be sure that the last paragraph hasn't thrown you too badly. We have previously discussed the use of the eighth binary digit to replace the plus and minus signs, and also two's complement arithmetic. If you are not clear on either of them, go back and refresh your memory or you will very quickly become bogged down in what is to come.

Selecting a position for the sign bit

The first thing that the programmer has to do when working with signed numbers is to set up the format for the numbers. So far, in discussing signed numbers we have assumed that the sign digit is the eighth digit of the byte. This is merely a convention which the programmer chooses to observe, enforced by the fact that the overflow bit can be used to record any corruption of the eighth digit, as we saw in the flags chapter. The programmer could equally well decide to use the first digit as a sign bit though in fact this does present practical problems.

The point of telling you this is to emphasise that, as the programmer, it is *your* job to set up the signed numbers, not the job of the 6510 chip. This involves you in deciding exactly where the sign bit is going to be placed so that you can then program around it being in that position. Simple enough, you say, I choose to place the sign bit in the eighth position, now let's get on with the difficult part. Well the difficult part comes sooner than you think — try setting up the number minus 255 (an eight digit binary number), when you have taken one digit out of an eight digit byte to represent the sign of the number.

The answer, of course, is that you can't do it and from the answer is to be drawn an important lesson. Before embarking on the use of signed numbers and writing all the routines based on placing the sign digit in a certain place, you need to think hard about the range of numbers you are going to need. In previous chapters we have noted that a single byte signed number can only be in the range −128 to +127. This range is quite limited for most purposes in serious machine code programs, so single byte signed numbers are seldom used. Two byte signed numbers allow a range of −32768 to +32767, which is probably adequate for most of the things you will want to try. When you move on to more complex applications, like multiplying two multi-byte numbers together (a horrendous task which we shall not describe), you may find yourself working with signed numbers which are four or more bytes long.

For the moment, however, we shall limit ourselves to two-byte numbers, placing the sign digit in the leftmost position available, the eighth digit of the high byte of the number. Any values we create will look as follows:

```
[<SIGN DIGIT><7 DIGITS>]   [8 DIGITS]
      HIGH BYTE              LOW BYTE
```

Once decided upon, this format must be looked upon as fixed, otherwise you will have to write extra sets of machine code routines to handle signed numbers with other formats. This means that numbers outside the range −32768 to +32767 will have to be avoided and that numbers in the range −128 to +127 will *still have to be given two bytes*, even though strictly they could be fitted into a single byte. In addition, any routines which you write to deal with the signed numbers will have to be split into two parts, one to deal with the low byte, which will not have a sign digit, and another section to deal with the high byte, which will.

So far, so good, but you may have noticed that we have so far completely ducked the question as to how you actually go about creating a signed number, whether the sign bit is in the eighth or the sixteenth position.

Creating a signed number

The complexity of creating a signed number will vary according to whether the number is positive or negative.

Positive numbers: we have already determined that we are going to work only with numbers in the range −32768 to +32767, numbers which can be fitted into two bytes with room left over for the highest digit to be reserved for the sign of the number. Since the representation of a positive number is a zero in the highest binary digit and 32767 (the highest positive number) will not affect the highest digit, precisely nothing has to be done in order to set up a positive number. All this required is that the program begins to treat the number as if it were signed.

Negative numbers: a simple rule suffices, though why it suffices is another matter. First take the number which you wish to represent as negative. In this case we shall take the example of 4232, which in unsigned binary would be:

00010000 10001000
The first action is to invert all the bits, resulting in:
11101111 01110111
and then add one:
11101111 01111000

There is a second method, which is slightly quicker, though it is slightly more prone to error. The first step is to take the binary representation of the number, in our case:

00010000 10001000

Start counting from the right of the number until a digit containing something other than '0' is found. In our case, the fourth digit from the right is a 1. Leave that digit alone, but from the next digit onwards, move to the right, inverting every digit up to and including the left-most. The result of this, in our case is:

11101111 01111000

exactly the same as by the previous method.

Having obtained your signed number in binary, it is up to you to then translate it into decimal or hexadecimal according to the method of placing the machine code program into memory that you are using. The hexadecimal route will be the easier since all you need to do is to take

each group of 4 binary digits, starting at the left, and to translate them into one hexadecimal digit in the range 0–F (0–15 decimal).

After reading all of this you will realise the truth of what we said earlier about the advantages of an assembler which can handle signed numbers directly.

Adding together signed numbers

Adding together two-byte signed numbers is just the same as adding together unsigned numbers, up to a point. That point is what happens if you generate a number which cannot be held in the 15 binary digits available once the sign digit has been reserved. When this happens the CPU, which does not know that you are doing clever things with signed numbers, carries one and places it in the 16th position, the one at the extreme left of the number. Whatever your sign digit was going to be, it is now the opposite. Problem number one.

Problem number two is just as simple, try adding the following two signed numbers, both representing −127:

10000001
10000001

The straightforward result should be 00000010, with the carry flag set to indicate that an extra 256 needs to be taken into account. But we are not talking about unsigned numbers, what has really happened is that the two sign digits have been added together and cancelled each other out. The resulting number, in signed arithmetic, is +2, which is not correct.

Both these problems are the result of things being carried into or out of the sign digit, and it is here that the overflow flag, which we have already examined comes in. By examining the overflow flag when two signed numbers have been added together, we can immediately see whether or not the sign digit is correct or not. The rule here is simple:

Start with zero.
If something is carried into the sign bit, add 1.
If something is carried out of the sign bit, add 1.
If the result is zero or two, the sign bit is correct; if the result is one, then the sign bit is incorrect, and the overflow flag will be set to one.

In addition, the way two's complement numbers are constructed means that the overflow flag is an infallible sign of whether the number created is too large for the seven or 15 binary digits allocated to it — if the overflow flag is set, the number is out of range and there has been a carry out of the number. At first sight it seems that this might not be true. If we add two

positive signed numbers, 127 and 127, the result is going to be too large for the range −128 to +127:

```
0111111
0111111
-------
1111110
```

Here the overflow flag is set to one, indicating that the sign is wrong and that the number is out of range. If we take the same binary digits and change the zero at the front to a one, we seem to get an anomaly:

```
11111111
11111111
--------
11111110 (and 1 in the carry flag)
```

Here there has been a carry into the sign digit but also a carry into the carry flag — according to the rules above, the overflow flag will be clear and the number is not out of range. In fact the overflow flag is quite correct, 11111111 in two's complement means −1 in decimal — for such a small number it is not really surprising that there is no overflow. To get to the kind of figures which would result in an overflow, say −65 plus −65, the two's complement notation would not have a 1 in the position of the seventh digit to overflow into the eighth:

```
10111111
10111111
--------
01111110
```

Here, the overflow into the carry bit is not balanced by a carry into the sign bit, so the overflow flag will be set, indicating correctly that the result is not − 2 but −2 plus −128 or −130.

It is not necessary that you remember all this to begin with, only that you realise that the overflow flag is two things:

1) An indication that the sign digit of a number is incorrect.

2) An indication that the result is too large for the number of binary digits you have chosen to work with.

What to do? Well in the second case, the answer is normally nothing. We have already explained that when using signed numbers, the format of the numbers is fixed, they are two-byte or one-byte (or however many

you choose) because the program *must* know where to find the sign digit. A number outside the range your format can deal with cannot be handled unless you want to change your format in the first place.

Even if you can tolerate losing the most significant binary digit of a number, something *must* be done about the change of sign since we cannot have positive numbers masquerading as negative, and vice-versa. The problem is solved by inverting the incorrect digit using the EOR instruction, which will be explained in Chapter 12.

The procedure for adding signed, two-byte numbers is, therefore, as follows:

1) Clear the carry flag, as in all addition.

2) Load the low byte of one of the values into the accumulator.

3) Use ADC to add the low byte of the other value.

4) Store the result somewhere.

5) Place the next byte of one of the values into the accumulator.

6) Use ADC to add the next byte of the other number.

7) Test the overflow flag.

8) If the overflow flag is set to one, invert the sign digit of the byte in the accumulator using EOR (see chapter 12).

The result will be a value whose low byte you have stored somewhere in memory, and whose high byte is in the accumulator, with a correct sign digit. If the overflow flag is set, the result is inaccurate by 32768. Whether this carry is positive or negative will depend on the correct sign of the result.

Example of the addition of two single byte signed numbers

```
ADD.   DATA      SOURCE CODE
00                 10 PRT
00                 20 ORG $C030
C030               30 SYM
C030               40 FINISH = $C013
C030   18          50 CLC
C031   A5FB        60 LDA $FB
C033   65FD        70 ADC $FD
```

```
C035  5002        80 BVC L000
C037  4980        90 EOR #$80
C039             100 L000
C039  4C13C0     110 JMP FINISH
C03C             120 END
```

```
6000 REM*********************************
6010 REM DATA FOR MACHINE CODE
6020 REM*********************************
6030 DATA 18,A5,FB,65,FD,50,02,49
6040 DATA 80
```

Commentary
Lines 50–70: The two numbers are added.

Lines 80–90: These lines employ two instructions which will not be properly explained until later chapters. The BVC command is rather like a GOTO in BASIC, but in this case it is only carried out if the overflow flag is clear. The EOR instruction is used to invert the sign bit of the result. Putting the two together, the lines mean that if the overflow bit is set, the EOR instruction reverses the sign bit to correct it. If the overflow bit is clear then the BVC will jump past the EOR instruction since there is nothing to correct.

Lines 100–110: The L000 in line 100 is not a machine code instruction but what is known as a *label*. It is there only to tell the assembler where the BVC in line 80 is meant to jump. Notice that according to the assembly language listing the L000 and the JMP instruction in the next line both have the same address. In jumping to the instruction with the address labelled L000, the program actually jumps to line 110.

Example of the addition of two 2-byte signed numbers

```
ADD.  DATA      SOURCE CODE
00                10 PRT
00                20 ORG $C030
C030              30 SYM
C030              40 FINISH = $C013
C030  18          50 CLC
C031  A5FB        60 LDA $FB
C033  65FD        70 ADC $FD
```

```
C035  AA          80 TAX
C036  A5FC        90 LDA $FC
C038  65FE       100 ADC $FE
C03A  5002       110 BVC L000
C03C  4980       120 EOR #$80
C03E             130 L000
C03E  4C13C0     140 JMP FINISH
C041             150 END
```

```
6000 REM*********************************
6010 REM DATA FOR MACHINE CODE
6020 REM*********************************
6030 DATA 18,A5,FB,65,FD,AA,A5,FC
6040 DATA 65,FE,50,02,49,80
```

Commentary
Lines 50–80: Nothing new here, the low bytes of the two numbers are added together and the result transferred to the X register for safe keeping.

Lines 90–100: Again, nothing exceptional — the two high bytes are added together.

Lines 110–140: The possible correction of the sign bit is applied to the high byte of the result.

Subtracting signed numbers

As with ADC and SBC on unsigned numbers, the only real difference in working with signed numbers is that there is a limitation on which of the values you are using has its bytes placed in the accumulator. The value which is placed into the accumulator will be the one which it is subtracted from.

The procedure for subtracting one signed number from another is:

1) Set the carry flag, as in all subtraction.

2) Load the low byte of the value to be subtracted from into the accumulator.

3) Use SBC to subtract the low byte of the other value.

4) Store the result somewhere.

5) Place the next byte of the value to be subtracted from into the accumulator.

6) Use SBC to subtract the next byte of the other number.

7) Test the overflow flag.

8) If the overflow flag is set to one, invert the sign digit of the byte in the accumulator using EOR (see Chapter 12).

As with ADC, the result will be a value whose low byte you have stored somewhere in memory, and whose high byte is in the accumulator, with a correct sign digit. If the overflow flag was set, the result is inaccurate by 32768. Whether this carry is positive or negative will depend on the correct sign of the result.

Example of subtraction of two single byte signed numbers

```
ADD.   DATA       SOURCE CODE
00                  10 PRT
00                  20 ORG $C030
C030                30 SYM
C030                40 FINISH = $C013
C030   38           50 SEC
C031   A5FB         60 LDA $FB
C033   E5FD         70 SBC $FD
C035   5002         80 BVC L000
C037   4980         90 EOR #$80
C039               100 L000
C039   4C13C0      110 JMP FINISH
C03C               120 END
```

```
6000 REM#################################
6010 REM DATA FOR MACHINE CODE
6020 REM#################################
6030 DATA 38,A5,FB,E5,FD,50,02,49
6040 DATA 80
```

Commentary
Lines 50–70: The lines we would expect to find in a normal subtraction.

Lines 80–100: As with the addition earlier, the overflow command is used to determine whether a correction has to be made to the sign bit of the result.

Example of subtraction of two 2-byte signed numbers

```
ADD.   DATA      SOURCE CODE
00                 10 PRT
00                 20 ORG $C030
C030               30 SYM
C030               40 FINISH = $C013
C030   38          50 SEC
C031   A5FB        60 LDA $FB
C033   E5FD        70 SBC $FD
C035   AA          80 TAX
C036   A5FC        90 LDA $FC
C038   E5FE       100 SBC $FE
C03A   5002       110 BVC L000
C03C   4980       120 EOR #$80
C03E              130 L000
C03E   4C13C0     140 JMP FINISH
C041              150 END
```

```
6000 REM#********************************
6010 REM DATA FOR MACHINE CODE
6020 REM********************************
6030 DATA 38,A5,FB,E5,FD,AA,A5,FC
6040 DATA E5,FE,50,02,49,80
```

Commentary
The routine is similar in structure to the two byte addition shown earlier in the chapter.

Binary coded decimal mode

One final form of arithmetic needs to be examined before we move on — binary coded decimal, or BCD, the form which the chip uses when it is switched to decimal mode. BCD is the reason that the status register contains a flag which we have not yet really explained — the decimal flag. The object of BCD is to allow the chip to work directly with decimal

numbers. This is by no means always necessary, since if decimal numbers have to be used calculations can be done in binary and a translation made into decimal when a result has been obtained. In some cases, however, perhaps when a program invites inputs in decimal or outputs decimal numbers to the screen, time and program space can be saved by working directly in decimal. Luckily the 6510 allows us to do just that. It is purely a matter of convenience that BCD is provided — the decimal numbering system is of no relevance or interest to the chip itself and is never used for its own internal purposes. Since human beings perversely insist on working in decimal, however, the designers of the 6510 have provided a means by which a few functions can be performed in decimal, namely adding and subtracting.

Having said that it is possible to work in decimal, you are probably wondering just how, since the whole structure of the chip, with its eight bit memory locations, seems to militate against it. The answer is that BCD is an entirely artificial way of using the chip, and wasteful of memory and processing power. If it shortens some programs, however, it can be worthwhile using it.

The essence of the BCD mode on the chip is that when the decimal flag is set, any value in the accumulator is regarded as being two four bit numbers. This is not just a convention, a way of looking at an eight bit number, the division is real because each of the two halves is treated as a single decimal digit. Instead of working on the basis that four bits can hold a number in the range 0–15, the chip cuts down the range to 0–9. If a number greater than nine is created in the lower four bits of the accumulator, for instance, a carry will be made to the upper four bits. If a number greater than 9 is created in the upper four bits, the carry flag is set. In other words, if you add a number to the contents of the accumulator while in decimal mode, any result and carries would be exactly as if you had added two decimal numbers.

Decimal mode is by no means foolproof. If, for instance, you were to place the value $FF into the accumulator (decimal 255), so that each half of the byte would be 15 when viewed separately, and then set the decimal flag and proceeded with addition or subtraction, the result would be nonsensical. All that decimal mode guarantees is that if you feed the chip values which do not exceed 9 in either of the two halves of the byte, they will be treated as decimal values for the purposes of addition and subtraction.

Decimal mode has advantages for some applications but at the same time its limitations need to be remembered. In particular, it is vital to remember that decimal mode cannot be used throughout the average program. If the chip were to treat numbers as if they were decimal values throughout a program then the program and probably the system would crash. The moment that you have ceased to need to work in decimal, the

decimal flag must be cleared in order to allow the chip to function normally.

Given below are two examples of the use of BCD mode in relation to single byte numbers. The principles of applying BCD to multiple byte numbers are very similar to those which we have already examined and should cause no problems if you want to experiment for yourself.

Example of the addition of single byte numbers in BCD

```
ADD.   DATA      SOURCE CODE
00                 10 PRT
00                 20 ORG $C030
C030               30 SYM
C030               40 FINISH = $C013
C030   18          50 CLC
C031   F8          60 SED
C032   A5FB        70 LDA $FB
C034   65FD        80 ADC $FD
C036   4C13C0      90 JMP FINISH
C039              100 END
```

```
6000 REM#################################
6010 REM DATA FOR MACHINE CODE
6020 REM#################################
6030 DATA 18,F8,A5,FB,65,FD
```

Commentary

Line 60: The only unusual line — once decimal mode has been set you are free to add as usual, with the chip doing all the necessary translations.

To test out decimal mode, try the following: place 9 in memory location $FB and 1 in memory location $FD. The result in the accumulator will be 16. This is not nonsense, what has happened is as follows.

The 9 in $FB was stored as (0000) (1001) or (0) (9)
The 1 in $FD was stored as (0000) (0001) or (0) (1)
The result, in BCD mode is (0001) (0000) or (1) (0)

The Code Runner program, which does not operate in BCD mode, reports this as a normal binary number with the decimal value of 16.

Example of subtraction of single byte numbers in BCD

```
ADD.   DATA      SOURCE CODE
00                 10 PRT
00                 20 ORG $C030
C030               30 SYM
C030               40 FINISH = $C013
C030   38          50 SEC
C031   F8          60 SED
C032   A5FB        70 LDA $FB
C034   E5FD        80 SBC $FD
C036   4C13C0      90 JMP FINISH
C039              100 END
```

```
6000 REM*********************************
6010 REM DATA FOR MACHINE CODE
6020 REM*********************************
6030 DATA 38,F8,A5,FB,E5,FD
```

Commentary
Once again a straightforward subtraction, except that the decimal flag is set before it is carried out.

CHAPTER 12
Logical Operators and Bit Instructions

So far we have pretty much taken it for granted that either a byte of memory is a unit to be loaded, stored, or transferred, or that if a byte is to be changed it is left to the CPU by means of instructions like ADC. In this chapter we shall move one stage down the ladder and look at how it is possible to manipulate the individual bits which go to make up a byte, and some of the reasons that this might need to be done.

In the course of the chapter we shall discuss logical operations, bit instructions and shift/rotate instructions.

Logical operations

At its simplest, a logical operation is a way of comparing two bytes as if they were merely patterns of bits, and of producing a third byte on the basis of this comparison. There are three types of comparison:

1) Are the bits in the same position in both bytes, both set? If so, the equivalent bit in the resulting byte will be set. This form of logical operation is covered by the AND instruction.

2) Is either one or both of the bits set? If so, the equivalent bit in the resulting byte will also be set. This form of logical operation is covered by the ORA instruction.

3) Is one or other of the bits but not both, set? If so, the equivalent bit in the resulting byte will also be set. This form of logical operation is covered by the EOR instructions.

In the case of all three of these instructions, the two bytes to be compared will be the contents of the accumulator and a byte of memory specified in the machine code program. The byte resulting from the comparison will be placed into the accumulator.

The AND instruction

Function: to create a byte which has bits set in each position where the

two original bytes both had a set bit.

Like all the logical operations, the working of AND is best understood by simply observing it in operation. Given below are a few examples of pairs of bytes and the third byte produced when AND is applied to them:

```
11110000 AND 00001111  =  00000000
10101010 AND 01010101  =  00000000
11110000 AND 11111111  =  11110000
10101010 AND 00001111  =  00001010
```

AND is used primarily for clearing bits, setting them to zero. For instance, let us assume that for some reason a machine code program wished to ensure that none of a series of values had any of the bits 4–7 set. This could be easily achieved by ANDing each of the values with 15, or 00001111 in binary. Since none of the bits 4–7 could be set in *both* of the bytes, they would all be reset. In the other half of the byte, the fact that there are set bits in every position in one byte, means that nothing will change — where there is a zero in the other byte, the two bits will not match and where there is a one AND will ensure that it remains.

A second use of AND is in the testing of individual bits in a specified byte. You already know, for instance that the result of ANDing a byte with a mask is placed into the accumulator. If the mask in the accumulator consists of a value where only one of the eight bits is set, then the result will be the same value or zero, according to whether the equivalent bit in the specified memory byte is also set. You also know that the zero flag indicates whether the last operation on the accumulator resulted in a zero value being created. Put these two together and you can see that using a mask like 00001000 with AND, and then sampling the zero flag, will show whether the fourth bit is set in the specified memory location.

The procedure for the use of AND is therefore very simple:

1) Load the controlling byte, the one which we are using to dictate the eventual shape of the byte, into the accumulator. This value is usually known as the *mask* (in the sense of a stencil used in drawing), since it is, in a way, placed over the other byte so that only certain bits can be seen — others are blanked out and reset. The reason that the mask is normally placed into the accumulator is that it will usually be specified in the program and can be loaded into the accumulator in immediate mode.

2) Apply the mask to a specified memory location using AND.

3) Do something with the result, which is found in the accumulator. It may be, for instance, that you wish to replace the original value in memory with the one created by AND, in which case you must store the

accumulator. You may on the other hand merely wish to take a decision on the basis of the result, without making any change to the original value.

Addressing modes available with AND:

Zero page	: AND $FF	: 25	: 25 FF
Immediate	: AND #$FF	: 29	: 29 FF
Absolute	: AND $02FF	: 2D	: 2D FF 02
Pre-indexed, X	: AND ($FF,X)	: 21	: 21 FF
Post-indexed, Y	: AND ($FF),Y	: 31	: 31 FF
Zero page, X	: AND $FF,X	: 35	: 35 FF
Absolute, X	: AND $02FF,X	: 3D	: 3D FF 02
Absolute, Y	: AND $02FF,Y	: 39	: 39 FF 02

Example of the use of AND

```
ADD.   DATA      SOURCE CODE
00                 10 PRT
00                 20 ORG $C030
C030               30 SYM
C030               40 FINISH = $C013
C030   A5FB        50 LDA $FB
C032   25FC        60 AND $FC
C034   4C13C0      70 JMP FINISH
C037               80 END
```

```
6000 REM#********************************
6010 REM DATA FOR MACHINE CODE
6020 REM********************************
6030 DATA A5,FB,25,FC
```

Commentary
The two numbers to be ANDed must be placed into memory locations $FB and $FC. The contents of $FB are copied into the accumulator and then ANDed with the contents of $FC.

The ORA instruction
Function: to create a byte which has bits set in each position where either or both of the original bytes had a set bit.

Some examples of the effect of OR:

```
11110000 OR 00001111  =  11111111
10101010 OR 01010101  =  11111111
11110000 OR 11111111  =  11111111
10101010 OR 00001111  =  10101111
```

OR is primarily used to set bits. When a byte is ORed with a mask, any bit set in the mask will automatically be set in the resulting byte. Digits in the original byte which are not paired with a set bit in the mask will be left unchanged whether they are set or reset.

The procedure for the use of ORA is as follows:

1) Load the value to be used as a mask into the accumulator.

2) Apply the mask to a specified byte in memory using OR.

3) The result will now be found in the accumulator and can either be used to replace the original byte in memory or as the basis for some other action by the program.

Addressing modes available with ORA:

Zero page	: ORA $FF	: 05	: 05 FF
Immediate	: ORA #$FF	: 09	: 09 FF
Absolute	: ORA $02FF	: 0D	: 0D FF 02
Pre-indexed, X	: ORA ($FF,X)	: 01	: 01 FF
Post-indexed, Y	: ORA ($FF), Y	: 11	: 11 FF
Zero page, X	: ORA $FF,X	: 15	: 15 FF
Absolute, X	: ORA $02FF,X	: 1D	: 1D FF 02
Absolute, Y	: ORA $02FF,Y	: 19	: 19 FF 02

Example of the use of ORA

```
ADD.   DATA      SOURCE CODE
00                 10 PRT
00                 20 ORG $C030
C030               30 SYM
C030               40 FINISH = $C013
C030   A5FB        50 LDA $FB
C032   05FC        60 ORA $FC
C034   4C13C0      70 JMP FINISH
C037               80 END
```

```
6000 REM#*******************************
6010 REM DATA FOR MACHINE CODE
6020 REM*******************************
6030 DATA A5,FB,05,FC
```

Commentary
The example is exactly the same in structure as that given for AND, except that the contents of the two memory locations are ORed together.

The EOR instruction
Function: to create a byte which has bits set in each position where either but not both of the original bytes had a set bit.

It is important to remember the distinction between ORA and EOR (which stands for Exclusive OR). Whereas ORA sets a bit if there is a set bit in either of the original bytes at that position, regardless of the combination, EOR will only set a bit if *one* of the original bytes had a set bit in that position. If *both* the original bytes have a one in the same position, the resulting byte will have a reset bit in that position.

Given below are some examples of the use of EOR:

11110000 EOR 00001111 = 11111111
10101010 EOR 01010101 = 11111111
11110000 EOR 11111111 = 00001111
10101010 EOR 00001111 = 10100101

EOR is primarily used for flipping the value of one or more bits, ie changing them to their opposite value regardless of their present state. In the chapter on signed arithmetic, for instance, we noted that if the overflow flag was set, it indicated that the sign bit of a value was incorrect and needed to be changed to its opposite. This can be done by applying the mask 10000000 (128 decimal) to the number. If bit seven is a one, EOR will reset, since both of the bits in the two bytes are set. If the bit seven is a zero, EOR will set it.

The procedure for the use of EOR is as follows:

1) Load the value to be used as a mask into the accumulator.

2) Apply the mask to a specified byte in memory using EOR.

3) The result will now be found in the accumulator and can either be used

to replace the original byte in memory or as the basis for some other action by the program.

Addressing modes available with EOR:

Zero page	: EOR $FF	: 45	: 45 FF
Immediate	: EOR #$FF	: 49	: 49 FF
Absolute	: EOR $02FF	: 4D	: 4D FF 02
Pre-indexed, X	: EOR ($FF,X)	: 41	: 41 FF
Post-indexed, Y	: EOR ($FF),Y	: 51	: 51 FF
Zero page, X	: EOR $FF,X	: 55	: 55 FF
Absolute, X	: EOR $02FF,X	: 5D	: 5D FF 02
Absolute, Y	: EOR $02FF,Y	: 59	: 59 FF 02

Example of the use of EOR

```
ADD.   DATA      SOURCE CODE
00                 10 PRT
00                 20 ORG $C030
C030               30 SYM
C030               40 FINISH = $C013
C030   A5FB        50 LDA $FB
C032   45FC        60 EOR $FC
C034   4C13C0      70 JMP FINISH
C037               80 END
```

```
6000 REM#********************************
6010 REM DATA FOR MACHINE CODE
6020 REM********************************
6030 DATA A5,FB,45,FC
```

Commentary
The example has the same structure as the previous two but applies EOR to the contents of the two memory locations.

The BIT instructions

In the section on AND we noted how that instruction could be used to test the contents of an individual bit in a specified location. There is, however,

another instruction which can be used for this purpose, though its applications are far more limited.

BIT is really a form of AND with a little added on, and the reason that it is not used more extensively is that it can only be used with two addressing modes.

The function of BIT is to provide:

1) A very straightforward test of the contents of bits 6 and 7 of a specified memory location and

2) To provide a rather less straightforward method of testing the state of an individual bit.

Testing bits 6 and 7 with BIT: Whatever the value in the accumulator, if you apply it to a specified memory location using BIT, the contents of bits 6 and 7 of the memory location will be transferred to bits 6 and 7 of the status register, which happen to be the sign and overflow flags. The contents of the accumulator are unchanged since, unlike AND, BIT does not place the result in the accumulator — the result is obtained and discarded. This is a useful technique for determining the sign of a number, since merely applying a BIT instruction to the address at which a number is held will set the sign flag to the sign of the number, without affecting the present contents of the accumulator.

Testing any bit with BIT: Apart from this automatic function, BIT can be used to test any bit in a specified memory byte by placing the appropriate mask in the accumulator eg 00001000 would be placed into the accumulator and applied to the specified memory byte to test the condition of bit 3. The zero flag would record whether the corresponding bit was set in the specified memory byte, and the accumulator would be unchanged. BIT appears at first sight to be a very useful command, since the mask placed in the accumulator is not corrupted when it is applied, as it is with AND. Unfortunately, the prospect of applying an unchanged mask to a series of bytes, which would have many applications, is spoiled by the fact that BIT can only be used in two addressing modes, so that a separate BIT instruction must be issued for every test. For this reason, BIT is seldom used other than for testing bit 7, where its advantages over AND are obvious.

Procedure for using BIT:

1) Unless you are simply testing bits 6 and 7, load the desired mask into the accumulator.

2) Apply it to the specified memory byte using BIT.

3) Test the sign and overflow flags for the contents of the 6th and 7th bits of the specified memory bytes.

4) Test the zero flag to see if the specified memory byte has any bits set at the same position as in the mask.

Addressing modes available with BIT:

Zero page	: BIT $FF	: 24	: 24 FF
Absolute	: BIT $02FF	: 2C	: 2C FF 02

Example of the use of BIT

```
ADD.   DATA      SOURCE CODE
00                 10 PRT
00                 20 ORG $C030
C030               30 SYM
C030               40 FINISH = $C013
C030   A908        50 LDA #$08
C032   85FB        60 STA $FB
C034               70 ;
C034   A904        80 LDA #$04
C036   24FB        90 BIT $FB
C038   4C13C0     100 JMP FINISH
C03B               110 END
```

```
6000 REM*********************************
6010 REM DATA FOR MACHINE CODE
6020 REM*********************************
6030 DATA A9,08,85,FB,A9,04,24,FB
```

Commentary
The effect of the ANDing of $08 and $04 using BIT is demonstrated by this routine. Note that the $04 loaded into the accumulator is still there at the end since the result of BIT is discarded. The zero flag, however, correctly displays that a zero was created — *i.e.* there were no matching bits.

Rotating and shifting

Having looked at some of the ways in which the individual digits in a byte can be changed, we now turn to another set of instructions which will allow us to move them about within the byte. The instructions are known as *shifts* and *rotates*, for reasons that will quickly become clear, since their effect is to move the bits within the byte to one side or another, in some cases transferring any bits which fall off one end of the byte to the other.

The shift instructions

The simplest form of this group of instructions is the shift. The two shift instructions are:

ASL – Arithmetic Shift Left
LSR – Logical Shift Right

The Arithmetic Shift Left instruction
Function: To move the bits in a specified location one place to the left. The contents of bit seven will be placed into the carry flag.

The simplest means of explaining the ASL instruction is to give you an example of what it does. Take the following byte, contained in the accumulator:

01010101
CARRY FLAG: 0

The ? in the carry flag indicates that the contents of the carry flag before the ASL instruction is carried out are irrelevant. Applying ASL results in the following:

10101010
CARRY FLAG: 0

Note that if you are operating with signed numbers, the sign bit is now to be found in the carry flag, so you may wish to make adjustments accordingly.

The procedure for applying ASL is as follows:

1) Identify the byte to be shifted. This may be in memory — there are four memory addressing modes available — or in the accumulator itself. The shift/rotate instructions are the only ones on the 6510 chip which

have an addressing mode (accumulator addressing) which allows them to be applied directly to the contents of the accumulator.

2) Apply ASL to the byte using the appropriate addressing mode.

3) If necessary, test the relevant flags. The zero flag will indicate whether the last set bit has been shifted out of the byte. The carry flag will indicate the contents of the sign bit of the byte before it was shifted.

The main uses of ASL are:

1) Multiplication by a power of 2. A simple shift of a byte to the left, effectively multiplies the value contained within the byte by 2. ASL on its own is of limited use in this respect since bits at the left hand end of the byte are lost if it is applied several times in succession. In these cases, ASL is used in conjunction with the rotate left instruction which we shall look at in a moment.

2) Used in a loop, ASL is an effective tool for examining the contents of each bit of a byte. Eight shifts will transfer each bit in turn to the carry flag, which can be easily tested, without the use of a mask.

The addressing modes available with ASL are:

Accumulator	: ASL A	: 0A	: 0A
Zero page	: ASL $FF	: 06	: 06 FF
Absolute	: ASL $02FF	: 0E	: 0E FF 02
Zero page, X	: ASL $FF,X	: 16	: 16 FF
Absolute, X	: ASL $02FF,X	: 1E	: 1E FF 02

Example of the use of ASL

```
ADD.   DATA      SOURCE CODE
00                 10 PRT
00                 20 ORG $C030
C030               30 SYM
C030               40 FINISH = $C013
C030   06FB        50 ASL $FB
C032   4C13C0      60 JMP FINISH
C035               70 END

6000 REM**********************************
6010 REM DATA FOR MACHINE CODE
```

```
6020 REM**********************************
6030 DATA 06,FB
```

Commentary
A single instruction is all that is required for you to use the Code Runner program to explore the effect of ASL on values of your choice placed into memory location $FB.

The Logical Shift Right instruction

Parallel to the ASL instruction is the LSR, though it is not exactly a mirror image of the former. Arithmetic shift left is so called because it preserves the sign bit (in the carry flag) it allows you to perform arithmetic operations conveniently — difficult if the sign of a number is being lost. Many processors have arithmetic shifts both left and right which allow the sign bit to be recovered. In addition, they have logical shifts, which merely shift the bits without making provision for the sign flag. This really only matters with shifts to the right. Shifts to the left almost universally place the most significant bit into the carry flag, so that any left shift can be called arithmetic.

If you are shifting bits to the right, however, to preserve the sign bit some special action must be taken, otherwise the sign bit will simply move across the bit 6 and only be detectable with something like AND or BIT. The 6510 instruction to shift to the right has no provision for preserving the sign bit somewhere accessible, so it is not called arithmetic shift right but logical shift right.

In all other respects, LSR is the same ASL. The bit which falls off the end of the byte is received into the carry flag, and the zero flag will indicate whether the shift has moved all the set bits out of the byte. Given below is an example of the use of LSR on particular byte:

10101010
CARRY FLAG:?
becomes
01010101
CARRY FLAG: 0

The procedure for the use of LSR is as follows:

1) Choose the byte to be operated upon which may, as with ASL, be the accumulator.

2) Apply LSR using the appropriate addressing mode.

3) According to your needs, test the carry flag for the previous contents of the zero bit or the zero flag to check whether the result of the LSR is zero. The sign of the number before LSR was applied to it can only be checked by sampling bit 6 of the result.

LSR is mainly used for:

1) Simple division by two on a single byte. In combination with other instructions it is an integral part of more complex forms of division. In combination with rotate instructions, it can be used to perform division by two on numbers which extend over more than one byte.

2) A test can be made of the contents of bit zero by shifting it into the carry flag, thus eliminating the need for a mask.

3) As with ASL, it can be used to test each bit of a byte in turn by loading them into the carry flag, again without the use of a mask.

The addressing modes available with LSR are:

Accumulator	: LSR A	: 4A	: 4A
Zero page	: LSR $FF	: 46	: 46 FF
Absolute	: LSR $02FF	: 4E	: 4E FF 02
Zero page, X	: LSR $FF,X	: 56	: 56 FF
Absolute, X	: LSR $02FF,X	: 5E	: 5E FF 02

Example of the use of LSR

```
ADD.   DATA      SOURCE CODE
00                 10 PRT
00                 20 ORG $C030
C030               30 SYM
C030               40 FINISH = $C013
C030   46FB        50 LSR $FB
C032   4C13C0      60 JMP FINISH
C035               70 END
```

```
6000 REM*********************************
6010 REM DATA FOR MACHINE CODE
6020 REM*********************************
6030 DATA 46,FB
```

Commentary
Once again, this machine code, patched into the Code Runner program will allow you to examine the effects of LSR on values placed into memory location $FB. You might like to pay particular attention to the effect of the instruction on numbers that you are regarding as having a positive or negative sign.

The rotate instructions

If you understand why shift instructions are called shift, ie because they move the individual bits one place to the left or right, then you should have no difficulty in seeing why the two rotate instructions, ROR (rotate right) and ROL (rotate left) are so called. Since ROR and ROL are literally mirror images of each other, they will be described together and once again the best way to understand these particular commands is to see them in action in an example:

10101010
CARRY FLAG: 1

applying ROR we get:

11010101
CARRY FLAG: 0

while with ROL:

01010101
CARRY FLAG: 1

Rotate is called rotate because bits which fall off one end of the byte are placed into the carry flag and the contents of the carry flag fed into the byte at the other end — the contents of the byte move through a circle consisting of the eight bits of the byte and the one bit of the carry flag — nothing is lost. If a rotate instruction is carried out nine times, the contents of the carry flag and the contents of the byte will be returned to exactly the state in which they started.

Rotate instructions set the flags in the same way as the shift instructions. If there is only one set bit, rotating it into the carry flag will set the zero flag. Rotating the byte in such a way that the sign bit becomes set will also set the sign flag.

The procedure for using either ROR or ROL is as follows:

1) Select the byte to be operated upon, which may be a byte in memory or the contents of the accumulator.

2) Ensure that the carry flag is at the value you desire, since it will be rotated into the main part of the byte.

3) Apply ROR or ROL to the byte using the appropriate addressing mode.

4) Test any flags as required by the program.
Rotate is very seldom used on its own, but rather in conjunction with shift instructions:

1) To carry out simple multiplication by 2 on a number held in more than one byte. The procedure is as follows:

a) clear the carry flag
b) shift the low byte of the number to the left with ASL
c) rotate the next byte of the number left using ROL. This adds the contents of the carry flag to the right hand end of the byte, ensuring that any overflow from the low byte is carried to the second.
d) repeat step c) for any subsequent bytes if the number consists of more than two bytes.

2) To carry out simple division by 2 on a number held in more than one byte. The procedure is as follows:

a) clear the carry flag
b) shift the high byte of the number to the right with LSR
c) rotate the next highest byte of the number right using ROR. This adds the contents of the carry flag to the left hand end of the byte, ensuring that any remainder from the high byte is carried to the next byte down.
d) repeat step c) for any subsequent bytes if the number consists of more than two bytes.

3) Both ROR and ROL can be used to reverse the order of the bits in a byte. A classic test given to students who have just learned about shift and rotate instructions is to ask them to devise a method for reversing the order of the bits in a byte using a series of shift and rotate instructions and two bytes of memory, one of which contains the byte to be reversed. Try it yourself, if you like — the answer is given below:

Answer: Shift the original byte in one direction and for each shift, rotate the second byte in the opposite direction. Do this eight times. Each shift

will take a bit from one end of the byte and place it into the carry flag. The rotate will feed the contents of the carry flag into the opposite end of the second byte.

The addressing modes available with ROL are:

Accumulator	: ROL A	: 2A	: 2A
Zero page	: ROL $FF	: 26	: 26 FF
Absolute	: ROL $02FF	: 2E	: 2E FF 02
Zero page, X	: ROL $FF,X	: 36	: 36FF
Absolute, X	: ROL $02FF,X	: 3E	: 3E FF 02

Example of the use of ROL in rotating a single byte value

```
ADD.   DATA      SOURCE CODE
00                 10 PRT
00                 20 ORG $C030
C030               30 SYM
C030               40 FINISH = $C013
C030   38          50 SEC
C031   26FB        60 ROL $FB
C033   4C13C0      70 JMP FINISH
C036               80 END
```

```
6000 REM*********************************
6010 REM DATA FOR MACHINE CODE
6020 REM*********************************
6030 DATA 38,26,FB
```

Example of the use of ROL in shifting a two-byte value

```
ADD.   DATA      SOURCE CODE
00                 10 PRT
00                 20 ORG $C030
C030               30 SYM
C030               40 FINISH = $C013
C030   06FB        50 ASL $FB
C032   26FC        60 ROL $FC
C034   4C13C0      70 JMP FINISH
C037               80 END
```

```
6000 REM#################################
6010 REM DATA FOR MACHINE CODE
6020 REM#################################
6030 DATA 06,FB,26,FC
```

Commentary
This routine follows the method outlined above for the rotation of a two-byte value to the left.

Line 50: The low byte of the number is *shifted* to the left, thus placing its high bit into the carry flag.

Line 60: Rotating the high byte ensures that the contents of the carry flag are brought in as the least significant bit of the result. In practice, if you place two-byte values into $FB-$FC you will find that the routine multiplies them by two provided that the original two byte value does not exceed 32767 (ie 127 in location $FC and 255 in location $FB, because 32767=255+256*127).

The addressing modes available with ROR are:

Accumulator	: ROR A	: 6A	: 6A
Zero page	: ROR $FF	: 66	: 66 FF
Absolute	: ROR $02FF	: 6E	: 6E FF 02
Zero page, X	: ROR $FF,X	: 76	: 76 FF
Absolute, X	: ROR $02FF,X	: 7E	: 7E FF 02

Example of the use of ROR for shifting two-byte values

```
ADD.   DATA      SOURCE CODE
00                 10 PRT
00                 20 ORG $C030
C030               30 SYM
C030               40 FINISH = $C013
C030   46FC        50 LSR $FC
C032   66FB        60 ROR $FB
C034   4C13C0      70 JMP FINISH
C037               80 END

6000 REM#################################
6010 REM DATA FOR MACHINE CODE
6020 REM#################################
6030 DATA 46,FC,66,FB
```

Commentary
The opposite to the procedure in the last example routine.

Example of the use of ROR on single byte signed numbers

```
ADD.   DATA      SOURCE CODE
00                 10 PRT
00                 20 ORG $C030
C030               30 SYM
C030               40 FINISH = $C013
C030   38          50 SEC
C031   24FB        60 BIT $FB
C033   3001        70 BMI L000
C035   18          80 CLC
C036               90 L000
C036   66FB       100 ROR $FB
C038   4C13C0     110 JMP FINISH
C03B               120 END
```

```
6000 REM*********************************
6010 REM DATA FOR MACHINE CODE
6020 REM*********************************
6030 DATA 38,24,FB,30,01,18,66,FB
```

Commentary
The problem tackled by this little routine has already been mentioned, and that is to preserve the sign bit of a value when the main body of the value is rotated. The manner in which this is done is to use another branch instruction, equivalent to BASIC's GOTO, but this time a jump is made if a number sampled is negative.

Lines 50–80: The carry flag is first set, then BIT applied to the chosen memory location. This, as we have seen, transfers the contents of bits six and seven to the overflow and sign flags. Lines 70 and 80 now clear the carry flag again if the BIT instruction has not set the sign flag, indicating that the value in memory location $FB is negative. The result of the process is that if the value in $FB is negative, the carry flag will be set, while if the value is positive the carry flag will be clear.

Line 100: The value in memory location $FB is now rotated to the right,

placing the contents of the carry flag, the original bit seven, back into its correct position and ensuring that the sign of the number does not change.

At this point, some readers will no doubt be hopping up and down because we have not mentioned the fact that the original sign bit, which was not part of the value, merely an indicator, has been rotated into the main body of the number. Never fear, the almost uncanny usefulness of two's complement arithmetic, which we use for signed numbers, takes care of this.

Supposing we take the negative number 10000001, or −127 in decimal and then rotate it according to the method above. The result is the binary number 11000000, with the original sign bit having moved to bit 6. But when we translate this number back into decimal we find that it is actually-64. Simply following the rules of two's complement arithmetic ensures a result that is sensible within the limits of an eight bit number.

Example of the use of ROR on double byte signed numbers

```
ADD.   DATA      SOURCE CODE
00                 10 PRT
00                 20 ORG $C030
C030               30 SYM
C030               40 FINISH = $C013
C030   38          50 SEC
C031   24FC        60 BIT $FC
C033   3001        70 BMI L000
C035   18          80 CLC
C036               90 L000
C036   66FC       100 ROR $FC
C038   66FB       110 ROR $FB
C03A   4C13C0     120 JMP FINISH
C03D              130 END

6000 REM*********************************
6010 REM DATA FOR MACHINE CODE
6020 REM*********************************
6030 DATA 38,24,FC,30,01,18,66,FC
6040 DATA 66,FB
```

Commentary

This second routine performs a shift to the right on a two-byte signed number, with the sign bit being preserved by the methods described for the previous routine.

CHAPTER 13
Making Comparisons

Now that we can add and subtract, we can proceed to a set of instructions which are, in practice, essential for the kind of addition and subtraction which is done in the average machine code program. These are known as the compare instructions and their function is to test whether two chosen values are equal or one is greater than the other. The three compare instructions are some of the most widely used in programming, just as the '<' and '>' operators are widely used in BASIC as the basis on which decisions are made. In addition, since there is no built in instruction like FOR . . . NEXT in machine code to provide automatic repetitions of a series of instructions within a loop, the compare instructions are widely used in creating loops which function like a simple BASIC construction like:

```
10 I=0
20 PRINT "THIS IS REPETITION NUMBER ";I
30 I=I+1
40 IF I<10 THEN GOTO 20
```

The compare instructions can be confusing for someone beginning machine code since they work on the same principles as SBC, subtracting a value from the contents of the accumulator and setting the flags appropriately, even though the results of the subtraction are discarded and the value in the accumulator is not altered. The most important flag in reading the results of a comparison is the carry flag, and you will recall from the previous chapters that in subtraction, if a value is subtracted from a larger value in the accumulator, then the carry flag is *set*, since subtraction works on the kind of logic used by two's complement arithmetic, where the roles of 1 and 0 are reversed.

The CMP instruction

Function: To subtract the contents of a specified memory byte from the contents of the accumulator and set the flags according to the result. The result itself is not retained and neither of the values used in the comparison are altered in any way.

The procedure for the use of CMP is as follows:

1) Ensure that one of the values to be compared is in the accumulator.

2) Choose the location in memory of the byte containing the value or specify that value in the program itself.

3) Apply CMP to the two values using an appropriate addressing mode, as specified in the table at the end of this section.

4) Examine the flags to ascertain the result of the comparison:
a) Zero flag: if this is set, then the two values are the same, since the result of the subtraction was zero.
b) Carry flag: if the carry flag is set, then the value in the accumulator is greater than or equal to the value with which it was compared — which of these two possibilities is the case can be determined by examining the zero flag. If the carry flag is clear, then the value in the accumulator is less than the value with which it was compared.

In many cases, comparisons have to be carried out on two-byte values. This is a more complex procedure which requires the use of instructions which we have not yet examined. In simple terms, the procedure is as follows:

1) Place the high byte of the first number to be compared into the accumulator.

2) Use CMP to compare it with the high byte of the second number.

3) Examine the zero flag. If it is clear, then the carry flag will indicate which of the two numbers is the greater, without having to make any further tests. If the zero flag is set, then the following steps will have to be taken:

4) Load the low byte of the first value to be compared into the accumulator.

5) Use CMP to compare with the low byte of the second number.

6) The carry and zero flags will now give the result of the comparison for the whole of the two byte number.

Note: If there are more than two bytes to the numbers, omit step six and repeat as necessary from the decision taken at step 3. The only reason that

you cannot do this at the moment is that we have not yet explained how to skip around program lines which are not needed, like steps 4–6.

Addressing modes available with CMP:

Zero page	: CMP $FF	: C5	: C5 FF
Absolute	: CMP $02FF	: CD	: CD FF 02
Immediate	: CMP #$FF	: C9	: C9 FF
Pre-indexed, X	: CMP ($FF,X)	: C1	: C1 FF
Post-indexed, Y	: CMP ($FF),Y	: D1	: D1 FF
Zero page, X	: CMP $FF,X	: D5	: D5 FF
Absolute, X	: CMP $02FF,X	: DD	: DD FF 02
Absolute, Y	: CMP $02FF,Y	: D9	: D9 FF 02

Example of the use of CMP

```
ADD.   DATA      SOURCE CODE
00                 10 PRT
00                 20 ORG $C030
C030               30 SYM
C030               40 FINISH = $C013
C030   A955        50 LDA #$55
C032   C966        60 CMP #$66
C034   4C13C0      70 JMP FINISH
C037              110 END
```

```
6000 REM#********************************
6010 REM DATA FOR MACHINE CODE
6020 REM********************************
6030 DATA A9,55,C9,66
```

Commentary
The routine uses immediate mode addressing to compare the values $55 and $66 and so is more limited in its scope than those which have gone before. A useful exercise for you would be to convert it so that it loaded the contents of, say, memory location $FB into the accumulator and then compared with the contents of memory location $FC.

The CPX and CPY instructions

The Compare X and Compare Y instructions are parallel to the CMP instruction in function, differing only in that instead of subtracting the contents of a specified memory byte from the contents of the accumulator, it is subtracted from either the X or Y register, depending on the instruction used.

The CPX and CPY instructions are more limited than CMP in that only three addressing modes are available to them, as shown in the table below.

The procedure for the use of CPX and CPY is exactly the same as that for CMP, except for the limitations imposed by the relatively fewer addressing modes and the fact that either the X or Y register will replace references to the accumulator.

CPX and CPY are almost always used for the testing of loop variables within programs since loops often form the basis of a series of operations on memory bytes using indexed addressing. Using one of the index registers both to define the memory byte to be operated upon and to determine the position within the loop is a very economical method of programming.

Addressing modes available with CPX:

Immediate	: CPX #$FF	: E0	: E0 FF
Zero page	: CPX $FF	: E4	: E4 FF
Absolute	: CPX $02FF	: EC	: EC FF 02

Addressing modes available with CPY:

Immediate	: CPY #$FF	: C0	: C0 FF
Zero page	: CPY $FF	: C4	: C4 FF
Absolute	: CPY $02FF	: CC	: CC FF 02

Example of the use of CPX

```
ADD.   DATA      SOURCE CODE
00                 10 PRT
00                 20 ORG $C030
C030               30 SYM
C030               40 FINISH = $C013
C030   A255        50 LDX #$55
C032   E055        60 CPX #$55
C034   4C13C0      70 JMP FINISH
C037              110 END
```

```
6000 REM#********************************
6010 REM DATA FOR MACHINE CODE
6020 REM********************************
6030 DATA A2,55,E0,55
```

Example of the use of CPY

```
ADD.   DATA      SOURCE CODE
00                 10 PRT
00                 20 ORG $C030
C030               30 SYM
C030               40 FINISH = $C013
C030   A055        50 LDY #$55
C032   C044        60 CPY #$44
C034   4C13C0      70 JMP FINISH
C037              110 END
```

```
6000 REM#********************************
6010 REM DATA FOR MACHINE CODE
6020 REM********************************
6030 DATA A0,55,C0,44
```

CHAPTER 14
Program Control

If you can imagine trying to program in BASIC without any form of GOTO, GOSUB, IF . . . THEN or FOR . . . NEXT instructions — in other words no way of executing any part of the program other than in the strict order in which the program is written — then you will realise that this chapter is a very important one in developing your understanding of machine code programming. In this chapter we shall look at the way in which the machine code program can be made to carry out all the different kinds of decisions as to which parts of the program are to be used under which circumstances. Programs can be written which do not require decisions or any changes in the order in which tasks are carried out, but such programs are seldom useful. One of the essential features of useful programming is that the program looks at what is happening as it is being executed and then makes decisions as to how to proceed. Such decisions involve choosing to perform one task rather than another, and that can only be done by jumping from one part of the program to another.

There are three methods of moving around a program provided to you in machine code: absolute jumps, relative branches and calls to subroutines (including returns from subroutines).

The usual method of executing a machine code program is to choose a starting point within memory at which to start examining and carrying out the instructions which make up the program. This, by the way, is one of the first points at which a program can break down. Take the following two bytes, which we shall assume are to be found at locations $C000 and $C001:

24 18

This is the machine code form of BIT $18, an instruction that we looked at in Chapter 13, and it would be carried out if we were to execute the program starting at $C000. If, however, you were to start executing the program not at $C000 but at $C001, the instruction would be read as simply 18, which is the opcode for an entirely different instruction, clear carry.

The point at which you jump into a machine code program can make an

enormous difference to what happens when you begin executing it. If you do not start in the correct place, then in most circumstances you will crash your system — at the very least the program is not going to do what it was designed for. This is an important point when it comes to program control, and not only when we are thinking about starting a program. Later on we shall see that a major problem in designing a program can be to ensure that when program execution jumps from one place to another, it jumps into the right place, rather than to the middle of an instruction.

After that digression, let's return to the question of how a machine code program is executed. Having been told to begin executing a program at a certain point, the CPU examines the byte at that point and translates it, if possible, into an instruction. Sometimes, the instruction requires some more information, *e.g.* an instruction with the immediate mode of addressing built in would require the CPU to pick up another byte from the program as the operand for the instruction. Apart from having to carry out the instruction, the CPU now has also to move past the instruction that it has just read, including its operand, to the start of the next instruction. This is accomplished by means of a special register known as the program counter, which is used to keep a record of where the CPU has reached in the execution of machine code program in the memory. The CPU has a built in knowledge of how long every instruction is, including its operand, and so has no difficulty in moving to the beginning of the next instruction. The execution of the program proceeds in this way in irregular steps of one, two or three bytes, as the CPU interprets instructions and moves on to the next.

All this goes on until the programmer includes an instruction in the program which dictates a different order to the way in which it is to be processed. The essence of such an instruction will be to tell the CPU not to execute the next instruction but to jump to a specified point in memory and to begin execution of the program there. The point in memory can be spelled out in the form of a two-byte address, in which case the program makes what is known as an absolute jump. Alternatively, the point to which a jump can be made can be specified as a relative branch, that is to say a number of bytes forward or backwards compared to the current position in the program. Relative branch instructions come in a variety of forms which parallel the IF . . . THEN . . . GOTO format in BASIC, allowing something to be tested (a flag) and a jump to be made (or not) on the basis of the result. Finally, the BASIC command GOSUB has its parallel in machine code in the form of an instruction to jump to another part of the program but to remember where execution had got to when the jump was made. Once the jump has been made, a return instruction like RETURN in BASIC brings execution back to the original point.

With that overview in mind, let's turn to the commands in detail.

Absolute jumps using JMP

Function: To allow the programmer to specify a position in memory to which execution of the program will jump when the JMP instruction is encountered.

There are good and bad reasons for using the JMP instructions, just as there are good and bad reasons for using GOTO in BASIC. The bad reasons usually boil down to the fact that the program is written in the wrong order and the JMPs (or GOTOs) are needed to execute the jumbled parts in the right order.

Good reasons are not as frequent as most programmers seem to assume, and many JMP instructions are simply there to redeem bad programming. Good reasons for the use of JMP might include:

1) The need to jump into a routine in the ROM of the 64. There are sometimes good reasons not to use JSR, the equivalent of BASIC's GOSUB to do this but they will be examined in the section on that instruction.

2) The creation of loops where the decision as to whether a loop will continue is not taken at the end. A very simple example in BASIC might be:–

```
10 INPUT A$
20 IF A$="STOP" THEN STOP
30 PRINT A$
40 GOTO 10
```

Here the GOTO in line 40 is an unconditional command, with no IF attached to it, since the decision is taken earlier in the loop.

3) Ending sections of a program where several different program sections, not all of which are to be used, have been written in sequence. Take the following simple BASIC program:

```
10 INPUT "GIVE ME A NUMBER BETWEEN 1 AND 10";NUMBER
20 IF NUMBER>=1 AND NUMBER<=10 THEN 50
30 PRINT "I SAID BETWEEN 1 AND 10. IDIOT!"
40 GOTO 60
50 PRINT "YOUR NUMBER WAS ";10-NUMBER;" LESS THAN 10"
60 INPUT "ANY MORE (Y/N):";Q$
70 IF Q$="Y" THEN GOTO 10
```

In this case there is an unconditional jump made at line 60 and it is a perfectly sound piece of programming, since having arrived at line sixty

there can be no doubt that line 50 is intended to be skipped — the decision has already been made at line 20.

4) JMP with indirect addressing mode (the only instruction to use this mode) is an effective way of executing one part of a program at the direction of the user. Thus the user might specify a number from 1 to 4 on the basis of a menu on the screen. On the basic of the input, one of four addresses would be transferred from a table to a known location. Finally JMP indirect would send program execution to that address.

These are just some examples drawn from parallels in BASIC. They are not intended to be exhaustive but they are intended to remind you that JMP is a good thing in its place but it is best to ensure it *has* to be used or your programs could become a mass of jumps as you seek the easy way out of bad program writing.

The procedure for using JMP is straightforward:

1) Select the memory location to which you wish the program to jump.

2) Include a JMP instruction in the program at the point at which you wish the jump to be made.

3) When program execution reaches the JMP instruction, the jump will be effected.

Addressing modes available with JMP:

Absolute	: JMP $FFF0	: 4C	: 4C F0 FF
Indirect	: JMP ($FFF0)	: 6C	: 6C F0 FF

Important note
Due to a bug in the 6510 chip itself, there is one important limitation on the use of JMP in indirect mode. If the address from which the final address is being picked up ends in FF, like 01FF, instead of taking the next byte (0200) as the second byte of the address, the processor will shuttle back to the beginning of that page in the memory (0100). This is a fault in the processor and there is no easy way round it, so caution must be exercised if nonsense addresses are not going to be created.

Example of using an indirect jump

```
ADD.   DATA      SOURCE CODE
00                 10 PRT
```

```
00                    20 ORG $C030
C030                  30 SYM
C030                  40 FINISH = $C013
C030  A913            50 LDA #$13
C032  A2C0            60 LDX #$C0
C034  85FB            70 STA $FB
C036  86FC            80 STX $FC
C038                  90 ;
C038  6CFB00         100 JMP ($FB)
C03B                 110 END
```

```
6000 REM#*********************************
6010 REM DATA FOR MACHINE CODE
6020 REM*********************************
6030 DATA A9,13,A2,C0,85,FB,86,FC
6040 DATA 6C,FB,00
```

Commentary
Lines 50–80: The address for the FINISH routine is loaded into $FB–$FC.
Line 100: The indirect jump is made on the basis of the address in $FB–FC, and FINISH is executed.

Using subroutines with JSR and RTS
Function: To allow a section of the program to be executed by a JSR call from another part of the program. Execution of this secondary section will continue until an RTS instruction is reached. At the point, execution of the program will be continued from the instruction following the original JSR instruction.

Subroutines are an essential aspect of good programming. They allow the main flow of a program to continue while minor tasks are handled in clearly identifiable units of the program. This tends to make programs easier to visualise and write and, not the least important, it makes them easier to read when you come to make changes later in the program's life. Subroutines also allow programs to be written more briefly, since the same subroutine to perform a common task can be used by several different parts of the program.

The subroutine instructions available to you on the 6510 chip are no different in their essentials than BASIC's GOSUB and RETURN. To use a subroutine in machine code you have first to write your subroutine, then to call it using the JSR instruction. The subroutine will be concluded

when an RTS instruction is found and the main program will continue from the instruction following the original JSR.

One point to remember, though it has been mentioned before, is that all of this works only if you do not corrupt the stack by adding or subtracting items from it during the course of the subroutine. If you need the use of the stack during the course of the subroutine, of course you are free to, but do ensure that you take off all the values you placed on and only those values or the RTS will return execution to a nonsense position in the program.

In addition, it needs to be remembered that each JSR instruction places its return address onto the stack. The actual address recorded is two bytes on from the byte containing the JSR opcode itself. Since JSR has only one addressing mode, absolute, which requires two bytes to specify an address to be jumped to, the addition of two always suffices to create an address pointing to the last byte of the JSR instruction. If you call a subroutine and then call another while you are in the first, you will have placed two return addresses on the stack. Any return from the second subroutine will return you, not to the original position but to the first subroutine — this is known as *nesting* of subroutines. On occasion programmers avoid this nesting by using a simple jump, JMP, to call a subroutine. The procedure is to call subroutine 1 which in turn calls subroutine 2 at some stage. The twist is that while subroutine 1 is called with a JSR instruction, subroutine 2 is called with JMP. The result is that when an RTS is encountered, the return is made not to subroutine 1 but to the original JSR. An example of this might be two subroutines which use the same ending. Rather than write the ending twice, it is written once at the end of one of the subroutines, terminating with an RTS. The other subroutine, when it reaches the point where the ending would have been, jumps with JMP to the last part of the first subroutine and uses that ending, including its RTS.

Addressing modes available with JSR:

Absolute	: JSR $FFF0	: 20	: 20 F0 FF

Addressing modes available with RTS:

Inherent	: RTS	: 60	: 60

Example of using RTS

```
ADD.   DATA      SOURCE CODE
00                 10 PRT
```

```
00                      20 ORG $C030
C030                    30 SYM
C030                    40 FINISH = $C013
C030  A9C0              50 LDA #$C0
C032  48                60 PHA
C033  A912              70 LDA #$12
C035  48                80 PHA
C036  60                90 RTS
C037                   100 END
```

```
6000 REM#********************************
6010 REM DATA FOR MACHINE CODE
6020 REM********************************
6030 DATA A9,C0,48,A9,12,48,60
```

Commentary
Lines 50–80: The address of finish is placed onto the stack:

Line 90: Executing RTS leads the chip to take the first two bytes off the stack, treat them as an address and then jump to that address.

The relative branch instructions

Function: To execute a jump within the range −128 to +127 memory bytes, counting from the byte after the branch instruction, on the basis of the condition of one or other of the flags.

In addition to the two forms we have already examined, the 6510 chip provides you with a range of other program control instructions which are nearer in form to the IF . . . THEN . . . GOTO format in BASIC. These are known as relative branches, with the word relative indicating that instead of having to specify the whole of an address in memory, the programmer has only to give a position relative to the current position of the program counter.

Relative branches use only one byte in specifying the jump to be made and this jump can be either forwards or backwards in memory. What this means is that we are back to the familiar topic of single byte signed numbers, and you will not be surprised to learn that a jump can be made in the range −128 to +127 bytes. The position from which the jump is made is the byte after the branch instruction. Some general examples of how this works, using an imaginary instruction, BRANCH, may help:

BRANCH 0
LDA #$FF

The effect of the branch instruction here is to do nothing at all. Since the base address for the branch is anyway the first byte after the instruction, ie the first byte of LDA #$FF, jumping zero bytes from that point will have no effect on the flow of the program whatsoever.

```
BRANCH -2
LDA #$FF
```

This is an infinite loop. Like all the branch instructions, our imaginary BRANCH is two bytes long. If we jump back two bytes from the beginning of the LDA instruction we shall find ourselves back at the beginning of BRANCH, and so on ad infinitum.

Finally:

```
BRANCH 2
LDA #$FF
CLC
```

Here the BRANCH instruction leads to the two byte LDA instruction being totally ignored, as execution jumps to the byte holding the opcode for CLear Carry.

Most of the problems which newcomers to machine code experience when using branch instructions arise simply out of miscounting, counting from the wrong place like the beginning of the branch instruction or forgetting that the address is a signed number. If you count with care and from the correct place, there is no reason for you to encounter any difficulty.

The conditions for a branch

We have already said that the branch instructions are rather like the IF . . . THEN GOTO construction in BASIC, but how? In fact, the 6510 gives you the option of testing for eight different conditions and making a branch if one of those conditions are fulfilled. The eight different conditions are based around four flags: carry, zero, sign and overflow.

For each of these flags there is one form of the branch instruction which will execute a branch if the flag is set and one which will execute a branch if the flag is clear. Given below is a table setting out the eight branch instructions in relation to their respective flags:

Flag	**Branch if set**	**Branch if clear**
Carry	Branch Carry Set (BCS)	Branch Carry Clear (BCC)
Zero	Branch Equal (BEQ)	Branch Not Equal (BNE)

Sign	Branch Minus (BMI)	Branch Plus (BPL)
Overflow	Branch O/flow Set (BVS)	Branch O/flow Cler (BVC)

BCS and BCC, the branches relating to the carry flag, are essential in routines for the various forms of arithmetic and for comparisons. Decisions frequently have to be made on the basis of whether the result of an arithmetic operation has exceeded the range of values which can be held in a single byte, and these two branches enable such decisions to be taken without strain. In Chapter 13 we looked at the comparison of numbers and noted that the carry flag records whether one is greater than the other. BCS and BCC are frequently used to send execution to different routines depending on the results of such a comparison. In Chapter 12 we looked at the shift instructions and noted how they could be used to move the contents of a byte, one bit at a time, into the carry flag to see which bits were set and which clear. Once again, any action based on the result is likely to be based on the use of BCS and BCC.

The BNE, and BEQ instructions are most often used in the case of comparisons. You may remember that in Chapter 12 we gave the outline of a procedure for comparing two byte numbers. The first act was to compare the high bytes and, if they were the same, to go on and compare the low bytes. The details were left deliberately vague because to accomplish the routine you need to use the BNE instruction to jump around the comparison of the low bytes if everything is decided by the comparison of the low bytes. BNE is frequently used at the end of loops, especially those which are to be executed more than 127 times (see the sign flag branches below). Here a register will be decremented each time the program passes through the loop and when the register reaches zero, BNE will no longer send execution back to the beginning of the loop.

BMI and BPL relate to the sign flag and are obviously useful in making decisions when it comes to signed arithmetic. BPL is also frequently used for loops which have to be carried out less than 128 times, since it allows the loop to be carried on up until the register being used to count the repetitions reaches zero. Only when a further repetition would send the register below zero does BPL cease to send execution back to the beginning of the loop.

The BVC and BVS instructions, which relate to the overflow flag, are used almost exclusively in signed arithmetic.

Branches and loops

In what goes above you will have found constant reference to loops and the use of branch instructions to create them. Loops are a vital part of machine code programming, even more so than in BASIC, since even quite straightforward functions often require a simple operation to be

carried out over and over again — like clearing the screen in machine code, which requires every byte of the screen memory to be cleared in turn.

For an example of the use of loops to create a delay in a program see Chapter 17.

Table of branch instructions:

On Carry Set	: BCS $FE	: B0	: B0 FE
On Carry Clear	: BCC $FE	: 90	: 90 FE
On Equal (Zero set)	: BEQ $FE	: F0	: F0 FE
On Non-Equal (Zero clear)	: BNE $FE	: D0	: D0 FE
On Minus (Sign set)	: BMI $FE	: 30	: 30 FE
On Plus (Sign clear)	: BPL $FE	: 10	: 10 FE
On Overflow Set	: BVS $FE	: 70	: 70 FE
On Overflow Clear	: BVC $FE	: 50	: 50 FE

Examples of the use of branches

In each of these two examples, there are two branch instructions to execute the FINISH routine, both of them based on the condition of the carry flag. Running both routines and looking at the contents of the accumulator will demonstrate that in the first example the first branch is carried out and the second ignored, while in the second illustration the second branch is carried out and the first ignored.

```
ADD.   DATA      SOURCE CODE
00                 10 PRT
00                 20 ORG $C030
C030               30 SYM
C030               40 FINISH = $C013
C030   A900        50 LDA #00
C032   18          60 CLC
C033   90DE        70 BCC FINISH
C035   A9FF        80 LDA #$FF
C037   B0DA        90 BCS FINISH
C039              100 END
```

```
6000 REM*********************************
6010 REM DATA FOR MACHINE CODE
6020 REM*********************************
6030 DATA A9,00,18,90,DE,A9,FF,B0
6040 DATA DA
```

```
ADD.   DATA      SOURCE CODE
00                 10 PRT
00                 20 ORG $C030
C030               30 SYM
C030               40 FINISH = $C013
C030   A900        50 LDA #00
C032   38          60 SEC
C033   90DE        70 BCC FINISH
C035   A9FF        80 LDA #$FF
C037   B0DA        90 BCS FINISH
C039              100 END
```

```
6000 REM*********************************
6010 REM DATA FOR MACHINE CODE
6020 REM*********************************
6030 DATA A9,00,38,90,DE,A9,FF,B0
6040 DATA DA
```

CHAPTER 15
Interrupts

In this chapter we turn to a more specialised kind of program control in the form of the BRK command, and its companion RTI.

BReaK exists for a rather unusual reason — its original purpose is to allow professional programmers to repair mistakes that they have made. You will know that the basis of your 64, apart from its 6510 CPU, is a set of complex machine code programs built into what is known as read only memory, chips which are designed with their programs permanently built in — which is why they are not lost when the 64 is switched off.

Such ROM programs are developed on another form of chip which is slightly more flexible, a PROM, or programmable ROM. The problem with such chips is that if, after the extensive development they undergo, a problem with the programming is found — and it almost inevitably will be — altering the whole structure of the program in the chip can be very expensive and time consuming.

The obvious solution is to provide what is known as a patch, a subroutine placed into some spare memory in the chip so that the main program does not have to be moved around. The patch will be activated by a call placed into the main program. Two problems now arise. Before a chip has a program etched into it, it is usually found with every bit set — every byte has the value $FF. The reason for this is that with the current generation of ROM chips it is relatively easy to permanently clear a bit which is set, but difficult to set permanently a bit which is clear. In other words, once a program has been installed, and some bits cleared as part of the process, it becomes very difficult to do anything else with them. To call the patch we would need to place a JSR instruction into the program contained in ROM, and if this involves setting any bits it could turn out to be impossible. The way around this was to provide an instruction with no bits set, an instruction whose opcode must therefore be zero. Whatever the state of the bits in a byte to be overwritten, they can always be cleared to zero. In addition, the instruction was only one byte long so that if all that needed to be replaced was a single byte instruction in the ROM, the new instruction would fit without having to overwrite other instructions — to insert a JSR instruction would require three bytes, even if it were possible to set the bits correctly.

The time and money saving instruction was called BRK. Its function

was to tell the chip to look for an address at a specific point in memory, to take an address from that point and then execute a machine code routine which is known as an interrupt routine from that address. The point at which the address is to be found is built into the 6510 itself, which is why the single zero of a BRK instruction does not need an address attached.

The BRK instruction as it exists on the Commodore 64 is extremely useful — courtesy of Commodore themselves. When the 6510 encounters a BRK instruction in a Commodore 64 machine code program the chip, as usual, looks to the place where it expects to find an address and is directed to a routine built into the ROM. From there, its attention is directed to the two bytes at $316–317 (790–1 decimal) which are in RAM. From these two bytes it takes an address and carries out any machine code instructions found at the point indicated by that address.

The advantage of this from the programmer's point of view is that since $316–7 are in RAM, the user can store there the address of a machine code routine to be executed and then call up that routine in a single byte with BRK.

Example of the use of BRK

```
ADD.   DATA      SOURCE CODE
00                 10 PRT
00                 20 ORG $C030
C030               30 SYM
C030               40 FINISH = $C013
C030   A913         50 LDA #$13
C032   A2C0         60 LDX #$C0
C034   8D1603       70 STA $316
C037   8E1703       80 STX $317
C03A                90 ;
C03A   00          100 BRK
C03B               110 END
```

```
6000 REM#********************************
6010 REM DATA FOR MACHINE CODE
6020 REM********************************
6030 DATA A9,13,A2,C0,8D,16,03,8E
6040 DATA 17,03,00
```

Commentary

Lines 50–60: These two bytes loaded into the A and X registers represent

the address of the FINISH routine which tidies up the registers before returning the Code Runner program to BASIC. In most of the demonstrations FINISH has been called up with a jump instruction.

Lines 70–80: The address of FINISH is placed into the two byte register at $316–7.

Line 100: A BRK instruction is issued. Since the address of FINISH has been placed into $316–7, the result of the BRK is simply to execute FINISH.

Returning with RTI

The correct command to end the execution of a subroutine called by means of BRK is RTI, which stands for ReTurn from Interrupt. On finding an RTI instruction, provided that the subroutine was called with BRK, program execution will return to the point following the BRK.

Confusion often arises as to the difference between RTS, return from subroutine, and RTI but it is most important that you distinguish between them.

When a subroutine is called with JSR, a two byte address is placed on the stack, that address being the last byte of the JSR instruction. When the program returns with RTS, the address is taken back off the stack, the program counter is incremented by one and the instruction following the JSR is executed.

BRK works very differently, storing first a two byte address *and* another byte representing the contents of the status register. Not only is an extra byte stored, the address placed on the stack is the address of the next instruction following BRK. When an RTI is encountered, the program counter and status register are restored but the program counter is not incremented — it is already pointing to the next instruction.

It is clear, then, that you *cannot* end a subroutine called with BRK with an RTS instruction. When the RTS picks up a return address from the stack, the two bytes will actually consist of the original contents of the status register and one byte of the original address in the program counter. In programs for the 6502 chip, which is almost identical to the 6510, you will sometimes find interrupt routines ended with PLP (pull status register off the stack), followed by RTS. This does return the correct address to the program counter but on the 6510 it does not deal with the fact that RTS will increment the program counter so that it misses the next byte of the program. If you come across PLP, RTS in a 6502 program — most of which will work on the 6510 — you must be sure that if it is being used to end an interrupt routine, you replace the two commands with a single RTI.

Example of the use of RTI

```
ADD.   DATA      SOURCE CODE
00                 10 PRT
00                 20 ORG $C030
C030               30 SYM
C030               40 FINISH = $C013
C030   A9C0         50 LDA #$C0
C032   48           60 PHA
C033   A913         70 LDA #$13
C035   48           80 PHA
C036   08           90 PHP
C037   40          100 RTI
C038               110 END


6000 REM**********************************
6010 REM DATA FOR MACHINE CODE
6020 REM**********************************
6030 DATA A9,C0,48,A9,13,48,08,40
```

Commentary
Lines 50–80: The address of the FINISH routine is placed directly onto the stack.

Line 90: The contents of the status register are also placed on the stack. There is no special relevance to the current contents of the register it is simply that the RTI needs an extra value on the stack so that it correctly picks up the address of FINISH. Any value would do in this particular case but in others it might be essential that the value stored was the true contents of the status register.

Line 100: A single RTI suffices to execute the FINISH routine and return the program to BASIC.

Addressing modes available with BRK and RTI
The only mode available with either of these instructions is the inherent mode, since the instructions explicitly imply the registers they will act upon.

Inherent	: BRK	: 00	: 00
Inherent	: RTI	: 40	: 40

CHAPTER 16
The Stack

In the early chapters of the book, when we looked in a general way at the working methods of a CPU we came across the concept of a stack — a queue of values to which new values can be added at one end and from which values can be taken, again from the same end.

On the Commodore 64 an area of memory which stretches from 256($0100) to 511($01FF) is set aside for the stack — a fact which it is wise to remember before you go storing values right in the middle of it. The order in which items are stored on the stack is in fact the reverse of what you might expect since the first item onto the stack is placed at 511 and subsequent values are placed in successively lower addresses. None of this need worry you, since it is very unlikely that you will ever want to access the stack memory directly. For all practical purposes it is easier to talk as if values were placed onto the stack and picture it as a pile to which items are added at the top and from which only the top item can be removed.

BASIC program to illustrate a first-in/first-out (FIFO) stack

The program is designed to picture on the screen the simplified stack shown in the illustration in Chapter 1.

```
10 REM PROGRAM TO SHOW FIFO STACK
20 SPTR = 0
30 FOR I = 0 TO 7
40 GOSUB 170
50 INPUT "INPUT~NUMBER~TO~PUT~ON~STACK:";T(SP
TR)
60 SPTR = (SPTR+1) AND 7
70 NEXT I
80 GOSUB 170
90 FOR I = 0 TO 7
100 INPUT "HIT~RETURN~TO~REMOVE~ITEM~FROM~ST
ACK";T$
110 SPTR = (SPTR-1) AND 7
120 GOSUB 170
```

```
130 PRINT "ITEM~REMOVED~WAS:" T(SPTR)
140 PRINT
150 NEXT I
160 END
170 REM*******************************
180 REM PRINT STACK
190 REM*******************************
200 PRINT "[CLEAR]"
210 PRINT "~~~~~~~~~~~~~~[C= A][^C][^C][^C][^C]
[^C][^C][C= S]"
220 FOR J = 0 TO 7
230 T$ = RIGHT$("~~~~~~~~"+STR$(T(J)),6)
240 PRINT "~~~~~~~~~~~~~~[^B]" T$ "[^B]" ;
250 IF J=SPTR THEN PRINT "<-~STACK~POINTER" :
GOTO 270
260 PRINT
270 IF J<7 THEN PRINT "~~~~~~~~~~~~~~[C= Q][^C]
[^C][^C][^C][^C][^C][C= W]"
280 NEXT J
290 PRINT "~~~~~~~~~~~~~~[C= Z][^C][^C][^C][^C]
[^C][^C][C= X]"
300 PRINT
310 RETURN
```

The stack is an integral part of the operation of the system since it is extensively used by the system as a notepad on which to remember important values which are temporarily not required. When a machine code, or indeed a BASIC program jumps to a subroutine, the present position within the program is stored on the stack so that when a return is made from the subroutine, the system knows exactly where to take up the program again.

However, the use of the stack is not limited to the system, since it is used quite extensively in programming. Newcomers to machine code programming often seem to regard the stack as if it were something almost mystical. In fact, it is one of the simplest, even crudest parts of the system. In essence, you should regard the stack as merely a notepad on which to store items of information, up to 256 items in all. The main limitation of the notepad is that the last item of information placed onto it is always the first to be removed, then the next to last and so on. Forgetting this 'last in, first out' principle is one of the major causes of machine code programs crashing. If you were to store values X, Y and Z on the stack temporarily, there is no point in asking for them back in the order X, Y, Z. You must program in such a way that the values are recalled in the order Z, Y, X.

The only real complication in using the stack is that it is shared by the system, which also uses it as a notepad. If you are too careless with your use of the stack then it is quite possible that the CPU, in going about its normal duties, will pick up the wrong values from the stack and the system will crash.

For both these reasons it is essential that when using the stack for your own purposes that you take the simple precaution of *counting* how many items you place on the stack and ensuring that they are removed when you have finished with them — and that no others are removed unless you are quite certain that it is necessary. For instance, we have already mentioned that if you call a subroutine in the course of the machine code program the place at which the program is to continue when the subroutine is ended is recorded on the stack. If, during the course of the subroutine, you add values to the stack which you fail to remove, or if you remove more items than you placed there, when the subroutine ends the system will trustingly pick up the two values from the top of the stack, treat them as the address to return to in the main part of the program and jump to somewhere ludicrous.

This is not meant to make you afraid of using the stack, which is an essential part of the machine code programmer's armoury, — it is simply a warning to ensure that you have done your counting correctly.

Underflow and overflow

One other limitation imposed on your use of the stack is that of its size. We have already noted that the stack is 256 bytes long, at least potentially, but it is important to realise the implications of that limitation. Since the system knows where the stack begins it uses only a single byte register, the stack pointer to record the position of the top of the stack relative to this beginning, and being a single byte it can only record values in the range 0–255. If you were to place a large number of items onto the stack, so that the stack pointer were to exceed 255, what would happen is that the pointer would reset to zero and any further items would begin to be placed at the beginning of the stack. The obvious drawback of this overflow is that items previously stored are being overwritten, with probably fatal consequences for the program.

Parallel to this problem of overflow is underflow. If so many items are pulled from the stack that the stack pointer goes below zero, it will reset to 255 and point to the end of the stack memory. In this case, unintended values will be picked up, once again with serious consequences.

The purpose of this warning is not to discourage you from storing large quantities of data on the stack, since it is very unlikely that you would ever want to, but to be aware that problems can arise unexpectedly. Some quite simple programs, for instance, use a technique called recursion,

where a sub-routine is called and proceeds to call itself. The technique can be extremely useful provided that it is remembered that each sub-routine call adds to the stack and each return from a subroutine subtracts from the stack. It is surprisingly easy to fall off the end in either direction.

Two BASIC programs to illustrate overflow and underflow

The first of these two programs demonstrates overflow by requesting nine values to be placed on the stack. The second program places eight items on the stack and seeks to remove nine, thus demonstrating underflow.

```
10 REM PROGRAM TO SHOW STACK OVERFLOW
20 SPTR = 0
30 FOR I = 0 TO 8
40 GOSUB 170
50 INPUT "INPUT~NUMBER~TO~PUT~ON~STACK:";T(SP
TR)
60 SPTR = (SPTR+1) AND 7
70 NEXT I
80 GOSUB 170
90 FOR I = 0 TO 8
100 INPUT "HIT~RETURN~TO~REMOVE~ITEM~FROM~STA
CK";T$
110 SPTR = (SPTR-1) AND 7
120 GOSUB 170
130 PRINT "ITEM~REMOVED~WAS:" T(SPTR)
140 PRINT
150 NEXT I
160 END
170 REM*******************************
180 REM PRINT STACK
190 REM*******************************
200 PRINT "[CLEAR]"
210 PRINT "~~~~~~~~~~~~~~[C= A][^C][^C][^C][^C]
[^C][^C][C= A]"
220 FOR J = 0 TO 7
230 T$ = RIGHT$("~~~~~~~"+STR$(T(J)),6)
240 PRINT "~~~~~~~~~~~~~~[^B]" T$ "[^B]" ;
250 IF J=SPTR THEN PRINT "-~STACK~POINTER" :
GOTO 270
260 PRINT
270 IF J<7 THEN PRINT "~~~~~~~~~~~~~~[C= Q][^C]
[^C][^C][^C][^C][^C][^C][C= W]"
280 NEXT J
```

```
290 PRINT "~~~~~~~~~~~~~~[C= Z][^C][^C][^C][^C]
[^C][^C][C= X]"
300 PRINT
310 RETURN


10 REM PROGRAM TO SHOW STACK UNDERFLOW
20 SPTR = 0
30 FOR I = 0 TO 6
40 GOSUB 170
50 INPUT "INPUT~NUMBER~TO~PUT~ON~STACK:";T(SP
TR)
60 SPTR = (SPTR+1) AND 7
70 NEXT I
80 GOSUB 170
90 FOR I = 0 TO 7
100 INPUT "HIT~RETURN~TO~REMOVE~ITEM~FROM~STA
CK";T$
110 SPTR = (SPTR-1) AND 7
120 GOSUB 170
130 PRINT "ITEM~REMOVED~WAS:" T(SPTR)
140 PRINT
150 NEXT I
160 END
170 REM*******************************
180 REM PRINT STACK
190 REM*******************************
200 PRINT "[CLEAR]"
210 PRINT "~~~~~~~~~~~~~~[C= A][^C][^C][^C][^C]
[^C][^C][C= S]"
220 FOR J = 0 TO 7
230 T$ = RIGHT$("~~~~~~~"+STR$(T(J)),6)
240 PRINT "~~~~~~~~~~~~~~[^B]" T$ "[^B]" ;
250 IF J=SPTR THEN PRINT "-~STACK~POINTER" :
GOTO 270
260 PRINT
270 IF J<7 THEN PRINT "~~~~~~~~~~~~~~[C= Q][^C]
[^C][^C][^C][^C][^C][C= W]"
280 NEXT J
290 PRINT "~~~~~~~~~~~~~~[C= Z][^C][^C][^C][^C]
[^C][^C][C= X]"
300 PRINT
310 RETURN
```

The Stack and the Accumulator — the PHA and PLA instructions

The first of the two sets of instructions which the 6510 chip provides for accessing the stack are those relating to the accumulator. The PHA instruction is short for PusH Accumulator. The word push is used with most types of chip to indicate that a value is being placed onto the stack. PLA stands for PulL Accumulator — once again pull is almost universally used, regardless of the type of chip, to indicate that a value is being taken off the stack.

The PHA instruction places the current contents of the accumulator onto the top of the stack, and the stack pointer is incremented by 1 so that it points to the new value. The contents of the accumulator are unchanged and there is no effect whatsoever on the various flags.

The PLA instruction takes the value at the top of the stack (the one currently pointed to by the stack pointer) and places it into the accumulator. The stack pointer is decremented by 1 to indicate that the top of the stack is now one value lower. Note that the actual contents of the stack are irrelevant to the position of the top of the stack. The values pulled from the stack are not actually removed, it is simply that the stack pointer moves down and they are now ignored. Subsequent pushes will overwrite them. In contrast to PHA, the PLA instruction does affect the flags in the same way that any other method of loading the accumulator would.

PHA and PLA are used to provide temporary storage for values which are in danger of being corrupted by some task being carried out by the program. It is often the case, for instance, that the values held in the accumulator and the other registers are significant for the main part of the program but that a call to a subroutine will change those values in such a way that they will make no sense when the main program continues after the subroutine is complete. In such circumstances, the PHA would be used to place the contents of the accumulator onto the stack. The contents of the X and Y registers, or any other values which needed to be preserved, would then be transferred to the accumulator in turn and stored on the stack in the same way. When all the values are stored, the subroutine can be called and can be allowed to make free use of all the registers in the confidence that whatever changes are made will have no effect on the stored values on the stack. When the subroutine has ended, the values can be pulled off the stack, remembering that they are in reverse order to that in which they were stored, and transferred from the accumulator to their original homes. In this way, providing that you count correctly the items being pushed onto and pulled off the stack, the limited number of registers available within the CPU can be used for a wide variety of tasks, almost at the same time, without danger of confusion.

Example showing the storage of the accumulator on the stack

```
ADD.   DATA      SOURCE CODE
00                 10 PRT
00                 20 ORG $C030
C030               30 SYM
C030               40 FINISH = $C013
C030   48          50 PHA
C031   4C13C0      60 JMP FINISH
C034               70 END


6000 REM*********************************
6010 REM DATA FOR MACHINE CODE
6020 REM*********************************
6030 DATA 48
```

Example showing the retrieval of the accumulator from the stack

```
ADD.   DATA      SOURCE CODE
00                 10 PRT
00                 20 ORG $C030
C030               30 SYM
C030               40 FINISH = $C013
C030   68          50 PLA
C031   4C13C0      60 JMP FINISH
C034               70 END


6000 REM*********************************
6010 REM DATA FOR MACHINE CODE
6020 REM*********************************
6030 DATA 68
```

Commentary

In both cases the routines work with whatever values happen to be in the accumulator or on the stack at the time. You might like to try altering the first routine so that it picks up a value from memory location $FB, places that value into the accumulator and then pushes it onto the stack.

Storing the contents of the status register with the PHP and PLP instructions
The only problem which remains to be solved when storing items on the stack is that of the processor status register. Thinking about the example of calling a subroutine, given above, the whole procedure is not of much use if the flags cannot be stored along with registers. The other registers or memory locations can be transferred to the accumulator and then to the stack. The status register, however, cannot be transferred to any other register of memory location. Accordingly it is provided with its own special stack instructions, PHP (PusH Processor status register) and PLP (PulL Processor status register). Their effect is to push the current contents of the status register onto the stack, without changing those contents, or to pull the value currently at the top of the stack back into the status register.

PHP and PLP are often used in conjunction with PHA and PLA to save the status register along with the other registers. On other occasions it is used alone. It is common, for instance, to wish to make a number of tests or manipulations on the result of a calculation. The problem is that these manipulations will normally change the flags so that when the second test comes to be made the flags refer not to the original result but to what was done afterwards. In such cases the status register can be pushed onto the stack, a test made of a flag, then the original value pulled back into the status register so that another flag can be tested.

Example of the use of PHP

```
ADD.   DATA      SOURCE CODE
00                 10 PRT
00                 20 ORG $C030
C030               30 SYM
C030               40 FINISH = $C013
C030   08          50 PHP
C031   4C13C0      60 JMP FINISH
C034               70 END

6000 REM*********************************
6010 REM DATA FOR MACHINE CODE
6020 REM*********************************
6030 DATA 08
```

Example of the use of PLP

```
ADD.   DATA      SOURCE CODE
00                 10 PRT
```

```
00                  20 ORG $C030
C030                30 SYM
C030                40 FINISH = $C013
C030  28            50 PLP
C031  4C13C0        60 JMP FINISH
C034                70 END
```

```
6000 REM*********************************
6010 REM DATA FOR MACHINE CODE
6020 REM*********************************
6030 DATA 28
```

Commentary
These demonstrations reveal little other than that the push or pull has occurred.

Addressing modes available with the push and pull instructions
There are no addressing modes, as we would normally interpret them, available with the push and pull instructions. The two locations they refer to, either the accumulator and the stack or the status register and the stack, are built into the instructions themselves, as with the transfer instructions we looked at in Chapter 7. The form of the instructions is as follows:

Push accumulator	: PHA	: 48	: 48
Pull accumulator	: PLA	: 68	: 68
Push status register	: PHP	: 08	: 08
Pull status register	: PLP	: 28	: 28

CHAPTER 17
Busy Doing Nothing

One instruction which needs a brief comment of its own but should cause you little difficulty is NOP, which stands for No OPeration. The essence of NOP is that it causes the CPU to wait for two microseconds, without affecting the contents of any registers or memory locations.

NOP can be a very useful instruction for beginners because it can be used to simplify some machine code routines which time something and, perhaps more importantly, it is a useful aid in debugging programs.

NOP and timing

As you advance in prowess with your machine code programming you will increasingly find yourself either using loops to time actions or using them simply to slow a program down at certain points. In standard machine code games, for instance, there is no point in moving an object across a screen at the maximum speed available since it would hardly be visible, if at all. Very often in such games the commands which create movement are separated by quite large loops which do nothing except slow the program down.

NOP can be useful in these circumstances because it is far easier, for instance, to place two or three NOP instructions inside a loop than it is to use two or more loops to overcome the limitation that a single loop cannot be executed more than 256 times. The two following examples illustrate the point.

Example of a timing loop without NOP instructions

```
ADD.   DATA      SOURCE CODE
00                 10 PRT
00                 20 ORG $C030
C030               30 SYM
C030               40 FINISH = $C013
C030   AD11D0      50 LDA $D011
C033   29EF        60 AND #$EF
C035   8D11D0      70 STA $D011
```

```
C038   A200          80 LDX #$00
C03A   86FB          90 STX $FB
C03C   A000         100 LDY #$00
C03E                110 L000
C03E   88           120 DEY
C03F   D0FD         130 BNE L000
C041   CA           140 DEX
C042   D0FA         150 BNE L000
C044   C6FB         160 DEC $FB
C046   D0F6         170 BNE L000
C048   0910         180 ORA #$10
C04A   8D11D0       190 STA $D011
C04D   4C13C0       200 JMP FINISH
C050                210 END
```

```
6000 REM#********************************
6010 REM DATA FOR MACHINE CODE
6020 REM********************************
6030 DATA AD,11,D0,29,EF,8D,11,D0
6040 DATA A2,00,86,FB,A0,00,88,D0
6050 DATA FD,CA,D0,FA,C6,FB,D0,F6
6060 DATA 09,10,8D,11,D0
```

Commentary
Lines 50–70: Clear a single bit in a register of the chip which controls the screen display, the VIC chip. The effect is to switch the screen off.

Lines 80–100: The X and Y registers and memory location $FB are all loaded with zero, without affecting the accumulator, which contains the contents of the register in the VIC chip.

Lines 110–170: Three nested loops from zero to 255 — a total delay equivalent to a loop executed 4278190080 times. The time involved is somewhere in the region of 1 minute.

Lines 180–190: The lines switch the screen back on.

Example of timing loops using NOP

```
ADD.   DATA       SOURCE CODE
00                  10 PRT
00                  20 ORG $C030
```

```
C030                30 SYM
C030                40 FINISH = $C013
C030  AD11D0        50 LDA $D011
C033  29EF          60 AND #$EF
C035  8D11D0        70 STA $D011
C038  A200          80 LDX #$00
C03A  86FB          90 STX $FB
C03C  A000         100 LDY #$00
C03E               110 L000
C03E  EA           120 NOP
C03F  EA           130 NOP
C040  EA           140 NOP
C041  EA           150 NOP
C042  EA           160 NOP
C043  88           170 DEY
C044  D0F8         180 BNE L000
C046  CA           190 DEX
C047  D0F5         200 BNE L000
C049  C6FB         210 DEC $FB
C04B  D0F1         220 BNE L000
C04D  0910         230 ORA #$10
C04F  8D11D0       240 STA $D011
C052  4C13C0       250 JMP FINISH
C055               260 END
```

```
6000 REM*********************************
6010 REM DATA FOR MACHINE CODE
6020 REM*********************************
6030 DATA AD,11,D0,29,EF,8D,11,D0
6040 DATA A2,00,86,FB,A0,00,EA,EA
6050 DATA EA,EA,EA,88,D0,F8,CA,D0
6060 DATA F5,C6,FB,D0,F1,09,10,8D
6070 DATA 11,D0
```

Commentary

Exactly the same routine except that in the inner loop, five NOP instructions have been added. The effect is to increase the time for which the screen is turned off, to around 4 minutes. To achieve this without the NOPs would have involved the addition of another loop, with all its complications.

NOP and debugging

The other main use of NOP, and one which will probably be more important to you at this stage in your career is in debugging, firstly by allowing you to give a program a more flexible structure, and secondly by allowing individual instructions in a malfunctioning program to be isolated.

a) NOP and the structure of a program: when developing programs it is often useful to separate individual routines with blocks of NOP instructions. The advantage of this is that if a routine needs to be altered so that it becomes longer, the additions will simply wipe out a few NOPs rather than forcing you to move the whole of the rest of the program up in the memory.

b) NOP and debugging: one useful technique when a program is performing in an unexpected way, is to replace commands which you suspect may be causing the trouble with NOP instructions. This is equivalent to placing a REM at the start of a line in BASIC to see whether removing the line solves a programming problem. When using NOP in this way it is important to remember that it is the whole of the instruction which must be replaced, and not simply the opcode.

Format of NOP instruction:

Inherent addressing	: NOP	: EA

Epilogue

At this point you may sit back and award yourself a slightly smug pat on the back. Though not very advanced in the field yet, you have completed your first steps towards becoming a fully fledged machine code programmer.

With the experience you have gained from the routines contained in this book you can start to experiment on your own behalf. The Code Runner utility may well come in useful here, allowing you to construct simple routines of your own without too much fear of them crashing the system. The next step will be to get yourself an assembler and some practical books of machine code programming which will show you how to apply what you have learned to programs which *do* something.

The main thing is that now you know there is nothing to be frightened of, enjoy your programming — that's what microcomputers are for.

INDEX

NOTES

NOTES

NOTES

This is the book for everyone who wants to understand machine code on Commodore's best-selling micro. It sets out to explain from the simplest beginnings what machine code is, the way binary arithmetic works, the meaning of the individual commands and examples of commands working together. It is not a collection of incomprehensible routines, nor is it a stern textbook of tables and techniques.

The book takes you past the first and most difficult barrier to becoming a machine code programmer, namely, understanding what machine code is about and how it is not as frightening as beginners often fear. To do this the authors have packed the book with examples and do-it-yourself experiments, as well as some light-hearted comparisons with the world outside computers.

David Lawrence is the author of a large number of best-selling computer titles, including The Working Commodore 64, The Working C16 and Advanced Programming Techniques on the Commodore 64. He is currently working on a number of titles for the Commodore C16 and Plus 4 computers. Mark England is co-author, with David Lawrence, of the acclaimed Commodore 64 Machine Code Master and Commodore 64 Disk Companion. He is also the author of the successful Mastercode Assembler.

GB £ NET +006.95

ISBN 0-946408-73-4

00695

9 780946 408733

SUNSHINE

ISBN 0 946408 73 4

£6.95 net