CHAPTER ONE

ASSEMBLY LANGUAGE PROGRAMMING

Assembly code is not much more difficult to learn than BASIC. Machine code is the language the computer really understands underneath all the flashy BASIC commands. The heart of the BBC Micro is a Central Processing Unit (CPU) called the 6502. This CPU does all the 'thinking' and machine code is the language that the chip 'thinks' in.

BASIC is a high level language. This means, simply, that it is far more sophisticated than machine code. For this reason, another chip is needed to interpret the BASIC commands, on a line-by-line basis, into machine code for the 6502 to understand. This chip is a Read Only Memory (ROM) with a machine code program permanently programmed into it. This program is called the BASIC INTERPRETER. This must not be confused with a BASIC COMPILER which takes a BASIC program and converts this entirely into a machine code program so that the original BASIC can be scrapped and the faster machine code used instead.

Unfortunately, because BASIC has to be interpreted, it is very slow. If, however, we could talk to the 6502 directly, in its own terms, we could run programs much, much faster. The assembler is the means we use to talk to the 6502 directly. It effectively by-passes the BASIC interpreter. But before we can learn to use it, it is important to understand the terms BINARY and HEXADECIMAL.

Number Systems

In everyday life we use a number system that we call DECIMAL. This is based on the idea that we count in tens. In decimal we have ten symbols that we use to represent the numbers zero to nine. To represent larger numbers we put these symbols together in a line with the furthest digit right representing the number of ones, the next one to the left representing the number of tens, then hundreds, and so on. So we can look at a number as if it were in a series of columns, each with a column heading saying what it represents.

1000s	100s	10s	1s
5	6	3	1

The computer, however, uses a system based on the idea of having only two symbols to count with − '0' and '1'. Again we use headings, but as the largest number the first column can represent is 1, the second column must count twos, the third fours, the fourth eights, etcetera. In other words, the column headings are powers of two. This counting system is called the binary system.

Binary	Decimal	Binary	Decimal
0	0	110	6
1	1	111	7
10	2	1000	8
11	3	1001	9
100	4	1010	10
101	5		

so 101011 in binary represents

1 times 1 = 1 (first column)
1 times 2 = 2 (second column)
0 times 4 = 0 (third column)
1 times 8 = 8 (fourth column)
0 times 16 = 0 (fifth column)
1 times 32 = 32 (sixth column)

43 in decimal

We call each digit in binary a BIT (BInary digiT).

The bit furthest left is called the MOST SIGNIFICANT BIT (MSB), because it has the largest column heading, and the bit furthest right is called the LEAST SIGNIFICANT BIT (LSB) because it has the smallest column heading. This system is very long-winded but is suits the computer well, as the computer can represent 1 by a circuit being on, and 0 by a circuit being off.

Binary is not an ideal system for humans; for example, if we wanted to represent the decimal number 2141928901 in binary, it would be 11111111010101100111101111000101 − quite an eyeful.

It is convenient to have the computer translate into decimal for us, so we need a counting system, half-way between binary and decimal, which is as easy for us to read as it is for the computer. Such a system is HEXADECIMAL. To convert from binary to hexadecimal (or HEX for short), we split the binary number into groups of four bits (adding a few zeros on the left, if necessary, to make up complete groups of four bits). In each group there are sixteen possible combinations of 0s and 1s so we assign each combination a symbol. Then, by running the symbols together, we have a means of representing the number.

Binary	Hex	Binary	Hex
0000	0	1000	8
0001	1	1001	9
0010	2	1010	A
0011	3	1011	B
0100	4	1100	C
0101	5	1101	D
0110	6	1110	E
0111	7	1111	F

So if we were to take our binary example of 11111111010101100111101111000101 it would be coded like this:

0111	1111	1010	1011	0011	1101	1100	0101
7	F	A	B	3	D	C	5

This may well seem very complicated, but

hexadecimal is like decimal and binary; only, instead of using base 10 or base 2, it uses base 16. In the decimal system, you 'carry' 1 into the next column to the left as soon as you reach ten in one column. In the hexadecimal system, you 'carry' one into the next column to the left as soon as you reach sixteen in one column. Imagine you have two, ten or sixteen fingers and you can't go wrong!

So that we dont get confused between decimal and hexadecimal, we put an & at the beginning of any hex number − thus decimal 2141928901 = &7FAB3DC5

Just in case you aren't confused already, there is yet another system of counting used in computers − BINARY CODED DECIMAL (BCD for short). In this system, each digit of a decimal number is converted into a four-bit binary number. Then the four-bit sections are run together to make a BCD number, e.g.

```
 2    1    4    1    9    2    8    9    0    1
0010 0001 0100 0001 1001 0010 1000 1001 0000 0001
```

Thus 2141928901 in decimal is 0010000101000001100100101000100100000001 in BCD. This number is slightly longer than its binary equivalent, but BCD is occasionally useful. For example, if you have a score in a game, you will want it to appear in decimal. By keeping the score stored in BCD it is then relatively easy to display it on the screen. If it is stored in binary it must first be converted to BCD before being displayed.

Now for a bit of terminology. The processor in the BBC Micro, the 6502, is an eight-bit processor. This means that it works in groups of eight bits at a time. With this system, the computer can only handle unsigned numbers from 00000000 to 11111111 (0 to 255 in decimal). This can be very annoying if you happen to want to use the number 256, but there are ways around this, as we will see later. Each group of eight bits is called a BYTE. Incidentally, each digit in hex is called a nibble because it represents four bits. Four bits are half a byte − think about it!

Maths in binary is, in principle, exactly the same as maths in decimal:

Addition:

```
  109      1101101
 +38      +100110   remembering that 1 + 1 = 10
 ____     ________
  147     10010011
```

Subtraction:

```
  109      1101101
- 38       -100110   remembering to borrow
 ____     _______
  71       1000111
```

However, what happens when we get a negative number? To handle negative numbers, we use a system called TWO'S COMPLIMENT. This method needs a fixed number of bits to each number. In the 6502, we use eight bits or one byte. To make a negative number, we take the positive value in binary and 'flip' each bit (0 becomes 1 and 1 becomes 0). This then has the disadvantage that -0 and +0 will have different codes, so we add 1 to our negative number to make -0 equal to 0.

Examples

-1 is represented by 00000001 flipped to 11111110 and then adding 1 = 11111111.

-0 is represented by 00000000 flipped to 11111111 and then adding 1 = 00000000.

In the second example, when adding 1 to 11111111, we should have got 100000000, but as the computer will only handle eight bits at a time, the ninth bit is lost, leaving 00000000. Let's try another example:

```
 -39    =  11011001     (in two's compliment)
  39    =  00100111

 -39    + 39 =
          11011001
        + 00100111
          ___________
          100000000  but we ignore the ninth bit,
```

So

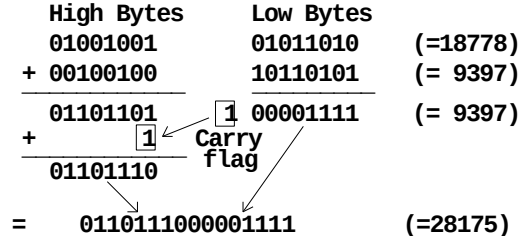
-39 + 39 = 0

With this system, we don't use the most significant bit − the leftmost bit − as part of the number itself, but we use it instead to show whether the number is positive (we set this bit to 0) or negative (we set this bit to 1). This bit is often called the sign bit. Thus the largest number we can store using the eight bits is 01111111, or 127, and the smallest number we can store is 100000000 or -128 (see Appendix A).

Of course, it is not very useful to have a computer which can only count from -128 to 127. However, if we use TWO bytes to store a number, and again use the most significant bit (the leftmost one) to show whether the number is negative or positive, we can count from 1000000000000000 (-32768) to 0111111111111111 (+32767).

However, the computer only provides for the adding of eight-bit numbers, and there could be a ninth-bit 'spill-over' when adding the most significant (leftmost) bits. So we need to use a thing called a CARRY FLAG. This is a single bit in the CPU which can either be set to 1 or cleared to 0. First we add the least significant bytes (or low bytes as they are sometimes called), and if we lose a ninth bit in the answer because of a 'spill-over', the computer sets the carry flag to one to show this; otherwise, the computer clears it to zero. Then we add the two most significant bytes (or high bytes) and if the carry flag has been set to one from the addition of the low bytes then the computer adds one to the result. As we shall see later, the add command does most of this for us.

For example



Thus the answer is 0110111000001111. Conveniently for us, when the computer adds two bytes together, it automatically adds in the carry flag and then puts the ninth bit of the answer, whether 0 or 1, back into the carry flag. This means that, if we want to, we can add three-byte numbers or even larger number to be taken away and flips all its bits. Then clear the carry flag before we add the low bytes. When we have just begun addition we have no carry to consider, so the carry flag must be cleared at the beginning of the addition.

Similarly, large numbers can be subtracted using the carry flag. However, with the subtract command, the carry flag is used as a BORROW flag. Subtracting can be seen as adding the negative of a number; thus 27-5 is the same as 27 + -5.

The subtract command in machine code takes the number to be taken away and flips all its bits. Then it ADDS the two numbers together. Remember that, earlier, we said that we represented a negative number by flipping all its bits and adding one. If, TO BEGIN WITH, the carry flag is set to one, this provides the addition of one needed to complete the negative number in its two's-complement form. Thus the carry flag must be SET to one to produce proper subtraction. As with addition, the carry flag is set to the ninth bit of the result after subtraction. Conveniently, this works out so that if a borrow does occur during this operation, the flag is cleared; otherwise it is set. This means that if we then add a set of high bytes the carry flag will ensure that any necessary borrowing is done.

Let's try, as a simple example, working out 27-5.

27-5 in decimal = 00011011 - 00000101 = 22

The computer takes the 00000101 and flips each bit to make the number 11111010. It then adds the 00011011:

```
   11111010
  +00011011
  ________
1 00010101
```

It then adds the carry ALREADY set to one.

```
100010101
      +1 from carry flag
_________
100010110
```

The ninth bit is transferred to the carry flag leaving us with 00010110 which equals 22 in decimal.

Let's now try a two-byte example by subtracting 9397 from 18778.

The computer first takes the low byte of 9337 (10110101) and flips it to 01001010 then adds the low byte of 18778 (01011010). It also adds one from the carry flag (which we have previously set).

```
 01001010
+01011010
 ________
 10100100
       +1 from carry
 ________
 010100101  this is low byte of
answer.
```

The ninth bit of this answer (0) is transferred to the carry flag. The computer now takes the high byte of 9397 (00100100) and flips it to 11011011. It then adds the high byte of 18778 (01001001). It finally adds the carry flag which is now zero.

```
 11011011
+01001001
 ________
100100100
       +0 from carry flag
 _________
 100100100  this is high byte of
answer.
```

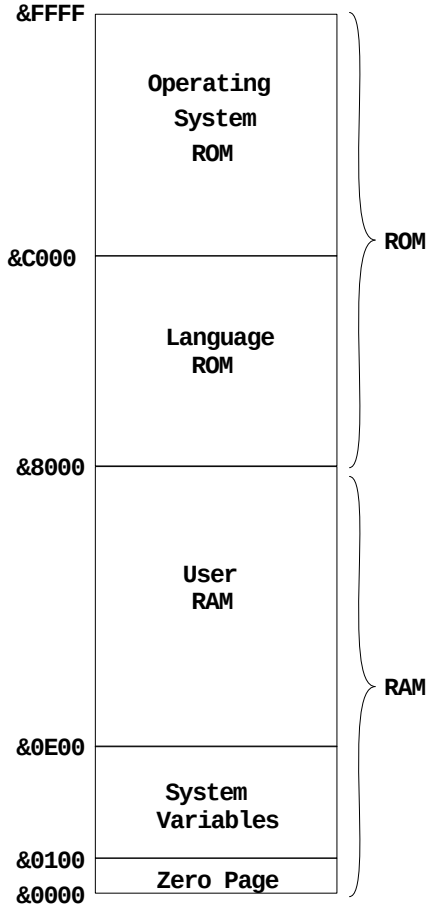
The ninth bit is transferred to the carry flag. The answer is formed from the results of the two additions: 0010010010100101 or 9381 in decimal.

The memory

Let's now look at the memory. The computer does not know what any particular byte in its memory

means. It is up to the programmer to decide, by giving the appropriate instructions, whether a particular byte represents a character of text, a binary number or a BCD number, the low or high byte of a two-byte number; or whatever.

The Model B Memory Map



Each byte of memory is numbered and given an address. An address is a number which, when given to the computer, tells the computer which byte of memory you are dealing with.

The 6502 uses two-byte numbers to represent addresses. The maximum number of different combinations that can be made with sixteen bits is two to the power of sixteen. Thus the 6502 can

handle two to the power of sixteen, or 65536 locations. These locations are each given an address number from 0 to 65535, and no two locations have the same address.

On the BBC Micro, the memory is split into two main sections. These are RAM (addresses 0 to 32767) and ROM (addresses 32768 to 65535). RAM is memory which we can alter and store our programs and data in; ROM is memory which cannot be changed.

The memory is divided up into PAGES. A page is all the memory locations that can be addressed with the same high byte. Thus location &1200 to &12FF make up one page. These pages are numbered according to the high byte of the address; thus &100 to &1FF is page 1. The high byte is 1 and the low byte goes from &00 to &FF which means that a page consists of 256 bytes of memory. Similarly, &0000 to &00FF is ZERO PAGE. Zero page is used a lot for the storage of variables. This is because, as we shall see later, the machine code command for looking at zero page is shorter and faster than the normal commands.

The CPU

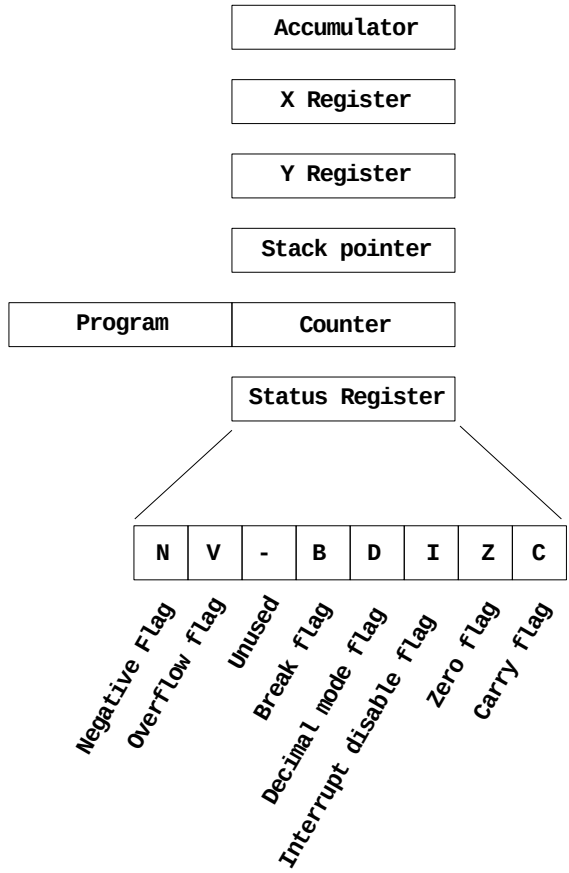
Before we can start to look at commands in machine code, we must look at the way in which the processor is organised.

As far as we are concerned the CPU contains six registers − very simple memories capable of storing one or two bytes. They are inside the processor, not part of the computer's main memory.

The most important one is the ACCUMULATOR, or A register. This register is the one in which all the work is done.

Two other important registers, the X register and the Y register, are very useful for storing extra numbers we may need at any time − these are sometimes called the index registers.

Next is the stack pointer, which points into 256 bytes of memory, located from addresses &100 to &1FF, called the stack. This is a FIRST IN LAST OUT buffer: it behaves like a letter spike − the first thing you put on it is always the last to come off.



Then there is the program counter. In machine code each command is represented by either one two or three bytes of memory. These bytes must be adjacent in memory. The processor looks at the first byte, which is always the command byte and tells the computer what it is to do. The processor then knows (by examining the first, command, byte) how many bytes of data to expect after the command: some commands have no explicit data, some have one byte of data, others have two bytes. The next command must follow on straight after the data of the previous command otherwise the processor will not know where to look for it.

The program counter is a two-byte register which

holds the address of the memory location where the current command is stored. As the program runs, this register is automatically incremented by one every time a new byte of the program is loaded into the processor.

Finally, there is the status register. This register contains eight one-bit flags, one of which is not used. These flags can either be set (equal to 1) or clear (equal to 0) and are affected by some commands. Each shows that a particular circumstance has occurred. We have already met the carry flag which is affected by the 'add' and 'subtract' commands.

Commands

The accumulator is where all the work is done so we need a command to load the contents of a particular byte of memory into the accumulator. This command is called 'load accumulator'. It is stored in memory as a one-byte number. However, to save us from having to remember the number that corresponds to each of the many machine code commands we use a program called an assembler to allow us to type something called an assembler to allow us to type something a little more understandable. It would be very cumbersome to have to type in a command like 'load accumulator' each time we want to use it, so each command in machine code is given a three-letter mnemonic which is easy to remember. The mnemonic for 'LoaD Accumulator' is LDA. So if we wanted to load the accumulator with the contents of memory address &1EFA, we would use the command:

```
LDA &1EFA
```

This command does not alter the contents of the memory, but makes a copy of the byte which is stored at &1EFA and places that in the accumulator. The previous contents of the accumulator are lost, having been replaced by the new byte.

Similarly, we need a command to store the contents of the accumulator in a specific memory location. This command is 'STore Accumulator' − STA. So, to store the contents of the accumulator at

&3F5D, we would use:

```
STA &3F5D
```

This time the contents of the accumulator are not altered, but the previous contents of the memory location (in this case the contents of &3F5D) are lost, having been replaced by the byte from the accumulator.

So now we have two assembly code commands, but we need to know how to use them. They must be used as a program and we need to know how to use the assembler.

BBC BASIC contains a complete assembler which is very easy to use but you have to tell the BASIC interpreter where the assembler commands begin and end, and where in the memory to store the program. The easiest way to store the machine code program is to use the BASIC DIM command. If you use DIM without brackets, it reserves some memory which BASIC will keep totally free. This is ideal for putting machine code routines. All you have to do is this:

```
10 DIM X% 100
```

Note that there are no brackets around the number.

This would reserve 100 bytes of memory. The address of the first of these bytes is put into the variable X%. It is sensible to use an integer variable for this job, as the address will always be an integer. However, the computer still does not know that this address is where we want the machine code to go. The computer will automatically start putting machine code at the location stored in the variable P%. So we ourselves have to set P% to the start of the reserved space, which is already stored in X%. Then we must tell the computer that from now on all commands will be in assembly code. The command to do this is [. So, before we can use the assembler, we must have something like this:

```
10 DIM X% 100  \ Reserve memory and store
                 starting address in X%
```

```
20 P% = X%  \ Tell the assembler to store
              the program at X%
30 [        \ Following code is in assembly
              language
```

Now we can write our assembly language program. Once we have finished writing it, we must put a closing square bracket] to show that we are going back to BASIC. From now on, the shorter examples in this book will ignore the BASIC part of the assembler.

Look at the following program:

```
LDA &2000
CLC
ADC &2001
STA &2000
```

This program uses two commands we have not seen before − CLC and ADC. CLC is the mnemonic for ‘CLear Carry’ and, as its name suggests, it clears the carry flag. Remember that we have to do this before we can add any numbers together. ADC is ‘ADd with Carry’. This commands adds the contents of the memory location specified after it to the accumulator, using the carry flag as explained above. So this program takes the contents of address &2000, adds the contents of address &2001, and stores the result back at address &2000. However, the computer will not know what to do when it has done this, so we have to add another command, RTS, ‘ReTurn from Subroutine’, which tells the processor to go back to what it was doing before. So our small program would look like this:

```
10 DIM X%15
20 P%=X%
30 [
40 LDA &2000
50 CLC
60 ADC &2001
70 STA &2000
80 RTS
90 ]
```

We have dimensioned X% to reserve 15 bytes of

memory. As the longest any command can be is 3 bytes, reserving three bytes for each command should leave ample room for this program. If you don't allow enough room, the program will very likely crash. Another point is that locations &2000 and &2001 are right in the middle of the program memory − if this routine were part of a long program, it would wipe out a part of itself. For most of your programming purposes, the BBC micro conveniently reserves 32 bytes of memory for storing variables. These 32 bytes are at locations &70 to &8F, so it is safest to use this memory where possible.

If we wanted the addition to be done in Binary Coded Decimal, we could preface it by the SED ('SEt Decimal mode') command which sets the decimal flag in the status register. When this flag has been set, all further addition and subtraction is done in BCD. However remember to use the CLD ('CLear Decimal mode') command afterwards to take the computer back into binary mode.

An assembly code program DOES NOT run as it is assembled. The assembler merely ENCODES the program into machine code and stores it away for future reference. To actually run the assembled code, use the CALL command, equivalent to GOSUB; only, you must specify the address of the start of the program rather than giving a line number. The RTS command at the end is the equivalent of the BASIC command RETURN. Here are some other commands:

DEC 'DECrement memory by one' − This subtracts one from the contents of the memory location specified.

INC 'INCrement memory by one' − This adds one to the contents of the memory location specified.

LDX 'LoaD X register from memory' − This copies the contents of a memory location into the X register.

LDY 'LoaD Y register from memory' − This copies the contents of a memory location into the Y register.

STX ‘STore X in memory’ − This copies the contents of the X register into a memory location.
STY ‘STore Y in memory’ − This copies the contents of the Y register into a memory location.
SBC ‘SuBtract memory from accumulator with Carry’.

Remember that the carry flag must be set with SEC (‘SEt Carry flag’) before using SBC, and that, as with ADC, more than one byte may be subtracted.

Addressing modes

So far, we have seen that the command LDA &1EFA loads the accumulator with the contents of the memory location &1EFA. However, LDA can get a byte from the memory IN SEVERAL DIFFERENT WAYS.

Most machine code commands can be used in several different ways, called ADDRESSING MODES, as they are the methods by which the processor finds the address of a byte to work on. The mode we have used up to now is called ABSOLUTE ADDRESSING. However, there are thirteen different addressing modes which we can use, though not all can be used with each command. We have already seen one example of another addressing mode. CLC is an example of IMPLIED addressing. This mode is used in commands that do not need any explicit data to work upon. Other examples:

SEC ‘SEt Carry’
RTS ‘ReTurn from Subroutine’
INX ‘INcrement X register’ − This increases the contents of the X register by 1.
INY ‘INcrement Y register’ − This increases the contents of the Y register by 1.

DEX ‘DEcrement X register’ − This decreases the contents of the X register by 1.
DEY ‘DEcrement Y register’ − This decreases the contents of the Y register by 1.

(Note that if the X register contains &FF and the

command INX is used, the answer should be &100; but, because the X register only has eight bits, the largest number it can hold is &FF. Thus the ninth bit is lost (it is NOT transferred to the carry flag) so the result left in the X register is 0. Similarly, if the X register contains 0 and the DEX command is used, the result is 255. The same is true for INY and DEY.)

NOP 'No OPeration', just waste a tiny bit of time.

TAX 'Transfer Accumulator to X register' − The contents of the accumulator remain the same, X changes.

TAY 'Transfer Accumulator to Y register' − The contents of the accumulator remain the same, Y changes.

TXA 'Transfer X register to Accumulator' − The contents of the X register remain the same, A changes.

TYA 'Transfer Y register to Accumulator' − The contents of the Y register remain the same, A changes.

Notice that in implied addressing, the mnemonic is not followed by any explicit data. The processor knows what the 'implied' data is.

Another useful addressing mode is IMMEDIATE addressing. Here the byte of data actually used for the command to operate on is placed after the command with a 'hash' (#) mark to show that immediate addressing has been used. For example, LDA #&CA would put the number &CA into the accumulator. If you examined the accumulator after using this command you would find &CA (decimal 202) in it.

It is often the case that you want to load the accumulator with the contents of a location whose address you don't actually know explicitly. Say, for example, you wanted to load the accumulator with the contents of the byte at PAGE. (PAGE is a variable that contains the address of the first byte of a BASIC program.) Normally this would be at &E00, but on disc machines it is at &1900. The BBC's assembler allows instructions such as LDA PAGE. This means that when the program is assembled,

the computer will take the address to be whatever PAGE is currently set to. However, once assembled, this cannot be changed. Similarly, complicated expressions can be used as addresses in the assembler, for example LDX PAGE + (A%-1)*2.

Another command that is very useful is JMP – 'JuMP to address'. This is the equivalent of the BASIC command GOTO, but it refers to an address in the memory rather than to a line number. This can be very inconvenient as we don't always know of-hand the address of the command we want to jump to. So the assembler provides another useful system, called LABELS. A label is a variable set to equal the address of a specific command. We precede the command by a full stop, then a variable name (something relevant, e.g. 'start' or 'sounds'), then a space. When the assembler comes across this, it sets the variable to equal the address at which the command is stored. This saves us the immense trouble of calculating the address ourselves. Then we can jump to the right address by simply using the variable name after the JMP command, e.g.

```
.start LDA &2000
       CLC
       ADC &2001
       STA &2000
       JMP start
```

However, though easy to read, it need not be set out like this. The assembler allows this sort of thing:

```
100.start:LDA&2000:CLC:ADC&2001:STA&2000:JMPstart
```

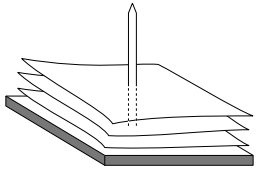
This saves memory and is easier to type in.

The equivalent of the BASIC command GOSUB is JSR ('Jump to SubRoutine') and is used like JMP. The BASIC command CALL, in fact, uses the JSR command to jump to a machine-code routine.

As the processor looks at a command it advances the program counter. Thus, by the time it has looked at a command to see what it has to do, the program counter will point to the beginning of the NEXT

command. When the processor comes across a JSR command, it takes the program counter (which points to the beginning of the next command, remember) and saves it on the stack as two bytes: low first, then high. In doing so, it moves the stack pointer to point to the next byte after the top of the stack. Then, at the end of the subroutine, when the processor meets an RTS command, it takes the top two bytes off the stack again and puts them back into the program counter. Thus the processor returns to the command AFTER the JSR command. As the stack can hold 128 two-byte addresses, the MAXIMUM number of nested subroutines is 128. This decreases if the stack is also being used for other purposes.

It is interesting to note that the stack is stored in page one of the memory (&100 to &1FF) in such a way that &1FF is the first byte of the stack and all subsequent entries are stored in order backwards through the stack. It might help to think of an upside-down paper spike.



It is also often useful to be able to save the contents of the accumulator on the stack and retrieve these contents again later; so two commands are provided to do this.

PHA 'PusH Accumulator onto stack'
PLA 'PuLl Accumulator off stack'

There are also two useful commands for anyone wanting to 'edit' the stack, and these are:

TSX 'Transfer contents of Stack pointer to X register'
TXS 'Transfer X register to Stack pointer'

(Notice that it is the stack pointer not the stack area that is transferred to the X register and vice versa.)

Conditional branches

It is useless to have a language without some form of IF . . . THEN . . . conditional command. This is provided by making use of the flags.

There are four flags which are used for conditional

jumps, or CONDITIONAL BRANCHES as they are called in machine code. These are 'Carry', 'Zero', 'Negative' and 'Overflow'.

The zero flag, as its name suggests, is set when the result of some command is zero.

The negative flag is set if the result is negative − if bit seven is set (bit seven is the most significant bit of a byte); zero is treated as positive.

The overflow flag is more complicated. It is set either if there is a carry from bit 6 to bit 7 but the carry flag is not set; or if there is no carry flag from bit 6 to 7 and the carry flag is set. Strangely enough, there is a command CLV − 'CLear oVerflow' − but no 'set overflow' command. This flag is seldom used.

Once we have a flag set or cleared, we can do a conditional branch to another part of the program depending on the state (whether it is zero or one) of the flag. There are eight commands to do this:

BCC 'Branch if Carry Clear'
BCS 'Branch if Carry Set'
BEQ 'Branch if EQual to zero'
BNE 'Branch if Not Equal to zero'
BMI 'Branch if Minus'
BPL 'Branch if PLus'
BVC 'Branch if oVerflow Clear'
BVS 'Branch if oVerflow Set'

These commands are followed by a one-byte positive or negative two's complement number. This gives the number of bytes to go forward in the program. If it is negative then the processor will go backwards. Thus BEQ &67 would move on &67 bytes if the result were zero, and BNE &FD (which could also be typed as BNE -3) would go back three bytes if the result were not zero. It is difficult to calculate how many bytes to go forward to reach a particular command; so, again, labels may be used. However, because only one byte is used to contain

the OFFSET, the furthest back you can go is 128 bytes (negatively); the furthest forwards is 127 bytes (positively). Thus it is important to keep branches short. If necessary, you can do the following:

```
       BNE skip   \ if not carry on
       JMP equal  \ if so jump to equal
.skip ....        \ carry on
```

Here, the routine EQUAL is a routine we want to jump to if the zero flag is set. This routine is too far away from this section of the program to use a simple branch command. Instead we branch to SKIP if the zero flag is not set. Otherwise we jump to EQUAL as JMP can reach anywhere in memory.

Conditional branches are very useful for delay routines. We have not yet used the X and Y registers much but they are often used for delays. If we want a very short delay, we can load the X register (or the Y register) with 0, then use the command DEX (which subtracts one from the X register and sets the negative and zero flags according to the result that is stored in the X register.) This will leave 255 in the X register. The zero flag will therefore not be set. So if we do a BNE back to the DEX command (labelled as LOOP below), it will branch back. However, this time, the X register will contain 255. The result of all this is that the X register counts down from 0, then 255, all the way down to 0 again. When X finally reaches 0 again the zero flag is set and the BNE command fails to branch so the program carries on to the next command.

```
       LDX #0
.loop  DEX
       BNE loop
```

This produces a delay of about 0.0006 of a second. For a longer delay, we can create a nested loop around this using the Y register:

```
       LDY #0

.Yloop LDX #0
.Xloop DEX
```

```
        BNE Xloop
        DEY
        BNE Yloop
```

This produces a delay of about 0.16 of a second. If we want an even longer delay, we have to press the A register into service as well. We don't have a 'decrement A' command so we have to use SEC and SBC #1:

```
        LDA #0

.Aloop LDY #0
.Yloop LDX #0
.Xloop DEX
        BNE Xloop
        DEY
        BNE Yloop

        SEC
        SBC #1
        BNE Aloop
```

This produces a delay of about 42 seconds which should be enough for most purposes. Different delays can be obtained by changing the values initially loaded into each register.

A problem occurs when labels are used, whether in jumps, subroutines or branches. As an example:

```
        BNE skip
        LDA #0
        STA &70
.skip   DEX
```

The assembler obviously assembles the code in the order in which it is in the program. So, while assembling the above program, it will not yet have defined the variable SKIP when it gets to the command BNE Skip because it will not have command across the label SKIP and so it gives the error 'No such variable'. To stop this happening, the BBC Micro's assembler can be used as a TWO-PASS ASSEMBLER.

In a two-pass assembler, the code is assembled twice. The first time, the assembler ignores any

reference to labels it has not come across, at the same time leaving a space to fill in their values later. By the time it has finished the first pass it has found all the labels and so will have defined all the variables. Then, on the second pass, it assembles everything, including the reference to labels, because it now has all the addresses it needs stored in the variables. The assembler on the BBC Micro does not do this automatically, but it provides a useful command, OPT. This can only be used within the square brackets and is not assembled into machine code. It is thus known as a pseudo-operation. OPT is followed by a number or variable. Here is what the different numbers following OPT do:

0 Do not print assembled code and ignore errors.
1 Print assembled code and ignore errors.
2 Do not print assembled code and take note of errors.
3 Print assembled code and take note of errors.

If we set OPT to 0 or 1 on the first pass, and set it to 2 or 3 on the second pass, the errors that will naturally occur wherever there are 'forward' references to labels will be ignored on the first pass, while the labels are calculated; then any other errors will show up on the second pass. The easiest way to use OPT is by putting a FOR ... NEXT ... loop around the assembly code and setting OPT equal to the loop variable. When the assembler is enabled with the open brackets, OPT is automatically set to 3. Thus the first command within the brackets should be the OPT command. So the BASIC code to precede a section of assembly code now becomes:

```
10 DIM mc% 100
20 FOR pass% = 0 TO 3 STEP 3
30 P% = mc%
40 [OPTpass%   \ On first pass, OPT is set
          .           \ to 0, so ignore all
errors.
        .        \ On second pass, OPT is set
```

```
        .           \ to 3, so report any errors
    assembly code   \ and print assembled code.
        .
        .
        .
140 ]
150 NEXT
160 CALL mc%
```

OPT is not needed if all the branches, jumps and so on refer to EARLIER parts of the program − that is, if the program does not have any 'forward' references to labels. Note that P% must be reset to mc% at the beginning of each pass, because P% is incremented by the assembler as it assembles the code. This is sometimes useful as, after the] command, P% will point to the first free byte of memory after the machine code. For example, if you wish to save a piece of assembled code directly, using *SAVE, P% will, after the code has been assembled, give the end address of the code.

A command that goes with the branch command:

CMP 'CoMPare memory with accumulator'

This needs a little explanation. CMP subtracts the specified byte from the accumulator and sets the carry, zero and negative flags in the normal way. However, it doesn't store the result in the accumulator as SBC does, so the accumulator is not altered. In fact the result is lost completely; all that is altered are the flags. The contents of the memory location are not altered either. Also, this command automatically sets the carry flag at the beginning, so you don't have to worry about that.

The result of all this is that if the two numbers are equal, then taking one from the other leaves zero and the zero flag is set. If the accumulator is greater than the memory byte, then the carry will be set. The negative flag will be set equal to bit 7 of the result of the subtraction.

Probably the most common use of this command is for checking if the accumulator is equal to a particular value. You can put as many of these tests one after the other as you like because the accumulator

is not affected! This could be very useful if, for instance, you are checking the GET command for the keys in a game − you could use a routine like this:

```
CMP # ASC("Z")
BEQ left
CMP # ASC("X")
BEQ right
```

There are also two equivalent commands for the index registers:

CPX 'Compare memory with X register'
CPY 'Compare memory with Y register'

Two other useful commands are:

PHP 'PusH Processor status register' − Put contents of status register on stack.
PLP 'PulL Processor status register' − Pull byte from stack and place in status register.

These push and pull the current flag states to and from the stack and can be used to 'save' the results of a CMP command.

An addressing mode that is used a lot in the BBC Micro is INDIRECT ADDRESSING. There is only one command that uses this mode, and that is JMP. Instead of jumping to the address specified after JMP, the CPU takes the byte contained at that address and the byte immediately following that and uses these two bytes as the low and high bytes respectively of the address the computer actually jumps to.

For example, there is a very useful subroutine stored in the Operating System ROM called OSWRCH (short for 'Operating System WRite Character'). This is the routine BASIC uses for its VDU command (remember when using this that, as with VDU, a carriage return does not also produce a line feed, i.e. VDU13 returns to the beginning of the same line). Thus if we wished to print the letter 'A' from the assembler, we would load the accumulator

with the ASCII code for 'A' and then JSR to location &FFEE (where the routine starts). However, at this location there is an indirect jump instruction with the address &20E. The computer then jumps to the address pointed to by locations &20E and &20F (&20E contains the low byte and &20F the high byte) − and this is where the actual routine starts. This system is sometimes called VECTORING. In this example, locations &20E and &20F make up a vector. As &20E and &20F are in RAM, we can change their contents so that, for example, the VDU command will jump to our own subroutine instead of the Operating System routine through our altering &20E and &20F to point to our own routine. This JMP command is typed with the address in brackets, e.g.

```
JMP (&20E)
```

One thing must be kept in mind when using this command. When the computer adds one to the address to fetch the high byte, it does not carry into the high byte of the address. Thus if we were to use the command JMP (&13FF), this would jump to the address stored at &13FF (low byte) and &1300 (high byte)! (&FF is incremented to &100 but the 1 isn't carried over to the high byte and so the result is &1300.) As an example of how to use the OSWRCH vector, the following program makes the computer always print a full stop instead of a space.

```
 10 DIM X% 100
 20 FOR A%=0 TO3 STEP3
 30   P%=X%
 40   [
 50   OPTA%
 60  .start CMP #ASC(" ") \ Check if char is
                            space
 70         BNE print     \ if not, print it
 80         LDA #ASC(".") \ if so, load full
                            stop
 90  .print JMP (&230)    \ then go to main
                            routine
100   ]
110   NEXT
120 ?&230=?&20E       :REM Copy vector into
```

```
130 ?&231=?&20F       :REM &230 and &231
140 ?&20E= start MOD256   :REM then set to
150 ?&20F= start DIV256   :REM new routine.
```

The \ symbol is equivalent to the BASIC REM statement − the computer ignores everything else on that line.

Lines 120 and 130 make a copy of the VDU vector in a spare vector (&230) that the operating system does not use, so that we can jump to the original OSWRCH routine to print a character.

This program can be very useful for checking for spaces accidentally typed at the end of lines, but remember to press BREAK to clear it before attempting to edit lines, otherwise the spaces you really want will become full stops.

Although there are no proper multiplication and division commands in machine code, there are two commands for multiplication and division by 2:

ASL ‘Arithmetic Shift Left memory’
LSR ‘Logical Shift Right memory’

Their names don’t give much of a clue as to how they work. ASL takes all the bits in a byte and shifts them one place to the left. The least significant bit (the rightmost one) becomes zero and the most significant bit is placed in the carry flag. The original contents of the carry flag are thus lost.

For example:

```
C  7 6 5 4 3 2 1 0
0  1 1 0 1 0 1 1 1
```

becomes, after ASL

```
C  7 6 5 4 3 2 1 0
1  1 0 1 0 1 1 1 0
```

The result of this is to multiply the byte by two.

LSR does the opposite. It shifts all the bits to the right, sets the most significant bit to zero and shifts the least significant bit into the carry. Again, the

original contents of the carry are lost, e.g.

```
7 6 5 4 3 2 1 0  C
1 1 0 1 0 1 1 1  0
```

becomes, after LSR

```
7 6 5 4 3 2 1 0  C
0 1 1 0 1 0 1 1  1
```

The result of this is to divide the byte by two.

Apart from absolute addressing, these two commands can be used with another addressing mode called ACCUMULATOR addressing. As its name implies, in this mode the byte used and affected is the one contained in the accumulator and not in a memory location. This mode is used from the assembler by placing the letter A after the command.

```
ASL A
LSR A
```

(The spaces are not necessary.)

The remaining two commands that can use accumulator addressing mode are:

ROL ‘ROtate Left memory’
ROR ‘ROtate Right memory’

These commands are similar to ASL and LSR but they do not lose the original carry contents entirely. ROL shifts the byte left. As it does so the old contents of the carry are shifted into bit zero and the old contents of bit 7 are shifted into the carry. Thus the whole byte plus the carry flag − nine bits in total − is rotated one bit to the left, e.g.

```
C  7 6 5 4 3 2 1 0
1  0 0 1 0 1 1 1 0
```

becomes, after ROL

```
C  7 6 5 4 3 2 1 0
0  0 1 0 1 1 1 0 1
```

ROR is the exact opposite, e.g.

```
7 6 5 4 3 2 1 0  C
0 0 1 0 1 1 1 0  1
```

becomes, after ROR

```
7 6 5 4 3 2 1 0  C
1 0 0 1 0 1 1 1  0
```

By combining ASL and ROL commands, a number of two or more bytes can be multiplied by two.

1. The least significant byte of the number is multiplied using ASL. This leaves its most significant but in the carry.
2. The next byte up is multiplied with the ROL command. The carry resulting from step 1 is thus now placed in bit zero of the second byte and bit seven of the second byte is left in the carry, ready for another ROL command on the next byte up (if any).

For example,

```
     High byte        Low byte
     00110101         10101101   =13741 (decimal)
        V                V
       ROL              ASL
        V                V
C                C
0 <  01101011  < 1  < 01011010  =27482 (decimal)
```

The original number is doubled.

Similarly, numbers comprising two or more bytes can be divided by two using LSR on the high byte first then ROR on successive lower bytes, e.g.

```
  High byte Low byte
  00110101 10101101          =27482 (decimal)
     V        V
    LSR      ROR
     V        V
              C               C
```

01101011 > 1> 10101101> 0 = 13741 (decimal)

The original number is halved.

If the original number is odd, this method will lose the 'half' at the end of the answer into the carry flag, i.e. the carry flag contains the remainder after division by two. This can be used to test if a number is even or odd by testing the carry flag.

As an example of what can be done with these four commands, let's write a short program that will multiply the contents of the accumulator by 10 and store the answer at &70 (low) and &71 (high).

To do this we must first multiply by five and then multiply by two. To multiply by five we can multiply the number by four and then add the original number to make five lots of the original number.

The first thing to do is this:

```
.mten   STA &70
        LDX #0
        STX &71
```

The original number is now stored as a two-byte number in &70 (low) and &71 (high). We will need to add the original number to the answer later, so we can leave the original number in the accumulator. Next, we can multiply the contents of &70 and &71 by 4 by multiplying them by 2 twice.

```
        ASL &70
        ROL &71
        ASL &70
        ROL &71
```

We now have four times the original number in &70 and &71. To make five times the original number we must add the accumulator (which STILL contains the original number) to &70 and &71.

```
        CLC
        ADC &70
        STA &70
        BCC skip
```

```
        INC &71
.skip   ...
```

Note that because we are adding zero to the high byte, it is quicker to use a branch and an INC command than to use a further ADC command.

We now have give times the original number in &70 and &71. Finally, we need to multiply this by two to get 10 times the original number.

```
.skip   ASL &70
        ROL &71
        RTS
```

The index registers

The index (X and Y) registers are used a lot for what is known as ABSOLUTE INDEXED addressing. This mode is similar to absolute addressing, but an X or Y is placed after the address (separated from it by a comma). The contents of the register are added to the address to form the actual address from which a byte is fetched or saved. This is very useful for arrays where the index register can be used to point into a one-dimensional array up to 256 elements long. Here is an example of this:

```
 10 DIM mc% 30
 20 FOR pass% = 0 TO 3 STEP 3
 30 P% = mc%
 40 [OPT mc%
 50 .start LDX #0
 60 .loop  LDA array ,X
 70        JSR &FFE3
 80        INX
 90        CPX #8
100        BNE loop
110        RTS
120 .array EQUS CHR$13+"Hello."+CHR$13
130 ]
140 NEXT
150 CALL start
```

The command EQUS is one of four similar commands available only with BASIC II which are for storing data in the middle of assembly code.

The first is EQUB. This must be followed by a one-byte number. This byte is then inserted into the

middle of the assembly code. Make sure that the processor will never try to run this byte as an instruction − if it did, it would almost certainly crash. This byte can then be used as a byte of DATA.

For example, if you have run out of room in the variable space from &70 to &8F, you could put a series of EQUB commands after the main program each preceded by a label and then use the labels as free memory locations.

The second command is EQUW which is identical except that it uses a two-byte number. This number is stored low-byte first.

The third is EQUB which uses a four-byte number. This is stored low-byte first.

The fourth is EQUS which uses a string. This is stored in the order the characters appear in the string.

In the program above, the string 'Hello.' between two carriage returns is placed at the label ARRAY and can then be referred to by the program, a byte at a time, using the X register to point into the string.

For those of you with BASIC I these four commands are not available. To get around this you will need to replace them with these pieces of code:

```
100 .temp    EQUB &CA
```

is replaced by:

```
100 .temp ]
102 ?P%=&CA
104 P%=P%+1
106 [OPTpass%
```

Because P% always points to the beginning of the next machine code command we can exit the assembler and place the correct byte in the memory in the correct place. We then need to increment P% to point to the next byte. Finally we can re-enter the assembler (remembering to set OPT as this is always reset to 3 on entry to the assembler).

Likewise for the other three commands:

```
100 .temp    EQUW &1CA3
```

is replaced by:

```
100 .temp ]
102 ?P%=&A3:P%?1=&1C
104 P%=P%+2
106 [OPTpass%
```

and

```
100 .temp    EQUD &12345678
```

is replaced by

```
100 .temp ]
102 !P%=&12345678
104 P%=P%+4
106 [OPTpass%
```

```
100 .temp    EQUS "HELLO"
```

is replaced by:

```
100 .temp ]
102 $P%="HELLO"
104 P%=P%+LEN"HELLO"
106 [OPTpass%
```

The subroutine at &FFE3 is a ready-made operating system routine. It checks whether the accumulator contains a carriage return and, if so, prints both a carriage return and a line feed; if not, it jumps to OSWRCH which prints the character in the accumulator. This routine is called OSASCI.

Another example of a simple use of the index registers is for filing a block of memory locations. The next example clears a Mode 7 screen. By changing the character the program uses, the screen may be filled with any charater.

```
10 MODE 7
20 DIM mc% 30
30 FOR A% = 0 TO 3 STEP 3
```

```
 40 P% = mc%
 50 [OPTA%
 60 .clear LDA #ASC(" ")
 70        LDX #0
 80 .loop  STA &7C00,X
 90        STA &7D00,X
100        STA &7E00,X
110        STA &7F00,X
120        DEX
130        BNE loop
140        RTS
150 ]
160 NEXT
170 CALL clear
```

In this program, because the screen takes up 1K (which is 4 pages) starting at &7C00, one absolute indexed X command can only clear one quarter of the screen, as the X register can only go from 0 to 255; so four commands are used, one for each page. Of course, this whole program could be replaced by a call to OSWRCH with the accumulator containing 12. (This is the ASCII code for 'Clear Screen'.)

Logical commands

A very useful command is:

ORA 'OR Accumulator with memory'

This works on each bit of the two bytes (one in the accumulator, the other from the memory) separately. If either of the corresponding bits in the two bytes is 1 or both are 1, then the corresponding bit in the answer is 1. If, however, they are both 0, then the answer will be 0. The result goes back into the corresponding bit of the accumulator. This OR function can be used to set particular bits in a byte to 1 and leave the others untouched.

first bit	second bit	answer bit
0	0	0
0	1	1
1	0	1
1	1	1

E.g.

```
01010110

   OR

00011111
________
01011111
```

A similar command is

AND ‘AND accumulator with memory’

This command only sets the bit in the answer to 1 if the corresponding bits in the first byte AND the second byte are BOTH 1. The AND function can be used to set particular bits in a byte to 0 leaving the others untouched. Both this and the OR command are sometimes called MASK commands because they mask particular bits to 1 or 0.

first bit	second bit	answer bit
0	0	0
0	1	0
1	0	0
1	1	1

E.g.

```
01010110

   AND

11110000
________
01010000
```

The third and last command along these lines is the EOR or Exclusive-OR command. This sets the answer bit to 1 if one of the corresponding bits in the first byte OR the second byte is one, but not if they are both one or both 0. This command can be used to flip particular bits in a byte from 1 to 0 or 0 to 1 leaving the others untouched. This is how the processor flips all the bits to make a negative number when doing a subtract command.

first bit	second bit	answer bit
0	0	0
0	1	1
1	0	1
1	1	0

E.g.

```
01010110

  EOR

00001111
________
01011001

01010110

  EOR

11111111
________
10101001
```

Notice that, as using EOR with 255 flips all the bits, a second use of EOR with 255 will flip them all back again leaving the original number. This is very useful in games graphics. A figure in a game can be Exclusive-OR'ed with the screen to put it on and then Exclusive-OR'ed again to remove it.

BIT does the same as AND but doesn't place the answer back in the accumulator. It forgets the answer, but it does set the status register flags. If the answer is zero then the zero flag is set (otherwise it is cleared). Also, bits 6 and 7 (the most significant-but-one and the most significant) of the byte taken from memory are placed in the overflow and negative flags respectively. This can be used in a variety of ways. By using BIT on a byte in the memory, bits 6 and 7 can be tested using the flags. But its main use is that, by setting just one particular bit of one of the numbers to 1, leaving the others at zero, and using BIT on the two numbers, the zero flag will then indicate whether the corresponding bit in the other number is 0 or 1.

E.g.

```
00010000  (in accumulator)

  BIT

01010101  (in memory)
________
00010000  which is not zero, so the zero
          flag will be clear

00010000  (in accumulator)

   BIT

10100101  (in memory)
________
00000000  which is zero, so the zero
          flag will be set
```

Indexed indirect addressing

The four remaining machine code commands are concerned with INTERRUPTS; they are covered in chapter four. There are still, however, two addressing modes to be covered.

The first of these is POST-INDEXED INDIRECT ADDRESSING. In this mode, a one-byte number is specified after the command, and this refers to an address in zero page. The processor takes the byte at this address and the byte at the address after it to form a new address. As in indirect addressing, the low byte is stored first followed by the high byte. Also, if the address specified is &FF, the two bytes used will be the ones at &FF (low byte) and &00 (high byte). Once the processor has this address, it adds the contents of the Y register (this mode can only be used with the Y register) to it. This now forms the address of the byte the processor actually works on. Within the assembler, the zero page address is put in brackets with a comma and a Y after the brackets e.g.

```
LDA (&70),Y
```

If location &70 contains, say, &00 and location &71 contains &30, the address 'pointed to' would be &3000. If the command is used with Y ranging from 0 to 5, then locations &3000, &3001, &3002, &3003, &3004 and &3005 would be successively addressed.

This mode uses just two zero page locations to hold a variable that itself points into the memory. For example, if we wished to clear a Mode 0 screen, we place the address of the beginning of the screen in, say &70 and &71. Then we could use a loop of Y to store zeros at 256 locations starting with that address, then add 256 to the address and repeat the procedure until the screen is fully cleared, e.g.

```
.clear  LDA #0        \ Place start of
        STA &70       \ screen address
        LDA #&30      \ in &70 (low)
        STA &71       \ and &71 (high)
        LDA #0        \ Fill with 0
        LDY #0        \ start Y at 0
.loop   STA (&70),Y   \ store on screen
        INY           \ next byte
        BNE loop      \ 'til 256 done
        INC &71       \ then next 256
        BIT &71       \ check bit 7 of &71
        BPL loop      \ if clear repeat
        RTS           \ if set, then &8000
                        reached therefore
                        screen
                        cleared so end.
```

The last addressing mode is PRE-INDEXED INDIRECT ADDRESSING. It uses the X register only. The X register is added to the zero page address itself rather than to the address stored at that zero page address. In other words, the processor takes the one-byte address after the command, adds the X register (it ignores any carry so that the result is still a zero page address). Then it takes the byte at that address and the byte at the address after it to form a two-byte address (as before, low then high) and this is the address of the byte that it uses. This command is typed in the assembler like this:

```
STA (&70,X)
```

If X is 5, this instruction would form the actual address from the contents of &75 (low) and &76 (high).

Note that the X is WITHIN the brackets this time. The use of this is that a table of addresses can be formed in zero page and perhaps contain the addresses of a series of missiles on the screen in a game. However, because only limited zero page locations are usually available to you as a machine code programmer, the application for this mode on the BBC Micro are very limited.

There are some commands that people assume can be used but which the computer does not allow. For instance, there is a command ORA for the accumulator, but there are no equivalent commands for the X and Y registers. Another point is that not all commands support all addressing modes. For instance, the command INC A is not valid − to do this you will have to use the commands:

```
CLC
ADC #1
```