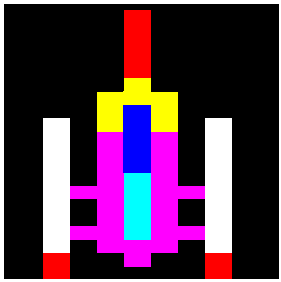


3

JUMPS, BRANCHES AND LOOPS

When an assembler program has been assembled and is being executed, the address of the next instruction to be executed is kept in a register called the 'program counter'. All the programs met so far have been executed in the order that the instructions were written, so the program counter has just steadily increased until it reached the last instruction. This chapter introduces the **jump** and **branch** instructions which can make the program counter jump over instructions or move back to previous ones to execute them again. These instructions make it possible to implement **loops** and perform different instructions depending on the outcome of previous ones.

3.1 Jumps

Ordinary jumps

Mnemonic	Description
JMP	**jump** to instruction whose address is given

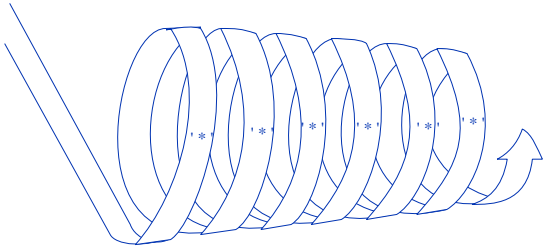
The JMP instruction is followed by the address of the instruction to be executed next, e.g.

```
JMP &E48     or     JMP addr
```

Instead of describing the address by a number, we can use a 'label' to indicate to the assembler where we want to go. In the assembler, labels are variables prefixed with a full stop (.).

```
 10 oswrch =&FFEE
 20 DIM P% 100
 30[
 40.enter
 50 LDA #ASC"*"
 60.Loop
 70 JSR oswrch
 80 JMP Loop
 90]
100 END
```

When the program is assembled the address corresponding to the label '.loop' will be inserted in the machine code. When the code is executed the value of the program counter will be set to this address and the CPU will collect its next instruction from the location with that address and will continue executing from there.

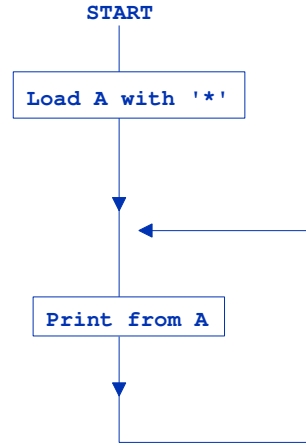


The label '.enter' at the start of the program has been included so that this label can be called in order to execute the program. This is a better way of executing a program than calling TOP since TOP doesn't always point to the first machine code instruction. This is true for the above example since TOP will point to the assignment statement 'oswrch=&FFEE'.

The program will output an asterisk (*), and then jump back to the previous instruction. The program has become stuck in an endless loop! Compare this program with the following BASIC program:

```
10 A = ASC"*"
20 VDU A
30 GOTO 20
```

A flowchart for this program is as follows:



To get out of the BASIC loop you press ESCAPE. This will not automatically halt machine code programs, however. To exit from a machine code loop without losing the program you must press BREAK, and then type ' OLD' to retrieve the original program.

Jumps to subroutines

Mnemonic	**Description**
JSR	jump to subroutine

Examples of this instruction have been used previously. Like the JMP instruction it is followed by a two-byte address, e.g.

```
JSR oswrch
```

In this case the address of the instruction directly following the JSR instruction in the code is noted, and then the value of the program counter is set to the address of ' oswrch'The CPU will go to this address for its next instruction and start executing the code from there until it meets an RTS. This will set the program counter to the address which was noted earlier so that the CPU can then continue executing the code following the JSR instruction. Subroutines jumped to can either be part of the assembler program or, as in this example, sub-routines which exist in the operating system memory.

3.2 The zero and negative flags

There are several flags in the CPU which can be set or cleared depending on the outcome of certain instructions. The carry flag was introduced in the previous chapter, this is set or cleared as the result of an ADC (add with carry) instruction. Another very useful one is the zero flag, called Z. This is set if the result of the previous operation gave zero, and is cleared otherwise, e.g.

```
LDA &80
```

would set the zero flag if the contents of &80 were zero.

Similarly the negative flag, N, is set if the result of the previous operation was negative in two' s complement notation, i.e. if the top bit was set, e.g.

```
LDA &80
```

would set the negative flag if the number stored in location &80 was greater than 127 (01111111).

The conditions of all the flags are stored in a byte called the status register (P), and each flag is represented by one bit: e.g. the top bit of the status register is set if N=1 and the bottom bit is set if C=I.

3.3 Conditional branches

Conditional branches enable the program to act on the outcome of an operation. There are eight different branch instructions, six of which are introduced.

Mnemonic	Description	Status
BEQ	**b**ranch if **eq**ual to zero	(ie Z=1)
BNE	**b**ranch if **n**ot **e**qual to zero	(ie Z=0)
BCC	**b**ranch if **c**arry **c**lear	(ie C=0)
BCS	**b**ranch if **c**arry **s**et	(ie C=1)
BPL	**b**ranch if **pl**us	(ie N=0)
BMI	**b**ranch if **mi**nus	(ie N=1)

The conditional branch instructions test the state of the various condition flags, e.g. the zero flag and negative flag. If the condition is not satisfied then it carries on executing, but if the condition is satisfied then the computer goes to the place indicated by the byte following the branch op-code. This byte is stored as a relative address, thus if you say

```
BCS notzero
```

the assembler works out the difference (in bytes) between the current instruction and the place where the label ' .notzerois, and puts this value after the op-code. This means that the value of this byte is used, in conjunction with the address of the current instruction, to tell the CPU where to go next.

Because only a single byte is allowed in this relative addressing mode, the branch instructions can only point to one of 255 nearby bytes. The two' s complement representation of numbers is used to give the offset relative to the current address. Branches which point forwards are restricted to 0-127 bytes beyond the current location. The value of the byte following the op-code for these is then 0-127. Branches which point backwards to places at lower addresses in memory require a negative value to be added to the current location. These use the numbers 128-255 to represent the values -128 to -1.

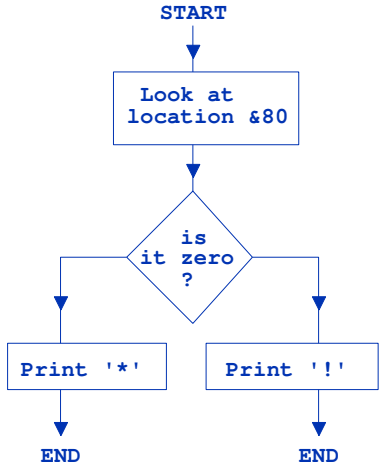
The JMP instruction does not use relative addressing; it is followed by two bytes which specify the absolute address which will be the destination. Hence the branch instruction is shorter than the jump instruction, the jump being three bytes long (op-code and two-byte address) and the branch being two bytes long (op-code and one-byte offset). This difference is automatically looked after by the assembler.

The following simple program will print an

exclamation mark if ' character'contains zero, and a star if it does not. The comments to the right of the assembler statements may be omitted when you enter the program.

```
 10 DIM P% 100
 20 character=&80
 30 oswrch = &FFEE
 40[
 50.enter
 60   LDA character
 70   BEQ exclamation    If zero print '!'
 80   LDA #ASC"*"        Star
 90   JSR oswrch         Print it
100   RTS                Return
110.exclamation
120   LDA #ASC"!"        ExcLamation mark
130   JSR oswrch         Print it
140   RTS                Return
150]
160 END
```

A flowchart for this program is as follows:



Note that the above program can be made shorter, by replacing the instructions

```
JSR oswrch
RTS
```

with the single instruction

```
JMP oswrch
```

Replacing JSR and RTS instructions by a JMP to a subroutine reduces the size of both a source program and the object code it produces, and hence increases execution speed.

Now assemble the program by typing RUN. You should get the message:

```
No such variable at Line 70
```

This is because the assembler processes the mnemonic instructions in the order in which they are listed in the program. Therefore when it encounters ' BEQ exclamation' it has not yet found the label exclamation' so it cannot work out the offset which is required in the following byte. This is known as the forward-reference problem, and is easily overcome using the method of two-pass assembly which is explained below.

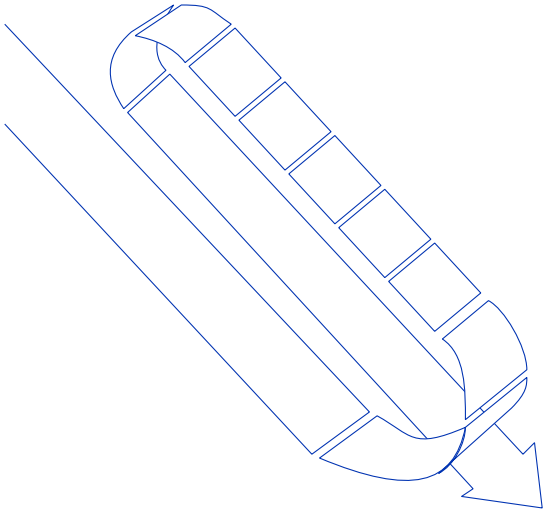
3.4 Two-pass assembly

When a program contains forward references it needs to be assembled twice. During the first pass of the assembler the addresses of all the labels are noted so that during the second pass the offsets of the branch instructions can be included. And the assembler must be told not to worry when, during the first pass, it comes across errors of the sort indicated above.

This can be done using the OPT statement, an assembler directive which has a single parameter for which the following values are possible:

OPT 0 No error messages, and no listing
OPT 1 No error messages, and listing
OPT 2 Error messages reported, and no listing
OPT 3 Error messages reported, and listing (Default)

Thus to suppress messages and a listing on the first pass, and to restore them on the second pass, we need to use OPT 0 and OPT 3 respectively. This can be effected by placing the directive inside a FOR ... NEXT loop, which goes from 0 to 3 in steps of 3. Then



the value of the control variable is used as the parameter of the OPT statement. So, to alter the program which was given above, simply enter these lines:

```
 10 DIM code 100
 23 FOR pass = 0 TO 3 STEP 3
 26 P% = code
 30[ OPT pass
145 NEXT pass
```

This time the error message will not be produced and the correct offset will be calculated for the branch instruction.

Note lines 10 and 26, which replace the old ' DIM P% 100' statement. P% must be reset to the starting value each time that the code is assembled.

Now execute the program by typing

```
CALL enter
```

and verify that the program behaves as it should for different values in &80.

3.5 X and Y registers

The CPU contains two registers, called the X and Y registers, in addition to the accumulator. As with the accumulator, there are instructions to load and store the X and Y registers:

Mnemonic	Description	Symbol
LDX	load **X** register from memory	X=M
LDY	load **Y** register from memory	Y=M
STX	store **X** register to memory	M=X
STY	store **Y** register to memory	M=Y

However, unlike the accumulator, the X and Y registers cannot be used as one of the operands in arithmetic instructions; they have their own special uses which will be outlined later.

The X and Y registers are particularly useful as the control variables in iterative loops, because four special instructions exist which will either increment (add 1 to) or decrement (subtract 1 from) their values.

Mnemonic	Description	Symbol
INX	increment X register	X=X+1
INY	increment Y register	Y=Y+1
DEX	decrement X register	X=X-1
DEY	decrement Y register	Y=Y-1

Note that these instructions do not affect the carry flag: incrementing &FF will give &00 without changing the carry bit. The zero and negative flags are, however, affected by these instructions.

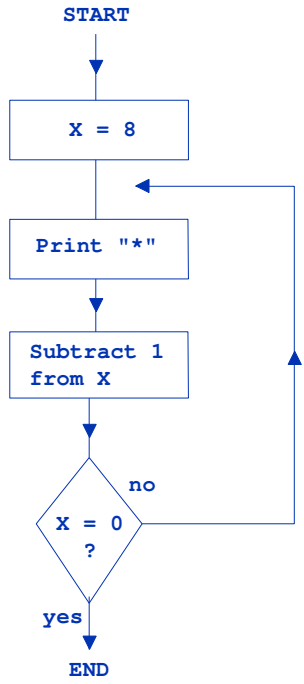
3.6 Iterative loops

The iterative loop enables the same set of instructions to be executed a fixed number of times, e.g.

```
10 DIM P% 100
20 oswrch = &FFEE
30[
40.enter
50 LDX #8        Initialise X
```

```
 60 LDA #ASC"*"    Code for star
 70. Loop
 80 JSR oswrch     Output star
 90 DEX            Count it
100 BNE loop       All done?
110 RTS
120 ] : END
```

A flowchart for the program is as follows:



Assemble the program by typing RUN. This program prints out a star, decrements the X register, and then branches back if the result after decrementing the X register is not zero. Consider what value X will have on successive trips around the loop and predict how many stars will be printed out; then execute the program with ' CALLenter' and see if your prediction was correct. (If you were wrong, try thinking about the case where X was initially set to 1 instead of 8 in line 50.)

How many stars are printed if you change the instruction on line 50 to ' LDX #0' ?

3.7 Comparing values

In the previous example the condition X=0 was used to terminate the loop. Sometimes we might want to count up from 0 and terminate on some other specified value. The compare instruction can be used to compare the contents of a register with a value in memory; if the two are the same, the zero flag will be set. If they are not the same, the zero flag will be cleared. The compare instruction also affects the carry flag by setting it to 1 if the register is greater than or equal to the value in memory, and 0 otherwise.

Mnemonic	Description	Symbol
CMP	**c**o**mp**are accumulator with memory	A-M
CPX	**c**om**p**are **X** register with memory	X-M
CPY	**c**om**p**are **Y** register with memory	Y-M

Note that the compare instruction does not affect its two operands, it just changes the flags as a result of the comparison.

The next example again prints eight stars, but this time it uses X as a counter to count upwards from 0 to 8.

```
10 DIM P% 100
20 oswrch=&FFEE
30[
40.enter
50   LDX #0            Start at zero
60.loop
```

```
 70   LDA #ASC"*"    Code for star
 80   JSR oswrch     Output star
 90   INX            Next X
100   CPX #8         All done?
110   BNE loop       If not then repeat
120   RTS            Else return
130]
140 END
```

In this program X takes the values 0, 1, 2, 3, 4, 5, 6, and 7. The last time around the loop X is incremented to 8, and the loop terminates. Try drawing a flowchart for this program.

3.8 Using the control variable

In the previous two examples X was simply used as a counter, and so it made no difference whether we counted up or down. However, it is often useful to use the value of the control variable in the program. For example, we could print out the character in the X register each time around the loop. The order in which we want the characters would then determine whether we count up or down. We therefore need a way of transferring the value in the X register to the accumulator so that it can be printed out by the OSWRCH routine. One way would be to execute:

```
STX tempaddr
LDA tempaddr
```

where ' tempaddr'is not being used for any other purpose. However, there is a more convenient way, using one of four new instructions:

Mnemonic	Description	Symbol
TAX	**t**ransfer **a**ccumulator to **X** register	X=A
TAY	**t**ransfer **a**ccumulator to **Y** register	Y=A
TXA	**t**ransfer **X** register to **a**ccumulator	A=X
TYA	**t**ransfer **Y** register to **a**ccumulator	A=Y

Note that the transfer instructions only affect the register being transferred to.

The following example prints out the alphabet by making X cover the range A to Z.

```
 10 DIM P% 100
 20 oswrch = &FFEE
 30[
 40.enter
 50   LDX #ASC"A"          Start with the Letter A
 60.loop
 70   TXA                  Put it in the accumuLator
 80   JSR oswrch           Print it
 90   INX                  Next one
100   CPX # (ASC"Z"+1)     Finished ?
110   BNE loop             If so - continue
120   RTS                  Else return
130]
140 END
```

All these examples could have used Y as the control variable instead of X in exactly the same way.

3.9 Conditional assembly

Assembler source text can contain tests, and assemble different statements depending on the outcome of these tests. This is especially useful where slightly different versions of a program are needed for many different purposes. Rather than creating a different source file for each different version, a single variable can determine the changes using conditional assembly, e.g.

```
 10 DIM CODE%100
 20 char=&70
 30 oswrch=&F FEE
 40 osrdch=&FFEO
 50 beLL=7                      Bleep = VDU 9
 60 prompt=ASC":"
 70 INPUT"bell",beLL$           Input 'bell$'
 80 bellflag=INSTR("Yy",bell$)  bellflag' is true if
 90 FOR pass=0 TO 3 STEP 3      bell$ = 'y' or 'Y'
100 P%=CODE%
110[ OPTpass
120.enter
130]
```

```
140 IF bellflag THEN [OPT pass:LDA #bell :JSR oswrch:]
150[OPT pass
160 LDA #prompt                  Load A with ':' prompt
170 JSR oswrch                   Print from A
180 JSR osrdch                   Read in a character
190 STA char                     store it at char and
200 JSR oswrch                   print it out
210 RTS
220]
230 NEXTpass
```

When this program is run it asks if you want the computer to bleep or not and sets ' bellflag' accordingly. Then when the machine code is executed it inputs a character from the keyboard, bleeping if ' bellflag' is set to remind you that an input is required, and prints out a character corresponding to the first key pressed. This character is also saved at the address ' char' .