Quick QL Machine Language

Alan Giles, M.A.

MELBOURNE HOUSE
PUBLISHERS

IN THE UNITED KINGDOM —
Melbourne House (Publishers) Ltd
Castle Yard House
Castle Yard
Richmond, TW10 6TF

IN THE UNITED STATES OF AMERICA —
Melbourne House Software Inc.
347 Reedwood Drive
Nashville TN 37217

IN AUSTRALIA —
Melbourne House (Australia) Pty Ltd
2nd Floor, 70 Park Street
South Melbourne, Victoria 3205

Cataloguing in Publication

Giles, Alan.
Quick QL machine language.
Bibliography.
Includes index.
ISBN Ø 86161 181 Ø.
1. Sinclair QL (Computer) — Programming. I. Title.
ØØ1.64'24

EDITION: 7 6 5 4 3 2 1
PRINTING: F E D C B A 9 8 7 6 5 4 3 2 1
YEAR: 9Ø 89 88 87 86 85 84

Preface

The Sinclair QL is the first of a new breed of computer. It breaks away from the moulds of both the cheap home computer and the more expensive business microcomputer. The QL is neither of these devices, and yet it tries to be both. Only time will tell whether a machine of this size, performance and cost will find a niche in the market place.

The QL is, however, a very interesting machine. Its use of the Motorola 68ØØ8 processor chip makes sure of that. The 68ØØ8 is the baby of a family of microprocessors based around a similar instruction set. The largest member of the family, the 68Ø2Ø, is a true 32 bit microprocessor.

This book sets out to reveal the power of the 68ØØØ instruction set, discussing each instruction in turn, and gives that power to you by including listings for both a 68ØØØ assembler and a disassembler.

As each instruction is introduced I will describe, line by line, how the disassembler deals with it. In the process I will introduce relevant features of QL SuperBASIC and QDOS. Chapter 11 presents information about using machine code programs with QDOS.

This book would have been impossible without a number of other people, and I would like to thank the following for their help:

The two Ians for their invaluable help in exploring the outer reaches of the 68ØØØ instruction set.

Sinclair Research Limited for lending me a QL. (VER$="AH", on which the programs in the book have been tested.)

And finally, Alfred Milgrom and Melbourne House, without whom this book would not have been produced.

Contents

Chapter 1 68ØØØ Machine Code, An Introduction

QL SuperBASIC is, to use a hackneyed phrase, like a breath of fresh air for those of us who are used to the old fashioned BASICs like, dare I say it, Spectrum BASIC. I find that SuperBASIC's **PROCedure**s, **FuNction**s, **LOCal** variables, **SELect** statement and so on, vastly reduce the amount of trivial detail that you have to hold in your head while writing a program. You no longer need to remember line numbers, names of variables to set up before **GO SUB**s, and so forth. This enables the writing of programs in a shorter time which are larger, more powerful and less bug ridden than equivalent Spectrum BASIC programs.

But, as always, to meet the challenge of writing the fastest, most powerful, most memory efficient program on the QL, you need to write programs in machine code. So, I set out to teach myself machine code for the 68ØØ8, the processor at the heart of the QL, and to write an assembler to enable the development of machine code programs. On the way to achieving this aim, I wrote a disassembler which, by letting you look at the workings of the QL's QDOS and SuperBASIC ROM set, can help in the understanding of both 68ØØØ machine code and the QL.

Both the assembler and disassembler are written in SuperBASIC and are presented in full in this book so that you can type them in yourself. Typing the programs into the computer is one of the best ways of forcing yourself to digest the full details of each instruction.

We will look first at the disassembler, for two reasons. Firstly, it allows the instruction set to be examined in related groups of similar types of instruction, rather than imposing a random ordering such as the alphabetical order of the assembly language mnemonics. Secondly, the disassembler is a shorter program than the assembler, so you will have a working program for your QL sooner than if we started with the assembler. When you have finished typing in the disassembler you should be ready for the challenge of typing in the assembler.

Before diving into the listings, I will set the scene by describing some of the architecture of the 68ØØØ family of processors.

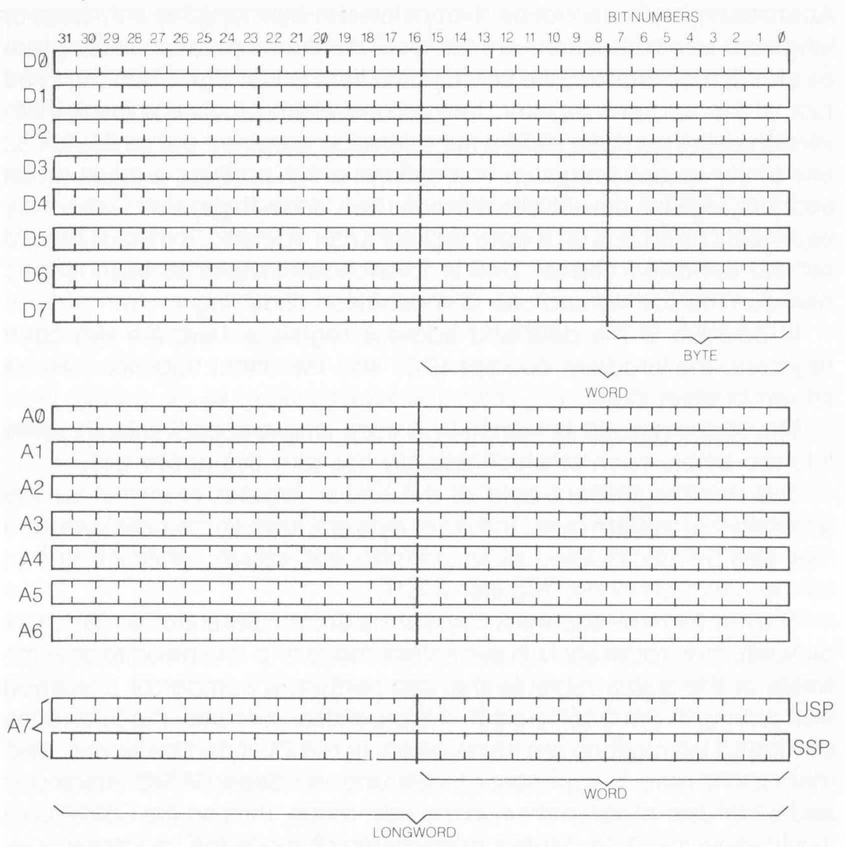
As often remarked, the 68ØØ8 is the baby 8 bit bus processor in a family of very similar processors produced by Motorola. The various members of the family are listed in table 1.1. An advantage of such a family is that Sinclair could redesign the QL hardware to use one of the larger, faster, more expensive, members of the family to produce a new version of the QL, which would be able to run the same software as the old version, even down to the fine detail in the QL ROM. All the family members have the same instruction set, except that the 68Ø1Ø and 68Ø2Ø have some extra instructions to help use hardware memory management and protection devices, and the 68Ø2Ø has some extra addressing modes, including the use of 32 bit relative displacements.

PROCESSOR	EXTERNAL DATA BUS WIDTH	EXTERNAL ADDRESS SPACE SIZE	INTERNAL DATA PATHWAY WIDTH
68ØØ8	8 BITS	$2^{2Ø}$ BYTES	16 BITS
68ØØØ	16 BITS	2^{24} BYTES	16 BITS
68Ø1Ø	16 BITS	2^{24} BYTES	32 BITS
68Ø2Ø	32 BITS	2^{32} BYTES	32 BITS

TABLE 1.1 THE 68ØØØ FAMILY

The 68ØØØ's processing activities are based around the set of 17 registers shown in table 1.2. Each register contains 32 bits. Eight of the registers are called data registers and are named DØ to D7. The remaining nine registers are address registers named AØ to A7, there being two registers called A7, only one of which is accessible at a time. Which version of A7 is available for use depends on the setting of the supervisor status bit which we shall meet shortly. This is somewhat like the alternate register set in the Z8Ø. Although any address register may be used as a stack pointer, certain instructions such as subroutine calls use A7 implicitly as a stack pointer, so assemblers often allow A7 to be called SP. The version of A7 accessible in supervisor mode is called the

supervisor stack pointer or SSP, and that available in user mode the user stack pointer or USP.



BIT NUMBERS
31 30 29 28 27 26 25 24 23 22 21 2Ø 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 Ø
DØ
D1
D2
D3
D4
D5
D6
D7
BYTE
WORD
AØ
A1
A2
A3
A4
A5
A6
A7
USP
SSP
WORD
LONGWORD

TABLE 1.2 DATA AND ADDRESS REGISTERS

One of the beauties of the 68ØØØ instruction set is its ability to handle byte (8 bit), word (16 bit), and longword (32 bit) data items in a similar fashion. There are often three versions of the same instruction, with extensions ".B", ".W" and ".L" for the three lengths. Many instructions treat numbers as signed numbers using twos complement notation. In this notation, for instance, minus 1 is stored as the number 2^8-1 in a byte, 2^{16}-1 in a word, or 2^{32}-1 in a longword. All negative numbers have their most significant bit set, and this bit is known as the sign bit.

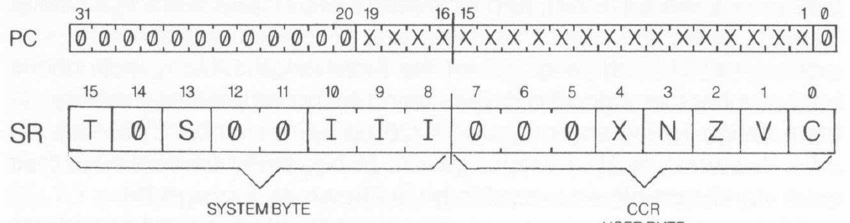
Where an instruction can operate on either data registers or address registers, the length specification has different effects. A byte or word operation on a data register only affects the least significant byte or word, the remaining 24 or 16 bits of the register are left untouched. Address registers cannot be manipulated in byte lengths: only word or longword size instructions are allowed. **If a word operation takes place on an address register, the source data item is first sign extended to 32** bits of the address register. **Instructions which produce a result in an** version of the number, before the operation is carried out on the full 32 bits of the address register. **Instructions which produce a result on an address register do not set the condition code flags**, even when the equivalent operation on a data register or an operand in memory would set the condition codes. This is to allow addresses to be changed between the setting and use of a condition code flag.

In addition to the data and address registers there are two other registers, the program counter (PC) and the status register (SR) as shown in table 1.3

The status register is nominally a word long and conveniently splits into two bytes, each of which actually has only five active bits.

The most significant byte of the status register is known as the supervisor or system byte. The other byte is known as the user byte, and may also be manipulated as an independent register, when it is known as the condition codes register (CCR).

The key bit in the supervisor byte is the Supervisor status bit (**S**). If this bit is set, the processor is in supervisor mode and is allowed to alter the whole of the status register and can perform a number of privileged instructions. If the **S** bit is clear, the processor is in user mode and the privileged instructions are not allowed. In the QL it can be loosely said that QDOS runs in supervisor mode and the SuperBASIC interpreter and other user programs run in the user mode. Pins on the 68008 chip would allow the QL hardware to check which mode the processor is in, and prevent access to certain devices or memory as the designer felt appropriate, but the QL does not use this feature. As mentioned above, there are two versions of the A7 register, one active in supervisor mode,



31
20 19
16 15
1 0
PC
0 0 0 0 0 0 0 0 0 0 0 0 X X X X X X X X X X X X X X X X X X X 0
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
SR
T 0 S 0 0 I I I 0 0 0 X N Z V C
SYSTEM BYTE
CCR
USER BYTE

TABLE 1.3 SPECIAL PURPOSE REGISTERS

the other in user mode. Space on the supervisor stack in the QL is limited, and to reduce the chance of running out of stack space, programs should normally be run in user mode. The **T** bit in the status register is the Trace status bit. If set, an interrupt-like trap occurs after every instruction, allowing the execution of a program being tested to be traced instruction by instruction.

The I bits are an external Interrupt mask. When a hardware device wants to interrupt the processor it presents an interrupt level number from 1 to 7 to the processor. Only level numbers which are higher than the value of the current interrupt mask will cause the 68008 to call an interrupt processing routine. Level 7 is special in that it always causes an interrupt, and can thus be treated as the equivalent of a non-maskable interrupt available on other processors. When an interrupt is accepted by the processor, the **I** bits are changed to the interrupting level number so as to prevent lower priority devices from diverting the processing of the interrupt. On the QL, all normal devices cause level 2 interrupts when they require processor action, and only level 7 interrupts are recognised by QDOS in a manner which would allow you to write your own interrupt handling routines (see Chapter 6 for details).

The condition code flags are the normal arithmetic result codes, which are similar to those you find in the Z80 processor used in the Spectrum. **C** is a **C**arry (or in the case of subtraction, a borrow) from the most significant bit of an arithmetic result, indicating an unsigned arithmetic overflow. It also has special meaning in the shift and rotate operations of Chapter 10. The **V** bit is the signed arithmetic o**V**erflow flag, indicating that a result had the opposite sign to that which might be expected from a simple examination of sign bits. **Z** indicates a **Z**ero result. **N** is a copy of the sign bit of the result which will be set if the result was **N**egative. **X** is the e**X**tended arithmetic bit, and during addition, subtraction and many of the shifts and rotates, is a simple copy of the **C** bit. However, **X** is not affected by as many instructions as **C**, and this allows useful instructions to be fitted in between the calculation which produces **X** and the operation which uses it.

The only register left to be discussed is the program counter (PC). The register is nominally 32 bits long, but as there are only 20 address pins on the 68008 chip, the most significant 12 bits of PC are irrelevant. Further, as all 68000 instructions are an even number of bytes long, Motorola have chosen to insist that all instructions start on an even byte boundary. Thus, the least significant bit of PC is always zero and only 19 bits of PC are active. Naturally, PC is incremented during the course of an instruction so that, after an instruction has been executed, PC points to the next instruction, unless the instruction was one which alters the flow of the program, in which case the instruction alters PC to point to the next instruction to be executed.

Chapter 2 SuperBASIC Initialisation and Control Routines for a Disassembler

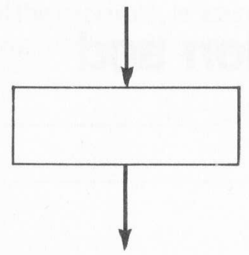
SuperBASIC has made it relatively easy to write both the assembler and disassembler in what is known as a top-down, structured manner.

In the top-down approach to programming you start writing a program to do the whole job, disassembly or assembly in our case, but you do the job by calling procedures or functions to do any self-contained or repetitive task. These procedures or functions are again broken down in a similar manner until the task is simple enough to be achieved directly in SuperBASIC. It is quite comforting to write SuperBASIC programs in this manner, as you are able to put off until later some complex task by naming it as a procedure. When you actually come to programming the task you will be able to concentrate exclusively on it, rather than worrying about the rest of the program.

The top-down, structured approach has the disadvantage that it is difficult to decide in advance exactly where to draw the dividing line which allocates some task to a procedure or function. Thus, there can be a tendency to duplicate sections of code which would have been more effectively separated off as a procedure. The specification for such a procedure is often only realised as you type in the same sort of code for the third or fourth time. An alternative approach to programming, called bottom-up, structured programming, tries to identify all these simple routines in the first instance and link these routines together to form the required program. Such an approach tends to need a lot of intuition in identifying the procedures to be written and how they should be called.

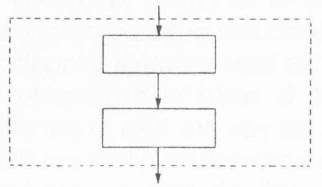
In practice, I try to get around some of the problems by writing programs in a different order to that in which they are presented, attacking the heart of the problem first, and then building around that in a technique I call middle-out, semi-structured programming.

An approach such as top-down, structured programming is a technique designed to make programmers think clearly about what it is they want the program to do. What really matters is the behaviour of the

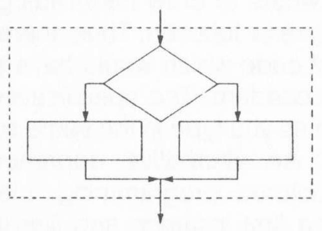


A structured program block may be a **PROCedure, FuNction**, or perhaps just a single statement. Indeed it can be any program unit with a single entry and a single exit. The actual contents of the block do not matter; all that matters is the function it performs.

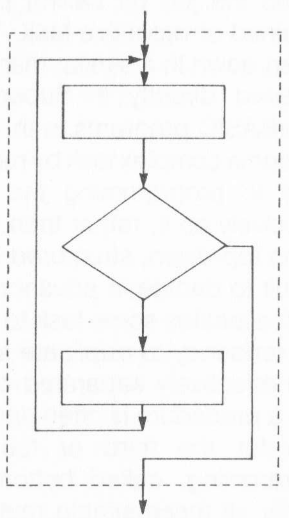
One way of proving that a block performs the task it was designed to perform is to insist that blocks combine in a restricted number of ways to form more complex blocks. For example:



Sequential composition of two blocks.



Alternation using **IF** with **THEN** block and **ELSE** block.

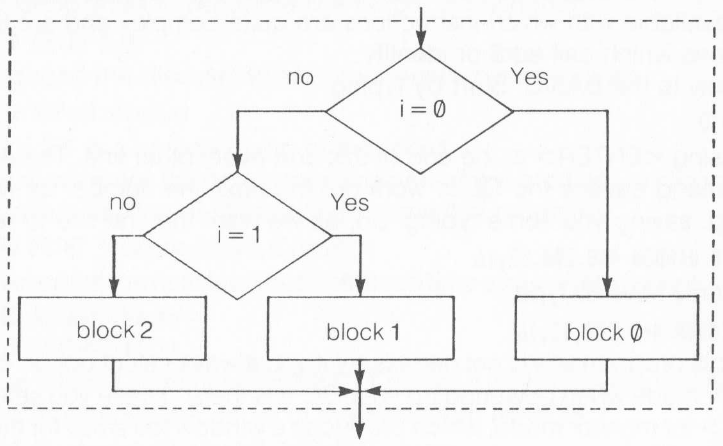


A general **REPeat** loop with a central conditional **EXIT**.

Any program can be constructed from simple blocks using a combination of these three techniques. For example the statement:

```
SELect ON i
=0
  block0
=1
  block1
=REMAINDER
  block2
END SELect
```

could be visualised as two nested **IF . . . THEN . . . ELSE** combinations.



program produced and, based on that behaviour, the confidence that the user has in the correctness of the program's results; plus, of course, if that confidence is not justified, the ease with which a bug can be corrected. There were a number of bugs in the disassembler and assembler when I first wrote them: I tracked down quite a few by combining much of the two programs together into a program which disassembled some code, then assembled the text produced and checked that the code produced was the same as the code disassembled. This revealed a number of errors and oversights, whose causes were relatively easy to identify with the help of the program structure. The differences which remain between disassembly and assembly of the QL ROM are now all explainable.

In the disassembler I have gone furthest down the road to structuring, by making all the variables **LOCal** and explicitly mentioning the names of all variables passed in procedure calls. The assembler needs much more information to be common to many procedures, and consequently I have taken the opposite approach, and most procedures are called without any explicit parameters.

Both the assembler and disassembler have the assembly language mnemonics and instruction code bit patterns embedded in the BASIC, because the 68000 instruction set is not repetitive enough for the more conventional instruction table approach to work. The only feature common to many instructions is a choice of addressing modes as exemplified by the function **adr$** in the disassembler or the procedure **identify** in the assembler. But, the rules about which addressing modes are available with which instructions are quite complex and are left to routines which call **adr$** or **identify**.

Now to the BASIC. Start by typing:

```
AUTO
```

pressing <ENTER> at the end of this and every other line. The **AUTO** command causes the QL to work out the next line number as we go along, saving you some typing. So, as we want the first line to read:

```
100 WINDOW 448,200,32,16
```

you only have to type:

```
WINDOW 448,200,32,16
```

This program line is not necessary if you always select option <F2> for TV mode when powering up your QL. It is there in case you selected <F1> for monitor mode, which produces a window too small for the line length of listing we would like to use. This particular **WINDOW** is worth remembering, as it resets the window size and position to the normal TV mode size, without you having to press the reset button.

The next line is also not absolutely necessary and, indeed, during program development you may wish to leave it out, as it closes the listing window (#2). I do this to reduce the multi-coloured screen flashing normally associated with the loading and starting of a program. Unfortunately, it is not enough simply to **CLOSE#2**, as subsequent **RUN**s of the program would generate a **channel not open** error. To prevent this, we make the line read:

```
110 OPEN#2;scr_:CLOSE#2
```

This opens channel 2 to the default screen, which does not generate an error even if channel 2 is already open, and then immediately closes it. Note the change in terminology from the Spectrum. The # numbers are called channel numbers rather than stream numbers.

```
120 MODE 4
```

We need to use the high resolution mode to display the line length needed by the disassembler. This command also has the effect of clearing the screen.

```
130 CSIZE 3,1
```

We will print the title in the largest size.

```
140 PRINT "68000▲Disassembler"
150 CSIZE 2,0
160 PRINT "©1984▲Alan▲Giles"
170 CSIZE 0,0
```

where ▲ is used to indicate a space. If you find that the small character size is too small to read, the line length is such that:

```
170 CSIZE 3,0
```

will spread the disassembler output neatly over two lines.

```
180 REPeat display
```

Figure 2.1 is a screen dump of the result.

Now we enter the actual working part of the program, a loop which displays a section of memory each time it is repeated:

```
190 PRINT "Code▲or▲Data?▲(C▲or▲D)";
```

We repeat the next few lines until such time as a correct reply is input.

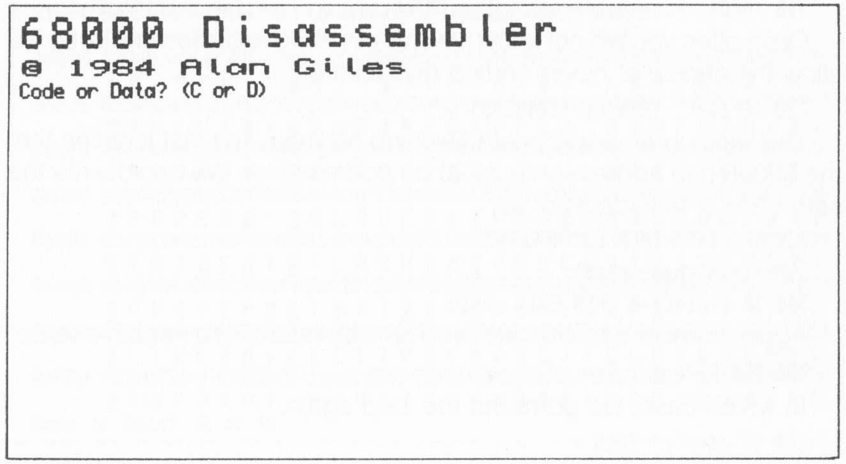
```
200 REPeat inmode
210 a$=INKEY$(-1)
```

The minus 1 parameter to **INKEY$** makes it wait until a key is pressed, unlike the immediate return of the Spectrum **INKEY$**.

```
220 IF a$=="c" OR a$=="d" THEN EXIT inmode
```

Note the use of == to do a case independent string comparison, so that it does not matter whether <CAPS LOCK> is operating or not when the key is pressed.

```
230 END REPeat inmode
```



```
68000 Disassembler
© 1984 Alan Giles
Code or Data? (C or D)
```

FIGURE 2.1

If an incorrect key was pressed, this causes the computer to loop round and wait for another letter to be typed.

```
240 PRINT !a$
```

We print out the letter keyed, as a reminder to the user.

```
250 REPeat instart
```

Now, we repeat a loop until the program is given a sensible starting adddress.

```
260 INPUT "Start␣address?␣(Use␣'$'␣to␣indicate␣hexadecimal)"!b$
```

The program accepts addresses in either decimal or hexadecimal notation; the standard Motorola notation is used, where hexadecimal numbers are prefixed by a dollar sign.

Conversion from a hexadecimal number string to a QL floating point number seems like a useful thing to do in a function, so we write:

```
270 start=number(b$)
```

Doing this also has the advantage of putting off working out how to do the conversion until we have had time to think about it; but we do need to decide how **number** will indicate any problems it finds in decoding **b$**. All addresses are positive, so we decide to return a negative number if there is an error.

```
280 IF start>=0 THEN EXIT instart
290 END REPeat instart
```

If there was an error, we try again.

```
300 REPeat infin
```

Now we do the same again for a finishing address.

```
310 INPUT "Finish␣address?␣(Use␣ENTER␣to␣carry␣on␣for␣ever)"!c$
```

Quite often you will not know how far you have to disassemble, so we allow the choice of never ending disassembly.

```
320 IF c$="" THEN c$="1048575"
```

This sets up an appropriate finishing address, the last location that the 68008 can address with its 20 bit address bus. We could write the same line in hexadecimal as:

```
320 IF c$="" THEN c$="$FFFFF"
330 finish=number(c$)
340 IF finish>=0 THEN EXIT infin
```

Again, **number** can indicate problems by returning a negative value.

```
350 END REPeat infin
```

In which case, we go round the loop again.

```
360 IF a$=="c" THEN
```

We now know enough to start disassembling, but this is just the sort of task which needs fitting into a procedure.

```
370 disassemble start,finish
```

Writing that line is a relief. We have put all the hard work in a package we can concentrate on later. Meanwhile we can get on with the much simpler case where you just want to see the memory contents as data, naturally in both numeric and character forms. To make life more complicated, we decide to place the character forms midway between the positions of the two hexadecimal digits of the numeric form using the QL **CURSOR** command. To do this we need to know which line the cursor is on. Figure 2.2 shows the sort of thing we are aiming for.

```
380 ELSE
390 SCROLL -10
```

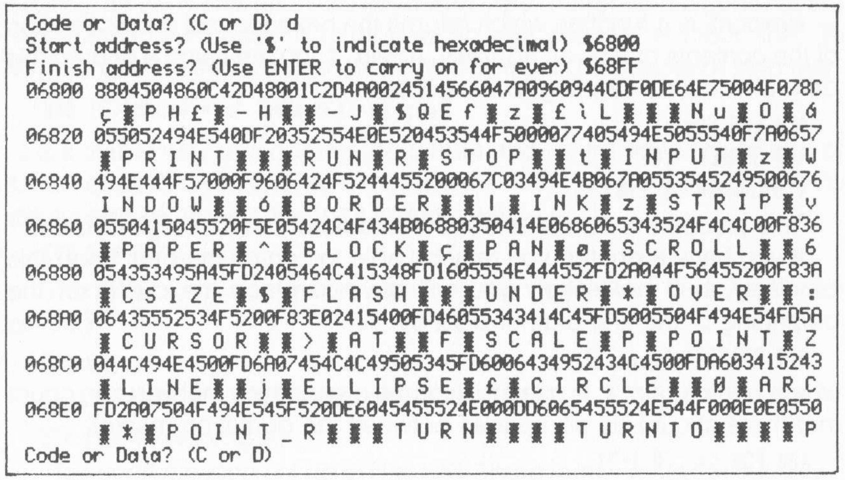
So, we make sure that the bottom line of the window is clear. Note that positive scrolls move the screen contents in a positive direction in QL pixel co-ordinates, that is, down the screen. For the more conventional interpretation of the word **SCROLL**, we need to use negative parameters.

```
400 CURSOR 0,190
```

Then we can safely place the cursor at the beginning of the last line.

```
410 FOR i=start TO finish STEP 32
```

There is room on a line for 32 bytes of data, but you might find the data easier to read if the character size is larger or there are fewer characters;



```
Code or Data? (C or D) d
Start address? (Use '$' to indicate hexadecimal) $6800
Finish address? (Use ENTER to carry on for ever) $68FF
06800 8804504860C42D48001C2D4A00245145666047A0960944CDF0DE64E75004F078C
       ç █ P H £ █ - H █ █ - J █ $ Q E f █ z █ £ ì L █ █ █ N u █ O █ á
06820 055052494E540DF20352554E0E520453544F5000077405494E5055540F7A0657
       █ P R I N T █ █ █ R U N █ R █ S T O P █ █ t █ I N P U T █ z █ W
06840 494E444F57000F9606424F5244455200067C03494E4B067A055354524950O676
       I N D O W █ █ ó █ B O R D E R █ █ | █ I N K █ z █ S T R I P █ v
06860 0550415045520F5E05424C4F434B06880350414E0686065343524F4C4C00F836
       █ P A P E R █ ^ █ B L O C K █ ç █ P A N █ ø █ S C R O L L █ █ 6
06880 054353495A45FD2405464C415348FD1605554E444552FD2A044F56455200F83A
       █ C S I Z E █ $ █ F L A S H █ █ █ U N D E R █ * █ O V E R █ █ :
068A0 06435552534F5200F83E02415400FD46055343414C45FD5005504F494E54FD5A
       █ C U R S O R █ █ > █ A T █ █ F █ S C A L E █ P █ P O I N T █ Z
068C0 044C494E4500FD6A07454C4C49505345FD6006434952434C4500FDA603415243
       █ L I N E █ █ j █ E L L I P S E █ £ █ C I R C L E █ █ Ø █ A R C
068E0 FD2A07504F494E545F520DE6045455524E000DD6065455524E544F000E0E0550
       █ * █ P O I N T _ R █ █ █ T U R N █ █ █ █ T U R N T O █ █ █ █ P
Code or Data? (C or D)
```

FIGURE 2.2

or, to help in reading longword tables, you might prefer to print the data in longwords of 4 bytes. I leave this decision to you.

```
420 PRINT hex5$(i)!;
```

We define a function to print the five hexadecimal digits of an address, and use it. Note that the semi-colon is needed to force the exclamation mark to cause a space to be printed between the address and the data which will follow.

At this point it is worth mentioning the possibility of disassembly output to a printer. It is probably best to synchronise output on both screen and printer, so that you can see any characters which might be left in the printer buffer. To use a printer in this way you need to open a channel to it, set up the baud rate and possibly send some opening control characters which will depend on the type of printer involved. Perhaps something like:

```
90 OPEN#4;ser1e:PRINT#4;CHR$(12);
```

Then every time data is printed to the screen it also needs to be sent to the printer, so you might add:

```
425 PRINT#4;hex5$(i)!;
```

To return to the general program, we now need to print the data in hexadecimal:

```
430 FOR j=i TO i+31
```

If you are printing less then 32 bytes per line you will need to alter the figure 31 in this line:

```
440 PRINT hexcon$(j);
```

hexcon$ is a function which returns the hexadecimal representation of the contents of a given address. Again, if you are using a printer, this data also needs sending to the printer.

```
450 END FOR j
460 SCROLL -10
470 CURSOR 39,190
```

This places the cursor under the middle of the first digit of data.

If you have a printer, you may be able to control the print head this precisely, but more likely you will have to place the cursor at the beginning of the first digit of data by:

```
475 PRINT#4;"▲▲▲▲▲▲";
```

where once again ▲ is used to indicate a space (so that you can count that there should be six spaces between the quotation marks).

```
480 FOR j=i TO i+31
```

On the screen, all characters except <ENTER> (**CHR$(10)**) have a symbolic representation. If you have a printer you may have problems with more characters, and want to exclude them. Here is a method for excluding **CHR$(10)** from the screen print.

```
490 IF PEEK(j)<>10 THEN
500 PRINT CHR$(PEEK(j))!;
510 ELSE
520 PRINT "\;";
530 END IF
```

The backslash symbol " \ " is used for **CHR$(10)** as a reminder that it causes a new line of printing; the semicolon is added to differentiate it from normal backslashes.

```
540 END FOR j
550 PRINT
```

This ends a block of 32 bytes of data output, and at this point it is worth discussing ways of pausing or stopping screen output so that you have time to examine the output. The QL manual frequently mentions <BREAK> which is keyed by holding down <CTRL> and pressing the <SPACE> bar, and as things stand you may use this to stop the output at any point, then using the command **RUN** will allow you to start the program again. There is another useful <CTRL> key pairing which I cannot find mentioned in my copy of the QL manuals; but if you hold down <CTRL> and press <F5> you pause the screen output; to resume output you press any key. Which key you do press does matter slightly, as it will end up in the keyboard input buffer, and a subsequent <BREAK> will cause it to appear in the channel 0 input line. To supplement these methods we can include our own pause or stop test.

```
560 IF INKEY$=="s" THEN NEXT display
```

is a suitable way of stopping the output and returning to the "Code or Data" question, you just have to press the <S> key, as **INKEY$** with no parameter returns its findings immediately.

```
570 END FOR i
```

The **FOR** loop terminates when all the requested data has been printed.

```
580 END IF
```

This matches the **IF** statement in line 360.

```
590 END REPeat display
```

This ends the loop set up in line 180. There is no way out of this loop, no **EXIT** or **GO TO** which would cause the program to look for further lines of code. So we have finished writing the disassembler—apart, that is, from one or two procedures and functions we have used.

```
600 DEFine FuNction number(a$)
```

When we used **number**, we said that it coped with both decimal and hexadecimal strings, **a$**, and returned their value, or a negative number if there was a problem.

```
610 LOCal i,j,k
```

We need a few counters, so we make them **LOCal** so that they do not affect other, similarly named, variables outside the function **number**.

```
620 IF a$="" THEN RETurn -1
```

The null string is the simplest case.

```
630 IF a$(1)>="0" AND a$(1)<="9" THEN RETurn a$
```

Decimal numbers can be handled very easily, we use the QL's coercion to convert the string **a$** to the floating point value **RETurn**ed by **number**.

```
640 IF a$(1)<>"$" THEN RETurn -1
```

If the string does not claim to be a hexadecimal number then we have our next problem case.

```
650 i=0
```

We will use **i** to contain the value we calculate.

```
660 FOR j=2 TO LEN(a$)
670 k=CODE(a$(j))
680 IF k>=CODE("a") THEN k=k-32
```

We want to scan the number and use **SELect** to deal with the various possible cases, but **SELect** does not work on strings, so we have to convert each character to a number and do our own test to ignore the difference between capital and lower case letters.

```
690 SELect ON k
700 =CODE("0") TO CODE("9"):i=i*16+k-CODE("0")
```

I always use the short **=number** form of **SELect** choices rather than **ON k=number** because the **ON k** is ignored and you could be led astray by using **SELect ON k** and then **ON j=** into thinking that somehow **j** would now be tested, when in fact SuperBASIC continues to test **k**.

```
710 =CODE("A") TO CODE("F"):i=i*16+k+10-CODE("A")
```

The above is the official, most easily understood, version of the calculation, emphasising that "A" represents the value 10. If you accept that the ASCII value of "A" is 65 then the calculation is very slightly quicker if you type:

```
710 =CODE("A") TO CODE("F"):i=i*16+k-55
720 =REMAINDER :RETurn -1
```

If any of the characters in the string is not a valid hexadecimal digit then there is no point in continuing, so we return with an error indication.

```
730 END SELect
740 END FOR j
750 RETurn i
```

When the **FOR** loop terminates we can return the value of **number**, which has now been calculated.

```
760 END DEFine
```

SuperBASIC has an option which would allow us to write

```
760 END DEFine number
```

but, as we cannot make one definition inside another, there is no confusion without the word **number**. We can, and later in the program do, nest **SELect** and **IF** to a potentially confusing level and we might like to add similar comments to **END SELect** and **END IF**, but these are not allowed by SuperBASIC.

```
770 DEFine FuNction hex5$(a)
780 LOCal i,a$
```

As usual, we need some **LOCal** variables.

```
790 a$=""
```

We start with nothing in a string, and add each hexadecimal digit in turn.

```
800 FOR i=4 TO 0 STEP -1
810 a$=a$&hex$(INT(a/16^i)-16*INT(a/16^(i+1)))
```

Unfortunately, we cannot use the integer operators **MOD** and **DIV** to select the hexadecimal digit we want as **MOD** and **DIV** only deal with 16 bit numbers and the address **a** may be a 20 bit number.

```
820 END FOR i
830 RETurn a$
840 END DEFine
850 DEFine FuNction hexcon$(a)
```

This is a related hexadecimal function, returning, as you may remember, a hexadecimal string representing the contents of address **a**.

```
860 RETurn hex$(PEEK(a)DIV 16)&hex$(PEEK(a)MOD 16)
870 END DEFine
```

This time we can use **DIV** and **MOD**, making life much simpler.
Finally, in this group of hexadecimal conversion functions we need:

```
880 DEFine FuNction hex$(a)
890 IF a<10 THEN RETurn a
```

Note the use of coercion again to deal with the hexadecimal digits 0 to 9.

```
900 RETurn CHR$(55+a)
```

The magic number 55 reappears to deal with "A" to "F".

Note that **hex$** does no error checking, and if supplied with a number outside the range 0 to 15 will generate erroneous results.

```
910 END DEFine
```

Now, at last, we can get down to the disassembler itself. Figure 2.3 is an example of the format we are aiming for.

```
920 DEFine PROCedure disassemble(start,finish)
930 LOCal i,pc,a$
```

As mentioned earlier, all 68000 machine code instructions are a multiple of 16 bits long and so start at any even byte address. Indeed, the 68008 will generate a **TRAP** known as an address error if you try to execute an instruction starting at an odd address.

```
940 IF start/2<>INT(start/2) THEN start=start-1
```

Again, we cannot use **MOD** because start may be longer than 16 bits.

```
950 REPeat loop
```

Everything the program does from this point on is repeated until the required section of code has been disassembled. This point does mark the start or end of a line of disassembly, so we add our test for pressing the <S> key.

```
960 IF INKEY$=="s" THEN RETurn
970 IF start>finish THEN PRINT FILL$("▴",29);"END":RETurn
```

```
Code or Data? (C or D) c
Start address? (Use '$' to indicate hexadecimal) $1B1A
Finish address? (Use ENTER to carry on for ever) $1B3E
01B1A 4E75                  RTS
01B1C 20280022              MOVE.L  $0022(A0),D0
01B20 6B1A                  BMI.S   $1A(PC)=$01B3C
01B22 4A40                  TST.W   D0
01B24 6B16                  BMI.S   $16(PC)=$01B3C
01B26 D0A80026              ADD.L   $0026(A0),D0
01B2A B068001E              CMP.W   $001E(A0),D0
01B2E 620C                  BHI.S   $0C(PC)=$01B3C
01B30 4840                  SWAP    D0
01B32 B068001C              CMP.W   $001C(A0),D0
01B36 6204                  BHI.S   $04(PC)=$01B3C
01B38 7000                  MOVEQ   #$00,D0
01B3A 4E75                  RTS
01B3C 70FC                  MOVEQ   #$FC,D0
01B3E 4E75                  RTS
                            END
Code or Data? (C or D)
```

FIGURE 2.3

This is the normal end of loop test, where ▴ stands for a blank as usual, and **END** is the conventional assembler mnemonic marking the end of a piece of code. If you are using a printer then, as usual, you will need to include a command to print the same text on the printer.

```
980 PRINT hex5$(start)!;
```

Each output line starts with the address of the item being disassembled.

```
990 pc=start
```

pc is our version of the program counter, it keeps track of how far disassembly has progressed, generally pointing to the next word to be disassembled.

```
1000 fault=0
```

We use the flag **fault** to indicate any problems with disassembly, so we start by setting it to indicate no problems.

```
1010 a$=dis$(pc)
```

Again, we put a large portion of the work yet to be done into a function. **dis$** is intended to return a text representation of the disassembled next instruction, and advance **pc** to point to the following instruction; but, it may indicate a fault, in which case both the text and **pc** may not be left in sensible places.

```
1020 IF fault THEN
1030 pc=start+2
```

If there is a fault then we may have been trying to disassemble some data, so we want to display it as data but switch back to disassembling code as quickly as possible, so we only display one word of data.

```
1040 a$="DC.B▴▴▴▴"
```

The usual assembler mnemonic for data is **DC** which stands for define constant. The ".B" indicates byte size data, which we will use in case the data is really a text string. Note that we will display all mnemonics in an eight character field, hence the four blanks.

```
1050 IF PEEK(start)>=CODE("▴") AND PEEK(start)<=CODE("©") THEN
1060 a$=a$&"'"&CHR$(PEEK(start))&"',"
```

Note the use of the apostrophe to mark a text constant used by the assembler; this is used in preference to the quotation marks which tend to be used by BASIC.

```
1070 ELSE
1080 a$=a$&PEEK(start)&","
```

If the byte is not a printable character, we display its value in decimal.

```
1090 END IF
```

We now have to do the same for the second byte of the word.

```
1100 IF PEEK(start+1)>=CODE("▲") AND PEEK(start+1)<=CODE("©") THEN
1110 a$=a$&"'"&CHR$(PEEK(start+1))&"'"
1120 ELSE
1130 a$=a$&PEEK(start+1)
1140 END IF
1150 END IF
```

This ends the **IF** statement started at line 1020 as, whether there was a fault or not, we now have a valid disassembly text in **a$** and a sensible value for **pc**, so we can print the hexadecimal codes which represent the instruction disassembled. The difference in values between **start** and **pc** allow us to determine how many bytes should be printed.

```
1160 FOR i=start TO start+10
1170 IF i>=pc THEN
1180 PRINT "▲▲";
1190 ELSE
1200 PRINT hexcon$(i);
1210 END IF
1220 END FOR i
1230 PRINT !a$
1240 start=pc
```

We advance the pointer **start** to the beginning of the next instruction, and can go round the loop again.

```
1250 END REPeat loop
1260 END DEFine
```

To try to disassemble the 68000 machine code as quickly as possible, we try to divide and conquer, splitting the operation codes into as many different groups as possible. The 68000 operation codes are not arranged in any particular order. Indeed, Motorola seem to have been at some pain to fill any holes in the code set with other instructions. However, the first hexadecimal digit of an instruction is a good guide to what an instruction does, and we will divide the problem on that basis.

```
1270 DEFine FuNction dis$(pc)
1280 LOCal j
1290 j=PEEK(pc) DIV 16+1
1300 ON j GO TO 1310,1320,1330,1340,1350,1360,1370,1380,1390,1400,141
     0,1420,1430,1440,1450,1460
```

This is the only **GO TO** in the program. As **GO TO** is generally frowned upon in structured programs, I shall explain why I have used it in place of **SELect**.

When choosing amongst a number of options, **SELect** makes a comparison with each and every number range listed until it finds the one that matches. The comparison is always with a number range, to allow for inaccuracy in the last digit of floating point numbers. In other words, **=1** causes **SELect** to actually check the range between 1-1E-7 and 1+1E-7. This multiple checking process is inherently slower than **ON j GO TO**, where the processor picks up the destination line number straight away. When there are only a few selections, the speed difference is not too noticeable but, with as many as 16 selections, this section of the program could represent quite a time investment.

Having taken the decision to use **ON j GO TO**, we can reduce the possibility of bugs by using it in a structured manner. We must ensure that all the destination line numbers are within the same function, and we use discrete line numbers, rather than lines of the type

```
1300 GO TO j*10+1310
```

which would not be handled correctly by **RENUM**.

With this apology we can continue:

```
1310 RETurn dis0$(pc)
1320 RETurn dis1$(pc)
1330 RETurn dis2$(pc)
1340 RETurn dis3$(pc)
1350 RETurn dis4$(pc)
1360 RETurn dis5$(pc)
1370 RETurn dis6$(pc)
1380 RETurn dis7$(pc)
1390 RETurn dis8$(pc)
1400 RETurn dis9$(pc)
1410 fault=1:RETurn ""
1420 RETurn disB$(pc)
1430 RETurn disC$(pc)
1440 RETurn disD$(pc)
1450 RETurn disE$(pc)
1460 fault=1:RETurn ""
1470 END DEFine
```

Operations beginning with the hexadecimal digits "A" or "F" are not considered as valid by the 68008. Motorola have chosen to reserve these ranges of codes for use by co-processors such as a floating point calculator chip or a memory management chip. The idea is that, when such an instruction is encountered, the co-processor can read the passing instruction and steal whatever memory accesses it needs to do its own operation. Alternatively, the trapping of illegal instructions within the 68008 processor would allow the co-processor operations to be done in software by the main processor. In practice the QL does not allow such extra instructions to be used, as will be explained in Chapter 6.

So, following some fairly heavy work in this chapter, we have managed to split the disassembly task up in to fourteen much simpler tasks, which we can proceed to write.

Chapter 3
MOVE and the 68000 Addressing Modes

Rather than dealing first with the instructions covered by **dis0$**, let us now look at that most general and widely used command **MOVE**. On the Z80 processor used in Sinclair's Spectrum, ZX81 and ZX80, the data movement command is called **LD** and is written thus:

```
LD      destination,source
```

On the 68008, the equivalent command is **MOVE** and it is written like this:

```
MOVE    source,destination
```

which is an example of a general rule that when the 68000 series can be different to the Z80, it is different.

There are many different versions of the **MOVE** command, some of which are fitted into odd places in the code table, like **LD A,I** and **LD A,R** were fitted into the Z80 code table; but the main group of **MOVE** commands occupies all the codes covered by the functions we have called **dis1$**, **dis2$** and **dis3$**.

```
1480 DEFine FuNction dis1$(pc)
1490 LOCal i,j,a$
1500 a$="MOVE.B▲▲"
```

All of **dis1$**, is concerned with moving byte sized items of data. As you can see in table 3.1, the remaining 12 bits of the instruction word are split into two groups of six bits which describe the source and destination data items.

```
1510 i=PEEK(pc)*4 MOD 64+PEEK(pc+1)DIV 64
```

makes **i** equal to the six bits describing the destination.

```
1520 IF i MOD 8=1 THEN fault=1:RETurn ""
```

The destination cannot be an address register, as byte operations are not allowed on address registers, so we reject that. The details of why **i MOD 8=1** means an address register will be covered shortly.

MOVE.B source, destination

0	0	0	1	DESTINATION REGISTER NUMBER	DESTINATION ADDRESSING MODE	SOURCE ADDRESSING MODE	SOURCE REGISTER NUMBER

MOVE.L source, destination

0	0	1	0	DESTINATION REGISTER NUMBER	DESTINATION ADDRESSING MODE	SOURCE ADDRESSING MODE	SOURCE REGISTER NUMBER

MOVE.W source, destination

0	0	1	1	DESTINATION REGISTER NUMBER	DESTINATION ADDRESSING MODE	SOURCE ADDRESSING MODE	SOURCE REGISTER NUMBER

TABLE 3.1 GENERAL MOVE COMMAND FORMAT

```
1530 IF i MOD 8=7 AND i>15 THEN fault=1:RETurn ""
```

Program counter relative addressing is also not allowed for the destination, as the theory is that the program is written in ROM and therefore PC relative addresses also refer to ROM. This is the most annoying addressing mode restriction for us, as we want to write programs in RAM which will have to use data addresses in the block of memory we reserve for both program and data and we will only know the data's relative address rather than its absolute address.

```
1540 IF PEEK(pc+1)MOD 64=60 THEN
```

A value of 60 in the source description bits indicates that the item to be moved is immediate data.

```
1550 pc=pc+4
```

As 68000 instructions must occupy an even number of bytes, despite the fact that we only need one byte of immediate data, two bytes must be reserved, along with the two bytes of the instruction code.

```
1560 IF PEEK(pc-2)<>0 AND PEEK(pc-2)<>255 THEN fault=1:RETurn ""
```

The extra byte is officially supposed to be zero, but many assemblers, including Motorola's own, can sign extend the single byte of immediate data into the spare byte by copying the sign bit into all the bits of the other byte. We want to reject any instruction the assembler cannot produce, so that the disassembler shows a better guess as to what the programmer actually wrote.

```
1570 a$=a$&"#$"&hexcon$(pc-1)
```

The # is the Motorola convention indicating immediate data and the $, of course, indicates a hexadecimal number.

```
1580 ELSE
```

If we are not dealing with immediate data, the movement can be from any general item to another.

```
1590 j=PEEK(pc+1)MOD 64
```

This makes j equal to the six bits describing the source of data.

```
1600 IF j DIV 8=1 THEN fault=1:RETurn ""
```

Address registers are not allowed as byte data sources.

```
1610 pc=pc+2
```

This time we only need to skip the two bytes of the instruction.

```
1620 a$=a$&adr$(j DIV 8,j MOD 8,pc)
```

adr$** is our clever function which will hopefully sort out all the possible addressing modes. The parameters **adr$ needs are the 3 bit addressing mode pattern from the instruction, the 3 bit register number pattern, and the current **pc**. The **pc** is needed in case the addressing mode requires extension words. If extension words are needed, the source extension comes before the destination extension, so we have to call **adr$** to decode the source before we call it to decode the destination. Of course, **adr$** must advance **pc** over any extension words it uses.

```
1630 END IF
1640 RETurn a$&","&adr$(i MOD 8,i DIV 8,pc)
1650 END DEFine
```

So, all that remains is to decode the common destination data item for both immediate and general forms of the instruction.

MOVE affects the condition code flags, clearing **V** and **C**, and setting **Z** and **N** according to the value of the item moved. The **X** flag is not affected.

dis2$** and **dis3$ are very similar to **dis1$**, except that address registers can be **MOVE**d in longword and word lengths.

```
1660 DEFine FuNction dis2$(pc)
1670 LOCal i,j,a$
1680 a$="MOVE.L  "
1690 i=PEEK(pc)*4 MOD 64+PEEK(pc+1)DIV 64
1700 IF i MOD 8=7 AND i>15 THEN fault=1:RETurn ""
1710 IF PEEK(pc+1)MOD 64=60 THEN
1720 pc=pc+6
1730 a$=a$&"#$"&hexcon$(pc-4)&hexcon$(pc-3)&hexcon$(pc-2)&hexcon$(pc-1)
```

Note how the most significant byte of any data comes first, at the lower numbered addresss. Again, this is opposite to the convention used on the Z80; also, of course, a 32 bit longword takes up four bytes.

```
1740 ELSE
1750 j=PEEK(pc+1) MOD 64
1760 pc=pc+2
1770 a$=a$&adr$(j DIV 8,j MOD 8,pc)
1780 END IF
1790 RETurn a$&","&adr$(i MOD 8,i DIV 8,pc)
1800 END DEFine
```

If the destination of **MOVE.L** is an address register, no flags are altered.

```
1810 DEFine FuNction dis3$(pc)
1820 LOCal i,j,a$
1830 a$="MOVE.W▲▲"
```

The default size is word, so we could have typed:

```
1830 a$="MOVE▲▲▲▲"
```

but it is often useful to remind yourself that **MOVE** only transfers 16 bits, so I would recommend the first version.

```
1840 i=PEEK(pc)*4 MOD 64+PEEK(pc+1) DIV 64
1850 IF i MOD 8=7 AND i>15 THEN fault=1:RETurn ""
1860 IF PEEK(pc+1) MOD 64=60 THEN
1870 pc=pc+4
1880 a$=a$&"#$"&hexcon$(pc-2)&hexcon$(pc-1)
1890 ELSE
1900 j=PEEK(pc+1) MOD 64
1910 pc=pc+2
1920 a$=a$&adr$(j DIV 8,j MOD 8,pc)
1930 END IF
1940 RETurn a$&","&adr$(i MOD 8,i DIV 8,pc)
1950 END DEFine
```

We now only need to understand the 68000 addressing modes in order to have disposed of three sixteenths of the instruction set.

```
1960 DEFine FuNction adr$(m,j,pc)
1970 LOCal k,a$
1980 type=m
1990 reg=j
2000 SELect ON type
```

Unfortunately, we have a problem. We want to **SELect** on the addressing mode **m** passed to **adr$**; but **SELect** only works on floating

point variables and, despite appearances, **m** is usually not a floating point variable. The use of **MOD** or **DIV** in the expressions passed to **adr$** by **dis1$** etc., actually created an integer result, and the parameters of functions or procedures retain the characteristics of the external parameters rather than adopting the characteristics normally associated with the form of variable name used (% for integers, **$** for strings or nothing for floating point numbers). This would be alright if **SELect** coerced the parameter to the correct type but, in the version of the QL I used, this has been overlooked, and **SELect** regards such parameters as having value zero. Consequently, we must first copy **m**, and j to proper floating point variables. Using global variables rather than **LOCal** variables for these numbers also seems to prevent a recurrence of the problem in some obscure situations, despite the transfer to proper floating point variables. It may be that the QL version you have overcomes this bug, but it is probably best to leave these lines in, in case you ever use the program on a QL with the bug.

```
2010 =0:RETurn "D"&reg
```

Addressing mode zero refers to one of the data registers D0 to D7. Note the use of coercion to form the register name from the floating point value **reg** which was, of course, coerced from the integer j passed to the function.

```
2020 =1:RETurn "A"&reg
```

Addressing mode one refers to one of the address registers A0 to A7. Remember the comments in Chapter 1, that all 32 bits of address registers are always affected when they are used as destinations, but no flags are altered. Also, there are two copies of address register A7.

```
2030 =2:RETurn "(A"&reg&")"
```

Addressing mode 2 refers to a memory location whose address is held in one of the address registers. If you are used to Z80 assembly language, you will recognise the use of brackets to indicate the phrase 'addressed by the contents of'. When a word or longword item is referenced in this way, the address must be even, or an addressing error **TRAP** will occur. The address in the register points to the most significant byte of the data item, subsequent bytes are taken from sequential addresses; so, if A6 contains $00030000 and memory at addresses $30000 to $30003 contains $01, $02, $03 and $04 respectively, then **MOVE.L (A6),D1** would put the number $01020304 into the D1 register.

ADDRESSING MODE	REGISTER NUMBER	EXTENSION WORDS	ASSEMBLER FORM
0 0 0	n	—	Dn
0 0 1	n	—	An
0 1 0	n	—	(An)
0 1 1	n	—	(An)+
1 0 0	n	—	-(An)
1 0 1	n	d_{16}	d_{16}(An)
1 1 0	n	D/A M W/L 0 0 0 d_8	d_8(An, D/A M W/L)
1 1 1	0 0 0	d_{16}	d_{16}
1 1 1	0 0 1	d_{32}	d_{32}
1 1 1	0 1 0	d_{16}	d_{16}(PC) or LABEL
1 1 1	0 1 1	D/A M W/L 0 0 0 d_8	d_8(PC, D/A M W/L) or LABEL (D/A M W/L)
1 1 1	1 0 0 THE EXACT INTERPRETATION OF THIS MODE DEPENDS ON THE REST OF THE INSTRUCTION.	0 0 0 0 0 0 0 0 d_8 or d_{16} or d_{32}	SR or CCR or #d_8 or #d_{16} or #d_{32}

TABLE 3.2 ADDRESSING MODES

```
2040 =3:RETurn "(A"&reg&")+"
```

Addressing mode 3 is very similar to mode 2, except that, after the operation, the address register is incremented by the length of the data item (1 if it is a byte, 2 if a word, 4 if a longword). This mode allows operations to be carried out on sequential items in a table without any need for separate incrementing instructions.

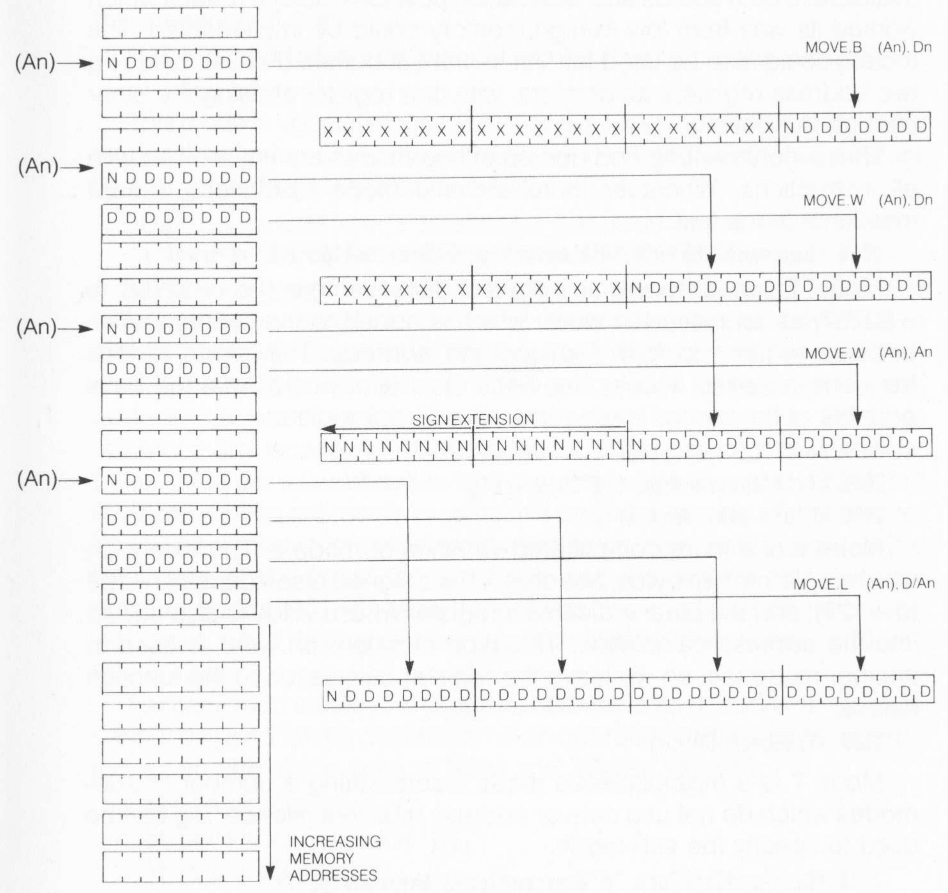

(An)
MOVE.B (An), Dn
X X X X X X X X X X X X X X X X X X X X X X X X N D D D D D D D
MOVE.W (An), Dn
X X X X X X X X X X X X X X X X N D D D D D D D D D D D D D D D
MOVE.W (An), An
SIGN EXTENSION
N N N N N N N N N N N N N N N N N D D D D D D D D D D D D D D D
MOVE.L (An), D/An
N D D D D D D D D D D D D D D D D D D D D D D D D D D D D D D D
N D D D D D D D
D D D D D D D D
INCREASING MEMORY ADDRESSES

TABLE 3.3 REGISTER LOADING EXAMPLES

```
2050 =4:RETurn "-(A"&reg&")"
```

Mode 4 is similar again, but this time the address register is decremented before the operation takes place. Modes 3 and 4 together allow the operation of a stack with any of the address registers acting as stack pointers. Conventionally, the stack starts at a high valued address and works its way down memory, so that:

```
        MOVE.L  D6,-(A5)
```

would push the four bytes of the D6 register on to the A5 stack, and:

```
        MOVE.L  (A5)+,D4
```

would pull the same bytes off again into D4; but both modes are available in both source and destination positions, so that a stack which worked its way from low to high memory could be implemented. The modes could also be used for first in first out buffers (FIFOs), by using two address registers as pointers, with one register chasing the other through the buffer.

These incrementing and decrementing modes are not allowed with all instructions. Wherever there are any mode restrictions, I shall mention it in the text.

```
2060 =5:pc=pc+2:RETurn "$"&hexcon$(pc-2)&hexcon$(pc-1)&"(A"&reg&")"
```

Mode 5 has a sixteen bit signed displacement (i.e. −32768 to +32767) as an extension word, which is added to the contents of an address register to form the operand address. This mode is thus frequently used to access the items of a data record, with the base address of the record loaded into an address register.

```
2070 =6:k=PEEK(pc):pc=pc+2
2080 a$="$"&hexcon$(pc-1)&"(A"&reg&","
2090 RETurn a$&index$(k)
```

Mode 6 is a more complicated variation of mode 5. The extension word is split into two bytes, one of which is a signed displacement (−128 to +127), and the other indicates a register whose value is also added into the address calculation. This type of extension word is used in another mode too, so we leave the register calculation to the function **index$**.

```
2100 =7:SELect ON reg
```

Mode 7 is a miscellaneous mode incorporating a number of sub-modes which do not use data or address registers. Hence, **reg** can be used to specify the sub-mode.

```
2110 =0:pc=pc+2:RETurn "$"&hexcon$(pc-2)&hexcon$(pc-1)
```

This mode is absolute short, that is, the address of the data item is given as a signed 16 bit extension word. As the number is treated as signed, it is sign extended before use as an address, and thus actually allows rapid access to the bottom 32k bytes and the top 32k bytes of memory space, namely $00000 to $07FFF and $F8000 to $FFFFF on the 68008. Note that there are no brackets around this number, despite this

mode addressing the contents of the memory location given by the number, rather than the number being used as immediate data. When a number is to be used directly as data, 68000 assemblers require it to be preceded by a # as in the **MOVE** commands listed earlier. The difference between the representation of absolute address and immediate data is so subtle that it is easy to mistake one for the other at first glance; so be wary of this possible confusion and double check for the presence of a #.

```
2120 =1:pc=pc+4:RETurn "$"&hexcon$(pc-4)&hexcon$(pc-3)&hexcon$(pc-2)&
     hexcon$(pc-1)
```

This mode is absolute long, and uses two extension words to address all possible memory locations. Note, again, that the most significant byte of a number always comes first.

```
2130 =2:pc=pc+2:RETurn "$"&hexcon$(pc-2)&hexcon$(pc-1)&"(PC)=$"&hex5$
     (pc-2+256*PEEK(pc-2)-PEEK(pc-2)DIV 128*65536+PEEK(pc-1))
```

This mode, program counter relative, is one of the most frequently used modes on the QL, as it allow routines to be located anywhere in memory without reassembly. The original intention to put all of QDOS and SuperBASIC into 32k of ROM space was based around this addressing mode, which would allow any instruction in ROM to reference any location in ROM. As the QL ROM actually occupies 48k bytes, there are a few absolute long memory references in the ROM, but most references are resolved using this program counter relative mode. The 16 bit signed extension word is used as an offset from the byte at the beginning of the extension word. We use the slightly non-standard notation **$dddd(PC)=$aaaaa** to indicate this mode which helps you work out what is happening by calculating the actual address referenced.

As mentioned earlier, the program counter relative addressing mode cannot be used for the destination of an operation as the 68008 regards this as an attempt to write to ROM and **TRAP**s the error.

```
2140 =3:k=PEEK(pc):pc=pc+2
2150 a$="$"&hexcon$(pc-1)&"(PC,"
2160 RETurn a$&index$(k)
```

This mode is rather like mode 6, allowing an 8 bit signed displacement and a register to be added to the program counter to form the required address. Again, PC refers to the byte at the start of the extension word, and the mode cannot be used for the destination of an operation.

```
2170 =REMAINDER :fault=1:RETurn ""
```

Actually, as we saw earlier, **type**=7 and **reg**=4, often refers to an immediate data operand; but it also sometimes refers to the status register SR. So, the decoding of that mode will be done outside **adr$**. All other modes are illegal on the 68ØØ8, though some are used for the extended modes of the 68Ø2Ø.

```
2180 END SELect
2190 END SELect
2200 END DEFine
```

completes the function **adr$**, apart from the full interpretation of the indexed addressing modes.

```
2210 DEFine FuNction index$(k)
2220 LOCal a$
2230 IF k<128 THEN
2240 a$="D"
2250 ELSE
2260 a$="A"
2270 END IF
```

The register used for indexing can be either a data or address register, depending on the most significant bit of the extension word.

```
2280 a$=a$&(k MOD 128 DIV 16)
```

The next three bits contain the register number.

```
2290 IF k MOD 8<>0 THEN fault=1:RETurn ""
```

The last three bits must always be zero.

```
2300 IF k MOD 16=0 THEN
2310 RETurn a$&".W)"
2320 END IF
2330 RETurn a$&".L)"
2340 END DEFine
```

The remaining bit allows us to cope with the case when there are only 16 bits of valid data in a data register. Indeed, the size word is assumed if no size extension is given. As usual, word data is sign extended to 32 bits, as is the 8 bit index offset, before they are both added to the address register or program counter coded into the instruction word.

You might think that such a complicated indexed addressing mode is a powerful but rarely used mode, a challenge to the mind of a devious programmer; but the QL uses a number of complex task and channel tables and, to access them, this mode is very useful. It is also a useful mode in conjunction with the load effective address command **LEA**, when it allows complex addition of data to an address register, as we shall see later.

Chapter 4 Some Immediate Data Commands, Bit Tests and Peripheral Data Transfers

Chapter 3 dealt with the 68ØØØ instructions slightly out of numerical order. We now return to an approximation of numeric order to deal with those commands which begin with the hexadecimal digit zero.

```
2350 DEFine FuNction dis0$(pc)
2360 LOCal i,a$
2370 i=PEEK(pc)
```

Much of this group of instructions is concerned with operations involving immediate data. We saw how to deal with immediate data in the last chapter, so we select those commands first.

```
2380 IF i MOD 2=0 AND i<>8 THEN
2390 IF i DIV 4 MOD 2=1 THEN IF PEEK(pc+1) MOD 64=60 THEN fault=1:RET
   urn ""
```

This line excludes the possibility of adding or subtracting data to or from the status register, a slightly nonsensical thing to do. Apart from this exception, the data sizes and parameters are all coded in the same way, so we will use a function to decode them.

```
2400 a$=imm$(pc):IF fault THEN RETurn ""
```

The value **i** determines which command is involved, as follows:

```
2410 SELect ON i
2420 =0:RETurn "OR"&a$(1 TO 2)&" "&a$(3 TO)
```

The mnemonic **OR** is one letter shorter than the other mnemonics in this range. The first two characters of **a$** hold the size extension (".B", ".W" or ".L"), so an extra space is introduced to correctly align the output.

OR itself performs a bitwise OR of the immediate data and the destination data item. Thus **OR** allows you to set any required pattern of bits in the destination item. The instruction clears the **V** and **C** condition codes and sets **N** and **Z** according to the result of the operation, while leaving **X** alone. This range of immediate commands is not allowed to affect address registers, so there is no matching comment about the non-setting of flags for address register results.

```
2430 =2:RETurn "AND"&a$
```

AND is the opposite of **OR** in that it allows any pattern of bits in a destination data item to be cleared, by performing a bitwise AND of the immediate data and the destination data item. The same range of flags are set according to the result as are done by **OR**.

OR.size # data, destination

0	0	0	0	0	0	0	0	SIZE	DESTINATION ADDRESSING MODE	DESTINATION REGISTER NUMBER

SIZE:			
	0	0	= .B
	0	1	= .W
	1	0	= .L

AND.size # data, destination

0	0	0	0	0	0	1	0	SIZE	DESTINATION ADDRESSING MODE	DESTINATION REGISTER NUMBER

SUB.size # data, destination

0	0	0	0	0	1	0	0	SIZE	DESTINATION ADDRESSING MODE	DESTINATION REGISTER NUMBER

ADD.size # data, destination

0	0	0	0	0	1	1	0	SIZE	DESTINATION ADDRESSING MODE	DESTINATION REGISTER NUMBER

EOR.size # data, destination

0	0	0	0	1	0	1	0	SIZE	DESTINATION ADDRESSING MODE	DESTINATION REGISTER NUMBER

CMP.size # data, destination

0	0	0	0	1	1	0	0	SIZE	DESTINATION ADDRESSING MODE	DESTINATION REGISTER NUMBER

TABLE 4.1 IMMEDIATE DATA COMMANDS

```
2440 =4:RETurn "SUB"&a$
```

SUB subtracts the immediate data from the destination item. All the flags **C**, **V**, **Z**, **N** and **X** are set according to the result. Subsequent tests determine whether you consider the items involved to be signed or unsigned, or possibly the least significant item in an extended, multi-item, number.

```
2450 =6:RETurn "ADD"&a$
```

ADD is the opposite of **SUB**, adding the immediate data to the destination item and setting all flags according to the result.

```
2460 =10:RETurn "EOR"&a$
```

EOR performs a bitwise Exclusive OR of the immediate data with the destination item, thus inverting all those bits in the destination item with matching bits set in the immediate data. The **X** flag is unaffected by this operation, **C** and **V** are cleared, **N** and **Z** are set according to the result.

```
2470 =12:RETurn "CMP"&a$
```

CMP (CoMPare) is similar to **SUB** and sets the condition code flags in the same way, apart from **X** which is unaffected, but does not load the result into the destination.

```
2480 =REMAINDER :fault=1:RETurn ""
2490 END SELect
2500 END IF
```

The next group of instructions we shall deal with is designed specially to help with the transfer of data to and from peripherals. Motorola have always chosen to locate peripherals at normal memory locations, thus removing the need for special **IN** and **OUT** commands. Indeed, the normal **MOVE** command is all that is needed to perform input and output through peripheral devices. However, the 68000 family does have a special command **MOVEP** for dealing with peripherals. The command is not particularly useful on the 68008; it is designed to overcome a problem with the 68000 and its 16 bit data bus. Suppose you attach an 8 bit peripheral chip to a 16 bit data bus. You may wish to transfer the bytes of a word or a longword to sequential ports on the peripheral chip, but because of the address numbering scheme these ports are actually located at alternate addresses in memory space. Also, depending on whether the peripheral is wired to the high or low order byte of the bus, the addresses can be even or odd, so that the usual rules on only accessing even addresses are relaxed. On a 68008, a designer may wire a peripheral to alternate addresses to maintain

MOVEP.W/L d$_{16}$(As), Dd

0	0	0	0	d	1	0	W/L	0	0	1	s

MOVEP.W/L Ds, d$_{16}$(Ad)

0	0	0	0	s	1	1	W/L	0	0	1	d

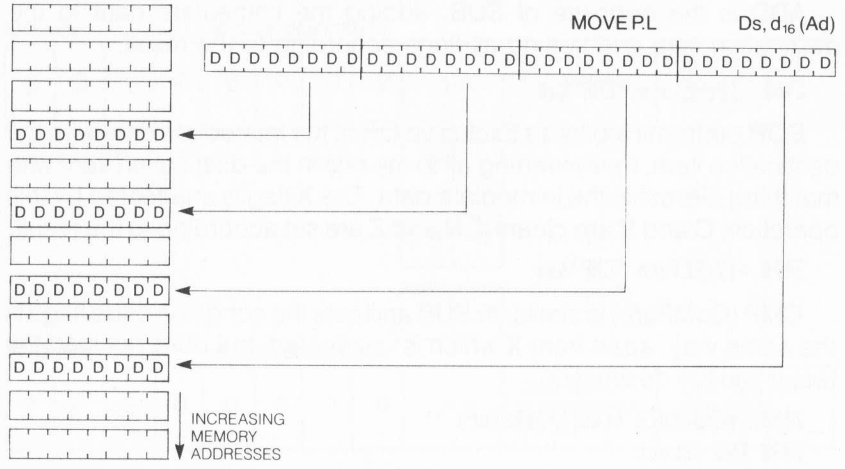


TABLE 4.2 PERIPHERAL DATA MOVEMENT

software compatability with a 16 bit bus 68000 or 68010 system. On the QL, no such devices have been wired and **MOVEP** is, therefore, not very useful, although it could do some devious things. However we shall still decode it.

```
2510 IF PEEK(pc+1) DIV 8 MOD 8=1 THEN
2520 IF i=8 THEN fault=1:RETurn ""
2530 a$="MOVEP"
2540 IF PEEK(pc+1) DIV 64 MOD 2=0 THEN
2550 a$=a$&".W␣"
2560 ELSE
2570 a$=a$&".L␣"
2580 END IF
2590 pc=pc+4
2600 IF PEEK(pc-3) DIV 128=0 THEN
```

```
2610 RETurn a$&"$"&hexcon$(pc-2)&hexcon$(pc-1)&"(A"&(PEEK(pc-3)MOD 8)
     &"),D"&(PEEK(pc-4)DIV 2)
2620 END IF
2630 RETurn a$&"D"&(PEEK(pc-4)DIV 2)&",$"&hexcon$(pc-2)&hexcon$(pc-1)
     &"(A"&(PEEK(pc-3) MOD 8)&")"
2640 END IF
```

Note that the peripheral is always addressed by a 16 bit offset from the contents of an address register and the word or longword of data comes from or is returned to a data register. No flags are affected by this instruction.

The remaining commands in this section address and test or change individual bits in memory.

```
2650 i=PEEK(pc+1) DIV 64
2660 IF PEEK(pc+1) MOD 64>=58 THEN fault=1:RETurn ""
```

This line rejects program counter relative addresses, as it is not sensible to want to change bits in ROM. The 68008 will **TRAP** any such attempt as an invalid instruction.

```
2670 SELect ON i
2680 =0:a$="BTST"
```

BTST simply tests an addressed bit and sets the **Z** flag according to its value; other flags are left untouched.

```
2690 =1:a$="BCHG"
```

BCHG first tests the addressed bit, then changes its value to the opposite condition.

```
2700 =2:a$="BCLR"
```

BCLR tests the bit, then clears its value to zero.

```
2710 =3:a$="BSET"
```

BSET tests the bit, then sets its value to one.

```
2720 END SELect
2730 i=PEEK(pc+1) MOD 64
2740 a$=a$&"␣␣␣␣"
```

The bit to be affected is addressed using the normal addressing modes discussed in the previous chapter. In addition, the bit is identified by its number, which is given either as immediate data or the contents of a data register. If the item addressed is in memory space, it is of size byte and the bit number ranges from 0 to 7; outside this range, the actual bit addressed is calculated by taking the given number modulo 8. If the item addressed is a data register, bit numbers are allowed in the range Ø to 31 and, if outside this range, the number is taken modulo 32. Address registers are not valid destinations for bit tests.

```
2750 IF i DIV 8=1 THEN fault=1:RETurn ""
2760 IF PEEK(pc)=8 THEN
2770 IF PEEK(pc+2)<>0 THEN fault=1:RETurn ""
```

Assemblers cannot generate a non-zero first byte for the immediate data, if that is used as the bit number, so we reject any such faults.

```
2780 a$=a$&"#"&PEEK(pc+3)
```

We use coercion to show the decimal bit number in the mnemonic form of the command.

```
2790 pc=pc+4
2800 ELSE
2810 a$=a$&"D"&(PEEK(pc)DIV 2)
2820 pc=pc+2
2830 END IF
2840 RETurn a$&","&adr$(i DIV 8,i MOD 8,pc)
2850 END DEFine
```

Bop # number, destination

Ø	Ø	Ø	Ø	1	Ø	Ø	Ø	op	DESTINATION ADDRESSING MODE	DESTINATION REGISTER NUMBER

Bop Ds, destination

Ø	Ø	Ø	Ø	s	1	op	DESTINATION ADDRESSING MODE	DESTINATION REGISTER NUMBER

op:			
	Ø	Ø	= TST
	Ø	1	= CHG
	1	Ø	= CLR
	1	1	= SET

TABLE 4.3 BIT OPERATIONS

As mentioned earlier, the byte of memory or the data register is identified using the normal addressing modes decoded by **adr$**.

All that remains to do, is to clear up the details of the immediate data commands covered at the start of this chapter and, in particular, the function **imm$**.

```
2860 DEFine FuNction imm$(pc)
2870 LOCal i,j,a$
2880 i=PEEK(pc+1) MOD 64
2890 j=PEEK(pc+1) DIV 64
2900 SELect ON j
2910 =0:a$=".B▲▲▲#$"&hexcon$(pc+3);pc=pc+4
2920 IF PEEK(pc-2)<>0 AND PEEK(pc-2)<>255 THEN fault=1:RETurn ""
2930 =1:a$=".W▲▲▲#$"&hexcon$(pc+2)&hexcon$(pc+3)
2940 pc=pc+4
2950 =2:a$=".L▲▲▲#$"&hexcon$(pc+2)&hexcon$(pc+3)&hexcon$(pc+4)&hexcon
     $(pc+5)
2960 pc=pc+6
2970 =3:fault=1:RETurn ""
2980 END SELect
2990 IF i=60 THEN RETurn a$&",SR"
```

It is possible for **OR**, **AND** and **EOR** with immediate data to take the status register as destination, in which case the earlier comments about the resultant setting of flags are superseded by the direct effects of the instruction. Byte sized operations simply affect the condition code register. Word sized operations may affect all the status register and are thus privileged, that is, they are only allowed if the supervisor bit is set before the instruction starts. The instruction may clear the supervisor bit and, in this case, all the changes associated with changing mode take place, privileged instructions become restricted and the user stack pointer becomes the current A7 register.

```
3000 IF i DIV 8=1 OR i>=58 THEN fault=1:RETurn ""
```

Immediate operations are not allowed on address registers or program counter relative addresses, otherwise all addressing modes are allowed.

```
3010 RETurn a$&","&adr$(i DIV 8,i MOD 8,pc)
3020 END DEFine
```