Chapter Seven
Introduction to Assembly Language

There is no question that BASIC is easy to learn and easy to use. It is also good enough for most applications. However, it has one major drawback - it is slow. The way to get the maximum performance from any computer is to write programs in its own language - the machine's *assembly language*. Notice that it is *the machine's* assembly language, because it is important to realise that there isn't a single language called 'assembly language'. Each machine, or rather each microprocessor, has its very own language. The BBC Micro uses a 6502 microprocessor, so its assembly language is more properly called '6502 assembly language'.

There is no avoiding the fact that assembly language is more difficult to learn than BASIC. If this were not the case everyone would learn assembly language before BASIC! Even though assembler isn't as easy as BASIC it is very well worth knowing and the BBC Micro makes it as easy as possible by having a built-in assembler. Using this facility you can mix BASIC and assembler very easily. You can write most of any program in BASIC can use assembler whenever BASIC proves to be too slow. This is really the best of both worlds!

This is the first of two chapters dealing with assembly language on the BBC Micro. The main subject of this chapter is the 6502, its assembly language and the BBC Micro's assembler. The next chapter shows how our knowledge of assembler, together with an understanding of the hardware, can be used to advantage. You could say that this chapter aims to teach you assembler and the next aims to teach you how to use assembler.

What makes assembler different?

The statement that BASIC is slow and assembler is fast may have made you wonder why this is so? The main reason for this speed

difference is that BASIC is not a language that the 6502 inside the BBC Micro can obey directly. As already mentioned, the only language that it can obey is 6502 assembly language. The execution of your BASIC program involves the use of another program, called an interpreter. When you type RUN, the interpreter (in ROM 0 - see Chapter One) looks at the first line of your program and carries out your instructions. For example, if your first line reads GOTO 1000 the interpreter first identifies the GOTO and as a result looks for line number 1000. Once it finds line number 1000 (which may involve examining a lot of line numbers) it 'looks' to see what the line tells it to do next. Notice that your BASIC program doesn't directly control what the machine is doing. It controls what the interpreter, an assembly language program, is doing and it is the interpreter that controls the action of the 6502 microprocessor. This is, of course, the reason why BASIC is slower than assembly code.

Assembly language is executed by the 6502 without any intervening programs. If you give an assembly language command such as JMP &2000 (JMP is short for JUMP and so the instruction reads jump to &2000) which is the assembly language equivalent of GOTO then the 6502 carries it out at once. Not only does it carry it out immediately, there is no searching for the specified line number because assembly language doesn't work in terms of line numbers. Whenever an assembly language instruction refers to a position in a program it uses memory addresses. Thus JMP &2000 means jump to (or go to) memory location &2000 (remember the & means that the number following is in hexadecimal) and carry out the instruction that you find stored there. This is an advantage that the 6502 doesn't have to spend time searching for a line number. After all, memory location &2000 is always in the same place, but it does mean that assembly language programs can only be run in the area of memory that they are written for. When you write a BASIC program you make no reference as to where in memory it should 'live' - in fact, it is one of the good things about BASIC that the program's location is irrelevant. Whenever you write GOTO 2000 you don't need to worry about where line 2000 is - the BASIC interpreter looks after you! This use of memory locations typifies assembly language. If you want to store some information then you have to say where you want it stored. In BASIC you simply use a variable and let the interpreter decide where things are stored.

Inside the 6502

The range of instructions that you can give the 6502 in assembly language is very much more limited than the sort of things you can write in BASIC. In particular, you can only handle one memory location at a time. Things are even more restricted than this because before you can do anything to the contents of a memory location, it must be brought ' inside the 6502. This idea is easier to understand once you know that there are a number of special places inside the 6502 called registers where the contents of a memory location can be stored. Registers are the places where all the work gets done. For example, if you want to add the contents of two memory locations together you first have to load a register with the contents of the first location and then issue a command that adds the contents of the second location to the contents of the register. If you want the answer stored in a third memory location then you have to add yet another instruction to store the contents of the register in the memory location. Notice that something that would have been one instruction in BASIC, such as LET A=B+C, has become three instructions in assembly language!

As most 6502 operations involve the use of at least one register it is important to know what registers the 6502 has. To this end a summary can be seen in Figure 7.1. These six registers A, X, Y, PC, S

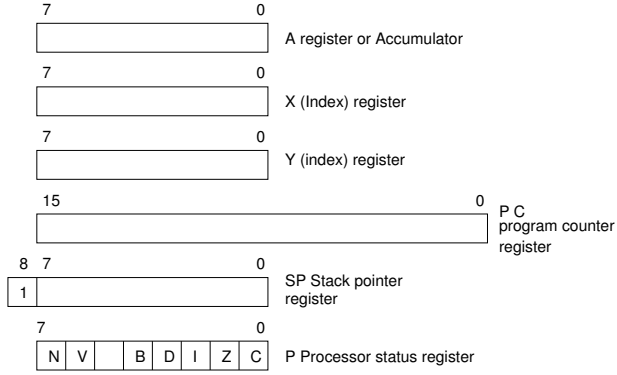

Fig. 7.1. The 6502 registers.

and P are the only registers that the 6502 has and part of the problem of assembly language programming is finding out how to do something useful with so few! As you might have guessed, each register has a different role to play. For example, the A register can be used to do things like arithmetic etc., while the X and Y registers are reserved for other duties. To become a good assembly language programmer you must know not only what registers are inside the 6502 but also what they can be used for.

The only register that need concern us for the moment is the A register. The A register is used for operations such as addition and subtraction. Surprisingly, it is the only register in the 6502 that can be used to carry out calculations of this sort. So if you want to do some arithmetic it is sure to involve the A register. For example, the small addition program that was introduced earlier involves the A register. Coded in 6502 assembly language it is:

```
LDA &2000
ADC &2001
STA &2002
```

The first instruction means 'LoaDA from memory location &2000'. The second instruction is 'ADd with Carry memory location &2001 to the A register' which, for the moment, can be taken to mean 'add the contents of memory location &2001 to the A register'. The final instruction is 'STore the A register in memory location &2002'. This is a complete assembly language program as long as we assume that the two numbers to be added together are already stored in memory locations &2000 and &2001 and that the answer is actually needed in location &2002!

From assembly language to machine code

Although the example given at the end of the last section is a complete assembly language program it is not in a form that the 6502 can use. Humans find instructions written in the form LDA useful because it helps you to remember that the instruction means 'LoaD the A register'. However, as the 6502 cannot read letters, command such as LDA have to be presented in the form of a numeric code. Every instruction that the 6502 obeys can be written in two ways - as a number and as a three-letter *mnemonic code*. The number form is for the 6502 and the letter form is easier for humans to deal with. For example, LDA is the mnemonic code for 'load the A register from a memory location' and its

code is &AD. Before any assembly language program can be carried out by the 6502 it must be changed from mnemonic code form into a list of numbers - *machine code*. This could be done manually by looking up the machine code that corresponds to each mnemonic code in a table of such codes. However, looking that up in such a table, though simple, is such a time-consuming task that it is easier to get a program to do the conversion for us. Such a program is called an *assembler*.

The BBC assembler

The BBC Micro has within it a fairly standard 6502 assembler. This means that you can type in the mnemonic codes of assembly language and have the BBC Micro convert them to machine code. To see this try the following:

```
10 P%=&3000
20 [
30 LDA &2000
40 ADC &2001
50 STA &2002
60 ]
```

The ' [in line 20 indicates that what follows is assembler rather than BASIC. As you might guess, the matching ']in line 60 marks the end of assembler and the start of more BASIC (if any). In general, the BBC Micro treats anything between square brackets as assembler. Lines 30 to 50 should be familiar as they formed the earlier example that added two numbers together. The only unexplained line is line 10. You will recognise P% as one of the *resident integer variables* discussed in Chapter Two. Just like a program in any language, an assembly language program needs to be stored somewhere in memory and the value of P% governs where. Setting P% to &3000 means that the first number that makes up the machine code of the program will be stored in location &3000. If you run the program you should see:

```
>RUN
3000
3000 AD 00 20 LDA &2000
3003 6D 01 20 ADC &2001
3006 8D 02 20 STA &2002
>
```

If you look at the first line of the display following the 3000 you can see AD which is the machine code for LDA. Following this the numbers 00 20 can be recognised as the address of the memory location that A is to be loaded from, but written in the wrong order! Each of the subsequent lines is in the same format, i.e. 6D is the machine code for ADC and once again the address of the memory location follows but in the wrong order. The effect of setting P% to 3000 can also be seen in the column of numbers on the left-hand side. The first number in the machine code, AD is stored in memory location &3000, the second 00 is stored in &3001 and the third &20 in &3002 and so on to the end of the program. If you want to check that it is true type in (in immediate mode):

```
PRINT ~?&3000
```

which will print the contents of location &3000 in hexadecimal so that you can compare it with what you expect to be stored there. Try the same command with different addresses just to confirm that the program has been stored in the memory locations you expect.

It is important to be absolutely clear what has happened in the above program. The opening [told the machine that what followed was assembler and this caused the BBC Micro's assembler to convert the mnemonic codes to machine code and store the numbers in memory starting at the address stored in P%. The closing] switched the machine back to BASIC statements. In this case there were no BASIC statements so the program stopped as you would expect. Notice that there is no mention of 'running' the assembly language program. All that has happened is that it has been converted to machine code and stored in memory starting at &3000. The result of this program can be summarised in terms of getting the assembly language ready to run somewhere in memory. Running the program is a separate step.

Making space

The BBC Micro's RAM is used for all sorts of things - temporary memory storage for the MOS, space for BASIC variables, video storage, and so on. To be able to store machine code safely we have to find an area of memory that isn't going to be used to store anything else afterwards. Storing machine code in areas of memory that are used for other purposes can be disastrous - you can lose control of the machine, the only cure for which is to switch the machine off and on.

In the addition example in the last section, the area of memory starting at &3000 is used to store either lines of BASIC or BASIC variables (see Chapter Two for more information). In this case, as the program is so small and doesn' tuse any variables it is a good bet that the &3000 area is unused. However, if the assembler language in the example was part of a larger program, progress would not be so easy. We obviously need some way of reserving an area of memory that can be used to store machine code. The standard way of reserving memory to store anything is to use a BASIC variable or a BASIC array. BBC BASIC provides a special sort of array, a *byte array*, that can be used to reserve memory for machine code. Try the following program:

```
10 DIM CODE% 10
20 P%=CODE
30 [
40 LDA &2000
50 ADC &2001
60 STA &2002
70 ]
```

Line 10 is a special version of the DIM statement - notice that there are no brackets around the 10. It works in roughly the same way as a normal DIM statement in that it reserves storage in the variables area but there are two important differences. The statement DIM CODE% 10 reserves 11 memory locations and stores the address of the first memory location in the variable CODE%. In general the statement:

DIM numeric veriable size

will reserve *size*+1 memory locations and store the address of the first in *numeric variable*. The variable created by this statement can be used just like any other variable - it can be assigned to or used in arithmetic expressions. In particular, it can be used to set P% to the address of the reserved area of memory. This is exactly what is done in line 20. If you run the program you will see that the assembly language is changed to machine code as in the earlier example but now it is stored in the area of memory reserved by the DIM statement in line 10. The actual location of this area of memory will change as the size of the BASIC program that the assembly language is part of changes but the most important thing is that it will not change or be used for anything else after you type RUN.

There is another way of reserving memory for use by assembly language programs but this is a little more specialised and will be introduced later.

A running program

At this point we know how to use the BBC assembler to translate assembler to machine code and store it in an area of reserved memory. Unfortunately, before we can move onto an assembly language program that can be run we will have to abandon the addition example that has served us so well and write something a little more useful. The trouble with the addition example is that it added together two numbers that were already supposed to be stored in &2000 and &2001 and then stored the answer in &2002. This is not the best way to get information to and from a program!

An interesting example is provided by a program that writes the letter 'A' on the screen - over and over again. In other words we will write the assembly language equivalent of:

```
10 PRINT "A";
20 GOTO 10
```

The first problem we have to solve is to find a way of writing something on the screen. We could use an assembly language version of the peek and poke graphics techniques introduced in Chapter Four but this would be very difficult and long-winded. Instead, we can make use of a subroutine in the MOS that writes characters on the screen. This subroutine is a machine code subroutine that is already in ROM so in this sense we are not cheating - our whole program is still machine code! A description of the subroutine can be found, along with other useful subroutines, in the User Guide. They are, however, worth repeating here.

The OSASCI (Operating System ASCII print) subroutine begins at memory location &FFE3. It will print the character whose ASCII code is stored in the A register on the screen at the cursor's current position and then move the cursor on one place.

To use this subroutine all we need to know is how to load the ASCII code of the character that we want to appear on the screen and also what the assembly language equivalent of GOSUB is. The first problem is solved by using the 6502's *immediate mode*. If you write LDA &02 this is taken to mean 'load the A register with the contents of memory location &0002'. However, if you write LDA #&02 then this is taken to mean 'load the A register with the number 2'. In assembler a

number should be taken to be the address of a memory location unless it is preceded by ' #'Using this information it should be easy for you to work out how to load the A register with the ASCII code for ' A'The answer is LDA #65 (as 65 decimal is the ASCII code for A). The second problem is even easier to solve by using the JSR - Jump to SubRoutine - command, which is almost the exact assembly language equivalent of GOSUB except that it transfers control to an address rather than a line number. While we are on the subject, it is worth mentioning that the assembly language equivalent of RETURN is simply RTS - ReTurn from Subroutine.

We can now begin to write the program:

```
10 DIM CODE% 20
20 P%=CODE%
30 [
40  LDA #65
50  JSR &FFE3
60 ]
```

Lines 10 and 20 are familiar from the previous example. The size of the area reserved is set to 21 bytes which should be more than big enough for the machine code generated. Line 40 is the first line of the assembly language proper and this simply loads the A register with 65 (decimal). Line 50 is the jump to the operating system subroutine. This chunk of program will print the letter A on the screen once. What we want to do, however, is print the letter A repeatedly on the screen. The easiest way to do this is to add JMP (JuMP), the assembly language equivalent of GOTO, to the end of the program. To print the letter A repeatedly we want to JMP back to the instruction that jumps to the subroutine, namely JSR instruction. The problem is how to specify this in the JMP instruction. If you recall the earlier discussion, assembly language instructions work with memory addresses so we need to know the address of the JSR instruction. This is something we could find out by running the above program, noting the address at which the machine code equivalent of JSR is stored. Unfortunately, this won't do us any good because as soon as we add the JMP instruction to the program the area of memory where the program is stored moves, and with it the location of the JSR instruction.

The solution to this problem lies in the use of *labels*. A label is a standard BASIC variable that is used to store the address of a memory location. So, for example, CODE% is a label which indicates the start of the reserved area of memory. You can define a label within an

assembly language program by writing a full stop and then its name. The label then stores the address that the next instruction will be stored in. Once defined, you can use a label anywhere that you can use an address. Putting these two pieces of information together we can write the program as:

```
10 DIM CODE% 10
20 P%=CODE%
30 [
40       LDA #65
50 .LOOP JSR &FFE3
60       JMP LOOP
70 ]
```

Line 50 now defines the label 'LOOP' as the address of the JSR instruction and Line 60 can be read as 'JuMP to LOOP' and will transfer control to the JSR instruction no matter what address it is actually at. To prove that labels are really just BASIC variables try PRINT LOOP after running the program - the number that is printed as the address of the JSR instruction.

Now we have a complete program, the only thing left to do is to run the machine code that results from the assembler. To do this we need to use one of two BASIC statements that transfer control to machine code, USR and CALL.

The easiest to use in this situation is CALL 'address' which will transfer control to the machine code instruction stored at 'address'. If you add:

80 CALL CODE%

and run the program you will at last see the screen filled with letter As! If you try to stop the program you will find that the ESCAPE key has no effect and the only way that you can stop the program is to press BREAK. You shouldn't worry. This is simply a reflection of the fact that machine code isn't as easy to control as BASIC! (You can type OLD to get the program back.)

Addressing modes

Now we have written a working program all that remains to do is expand our knowledge of 6502 instructions. Roughly speaking there are two parts to every instruction - one that gives information about what to do and another than gives information about what do it to. For example, in LDA &2000, the LDA is what to do and the &2000 is where to do it from. Technically, the ways in which you can specify where to carry out an operation are referred to as *addressing modes*. It

is easier to learn the range of addressing modes that the 6502 has and then look at what instructions can be used with what addressing modes rather than to treat everything as a special case. On first reading don' t try to remember all the addressing modes, just familiarise yourself with their names and the ideas involved. Like most things, addressing modes are easier to understand and remember when you actually need to use them.

Absolute addressing
This is the simplest and most used method of addressing. The address of the memory location that oontains the data to be operated on is written following the operation code. For example, LDA &2000 or STA DATA (where DATA is a label).

Zero page addressing
This is a special case of absolute addressing that is used when the address is in the range 0 to 255. Zero page memory locations are special because their addresses can be held in a register or a single memory location. This is because the range 0 to 255 can be represented using only eight bits. However, the address of any other memory location cannot be held in a register and has to be stored using two memory locations because it requires sixteen bits to represent it. There are a number of other addressing modes that only work with the ' help' of page zero memory locations - for example, LDA $05.

Immediate addressing
In this form of addressing, the value to be operated on is written, preceded by #, after the operation code. For example, LDA #20 loads the value 20 into the A register; LDA #DATA loads the value of the label DATA into the A register.

Accumulator and implied addressing
Sometimes an instruction can be applied to either a memory location or to the accumulator. To show the accumulator is the subject of the operation you must write ' A'where an address would normally go. This is known as *accumulator addressing*. In other cases the place where the operation is to be carried out is implied in the instruction. For example, there is no address specified in RTS. This is known as *implied addressing*. Neither are very common nor particularly difficult to understand when they occur.

Relative addressing

This form of addressing tends to look like absolute addressing from the user's point of view. If you are at a particular point in a program you could specify a memory location by saying how many locations above or below the current location it is. This is known as *relative addressing*. The only instructions that use relative addressing are the 'branch' instructions, and the BBC assembler automatically converts absolute addresses to relative when used with branch instructions.

Indexed addressing

This is the most complicated and versatile form of addressing that we have looked at so far. It also involves two register that we haven't used until now - the X and Y registers. An example of an instruction using indexed addressing is LDA &2312,X. The address that the A register is loaded from is given by adding the contents of the X register to the absolute address on the left of the comma. For example, if X contained &50, then A would be loaded from &2321+&50 or &2371. In general, the address used in indexed addressing is obtained by adding the absolute (or zero page) address to the left of the comma to the contents of the register (X or Y) to the right - for example, LDA 34,X STA DATA,Y etc.

The main use for indexing is in 'scanning' through a table of values. The absolute address is the start of the table and the index register is set to the *offset*, that is the distance from the start of the table that you wish to examine. Moving through the table is carried out using INX (INcrement X), INY (INcrement Y), DEX (DEcrement X), and DEY (DEcrement Y).

Indirect addressing

This is a form of addressing that often confuses beginners. However, there is nothing complicated about it. Indirect addressing is usually denoted by enclosing an address in brackets, e.g. (&2320) is an indirect address. The location that an indirect address refers to isn't the location that finally interests us; instead, it contains the address of the location that interests us. For example, (&2320) refers to the memory location &2320 the contents of which are then used as the address of the memory location that will be used in the operation. You can use indirection brackets around any other type of addressing, or indeed any part of any type of addressing. For example, (&34,X) is an instruction

using indexed indirect addressing. To work out the memory location it refers to, you have to work out the address within the brackets and then use the contents of this memory location as the address of the memory location that the instruction refers to. (It is a restriction of the 6502 that this sort of indirect addressing only works if the address in the brackets is a zero page address, i.e. is in the range 0 to 255).

There are restrictions on the way you can use indirection with the 6502. In particular there are only three ways that indirection brackets can be combined with other addressing modes.

1. *Absolute indirect*. This is the basic form of indirection where an absolute address is enclosed in an indirection bracket. The final address is simply the contents of the memory location whose address is between the brackets. There is one complication in that two memory locations are involved in absolute indirection, the one referred to by the absolute address and the next highest. This is because two memory locations are required to hold an address. There is also one major restriction in that absolute indirect addressing can only be used with one 6502 instruction - JMP. Thus JMP (&1234) takes the contents of memory locations &1234 and &1235 and treats them as the address to jump to.

2. *Indexed indirect addressing*. This form of indirect addressing only works using the X register and is written (&32,X). The final address is worked out by adding the zero page absolute address to the contents of the X register and then using the result as the address of a page zero memory location. This page zero memory location together with the one above it hold the address of the memory location that the instruction will use. For example, LDA (&32,X) first adds 32 to the contents of the X register. The resulting number is then used as a zero page address which, together with the next highest memory location, contains the absolute address of the memory location the A register is loaded from.

3. Indirect indexed addressing. This form of indirection is written (&23),Y and can only be used with the Y register. The final address is obtained by using the contents of the zero page address in brackets together with the next higher location as the address that is added to the contents of the Y register. For example, STA (&43),Y takes the contents of locations 43 and 44 and adds the resulting number to the contents of Y to form the address of the location that A is stored in.

You may be puzzled as to what some of the above addressing modes are used for. Don't worry too much because when you need to

use an addressing mode its purpose will become clear! The only addressing modes that you need to understand at this point are: absolute, immediate and indexed.

The 6502's registers

The A register is by now familiar to us from earlier examples and the X and Y registers were introduced in the section above. We therefore already known something about three of the 6502's registers. However, this still leaves the PC, SP and P registers unexplained. For reference purposes, descriptions of each register in turn are given below.

The A register
This register is use for nearly all *data manipulation* - for example, addition and subtraction. It is the register that does most of the work in a program.

The X and Y registers
These registers are called *index registers* because of the role they play in indexed addressing. A good way of thinking about this is that the A register is used for handling data and the X and Y registers are concerned with addresses. However, this is not the whole truth because the X and Y registers can both be loaded from and stored in memory and this sometimes makes them useful for holding temporary results and moving data about when the A register is otherwise occupied. Notice that the X and Y registers are not entirely identical; there are some instructions that can use only the X register for index addressing.

The PC register
The PC or Program Counter register is almost an internal register that is used by the 6502 itself, in that there are no instructions that explicitly make use of the PC register. It used to hold the address of the current machine code instruction, that is, one that the 6502 is obeying. The only instructions which modify the PC register are things like JMP &3432, which causes the PC register to be loaded with &3432 which then, of course, becomes the address of the next instruction to be carried out. It is not very often that a programmer needs to think about the PC register - it takes care of itself.

The SP register

The SP or Stack Pointer is another of the 6502's registers that the programmer needn't worry about too much. The SP register holds an address in a region of memory from 256 to 511. This region is known as the *stack*, hence the name *stack pointer*. There are only two operations that can be applied to the SP register - push and pull. The SP starts with the address of the top of the stack, i.e. 511. A push instruction stores data on the stack at the address that the SP is pointing at and then subtracts one from the SP register so that it is still pointing to an unused memory location. A pull instruction works the opposite way from a push instruction in that it adds one to the SP register and then retrieves data from the memory location that it is pointing at. These are the basic stack operations.

The 6502 itself uses the stack from temporary storage. For example, the JSR instruction stores the return address on the stack and the RTS instruction retrieves it from the stack. The stack can also be used for temporary storage in programs using instructions like PLA - PuLl A - which pulls data off the stack and stores it in A. However, unless you are absolutely sure what you are doing, the stack is best left alone.

The P register

The P or Condition register is different from all the other registers in that it is not used to hold either addresses or data. Instead, it stores information concerning the result of the last instruction executed and the machine in general. Each of the bits in the register can be thought of as *flags* that indicate a different condition. If you look again at Figure 7.1 you will see that each flag is given a single letter name. The B,D and I flags are concerned with the state of the machine. The B flag is 1 following a BRK instruction. The D flag sets the arithmetic mode of the 6502. When it is 0, arithmetic is done in binary and when it is 1 arithmetic is done in a form of decimal known as BCD - Binary Coded Decimal. The I flag is concerned with the servicing of interrupts. If it is a 1, then the 6502 cannot be interrupted. Interrupts are beyond the scope of this introduction to assembly language and my advice is to avoid their use! The flags that are most interesting to the programmer are the N,V,Z and C flags. These are set according to the result of the last instruction. For example, the N flag is 1 if the result of the last instruction was Negative. The V flag is a 1 if the result of the last operation was too big to be stored in a single memory location, i.e. it is

the oVerflow flag. The Z flag is a 1 if the result of the last instruction was exactly Zero. Finally, the C flag is 1 if there was a Carry from the last operation.

The main use of the P register is via the branch group instructions. For example, suppose you want to transfer control somewhere but only when the contents of the A register are zero. This can be done by using BEQ, meaning 'Branch if EQual to zero', which tests the Z flag and jumps to the location only if it is equal to 1, i.e. if the result of the last instruction was zero. Thus the loop:

```
.LOOP DEX
      BEQ EXIT
      JMP LOOP
```

will come to an end when the X register contains zero. There is a simpler way to write this loop using the BNE 'Branch if Not Equal to zero' instruction:

```
.LOOP DEX
      BNE LOOP
```

which will keep on looping until the X register is zero. The P register used in conjunction with the branch instructions is the assembly language equivalent of the BASIC IF statement.

The 6502's instruction set

The only way to learn the instruction set of a microprocessor is to write the programs. However, it is necessary to have a rough idea of the sort of things a micro can do, so the full instruction set is listed with comments in Table 7.1.

Table 7.1. The 6502 instruction set

Mnemonic code	*Brief description*	*Addressing modes*
ADC	Add memory to accumulator with carry.	IMM ABS ZPG (ABS,X) (ABS),Y ZPG,X ABS,X ABS,Y
AND	AND memory with accumulator	IMM ABS ZPG (ABS,X) (ABS),Y ZPG,X ABS,X ABS,Y
ASL	Shift left one bit (memory or accumulator)	ABS ZPG ACC ZPG,X ABS,X
BCC	Branch on carry clear	REL
BCS	Branch on carry set.	REL

BEQ	Branch on result zero.	REL

Mnemonic code	*Brief description*	*Addressing modes*
BIT	Test bits in memory with accumulator	ABS ZPG
BMI	Branch on result minus.	REL
BNE	Branch on result not zero.	REL
BPL	Branch on result plus	.REL
BRK	Force break.	IMP
BVC	Branch on overflow clear.	REL
BVS	Branch on overflow set.	REL
CLC	Clear carry flag.	IMP
CLD	Clear decimal mode.	IMP
CLI	Clear interrupt disable bit.	IMP
CLV	Clear overflow flag.	IMP
CMP	Compare memory and accumulator	IMM ABS ZPG (ABS,X) (ABS),Y ZPG,X ABS,X ABS,Y
CPX	Compare memory and index X.	IMM ABS ZPG
CPY	Compare memory and index Y.	IMM ABS ZPG
DEC	Decrement memory by one.	ABS ZPG ZPG,X ABS,X
DEX	Decrement index X by one	IMP
DEY	Decrement index Y by one	IMP
EOR	Exclusive or memory with accumulator	IMM ABS ZPG (ABS,X) (ABS),Y ZPG,X ABS,X ABS,Y
INC	Increment memory by one	ABS ZPG ZPG,X ABS,X
INX	Increment index X by one.	IMP
INY	Increment index Y by one.	IMP
JMP	Jump to new location	ABS IDR
JSR	Jump to new location saving return addres.	ABS
LDA	Load accumulator with memory.	IMM ABS ZPG (ABS,X) (ABS),Y ZPG,X ABS,X ABS,Y
LDX	Load index X with memory.	IMM ABS ZPG ABS,X ABS,Y
LDY	Load index Y with memory.	IMM ABS ZPG ZPG,X ABS,X
LSR	Shift one bit right (memory or accumulator.	ABS ZPG ACC ZPG,X ABS,X

Mnemonic code	*Brief description*	*Addressing modes*
NOP	No operation.	IMP
ORA	OR memory with accumulator.	IMM ABS ZPG (ABS,X) (ABS),Y ZPG,X ABS,X ABS,Y
PHA	Push accumulator on stack.	IMP
PHP	Push processor status on stack.	IMP
PLA	Pull accumulator from stack.	IMP
PLP	Pull processor status from stack.	IMP
ROL	Rotate one bit left (memory or accumulator).	ABS ZPG ACC ZPG,X ABS,X
ROR	Rotate one bit right (memory or accumulator).	ABS ZPG ACC ZPG,X ABS,X
RTI	Return from interrupt.	IMP
RTS	Return from subroutine.	IMP
SBC	Subtract memory from accumulator with borrow.	IMM ABS ZPG (ABS,X) (ABS),Y ZPG,X ABS,X ABS,Y
SEC	Set carry flag.	IMP
SED	Set decimal mode.	IMP
SEI	Set interrupt disable status.	IMP
STA	Store accumulator in memory.	ABS ZPG (ABS,X) (ABS),Y ZPG,X ABS,X ABS,Y
STX	Store index X in memory.	ABS ZPG ZPG,Y
STY	Store index Y in memory.	ABS ZPG ZPG,X
TAX	Transfer accumulator to index X.	IMP
TAY	Transfer accumulator to index Y.	IMP
TSX	Transfer stack pointer to index X.	IMP
TXA	Transfer index X to accumulator.	IMP
TXS	Transfer index X to stack pointer.	IMP
TYA	Transfer index Y to accumulator.	IMP

Key to addressing modes:

IMM	Immediate	ZPG,X	Zero page, X indexed
ABS	Absolute	ZPG,Y	Zero page, Y indexed
ZPG	Zero page	ABS,X	X indexed
ACC	Accumulator	ABS,Y	X indexed
IMP	Implied	REL	Relative
(ABS,X)	Indexed indirect		
(ABS),Y	Indirect indexed		

Forward references - two-pass assembly

If you try the following short program

```
10 DIM CODE% 10
20 P%=CODE%
30 [
40  LDA #0
50  BEQ EXIT
60  LDA #0
70  .EXIT LDA #0
80 ]
```

you will get an error message. The trouble doesn't lie in the program in the sense that there is nothing wrong with the code. It may not be very useful but it is correct. The trouble comes from the use of the label EXIT in line 50 before it has been defined in line 70. This is exactly the same problem as using a variable before it has been assigned in a BASIC program. The solution, however, is a little more difficult. You cannot move the definition of EXIT before line 50 for obvious reasons (it labels the position of the LDA instruction in line 70!). When you reach line 80 the label EXIT is defined. In fact, when you reach the closing bracket any labels used in the program are defined. If, at this point, the assembler is directed to have another go at converting the program to machine code, there is no longer the problem about labels not being defined. As labels are just standard BASIC variables they retain their definitions until the end of the BASIC program that the assembly language is part of. This use the assembler twice is known as two-pass assembly. To make the assembler examine the same assembly language twice all we need is a simple FOR loop. If you add:

```
15 FOR P=1 TO 2
```

and

```
85 NEXT P
```

then the assembly language will be assembled twice. Unfortunately, this simple method doesn't work because the program still gives an error message and stops when it reaches the first use of EXIT. What we need is some way of saying 'ignore errors and don't bother to produce a listing on the first pass through the program'. After all, the first pass through the program is simply collecting definitions of all the labels so why worry about any errors? If they are real errors they will still be there on the second pass. The assembler contains a command for just this purpose - OPT. There are four possible OPTions which have the following effects:

OPT 0 Ignore errors and don't produce a listing.
OPT 1 Ignore errors but list the program.
OPT 2 Report errors but not listing.
OPT 3 Report errors and produce a listing.

On the first pass we clearly want OPT 0 and on the second pass we want OPT 3. This is easy to arrange. Change the FOR loop in the program to read:

```
15 FOR P=0 TO 3 STEP 3
```

and change line 30 to

```
30 [OPT P
```

This will cure the problem entirely! In general it is usual to use two-pass assembly with a listing produced on the second pass but once you have a working program there is no need to see the listing every time it is run so change OPT 3 to OPT 2. There are plenty of real examples of two-pass assembly in the next chapter.

Mixing BASIC and assembler

There is no problem with mixing BASIC and assembler with the BBC Micro. In fact we have been doing it from the very beginning of this chapter. However, in practice, mixing BASIC and assembler involves passing information between BASIC programs and assembly language programs and this requires a few more details.

There are two ways of calling an assembly language subroutine

from BASIC - the USR function and the CALL statement.

The USR function is the best one to use if the assembly language program needs only a small number of values passed to it and needs to return only a small number of answers. The way that information is passed to the assembly language program is via the resident integer variables. The command USR(address) will transfer control to the machine code starting at the memory location at 'address'. Notice that 'address' can be a constant or a label. For example, USR(CODE%) is allowed. Following the USR statement and before the machine code is actually set running, the contents of A% are stored in the A register and likewise the contents of X% and Y% are stored in the registers of the same name. (In fact it is only the lower byte of these memory locations that is stored in the registers but this makes no difference as long as their contents are in the range 0 to 255). In addition, the C flag in the P register is set to the least significant bit in the variable C%.

This is how information is passed to the machine code. How, then, is it passed back? The answer is that, like all functions, USR returns a single number as its result. This single number is made up of the contents of four of the machine's registers. USR returns a four byte integer and the contents of the registers are stored one to a byte in the order C, Y, X and A. The only problem in making any sense of this result is to separate them out. This is most easily done by using AND. If you want the contents of the A register as the result then, for example, use:

```
ANS=USR(CODE%) AND &00FF
```

which blanks out all the bytes of the answer except the least significant which contains the A register's value. The only thing that hasn't been mentioned is how to get back from assembler to BASIC. Both a USR function or a CALL statement transfer control to the machine code using a JSR instruction, so to return to BASIC simply use an RTS.

The CALL statement is similar to the USR function in that it transfers control back to a machine code subroutine and transfers the contents of the resident integer variables A%,X%,Y% and C% to the contents of the resident integer variables or bits. However, instead of returning a single numerical answer it can return any number of answers of any type. It can also pass any variables to the machine code subroutine. The way that this works is rather complicated in that it involves the setting up of a *parameter block*. This is simply a list of

machine addresses where the variables concerned are stored. Manipulation of BASIC variables from machine code needs a knowledge of how they are stored (see Chapter Two) and it not really within the scope of this chapter. The rules for using CALL with parameters are given in the User Guide and will not be repeated here.

There is an easy way to return limited information from both USR and CALL. The addresses at which the resident integer variables are stored were given in Chapter Two. Using this information, you can store any registers you like in these variables before returning to BASIC. An example of this technique will be found in Chapter Eight.

Conclusion

This chapter has been able to provide only an introduction to assembly language on the BBC Micro. After reading it you should have grasped the fundamental ideas of the instruction set, addressing modes, the registers and the BBC Micro's internal assembler. When you sit down to write your own assembly language programs, however, you will require further detailed information and will need to use a reference book about 6502 assembly language. It is worth repeating that the only way to learn assembly language is to use it. A number of examples is given in the next chapter along with more advanced ways of using the BBC Micro's assembler.