Chapter 5
Sundry Unary Operations

In Chapter 2 we rapidly disposed of the two sixteenths of the 68000 instruction set reserved for use with co-processors. Chapter 3 covered another three sixteenths of the instruction set and all the addressing modes, while Chapter 4 covered just one sixteenth of the instruction set. The next sixteenth is so full of interesting instructions that we shall split its discussion across the next two chapters.

```
3030 DEFine FuNction dis4$(pc)
3040 LOCal i,j,k,a$,b$
3050 IF PEEK(pc)MOD 2=1 THEN
3060 i=PEEK(pc)DIV 2 MOD 8
3070 IF PEEK(pc+1)<192 THEN
3080 a$="CHK▲▲▲▲▲"
```

One of the easiest instructions to pluck out of the mass in **dis4$** is **CHK**. The mnemonic stands for check and is designed to allow rapid checking of the contents of a data register against both upper and lower bounds. The QL uses **CHK** to test array bound limits and string length limits in SuperBASIC. The comparison is always with the least significant 16 bits of a data register, treated as a signed number. If this number is negative, or greater than the data source, then a **TRAP** is invoked.

```
3090 IF PEEK(pc+1)=188 THEN
3100 pc=pc+4
3110 RETurn a$&"#$"&hexcon$(pc-2)&hexcon$(pc-1)&",D"&i
3120 END IF
```

CHK has an immediate data version and a more general version which allows any normal addressing mode, apart from the contents of an address register, as the upper bound data source. Flag settings are undefined after using **CHK**, except that **X** is unaltered.

```
3130 pc=pc+2
3140 j=PEEK(pc-1)MOD 64
3150 IF PEEK(pc-1)<128 OR j DIV 8=1 THEN fault=1:RETurn ""
```

```
3160 RETurn a$&adr$(j DIV 8,j MOD 8,pc)&",D"&i
3170 END IF
3180 pc=pc+2
3190 j=PEEK(pc-1)MOD 64
3200 IF j<=15 OR (j)=24 AND j(=39) THEN fault=1:RETurn ""
3210 RETurn "LEA▲▲▲▲▲"&adr$(j DIV 8,j MOD 8,pc)&",A"&i
3220 END IF
```

LEA stands for load effective address. Rather than dealing with the data item identified by its source addressing mode, the address formed by the source addressing mode is loaded into the full 32 bits of an address register. Obviously, there is no way of describing the address of a data or address register. Equally, the postincrement and predecrement modes are not allowed. **LEA** helps us to get round the problem of being unable to alter items addressed using program counter relative addressing. For instance, we could write:

```
LEA     $76(PC),A1
MOVE.W  $FFFF,(A1)
```

while we could not write:

```
MOVE.W  $FFFF,$76(PC)
```

without generating an error. **LEA** is also useful as a variant of the **ADD** instruction, allowing complex additions to an address register, such as:

```
LEA     $76(A2,D6.W),A1
```

which adds $76 to A2 to the sign extended low word of D6, and places the result in A1.

As might be expected, the use of **LEA** does not alter any of the condition code flags.

The rest of **dis4$** is rather more complicated:

```
3230 i=PEEK(pc) MOD 16
3240 pc=pc+2
3250 j=PEEK(pc-1) DIV 64
3260 k=PEEK(pc-1) MOD 64
3270 IF i(14 AND k DIV 8=1 THEN fault=1:RETurn ""
3280 SELect ON i
3290 =0:SELect ON j
3300 =0:a$="NEGX.B▲▲"
```

NEGX negates an item directly, also subtracting the value of the extended arithmetic bit **X**. All the condition codes are set according to the result, except for **Z**. **Z** is cleared if the result is non-zero, but left unchanged if the result was zero. This allows you to set the **Z** bit before

starting a series of **NEGX** on an extended number; testing **Z** afterwards determines if all the items were zero.

NEGX does not operate on address registers or through program counter relative addressing modes, in the same way as many of the commands in **dis4$**.

CHK source, Dd

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	d				1	1	0	SOURCE ADDRESSING MODE			SOURCE REGISTER NUMBER	

LEA source, Ad

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	d				1	1	1	SOURCE ADDRESSING MODE			SOURCE REGISTER NUMBER	

NEGX.size destination

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	0	SIZE			DESTINATION ADDRESSING MODE			DESTINATION REGISTER NUMBER	

CLR.size destination

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	1	0	SIZE			DESTINATION ADDRESSING MODE			DESTINATION REGISTER NUMBER	

NEG.size destination

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	SIZE			DESTINATION ADDRESSING MODE			DESTINATION REGISTER NUMBER	

NOT.size destination

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	1	0	SIZE			DESTINATION ADDRESSING MODE			DESTINATION REGISTER NUMBER	

MOVE SR, destination

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	0	1	1		DESTINATION ADDRESSING MODE			DESTINATION REGISTER NUMBER	

MOVE source, CCR/SR

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	CCR/SR	0	1	1		SOURCE ADDRESSING MODE			SOURCE REGISTER NUMBER	

TABLE 5.1 SUNDRY INSTRUCTIONS IN dis4$

```
3310 =1:a$="NEGX.W  "
3320 =2:a$="NEGX.L  "
3330 =3:a$="MOVE  SR,"
3340 END SELect
3350 IF k)=58 THEN fault=1:RETurn ""
3360 RETurn a$&adr$( k DIV 8,k MOD 8,pc)
```

The **MOVE SR** variant of the **MOVE** command, transfers the 16 bits of the status register to the addressed item, without altering the status flags. Perhaps,

```
MOVE    SR,-(A7)
```

is the most useful variant of this instruction, pushing the status register on to the A7 stack, where it will be available for subsequent restoration.

```
3370 =2:SELect ON j
3380 =0:a$="CLR.B"
3390 =1:a$="CLR.W"
3400 =2:a$="CLR.L"
3410 =3:fault=1:RETurn ""
3420 END SELect
3430 IF k)=58 THEN fault=1:RETurn ""
3440 RETurn a$&"  "&adr$(k DIV 8,k MOD 8,pc)
```

CLR clears all the bits of the addressed item to zero; but, being part of a section of the 68000 instruction set designed to first read an item from memory, then act on its value before storing the result back in the original location, **CLR** reads the item before clearing it. This does not normally cause any problems, but if the address is that of a peripheral port, reading the port may trigger some undesirable action by the peripheral. In such a case, an immediate data **MOVE** of zero, or the use of the **SF** (set false) instruction which we shall meet later, would be more appropriate.

CLR cannot alter address registers or program counter relative address items. It clears the **C**, **V** and **N** flags and sets the **Z** flag while leaving **X** unaffected.

```
3450 =4:SELect ON j
3460 =3:IF k=60 THEN
3470 IF PEEK(pc)<>0 THEN fault=1:RETurn ""
3480 pc=pc+2
3490 a$="#$"&hexcon$(pc-1)
3500 ELSE
3510 a$=adr$(k DIV 8,k MOD 8,pc)
3520 END IF
3530 RETurn "MOVE  "&a$&",CCR"
```


This variant of **MOVE** can be used to set up any required pattern of condition code bits in the status register. Despite the destination being of size byte, the source is treated as a word item, so that the instruction properly reverses the only way of storing the condition codes, which we saw earlier, was as part of the 16 bit status word. This command directly affects the condition codes.

```
3540 =0:a$="NEG.B"
3550 =1:a$="NEG.W"
3560 =2:a$="NEG.L"
3570 END SELect
3580 IF k)=58 THEN fault=1:RETurn ""
3590 RETurn a$&"  "&adr$(k DIV 8,k MOD 8,pc)
```

NEG means negate and is very similar to **NEGX** in that it constructs the negative of the original number, unaffected by the value of the **X** bit, but it sets all the condition code flags including **X** and **Z**. It could thus be used to negate the least significant item of an extended number, whose higher order elements are then negated using **NEGX**. A simple definition of a negative which may be useful to remember is 'that number which when added to the original gives zero as a result (ignoring any carry)'.

```
3600 =6:SELect ON j
3610 =3:IF k=60 THEN
3620 a$="#$"&hexcon$(pc)&hexcon$(pc+1)
3630 pc=pc+2
3640 ELSE
3650 a$=adr$(k DIV 8,k MOD 8,pc)
3660 END IF
3670 RETurn "MOVE  "&a$&",SR"
```

This variant of **MOVE** can alter all 16 bits of the status register, including the supervisor bit. It is thus a privileged instruction, and only allowed if the supervisor bit is set before the instruction starts. It is common for a supervisor program which is about to initiate a user program to use this command to clear the supervisor bit and thus switch modes.

```
3680 =0:a$="NOT.B"
3690 =1:a$="NOT.W"
3700 =2:a$="NOT.L"
3710 END SELect
3720 IF k)=58 THEN fault=1:RETurn ""
3730 RETurn a$&"  "&adr$(k DIV 8,k MOD 8,pc)
```

NOT inverts every bit of the addressed item, clears the **C** and **V** bits, sets **N** and **Z** according to the result, and leaves **X** unaffected.

```
3740 =8:SELect ON j
3750 =0:IF k>=58 THEN fault=1:RETurn ""
3760 RETurn "NBCD    "&adr$(k DIV 8,k MOD 8,pc)
```

NBCD is very much like **NEGX.B** in the way it works, but the key difference is that it deals with binary coded decimal numbers. Binary coded decimal numbers are a useful alternative way of representing numbers inside the computer; each byte of a BCD number is used to contain two four bit digits, which can only take values 0 to 9. BCD is more accurate than the normal binary representation of numbers, in that it is nearer to the form in which numbers are normally displayed. Thus 1/10, which is a never ending recurring number in its binary representation, can be accurately represented in BCD notation, and additions and subtractions do not compound the error introduced by the limited length of the binary representation. BCD is thus a better notation if you are only going to involve yourself in adding and subtracting, but its advantages disappear if you want to become involved with multiplication and division. In some high level languages it is possible to specify the use of BCD arithmetic inside the computer in order to maintain the desired type of accuracy, but the QL ROM does not use BCD arithmetic.

NBCD produces a result formed by subtracting the value of the addressed byte and the value of the **X** bit from 100 in BCD arithmetic. The **C** and **X** bits are set according to the result, **Z** is cleared if the result is non-zero, but unaffected if the result is zero, and **N** and **V** are undefined and meaningless.

```
3770 =1:IF k<8 THEN RETurn "SWAP    D"&k
```

SWAP switches around the two 16 bit halves of a data register. This is particularly useful in conjunction with the multiply and divide instructions we shall meet later, but it also allows the top 16 bits of a data register to be used as a fast temporary storage location. The **C** and **V** bits are cleared by the instruction, **Z** is set if all 32 bits of the register are zero and is cleared otherwise, **N** is set according to the value of the most significant bit of the result. **X** is unaffected by the operation.

```
3780 IF k<=15 OR (k)=24 AND k<=39) THEN fault=1:RETurn ""
3790 RETurn "PEA     "&adr$(k DIV 8,k MOD 8,pc)
```

PEA stands for push effective address and like **LEA** it calculates an address then uses that as data, rather than the item it addresses.

PEA uses the A7 stack, and could thus be considered as:

```
LEA     address,-(A7)
```


NBCD										destination	
0	1	0	0	1	0	0	0	0	0	DESTINATION ADDRESSING MODE	DESTINATION REGISTER NUMBER

SWAP													Dd
0	1	0	0	1	0	0	0	0	1	0	0	0	d

PEA									source	
0	1	0	0	1	0	0	0	0	1	SOURCE ADDRESSING MODE	SOURCE REGISTER NUMBER

EXT.W/L													Dd
0	1	0	0	1	0	0	0	1	W/L	0	0	0	d

TST.size									destination	
0	1	0	0	1	0	1	0	SIZE	DESTINATION ADDRESSING MODE	DESTINATION REGISTER NUMBER

TAS										destination	
0	1	0	0	1	0	1	0	1	1	DESTINATION ADDRESSING MODE	DESTINATION REGISTER NUMBER

TABLE 5.2 SUNDRY INSTRUCTIONS IN dis4$ (CONT.)

Predecrement and postincrement modes are not allowed for the address, but program counter relative modes are allowed. Four bytes of data are always pushed, and no condition codes are affected.

```
3800 =2:SELect ON k
3810 =0 TO 7:RETurn "EXT.W    D"&k
```

EXT.W sign extends the low byte of a data register to make a 16 bit word with the same signed value. This is done by copying bit 7 of the data register into bits 8 to 15. If you have been doing some byte sized data operations and want the result to affect a word sized item, you often need the size of the two operands to match and **EXT** is needed. **EXT** clears the **C** and **V** flags and sets **N** and **Z** appropriately while leaving **X** unaffected.

```
3820 =32 TO 39:RETurn "MOVEM.W  "&regmaskpredec$(pc)&",-(A"&(k MOD 8)&
     ")"
3830 =16 TO 23,40 TO 57:RETurn "MOVEM.W  "&regmaskpostinc$(pc)&","&adr
     $(k DIV 8,k MOD 8,pc)
3840 =REMAINDER :fault=1:RETurn ""
3850 END SELect
```

MOVEM stands for move multiple. It is designed to save the contents of a number of registers in a single construction. Subroutines often want to return with registers unaffected, but need to use the registers, so they need to save and restore a number of registers. An extension word in the **MOVEM** instruction allows any combination of the 16 current registers to be saved or reloaded. The actual bit pattern in the extension word is different if the registers are pushed on to a stack using the predecrement mode to the pattern used with other addressing modes when the registers are saved at increasing addresses. The registers are always saved so that D0 occupies a lower address than D1, then D2 to D7 and A0 to A7, except that an unsaved register does not reserve any space in memory. The actual register mask is decoded by functions **regmaskpostinc$** and **regmaskpredec$**. Since **regmaskpostinc$** advances **pc** by 2, moving over the extension word, it is essential that line 3830 is evaluated from left to right for **adr$** to pick up the appropriate extension word if it needs one.

Note that postincrement and program counter relative addressing modes are not allowed, and no flags are affected by the instruction.

```
3860 =3:SELect ON k
3870 =0 TO 7:RETurn "EXT.L   D"&k
```

MOVEM. W/L register-list, –(Ad)

0	1	0	0	1	0	0	0	1	W/L	1	0	0	d		
D0	D1	D2	D3	D4	D5	D6	D7	A0	A1	A2	A3	A4	A5	A6	A7

MOVEM. W/L register-list, destination

0	1	0	0	1	0	0	0	1	W/L	DESTINATION ADDRESSING MODE		DESTINATION REGISTER NUMBER			
A7	A6	A5	A4	A3	A2	A1	A0	D7	D6	D5	D4	D3	D2	D1	D0

MOVEM. W/L source, register-list

0	1	0	0	1	1	0	0	1	W/L	SOURCE ADDRESSING MODE		SOURCE REGISTER NUMBER			
A7	A6	A5	A4	A3	A2	A1	A0	D7	D6	D5	D4	D3	D2	D1	D0

TABLE 5.3 MOVEM INSTRUCTIONS

EXT.L sign extends a 16 bit word in a data register to the full 32 bits in a similar manner to **EXT.W**. To sign extend a byte to the full 32 bits, you thus need to use **EXT.W** followed by **EXT.L**.

```
3880 =32 TO 39:RETurn "MOVEM.L  "&regmaskpredec$(pc)&",-(A"&(k MOD 8)&
     ")"
3890 =16 TO 23,40 TO 57:RETurn "MOVEM.L  "&regmaskpostinc$(pc)&","&adr
     $(k DIV 8,k MOD 8,pc)
3900 =REMAINDER :fault=1:RETurn ""
3910 END SELect
3920 END SELect
```

MOVEM.L is identical to **MOVEM.W** except that it saves all 32 bits of the registers involved rather than just the 16 bits saved by **MOVEM.W.** It is, of course, at this point that we realise that it might have been more efficient to combine the **ON j=2** and **ON j=3** selections, as they were identifical apart from the ".L" in place of the ".W" specifying the data item sizes.

```
3930 =10:SELect ON j
3940 =0:a$="TST.B"
```

TST tests a data item, and sets the **N** and **Z** flags accordingly. The **C** and **V** flags are cleared, and **X** is unaffected.

```
3950 =1:a$="TST.W"
3960 =2:a$="TST.L"
3970 =3:a$="TAS  "
```

TAS tests a byte sized data item, sets the flags according to its value, and then in an indivisible operation sets the sign bit of the byte to one. **TAS** constantly selects the appropriate RAM address throughout the read, modify, write cycle, so that if the RAM is being shared by another processor, its contents can neither be examined nor changed by the other processor during the cycle. **TAS** can thus be used to operate semaphores where two or more processors sharing some memory or peripheral can indicate to the others that a device is in use by setting this flag in an agreed byte. Equally, **TAS** can be used by one program to indicate to other programs on the same processor that a resource is in use, though this function could be achieved using **BSET**. **BSET** cannot be used with multiple processors, because there is a period between reading the bit and setting it when another processor could access the RAM and either read the value which was about to change or change the value which had just been tested, resulting in two processors subsequently accessing the same resource in a conflicting manner. QL owners need not worry about this problem.

```
3980 END SELect
3990 IF k>=58 THEN fault=1:RETurn ""
4000 RETurn a$&"␣␣␣"&adr$(k DIV 8,k MOD 8,pc)
4010 =12:IF j<2 OR k<16 OR k DIV 8=4 THEN fault=1:RETurn ""
4020 a$=regmaskpostinc$(pc)
4030 IF j=2 THEN
4040 b$="W␣"
4050 ELSE
4060 b$="L␣"
4070 END IF
4080 RETurn "MOVEM."&b$&adr$(k DIV 8,k MOD 8,pc)&","&a$
```

This adds the missing part of the **MOVEM** command which restores a set of registers from memory. As the registers are always read from increasing addresses during restoration, the same bit pattern is used in the register list extension word. This extension word always directly follows the instruction word, despite it referring to the destination for the data. If the register source is addressed using a program counter relative mode, the value of the program counter added into the address calculation is the start of the addressing mode extension word, four bytes after the start of the instruction. Predecrement addressing modes are obviously not allowed, and the command does not affect any flags.

```
4090 =14:RETurn dis4E$(pc,j,k)
4100 END SELect
4110 END DEFine
```

We will leave the commands beginning with $4E until the next chapter and, in the meantime, clear up the register list decoding for the **MOVEM** commands.

You have seen that when a static address, one that is neither predecrementing nor postincrementing, is used, the function **regmaskpostinc$** is applied for both the saving and restoring of registers. This is because the address supplied is taken as the lowest address of a block of memory, and incrementation takes place inside the 68008, though the result of incrementing is not put back in the address source, thus:

```
MOVEM.L D0-D7,(A6)
```

will save all the data registers in a 32 byte block of memory addressed by A6, and as A6 is not changed by the operation,

```
MOVEM.L (A6),D0-D7
```

will subsequently restore the contents.

The way to remember the extension word bit patterns for **regmaskpredec$** and **regmaskpostinc$** is to think of the extension word

being examined by the 68008 bit by bit starting at the least significant bit, and also to remember that D0 ends up stored in the lowest numbered address and A7 in the highest numbered.

```
4120 DEFine FuNction regmaskpostinc$(pc)
4130 LOCal a$,d$,i,j
4140 a$=""
4150 j=PEEK(pc)
4160 FOR i=0 TO 7
4170 IF j DIV 2^i MOD 2=1 THEN a$=a$&"A"&i
4180 END FOR i
4190 compress a$
```

As you saw in the examples above, the programmer is allowed to miss out register names by putting a hyphen between the names of the first and last registers in a range. **compress** achieves this alteration, and also adds "/" between other register names in the manner usually required by assemblers.

```
4200 d$=""
4210 pc=pc+2
4220 j=PEEK(pc-1)
4230 FOR i=0 TO 7
4240 IF j DIV 2^i MOD 2=1 THEN d$=d$&"D"&i
4250 END FOR i
4260 compress d$
4270 RETurn combine$(d$,a$)
4280 END DEFine
4290 DEFine FuNction regmaskpredec$(pc)
4300 LOCal a$,d$,i,j
4310 d$=""
4320 j=PEEK(pc)
4330 FOR i=0 TO 7
4340 IF j*2^i DIV 128 MOD 2=1 THEN d$=d$&"D"&i
4350 END FOR i
4360 compress d$
4370 a$=""
4380 pc=pc+2
4390 j=PEEK(pc-1)
4400 FOR i=0 TO 7
4410 IF j*2^i DIV 128 MOD 2=1 THEN a$=a$&"A"&i
4420 END FOR i
4430 compress a$
4440 RETurn combine$(d$,a$)
```

```
4450 END DEFine
4460 DEFine PROCedure compress(a$)
4470 LOCal i,j
4480 i=LEN(a$)
4490 SELect ON i
```

Short strings are relatively easy to **compress**.

```
4500 =0 TO 2:RETurn
```

Indeed, there is no shorter representation of zero or one registers.

```
4510 =4:a$=a$(1 TO 2)&"/"&a$(3 TO 4):RETurn
```

Two register names can always be separated by a "/".

```
4520 =REMAINDER
```

Longer register lists are the problem. We split the problem into two stages, first marking where compression is possible by crossing out the "A" or "D" with a "–".

```
4530 FOR j=2 TO i-4 STEP 2
4540 IF a$(j)+1=a$(j+2) THEN
```

Coercion is a really useful technique, as long as you can understand what is happening. Here we are checking that adjacent registers in **a$** have successive numbers.

```
4550 IF a$(j)+2=a$(j+4) THEN a$(j+1)="-"
4560 END IF
4570 END FOR j
```

Now we will scan through the string looking for hyphens and removing the register names in the middle of ranges. The first hyphen may be in the third position in the string.

```
4580 j=3
4590 REPeat juggle
```

As yet, I am unsure how many times we shall need to repeat this next section, so we use a **REPeat** loop.

```
4600 IF j>LEN(a$) THEN RETurn
```

As the length of **a$** is going to vary during the course of this loop, we cannot check **j** against the fixed number **i**. However, we know that we have finished when all the string has been scanned.

```
4610 IF a$(j)<>"-" THEN a$=a$(1 TO j-1)&"/"&a$(j TO):j=j+3:NEXT juggle
```

If the scanned character is not a hyphen then no compression can take place. Indeed, we need to add a "/" to separate the two register names, and advance **j** to the next potential hyphen before restarting the loop.


```
4620 IF a$(j+2)="-" THEN a$=a$(1 TO j-1)&a$(j+2 TO):NEXT juggle
```

If there are two consecutive hyphens, we can remove the two characters **a$(j)** and **a$(j+1)** and **j** will already point to the next register letter (we know this is a hyphen, but we have to recheck some items, so we start the loop again).

```
4630 a$=a$(1 TO j)&a$(j+2 TO)
4640 j=j+3
4650 END REPeat juggle
4660 END SELect
4670 END DEFine
```

This final tidying up copes with an isolated hyphen, removing the excess register number and advancing **j** to point to the next register letter.

It is conventional to show the list of data registers before the list of address registers because they occupy the lower memory addresses, so **combine$** adds the two lists together in that order.

```
4680 DEFine FuNction combine$(d$,a$)
4690 IF d$="" THEN
4700 IF a$="" THEN fault=1:RETurn ""
4710 RETurn a$
4720 END IF
4730 IF a$="" THEN RETurn d$
4740 RETurn d$&"/"&a$
4750 END DEFine
```

Chapter 6
Interrupts, TRAPs, Subroutines and Jumps

We have so far only hinted at the **TRAP** mechanism which diverts the 68008 from the normal program flow when an error occurs. Now is the time to go into the concept of **TRAP**s and exceptions in much greater details, as many of the commands with a first byte of $4E involve more **TRAP**s.

Nominally, the first kilobyte of the 68008 address space is allocated to exception vectors, that is, a list of addresses to which the program jumps when an error, **TRAP**, interrupt, exception, or whatever else you like to call a diversion from the processor's normal task, occurs. Not all the exceptions allocated by Motorola are possible on the QL, nor are all the possible exceptions catered for.

We make a short diversion from the dissembler itself to discuss the exception vectors in turn. Each vector is four bytes long and has a vector number which when multiplied by four gives the address in memory where the vector is held.

Vectors 0 and 1: Addresses 0 to 7 are the reset vectors. On power up, or pressing the QL reset button, vector 0 is read into the supervisor stack pointer, vector 1 is read into the program counter, the status register is altered so the bit **S** is set, the **T** bit is cleared, the interrupt level is set to 7, and processing is initiated. No record is kept of the processor status before the reset happened, so any program which was running when you pressed reset cannot be restarted.

The 68008 has three output pins which identify to its associated hardware the type of item it is addressing, with the possible types being user data, user program, supervisor data, supervisor program and interrupt acknowledge. These signals allow the external hardware to protect memory or peripherals from access or alteration by an unauthorised user program, or allow the same address to refer to different chips depending on this addressing mode. The 68010 and 68020 processors in the 68000 family have special instructions which

allow a supervisor program to manipulate the addressing mode outputs so that the supervisor program can access other address spaces.

Vectors 0 and 1 are in supervisor program space, the remaining exception vectors are in supervisor data space. This allows vectors 0 and 1 and the initial program they set running to be in ROM, but the remaining vectors may be in RAM and thus be free to be changed by the program.

NAME	OFFSET TO BE ADDED TO THE CONTENTS OF $28050 TO FIND THE NEW VECTOR	68000 VECTOR NUMBER
ADDRESS ERROR	$0054	3
ILLEGAL INSTRUCTION	$0058	4
DIVISION BY ZERO	$005C	5
CHK OUTSIDE RANGE	$006Ø	6
TRAPV WHEN OVERFLOW	$0064	7
PRIVILEGE VIOLATION	$0068	8
TRACE EXCEPTION	$006C	9
LEVEL 7 INTERRUPT	$007Ø	31
TRAP #5	$0074	37
TRAP #6	$0078	38
TRAP #7	$007C	39
TRAP #8	$008Ø	4Ø
TRAP #9	$0084	41
TRAP #$A	$0088	42
TRAP #$B	$008C	43
TRAP #$C	$009Ø	44
TRAP #$D	$0094	45
TRAP #$E	$0098	46
TRAP #$F	$009C	47

TABLE 6.1 QL REDIRECTABLE EXCEPTION VECTORS (VERSION "AH")

The QL does not use the addressing mode signals, all the QL addressing modes share a common address space, none of which is protected against being overwritten by wild user programs. Thus, all the QL exception vectors are in ROM, but the QL does allow a number of the vectors to be redirected by a table in RAM. Such redirectable vectors will be identified in the following list.

When a vector is redirectable, the QL examines the longword system variable starting at address $28050. If this is zero, no redirection takes place, otherwise its contents are used as the base address of a redirection table, to which an offset is added as indicated in table 6.1. The longword at the resultant address is taken as the starting address of your exception processing routine.

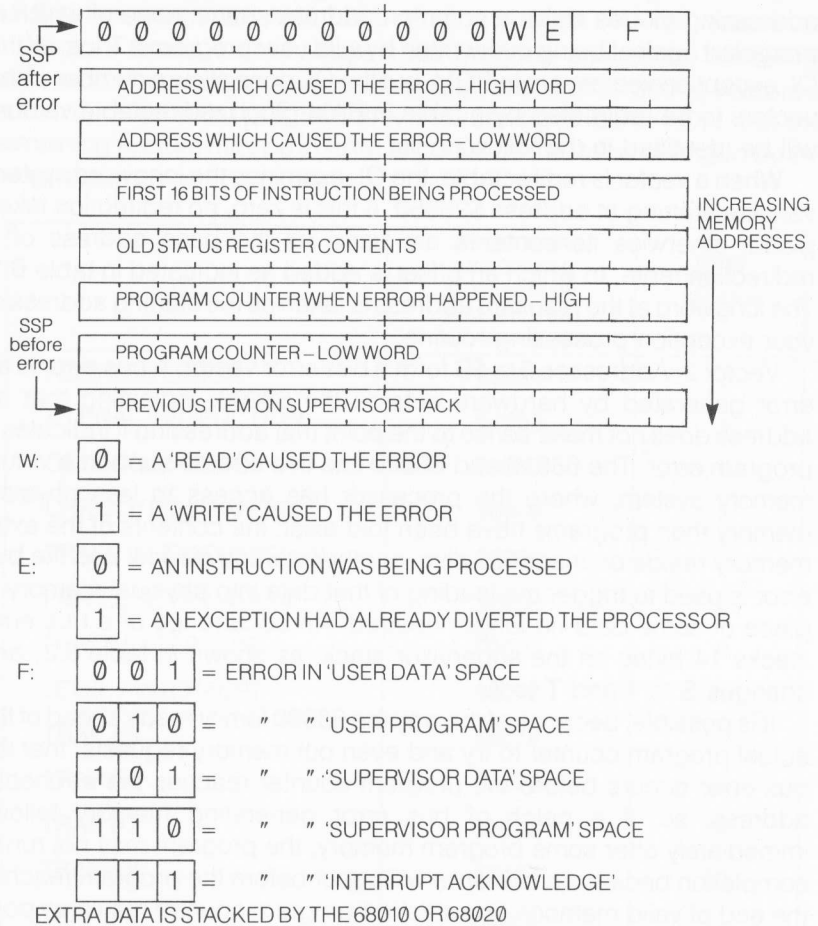
Vector 2: Addresses 8 to $B form a bus error vector. A bus error is an error generated by hardware outside the 68008, indicating that an address does not make sense to the point that addressing it indicates a program error. The 68010 and 68020 use this error to support a virtual memory system, where the processor has access to less physical memory than programs have been told exist; the contents of the extra memory reside on magnetic disc, or other storage device, and the bus error is used to trigger the loading of that data into physical memory in place of some data no longer needed. An occurrence of a bus error stacks 14 bytes on the supervisor stack, as shown in table 6.2, and changes **S** to 1 and **T** to 0.

It is possible, because of the way the 68000 family reads ahead of the actual program counter to try and even out memory requests, that the bus error occurs before the program counter reaches the erroneous address; so, if a patch of bus error generating memory follows immediately after some program memory, the program may not run to completion because a bus error will occur before the program reaches the end of valid memory. The stacked program counter may also point part way through the instruction being executed when the error occurred, so it may be necessary to search backwards from the given program counter to find the start of the instruction which caused the error.

The QL does not support bus errors. None of the internal QL hardware generates bus errors. External hardware plugged into the peripheral expansion port may generate bus errors, but the QL response is to make an immediate return to the code which caused the error, to try to continue as if nothing had happened.

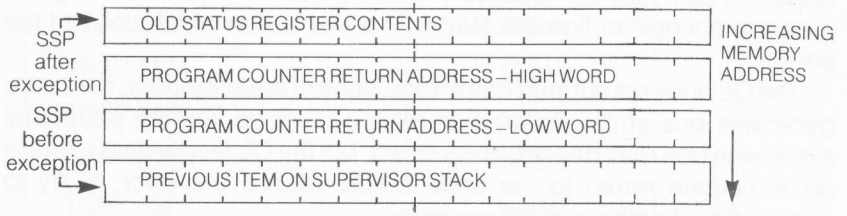
Vector 3: Addresses $C to $F form the address error vector. An address error occurs when the 68008 accesses a word or longword

BUS ERROR OR ADDRESS ERROR EXCEPTIONS



EXTRA DATA IS STACKED BY THE 68010 OR 68020

ALL OTHER EXCEPTIONS



AN EXTRA WORD GIVING THE EXCEPTION VECTOR NUMBER IS STACKED BY THE 68010 OR 68020

TABLE 6.2 DATA STACKED BY EXCEPTIONS (68008 OR 68000)

item at an odd address. The same information is stacked as for a bus error. This vector is redirectable, and the new vector starts $54 bytes after the address given in $28050.

Vector 4: Addresses $10 to $13 form the illegal instruction vector. This and all subsequent exceptions only cause 6 bytes to be placed on the supervisor stack, namely the program counter and the status register. As usual, the status flags are also changed to indicate supervisor, untraced mode.

This vector indicates where program control will transfer to if one of most of the codes which cause the disassembler to set **fault=1** are executed. Again, the vector is redirectable on the QL.

Vector 5: is the division by zero vector. Rather than just setting the **V** flag, division by zero causes this **TRAP**. This vector is software redirectable on the QL.

Vector 6: is the **CHK** vector. If a **CHK** fails, this vector defines where execution continues. It is redirectable on the QL.

Vector 7: is the **TRAPV** vector. If the **V** flag is set when **TRAPV** is executed, this QL redirectable vector comes into play.

Vector 8: is the privilege violation vector. If a user program tries to alter the contents of the whole status register, or tries to use one of the privileged instructions we shall meet later in this chapter, processing moves to the address defined by this QL redirectable vector.

Vector 9: is the trace vector. If the **T** bit is set in the status register, this vector is followed after every instruction is executed. Trace mode is switched off while the exception is processed, and will normally be switched on again by the restoration of the old status register from the stack, when the processor returns to execute the next instruction. As usual, the vector is redirectable on the QL.

Vector 10: is the vector for the unimplemented **disA$** type of instruction. Operation codes beginning with $A were intended by Motorola to mimic in software the operation of a possible co-processor.

The QL designers (certainly in version "AH" of the QL ROM) have overlooked this **TRAP** and this vector contains part of a QL exception processing routine, namely $61266124. Execution of an instruction beginning with $A will thus cause a jump to address $66124, which is in the space reserved for the 512k byte expansion RAM.

Vector 11: is similar to vector 10, being intended for those instructions which start with the hexadecimal digit $F. The version "AH" ROM contains $61226120 at this location, which will cause a jump to address $26120, a location in the screen memory.

Thus, while the execution of an instruction beginning $A or $F probably indicates that the QL has crashed, the QL programmers have chosen not to take advantage of the 68008's identification of such crashes, which might thus have prevented any very disastrous results of the crash. Instead, the 68008 will compound the crash problems, switching into supervisor mode and starting to execute data or non-existent memory.

The next few vector numbers are officially reserved by Motorola for some, as yet unannounced, purpose.

Vector 24: Addresses $60 to $63 are the spurious interrupt vector. These occur if interrupting hardware presents a level number which causes an interrupt, but by the time the 68008 goes back to read the level number in order to determine where execution should continue, the interrupt level number has returned to zero. The QL ignores such interrupts, doing an immediate return to the interrupted program.

Vector 25: is for level 1 interrupts; these are also actively ignored by the QL.

Vector 26: is for level 2 interrupts. These are obviously used by the QL hardware, as the vector points to a complex routine in ROM. When an interrupt occurs, the I bits in the status register are set to the interrupting level, in this case 2, preventing further interrupts at the same or lower levels unless the software changes the setting of the I bits.

Vector 27 to 30: are for level 3 to level 6 interrupts. The QL ignores these.

Vector 31: is for level 7 interrupts, the equivalent of a non-maskable interrupt, as level 7 interrupts cause an exception even when all three I status bits are set to 1. The QL can redirect this interrupt through its RAM table.

Vector 32: is for the command **TRAP#0**. The **TRAP** command is like the Z80 processor's **RST** command in that it allows a short instruction to call an operating system routine. On the QL, **TRAP#0** is implemented as a request to change to supervisor mode.

Vector 33 to 36: are for **TRAP#1** to **TRAP#4**. These are used in the QL for QDOS subroutine calls.

Vector 37 to 47: are for **TRAP#5** to **TRAP#$F**. These vectors are software redirectable on the QL.

An external device plugged into the QL expansion port which causes an interrupt can invoke any vector from 0 to 255, and is expected by Motorola to use a vector from 48 to 255. On the QL, most of this area of the ROM contains program, and so it is probably best for any external

device to be designed to invoke a level 7 interrupt and use the facility of the QL to vector exceptions through a table in RAM.

Let us now return to the disassembler program itself.

```
4760 DEFine FuNction dis4E$(pc,j,k)
4770 IF j=0 THEN fault=1:RETurn ""
4780 IF j>1 THEN
4790 IF k<=15 OR (k)=24 AND k<=39) THEN fault=1:RETurn ""
4800 IF j=2 THEN RETurn "JSR▲▲▲▲▲"&adr$(k DIV 8,k MOD 8,pc)
4810 RETurn "JMP▲▲▲▲▲"&adr$(k DIV 8,k MOD 8,pc)
4820 END IF
```

JMP and **JSR** (jump to subroutine) are very similar instructions. In a similar manner to **LEA** and **PEA** they calculate an effective address and use that rather than the data it addresses, in this case by loading the effective address into the program counter. In addition, **JSR** saves the address of the following instruction as 4 bytes on the A7 stack in order to enable the processor to return to the calling routine. Neither instruction alters the condition code flags.

```
4830 SELect ON k
4840 =0 TO 15:RETurn "TRAP▲▲▲▲#$"&hex$(k)
```

This is the **TRAP** instruction mentioned earlier in the chapter, which can call one of sixteen operating system subroutines. The call preserves the status register, and changes to supervisor state.

```
4850 =16 TO 23:pc=pc+2
4860 RETurn "LINK▲▲▲▲A"&(k MOD 8)&",#$"&hexcon$(pc-2)&hexcon$(pc-1)
```

LINK is a useful command for high level languages, operating systems or other large programs, particularly those involving recursion (where a routine calls itself either directly or indirectly). One of the address registers is nominated as a frame pointer which is going to point to a part of the stack reserved for the local variables of the current subroutine. **LINK** is normally used on entry to the subroutine in order to set up the frame. It assumes that the named address register contains a previous frame pointer, or other important information, so it is pushed on to the A7 stack. The new value of A7 is copied into the frame pointer register and the immediate data word is added to A7 to reserve space on the stack. The immediate data word is thus minus the number of bytes of local data required: to make sense this number should be even and negative. For instance,

```
LINK    A3,$FFF0
```

reserves 16 bytes of data space, which can be addressed by the

indexed addressing forms **$FFF0(A3)** to **$FFFF(A3)**. If you need to access the previous subroutine's local variables, something like:

```
MOVE.L  (A3),A2
```

could be used to pick up the previous frame pointer.

```
4870 =24 TO 31:RETurn "UNLK    A"&(k MOD 8)
```

UNLK unlinks a frame from the stack, the reverse of **LINK**. First, the frame pointer register is copied into A7, then the old frame pointer is popped off the A7 stack into the frame pointer register. Never be tempted to use A7 as a frame pointer, the instruction set has space for this instruction, but the multiple use of the stack pointer will cause confusion.

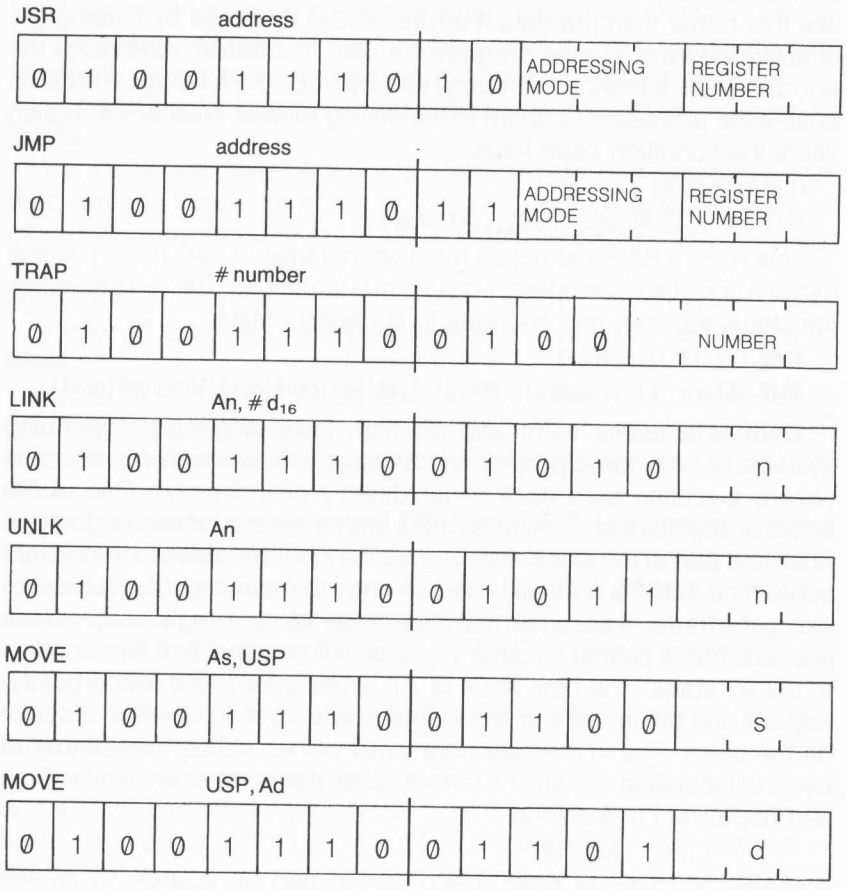


TABLE 6.3 JUMPS AND TRAPS

You can allow a stack frame to grow during the course of a subroutine by pushing more items on to the A7 stack as you decide that you need them, as **UNLK** will always manage to tidy up the stack, deleting however many unpopped items are left. So,

```
LINK    A3,#0
```

is quite sensible, it maintains a frame pointer chain, but leaves the actual frame contents to be set during the subroutine.

```
4880 =32 TO 39:RETurn "MOVE    A"&(k MOD 8)&",USP"
```

This is a privileged version of the **MOVE** command, allowing a supervisor program to set up all 32 bits of the user stack pointer (A7 in user mode) before setting a user program running.

```
4890 =40 TO 47:RETurn "MOVE    USP,A"&(k MOD 8)
```

Again, the command is privileged and it moves all 32 bits despite the absence of a ".L" after the mnemonic. It allows a supervisor program to read the contents of the user stack pointer. In a trace routine, for instance, this could then be printed on the screen, or checked against a range of valid addresses to warn of a program with a wild stack.

```
4900 =48:RETurn "RESET"
```

The **RESET** command is unconnected with the reset vectors or the reset switch on the QL. It outputs a pulse on one of the 68008 pins which is intended to reset some peripheral chips. Program execution continues with the next instruction. **RESET** is a privileged command.

```
4910 =49:RETurn "NOP"
```

NOP stands for no operation. A command common to many microprocessors, it is useful in producing precise timing in time critical operations, and in filling spaces where program errors have been removed.

```
4920 =50:pc=pc+2
4930 RETurn "STOP    #$"&hexcon$(pc-2)&hex_on$(pc-1)
```

Like most microcomputer **STOP** or **HALT** commands, this one pauses the operation of the 68008 until an interrupt occurs, after the processing of which, execution continues at the instruction following the **STOP**. The immediate data is loaded into the status register, and in particular the I status bits, allowing the command to specify the acceptable interrupt levels. **STOP** is a privileged command, and if the S bit in the immediate data is clear, that will also generate a privilege violation **TRAP**.

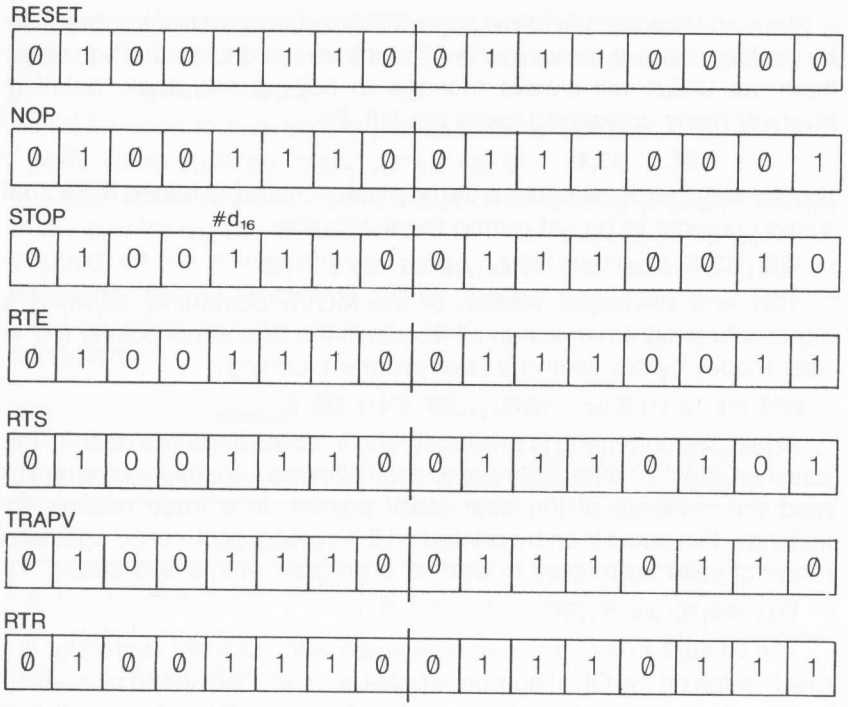


TABLE 6.4 INSTRUCTIONS WITHOUT PARAMETERS, AND STOP

```
4940 =51:RETurn "RTE"
```

RTE stands for return from exception. It is a privileged command, as it takes two bytes from the supervisor stack and places them in the status register, and then takes the next four bytes from the supervisor stack and places them in the program counter. It is thus the command to use to return from a **TRAP**, interrupt or other exception, restoring the full status register. Exception processing routines must be careful to preserve any other registers which they use.

```
4950 =53:RETurn "RTS"
```

RTS stands for return from subroutine. It pops four bytes from the current A7 stack into the program counter, the reverse of **JSR**.

```
4960 =54:RETurn "TRAPV"
```

TRAPV, trap on overflow, causes a trap to vector 7 if the **V** condition code flag is set. A program which is concerned to identify and deal with all arithmetic overflows would use **TRAPV** after every arithmetic instruction.


```
4970 =55:RETurn "RTR"
```

RTR stands for restore and is a non-privileged version of **RTE** in that it takes six bytes from the current A7 stack, placing the second byte in the condition codes register, and the last four bytes in the program counter while leaving the rest of the status register untouched.

We mentioned earlier the QL's use of **TRAP#0** to switch to supervisor mode. The actual code executed by **TRAP#0** is:

```
ADDQ    #2,A7
RTS
```

We have not met **ADDQ** yet, but it is simply a short form of an immediate add, and by adding two to A7, the program moves past the status register contents which form the last item on the stack, and then loads the next four bytes into the program counter. As the **S** flag is set by a **TRAP**, the result is that the processor returns to the calling point with supervisor mode set, and the supervisor version of A7 available, rather than the user stack pointer.

However, consider an alternative **TRAP#0** routine:

```
RTR
```

This would take six bytes from the supervisor stack, the last four being the program counter, leaving the processor in supervisor mode. This version of the **TRAP#0** routine is two bytes shorter than the original and is also fractionally faster.

```
4980 =REMAINDER :fault=1:RETurn ""
4990 END SELect
5000 END DEFine
```

Chapter 7
'Quick' Arithmetic and Conditional Operations

Before we look at **dis5$**, which would be the next function to handle if we followed strict numerical order, we will make a short digression:

```
5010 DEFine FuNction dis7$(pc)
5020 IF PEEK(pc)MOD 2=1 THEN fault=1:RETurn ""
5030 pc=pc+2
5040 RETurn "MOVEQ␣␣␣#$"&hexcon$(pc-1)&",D"&(PEEK(pc-2)DIV 2 MOD 8)
5050 END DEFine
```

MOVEQ stands for move quick, being a very short, and hence fast, form of movement of immediate data to a data register. The eight bits of immediate data are sign extended to fill all 32 bits of the data register, rather than just affecting the least significant byte. As usual with **MOVE**, the **C** and **V** flags are cleared by the operation, **Z** and **N** are set according to the value of the result and **X** is unaffected. Note that **MOVEQ #0** is actually a slightly faster way of clearing a data register than **CLR.L**.

Now we return to **dis5$**.

```
5060 DEFine FuNction dis5$(pc)
5070 LOCal i,j,a$
5080 i=PEEK(pc)MOD 16
5090 j=PEEK(pc+1)
5100 pc=pc+2
5110 IF j<192 THEN
5120 IF i MOD 2=0 THEN
5130 a$="ADDQ."
5140 ELSE
5150 a$="SUBQ."
5160 END IF
5170 IF j MOD 64)=58 THEN fault=1:RETurn ""
5180 IF i<2 THEN i=16
```

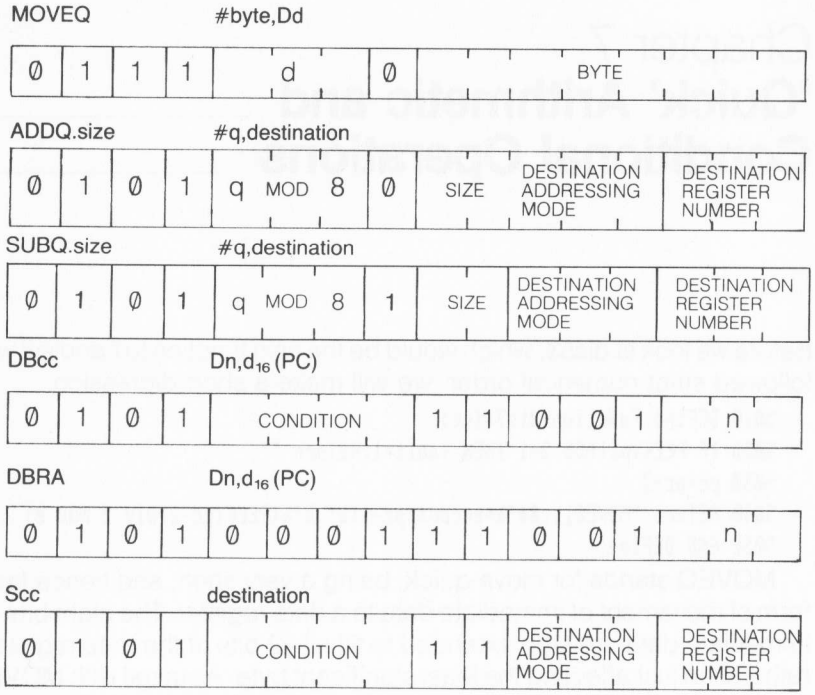


FIGURE 7.1 QUICK ARITHMETIC AND CONDITIONAL ARITHMETIC

ADDQ and **SUBQ** (add quick and subtract quick) can only add or subtract a number in the range 1 to 8, the value of 8 replacing the not very useful value of zero.

```
5190 SELect ON j
5200 =8 TO 15:fault=1:RETurn ""
5210 =0 TO 63:a$=a$&"B"
```

Address registers cannot be used in byte lengths, and this order of program lines rejects such an attempt. Note, in general, that where **SELect** ranges overlap, the first range which gives a match is obeyed, and then the QL ignores all subsequent ranges, even if they also match the selected item.

```
5220 =64 TO 127:a$=a$&"W"
5230 =REMAINDER :a$=a$&"L"
5240 END SELect
5250 RETurn a$&"▲▲#"&(i DIV 2)&","&adr$(j DIV 8 MOD 8,j MOD 8,pc)
5260 END IF
```

ADDQ and **SUBQ** can affect any normally addressable item, apart

from program counter relative items, as opposed to **MOVEQ** being limited to only affecting data registers. Remember that when address registers are affected, the source operands are sign extended to 32 bits before the operation takes place, so there is absolutely no difference in operation between word and longword forms of these instructions when an address register is the destination, the different instruction forms are simply retained for completeness. **ADDQ** and **SUBQ** affect all the flags, **C**, **V**, **Z**, **N** and **X** except when the destination is an address register, when no flags are affected.

```
5270 IF j DIV 8 MOD 8<>1 THEN
5280 IF j>=250 THEN fault=1:RETurn ""
5290 RETurn "S"&con$(i)&"▲▲▲▲"&adr$(j DIV 8 MOD 8,j MOD 8,pc)
5300 END IF
```

Scc stands for set conditionally, and it sets a byte in a data register or an addressed item to **$00** if the given condition test is false, but to **$FF** if the condition is true. Decoding of the condition tests is left to the function **con$**, as they are common to the next few instructions. **Scc** does not alter the condition code flags, instead it is a way of remembering some of their settings for later use. **SF**, set false, is particularly useful if you need to clear a byte register in a peripheral chip, where reading the register would produce unwanted side effects, as **CLR.B** actually performs a read before its write. You will find that the QL ROM uses **SF** quite frequently. **ST**, set true, is also useful, in that it quickly produces the value minus one.

```
5310 pc=pc+2
5320 RETurn "DB"&con$(i)&"▲▲▲▲D"&(j MOD 8)&",$"&hexcon$(pc-2)&hexcon$
     (pc-1)&"(PC)=$"&hex5$(pc-2+256*PEEK(pc-2)-PEEK(pc-2)DIV 128*6553
     6+PEEK(pc-1))
5330 END DEFine
```

DBcc stands for decrement and branch until. You see from line 5320 that its operation depends on a condition code test and a data register, and it may result in the processor making a program counter relative jump using a 16 bit signed offset. We have calculated the actual address to which the processor may jump, by adding the offset to the address of the start of the offset word, allowing for sign extension.

Decrement and branch first tests the condition represented by **con$**. If the required condition is true, no action takes place and the instruction following the **DBcc** is executed next. If the condition is false, the contents of the least significant 16 bits of the named data register are decremented by 1. If the result of this subtraction is minus one, no further action takes place and the instruction following the **DBcc** is executed

next. Otherwise the branch is taken. **DBcc** never alters the setting of the condition code flags itself.

DBT, decrement and branch until true, will thus never take the branch and is not a particularly useful instruction.

DBF, decrement and branch until false, is the most widely used **DBcc**, as it is roughly equivalent to the conventional decrement and branch instructions found on other computers (such as **DJNZ** on the Z80 processor). **DBF** is so useful that assemblers will often accept the mnemonic **DBRA** in its place.

Beware the fact that a loop constructed using **DBcc** will not terminate until the counter reaches minus one, so the loop will be executed one more time than the counter might make you think.

The multiple conditions of **DBcc** are occasionally quite useful. Suppose you are reading a line of text into a sixteen character buffer and want to stop, either when the line is terminated by <ENTER> or the buffer is full. The typical program to do this might look like:

```
        MOVEQ   $F,D1
LOOP    JSR     READCHAR
        CMP.B   $0A,D0
        DBEQ    D1,LOOP
```

where the subroutine **READCHAR** reads a character into the buffer and into D0. The loop terminates when D0 is equal to **$0**A or D1 reaches minus one.

dis6$ also deals with conditional instructions and straightforward branches.

```
5340 DEFine FuNction dis6$(pc)
5350 LOCal i,j,a$
5360 i=PEEK(pc)MOD 16
5370 j=PEEK(pc+1)
5380 pc=pc+2
5390 a$=con$(i)
5400 IF a$(2)="␣" THEN
5410 IF a$="T␣" THEN
5420 a$="RA"
5430 ELSE
5440 a$="SR"
5450 END IF
5460 END IF
```

Slightly different mnemonics to usual are used, **BT** is called **BRA**, which stands for branch or branch always rather than branch if true. The

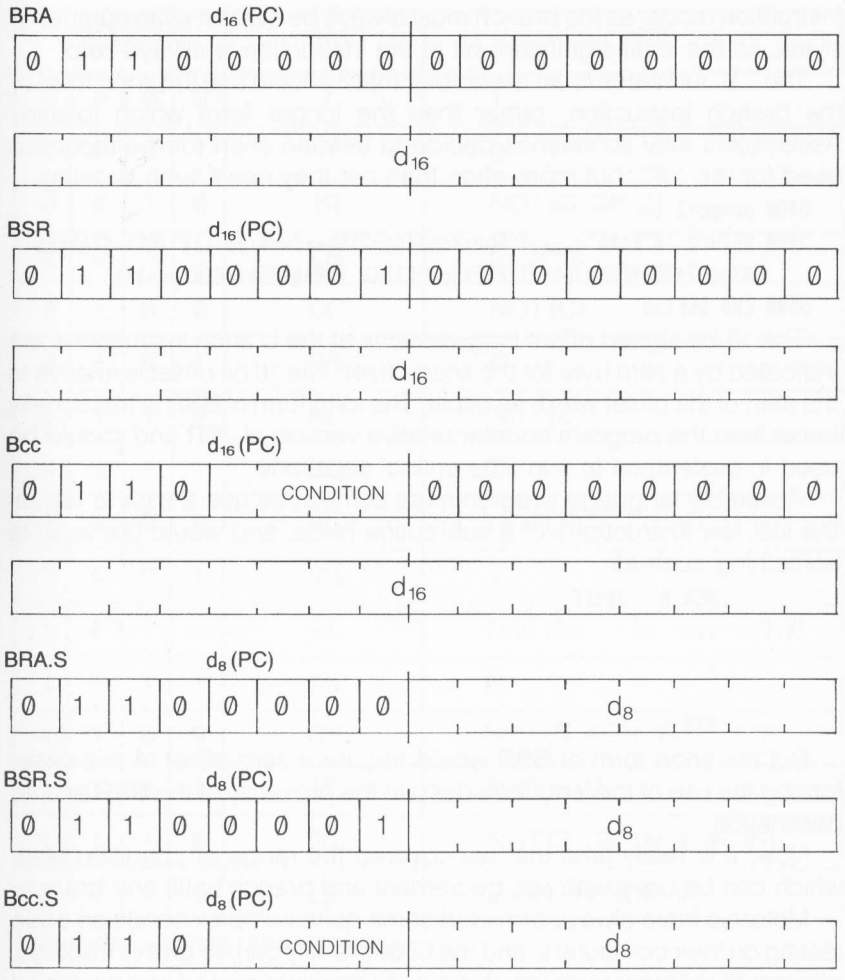


TABLE 7.2 BRANCHES

meaningless instruction **BF** is replaced by the more useful **BSR**, branch to subroutine, which pushes a return address onto the A7 stack.

```
5470 IF j<>0 THEN RETurn "B"&a$&".S␣␣␣$"&hexcon$(pc-1)&"(PC)=$"&hex5$
     (pc+j-j DIV 128*256)
```

This basic form of branch is shorter than **JMP** or **JSR**, which always require an extension word for program counter relative addressing; but it only allows a jump over a range of −128 to +126 bytes from the start of the next instruction. Indeed, there is a slight waste of space in this

instruction mode, as the branch must always be over an even number of bytes, so the least significant bit of the instruction is always zero.

The ".S" indicates to an assembler that it should use the short form of the branch instruction, rather than the longer form which follows. Assemblers may sometimes decide to use the short form without the need for the ".S", but more often than not they need such coaxing.

```
5480 pc=pc+2
5490 RETurn "B"&a$&"␣␣␣␣␣$"&hexcon$(pc-2)&hexcon$(pc-1)&"(PC)=$"&hex5
     $(pc-2+256*PEEK(pc-2)-PEEK(pc-2)DIV 128*65536+PEEK(pc-1))
5500 END DEFine
```

The 16 bit signed offset long versions of the branch instructions are indicated by a zero byte for the short offset. The 16 bit offset is relative to the start of the offset word, as usual. The long form of **BSR** is fractionally faster than the program counter relative version of **JSR** and should be used in preference to it in time critical situations.

Assembly language programmers sometimes use a trick to repeat the last few instructions of a subroutine twice, and would like to write something such as:

```
        BSR.S   TWICE
TWICE   ...
        .
        .
        RTS
```

But the short form of **BSR** would require a zero offset in this case, forcing the use of the long form despite the proximity of the **BSR** and its destination.

Now, it is really time that we covered the range of condition tests which can be used with set, decrement and branch until, and branch.

Motorola have always provided some quite complex condition code testing on their computers, and the 68000 family carries on this tradition, offering 16 different tests which cover all possible signed and unsigned arithmetic relationships.

```
5510 DEFine FuNction con$(i)
5520 SELect ON i
5530 =0:RETurn "T␣"
5540 =1:RETurn "F␣"
5550 =2:RETurn "HI"
5560 =3:RETurn "LS"
5570 =4:RETurn "CC"
5580 =5:RETurn "CS"
5590 =6:RETurn "NE"
```


CONDITION BIT PATTERN				NAME	TEST
0	0	0	0	T	1
0	0	0	1	F	0
0	0	1	0	HI	NOT (C OR Z)
0	0	1	1	LS	C OR Z
0	1	0	0	CC	NOT (C)
0	1	0	1	CS	C
0	1	1	0	NE	NOT (Z)
0	1	1	1	EQ	Z
1	0	0	0	VC	NOT (V)
1	0	0	1	VS	V
1	0	1	0	PL	NOT (N)
1	0	1	1	MI	N
1	1	0	0	GE	NOT (N XOR V)
1	1	0	1	LT	N XOR V
1	1	1	0	GT	NOT (Z OR (N XOR V))
1	1	1	1	LE	Z OR (N XOR V)

TABLE 7.3 CONDITIONAL TESTS

```
5600 =7:RETurn "EQ"
5610 =8:RETurn "VC"
5620 =9:RETurn "VS"
5630 =10:RETurn "PL"
5640 =11:RETurn "MI"
5650 =12:RETurn "GE"
5660 =13:RETurn "LT"
5670 =14:RETurn "GT"
5680 =15:RETurn "LE"
5690 END SELect
5700 END DEFine
```

We have already met the tests F and T which are respectively always false and always true.

EQ, equal, tests the Z condition flag and is true if Z is set.

NE, not equal, is the opposite, being true if Z is clear.

CS, carry set, is true if C is set, amazingly enough.

CC, carry clear, is true if C is clear.

VS, overflow set, is true if V is set.

VC, overflow clear, is true if V is clear.

MI, minus, is true if N is set.

PL, plus, is true if N is clear; so, as usual, zero is a positive number.

The remaining tests all actually test more than one bit in the condition code register.

LS, lower than or the same, is true if C is set or Z is set and thus makes an unsigned arithmetic test.

HI, higher than, is the opposite, being true only if both C and Z are clear.

Some assemblers accept LO and HS meaning lower than, and higher than or the same respectively, as alternatives for CS and CC respectively, as they are also unsigned arithmetic tests.

LT, less than, is a signed arithmetic test, true if N is set and V is clear (a straightforward less than), or if N is clear and V is set (a less than involving numbers so far apart that overflow occurred).

GE, greater than or equal, is the opposite, true if N is clear and V is clear, or if N is set and V is set.

LE, less than or equal, is an extension of LT being true if LT is true, or Z is set.

GT, greater than, is the opposite, being true only if GE is true and Z is clear.

It is thus possible to test any single condition code flag, apart from X, and all sensible arithmetic comparisons.

Suppose you want to write:

```
IF D1<D0 THEN GOTO LABEL
```

in assembler.

If you are doing signed arithmetic this would assemble as:

```
        CMP     D0,D1
        BLT     LABEL
```

Or, in unsigned arithmetic:

```
        CMP     D0,D1
        BCS     LABEL
```

as BCS is equivalent to BLO, the test you want to do.

Chapter 8
ADDition and SUBtraction

```
5710 DEFine FuNction dis9$(pc):RETurn "SUB"&dis9orD$(pc):END DEFine
5720 DEFine FuNction disD$(pc):RETurn "ADD"&dis9orD$(pc):END DEFine
```

Addition and subtraction instructions are provided in matching sets, so that we can write a common function to decode both types of instruction.

```
5730 DEFine FuNction dis9orD$(pc)
5740 LOCal i,j,k,a$
5750 i=PEEK(pc)MOD 16
5760 j=PEEK(pc+1) DIV 64
5770 k=PEEK(pc+1)MOD 64
5780 pc=pc+2
5790 IF i MOD 2=0 THEN
5800 SELect ON j
5810 =3:IF k=60 THEN
5820 pc=pc+2
5830 RETurn ".W▲▲▲#$"&hexcon$(pc-2)&hexcon$(pc-1)&",A"&(i DIV 2)
5840 END IF
5850 RETurn ".W▲▲▲"&adr$(k DIV 8,k MOD 8,pc)&",A"&(i DIV 2)
```

This first sub-group of commands allows the addition or subtraction of any addressed word or immediate data, to or from an address register. As mentioned before, the source operand is sign extended to 32 bits and the addition or subtraction affects all 32 bits of the destination register, and no condition code flags are affected by the instruction.

```
5860 =0:a$=".B"
5870 =1:a$=".W"
5880 =2:a$=".L"
5890 END SELect
5900 RETurn a$&"▲▲▲"&adr$(k DIV 8,k MOD 8,pc)&",D"&(i DIV 2)
5910 END IF
```

When adding to or subtracting from data registers, as in this group of instructions, only the least significant byte, word or longword is affected in the destination register, and all the condition code flags are set depending on the result.

```
5920 SELect ON j
5930 =3:IF k=60 THEN
5940 pc=pc+4
5950 RETurn ".L   #$"&hexcon$(pc-4)&hexcon$(pc-3)&hexcon$(pc-2)&hexco
     n$(pc-1)&",A"&(i DIV 2)
5960 END IF
5970 RETurn ".L   "&adr$(k DIV 8,k MOD 8,pc)&",A"&(i DIV 2)
```

This completes the commands which have address registers as the destination. The same conditions apply as the earlier group of address altering commands.

```
5980 =0:a$=".B"
5990 =1:a$=".W"
6000 =2:a$=".L"
6010 END SELect
6020 IF k<8 THEN
6030 RETurn "X"&a$&"   D"&k&",D"&(i DIV 2)
6040 END IF
```

ADDX and **SUBX** are extended arithmetic commands which also add or subtract the value of the **X** condition code bit into the operation. All condition code flags are altered to reflect the result of the operation, except that **Z** is unaffected if the result is zero, so that **Z** can reflect the value of the whole extended number.

```
6050 IF k<16 THEN
6060 RETurn "X"&a$&"   -(A"&(k-8)&"),-(A"&(i DIV 2)&")"
6070 END IF
```

This command is the more usual extended arithmetic command allowing you to scan two extended numbers stored in memory, starting with the least significant byte, word or longword. Before starting an extended operation, it is usual to clear **X** and set **Z** to obtain the right result for both the number and the flags.

```
6080 IF k>=58 THEN fault=1:RETurn ""
6090 RETurn a$&"   D"&(i DIV 2)&","&adr$(k DIV 8,k MOD 8,pc)
6100 END DEFine
```


SUB/ADD.W/L source, Ad

1	SUB/ADD	0	1	d	W/L	1	1	SOURCE ADDRESSING MODE	SOURCE REGISTER NUMBER

SUB/ADD.size source, Dd

1	SUB/ADD	0	1	d	0	SIZE	SOURCE ADDRESSING MODE	SOURCE REGISTER NUMBER

SUBX/ADDX.size Ds, Dd

1	SUBX/ADDX	0	1	d	1	SIZE	0	0	0	s

SUBX/ADDX.size -(As),-(Ad)

1	SUBX/ADDX	0	1	d	1	SIZE	0	0	1	s

SUB/ADD.size Ds, destination

1	SUB/ADD	0	1	s	1	SIZE	DESTINATION ADDRESSING MODE	DESTINATION REGISTER NUMBER

TABLE 8.1 ADDITION AND SUBTRACTION

And, finally, we have a mode which allows a value held in a data register to affect a data item in memory. All the condition code flags are affected by this operation.

Note how **ADD** and **SUB** do not have all the various combinations of source and destination addressing modes which are available to **MOVE**. There are instructions available for whenever the destination is a data register or an address register, or if those are not suitable, there are modes for whenever the source is a data register or, as we saw in Chapter 4, when the source is immediate data; but operations where both source and destination are in memory are not allowed, excepting the special mode available for extended arithmetic.