3 The Instruction Repertory

In this Section are given detailed descriptions of the various functions in the Orion instruction repertory. Each description is given in a precise symbolic form and also verbally.

Exceptions and special cases are given and brief examples of the uses of some instructions.

When discussing some of the simpler groups of functions, a general introduction to the group is given, followed by a short description of the individual functions within that group. The more complex instructions are described individually in detail.

It is assumed that the reader is familiar with the differences between the 2-address and 3-address types and with the modification and replacement facilities.

In these descriptions the X- and Y-addresses are the effective addresses at the time the instruction is obeyed, after all replacements and/or modifications have been done, including possibly pre-modification by one or more 116 and/or 117 instructions.

Timing of Orion Programs

Orion 1, like other machines of its generation, does not have fixed instruction times as do simple serial machines. The following are some of the more important factors which introduce variations into instruction times.

1) Carries

Orion has a parallel adder with a carry detection circuit and any micro-program operation which refers to the result of an addition or subtraction may be subject to a delay whilst waiting for this carry to die down. This applies particularly to the carries generated in a modification and the carry produced as a result of arithmetic orders although in this latter case, due to the carry-speed circuits and the fact that the next instruction is read before the result is written away, the maximum delay is 8 microsecs.

2) Address Checking

The address-checking circuits on Orion, for both lock-outs and reservations, operate in a synchronous serial manner. Reading from the core store can take place without waiting for the address checking to be completed, but at the end of each instruction that jumps or performs a write there may be a delay for this checking to be completed. The core store operates in 12 microsecs. and an address can be checked in 16 microsecs so that instructions which do 4 operations on three addresses, such as the two address 00 instruction, can operate in 48 microsecs although this depends on starting the instruction at the correct phasing with respect to the address checking. If an address which is being checked for lock-outs agrees with one of the lock-out addresses in the most significant 7 bits of the address (this address having been left justified through n places when the store has 2^{15-n} words) then the lock-out checking will take an extra 16 microsecs even though the word may not be locked out.

3) Hesitations

When a peripheral transfer is initiated the number of words or characters called for is added into the program timer although when reading magnetic tape or using modes 1 or 21 on paper tape there may be less actual hesitations. When a program is running at the same time as a peripheral device the peripheral device control unit has priority over the central machine in use of the core store and whenever the peripheral uses a core store cycle the timer does not get augmented; this happens whether or not the computer required access to the store at that time. The occurrence of hesitations prevents the computer from making effective use of the 12 microsec core store cycle, on the other hand if a hesitation occurs at the time a jump instruction is about to jump, the jump can actually take place quicker than it otherwise would.

The time taken to obey a section of program on Orion is not directly the sum of the times for the individual instructions since each instruction time is a function of the conditions at entry and so depends on the preceding instructions in the program. None-the-less it is hoped that the figures given below are useful averages when due notice is taken of the effects described above. While efforts have been made to ensure accuracy, the times required by instructions may, in the end, be different for technical reasons from those given here.

3.A Instruction Times for Orion 1

This section gives the average time taken to obey instructions. For those which involve an arithmetical process (this includes comparisons made for jumps and table searching etc.) the time may be faster or slower than that stated, depending on the carries involved, i.e. on the operands. The timing for the floating-point instructions depends to a great extent on the operands.

3.A.1 Orion 1 Instruction Times (Times in micro seconds)

Replacement of an address takes 16 microseconds.

	3-address	2-address	2-add unmodified
Group 0	80	80	64
(03 and 04)	64	80	64
Group 1	64	64	48
(13 and 14)	48	64	48
Group 2	As for Group 0		
Group 3			
30,31	208	208	192
32	240	240	224
33	272	288	272
34	80+F	80+F	64+F

where $F = 16\left[\frac{n+1}{8}\right] + 32\left[\frac{n-2}{8}\right]$ where n is the number of significant bits in Y.

Group 4 (average times for 4 runs)

	3-address	2-address	2-add unmodified
40	765	773	755
41	700	718	686
42	1725	1737	1728
43	588	598	571
44	620	626	608
45	621	631	610

Group 5

n is the number of places shifted.

	3-address	2-address	2-add unmodified
50 to 53			
n<24	64+4n	64+4n	48+4n
n≥24	64+4(n-24)	64+4(n-24)	48+4(n-24)
54 to 57			
n<48	128+4n	140+4n	112+4n
n>48	128+4(n-48)	140+4(n-48)	112+4(n-48)
54 for n=48	129+192	140+192	112+192
55-57 for n=48	As for n>48		

For 54 and 55 if x*<0 then partial justification takes 16 microseconds.

	3-address	2-address	2-add unmodified
Group 6			
	Jump, No Jump	Jump, No Jump	Jump, No Jump
60 to 62	120, 80	96, 64	64, 32
63 and 64	112, 80	96, 64	64, 32
65	120, 64	96, 64	64, 32
66	120, 64	96, 64	72, 32
67	104, 80	96, 64	72, 32
Group 7			
	Jump, No Jump	Jump, No Jump	Jump, No Jump
70 to 73	96, 64	80, 48	64, 32
74	80, 64	--, 48	--, 32
75	96, 48	80, --	64, --
76 and 77	96, 48	80, 48	64, 32
Group 8			
	Jump, No Jump	Jump, No Jump	Jump, No Jump
80	128, 64	144, 80	128, 64
81	80, 112	96, 128	80, 112
82	112, 80	128, 96	112, 80
83	96, 96	120, 112	96, 96
84	96, 166	128, 192	96, 166
85	176, 80	208, 112	176, 80
86	64	80	64
87	72	96	64
Group 9			
90	160	170	153
91	176	186	169
92	160	168	152
93	134	144	120
94	272	287	256
95	400	416	397
Group 10			
100	560	560	528
101	1203	1251	1203
102	129+4n	129+4n	113+4n
where n = no. of shifts to standardise			
103	98+4n	98+4n	82+4n
n=m or m-24, m is no. of places shifted			
104	400	400	384
Group 11			
110 and 112	96	80	64
111	88	80	64
114	112	112	96
115	80	92	64
116	28	32	28
117	32	32	32

Group 12

120 $Y \geq 0$, $n = Y$ $Y < 0$, $n = 3-Y$	68+4n	76+4n	48+4n
121 $0 \leq Y \leq 23$, $n = Y$ $24 \leq Y \leq 255$, $n = Y-24$ $-23 \leq Y \leq -1$, $n = 3-Y$ $-255 \leq Y \leq -24$, $n = -21-Y$	64+4n	64+4n	48+4n
122	80	96	112
123	144+24E	172+24E	176

where E=8-n for n≠0 and E=0 for n=0, n being the number of characters to be changed.

124	96+4n	76+4n	48+4n

where n is the no. of places shifted. If Y<0 add 12 microsecs

125	224+4m		

where m is the no. of shifts needed to standardise.

126	113	124	101

Group 14

142	112+20Y	128+20Y	112+20Y
143	96+30n	96+34n	96+30n

n is the number of words

144,145,146	112+20n	illegal	illegal

3.A.2 Orion 2 Instruction Times

Times in microseconds

Replacement of an address takes 2.5 microseconds.
Modification of an address takes 3.5 microseconds.
Modification of two addresses takes 4.5 microseconds.

* An asterisk indicates a typical value.

	3-address and 2-address unmodified
Group 0	
00, 01, 02, 05, 07	12
03	10
04	8.5
06	11
Group 1	
10, 12, 15	10
11	11
13	8.5
14	6.5
16	8.5
17	10.5
Group 2	
20, 21, 22, 25, 27	10.5
23	9.5
24	8.5
26	10

Add 1 microsecond for an odd pseudo-register.

Group 3	
30	41.5
31	38
32	42
33	50
34	40.5

Group 4

40	152
41	150
42	294
43	150
44	153
45	153

Group 5

L = left shift R = right shift

where L and R are directions of shift performed by the computer.

n = shift number

[x] = integral part of x.
Add 1 microsecond if shift number is negative.

$\delta(x) = 1,\ x \neq 0;\quad \delta(x) = 0,\ x = 0.$

50, 51 L	$11 + n$
50, 51 R	$11 + \left[\frac{n-1}{2}\right] + (n-1)\bmod 2 + \delta(n)$
52,53 L	$11 + 2\left[\frac{n}{8}\right] + \left[\frac{n \bmod 8}{2}\right] + n \bmod 2$
52, 53 R	$11 + \left[\frac{n}{8}\right] + \left[\frac{n \bmod 8}{2}\right] + n \bmod 2$
↑ 54, 55 L	$19.5 + 2n$
↑ 54,55 R	$19.5 + \left[\frac{n-1}{2}\right] + 2((n-1)\bmod 2) + 2\,\delta(n)$
56, 57 L	$18.5 + 4\left[\frac{n}{8}\right] + 2\left[\frac{n \bmod 8}{2}\right] + 2(n \bmod 2)$
56, 57 R	$18.5 + 2\left[\frac{n}{8}\right] + \left[\frac{n \bmod 8}{2}\right] + 2(n \bmod 2)$

↑ Add 1.6 microseconds for a partial justification.

unmodified	3-address	2-address
Group 6		
60 to 65	10.5	8.5
≠67, 67	10.5	8.5

≠ Add 1 microsecond for an odd pseudo-register.

Group 7

	3-address	2-address unmodified
70 to 75	8.5	6
76, 77	10.5	6

Group 8

	3-address or 2-address unmodified
80 to 83	11
84 Successful jump) 85 Unsuccessful jump)	10.5 + 2.8 ((n+1)mod 8)
84 Unsuccessful jump) 85 Successful jump)	30
86	7.0
87	7.5

Group 9

90, 91	24 + n + 1.6m
92	25 + n + 1.6m
93	16.5 + 1.6m
*94	54
*95	160
97	26 + n + 1.6m

n is the difference in exponents
m is the number of places of shifting needed for restandardisation
For 90, 91, 92, 93, and 97, add 1 microsecond if $y_e > x_e$.

Group 10

*100	130
*101	140
*102	380
*103	288
*104	50

Group 11

	3-address	2-address unmodified
110	15	12
111, 112 ≠	14	10.5
114	15	15
115	12.5	12.5
116, 117	8.5	8.5

≠ Add 1 microsecond for an odd pseudo-register.

Group 12

120	13.5 + 1.6n + m	13.5 + 1.6n + m
	n is the number of places shifted.	m is the number of 1's
121	$11 + \left[\frac{n}{8}\right] + \left[\frac{n \bmod 8}{2}\right] + n \bmod 2$	Same as 3-address
	n is the number of places shifted.	
122	10.5 + m	13 + m
	m is the number of characters shifted.	
123	21 + 2m (m≠0); 14.5 (m=0)	23.5+2m(m≠0); 16.5(m=0)
	m is the number of characters unchanged.	
124	16 + 2.6n	12.5 + 2.6n
	Add 1 microsecond if a 1 is found. Add 1 microsecond of Y < 0	
	n is the number of places shifted.	
125	217 + 70m	217 + 70m
	m is the number of places shifting needed for restandardisation.	
*126	220	220

Group 14

142 pair	36.5 + 6.1n	36.5 + 6.1n
143	17.5 + 2.9n	17.5 + 2.9n
144, 145	13.5 + 4.9n	illegal
146	16 + 7.5n	illegal

3.0 Group 0

The instructions in this group perform the stated arithmetical or logical operations involving y, the 48-bit word in register Y and (except for the 03 and 04 functions) x, the 48-bit word in register X. The result is written into accumulator Z in 3-address type or register X in 2-address type (modified or unmodified).

Function number 00 [0000 000]

(3) $z' = x + y$ (2) $x' = x + y$

Add x to y, writing the result to Z or X. OVR is set if the result is out of range: this cannot happen if x and y are of opposite signs.

Function number 01 [0000 001]

(3) $z' = x - y$ (2) $x' = x - y$

Subtract y from x, writing the result into Z or X. OVR is set if the result is out of range: this cannot happen if x and y are of the same sign. OVR will be set if $x \geq 0$ and $y_F = -1.0$.

Function number 02 [0000 010]

(3) $z' = y - x$ (2) $x' = y - x$

Subtract x from y, writing the result into Z or X. OVR is set if the result is out of range: this cannot happen if x and y are of the same sign. OVR will be set if $y \geq 0$ and $x_F = -1.0$.

The result of this instruction is equal in magnitude and opposite in sign to that of a 01-instruction with the same X and Y addresses.

If, in a 3-address 01-instruction, the X- and Y-addresses have been written in the wrong sequence, the error is most easily corrected by changing the function number to 02.

Function number 03 [0000 011]

(3) $z' = -y$ (2) $x' = -y$

Negate y into Z or X. OVR is set if and only if $y_F = -1.0$.

In 3-address form the X address is irrelevant and therefore it is recommended that this form should not be used.

If it is desired to overwrite a number by its own negative an instruction typified by

```
            03          A100        A100
```

can be used but it is faster and therefore preferable to use

```
            12          A100          0
```

Note that when a 2-address 03-instruction is obeyed, x is extracted from the store, even though it is not used. When a 3-address 03-instruction is obeyed the X-address is not checked for lock-outs or for violation of reservations unless replaced (see 2.2.23 and 2.2.24).

<u>Function number 04 [0000 100]</u>

(3) $z' = y$ (2) $x' = y$

Copy y into Z or X. OVR cannot be set by this instruction.

It is not necessary to regard y as a number.

In 3-address form X is irrelevant and therefore it is recommended that this form should not be used.

Note that when a 2-address 04-instruction is obeyed, x is extracted from the store, even though it is not used. When a 3-address 04-instruction is obeyed the X-address is not checked for lock-outs or for violation of reservations unless replaced. (See 2.2.23 and 2.2.24).

<u>Function number 05 [0000 101]</u>

(3) $z' = x \& y$ (2) $x' = x \& y$

Perform the logical operation 'and' between x and y, writing the result into Z or X. OVR cannot be set by this instruction.

For a fuller discussion of the logical operations see Section 3.2.1.

<u>Function number 06 [0000 110]</u>

(3) $z' = x \vee y$ (2) $x' = x \vee y$

Perform the logical operation 'or' between x and y, writing the result into Z or X. OVR cannot be set by this instruction.

For a fuller discussion of the logical operations see Section 3.2.1.

<u>Function number 07 [000 111]</u>

(3) $z' = x \not\equiv y$ (2) $x' = x \not\equiv y$

Perform the logical operation 'not equivalent' between x and y writing the result into 2 or X. OVR cannot be set by this instruction.

For a fuller discussion of the logical operations see Section 3.2.1.

3.1 Group 1

The instructions of this group, except 13 and 14, perform some arithmetical or logical operation between two operands, one of which is the 48-bit word x (a signed number in functions 10, 11 and 12, and a logical quantity in functions 15, 16 and 17). The other operand is the 15-bit representation of Y itself: in fact, by modification and/or replacement, Y may be a 24-bit quantity. The functions 13 and 14 use only the operand Y. In 3-address form the result is written into accumulator Z and in 2-address form it is written into register X.

The operand Y can be regarded as a 48-bit word in which the m.s. 33 or 24 bits are all 0's and the l.s. 15 or 24 bits are the binary representation of Y.

If Y is written as a Basic or a Symbolic Address, the Y-operand is the corresponding 15-bit machine address.

Y is essentially unsigned in the instructions of group 1. For convenience in writing and punching programs it can be written as a negative integer. It is then stored and used as the 15-bit representation of this negative integer. Thus if Y is written as -10 it is stored and used as the 15-bit representation of -10 with the effective value of 32768-10 = 32758. If a Y-operand is written as a negative integer and is then modified, the effective Y is the 24-bit sum of the 24-bit modifier and the 15-bit unsigned representation of the written Y- operand.

Function number 10 [0001 000]

(3) $z' = x + Y$ (2) $x' = x + Y$

Add Y to x, writing the result into Z or X. OVR is set if the result is out of range.

A few examples follow:

```
(a)     10        A100      27        A3
```

sets $(A3)_I' = (A100)_I + 27$ or, equally, $(A3)_F' = (A100)_F + (27 \times 2^{-47})$

```
(b)     10        A100      (A200)    A3
```

sets $(A3)' = (A100) + A200_m$, i.e. the 24-bit unsigned modifier in A200 is added to the signed 48-bit number in A100. Note that this is not the same as

```
        00        A100      A200      A3
```

in which the two signed 48-bit numbers (A100) and (A200) are added together.

(c) 10 A100 A200 A3

forms the sum of the signed 48-bit number in A100 and the machine address represented by the Basic Address A200. (The Y-address can alternatively be written as a Symbolic Address).

(d) 10 A100 0

does not change the content of any register. Such an instruction can be useful if it is desired to hold up a program until e.g. a peripheral transfer (ending at A100 in this case) has been completed satisfactorily.

(e) 10Y A100 (A200) A7

sets (A100)′ = (A100) + $A200_m$ + $A7_m$, i.e. it forms the sum of three numbers by a single instruction, saving one instruction and about 20 microsecs. over the pair of 00-instructions which constitute an alternative method. It is, of course, necessary that $A200_m$ and $A7_m$ are unsigned (non-negative) 24-bit numbers.

Function number 11 [0001 001]

(3) $z' = x - Y$ (2) $x' = x - Y$

Subtract Y from x, writing the result into Z or X. OVR is set if the result is out of range.

The examples given for the 10-instruction are equally valid for the 11-instruction with the appropriate alterations to allow for the fact that Y is now subtracted.

Function number 12 [0001 010]

(3) $z' = Y - x$ (2) $x' = Y - x$

From Y subtract x, writing the result into Z or X. OVR is set if the result is out of range.

A special case of a 2-address 12-instruction is, typically

12 A100 0 (A100)′ = 0 - (A100) = -(A100)

which is the fastest way to overwrite a number by its own negative. This particular instruction will set OVR if $x_F = -1.0$.

Function number 13 [0001 011]

(3) $z' = -Y$ (2) $x' = -Y$

Negate Y, writing the result into Z or X. OVR cannot be set by this instruction.

In 3-address form X is irrelevant and therefore it is recommended that this instruction should be used in 2-address form only.

In 2-address type, x is read from the store but is not used. In 3-address type X is not checked for lockouts or violation of reservations unless replaced (see 2.2.23 and 2.2.24)

Function number 14 [0001 100]

(3) $z' = Y$ (2) $x' = Y$

Copy Y into Z or X. OVR cannot be set by this instruction.

In 3-address form X is irrelevant and therefore it is recommended that this instruction should be used in 2-address form only. In 2-address type, x is read but not used. In 3-address type X is not checked for lockouts or violation of reservations unless replaced (see 2.2.23 and 2.2.24).

Some examples follow

(a) 14 A100 27

set the integer +27 or the fraction 27×2^{-47} in A100.

(b) 14 A100 A200

sets, in A100, the machine address corresponding to Basic Address A200. (If Y = A0, then this instruction sets x_m = datum point). Note that Y can also be written as a Symbolic Address.

(c) 14 A100 (A200)

copies $A200_m$ into A100, leaving $A100_u$ clear.

Function number 15 [0001 101]

(3) $z' = x \& Y$ (2) $x' = x \& Y$

Perform the logical operation 'and' between x and Y, writing the result into Z or X. OVR cannot be set by this instruction.

For a further discussion of the logical operations see Section 3.2.1.

When using this function it is necessary to compute the numerical value of the required mask Y.

Function number 16 [0001 110]

(3) $z' = x \vee Y$ (2) $x' = x \vee Y$

Perform the logical operation 'or' between x and Y, writing the result into Z or X. OVR cannot be set by this instruction.

When using this instruction it is usually necessary to compute the numerical value of the required mask Y.

For a fuller discussion of the logical operations see Section 3.2.1.

<u>Function number 17 [0001 111]</u>

(3) $z' = x \not\equiv Y$ (2) $x' = x \not\equiv Y$

Perform the logical operation 'not equivalent' between x and Y, writing the result into Z or X. OVR cannot be set by this instruction.

When using this instruction it is usually necessary to compute the numerical value of the required mask Y.

For a fuller discussion of the logical operations see Section 3.2.1.

3.2 Group 2

These instructions perform some arithmetical or logical operation between the 48-bit content of the pseudo-register numbered Y and (with the exception of the 23- and 24-instructions) the 48-bit word x.

With function-numbers 20, 21 and 22 both words are regarded as signed numbers while with function-numbers 25, 26 and 27 both words are regarded as logical quantities. The functions numbered 23 and 24 use only the content of the pseudo-register.

The contents of the pseudo-registers cannot be altered, with the following exceptions:-

i) Pseudo-registers 4,5,6 and 7 reflect the current state of the overflow indicator for the program currently being obeyed; reference to either 4 or 5 causes the overflow indicator to be left clear.

ii) Pseudo-registers 18 and 19 reflect the current settings of the hand-switches. These may be altered at will by the operator.

iii) 2 and 3 contain local Civil Time and thus their contents change with time.

The pseudo-register address is taken modulo 32; if the result is 20 or greater then the content is taken as 0^{48} if the address is even, and as 1^{48} if the address is odd.

Generally, an instruction of group 2 is similar to the corresponding one in group 0 but with pY in place of y.

<u>Function number 20 [0010 000]</u>

(3) $z' = x + pY$ (2) $x' = x + pY$

Form the sum of x and pY, writing the result into Z or X. OVR is set if the result is out of range.

<u>Function number 21 [0010 001]</u>

(3) $z' = x - pY$ (2) $x' = x - pY$

Subtract pY from x, writing the result into Z or X. OVR is set if the result is out of range.

<u>Function number 22 [0010 010]</u>

(3) $z' = pY - x$ (2) $x' = pY - x$

Subtract x from pY, writing the result into Z or X. OVR is set if the result is out of range.

*Function number 23 [0010 011]

(3) $z' = -pY$ (2) $x' = -pY$

Negate pY, writing the result into Z or X. OVR is set if and only if either Y = 10 (p10 = -1.0) or Y= 18 or 19 and the handswitch settings are such that p18 or pl9 = -1.0.

In 3-address form X is irrelevant and therefore it is recommended that this instruction should be used in 2-address form only.

*Function number 24 [0010 100]

(3) $z' = pY$ (2) $x' = pY$

Copy pY into Z or X. OVR cannot be set by this instruction.

In 3-address form X is irrelevant and therefore it is recommended that this instruction should be used in 2-address form only.

Function number 25 [0010 101]

(3) $z' = x \& pY$ (2) $x' = x \& pY$

Perform the logical operation 'and' between x and pY, writing the result into Z or X. OVR cannot be set by this instruction.

For a fuller discussion of the logical operations see Section 3.2.1.

Function number 26 [0010 110]

(3) $z' = x \vee pY$ (2) $x' = x \vee pY$

Perform the logical operation 'or' between x and pY, writing the result into Z or X. OVR cannot be set by this instruction.

For a fuller discussion of the logical operations see Section 3.2.1.

Function number 27 [0010 111]

(3) $z' = x \not\equiv pY$ (2) $x' = x \not\equiv pY$

Perform the logical operation 'not equivalent' between x and pY, writing the result into Z or X. OVR cannot be set by this instruction.

For a fuller discussion of the logical operations see Section 3.2.1.

* Functions 23 and 24:
2-address type, x is read but not used.
3-address type, X is not checked for lockouts or reservation violation unless replaced (see 2.2.23 and 2.2.24)

3.2.1 The Logical Operations.

The logical operations provided by function-numbers 05, 06, 07, 15, 16, 17, 25, 26 and 27 are as follows:-

i) AND (&).

Given two operands a, and b, form the result c = a & b by placing in the result a 1-bit in those digital positions where both a and b have 1-bits. All other positions in the result have zeros. Thus if a and b are 4-bit words, e.g.

```
                           a = 1100
                           b = 0110
then the result is c = a & b = 0100
```

ii) OR (∨).

Given two operands a, and b, form the result c = a ∨ b by placing in the result a 1-bit in those digital positions where either a or b or both have a 1-bit, placing zeros in all other positions.

```
Thus if      a = 1100
and          b = 0110
then c = a ∨ b = 1110
```

iii) NOT EQUIVALENT (≢).

Given two operands a and b, form the result c = a ≢ b by placing in the result a 1-bit in those digital positions where the bits in a and b are not the same (i.e. where one has a 1-bit and the other a 0-bit) and placing zeros in the result in all other positions.

```
Thus if      a = 1100
and          b = 0110
then c = a ≢ b = 1010.
```

3.3 Group 3

All the legal instructions of group 3 are concerned with multiplication, the range provided allowing for single-length, double-length and accumulative double-length products.

Function-numbers 30 to 34 use as one operand the 48-bit signed number x. Functions 30 to 33 use also the 48-bit signed number y, while function 33 uses a third operand, namely the double-length signed number z:.

Function 34 uses, as the second operand, the unsigned 15- or 24-bit number Y instead of y.

The function-numbers 35, 36 and 37 are unassigned and are treated as illegal.

Function number 30 [0011 000]

(3) $z_I' = x_I y_I$ (2) $x_I' = x_I y_I$

Single-length product of integers. Regarding x and y as signed integers, form their product, writing the result into Z or X. OVR is set if the result exceeds single-length capacity.

In effect the result of this instruction is the l.s. half of the full double-length product, adjusted to be correctly signed.

If one (or both) of the operands is non-negative and can be represented by 24 bits or less it may be faster to use a 34 instruction (q.v.) with replacement of the Y-address.

Function number 31 [0011 001]

(3) $z_F' = (x_F.y_F)_r$ (2) $x_F' = (x_F.y_F)_r$

Rounded single-length product of fractions. Regarding x and y as fractions, form their product, rounded to single-length, and write the result into Z or X. OVR will be set if and only if
$x_F = y_F = -1.0$.

When this instruction is obeyed the full double-length product in standard form is formed. The m.s. half is then rounded by adding 2^{-47} to it if the l.s. half is equal to or exceeds ½ and is then stored as the result. Thus the result is that integral multiple of 2^{-47} which is nearest to the true product of x_F and y_F and is the algebraically greater if the true product is an odd multiple of 2^{-48}.

<u>Function number 32 [0011 010]</u>

(3) z:′ = xy (2) x:′ = xy

Double-length product. Form the full double length product of x and y, writing the result into Z and Z+1 or X and X+1. The result is in standard form, i.e. with the l.s. half non-negative. OVR is set if and only if $x_F = y_F = -1.0$.

The consequence of the standard form for the d.l. product are illustrated by the table below.

x	y	z′ (or x′)	z*′ (or x*′)
+5	+9	+0	+45
+0.5	+9	+4	+0.5
+5	-9	-1	$2^{47}-45$
-0.5	+9	-5	+0.5
+0.5	+0.875	+0.4375	+0
-0.5	+0.875	-0.4375	+0

If x and y are both fractions or both integers then the result is a double-length fraction or integer respectively. If one of the operands is a fraction and the other is an integer then the product is a standard double-length mid-point number.

<u>Function number 33 [0011 011]</u>

(3) z:′ = z: + xy (2) x:′ = x: + xy

Accumulative multiplication. Form the full double-length product of x and y and add it to the double-length number z: in Z and Z+1 (3-address) or to x: in X and X+1 (2-address). OVR is set if the result exceeds double-length capacity.

The result is left in standard form, i.e. with z*′ (or x*′) ≥ 0 but it is not necessary that z: (or x:) should be in standard form initially since a ‘partial justify’ operation is performed on z: (or x:). Overflow caused by the ‘partial justify’ is dealt with correctly. In the 2-address form any carry into x caused by the ‘partial justify’ is not considered in forming the product xy.

It is, of course, essential that the scalings of x, y and z: (or x:) are compatible. The increment xy and the result are correctly signed.

<u>Function number 34 [0011 100]</u>

(3) z′ = xY (2) x′ = xY

Multiply x by Y, writing the result into Z or X. Y is an unsigned 15- or 24-bit number while x is a signed 48-bit number. The result is correctly signed. OVR is set if the result exceeds single-length capacity.

This function is closely related to the 30. The result given is, in effect, the signed l.s. half of the full double-length product of two 48-bit words, the m.s. 24 or 33 bits of one of which are all 0’s.

Thus, if the content of A100 is regarded as an integer i then the instruction

```
34      A100     27     A7
```

will set in A7 the integer equal to 27i.

In certain cases the 34 function can be used instead of the 30. This is so if at least one of the operands is a non-negative 24-bit number. Suppose that such a number j is in A200, while in A100 is a signed 48-bit number k. Then the instruction

```
34      A100    (A200)  A7
```

will set in A7 the signed product jk. This may be faster than the direct instruction

```
          30     A100     A200     A7
```

(see notes on timing below).

The function 34, is, of course, designed primarily for multiplication by program constants, e.g. for multiplying pounds sterling by 240 to obtain the equivalent number of pence.

<u>Function number 35 [0011 101]</u>

<u>Function number 36 [0011 110]</u>

<u>Function number 37 [0011 111]</u>

These instructions are illegal.

3.4 Group 4

The legal instructions of group 4 are concerned with the division process. Function-numbers 46 and 47 are unassigned and if encountered, will cause entry to the Monitor Routine, the program being suspended.

The functions numbered 40 to 45 each use two operands. One of these is the signed 48-bit number y. The other is either the signed 48-bit number x or the standard double-length number x:. In all cases x or x: is the dividend (numerator) and y is the divisor (denominator), i.e. the instructions divide x or x: by y.

In all cases if y = 0 the Monitor Program will be entered and the program suspended.

The results occupy either one word or two, depending on the actual function used. Single word results are written into Z in 3-address form and into X in 2-address form. Results which occupy two words are written into Z and Z+1 in 3-address form and into X and X+1 in 2-address form. Nothing should be assumed about the quotient and remainder of a division instruction which overflows.

Function Number 40 [0100 000]

(3) $z_I' + (z^*_I'/y_I) = x_I/y_I; \quad 0 \le z^*_I'/y_I < 1$

(2) As 3-address with x_I' and x^*_I' in place of z_I' and z^*_I' respectively.

Unrounded division of integers. Divide x_I by y_I, writing the unrounded integral quotient into Z and the remainder into Z+1. The remainder is always of the same sign as y and numerically less than y. The quotient and remainder, in 3-address form, satisfy

$$z_I'.y_I + z^*_I' = x_I$$

OVR is set if x is -2^{47} and y is -1.

The results obtained for selected values of x_I and y_I are tabulated below.

x_I	y_I	z_I'	z^*_I'
+37	+5	+7	+2
-37	+5	-8	+3
+37	-5	-8	-3
-37	-5	+7	-2
+1	+5	0	+1
+1	-5	-1	-4

Note also that:-

i) z_I' = quotient, $z^*_F{}'$ = remainder when x_F is divided y_F.

ii) z_F' = unrounded quotient when x_F is divided by y_I.

Although the remainder is such that it can in turn be divided by y to give the non-negative l.s. half of a double-length quotient it is not necessary to do so, since the 42-instruction gives a result in this form.

Function Number 41 [0100 001]

(3) $z_I' = (x_I/y_I)_r$ or $(x_F/y_F)_r$; $-½ \le x/y - z_I' < ½$

(2) As 3-address but with x_I' in place of z_I'.

Rounded division of integers. Divide x by y, to produce a rounded integral quotient, writing this into Z or X.

The rounding is such that if the rounded quotient is subtracted from the true quotient of x_I/y_I the difference is numerically less than ½ or equal to -½.**

In fact the rounded quotient is the nearest integer to x/y and is the algebraically greater if x/y is an odd multiple of ½.

The results of some selected cases are tabulated below.

x_I	y_I	z_I'
+13	+2	+7
-13	+2	-6
+13	-2	-6
-13	-2	+7
+37	+5	+7
-37	+5	-7

OVR is set if $x = -2^{47}$ and $y = -1$.

** Editors note: This should be "and greater than or equal to -½"

Function Number 42 [0010 010]

(3) $z{:}_M' = (x_I/y_I)_r$ or $(x_F/y_F)_r$; $-\frac{1}{2}\,\varepsilon \le (x/y) - z{:}_M' < \frac{1}{2}\,\varepsilon$

(2) As 3-address but with $x{:}_M'$ in place of $z{:}_M'$.

Division with rounded double-length quotient. Divide x by y to give a mixed-number quotient, writing the signed integral part into Z or X and the non-negative fractional part into Z+1 or X+1. OVR is set if

$$x_I = -2^{47} \text{ and } y_I = -1$$

(or the fractional interpretations of those words).

The results given in some selected cases are tabulated below.

x_I	y_I	z_I'	$z^*{}_F'$
+37	+8	+4	+0.625
-37	+8	-5	+0.375
+37	-8	-5	+0.375
-37	-8	+4	+0.625
+1	+8	0	+0.125
+1	-8	-1	+0.875

The quotient is rounded to the nearest multiple of 2^{-47} and to the algebraically greater if the true value of x/y is an odd multiple of 2^{-48}.

Note also that:-

i) $z{:}_F'$ = rounded d.l. quotient when x_F is divided by y_I;

ii) $z{:}_I'$ = rounded d.l. quotient when x_I is divided by y_F.

Function Number 43 [0100 011]

(3) $z_F' = (x_F/y_F)_r$ or $(x_I/y_I)_r$; $-\frac{1}{2}\,\varepsilon \leq x/y - z_F' < \frac{1}{2}\,\varepsilon$

$|x| < |y|$ or $x = -y$

(2) As 3-address but with x_F' in place of z_F'.

Rounded division of x by y producing a fractional quotient, which is written into Z, or X. The operands must be such that x is numerically less than y or is equal to -y. OVR is set if neither of these conditions is satisfied. If x and y have the same scaling factor (e.g. if both are fractions or if both are integers) than z′ is a fraction.

The result is the multiple of 2^{-47} nearest to the true value of x/y and is the algebraically larger if x/y is an odd multiple of 2^{-48}.

Note also that $z_I' = (x_I/y_F)_r$

This instruction can be used, following a 44-instruction (q.v.), to produce a rounded standard double-length quotient when x: is divided by y.

Function Number 44 [0100 100]

(3) $z_I' + (z^*_I{}'/y_I) = x:_I/y_I$; $0 \leq z^*_I{}'/y_I < 1$

(2) As 3-address but with x_I' and $x^*_I{}'$ in place of z_I' and $z^*_I{}'$.

Unrounded division with double-length dividend (numerator).

The standard d.l. number x: is divided by the s.l. number y producing an integral quotient and a remainder. The quotient is written into Z or X and the remainder into Z+1 or X+1. OVR is set if z′ exceeds single-length capacity. If x: is not in standard form the OMP is entered; the OMP does a 'partial justify' and obeys the instruction.

For comments and examples relating to the values of quotients and remainders please see under function-number 40; in essence functions 40 and 44 differ only in using single-and double-length dividends respectively.

If the remainder from a 44-instruction is divided by the original divisor (using a 43-instruction) the result is necessarily a non-negative fraction. Thus if a is a standard d.l. number in A100 and A101 and if b is an s.l. number in A200 then after obeying the instructions

```
        44      A100     A200     A7

        43      A8       A200
```

A7 and A8 will together contain the standard d.l. mid-point (rounded) value of a/b, with the signed integral part in A7 and the non-negative fractional part in A8.

<u>Function number 45 [0100 101]</u>

(3) $z_F' = (x:_F/y_F)_r$ or $(x:_M/y_I)_r$; $-\frac{1}{2}\varepsilon \leq (x:_F/y_F) - z_F' < \frac{1}{2}\varepsilon$

(2) As 3-address but with x_F' in place of z_F'

Rounded division with double-length dividend. Divide the standard d.l. number x: by the s.l. number y to produce a rounded fractional quotient, writing this quotient into Z or X. The operands must be such that $|x:_F| < |y_F|$ or $x:_F = -y_F$. OVR is set if neither of these conditions is satisfied. If x: is not in standard form, the O&MP is entered; the OMP does a 'partial justify' and obeys the instruction.

The result is that multiple of 2^{-47} which is nearest to the true value of $x:_F/y_F$ being the algebraically larger if the true quotient is an odd multiple of 2^{-48}.

Note also that

$z_I' = (x:_M/y_F)_r$ or $(x:_I/y_I)_r$; $-\frac{1}{2} \leq (x:_M/y_F) - z_I' < \frac{1}{2}$

i.e. the quotient may be regarded as the rounded integral quotient when $x:_M$ is divided by y_F or when $x:_I$ is divided by y_I.

<u>Function number 46 [0100 110]</u>

<u>Function number 47 [0100 111]</u>

Illegal instructions

3.5 Group 5.

The instructions of this group are concerned with simple shifts. Facilities are provided for shifting single- and double-length quantities.

In all these instructions the quantity to be shifted is that in register X if single-length or that in registers X and X+l if double-length. If the instruction is of 3-address type the result is written into Z (or Z and Z+1) while in 2-address type the result is written into X (or X and X+1).

The number of places by which the quantity is shifted (the 'shift number') is Y, which may be a number written in the instruction or may be obtained by replacement; whichever alternative is used, the shift number may also be modified using internal and/or external modification. If Y is written in the instruction as a negative integer it is treated as such, i.e. although it is stored as 15 bits, the sign bit is repeated in the control circuits when the instruction comes to be obeyed.*

Generally a shift may be 'up' or 'down'. (Alternative terms are 'left' and 'right' respectively). If a quantity is shifted up, each bit is moved through Y digit-positions towards the more-significant end, while if it is shifted down, each bit is moved through Y digit-positions towards the less-significant end. In general, such movement results in some digit-positions becoming vacant and in some bits being moved off the end of the word. In double-length shifts some bits are transferred from one word to the other. The action in such cases depends on whether the shift is logical or arithmetical.

(a) Logical, single-length. Those bits which are moved off the end of the word are lost; those digit positions vacated at the other end of the word are filled with zeros. OVR cannot be set.

(b) Logical, double-length. Action at the ends of the d.l. quantity is as stated under (a) above. Those bits which pass from one word to the other do so without special treatment; the effect is as though the two registers containing the d.l. quantity had been temporarily joined to form a single 96-bit register. OVR cannot be set.

(c) Arithmetical, single-length.

  i) Shift up. Digit positions vacated at the l.s. end are filled with zeros. OVR is set if the result exceeds capacity (i.e. if significant 1-bits are shifted off the m.s. end, or if the sign apparently changes).

  ii) Shift down. The result is rounded; if the last bit shifted off the l.s. end is a 1, a 1-bit is added to the least-significant end after the shift before the

* For fuller details of formation of shift-number, see section 3.5, page 3.

result is written away. The sign-bit is propagated; digit-positions vacated at the m.s. end are filled with copies of the sign-bit. These special actions lead to the result being correctly signed and rounded. OVR cannot be set by a shift down.

(d) Arithmetical, double-length. The action at the ends of the d.l. number is as stated in (c) above. In addition a double-length arithmetical shift includes a 'partial-justify' operation. If the l.s. word of the pair is negative (i.e its sign-bit is a 1) the d.l. quantity is adjusted so that the l.s. word is non-negative (its sign-bit is a 0), the m.s. word being adjusted so that the value of the double-length quantity is unchanged. Thereafter the sign-bit of the l.s. word is excluded from the shifting process. Those bits which pass from one word to the other effectively 'skip over' D0 of the l.s. word. For example, if a d.l. quantity is shifted arithmetically up by one place, then D1 of the l.s. word of the original quantity is shifted to become D47 of the m.s. word of the result, passing over D0 of the l.s. word. OVR cannot be set by a shift down but will be set if the result of an upward shift exceeds capacity. The 'partial justify' is still performed (and may set OVR) if Y=0.

In all instructions of this group, the shift number Y is treated as signed. Thus if Y is negative, a function-number which nominally produces a shift up will in fact produce a shift down and vice-versa.

Shift Number

The shift-number is formed as described below.

i) If Y is not replaced, the 15-bit representation of Y is extended upwards to 24 bits by repeating the m.s. bit of Y.

ii) If Y is replaced, it is replaced by y_m as usual. (Y may have been pre-modified (116-instruction) before the replacement).

iii) If Y is internally modified (2-address modified instruction), z_m is added to the 24-bit result of (i) or (ii).

iv) If Y is externally modified (116- or 117-instruction) and:-

a) the Y-address in the pre-modify instruction is not replaced: the 15-bit Y, right-justified , from the pre-modify instruction is added to the 24-bit result of (i), (ii) or (iii).

b) the Y-address in the pre-modify instruction is replaced: the 24-bit modifier in the replaced Y-address of the pre-modify instruction is added to result of (i), (ii) or (iii).

Thus a 24-bit effective shift-number is obtained. If the m.s. bit of this is a 1, the shift-number is negated and the shift takes place in the opposite direction to the nominal sense for the function. Of the 24-bit shift-number (after the negation, if any) the logical 'or' operation is performed between the eighth bit from the l.s. end and each of the 16 m.s. bits, producing an 8-bit shift-number; in effect, if any of the 17 m.s. bits is a 1, the eighth bit is made a 1. Hence if the 24-bit shift number exceeds, in modulus, 255, the actual number of places shifted is 128+ the l.s. bits of the shift number. In such a case the result of a group 5 instruction is always zero but OVR may be set by an arithmetical shift up. A consequence of this treatment of the shift number is that a shift can never take more than one millisecond.

Function number 50 [0101 000]

3-address: $z' = x.2^Y$ 2-address: $x' = x.2^Y$

Multiply the number x by 2^Y, writing the result into Z or X. The result is correctly signed and, if Y is negative, rounded. OVR is set if the result is out of range.

Function number 51 [0101 001]

3-address: $z' = x.2^{-Y}$ 2-address: $x' = x.2^{-Y}$

Divide x by 2^Y writing the result into Z or X. The result is correctly signed and rounded. OVR is set if the result is out of range which can happen only if Y is negative.

Function number 52 [0101 010]

3-address: $z' = x_L$ shifted logically up Y places.

2-address: $x' = x_L$ shifted logically up Y places.

OVR cannot be set.

Function number 53 [0101 011]

3-address: $z' = x_L$ shifted logically down Y places.

2-address: $x' = x_L$ shifted logically down Y places.

OVR cannot be set.

Function number 54 [0101 100]

3-address: $z:' = x:.2^Y$ 2-address: $x:' = x:.2^Y$

Multiply the double-length number x: by 2^Y, writing the result into Z and Z+1 or into X and X+1. The result is correctly signed and, if Y is negative, rounded. OVR is set if the result is out of range.

Note that the result is in standard form, i.e. with the l.s. word non-negative even if the original d.l. number is not in standard form. This means, for example, that if Y = 0 and $x^* < 0$, the result will be such that $z:'$ (or $x:'$) = x: but $z^{*\prime}$ (or $x^{*\prime}$) $\neq x^*$. Thus if, say $x_I = 2^{47}k-j$ then, after a 54-instruction with Y = 0, $z_I' = k-1$ and $z^*_I{}' = 2^{47}-j$ giving $z:_I{}' = 2^{47}(k-1) + (2^{47}-j)$

$$= 2^{47}k - j$$
$$= x:_I$$

If, in a similar case, Y = 1 then the results would be $z_I' = 2k-1$ and $z*_I' = 2^{47}-2j$ giving

$$\begin{aligned} z{:}_I' &= 2^{47}(2k-1) + (2^{47} - 2j) \\ &= 2(2^{47}k - j) \\ &= 2x{:}_I \end{aligned}$$

Function number 55 [0101 101]

3-address: $z{:}' = x{:}.2^{-Y}$ 2-address: $x{:}' = x{:}.2^{-Y}$

Divide the double-length number x: by 2^Y, writing the result into Z and Z+1 or into X and X+1 . The result is correctly signed and rounded. OVR is set if the result is out of range which can happen only if Y is negative. The result is left in standard form (see discussion under function-number 54 above). If the 'partial justify' sets OVR, the correct result is still obtained, i.e. after a shift down OVR will left clear.

Function number 56 [0101 110]

3-address: $z{:}' = x{:}_L$ shifted logically up Y places.
2-address: x:' = x:L shifted logically up Y places.

OVR cannot be set

Note that a special use of a 56-instruction is typified by

```
        56        A100        0        A7
```

which sets (A7)' = (A100) and (A8)' = (A101), i.e. copies two consecutive words into two consecutive accumulators by a single instruction, taking some 72 microsecs. as against 96 microsecs. for two 04-instructions. This use of a 56-instruction is permissible since there is no 'partial-justify' operation when performing a logical shift.

Function number 57 [0101 111]

3-address: $z{:}' = x{:}_L$ shifted logically down Y places.
2-address: $x{:}' = x{:}_L$ shifted logically down Y places.

OVR cannot be set.

A 57-instruction may be used for a two-word copy as discussed under function-number 56 above.

Another special use of a 57-instruction is to clear to zero two consecutive accumulators. This done by e.g.

```
        57        A0        48        A7
```

which takes some 72 microsecs. (the last 48 places shifted take no time). Two 14-instructions take the same time but occupy one more register and a 143-instruction clearing two words takes about 80 microsecs.

3.6 Group 6

The instructions of this group are 'jump' or 'discrimination' instructions. With function-numbers 60 to 65 a comparison is made between the signed 48-bit numbers y and z while with function-numbers 66 and 67 the comparison is between pY and z. (In 2-address form z = 0 for all function-numbers). Depending on the result of this comparison the next instruction obeyed is either that stored in register X or the one stored in the next-higher numbered register to that containing the 'jump' instruction itself. This is achieved by arranging for the control-number either to be set to X (strictly, to the machine-address represented by X, which may be a Basic or a Symbolic Address) or to be increased by unity from its current value.

The replacement and modification facilities operate in the normal way; in particular the destination address X may be replaced and/or modified. Such treatment of the X-address yields a 'switch', that is, a single instruction which causes the computer to follow one or another of several possible courses of action, depending on the result of a comparison between two numbers and on the value of a third number (a modifier). Note that when a jump instruction does not cause a jump, the destination address is not checked for lock-outs or reservations, unless replaced.

<u>Function number 60 [0110 000]</u>

3-address: Jump to X if $y = z$ 2-address: Jump to X if $y=0$.

<u>Function number 61 [0110 001]</u>

3-address: Jump to X if $y \neq z$ 2-address: Jump to X if $y\neq 0$.

<u>Function number 62 [0110 010]</u>

3-address: Jump to X if $y > z$ (i.e. $z < y$)
2-address: Jump to X if $y > 0$ (i.e. y positive)

<u>Function number 63 [0110 011]</u>

3-address: Jump to X if $y \ngtr z$ (i.e. $z \geq y$)

2-address: Jump to X if $y \ngtr 0$ (i.e. y negative or zero).

<u>Function number 64 [0110 100]</u>

3-address: Jump to X if $y < z$ (i.e. $z \geq y$)
2-address: Jump to X if $y < 0$ (i.e. y negative, $y_s=1$)

Function number 65 [0110 101]

3-address: Jump to X if $y \nless z$ (i.e. $z \leq y$)

2-address: Jump to X if $y \nless 0$ (i.e. y non-negative, $y_s = 0$)

Function number 66 [0110 110]

3-address: Jump to X if pY = z
2-address: Jump to X if pY = 0

Although pY can be the content of any of the pseudo-registers, the most common use of this instruction is to test the state of the overflow indicator, by reference to P4, P5, P6 or P7.

Function number 67 [0110 111]

3-address: Jump to X if $pY \neq z$

2-address: Jump to X if $pY \neq 0$.

See comment under function-number 66.

3.7 Group 7

The instructions of this group are concerned with 'jumps' in a manner similar to those of group 6. With the present group, with the exceptions of function-numbers 76 and 77, the comparison is made between the 15- or 24-bit quantity Y and the 24-bit modifier z_m in accumulator z (or between Y and zero if the instruction is of 2-address type).

If Y is used without replacement or modification, it is essentially the 15-bit representation of the Y-address written in the instruction (it may be written as an integer, signed or unsigned, as a Basic or a Symbolic Address or as a combination of any or all of these three) and is extended to 24-bits in the arithmetic unit by prefixing the 15-bit quantity by nine 0-bits. If, however, Y is modified and/or replaced it is a genuine 24-bit quantity. In all cases both Y and z_m are treated as unsigned.

Note that when a jump instruction does not cause a jump the destination address is not checked for lock-outs or for violation of reservations unless replaced.

Function number 70 [0111 000]

3-address: Jump to X if Y = z_m

2-address: Jump to X if Y = 0

Some examples follow.

The instruction

```
     70     A200       50       A7
```

will cause a jump to A200 if $A7_m = 50$, the instruction

```
     70     A200     (A100)     A7
```

will cause a jump to A200 if $A7_m = A100_m$, while the instruction

```
     70     A200     (A100)
```

will cause a jump if $A100_m = 0$.

An instruction such as

```
     70Y        A200         (A100)       A7
```

will cause a jump if the l.s. 24-bits of [A100] + [A7] are zero. This will be so either if

i) both $[A100]_m$ and $[A7]_m$ are zero

or ii) if $[A100]_m + [A7]_m = 2^{24}$, i.e. the addition produces 24 0-bits and a carry into the 25th place, this carry bit not being recorded by the computer logic.

Note that an instruction such as

```
     70         A200         0
```

causes an unconditional jump but function-number 75 (q.v.) is to be preferred for this purpose.

<u>Function-number 71 [0111 001]</u>

3-address: Jump to X if $Y \neq z_m$
2-address: Jump to X if $Y \neq 0$

The examples given for function number 70 are also applicable to function-number 71, with the equalities replaced by inequalities.

<u>Function number 72 [0111 010]</u>

3-address: Jump to X if $Y > z_m$ (i.e. $z_m < Y$)
2-address: Jump to X if $Y > 0$

Note that since Y is treated as unsigned, the 2-address forms of the 71- and 72-instructions are equivalent in effect.

<u>Function number 73 [0111 011]</u>

3-address; Jump to X if $Y \not> z_m$ (i.e. $z_m \geq Y$)

2-address; Jump to X if $Y \not> 0$

Note that since Y is treated as unsigned, the 2-address forms of the 70- and 73-instructions are equivalent in effect.

<u>Function number 74 [0111 100]</u>

3-address: Jump to X if $Y < z_m$ (i.e. $z_m > Y$)
2-address: Jump to X if $Y < 0$.

Note that since Y is treated as unsigned, a 2-address 74-instruction will never cause a jump; it is the recommended dummy.

<u>Function number 75 [0111 101]</u>

3-address: Jump to X if $Y \not< z_m$ (i.e. $z_m \leq Y$)

2-address: Jump to X if $Y \not< 0$.

Since Y is treated as unsigned, a 2-address 75-instruction will always cause a jump, regardless of the actual value of Y. Therefore it is recommended as the standard 'unconditional jump'. The Y-address field may, if desired, be used to store any 15-bit quantity.

<u>Function number 76 [0111 110]</u>

3-address: Jump to X if $\neg y \ \& \ z = 0$.

2-address: Jump to X if $Y = 0$ (not recommended)

With this function-number the decision whether to jump or not is made on the result of a logical operation involving two operands, z and y, of which y is usually a program constant. The 76-instruction is designed primarily for detecting field overflow. Suppose, for example that y is $0^7\ 1^7\ 0^{34}$, and is stored in A100. Then the instruction

```
          76   A200      A100      A7
```

will cause a jump to A200 if all of the first 7 bits and the last 34 bits in A7 are zeros, that is, if all the 1-bits in A7 are contained within the field covered by the 1-bits in A100. The mask y may contain any number of 1-bits distributed in any desired way within the word. Note that the logical negation of y occurs only in the arithmetic unit and associated circuits and the stored word y is not affected.

<u>Function number 77 [0111 111]</u>

3-address: Jump to X if $\neg y \ \& \ z \neq 0$

2-address: Jump to X if $Y \neq 0$ (not recommended)

This function-number is closely related to the 76-function. It will cause a jump if there is one or more 1-bit outside the field(s) covered by the 1-bits in the mask y. For other comments see function number 76.

3.8 Group 8

The instructions in this group fall into three logically distinct sub-groups, respectively concerned with:-

i) (80 to 83) counting, and stepping modifiers.

ii) (84 and 85) searching for specified characters.

iii) (86 and 87) entering and leaving subroutines.

3.8.1 Function numbers 80 to 83.

By means of these functions a counter may be increased or decreased by unity each time a loop of instructions is obeyed until that loop has been traversed the desired number of times; this event is indicated by the value of the counter reaching some specified value.

In 3-address form the counter is the integer in accumulator Z, while in 2-address form it is the integer in register Y: if the counter is in an accumulator it may also be used as a modifier within the loop (or elsewhere).

The completion of the cycling process is determined, in 3-address form, by comparing the new value of z with the value of Y and, in 2-address form, by comparing the new value of y with zero. In making these comparisons only the modifier half of z or y is taken, limiting the value of the counter to $2^{24}-1$ (≈16½ million).

The value of Y may be set as the address field of the written instruction, in which case it is a 15-bit quantity, or by replacement or pre-modification, when it is a 24-bit quantity.

Function number 80 [1000 000]

3-address: $z' = z + 1$; jump to X if $z_m' = Y$

2-address: $y' = y + 1$; jump to X if $y_m' = 0$

Increase the counter by unity and jump if the result is equal to Y (3-address) or zero (2-address). OVR is set if z′ (or y′) is out of range.

Function number 81 [1000 001]

3-address: $z' = z + 1$; jump to X if $z_m' \neq Y$

2-address: $y' = y + 1$; jump to X if $y_m' \neq 0$

Increase the counter by unity and jump if the result does not equal Y (3-address) or zero (2-address). OVR is set if z′ or y′ is out of range.

A particular use of this instruction is in connection with a 'loop' which is to be obeyed once only. For example, a print sub-routine may be used repeatedly but the layout of the printed page may require that the action the first time the routine is used shall differ from the action on subsequent occasions. If, then, a counter is set initially to -1, an 81-instruction can be used to secure the alternative actions.

Function number 82 [1000 010]

3-address: $z' = z-1$; jump to X if $z_m' = Y$

2-address: $y' = y-1$; jump to X if $y_m' = 0$

Decrease the counter by unity and jump if the result is not equal to Y (3-address) or zero (2-address). OVR is set if z′ or y′ is out of range.

Function number 83 [1000 011]

3-address: $z' = z-1$; jump to X if $z_m' \neq Y$

2-address: $y' = y-1$; jump to X if $y_m' \neq 0$

Decrease the counter by unity and jump if the result is not equal to Y (3-address) or zero (2-address). OVR is set if z′ or y′ is out of range.

3.8.2 Function numbers 84 and 85.

These two functions are used to determine whether or not a certain 6-bit character is present in a word. The character sought is specified by its value in the chosen internal code; this value is written as the Z-address in the instruction. The hardware then tests each of the successive 6-bit fields in y for equivalence with Z. The instructions indicate only whether the character is present or not; they give no indication regarding the number of times that character appears in the word, or its position within the word.

If the instruction is of 2-address type then Z is treated as zero and the character SP is sought.

<u>Function number 84 [1000 100]</u>

3-address: Jump to X if character value Z is in y

2-address: Jump to X if character <u>SP</u> is in y

<u>Function number 85 [1000 101]</u>

3-address: Jump to X if character value Z is not in y

2-address: Jump to X if character <u>SP</u> is not in y

3.8.3. Function numbers 86 and 87

86- and 87- instructions are designed primarily for entering and leaving sub-routines. In both cases, the 2-address form is the more conventional. In 3-address form Y is irrelevant and is not checked, for lockouts or violation of reservations unless replaced (see 2.2.23 and 2.2.24)

Function Number 86 [1000 110]

3-address: $z_m' = c + 1$, $z_s' = OVR$, $OVR' = 0$, jump to X

2-address: $y_m' = c + 1$, $y_s' = OVR$, $OVR' = 0$, jump to X

where c = control number (address of current instruction).

Enter subroutine at address X, leaving in y_m or z_m the address of the word following the 86- instruction. Before jumping, set the sign bit of y or z to 1 if OVR is set or to 0 if OVR is clear, leaving OVR clear. The address left in y or z is the link for use by an 87- instruction in the subroutine.

An 86- instruction gives an unconditional jump and can be used as such if it is desired simultaneously to ensure that OVR is cleared.

The X- and- Y- addresses can be replaced or modified if desired: replacement or modification of the X-address permits the subroutine to be entered at a variable address from the same point in the main program.

A three-address 86- instruction can be used to enter a subroutine and set a fifteen-bit parameter (in the Y-address) for use within the subroutine. Thus if the 86-instruction is

```
          86    SUB/E    PARAM    SUB/L
```

then the parameter can be copied into X1 by:

```
          04Y     X1     -1     L
```

In this case the register holding the link must be an accumulator. Note that this copies the whole 86-instruction into X1: it does not isolate the parameter.

Function Number 87 [1000 111]

3-address: $OVR' = OVR \vee z_s$, jump to $X + z_m$

2-address: $OVR' = OVR \vee y_s$, jump to $X + y_m$

Leave subroutine: on entry to the subroutine by an 86-instruction, z_m or y_m was set to contain the address of the next word. Thus, when an 87-instruction is obeyed within the subroutine, an unconditional jump occurs to the word X+1 words after the 86-instruction. Also the state of overflow when the 86-instruction was obeyed, is recorded in the sign bit of y or z. If OVR was set immediately before entry to the subroutine and/or OVR is set (and not subsequently cleared) during execution of the subroutine, then OVR is left set after obeying the 87-instruction, otherwise OVR is left clear.

The X- and Y-addresses can be replaced or modified if desired: replacement or modification of the X-address permits return to the main-program at a variable address.

3.9 Group 9

The functions of group 9 are concerned with floating-point arithmetic. Function-number 96 is unassigned and therefore illegal. For the other functions, the operands must be in standard single-length floating-point form and the result is given in the same form (except for function-number 97). Any attempt to obey floating-point instructions with operands not in the standard form causes suspension due to impermissible operand. Floating-point operations may be rounded or unrounded at the programmer's discretion (see section 5.2.5).

In standard floating-point form, the m.s. 40 bits represent x_a, the 'argument' or 'fixed-point part'. This is a signed fraction which is either zero or lies within one of the ranges

$$\tfrac{1}{2} \le x_a < 1 \quad \text{or} \quad -1 \le x_a < -\tfrac{1}{2}$$

Since the arithmetic unit can accept one bit of overflow, it is possible to, say, add two numbers to produce a sum exceeding +1 and still have the correct floating-point result.

The l.s. 8 bits of the word represent the 'characteristic' x_k. This is a non-negative integer in the range 0 to 255 and such that the exponent $x_e = x_k - 128$. Then the complete floating-point number x_G has the value

$$x_G = x_a . 2^{x_e}$$

where $-128 \le x_e < 127$.

If a floating-point number would be in the range

$$-2^{-129} \le x_G < 2^{-129}$$

then it is stored as zero (i.e. a clear word). Production of such a result is termed 'floating-point underflow'.

If an instruction would produce a result greater than or equal to 2^{127} or less than -2^{127}, floating-point overflow results and either the overflow indicator is set or monitoring action follows (see sections 5.2.3 and 5.3.23). This happens if any of the functions 90 to 95 would produce a result outside the above limits. If the program is not monitoring on floating point overflow the result of the instruction is zero.

Note that there is no 'copy' instruction, such as $x_G' = y_G$: the 04 function performs this operation. Function-numbers 102 and 103 are also associated with floating-point operations.

<u>Function-number 90 [1001 000]</u>

3-address: $z_G' = x_G + y_G$ 2-address: $x_G' = x_G + y_G$

Add the floating-point numbers in X and Y, writing the result into Z or X.

<u>Function-number 91 [1001 001]</u>

3-address: $z_G' = x_G - y_G$ 2-address: $x_G' = x_G - y_G$

Subtract the floating-point number in Y from that in X, writing the result into Z or X.

<u>Function-number 92 [1001 010]</u>

3-address: $z_G' = y_G - x_G$ 2-address: $x_G' = y_G - x_G$

Subtract the floating-point number in X from that in Y, writing the result into Z or X.

<u>Function-number 93 [1001 011]</u>

3-address: $z_G' = - y_G$ 2-address: $x_G' = - y_G$

Negate the floating-point number in Y, writing the result into Z or X. OVR will be set if $y_G = -2^{127}$. If $y_G = 2^{-127}$ underflow will occur.

In 3-address form, X is irrelevant and it is recommended that this form should not be used. If the 3-address is used, the X-address is not checked for lock-out or reservation violation unless replaced. In 2-address form, x_G is read from the store but is not used as an operand.

Function-number 94 [1001 100]

3-address: $z_G' = x_G.y_G$ 2-address: $x_G' = x_G.y_G$

Form the product of the numbers in X and Y, writing the result into Z or X, all numbers being in single-length floating-point form. OVR is set if the result exceeds capacity.

Function--number 95 [1001 101]

3-address: $z_G' = x_G/y_G$ 2-address: $x_G' = x_G/y_G$

Divide the number in X by that in Y, writing the result into Z or X, all numbers being in single-length floating-point form. OVR is set if the result exceeds capacity. If y=0 the program is suspended as with other division instructions (see section 3.4).

Function-number 96 [1001 110]

3-address: Unassigned, illegal 2-address: Unassigned, illegal

Function-number 97 [1001 111]

3-address: z_I' = number of places through which $(x_G - y_G)$ must be shifted up to produce a result in standard form.

2-address: As 3-address, but with x_I' = number of places.

Form (x_G-y_G) and count the number of places through which the result must be shifted up to bring the argument within one or the other of the standard ranges, then set this shift-number as an integer in Z or X. Note that (x_G-y_G) is formed only within the arithmetical unit and is not stored. OVR cannot be set.

The main purpose of this function is to measure the closeness to which the floating-point numbers agree, e.g. to decide when an iterative process has been taken to sufficient accuracy.

If x_G and y_G are exactly equal then

$$z_I' \text{ or } x_I' = 47$$

In borderline cases the number produced may differ by one from that one would expect from the formula. The details of this are too complicated to give here or to use in a programming trick.

If x_G-y_G would need to be shifted down to bring it within standard form the result produced is -1.

3.10 Group 10

The legal functions in this group are concerned with converting data from one form to another within the working store.

Function-number [1010 000]

(See also Section 4.4.6)

In the description of the effects of this function, the following notation is used, in addition to that specified in Section 0.4.

x_i - the i-th character in the word x; i=0(l)7

y_i - the i-th character in the word y; i=0(l)7

n - contents of a register in the arithmetic unit.

d_i - the value of the l.s. 4 bits of x_i

G_i - the m.s. 2 bits of x_i

r_i - the value of the l.s. 4 bits of y_i

T_i - the m.s. 2 bits of y_i

F - denotes a failure; the Monitor Routine is entered and the failure is regarded as being caused by an impermissible operand (see 5.2).

3-address: $z' = r_7(r_6(.....(r_1(r_0z+d_0)+d_1)+....)+d_7$
2-address; $x' = r_7(r_6(.....(r_1d_0+d_1)+d_2)+...)+d_7$

OVR will be set if the binary integer would exceed single-length capacity. This cannot happen from a single 2-address 100-instruction but may happen if using two 100-instructions as in the example later in this section. Given a number, as 6-bit characters, in X, convert it to a binary integer according to radices set in Y, writing the result into Z or X. In 3-address form the initial content of Z is taken into account in forming the binary integer but not in 2-address form.

In essence, the function operates as follows:- in 3-address form z is copied into a register in the A.U.; in 2-address form this register is cleared. The content of this register is then multiplied by r_0, the value of the l.s. 4 bits of character y_0. To the result is added d_0, the value of the l.s. 4 bits of character x_0. Then this result is in turn multiplied by r_1, and d_1 added. This process of multiplication and addition is repeated until all characters have been dealt with.

Since the radices r_i are set by program, the 100-instruction permits conversion of mixed-radix data.

It should be noted that the arithmetical processes involved require that, in the chosen internal code, the digits 0, 1, 2, ..., 11 shall be such that the values of their l.s. 4 bits shall be the value of the digit, e.g. 7 must have 0111 as its l.s. 4 bits.

The various T_i, the m.s. 2 bits in each radix character, are used to prescribe checks and special actions to be made concerning the G_i, the m.s. 2 bits of the appropriate datum-character; namely:-

If T_i is

(i) 00, perform conversion if G_i is 00 or 01
(ii) 01, perform conversion if G_i is also 01
(iii) 10, perform conversion regardless of G_i
(iv) 11, perform conversion regardless of G_i and also treat d_i as zero.

Also, in cases (i), (ii) and (iii) a check is made that $d_i < r_i$. Failure to satisfy any one or more of these checks leads to an 'impermissible operand' interruption followed by entry to the monitor routine.

(A flow-diagram of the full process is given on page 4 of this section.)

These checks are designed primarily to facilitate reading from punched cards. In fact, the 100-function is suitable only for fixed-format data; it should not be used for variable-format data.

Referring to section 4.4 pp 2, 3 and 4, it is seen that if a single hole is punched in a card column (usually representing a digit 0 to 9 in the card code) then the v (or x_i) character formed on reading that column has the value of d_i equal to the number of the row punched while G_i is made 01. Thus, if this character is converted by a radix character with T_i = 01, a check is made that there is one hole, and only one, in rows 0 to 9 of the card, i.e. that the punching represents a decimal digit. If the punching may be a single hole in any row (i.e. a duodecimal digit) then G_i will be 00 (if rows 10 or 11 punched) or 01 (if single punching in rows 0 to 9) and, to convert and check, T_i should be 00.

T_i = 11 may be used to ignore non-numerical characters within the fixed format.

The various possible T_i, namely 00,01,10 and 11 may be set on program input by adding 0, 16, 32 and 48 respectively to the value of r_i. Thus, to set r_i = 10 and T_i = 01, the character y_i can be written as 26 or as 10+16.

If the characters to be converted are in a single word and if the programmer is prepared to overwrite them by the binary integer then a 2-address 100-instruction can be used. If the characters occupy more than one word or if the characters must not be overwritten, the 3-address form should be used (in the latter case, Z must first be cleared to zero).

Thus, suppose that a sterling quantity has been read, in character form into A100 and A101, the m.s. characters being in A100. To produce a single-length binary result when converted to pence, the sum must not exceed £586406201480/10/7 of which the first 7 digits (up to 5864062) may be right-justified in A100, and the last 5 digits of £ together with two digits for shillings and one for pence may be in A101.

To convert this sum to binary pence, the following instructions may be used:

```
14        ACC       0
100       A100      RAD1      ACC
100       A101      RAD2      ACC
```

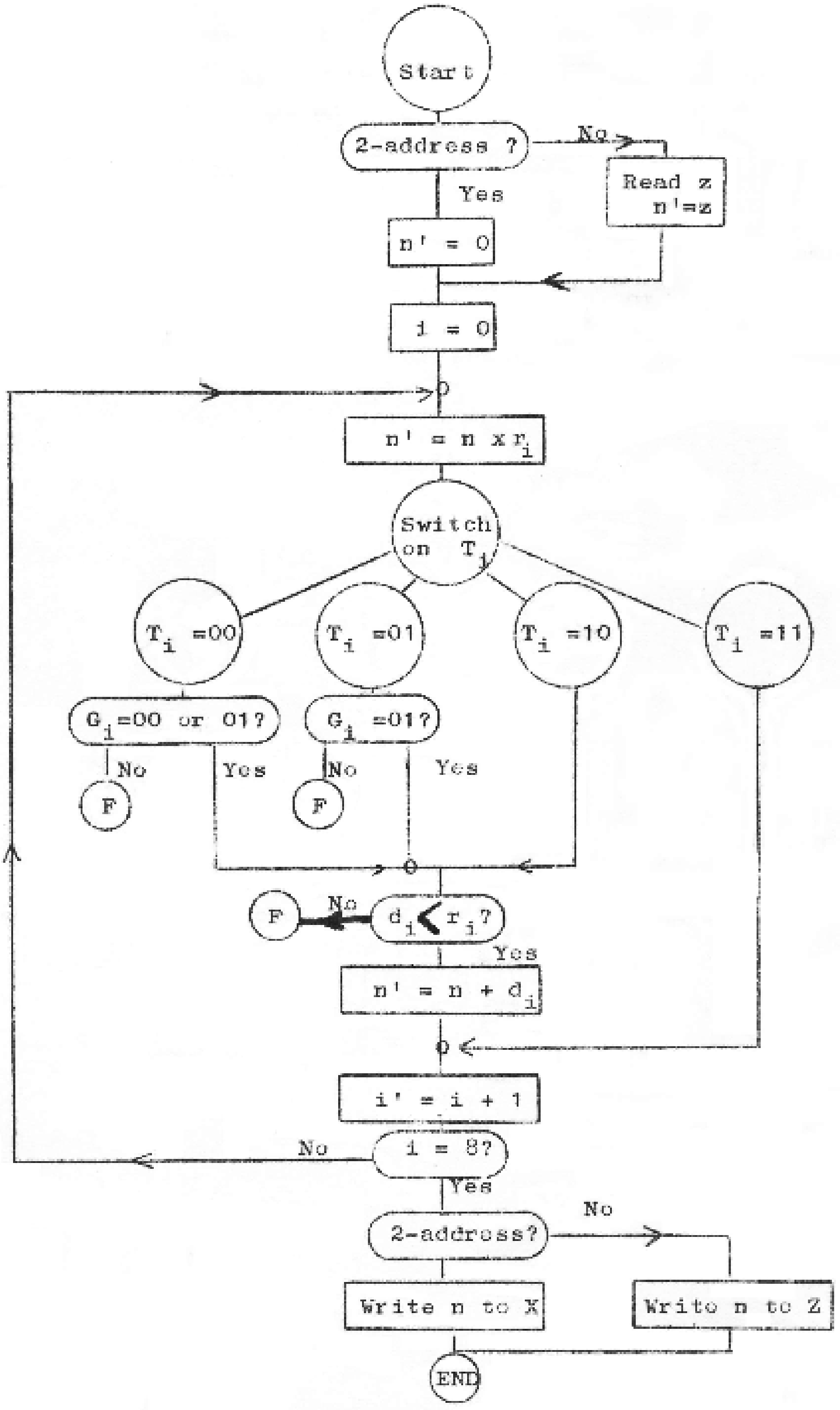
with the radices

```
RAD1)         10, 10, 10, 10, 10, 10, 10, 10
RAD2)         10, 10, 10, 10, 10,  2, 10, 12
```

ignoring any checking requirements.

The first instruction clears an accumulator, ready to receive the result of the first 100-instruction, which converts the m.s. digits of £'s as an ordinary decimal number. The second 100-instruction completes the decimal conversion of the £'s and then carries out the mixed-radix conversions to shillings and pence successively. (If the m.s. data-characters are in an accumulator, the first instruction is not necessary and the second can be 2-address.)

<u>Flow Diagram of the 100 instructions</u>



Function Number 101 [1010 001]

In the description of this function-number, the following notation is used, in addition to that of Section 0.4.

C_i,	the complete 6-bit character formed in the inner loop.
f_i,	the currently remaining fractional part of the number being converted.
d_i,	the value of the l.s. 4 bits of the character formed in the inner loop
r_i,	the value of the l.s. 4 bits of the i-th radix-character
T_i,	the m,s, 2 bits of the i-th radix-character
β,	an indicator. Initially clear, this is set when the first significant character is formed.
θ,	an indicator. It is set if the sign of the number to be converted is not the same as the sign of y, otherwise it is clear.
3-address:	x′ = characters resulting from conversion of z according to information in Y and Y+1.
2-address:	x′ = characters resulting from conversion of x according to information in Y and Y+1

Y contains an number by which z (or x) is to be divided (see Examples in Section 10.2 for particular cases). Y+1 contains characters each of which represents one radix and zero-suppression information for the appropriate character. y=0 causes impermissible operand action.

A flow-diagram for the 101-instruction is given on page 7 of this section. Broadly, the operation of this instruction can be described as follows.

The number N to be converted is read from Z or X. If the signs of N and y are not the same, θ is set to record the fact. N is then divided by y to give a non negative result and the stored quotient is inverted if θ is set. The fractional part is rounded up to 32 binary places.

If the integral part of the quotient is non-zero, OVR and β are set to record the fact that preceding significant digits exist (but will not be produced by this instruction alone).

The fractional part f_0 of the quotient is then multiplied by r_0. In general the result has an integral part d_0 and a fractional part f_1. d_0 is converted to the appropriate character C_0 (by adding 16 if $d_0 \neq 0$ or by special action if $d_0 = 0$), which is stored. Thereafter f_1 is multiplied by r_1, giving d_1, C_1 and f_2; f_2 is multiplied by r_2 and so on until all eight successive characters have been formed. These are then written away in X.

The effects of the various possible T_i are:-

If T_i is 00 a non-significant zero is converted as: <u>SP</u>

If T_i is 01 a non-significant zero is converted as: 0

If T_i is 10 a non-significant zero is converted as: Full stop

If T_i is 11 a non-significant zero is converted as: <u>UC</u>

These T_i are set by adding 0, 16, 32 and 48 respectively to the corresponding radix. <u>It should be noted that a zero produced by radix 1 or 2</u> (radix 1 always produces a zero) is treated as non-significant, even though a significant digit may have been produced previously.

OVR is set if:-

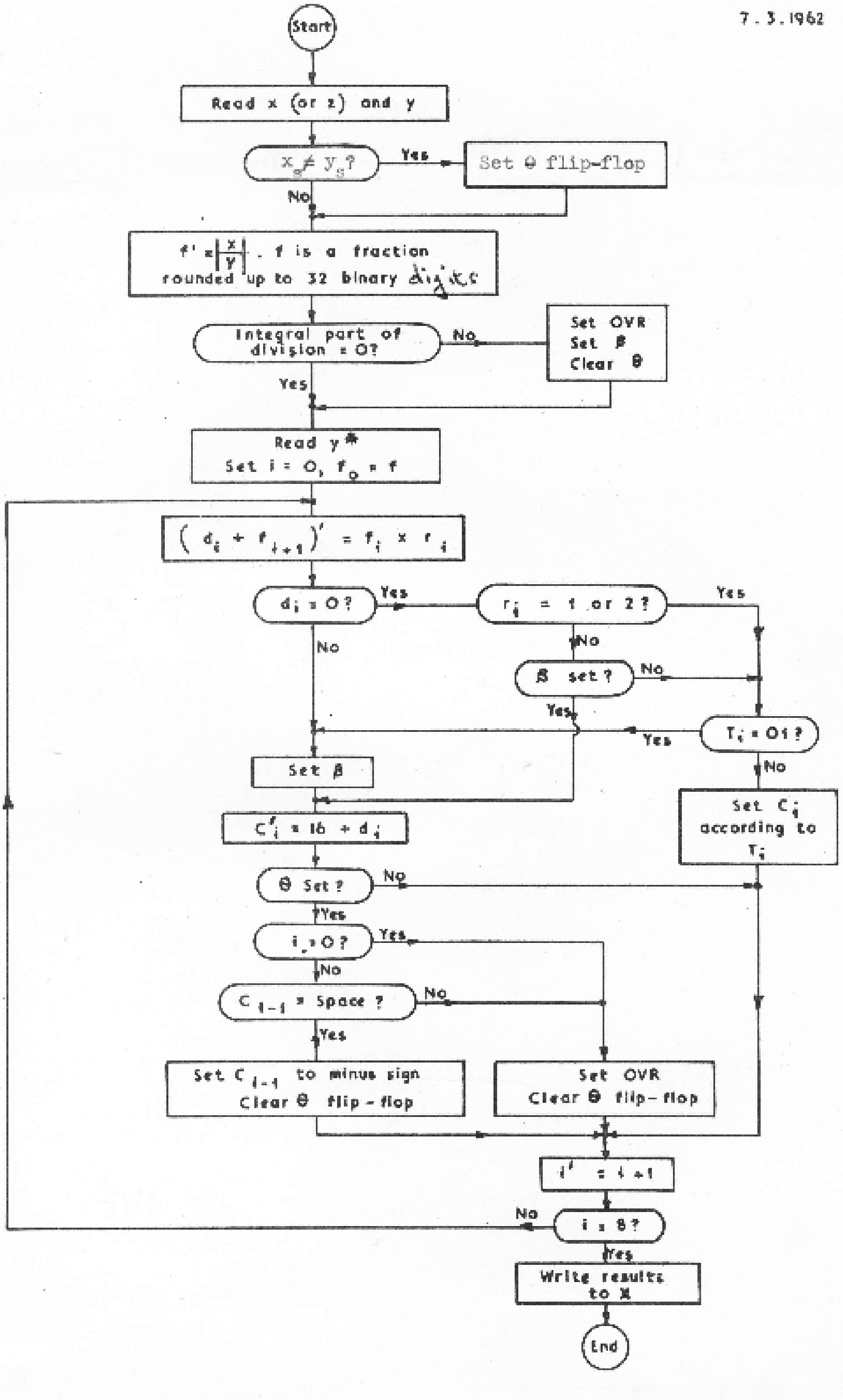
(i) $|y| \not> |z|$ (or $|x|$ if 2-address) i.e. if, when z (or x) is divided by y, the integral part is non-zero.

(ii) If θ is set (i.e. z_s (or x_s) $\neq y_s$) and the character immediately preceding the first significant digit-character is anything other than SP so that it cannot be replaced by a minus-sign.

The number in Y is normally the product of the radices given in Y+1. If this is not so the effect is to divide the number to be converted by $y/\Pi r_i$ - though care must be taken if the result is required exactly as the initial division is only evaluated to 32 binary places. For examples of this use of the 101-instruction see section 10.2.

Note that if one of x or y is negative and no significant digit is stored than the result is undefined.

Flow Diagram of the 101 instruction



Function Number 102 [1010 010]

3-address: $z_G' = x_F.2^Y$ 2-address: $x_G' = x_F.2^Y$

Convert a fixed-point number to floating-point form; multiply the fixed-point fraction in X by 2^Y, Y being treated as signed, and convert the result to standard single-length floating-point form, writing it into Z or X. The conversion may be rounded or unrounded (see section 5.2.5). Floating-point overflow will occur if the result is numerically too large (see section 5.2.3) and underflow if the result is numerically too small (see 3.9).

Y may be regarded as a shift-number and is formed according to the rules given in section 3.5, page 3.

The fraction x_F may have been produced by some arithmetical operation which set OVR. If the overflow was of one bit only, the correct result from 102-instruction is still obtained, as follows.

If, when the 102-instruction is encountered, OVR is clear, the conversion proceeds normally. However, if OVR is set then ±2.0 is added to x_F before it is standardised, the sign being opposite to that of x_F. The number of places needed to standardise the result (negative if OVR set) is then subtracted from 128+Y and this result stored as the exponent (l.s. 8 bits in x_G). Thus it is necessary to ensure that the state of OVR at the instant a 102-instruction is obeyed is the same as its state immediately after the formation of x_F, regardless of what may have happened to OVR in the meantime. Also, of course, derivation of x_F by multiplication, shifting or division may produce more than 1 bit of overflow.

OVR is left clear unless z_G' overflows.

Function Number 103 [1010 011]

3-address: $z_F' = x_G.2^Y$ 2-address: $x_F' = x_G.2^Y$

Convert a floating-point number to fixed-point form: multiply the standard single-length floating-point number in X by 2^Y, Y being treated as signed, convert it to a fixed-point rounded fraction, and write the result into Z or X.

If x is not properly standardised, suspension will occur/due to ‘impermissible operand’. OVR will occur if the result is not within single-length range for a fixed-point fraction.

Y may be regarded as a shift-number and is formed according to the rules of section 3.5 page 3.

<u>Function Number 104 [1010 100]</u>

Conversion after input from punched-card: given four pairs of characters in X, extract the corresponding four characters in the internal code from a code table starting in Y, then:-

3-address: $z_u' = z_m$, z_m = the four characters formed.

2-address: $x_u' = 0$, x_m' = the four characters formed.

OVR cannot be set.

The main details of the conversion and sample code-tables are given in section 4.4, pp.7 to 11. The generation, by hardware, of the four pairs of characters in X is described in section 4.4 pp.2, 3 and 4. Erroneous punching will be detected either automatically on reading the card (case (iii) of section 4.4.3) or by an ‘erase’ character being extracted from the code-table to represent the wrongly-punched column. The latter case can be detected by program by using an 84-or 85-instruction to search for <u>ER</u> in the result of the 104-instruction (by putting Z=63 in the 84- or 85- instruction).

Suppose that a card has been read in Mode 2 into the store area BUFF to BUFF+19, that the table of section 4.4 p.10 is in TAB to TAB+23 and that it is required to convert the data from columns 1 to 8 to alpha-numeric characters in A7. Then the instructions

```
104       BUFF       TAB       A7

104       BUFF+1     TAB       A7

84        ERROR      A7        63
```

will do this and jump to ERROR if any of these eight columns carried illegal punching.

If these columns represent purely numerical data to be converted to binary it is not necessary to use the 104-instruction, since the stored v characters are directly related to the value of the numerical digit punched. Thus if columns 1 to 8 represent the eight digits of a decimal number (with zeros punched), the number may be converted to binary by

```
52      BUFF      24      A7

10      A7        (BUFF+1)

100     A7        RAD
```

with

```
RAD) 10+16, 10+16, 10+16, 10+16, 10+16, 10+16, 10+16, 10+16
```

an error-interruption occurring if the punching in any column is not confined to a single hole in rows 0 to 9, possibly with punching in rows 10 and 11. Normally, a column punched in both curtates represents a non-numerical character. Consequently this method of conversion, by ignoring the u-characters, may convert non-numerical characters (whether punched legitimately or in error) as numerical characters.

The detailed action of the instruction is as follows:

Bits 2-5 of characters 0 and 4 of x are taken in the order

D26,D27,D28,D29,D2,D3,D4,D5

the first five of these bits are used as the word modifier and the last three as a character modifier, relative to Y. This modifier gives a character from the table which is the first of the four characters formed. The other three characters are formed similarly from characters 1 and 5, 2 and 6, 3 and 7 respectively of x. If x has been read from cards in a coded mode the bottom 4 bits of 0 are guaranteed to have a value less than 12, whatever the punching, so that in this case a 24 word table is sufficient. If x is an arbitrary word, 32 words of table may be needed.

Function Number 105 [1010 101]

Function Number 106 [1010 110]

Function Number 107 [1010 111]

These three function-numbers are unassigned and therefore illegal in any form.

3.11 Group 11

Function-number 110 [1011 000]

3-address: $x' = (x \& \neg y) \vee (z \& y)$

2-address: $x' = (x \& \neg y)$

3-address: Insert bit from z into x wherever y has a 1, leaving the bit in x unchanged wherever y has a 0.

2-address: Wherever y has a 1 make the bit in x an 0, wherever y has an 0 leave the bit in x unchanged.

OVR cannot be set.

The word y is used as a mask or 'filter'. The $(x \& \neg y)$ operation leaves the bits of x unchanged in those digit-positions where y has 0's and puts 0's in x wherever y has 1's. In 2-address form, this is the result which is written into X. In 3-address form, the process is continued by making the bit in x the same as that in z in those positions where y has 1's. This result is then written into X.

There is no restriction on the number or positions of the 1-bits in *y*; $y=0^{48}$ and $y = 1^{48}$ yield the trivial cases $x' = x$ and $x' = z$ (or 0) respectively.

Function-number 111 [1011 001]

3-address: $x' = (x \& \neg Y) \vee (z \& Y)$

2-address: $x' = (x \& \neg Y)$

Verbal description as for function-number 110 but with Y in place of y. OVR cannot be set.

The mask Y is confined to the l.s. end of the word (15-bits if Y is not modified or replaced, 24-bits otherwise) but the arrangement of bits within this field is unrestricted.

<u>Function-number 112 [1011 010]</u>

3-address: $x' = (x \,\&\, \neg pY) \vee (z \,\&\, pY)$

2-address: $x' = (x \,\&\, \neg pY)$

Verbal description as for 110 but with pY (content of pseudo-register) in place of y. OVR cannot be set.

<u>Function-number 113 [1011 011]</u>

This function is illegal.

<u>Function number 114 [1011 100]</u>

3-address: $z' = y \quad y' = x$

2-address: $x' = y \quad y' = x$

The three address form copies x and y to Y and Z respectively; the two address form interchanges x and y. The actual order of operations in the three address form is: read x, read y, write z′, write y′. Thus the instruction

```
                    114       A1        A2      A2
is equivalent to:   04        A2        A1
```

The most common use of the three address form is to search through a list where a pointer to the previous entry must be kept. The instruction

```
                    114       (A1)      A1      A2
```

has the effect of putting the next pointer into A1, preserving the previous one in A2.

Function-number 115 [1011 101]

3-address: $z' = x +$ (Y shifted cyclically right 6 places) with end around carry.

2-address: as 3-address but with x' in place of z'.

Add the l.s. 6 bits of Y to the m.s. 6 bits of x and the rest of Y to x_m with end around carry, writing the result to Z, 3-address, or to X, 2-address. Note that the end-around carry continues until all carries have died down; e.g. if A1 contains the integer -1 then after the instruction 115 A1 64 it will contain the integer +1. The instruction is chiefly used for stepping character modifiers as used with the 122 and 123 instructions (see section 3.12). It cannot set overflow.

Function number 116 [1011 110]

Add X, Y to the X and Y addresses of the next instruction before any replacement in that instruction.

Function-number 117 [1011 111]

Add X, Y to the X and Y addresses of the next instruction after any replacement in that instruction.

These two instructions are 'premodify' instructions - they are used to get complex forms of address modification which cannot be obtained with internal modification and replacement only. The instructions may themselves be replaced and/or modified as usual. The effective 24-bit X and Y addresses are formed in the arithmetic unit as usual; they are then held there until the next instruction has been extracted from the store, when they are added to the X and Y addresses of this instruction before replacement in the case of the 116 instruction, or after replacement for the 117 instruction. Note that this addition takes place in the arithmetic unit only - the instruction as stored is not changed.

Example Suppose A51 contains the address of a table B whose i^{th} member (where i is contained in $A63_m$) contains the address of a table C. It is required to copy the j^{th} member (where j is contained in $A61_m$) of C into A49. Then:

```
116Y      0          (A51)     A63
117       0          (0)
04        A49        (A61)
```

will have the desired effect.

The above description applies to both the two and three address forms of these instructions. The instructions are basically two-address - in the three address form Z is ignored and may be used to hold a constant if required.

A string of 116 and 117 instructions followed by a substantive instruction are, in many ways, regarded as a single compound instruction by the microprogram - for example a program may not be interrupted in the middle of a string for external reasons; if it is interrupted because, for example, one of the addresses referred to is locked out, then the whole string will be repeated when the program is next entered. In engineers' mode the maximum length of a string is 13; in programmers' mode a string longer than 13 causes entry to the Monitor Routine which interprets it correctly - thus in programmers' mode there is effectively no limit to the length of a string - but strings longer than 13 will take about 50 millisecs to execute.

3.12 Group 12

Function Number 120 [1100 000]

(3) If $Y > 0$ z_m' = number of 1-bits in bits 0 to Y-1 of x, z_s' = inverse of bit Y of x; rest of z′ clear.

If $Y = 0$ $z_m' = 0$, z_s' = inverse of bit 0 of x; rest of z′ clear.

If $Y < 0$ z_m' = number of 1-bits in bits 47 to 48+Y of x.

z_s' = inverse of bit 47+Y of x; rest of z′ clear.

(2) As 3-address but with x′ in place of z′.

Count 1-bits (sideways add). If $Y \geq 0$ then 1-bits are counted from the m.s. end of x, if $Y < 0$ from the l.s. end of x. The number of 1-bits in the first $|Y|$ bits of x is written into z_m or x_m and the sign bit of Z or X is set to the inverse of the next bit of x.

The formation of the signed Y is done as described for group 5 instructions (see 3.5 page 3). Note that if $Y > 48$ or $Y \leq -48$ the effect will be as if $Y = 48$ (except for the time). This instruction is used in a wide variety of subtle programming tricks, the most common being to form a dense set from a scattered set.

Function Number 121 [1100 001]

(3) z′ = x shifted circularly right Y places (Y signed).

(2) x′ = x shifted circularly right Y places (Y signed).

Shift x circularly right Y places if $Y \geq 0$ or circularly left -Y places if $Y < 0$ writing the result to Z (3-address) or to X (2-address). The instruction cannot set overflow.

The signed shift number Y is formed as described in section 3.5, page 3, except that if Y (after negation if necessary) is greater than 255 the Monitor Routine is entered - it takes the shift number modulo 48 and thus gives the correct result.

Function Number 122 [1100 010]

(3) x′ = y shifted circularly left Z characters.

(2) x′ = y shifted circularly left z_c characters.

In 3-address form y is shifted circularly left Z characters (i.e. 6Z bits) and written into X - only the l.s. 3 bits of Z are in fact used. This instruction is useful when it is desired to shift and copy into a register (not an accumulator) or into a replaced address. It is often faster than the 121 instruction as the shifting is done by characters in the store transfer gates and the time is independent of the number of places shifted.

In the two-address form z is effectively a character modifier. The X and/or Y addresses are modified in the usual way and the number of characters to be shifted is then taken from the m.s. 3 bits of z; y is shifted circularly left this number of character positions and written into X.

Thus if an integer in z is shifted circularly right 3 places it may be used as a character modifier - i.e. when used as the modifier in a Y modified 122 instruction, it will cause the selected character to be placed in the m.s. 6 bits of x - i.e. the 122 can be used as a table lookup instruction on a table with entries of length 6, 12 or 24 bits. The desired character can be made to appear at the l.s. end of x by suitably arranging the table. The character modifier can be stepped by a 115 instruction. Note that z_c is always used to modify the Y address (effectively) irrespective of whether X or Y or both is modified.

<u>Function Number 123 [1100 011]</u>

(3) Append y to $x:_L$	Z_3 characters down
(2) Append y to $x:_L$	z_c characters down

where Z_3 is the integer contained in the l.s. 3 bits of Z.

Append field: let $n=Z_3$ (3 address) or z_c (2 address). Then the 123 instruction leaves the first n characters of $x:_L$ unchanged, replaces the next 8 characters of $x:_L$ by the whole of y and clears the remaining 8-n characters (note that $0 \leq n \leq 7$).

The instruction is used to pack information head to tail, usually over several words; the most common case being the preparation of information for output. The three address form can be used if it is known where within the words the information is to be put; the two address form is used when this is given by a character modifier as for the 122 instruction; for the 123 instruction it is usually convenient to regard the character modifier as modifying X. After the 123 the character modifier can be suitably stepped by a 115 instruction - thus the packing can be done by 123 and 115 instructions without the programmer being conscious of word boundaries.

Function Number 124 [1100 100]

In this instruction x is shifted n places. Y is treated as signed (see 3.5 page 3) and |Y| specifies the maximum value of n unless Y>255 or Y<-255 in which case $n \leq 255$.

(3) If $Y \geq 0$ $x' = x$ shifted up logically the number of places required to shift off the m.s. 1-bit, but at most Y places.

$z' = n$ if a 1-bit has been shifted off, otherwise $z_s' = 1$ and $z_m' = n$

If $Y < 0$ $x' = x$ shifted down logically the number of places required to shift off the l.s. 1-bit, but at most -Y places.

$z' = n$ if a 1-bit has been shifted off, otherwise $z_s' = 1$ and $z_m' = n$.

(2) As for 3-address but $x' = n$ (or $x_s' = 1$ and $x_m' = n$). Find left or right most 1-bit: x is shifted up or down (depending on the sign of Y) until <u>either</u> a 1-bit has been shifted off <u>or</u> it has been shifted |Y| places. The number of places shifted is written into Z in the 3-address form and into X in the 2-address form. If there are no 1-bits in the m.s, or l.s. |Y| places of x then the sign bit of Z or X is set equal to 1.

An exception to the above rule occurs if X=Z in the 3-address form. In this case, since z is written first

$z' = x$ shifted up (or down) n places

and the number of places shifted is lost.

This instruction is commonly used to operate on fields where each bit represents a distinct entity. Note that after the instructions:

```
124    SPUD    48      BUZZ
 53    SPUD    (BUZZ)
```

the m.s. 1-bit of SPUD has been removed.

Function Number 125 [1100 101]

(3) $x{:}_F' = (x{:}_F+v)2^m$

$z_I' = z_I - m$

$OVR' = 0$

where $v = 0$ if OVR is clear

$v = +2$ if OVR is set and $x<0$

$v = -2$ if OVR is set and $x \geq 0$

and m is chosen so that $-1 \leq m \leq Y$ and

$\frac{1}{2} \leq x{:}_F' < 1$ or $-1 \leq x{:}_F' < -\frac{1}{2}$

or $m = Y$, when $-\frac{1}{2} \leq x{:}_F' < -\frac{1}{2}$

(2) Illegal

Standardise double-length floating point number. This instruction is intended for use on double-length floating point numbers (see Section 2.0 p.5(iv)). $x{:}_F$ is regarded as a double-length argument and z_I as the exponent. In general, $x{:}_F$ is shifted up until it lies in the range $\frac{1}{2} \leq x{:}_F < 1$ or $-1 \leq x{:}_F < \frac{1}{2}$ and the number of places shifted is subtracted from z_I. The instruction further assumes that if overflow is set, it is only one bit and is due to a previous fixed-point operation on $x{:}_F$. In this case the correct value of $x{:}_F$ is formed in the arithmetic unit before the standardising takes place. After the operation OVR is left clear even if z overflows. Y is used to specify the maximum number of places to be shifted and if the number cannot be standardised with $m \leq Y$ then $x{:}_F$ is left in the range $-\frac{1}{2} \leq x{:}_F < \frac{1}{2}$ and m=Y.

A number of special cases arise.

(i) If $x{:}_F$ is not a standard double-length fraction, i.e. $x^*<0$, an impermissible operand suspension will occur.

(ii) In the case where $m = -1$ (which can only happen if OVR is set) $x{:}_F$ is shifted down unrounded.

(iii) The operations in the arithmetic unit take place in the following order.

Read x - Read x* - Shift x: - write x - write x* - Read z - Subtract m - write z.

If, therefore, Z = X or X + 1, m will be subtracted from the appropriate operand (x′ or x*′).

<u>Function Number 126 [1100 110]</u>

(3) $z' + \varepsilon y' = x + \varepsilon y + \varepsilon v \quad y' \geq 0 \quad OVR' = 0$

where $v = 0$ if OVR clear; $v = +2$ if OVR set, $y < 0$; $v = -2$ if OVR set, $y \geq 0$

(2) As 3-address but with x′ in place of z′

Justify double-length number. This instruction performs the "carry" between the two halves of a double-length number - it assumes that OVR, if set, was set in operations in the l.s. half and that there was only one bit of overflow. It is thus normally necessary to obey a 126 instruction after every operation on the l.s. halves of double length numbers. The value of the carry =

0	if OVR	clear	$y\geq0$
-1	if OVR	clear	$y<0$
+1	if OVR	set	$y<0$
-2	if OVR	set	$y\geq0$

This carry is added to x (and written into X or Z) and OVR and y_s are cleared, OVR will be left clear unless the addition of the carry to x causes overflow.

If X=Y (2 address) or Z=Y (3 address) the arithmetic is correctly performed and the single result obtained will be the l.s. half (with sign bit and OVR cleared).

<u>Function Number 127 [1100 111]</u>

This instruction is illegal.

3.13 Group 13

Function Number 130 [1101 000]

Function Number 131 [1101 001]

Function Number 132 [1101 010]

Function Number 133 [1101 011]

Function Number 134 [1101 100]

Function Number 135 [1101 101]

Function Number 136 [1101 110]

Function Number 137 [1101 111]

These instructions are illegal.

3.14 Group 14

3.14.1 Instructions 140-143 - transfers, regions

The instructions 140 to 143 are concerned with peripheral or other transfers normally affecting a block of registers. Except for the 143 instruction - which copies identical information to all words in the block - these instructions occur in pairs, the second of which must always be a 142; the first member of the pair is a 140, 141 or 142 for transfers involving peripherals, the drum and other parts of the working store respectively. No instructions, other than 116 or 117, are allowed between the two members of the pair - an isolated 140, 141 or 142 instruction is illegal. The pair of instructions, together with any 116 or 117 instructions before either or both members of the pair, constitutes one compound instruction or string as described in Section 3.11, page 4 and is subject to the restrictions described there. Before the transfer is initiated the entire region specified is checked for lockouts - the instruction pair is locked out if any register in the region is locked out.

Function Number 140 [1110 000]

Select device Y in mode M. (See Section 4.1 for modes.)

The instruction is normally written 140.M 0 Y where M is an integer between 0 and 31 and Y is the name of the peripheral. The mode, M, is stored in the m.s. 5 bits of the X-address; the rest of X is not used (though if X is replaced or modified, this is done normally, the mode being taken from the corresponding place in the effective X-address). The Y address should be written as an asterisk followed by two letters and an integer between 0 and 31 e.g. *SP17. The letters allowed are as follows:

SR	=	seven-track tape reader
FR	=	five-track tape reader
CR	=	card reader (80 cols)
VR	=	card reader (65 cols)
MT	=	magnetic tape
SP	=	seven-track tape punch
FP	=	five-track tape punch
CP	=	card punch
LP	=	line printer

Other more specialised devices will have appropriate letter pairs allocated to them. The two letters are stored in 5 bits each (from A=1 to Z=26) followed by the 5 bits of the integer to give the 15 bit Y-address e.g. *SP17 is compiled by the input routine as

$$19 \times 2^{10} + 16 \times 2^{5} + 17 = 19985$$

The machine address of a peripheral is an 8-bit number usually known as the k-bits. The programmers’ address is correlated with the machine address as follows: while the program is running, absolute location 1 contains the address of the program's directory entry. The eighth word of this directory is the start of a table of the peripherals reserved for the program - each entry of this table has the programmers’ address of a peripheral device in its more significant half and the k-bits in the less significant half - the table being terminated by a clear word. The microprogram of the 140-instruction searches this table in the same manner as the 146-instruction (q.v.); if the programmers’ name is found the corresponding k-bits are extracted. If the clear word is reached without the programmers’ name being found then action occurs due to peripheral violation. In Engineers and Monitoring Modes this search does not take place and the Y-address of the 140-instruction must be the actual k-bits.

The meaning of the mode depends on the peripheral concerned - unassigned modes will cause suspension due to illegal instruction. All legal modes are of odd parity and, in general, modes with M<16 are for input transfers and M>16 for output transfers. For details of modes for individual peripheral devices please see the appropriate parts of section 4. Mode 16 is rather special - in all cases it disengages the device. The 140.16 instruction must be followed by a 142-instruction as usual but the addresses in this 142-instruction are neither read nor checked. Mode 13 interrogates the specified device; it writes one word of information into the address specified by the X-address of the following 142-instruction - the Y-address is not used and may take any value. The contents of this word are described in section 5.5. The 140.13, 142 instruction pair may be obeyed even if the device is busy or disengaged - thus enabling a program to test whether a device is engaged.

The above description applies equally to the two and three address forms of the instruction - in the 3-address form Z is not used.

Function Number 141 [1110 001]

Select drum in mode M, drum address Y.

The instruction is normally written 141.M 0 Y where M is an integer between 0 and 31 and Y is a drum address, relative to the program's drum datum points.

For this instruction a temporary notation is introduced - the fifteen bits of the X-address as written will be called X_9 to X_{23} (X_{23} being the l.s. end) - X_0 to X_8 being the nine bits added during modification or replacement; a similar notation is used for Y. In the simple case where the instruction is not replaced or modified, the mode is taken from X_9 to X_{13}. Bits X_{14} to X_{22} are added into positions Y_0 to Y_8 thus giving a 24-bit drum address. The 24-bit quantity may be written in the Y-address and will be correctly spread over the X and Y addresses by Basic or Symbolic Input - this is in fact always done if a mode has been written in the instruction. Bit X_{23} is ignored. If the X-address is replaced this combination of the X and Y addresses does not take place - in any case all pre-modification, replacement and internal modification are done after this operation thus giving a 24-bit Y-address, i.e. modification and replacement of X cannot affect the drum address. The mode is taken from X_9 to X_{13} after all modification and replacement of X has been done in the usual way.

There are only two normal modes available - mode 1 is "read from the drum", mode 21 is "write to the drum". There is also an "interrogate" mode, M=13, but this does not give any information of use to the ordinary programmer; it is used within the Monitor Program.

When the machine is in Engineers' Mode the drum address, built up as above, is the actual address used on the drum. In Programmers' Mode the drum datum point has to be added and a check made that the region specified is in the program's reserved region. The microprogram of the 141-instruction looks at the word before the program's index word (whose address is given in absolute location 1); this word contains 2^{24}-N in its upper half, and D in its lower half where D is the drum datum point and N is the number of words in the reserved region.

If the machine has more than one drum control the situation is rather more complicated. The program may now have several disconnected regions, one on each control, though they appear consecutively numbered to the programmer. Suppose the program has regions starting at D_1, D_2, D_3 of length N_1, N_2, N_3 respectively. Then the microprogram searches a table backwards starting at the word before the index word: successive entries in this table are:

$2^{24}-N_1$, D_1; $2^{24}-N_1-N_2$, D_2-N_1;

$2^{24}-N_1-N_2-N_3$, $D_2-N_1-N_2$; etc.

The search terminates when an entry is found such that adding the specified drum address to the upper half does not cause a change of sign or alternatively by the entry: $2^{24}-1$, 0 which terminates the table. If the transfer specified overlaps two or more regions the Monitor Program is entered - it splits the transfer into the appropriate transfers on the separate regions and obeys them successively.

The above description applies equally to the two and three address forms of the instruction - in the 3-address form Z is not used.

Function Number 142 [1110 010]

(a) After 140 or 141 instruction. Transfer Y words or characters to or from region starting at X.

(b) A pair of 142 instructions thus; 142 X_1 Y_1

142 X_2 Y_2

Copy Y_2 words from region starting at X_2 to region starting at X_1 (Y_1 is ignored).

(a) After a 140-instruction the 142-instruction causes a transfer to be executed on the device and in the mode specified by the 140-instruction. The working store region involved in the transfer starts at X and the transfer is of length Y words or characters, according to the device; sometimes Y specifies a maximum - e.g. read up to NL or at most Y characters. Y may not exceed 32767.

E.g. the instruction pair

```
140.1     0         *SR1

142       BUZZ      40
```

Says "read up to NL or at most 40 characters from *SR1 and store them starting at working store address BUZZ". Note that the whole region specified (in this case from BUZZ to BUZZ+4) is checked for lockouts and reservations before the transfer is initiated so that the whole region must be within reservations even if the very first character is NL.

Similarly after a 141-instruction a transfer is initiated to or from the drum (according to whether the mode in the 141 is 21 or 1) from or to the working store area starting at X and of length Y words.

It is illegal for a 142 after a 140 or 141 to have Y=O (except for those modes which do not do a transfer).

(b) A pair of 142-instructions does an internal transfer of length Y_2 words from the region starting at X_2 to that starting at X_1. The Y-address in the first 142, Y_1 , is ignored. Y_2 may be zero, in which case the pair has no effect. The regions

X_1 to X_1+Y_2-1 and X_2 to X_2+Y_2-1

must both be within reservations and are checked for lockouts. The transfer proceeds starting at the last word and working backwards unless the last word of the destination region lies within the source region - in which case it works forwards, thus always ensuring the correct result. The instruction pair is harmless if $X_1=X_2$.

Note that whereas peripheral and drum transfers are merely initiated by the 142-instruction and subsequently proceed autonomously while the computer carries on with the program, internal transfers are executed immediately - as they use the core store full time.

The two and three address forms of the 142-instruction are identical - i.e. in the three address form Z is ignored.

In the case of peripheral and drum transfers the term 16Y represents the hesitations involved in the transfer - these are actually charged to the program when the instruction is obeyed but take place later - the timer of the program running hesitates when the computer hesitates to allow a word to be read from or written to the core store. If the running program is not using the core store at the time it gets these 16 microsecs free. In the cases where Y specifies a maximum the program is charged for the full number of hesitations even though they may not all take place. Every peripheral or drum transfer causes at least one entry to the time-sharer. The time spent in the time-sharer is not included in the above times, neither is it charged.

In the case of the 142-pair Y should be read as Y_2 for both instructions - i.e. the time for the 142-pair is the fixed part of the times for the instructions plus 32 microsecs per word transferred.

Function Number 143 [1110 011]

(3) Write z to Y words starting at X.

(2) Clear (zeroize) Y words starting at X.

This instruction writes the same quantity (z or 0) into every word of the region X to X+Y-1. The whole region is checked for reservations and lockouts before the instruction is obeyed. If Y=0 the instruction is a dummy. The 3-address form with Z=A0 is treated as illegal by the hardware but is correctly interpreted by OMP. Its use is not recommended; it does nothing which cannot be done by the 2-address form.

3.14.2 Instructions 144-146: Table Search

The instructions 144, 145, 146 all search a table to find the first number with a particular property. The table starts at X and the first number w (in location W) is found such that w satisfies a particular inequality or equality (according to function) relative to y. W is then written into Z. At the start of the instruction the microprogram does not know how far it will have to search so it checks lockouts and reservations as it goes along. It is recommended that all tables to be searched should be terminated by a word guaranteed to end the search (otherwise if a wrong y comes along the search may continue till reservations are violated; such a word should be 1-ε for the 144-instruction, -1.0 for the 145-instruction and any word with a zero lower half for the 146-instruction.

These instructions are all essentially 3-address instructions - the two-address forms are illegal.

Function Number 144 [1110 100]

(3) W=address of first w such that W≥X, w≥y. Then z'=W.

(2) Illegal.

Search a table starting at X to find the first number w such that w is greater than or equal to y. Then the address of w is written into Z.

Function Number 145 [1110 101]

(3) W=address of first w such that $W \geq X$, $w \leq y$. Then z'=W

(2) Illegal.

Search a table starting at X to find the first number w such that w is less than or equal to y. Then the address of w is written into Z.

Function Number 146 {1110 110]

(3) W=address of first w such that $W \geq X$, $w_u = y_u$ or $w_m = 0$. Then z'=W

(2) Illegal.

Search a table starting at X to find the first number w such that the upper half of w equals the upper half of y or the lower half of w=0. Then the address of w is written into Z.

Function Number 147 [1110 111]

This instruction is illegal.

The information in this section refers in general both to Orion 1 and Orion 2. Described here are the differences to be noted when reading this for Orion 2.

Differences arise because the object program's peripheral transfer instructions are extracoded. (See 4.1 for more details)

The K-bits on Orion 2 are sequential and are not checked for odd parity.

The 143 instruction - the 3-address form with Z = A0, is not treated as illegal by the hardware.

3.15 Group 15 Time-sharing instructions

The instructions of group 15 are illegal in programmers' mode (except the three-address 150 instruction, which is specially treated by the monitor program); they are used in monitoring mode for special purposes, mainly connected with time-sharing. They are described below for the sake of completeness but any attempt to use them (except in the monitor program) will cause suspension due to illegal instruction. 15 group instructions in engineering mode have the same effect as in monitoring mode except that the 152-instruction does not enter programmers' mode.

The pre-150 instruction

When a program running in programmers' mode is interrupted for any reason a special piece of built-in micro-program called the pre-150 instruction is obeyed. An instruction cannot be read from store until the information in the arithmetic unit, about the program being interrupted, has been stored; so this special built in instruction is obeyed.

The first action of the pre-150 instruction is to set the monitoring stat - this inhibits further interruptions, removes restrictions on reservations etc. and validates group 15 instructions. The machine is now in monitoring mode. Absolute location 1 is now read - it gives the address of word 0 of the program's directory. The control address of the program is now written into the modifier position of word 2 of the directory and overflow in the sign bit - i.e. the link is stored in word 2 in the same manner as the 86 instruction stores it and overflow is cleared. Next the program timer is added into word 1 of the directory; if word 1 is positive after this addition this means that timer overflow has occurred i.e. the program has exceeded the time requested - and program failure is set as a reason for entry. The pre-150 instruction now writes the requirement of the program just left into the upper half of word 0 of the directory - the lower half being left unaltered. The requirement is as follows:

(a) Peripheral lockout - D0, D1 are clear, the k-bits of the device concerned are stored in D16 to D23 and the mode in D11 to D15.

(b) Drum lockout - D0, D1 are 10 with the m.s. 4 bits of the 24-bit absolute drum address (specifying the drum control number) in D20 to D23. These are clear for machines with only one drum control.

(c) Core store lockout - D0, D1 are 01 with the address locked out in D9 to D23. In all other cases the upper half of word 0 is cleared.

The pre-150 instruction ends by jumping to absolute address 2.

Timing The pre-150 instruction takes approximately 160 microsecs.

Function number 150 [1111 000]

(a) Programmers' mode

(3) Enter Monitor program to perform action Z, operands X and Y.

(2) Illegal.

This instruction, in programmers' mode is treated specially by the Monitor program. For further details see section 5.3.

(b) Monitoring and Engineers' Mode

Jump to Y if there is no reason for entry, otherwise x' = reason for entry. This instruction is normally obeyed in absolute address 2, immediately after the pre-150 instruction. The cases where there is no reason for entry are core store lockout, end of transfer and one second since last interrupt. The reasons for entry (and their values) are:

Peripheral lockout	128
Core store parity failure	32
Peripheral or drum incident	64
Reservation line parity failure	16
Spring cleaning (one minute)	8
Program incident	4

If there is no reason for entry the instruction simply jumps to Y and does not write to X. If there is a reason for entry the instruction does not jump and the integer representing the appropriate combination of reasons (most commonly only one reason) is written into X.

If the 150 instruction is obeyed in engineers' mode - or immediately after switching to programmers' mode - a reason for entry may come up due to some of the "reason for entry" stats having been set some time previously - these stats are only cleared by the 150 instruction. The two and three address forms are identical - Z is not used in the 3-address form.

Timing (microsecs)

	Jump	No Jump
3-address :	20	36
2-address :	40	56
2-address unmodified :	20	36

plus 16 microsecs for each replacement.

Function number 151 [1111 001]

(a) Programmers' Mode   This instruction is illegal.

(b) Monitoring and Engineers' Mode

Search a list starting at Y till a word is found whose upper half is a satisfied requirement. x' = address of this word.

The index words of the programs in the machine are in the form of a list - i.e. the lower half of each contains the address of the next, the upper half containing the requirement as described above under the pre-150 instruction. If the requirement is clear it is always satisfied; if it is a peripheral, the drum or a core store location the appropriate check takes place to see if it is now available; if D0 and D1 of the requirement are 11 the requirement is considered as not satisfied - this combination is never written by the pre-150 instruction but is inserted by the monitor program if it is necessary to stop a program for any reason.

The 151 instruction examines these words in turn, starting at y, and it looks at successive entries (effectively by repeated replacements) until an entry is found whose requirement is satisfied - the address of this word is written to X. It is normal for X to be absolute address 1 since 1 must contain the address of the index word when the program is entered.

The 151 instruction is the same in the two and three address forms - in the three address form Z is not used.

Timing (microsecs)

```
3-address            : 48 + 16(n+m)
2-address            : 68 + 16(n+m)
2-address unmodified : 48 + 16(n+m)
```

plus 16 microsecs for each replacement where n is the number of words searched and m is the number of those whose unsatisfied requirement is a core store lockout.

Function number 152 [1111 010]

(a) Programmers' Mode   This instruction is illegal,

(b) Monitoring and Engineers' Modes

Enter programmers' mode setting the reservations from x, jump to y_m setting OVR if y_s =1

This instruction is used to exit from the time-sharer. The Y address specifies the point at which ordinary program is to be entered - it is treated as a link in the same form as used by the 87 instruction. The reservations and monitoring conditions are taken from x as follows:

D_9 to D_{23} of x is the lower reservation and datum point (D_{18} to D_{23} should always be zero) - D_{33} to D_{47} of x is the upper reservation (i.e. the last location within reservations);

D0, D1, D24, D25 are set to 1 if monitoring on signals, overflow, jumps and floating point overflow respectively is required. D2 is a 1 for strong reservations (normally the case) and D26 is a 1 if floating point operations are not to be rounded.

Before the 152 instruction is obeyed absolute location 1 must be set (usually by a 151 instruction) as the address of word 0 of the program's directory. Since the link is held in word 2 of the directory (since that is where the pre-150 instruction stores it) and the reservations are conventionally held in word 5 of the directory the usual form of the 152 instruction is:

```
                 152XY     5         2         1
```

In engineers' mode the reservation lines are ignored and the 152 instruction obeys the link in Y and sets the monitoring condition stats.

The 152 instruction is the same in the two and three address forms - in the three address form Z is not used.

Timing (microsecs)

```
3-address            :  72
2-address            :  92
2-address unmodified :  72
plus 16 microsecs for each replacement.
```

Function number 153 [1111 011]

This instruction is illegal in all modes.

Function number 154 [1111 100]

(a) Programmers' Mode   This instruction is illegal.

(b) Monitoring and Engineers' Modes

Jump to Y if device not failed or write failure information to X and abandon the transfer. This instruction should be preceded by a 140.13 or 141.13 instructions to select a peripheral device or drum control.

When a program is interrupted due to a peripheral or drum incident the 150 instruction only gets 1 bit of information saying that this has occurred. It is necessary to interrogate all the devices until one of them says it has failed - it says this by the 154 instruction failing to jump. Thus the 154 instruction (preceded by a Y replaced 140.13 instruction) is used at the end of a loop scanning the peripherals - the loop is left when the failed device has been found. A word of information about the device is written into X and the transfer abandoned. This information consists of the current and finishing addresses of the transfer, which are written into D9-D23 and D33-D47 respectively, and a number of bits, giving either the type of failure or the state of various staticisors in the device, are written into D0-D7 and D24-D31. For further details of this information see the relevant parts of section 5.5.

In the case of the drum two words are obtained. The word written into X is similar to that for peripherals and in addition the failing drum address is written into X+1. For details of the information bits see section 5.5.5. The drum address in the preceding 141.13 instruction may be any address on the required drum control.

The 154 instruction is the same in the two and three address forms - in the three address form Z is not used.

<u>Timing</u> (microsecs)

		Jump	No Jump
3-address	:	20	116
2-address	:	40	136
2-address unmodified	:	20	116

plus 16 microsecs for each replacement.

<u>Function number 155 [1111 101]</u>

(a) <u>Programmers' Mode</u> This instruction is illegal.

(b) <u>Monitoring and Engineers' Mode</u> Abandon transfer on selected device.

This instruction should be preceded by a 140.13 or 141.13 instruction to select a peripheral device or drum control - the instruction then abandons the transfer, if any, on the selected device. In the case of character devices the 155 instruction abandons whatever transfer happens to be taking place on the control containing the selected device. The addresses are not used in the 155 instruction - they may have any value and there is no distinction between the two and three address forms.

<u>Timing</u> The 155 instruction takes about 20 microsecs.

<u>Function number 156 [1111 110]</u>

This instruction is illegal in all modes.

<u>Function number 157 [1111 111]</u>

(a) <u>Programmers' Mode</u> This instruction is illegal.

(b) <u>Monitoring and Engineers' Mode</u> Stop, displaying addresses.

The machine stops after obeying the instruction and will continue with the next instruction if the STOP and RUN buttons are pressed. The function lights say 157 and the TX and/or TY lights are lit in the 2-address case. The X and Y addresses, after all modification and replacement, can be seen in the upper and lower halves of H respectively. In the 3-address form Z is displayed in the upper half of J - in the 2-address form the upper half of J is the same as the upper half of H. The lower half of J is the address of the next instruction and the single step button causes the next instruction to be obeyed without being displayed.

The information in this section refers in general both to Orion 1 and Orion 2. Described here are the differences to be noted when reading the section for Orion 2.

Note timings are not correct for Orion 2.

<u>The pre-150 instruction</u>

Subsection (a) On Orion 2 D11 to D15 are clear.

Subsection (b) On Orion 2 these are not necessarily clear.

<u>Function Number 150</u>

Subsection (b) Core store parity failure and reservation line parity failure are not reasons for entry. If they do occur then the machine comes to a stop.

<u>Function Number 152.</u>

D18 to D23 need not be zero in X.

<u>Function Number 156.</u>

This instruction is not illegal in Monitoring and Engineers' Mode. Its effect is to jump to X if peripheral Y is involved in a transfer which is not completed.

<u>Function Number 157</u>

Subsection (b) This instruction stops the machine and displays X, X+1 (where appropriate) Y and Z addresses on the control panel in the appropriate registers. By carrying out appropriate action, the machine continues by obeying the next instruction etc.