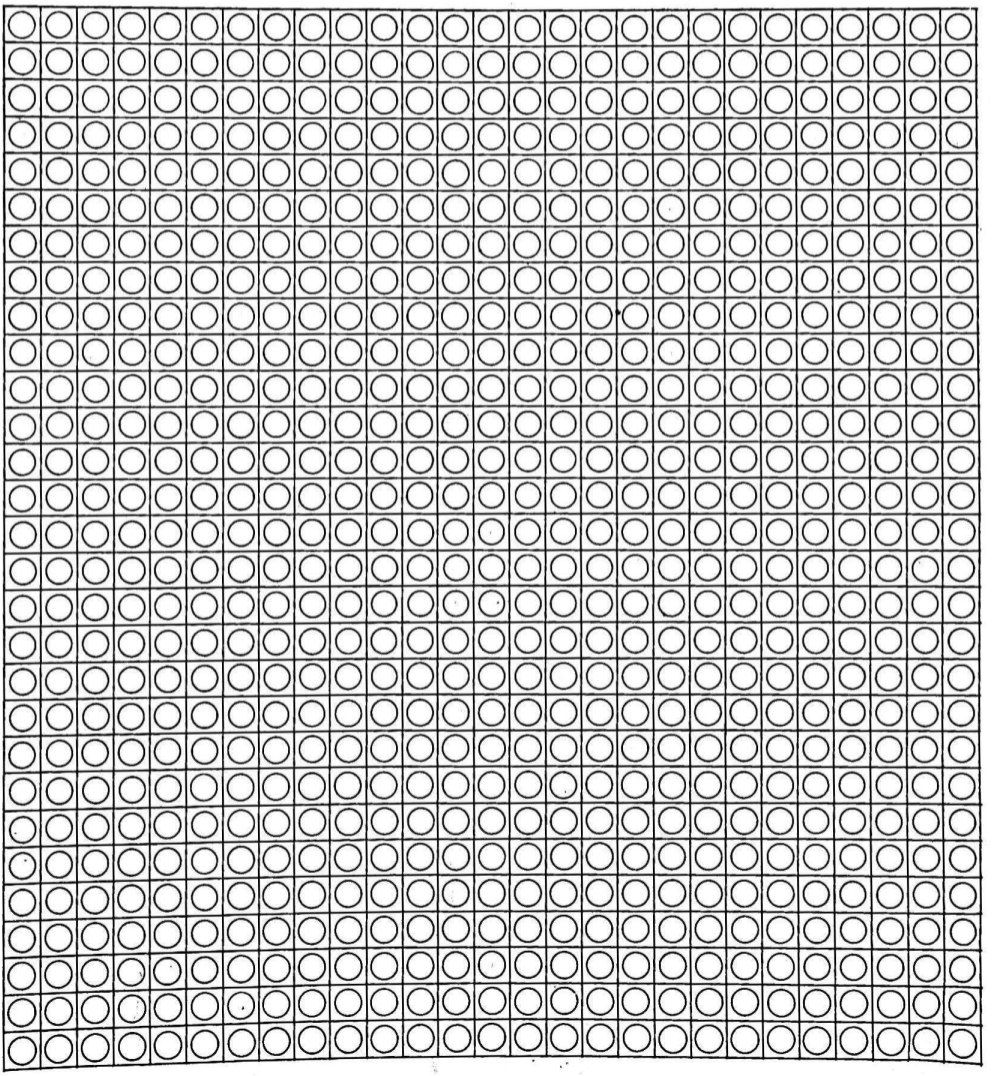
ICL Technical Literature

2900

2900

Order Code



Scanned from a physical document (donated by a kind former ICL employee)

Order Code

• THIS DOCUMENT IS COMPANY RESTRICTED •

Issue	1					
Des. Auth.						
CED						
Tech. Lit.						

The policy of International Computers Limited is one of continuous development and improvement of its products and services, and the right is therefore reserved to alter the information contained in this document without notice. ICL makes every endeavour to ensure the accuracy of the contents of this document but does not accept liability for any error or omission. Any equipment or software performance figures and times stated herein are those which ICL expects to be achieved in normal circumstances. Wherever practicable, ICL is willing to verify upon request the accuracy of any specific matter contained in this document.

Issued by Technical Literature Department,
International Computers Limited,
Wenlock Way, West Gorton,
Manchester. M12 5DR.

Printed by Reprographic Services,
West Gorton,
Manchester.

CONTENTS

APPENDICES

Related Documents

6594801 - 2900 Architecture.

1. INSTRUCTION FORMATS.

1.1 Hexadecimal Notation.

2900 is based on the use of the 8-bit byte, each byte consisting of two 4-bit groups which are hexadecimal in character, i.e. they are capable of holding the range of values 0-15. Hexadecimal values are denoted by a preceding # character, thus #9A denotes the binary equivalent 10011010, and, as a further example, #FC is equivalent to the binary value, 11111100.

The following table shows the hexadecimal scale together with binary and decimal equivalents:

Hexadecimal	Binary	Decimal
#0	0000	0
#1	0001	1
#2	0010	2
#3	0011	3
#4	0100	4
#5	0101	5
#6	0110	6
#7	0111	7
#8	1000	8
#9	1001	9
#A	1010	10
#B	1011	11
#C	1100	12
#D	1101	13
#E	1110	14
#F	1111	15

All diagrams are shown with the store Least Significant Word at the top of the diagram (this includes the diagrams showing the stack).

1.2 Architectural Mod. Levels

The concept of Architectural Mod'Levels (AML) is introduced to allow for progressive enhancement to the primitive level interface without instantaneously impacting the total population of 2900 systems.

The basic AML is AML0 and it is a rule that AMLs are forwards compatible such that software written for system at AML0 will run on systems at AML1.

The AML of systems can be read from line 16 of Block 0 of the Image Store.

1.2.1 Summary of Additional Facilities at AML1.

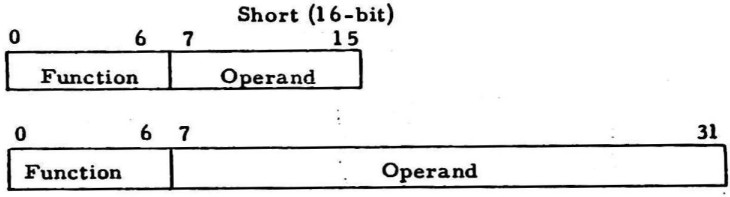
Facility

1. Additional format for RRTC
2. Alternative form of VALIDATE instruction
3. System Call count
4. Parameter space check for System Calls
5. Addition of bit string operand form
6. Addition of TEST, CLEAR and SET instructions
7. (B + N) Operand form
8. Vector Descriptor, type 0, size code 4 (≡ 16 bits)
9. Vector descriptor, for signed items.

1.3 Instruction Lengths

2900 makes a use of two distinct instruction lengths, i.e. short (16-bit) or long (32-bit). In either case the function occupies bits 0-6, the operand occupying either bits 7-15 or 7-31 according to instruction length. (Fig 1).

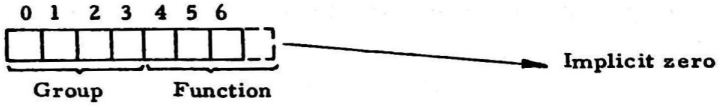
Fig 1. INSTRUCTION LENGTHS



1.4 Function Decode

From fig 2 it can be seen that the function code bits 0-6 are regarded as two hexadecimal characters, the L.S. bit of the second character being implied zero. The first character denotes the order code group, the second denotes the function number within the group. The order code thus contains 16 groups, each group containing up to 8 functions (since only even-numbered function values can be specified). The function codes # 00 and # FE are illegal. There are therefore 126 permissible function codes ranging from # 02 to # FC, some of these being as yet unassigned.

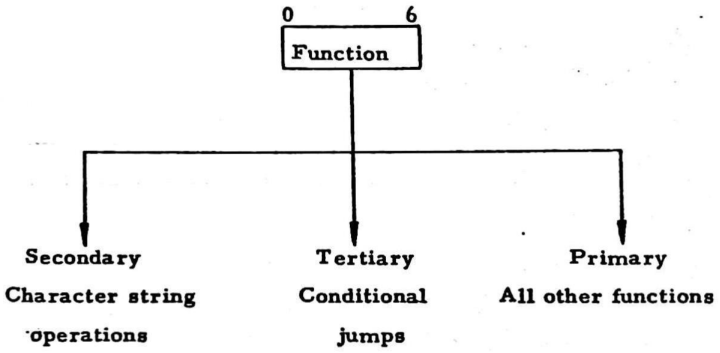
Fig 2. FUNCTION DECODE



1.5 Instruction Types

There are three distinct instruction types, primary, secondary and tertiary (fig 3), each requiring a different operand field format.

Fig 3 INSTRUCTION TYPES



Primary instructions use the operand field solely to specify either an address or a literal value. Secondary instructions are concerned with character string manipulation and thus require to use the operand field for specification of values such as a string length, byte mask, byte filler character etc.

The tertiary instructions are all conditional jumps and thus have to specify a jump address and also the jump condition. Identification of the instruction type determines the method of interpreting and decoding the operand field. The instruction type is recognised from the function bit decode as indicated in fig 2.

1.6 Operand Format of Primary Functions

Functions are provided which perform numerous operations on the special addressing registers, i.e. LNB, CTB, XNB, DR, SF and the central computational registers i.e. the accumulator ACC and the indexing register, B. A simplified summary of the available functions is shown in fig 4 below.

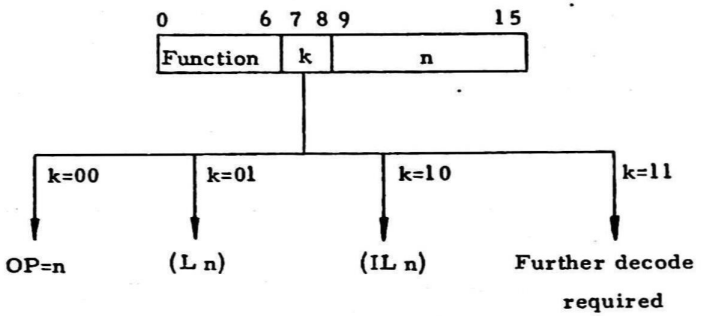
1. Load LNB, store LNB, raise LNB, load XNB, store XNB, adjust SF, load CTB, store CTB.
2. Simple jump functions (unconditional jumps), jump and link, decrement B and jump.
3. Functions used to jump into and return from procedures, i.e. call, exit, escape-exit.
4. Semaphore instructions; increment and test, test and decrement.
5. Simple non-computational operations (load, store, stack and load), computational functions (add, multiply, etc) and boolean operations on the ACC (not equivalence, and, or).
6. Operations on the index register B and the descriptor register DR.
7. Miscellaneous, e.g. idle, read real time clock.

Fig 4. PRIMARY FUNCTIONS

1.7 K Decode

Fig 5 shows the operand field for primary instructions. Taking a 16-bit instruction, the 9-bit operand field is decoded into a 2-bit k-field and a 7-bit n-field.

Fig 5. FORMAT OF OPERAND FIELD (PRIMARY INSTRUCTIONS) - k DECODE



k	OPERAND
00	Literal n
01	Literal at L n
10	Literal at IL n
11	Further Decode Required

n must be ≤ 127

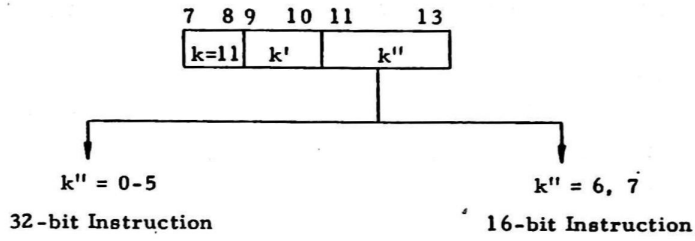
If k = 00 (i.e. 0) then n is taken as a literal operand and is interpreted as a signed integer in the range: $-64 \leq n \leq +63$.

If k = 01 (i.e. 1), the operand addressing form is L n, i.e. the contents of the item at the location LNB + n. In this case, n is interpreted as an unsigned integer in the range : $0 \leq n \leq 127$.

For k = 10 (i.e. 2), the operand addresssing form is IL n, i.e. indirect access to the final operand via a descriptor which is stored at the location LNB + n. Again, n is an unsigned integer in the range: $0 \leq n \leq 127$.

1.8 k = 3 decode.

If k = 11 (i.e. 3), the hardware has to perform another decode of two further fields k' and k''. See fig 6.



k' defines whether access is direct or indirect, If indirect, gives information about descriptor.

k'	
00	Direct access
01	Indirect access (desc. in DR modified)
10	Indirect access (desc. in store)
11	Indirect access (desc. in store modified)

Fig 6. FORMAT OF OPERAND FIELD (PRIMARY INSTRUCTIONS) - k' DECODE

The k'' - field occupies bits 11-13 and indicates the register by means of which the operand is accessed. It also determines whether the instruction length is 16 or 32 bits. Note, that for a 16-bit instruction the quantity n is written as a lower-case character; for 32-bit instructions, it is specified as an upper-case character N. This, being an 18-bit field, has a greater range than n, which is, of course 7 bits long.

This manual has been written in terms of SFL formats.

The table given below lists STAPLE equivalents.

Table 1

SFL		STAPLE	SFL		STAPLE
.N		(signed literal)	.C	N	(CTB+N)
.D	N	(DR+N)	.DC	N	(DR+(CTB+N))
.G		(IS location N)	.IC	N	((CTB+N))
.GB		(IS location B)	.MIC	N	((CTB+N)+B)
.L	N	(LNB+N)	.T		(TOS
.DL	N	(DR+(LNB+N))	.DT		(DR+TOS)
.IL	N	((LNB+N))	.IT		(TOS)
.MIL	N	((LNB+N)+B)	.MIT		(TOS+B)
.X	N	(XNB+N)	.B		B
.DX	N	(DR+(XNB+N))	.B	N	(B+N)
.IX	N	((XNB+N))	.D		(DR)
.MIX	N	((XNB+N)+B)	.MD		(DR+B)
.P	N	(PC+N)			
.DP	N	(DR+(PC+N))			
.IP	N	((PC+N))			
.MIP	N	((PC+N)+B)			

An SFL instruction takes the form:

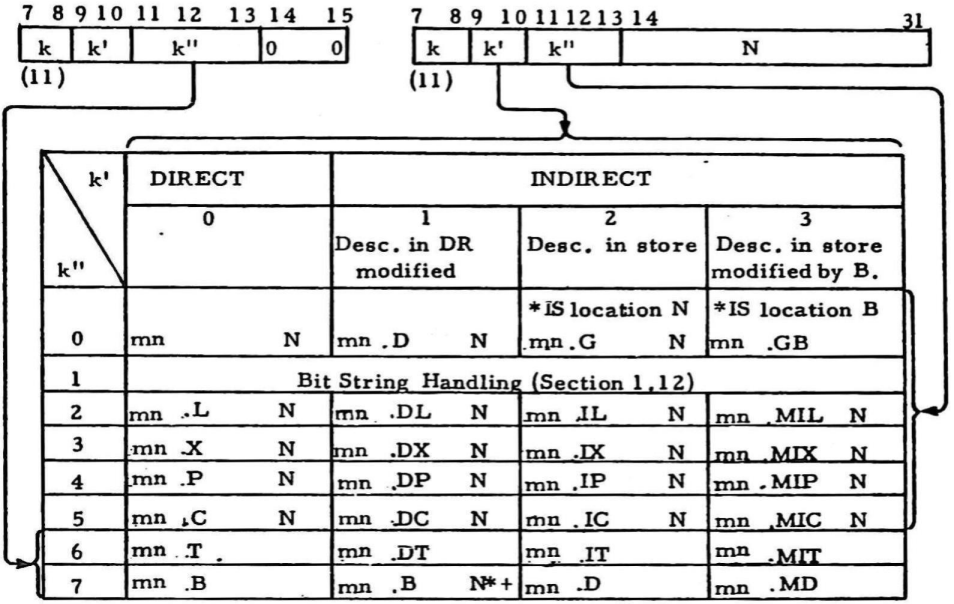
MNEM .VAR OP1, OP2, - - - - OPn

where:

MNEM is the mnemonic of the instruction (1 to 5 letters)

.VAR is the optional SFL variant

OP1, OP2, - - - - OPn are the operands of the instruction.



k' / k"	DIRECT	INDIRECT		
	0	1 Desc. in DR modified	2 Desc. in store	3 Desc. in store modified by B.
0	mn N	mn .D N	*IS location N mn .G N	*IS location B mn .GB
1	Bit String Handling (Section 1.12)			
2	mn .L N	mn .DL N	mn .IL N	mn .MIL N
3	mn .X N	mn .DX N	mn .IX N	mn .MIX N
4	mn .P N	mn .DP N	mn .IP N	mn .MIP N
5	mn .C N	mn .DC N	mn .IC N	mn .MIC N
6	mn .T	mn .DT	mn .IT	mn .MIT
7	mn .B	mn .B N*+	mn .D	mn .MD

i. PC contains address of current instruction.

ii. MIX N, MIT, MD indicate items pointed to by descriptors held in XNB+N, top 2 words of stack, and in DR respectively The 'M' indicates that the address in each descriptor is to be modified by B.

iii. Operand format for k" = 1 is as follows

AML0 - Unassigned

AML1 - given in section 1.12

vi. The equivalent STAPLE operand formats are in Table 1.

*Classed as direct address form.

+32 bit format. The format was unassigned at AML0 and is still provisional.

Fig. 7. FORMAT OF OPERAND FIELD (PRIMARY INSTRUCTIONS) - k" DECODE

1.9 Direct Access Decode - k' = 0

Figure 7 shows the instruction formats in greater detail. Taking a value of k' = 0, i.e. direct access, let us examine the possible values of k".

k" = 0 indicates that the operand is, in fact, the literal N, i.e. bits 14-31 of the instruction. N is a signed quantity in the range:

$-27^{17} < n < 2^{17} - 1$

k" = 1 Bit string handling (See section 1.12)

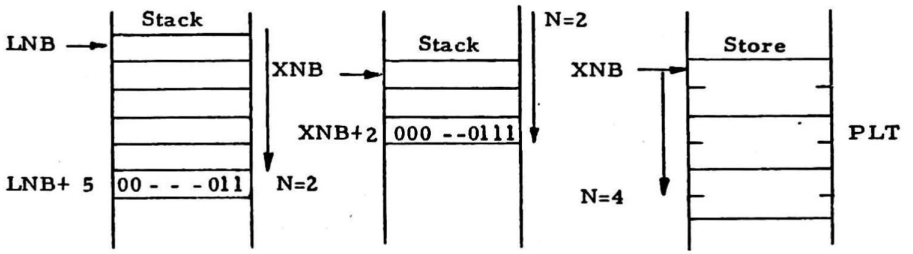
k" = 2-4 Decode

k" = 2 uses L N, i.e. the contents of the location LNB+N, as the operand. See fig 8 (a). LNB points, as always to a location within the stack. In our case N = 5, so the fifth item above LNB is accessed. The contents of that location are 3, so the operand used is 3. Note that here N is an unsigned (positive) quantity and access below LNB, (i.e. LNB-N) is not possible.

k" = 3 uses (X N) i.e. the contents of the location XNB+N, as the operand. In the example shown in fig 8 (b), N = 2 thus operand = 7. XNB is a hardware register which can point anywhere within the store. In fig 8 (b). XNB points to a location within the stack.

If XNB points at, say a table of descriptors in store, then operand access is as shown in fig 8 (c). The quantity N is a word count, so the operand form (XNB + 4) will, in fact, access the third descriptor in the table, i.e. the one at XNB + 4 and XNB + 5. A typical use for this operand form would be to load a descriptor into DR.

k" = 4 accesses P N, i.e. the contents of the item at PC+N are used. PC is the Program Counter, which contains the address of the current instruction, Here, N is a signed quantity.



(a) L N
Operand = 3

(b) X N
Operand = 7

(c) X N
Operand = Desc. at XNB+4, XNB+5.

Fig. 8 PRIMARY FUNCTIONS - DIRECT OPERAND ACCESS

(a) C N

(b) T

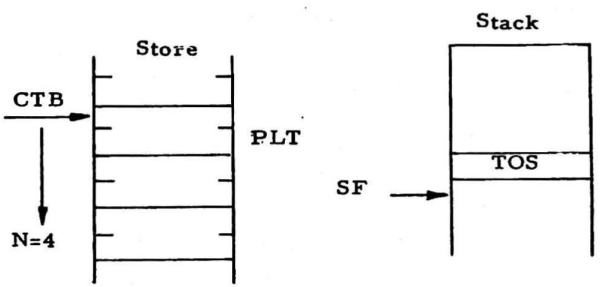


Fig. 9 DIRECT OPERAND ACCESS

k" = 5-7 Decode

k" = 5 uses C N (the contents of location CTB+N) as the operand CTB is a hardware register that may point anywhere in store. Thus, CTB is used in a similar way to XNB. CTB will usually be used to point to a PLT (Procedure Linkage Table) in Store. Fig. 9 (a) shows the operand (C+4) accessing the third descriptor in the PLT.

k" = 6 uses T (TOS). This points to the item to be read from the stack in the case of a read instruction (Load ACC for example). If a 4-word item is expected at TOS and it is really only two words long then, on a TOS access, a 4-word item will be picked up. Conversely, if a single word item is expected at TOS and it is 2 words long, a truncated item will be accessed giving incorrect results. (When writing programs it is important to keep track of the size of stacked items).

When using this operand format with an instruction which writes the operand away, the item is written to TOS + 1, in the case of a single word item, or to TOS + 1, TOS + 2 in the case of a double word item, etc., SF points to this next available location on the stack and will be incremented or decremented by the size of the item added or taken from the top of stack. See fig. 9 (b).

k" = 7 Here the actual contents of the B register will be used.

That covers k' = 0 which is DIRECT ACCESS.
In all the accesses shown in fig. 7, N is an unsigned positive quantity except when it is a literal, or when added to PC.

1.10 Indirect Access Decode

k' = 1, 2 and 3 all indicate indirect access format.

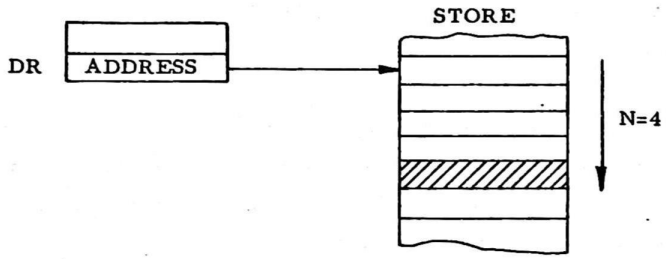
1.10.1 k' = 1 This indicates indirect access via a descriptor in DR.
k" = 0 Fig. 10 (a) shows DR pointing to a location in store. The address of the location is modified by the literal N and the item thus accessed is the operand.
k" = 1 Bit string handling (see 1.12)
k" = 2 Here the address in DR is modified by the contents of the location LNB+N. Fig. 10 (b) uses N = 4, the contents of LNB+4 are 2, so the operand is to be found in the location pointed to by the address in DR modified by two.

Similarly for k" = 3, 4 and 5 the address in DR is modified by (XNB+N), (PC+N) and (CTB+N) respectively to access the operand. For k" = 6 the address in DR is modified by the item at the top of stack.
k" = 7 Contents of B modified by N, is used as the operand (the exception case).

1.10.2 k' = 2 Moving on to k' = 2, this deals with indirect access via a descriptor in store.
We'll ignore k" = 0 for now and come back to it later (section 1.11).
k" = 1 See section 1.12.
k" = 2 Fig. 11 (a) shows a descriptor occupying the locations LNB+4, LNB+5 on the stack. The address field of the descriptor points to an item in store, which is used as the operand.
k" = 3 If XNB points within the stack, then the operand access is similar to that for IL N. However, if XNB points at say a PLT then the operand access is as shown in fig. 11 (b). N, which is a word address, is added to the address of the start of the PLT and the relevant descriptor is accessed. In the diagram N=6, so the fourth descriptor which is at XNB+6 and XNB+7 is accessed. The address field of this descriptor points at the operand.

Fig. 10. INDIRECT OPERAND ACCESS via Descriptor in DR

(a) D N



(b) DL N

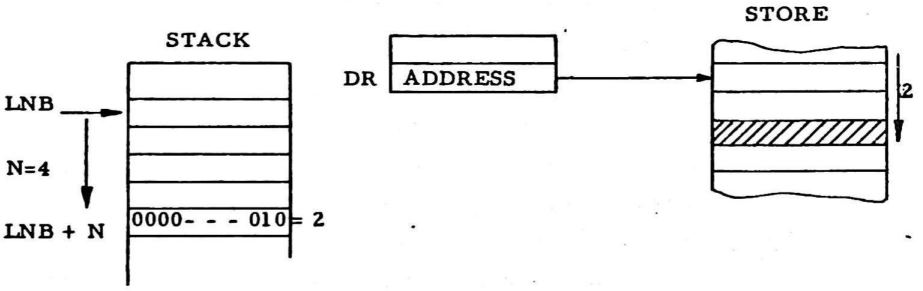
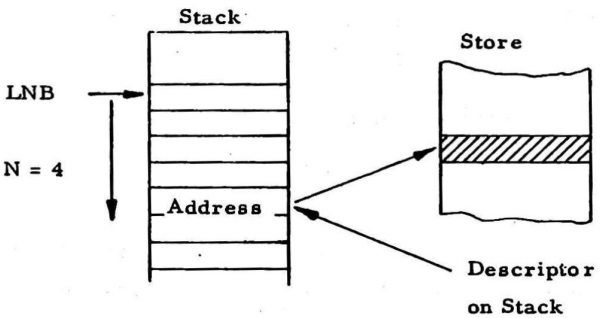
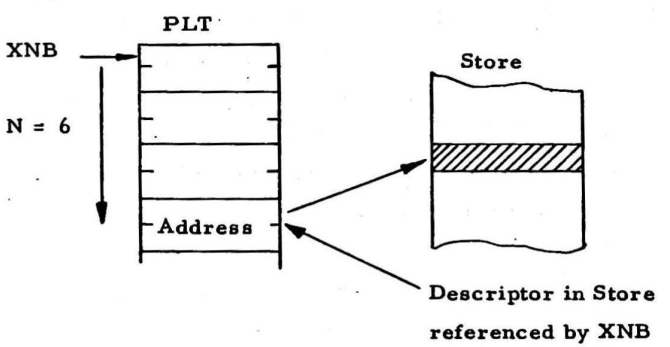


Fig. 11 - INDIRECT OPERAND ACCESS VIA A DESCRIPTOR

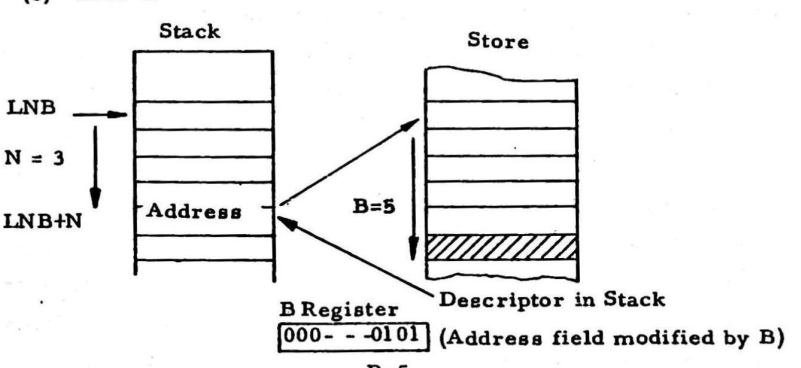
(a) IL N



(b) IX N



(c) MIL N



k'' = 4, 5. There is a descriptor in store at PC + N and CTB + N, respectively, and it is the address field which points at the operand to be used.

k'' = 6, 7. Indicate that the descriptors at TOS and in DR, respectively point at the operand.

1.10.3 k' = 3 deals with INDIRECT ACCESS via a descriptor in store. We'll tackle k'' = 0 later (section 1.11)

k'' = 1 see section 1.12

k'' = 2. There is a descriptor on the stack at LNB + N and LNB + N + 1

The address field points to an area in store. The address of this area is modified by the contents of the index register, B to give the operand required. See fig 11 (c).

k'' = 3, 4, 5. Similarly the addresses of the descriptors at (i) XNB+N and XNB+N+1, (ii) PC + N and PC + N + 1, (iii) CTB + N and CTB+N+1, respectively are modified by the contents of the B register to access the operand.

k'' = 6, 7. Indicates that there is a descriptor at TOS or in DR, respectively. The address field is modified by B to give the operand required.

1.11 Image Store Formats (k'' = 0 for k' = 2 and 3)

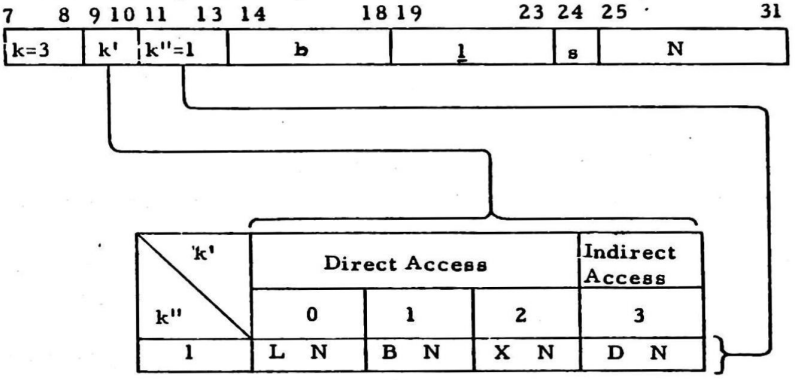
These primary formats are I. S. location N and I. S. location B - these are the IMAGE STORE formats. The image store mechanism is a method of addressing hardware registers in the system. It provides a method of performing operations at a hardware level, which for reasons of privilege, timing or level of control could not be performed at a software level, it is the image store mechanism which is used to kick off peripheral transfers. Two options of image store addressing i.e. I. S. location N and I. S. location B are offered. The former allows addressing up to $2^{18} - 1$, since N is an 18 bit literal (extended with zeros on the left). The latter allows addressing up to $2^{32} - 1$, since the contents of the B register are used.

1.12 Bit String Handling

Facilities for bit string handling are available in machines at AML 1.

NOTE:- In this section the lower case letter L is shown as l to differentiate it from 1 (the number one).

Format of Operand Field k = 3 and k'' = 1.



Where:

b and l define a bit string within a 32 bit word

b = number of most significant bit of the string

l = one less than the number of bits in the string.

(e.g. b = 7, l = 11, defines a 12 bit string occupying bits 7 to 18 inclusive of the 32 bit word.

For indirect access, the descriptor in DR must be a word vector descriptor type 0, size 5 (32 bits) and must be scaled i.e. USC not set.

The effect of the operand depends upon the instruction as follows:

(a) Operand fetch: The contents of the bit string location become the least significant l + 1 bits of the operand. The other bits of the operand are set as follows:

if s = 0, the filler is zero

if s = 1, the filler is the most significant bit of the bit string.

(b) Operand store: The least significant $\underline{1} + 1$ bits of the operand are stored in the bit string location.

if s = 0, the remainder of the operand is checked to be all zero.

if s = 1, the remainder of the operand is checked to be the same as the most significant bit of the stored bit string.

Note that the 32 bit word containing the bit string is written to store without hardware interlocks to prevent access from other units in the system, (except as defined for particular functions - INCT, TDEC)

There will be a program error interrupt if:

(i) bit string operand is accessed via (DR + N) and descriptor in DR is not type 0, size 32 bits, USC unset.

(ii) bit string operand has $b + \underline{1} > 31$ i.e. the bit string does not lie within a 32 bit word.

(iii) truncated part of operand not all zeros or all ones as defined in (b) above.

1.12.1 Functions Prohibited with this format.

Bit string operands are prohibited for the following functions

(i) J, JLK, CALL and DEBJ

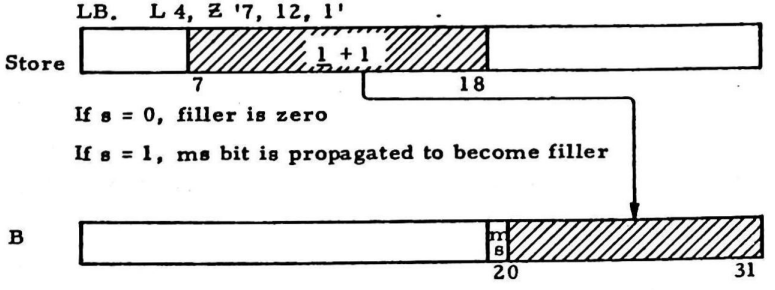
(ii) Functions which require literal operands

i.e. ESEX, IDLE, PRCL, RRTC, CDEC and CBIN.

1.12.2 Example 1: Load B from the location indicated by LNB+4.

The instruction in SFL would be:

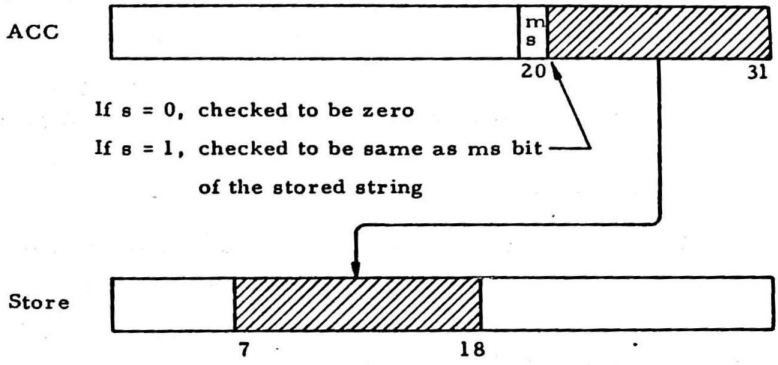
```
LB.   L 4, Z '7, 12, 1'
```



Example 2: Store contents of ACC

The instruction in SFL would be:

ST. L 4, Z '7, 12, 1'



Example 3: Clear bits 7 to 18 (inclusive) of a word located at LNB+4.

The instruction in SFL would be:

CLR. L 4, Z '7, 12, 0'

Here k = 3 b = 7 the first bit number

k' = 0

k'' = 1 $\underline{1}$ = 11 which is one less than the number of bits in the string

s = 0

Example 4: Test bits 1 and 2 of the word FRED

The instruction in SFL would be:

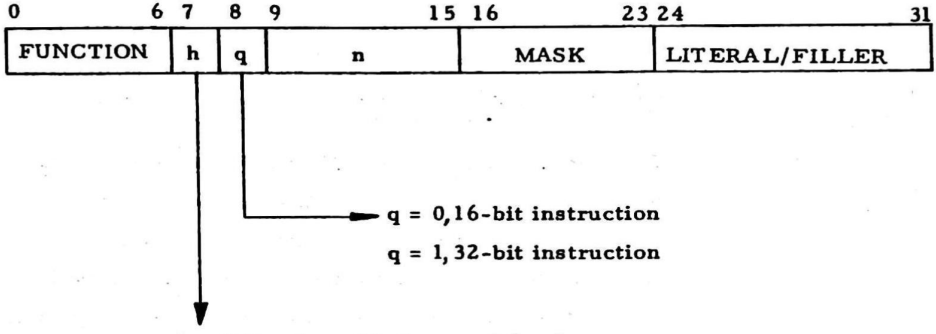
TEST. FRED, Z '1, 2, 0'

1.13 Operand Format of Secondary Functions

There are 16 possible functions in this group, not all of which have been allocated. They are included in the instruction repertoire specifically to assist commercial data processing establishments, where the handling of variable length data items is of paramount importance. All of these instructions are store-to-store functions. The range of instructions is quite comprehensive and it is possible to perform the following types of operations:-

1. Move a string of bytes from one part of store (referred to as the SOURCE) to another part of the store (known as the DESTINATION), carrying out editing or translating during the transfers.
2. Checking to determine if a pair of strings of bytes overlap in store.
3. Scan a string of bytes, searching for equality with a specified byte pattern.
4. Boolean operations between strings.
5. Scan a string of bytes, checking a table of check bits.
   Translate the contents of a string of bytes, according to a table of bytes, called a 'translate table'.
6. Pack and unpack decimal data.

Fig 12 SECONDARY FUNCTIONS (STORE-TO-STORE OPERATIONS)



Number of Bytes involved in a Store-to-Store Operation is referred to as L.

The secondary functions may be divided into the following 3 classes:-

Class 1 These operations take place between two byte strings. The SOURCE string is usually specified by a string descriptor held in the ACCUMULATOR - hence this can be referred to as the (ACC) string - and the DESTINATION string is normally specified by another descriptor, held in DR - generally referred to as the (DR) string.

Class 2 ACC is not involved in this class. The source string consists of a series of copies of a single byte - also called REFERENCE byte. In the case of the 16-bit instructions, the byte is obtained from the index register B. In the 32-bit format, the byte is normally obtained from the field which contains the literal filler byte.

Class 3 For these functions, the (DR) string interacts directly with the contents of the ACC itself.
The format of secondary instructions is shown in figure 12. Bits 0-6 contain the function field, bit 7 is the h-field, bit 8 is the q-field and bits 9-15 contain the n-field. For a 32-bit instruction, bits 16-23 contain the MASK byte and bits 24-31 the LITERAL/FILLER byte.

Taking the h-field first - this defines how the number of bytes(L) involved in a store-to-store operation is to be specified. If h = 0 then L is defined explicitly as $L = n+1$ where n is an unsigned integer in the range $0 < n < 127$. Of course, the n-field referred to, is the one in bits 9-15 of the instruction. The alternative way of defining L is implicitly, as the length field of the (DR) descriptor where L is in the range $0 < L < 2^{24}$. Here h = 1 and the n-field of the instruction is classed as reserved. The q-field indicates the length of the instruction: q = 0 is used for a 16-bit instruction. The significance of the mask and literal/filler bytes will be explained in detail in Chapter 10 which deals with store-to-store functions.

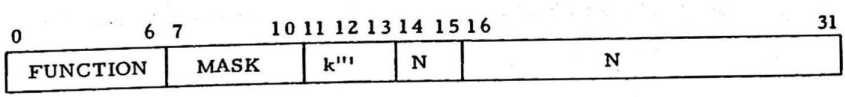
1.14 Operand Format of Tertiary Functions

Only three functions of this type have so far been assigned - See fig 13. They are all jumps and have the mnemonic codes JAT (jump on arithmetic true), JAF (jump on arithmetic false) and JCC (jump on condition code) - they will be described in detail in Chapter 6.
The format of the operand field is also shown in fig 13. Bits 0- 6 contain the function, bits 7-10 the MASK field and bits 11-13 the k''' field. For a 16-bit instruction, bits 14 and 15 are both zero, but for 32-bit instructions, bits 14-31 represent N.

Fig 13 TERTIARY FUNCTIONS

JAT Jump on Arithmetic (Condition) True
JAF Jump on Arithmetic (Condition) False
JCC Jump on Condition Code

TERTIARY GROUP FORMATS



k'''	OPERAND	
000	N	32 Bit Instruction
001	D N	
010	L N	
011	X N	
100	P N	
101	C N	
110	D	16 Bit Instruction
111	MD	

NOTE For 16-bit instruction, bits 14, 15 = 0
For 32-bit instruction, N = bits 14-31

The k''' - field provides for 8 different operand types. Note that for k''' = 0 or 4 the contents of the N-field are interpreted as a signed integer, in all other cases N is an unsigned integer. When k''' = 0, the contents of the N-field are treated as a number of half-words and are added to the current contents of the control register PC. In this case we have a relative jump. An interrupt will occur if the addition alters the segment number part of PC. Note that k''' = 6, 7 implies a 16-bit instruction.

1.15 Function Mnemonics

When writing programs in SFL, the mnemonic form of an instruction can be used rather than its hexadecimal representation. See fig 14.

Thus if we wish to load the B register with a value of -19, the hexadecimal representation of the instruction would be #7A6D. However, it is perfectly in order to write LB -19.

When the program is compiled the mnemonic form will be translated into the hexadecimal equivalent, which will be understood by the machine.

This mnemonic facility is useful since a program can be written more easily and far more quickly this way than by working out the hexadecimal pattern. Also it is much less prone to errors - once the compiler has been tested and is working it will always decode the mnemonics correctly, which is more than can be said for each programmer doing it individually.
Figure 14 shows three examples of the use of mnemonics in writing programs, and illustrates (in hexadecimal representation) how it would appear in store.

Fig 14 FUNCTION MNEMONICS

Format: function mnemonic operand

Example 1 LB -19 loads B register with -19

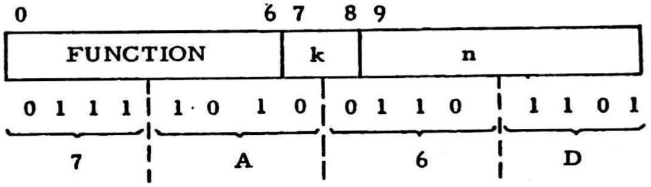
This is a Primary Instruction, therefore the Primary format is used, as shown in figure 5

function # 7A ≡ LB

operand n = -19 ≡ 1101101 (binary)

k = 00

Thus hex equivalent of LB -19 is # 7A6D:



Example 2 LB. L8 loads B register with the contents of location LNB+8. (k=01) The hex. pattern is 0111/1010/1000/1000 (i.e. # 7A88).

Example 3 LB IL 5 accesses descriptor in LNB+5, LNB+6. The address field points to an item in Store. This item is loaded into B. The hex. pattern is 0111/1010/0000/0101 (ie # 7A05). (k=10)

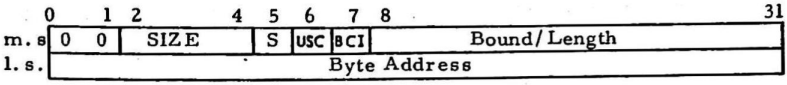
2. DESCRIPTOR FORMATS

All descriptors are 64 bits in length. The less significant 32 bits always contain a byte address, which may be modified in the course of accessing the information to which the descriptor refers.
The resultant address points to the left most (lowest addressed) byte of the information.

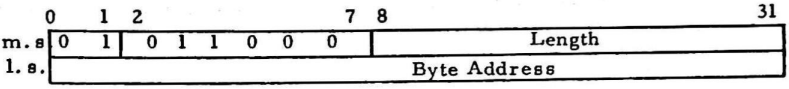
Descriptor types are distinguished by their more significant 32 bits.

The different types, and interpretations of the fields are as follows:-

Type 0 - Vector Descriptors



Type 1 - String Descriptors



Type 2 - Descriptor Descriptor

As for type 0 except bits 0, 1 = 10 and only size 64 allowed.

Type 3 - Miscellaneous Descriptors

Bits 2-7 define a subtype number

Subtype
- 32 - bounded code, first byte = 'E0'
- 33 - Unbounded code, first byte = 'E1'
- 34 - System Call, first byte = 'E3'
- 37 - Escape, first byte = 'E5'
- 40 - Semaphore (bounded), first byte = 'E8'
- 41 - Semaphore (unbounded), first byte = 'E9'
- 63 - Null, first byte = 'FF'

FIG 15 DESCRIPTOR FORMATS

2.1 Type 0 = Vector Descriptors (fig. 15)

Size The size of the addressed item in store. Permitted sizes, and the corresponding size codes, are as follows:-

Size (bits)	Code
1	0
8	3
16	4
32	5
64	6
128	7

When the size is 32, 64 or 128 bits, the two least significant bits of the byte address, after modification, if any, are ignored - i.e. 1-, 2-, and 4- word items are made to start on word boundaries. Use of other values will cause program errors as indicated in Chapter 12. When the size is 16 bits, the l.s. bit of the byte address after modification is ignored. Size code 4 is not available at AML0.

S Signed. If set to 1, then

a) if the operand is read from Store and the addressed item is smaller than the operand length, it is sign-extended.

b) if the operand is written to Store and the addressed item is greater than the operand length, the truncated bits are checked to be equal to the most significant bit of the stored item.

The S field is ignored at AML0.

USC Unscaled. Unless this bit is a 1, when a modifier is added to the address field it is scaled according to the size field; 1, 2, 3 and 4 places up for 16, 32, 64 and 128 bits, respectively, and 3 places down (logically) for 1 bit. In the latter case the least significant 3 bits of the shifted-down modifier specify the individual bit number (0 = most significant) which is to be accessed within the addressed byte. If the modifier is unscaled the accessed bit-number is undefined: however if the descriptor is unmodified, bit 0 is accessed.

BCI Bound Check Inhibit. Unless this bit is 1 any modifier added to the address is checked (before scaling) to ensure that it is less than the contents of the Bound field; in this case bits 0-7 of the 32-bit modifier must be all zeros.

Bound/ Length The contents of this field are unsigned (positive).

When a byte-vector descriptor, i.e. one with Type = 0, Size code = 3, is used as the operand of a store to store instruction, this field contains the length of the byte string. On other occasions when vector descriptors with any permitted size code are used, this field is spare if BCI = 1; if BCI = 0 its contents should be 1 greater than the largest permitted modifier.

2.2 Type 1 = String Descriptors (fig. 15)

Size Should be set to 011 - this is checked by store-to-store instructions; at any other times it is ignored (reserved).

S This bit is reserved for use within the I/O sub-system and is ignored by the OCP.

USC, BCI These fields are ignored (reserved) and should be set to 00. Modifications are not scaled or checked.

Length The length field contains the length, in bytes, of the string whose first byte is addressed by the contents of the address field (modified if the instruction specifies modification).

2.3 Type 2 = Descriptor Descriptors

Size and S fields i.e. bits 2-5, are ignored (reserved) and should be set to 1100. These function just like type 0 descriptors with size code 64 bits, and are interchangeable with the latter.

2.4 Type 3 = Miscellaneous Descriptors (fig. 15)

Bits 2 - 7 define a subtype number.

Subtypes (numbered decimally):

32, 33 Code (Bounded, Unbounded)

Code descriptors may be used to point to the destination instructions of Jump, Call and Exit instructions only. Bits 32-63 contain the byte address of the first byte of the instruction - bit 63 is ignored as instructions are halfword aligned. Any modifier added to the address is multiplied by 2 before addition. If subtype 32, bits 8-31 contain a bound which is used to check the modifier, if any, in exactly the same way as for type 0 and type 2 descriptors. If subtype 33, bits 8-31 may contain the identity of a microcode subroutine which may be entered after PC is set. If the microcode subroutine does not exist or if an error is encountered within the routines a jump is made to the instruction addressed by PC.

35 System Call

Bits 8-31 usually contain an entry displacement to index a System Call Index Table. Bits 32-63 usually contain an entry displacement to index the System Call Table indicated by the descriptor accessed from the System Call Index Table. System Call descriptors are only used by the Call and Exit instructions - in the latter case as a 'link descriptor'.

37 Escape

Escape descriptors are used to by-pass normal instruction sequencing rules. Whenever a descriptor in DR, which is being modified by MODD, or used to access information indirectly is found to be an Escape descriptor, a branch out of sequence occurs. Bits 32-63 contain the address of a word whose contents will be transferred to PC as part of the escape action. Bits 62, 63 are ignored so the address is word-aligned. Escape descriptors are not modified. Bits 8-31 are ignored (reserved).

40, 41 Semaphores (Bounded, Unbounded)

These descriptors are used to point to semaphore locations in store. The format is similar to a type 0, size code 5

descriptor and modification rules are the same.
The descriptor is restricted to use with INCT and TDEC instructions. The effect of use with other instructions is undefined. The descriptor must not be used to access the stack segment.
Access to the word pointed at by the descriptor is forced by hardware to bypass slave storage and is implemented by a Read Hold, Write Hold combination so as to prevent access to the store location while the word is being modified.
If slave storage is present, use of this descriptor clears the operand slave store of items from segments marked non-slaved (NS).

63 Null

A null descriptor is used to provide the NIL option when a descriptor is used as an optional parameter for a procedure. It is detected by the VALIDATE instruction and is invalid with all other instructions.
This facility is available at AML1.

3 OPERAND ADDRESSING AND ALIGNMENT

3.1 Operand Length (Primary and Tertiary Formats)

By 'operand length' is meant the number of bits in the addressed quantity on which an instruction actually operates. In the case of an instruction operating on ACC, this is usually determined by ACS, but not always - for instance in a Floating Divide Double instruction the operand length is $\frac{1}{2}$ ACS, for a Scale instruction the operand length is 32 bits regardless of ACS. The operand length is not necessarily the same as the length of the addressed item in store. When the latter is specified by a descriptor - e.g., a 5-byte string addressed by a string descriptor, or a single bit or byte addressed by a vector descriptor with the appropriate size indication - it may be used in an instruction for which the operand length is 64 bits. The rule is that when an operand is read from store or a register, the length of the addressed item must not exceed the operand length. However, if the length of the addressed item is smaller, after the item has been read it is extended with left hand zeroes to the required length.

When an operand is written into a store location or register of different length:

(a) If the stored item is addressed by a string (type 1) descriptor, its length must not exceed the operand length. In other cases, if the operand length is smaller, the operand is extended with left hand zeros to fill the store space.

(b) If the operand length is greater, the operand is truncated on the left until it fits the store space.

If the source or destination stored item is addressed by a string (type 1) descriptor and its length is zero or exceeds the operand length, a non-maskable program error (descriptor) interrupt occurs. If the length of an item read from the store other than via a string

descriptor exceeds the operand length, or if non-zero bits are truncated from an operand being written to the store or to B, the operation is suppressed and a Size interrupt occurs, unless the interrupt condition is masked. If the condition is masked the operation is not suppressed; the leftmost portion of the item taken from the store is ignored in the first case, and the non-zero bits truncated are treated as zeros in the second.

The exception to this rule occurs in the case of Jump-type instructions, where the operand, which overwrites PC, is conceptually 32 bits long: however when the operand is specified indirectly via a descriptor its length in store is permitted to be 32 or 64 bits, and in the latter case the least significant 32 bits overwrite PC, the more significant 32 bits being ignored.
Type 1 descriptors are not permitted. When accessing image store locations, the operand length must be 32 bits; otherwise, the action is undefined. The operand length required by each instruction is listed with the instruction description. The operand lengths for store-to-store instructions are specified in the secondary instruction format (see section 1.12).

3.2 Addressing Rule

The address of an item in the store is the address of its left-most (lowest numbered) byte. Where individual bits are addressed by modified vector descriptors, the bit number, from 0 (left-most bit) to 7, is concatenated to the address of the byte.

3.3 Word Alignment

Operands directly addressed in the store (i.e. using the operand forms)

T, L n, L N, P N, X N, C N and B N.

as well as modifiers and descriptors used in the corresponding indirect forms, start on word boundaries - i.e. their byte addresses are multiples of 4. This is ensured automatically thus:-

T, L n, L N, X N, C N : SF, LNB, XNB and CTB contain word aligned addresses.

P N : the least significant bit of the sum is ignored.

B N : the 2 l.s. bits of the sum are ignored.

Note that 64 and 128-bit items are not constrained to be on double or quadruple-word boundaries in store. Therefore such items are liable to cross page boundaries, or violate segment limits, even when the addresses of their first words have been checked and found 'legitimate' - the final address must be checked too.

3.4 Justification in Registers

In general quantities transferred from store, or as literals from the instruction format, to registers, and vice-versa, are right-justified in both registers and store locations. Sign extensions or zero filling on the left takes place according to rules stated elsewhere. Thus, in calculating the value of 'PC+N', N is assumed to be in the same units as the contents of PC, i.e. halfwords, and signed; while in calculating 'LNB + N', N is considered to be a number of words, and is unsigned, i.e. positive. An exception to this rule occurs when a stored quantity represents a virtual address, in which case it is a byte address; this particularly applies to the items transferred to PC by jump instructions. Thus for 'Load LNB' the operand is a byte number whose least significant 2 bits are ignored, rather than a word number.

3.5 Primary and Tertiary Format Operands

Literals

The operand forms n and N cause the operand to be generated by extending the quantities n (7 bits) or N (18 bits) on the left with copies of their most significant bits, to the required operand length. A signed literal specified as the operand for a jump instruction will be added to, rather than overwrite, PC. Interrupt occurs if this alters the segment number in PC.

Image Store

The operand forms G and GB cause the 32-bit image store location indicated by N, or by the contents of the B register to be accessed.

Top of Stack

The operand T causes the item at the top of the stack (of length = operand length) to be used as operand, and SF to be decremented by the operand length in words. Program error interrupt occurs if this causes SF to become < LNB.

For Store-type instructions the result is stored as a new top-of-stack item causing SF to be incremented. If storing the result violates the stack segment limit a virtual store interrupt occurs.

B

Causes the 32-bit contents of B to be read (extended with zeros if necessary) or overwritten.

Directly-Accessed Items in Store

For the operand forms L n, L N, X N, P N, C N and B N, the address of the operand is formed by adding N (or n) to the appropriate pointer location, to form a byte address which is a multiple of 4.

gth of item accessed = operand length. The rules for checking this addition vary from one form to another, as shown below:-

L n, L N : n extended with zeros; m.s. 2 bits of N must be zeros. No carry out of LNB permitted.

X N, C N : No check
N extended with zeros.

P N : N is regarded as a signed half-word displacement. Bits 1-17 of N are added to bits 14-30 of PC. Carry out of bit 14 of PC is checked equal to bit 0 of N, and is not added to bit 13, i.e. segment overflow is forbidden. The least significant bit of the sum is ignored. The operand must be wholly in the current code segment.

B N : No check - N is extended with zeros and is regarded as a word displacement. The 2 l.s. bits of B are ignored.

Failure of any check causes an interrupt.

Indirectly - Accessed Items in Store

The operand is accessed via a descriptor at the specified location.

There are two cases: in one case the descriptor is accessed like a directly accessed operand (i.e. as described above but of length = 64 bits) - the descriptor may be modified by the contents of B - and in the other case the descriptor is already in DR and may be modified by a directly accessed quantity. In both cases the descriptor is left in DR unmodified, after use, Unless otherwise stated, the descriptor may be of types 0, 1, 2 or Escape for any primary instructions.
For jump instructions (including Call) the same rules apply except that type 1 descriptors are not allowed, and the size code in a type 0 descriptor may only be 32 or 64 bits. For a 'Call' instruction 'Code' or 'System Call' types are also permitted. If the descriptor is of Escape type a jump out of sequence occurs.

Modifiers, whether obtained as directly accessed quantities or from B, are 32 bit quantities. When N is used as a modifier it is extended with zeros on the left. The modifier is added to the contents of the address field - other fields are unaffected.

When the modifier is added, a full 32-bit addition is performed, overflow due to scaling or to the addition being ignored. When bound checking is required (BCI not set) the most significant 8 bits of the modifier before scaling must be 0's and it must be less than the contents of the bound field, otherwise bound check interrupt occurs.

The checks for the directly-accessed items in store apply when any of these items is used as a descriptor or modifier. The rules for accessing and aligning operands are given in 3.1. Modifiers or descriptors taken from the top of the stack (using the forms given by k" = 6) cause SF to be decremented by 1 or 2 words, respectively.

An instruction may overwrite store locations which contained parts of the descriptor or descriptor modifier that it used to address its operand. This includes cases where the descriptor or modifier was the top-of-stack item and the instruction (e.g. Remainder Divide, or Stack-and-Load types) causes something to be stacked, though the operand itself is not on the stack.

3.6 Secondary Format Operands

The secondary format is only used by store-to-store functions. ACC may contain a source descriptor and DR contains the destination string descriptor (an Escape descriptor may be used in place of the latter).

4 STACK INSTRUCTIONS

4.1 Accessing the Stack

This chapter deals with the instructions which access the stack. Figure 16 should remind you of what the stack looks like - it is a last-in, first-out storage area which is 32 bits wide, i.e. it expands and contracts in steps of 32 bits (one word). The stack can be addressed by means of a number of hardware registers:-

SF - STACK FRONT (16 bits) may be regarded as a pointer, relative to the base of the stack, to the first unoccupied word in the stack. The instruction code allows for the removal of the top item from the stack, or the addition of a new item to the top of the stack - these operations automatically cause the contents of SF to be decremented, or incremented, by the appropriate number of words. SF may also be altered by an instruction which allows a specified number of locations to be added to, or deleted from, the stack. Since SF is only a relative pointer it ceases to have significance in another stack.

LNB - LOCAL NAME BASE (16 bits) contains the word address, relative to the base of the stack, of the first location in the local name space of a procedure. Again it does not have any significance outside its own stack. The quantity in LNB is always less than that in SF. The order code provides facilities for addressing items in the local name space relative to LNB.

There is an additional register XNB - EXTRA NAME BASE (30 bits) which may point to a word within the stack: however, since the length of XNB is 30 bits, it is not restricted to pointing within the stack- it may also be used for directly addressed off-stack areas. The order code allows for the addressing of items relative to XNB.

The TOP OF STACK (TOS) item may be 1, 2 or 4 words long - it is the programmer's responsibility to keep track of the data items used

and to ensure that the correct operand length be used in all operations.

For stack instructions the following restrictions apply:-

1. SF must always be greater than LNB.
2. Addressing below LNB i.e. (LNB - n) is not possible.
3. SF is not permitted to overflow into another segment.

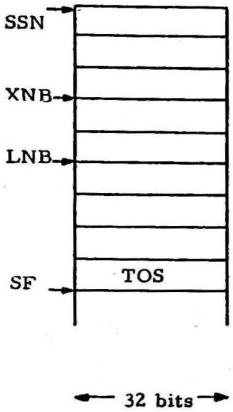
Thus we note that we cannot unstack below LNB and cannot obtain access to another local name space.

Now we are going to tackle each instruction in the order code in turn. For each instruction, there is a mnemonic code assigned, which is shown in brackets and this is followed by the function code in hexadecimal representation. The full instruction set is shown in Appendix 1.

4.2 Load LNB (LLN) #7C (see fig 17)

Operand length	:	32 bits
Description	:	Bits 14-29 of the operand are loaded to LNB. Bits 30, 31 are ignored. Bits 0-13 are checked equal to SSN. The new value of LNB is checked to be less than SF. LNB is unaltered if these checks are not satisfied.
CC	:	Unaltered
Program errors	:	Operand address errors. Bits 0-13 not equal to SSN (see 12.4/8.2). Bits 14-29 ≥ SF (see 12.4/8.3)

Fig 16 THE STACK



Hardware Registers

SSN = Stack Segment Number (14 bits)

LNB = Local Name Base (16 bits)

SF = Stack Front (16 bits)

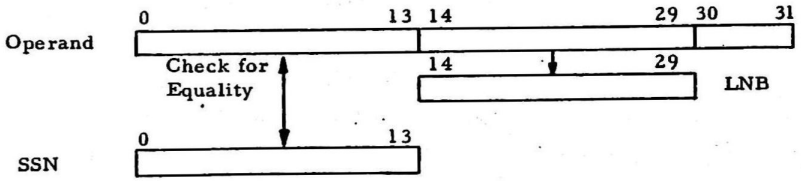
XNB = Extra Name Base (30 bits)

Only XNB can point outside as well within the Stack, the rest can only point within the Stack.

TOS = Top Of Stack (item).

Fig 17 LNB - LOAD AND STORE INSTRUCTIONS

LOAD LNB (LLN)



If $OP_{0-13} \neq$ SSN then program error interrupt occurs.

If LNB > SF then program error interrupt occurs.

4.3 Store LNB (STLN) #5C (see fig 18)

Operand length : 32 bits

Description : The contents of LNB, expanded to a 32-bit byte address by concatenating the contents of SSN on the left and 2 zero bits on the right, is stored. This instruction will usually be used to 'stack' the contents of LNB prior to a procedure call.

CC : Unaltered

Program errors : Operand addressing errors
Literal operand (see 12.8/12.1)
Non-zero bits of stored item truncated. (see 12.3/6.0)

4.4 Raise LNB (RALN) #6C (see fig 19)

Operand length : 32 bits

Description : LNB is set equal to the value of SF minus the operand. The operand is regarded as a number of words, which must be less than the word address in SF (so operand bits 0-15 must be zero), and the new value of LNB must not be less than the old. LNB is unaltered if these checks are not satisfied.

CC : Unaltered

Program errors : Operand addressing errors.
Operand $\leq$ 0 (see 12.4/8.4)
Operand > SF - LNB (see 12.4/8.5)

4.5 Load XNB (LXN) #7E (see fig 20)

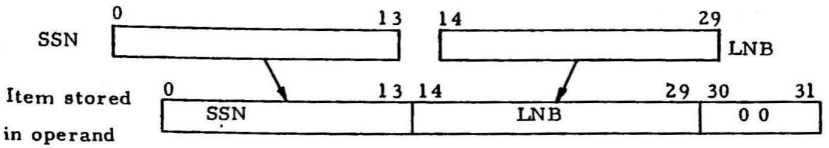
Operand length : 32 bits

Description : Bits 0-29 of the operand are loaded to XNB. Bits 30, 31 are ignored.

CC : Unaltered

Program errors : Operand addressing errors

Fig 18 STORE LNB (STLN)



Literal operands not permitted.

Fig 19 RAISE LNB (RALN) RALN 5 sets LNB equal to the value of SF minus 5.

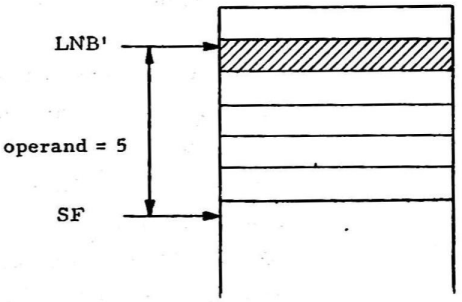
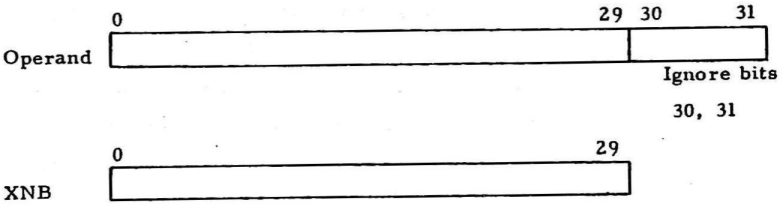


Fig 20 LOAD XNB (LXN)



Store XNB (STXN) # 4C

Operand length : 32 bits

Description : The contents of XNB, expanded to a 32-bit byte address by concatenating two zero bits on the right, is stored.

CC : Unaltered

Program errors : Operand addressing errors, Literal operand (see 12.8/12.1). Non-zero bits of stored item truncated.

4.6 Adjust SF (ASF) # 6E (see fig 21)

Operand length : 32 bits

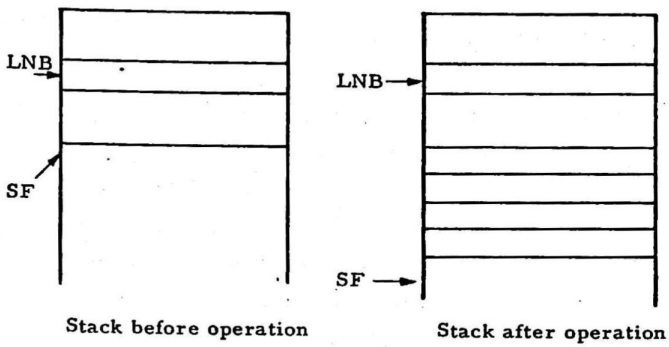
Description : The operand, regarded as a signed number (of words), is added to the word address in SF. Bits 0-15 of the operand must be all the same and must equal the carry out of the most significant bit of SF when performing the sum, i.e. segment overflow is not permitted. The result must be greater than LNB, SF is not adjusted if these checks are not satisfied. New stack locations are not cleared.
If the operand involves TOS, SF is decremented before being adjusted. If the location pointed at by SF, after adjustment lies beyond the stack segment limit, or lies in a page which is not available in main store, a virtual store condition occurs, as if that location had been accessed. In this case SF is not adjusted but the adjusted address must be left in the VSI parameter. Bits 25 and 26 of the VSI parameter may indicate that either a read or write access was attempted.

CC : Unaltered

Fig 21

ADJUST STACK FRONT (ASF)

e.g. ASF 4

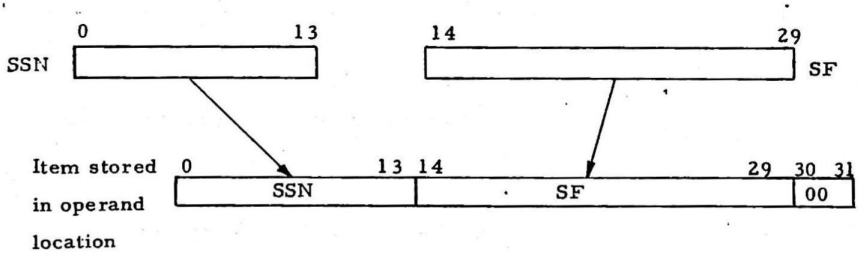


New SF$\leqslant$ LNB causes Program error interrupt

Operand too large (segment overflow) causes Program Error interrupt

Fig 22

STORE STACK FRONT (STSF)



Program errors : Operand addressing errors
New SF < LNB (see 12.4/8.6)
Operand too large (segment overflow) (see 12.4/8.7)

4.7 Store SF (STSF) #5E (see fig 22)

Operand length : 32 bits

Description : The contents of SF, expanded to a 32-bit byte address by concatenating the contents of SSN on the left and 2 zero bits on the right, is stored. In all cases, including those where the operand form involves the top of stack, the value of SF as it was at the beginning of the instruction is stored.

CC : Unaltered

Program errors : Operand addressing errors. Literal operand (see 12.8/12.1). Non-zero bits of stored item truncated (see 12.3/6.0).

4.8 Pre-Call (PRCL) #18

Operand length : 32 bits

Description : a) The operand is fetched. The operand must be a 7-bit literal.

b) If SF is even (bit 15=0), SF is incremented by 1.

c) The contents of LNB are expanded to a 32-bit byte address by concatenating the contents of SSN on the left and 2 zero bits on the right. Bit 31 is then set to 1 if SF was incremented in b) and the result is stacked. (SF is incremented by 1). Bit 31 of LNB as stacked is tested by the EXIT instruction, to control collapse of stack.

d) The action of Adjust SF (ASF) is now followed as described in 4.6. The operand is

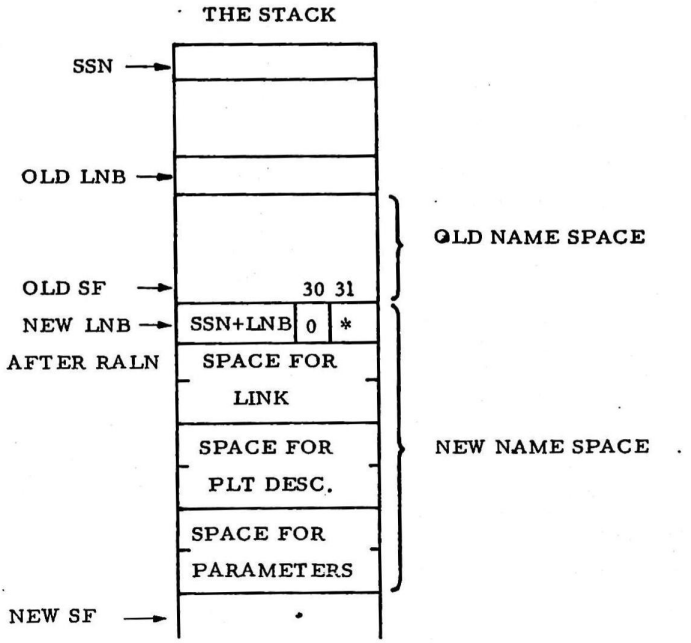
added to SF

CC : Unaltered.

Program errors: Operand addressing errors. New SF < LNB (see 12.4/8.6). Operand too large (see 12.4/8.7). Incorrect operand type (must be 7 bit literal) (see 12.8/12.2).

The Precall instruction may be used to set up the stack in preparation for the CALL instruction (see Section 6.8). SSN+LNB are stored at TOS and a new name space created for the Link descriptor, parameters and for procedure linkage table descriptors by adjusting Stack Front by a required amount as specified by the operand. See fig. 23.

Fig. 23 PRECALL



NOTE:* 1. For efficiency of Stack accesses on larger 2900 OCP's, if original SF is even, SF is first incremented by 1, and bit 31 of stored LNB is set to 1. This is to ensure that the items in the new name space will be aligned to an

even-word boundary after stacking SSN+LNB at TOS.

2. LNB must be raised to point to TOS by use of RALN instruction before the CALL instruction. (See section 4.4).

5 ACCUMULATOR INSTRUCTIONS

This chapter deals with the non-computational instructions which involve the ACC.

5.1 Load (L) #60

Operand length : ACS

Description : The operand is loaded to ACC. OV is cleared.

CC : Unaltered

Program errors : Operand addressing errors.

5.2 Set ACS 32 & Load (LSS) #62
Set ACS 64 & Load (LSD) #64
Set ACS 128 & Load (LSQ) #66

Operand length : New value of ACS

Description : A new value is loaded to ACS, and the operand (whose length is determined by the new value of ACS) is loaded to ACC. OV is cleared. There are three versions of the instruction, corresponding to the three possible values of ACS.

CC : Unaltered

Program errors : Operand addressing errors.

5.3 Store (ST) #48

Operand length : ACS

Description : The contents of ACC are transferred to the operand location. If the length of the latter is less than ACS, and any of the truncated more significant bits of the former are non-zero, an interrupt occurs. ACC is unaltered.

CC : Unaltered

Program errors : Operand address errors. Literal operand (see 12.8/12.1) Significant part of operand truncated. (see 12.3/6.0).

5.4 Load Upper Half (LUH) # 6A

Operand length	:	ACS
Description	:	ACS is doubled and the operand is loaded to the upper half of ACC. The lower half of ACC is unaltered. OV is cleared. ACS = 128 bits is not permitted.
CC	:	Unaltered.
Program errors	:	Operand addressing errors. ACS = 128 bits (see 12.9/13.4)

5.5 Store Upper Half (STUH) # 4A

Operand length	:	Half ACS
Description	:	The contents of the more significant half of ACC are stored in the operand location. ACS is halved. The lower half of ACC is unaltered. ACS = 32 bits is not permitted.
CC	:	Unaltered.
Program errors	:	Operand addressing errors. Literal operand (see 12.8/12.1) ACS = 32 bits (see 12.9/13.4) Significant part of operand truncated (see 12.3/6.0).

5.6 Copy DR (CYD) # 12

Operand length	:	Not applicable. Literal must be specified.
Description	:	The contents of DR are copied to ACC. ACS is set to 64 bits. OV is cleared. DR is unaltered.
CC	:	Unaltered.
Program errors	:	Only universal types listed in section 12.10.

5.7 Read Real Time Clock (RRTC) # 68

Operand length : 1 bit. Literal must be specified.

Description : ACS is set to 64 bits and the value of the hardware real-time clock is loaded to ACC as follows:

(i) For OCPs at AML0.

The X register is loaded into bits 0-31 of ACC.

The Y register is loaded into bits 32-63 of ACC.

(ii) For OCPs at AML1.

a) if the operand value = 0, ACC is loaded as above.

b) if the operand value = 1, bits 0-62 of ACC are set to the true binary value of the Real Time Clock such that bit 62 of ACC is equivalent to 2 μ sec of real time.

The algorithm is:

$$ACC = 2.\ (x_{0-31} + (x_{31} + y_0)) + 2.\ y_{1-31}$$

CC : Unaltered.

Program errors : Only universal types listed in section 12.10.

Notes:

1. There are two 32 bit registers RTCX and RTCY.
2. In this instruction the operand is meaningless and a literal must be specified.

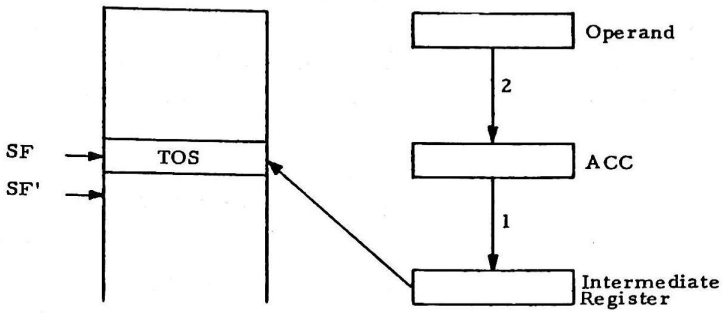
RTCX is maintained by software and can be considered as a continuation of RTCY, which is a hardware binary counter. Resolution at bit 31 of RTCY is 2 microseconds. By convention RTCX is a continuation of RTCY with bit 31 of RTCX duplicating bit 0 of RTCY. Whenever a carry from bit 1 of RTCY occurs an EXTERNAL interrupt is generated.

5.8 Stack and Load (SL) # 40 (see fig 24)

Operand length	:	ACS
Description	:	The contents of ACC are copied to an intermediate register. The operand is loaded to ACC, and the contents of the intermediate register are stacked, causing SF to be incremented by ACS. The intermediate register ensures that the operand forms T, DT, IT and MIT are valid. OV is cleared.
CC	:	Unaltered.
Program errors	:	Operand addressing errors.

NOTE:- If the operand is the top of stack item, i.e. SL/T then the contents of ACC are interchanged with the item at the top of stack. OV is cleared. The operand length equals ACS for this instruction.

Fig 24. STACK AND LOAD (SL)



Sequence of operation:-

1. Contents of ACC are loaded to an intermediate register.
2. Operand is loaded into ACC.
3. Contents of intermediate register put on top of stack. Thus stack is incremented by the original contents of the ACC.

If the operand is T, i.e. SL T, then SF is decremented as the operand is accessed and incremented as the original contents of ACC are put at TOS. Thus SL T swops the contents of ACC with the TOS item.

STACK, SET ACS 32 & LOAD (SLSS)
STACK, SET ACS 64 & LOAD (SLSD)
STACK, SET ACS 128 & LOAD (SLSQ)

This is as STACK & LOAD except that ACS is set to the appropriate size prior to loading the ACC.

5.9 Stack, Set ACS 32 & Load (SLSS) # 42
Stack, Set ACS 64 & Load (SLSD) # 44
Stack, Set ACS 128 & Load (SLSQ) # 46

Operand length : New value of ACS

Description : The contents of ACC (length determined by the original value of ACS) are copied to an intermediate register. ACS is set in a way depending on which of three versions of the instructions are used. The operand, of length determined by the new value of ACS, is loaded to ACC; and the contents of the intermediate register are stacked (causing SF to be incremented by the old value of ACS).
The intermediate register ensures that the operand forms T, DT, IT and MIT are valid.
OV is cleared.

CC : Unaltered.

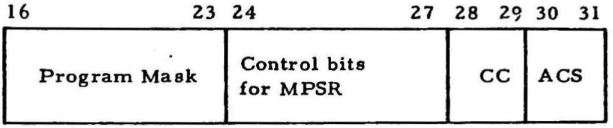
Program errors : Operand addressing errors.

5.10 Modify PSR (MPSR) # 32 (see fig 25)

PSR is the Program Status Register and controls fields such as overflow, Condition Code, ACS and Access Control Register.

Operand length : 32 bits.

Fig 25 MODIFY PSR (MPSR)



PSR (Program Status Register)

Only the l. s. 16 bits of operand are used.

If bit 27 = 1, bits 30, 31 overwrite ACS.

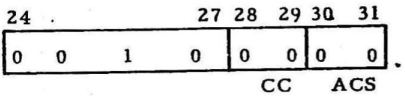
If bit 26 = 1, bits 28, 29 overwrite CC.

If bit 25 = 1, bits of program mask which correspond to zeros in operand bits 16 - 23 are made zeros.

If bit 24 = 1, bits of program mask which correspond to ones in operand bits 16-23 are made ones.

e. g. MPSR # 20

In this example is:-



Thus bit 26 is set to a one and CC is set to zero.

Description : The least significant 16 bits of the operand are used to alter the setting of the Program Mask, Condition Code, and ACS registers, as follows:

if bit 27 is 1, bits 30 and 31 overwrite ACS. (Program error if attempt is made to set ACS = 0).

if bit 27 is 0, ACS is unaltered and bits 30 and 31 may take any value.

if bit 26 is 1, CC is set to the value in bits 28 and 29. Otherwise, CC is unaltered and bits 28 and 29 may take any value.

if bit 24 is 1, bits of the Program Mask which correspond to 1's in operand bits 16-23 are made 1's; otherwise they are unaltered.

if bit 25 is 1, bits of the Program Mask which correspond to 0's in operand bits 16-23 are made 0's; otherwise they are unaltered.

bits 0-15 of the operand are ignored and may take any value.

CC : Unaltered if operand bit 26 = 0. Otherwise CC takes value specified in operand bits 28, 29.

(Note: ACS and/or CC may be set using a 7-bit positive literal operand).

Program errors : Operand addressing errors.

Attempt to set ACS = 0 (see 12.9/13.7)

Example:- Taking an example MPSR # 20. Here bit 26 is set to a one, which enables CC to be overwritten by the value of bits 28 and 29 - these are both zeros. Thus the net result is that CC is set to zero.

5.11 Copy PSR (CPSR) # 34

Operand Length : 32 bits

Description : The contents of the PM, CC and ACS fields of PSR are stored in the operand location, in the following 32-bit format:

Bits 0-15	0's
Bits 16-23	PM
Bits 24-27	1110 (see note below)
Bits 28, 29	CC
Bits 30, 31	ACS

Note: Subsequent use of this operand by 'Modify PSR' (5.10) causes PM and CC to be overwritten, but not ACS, unless bit 27 is made 1.

CC : Unaltered.

Program errors : Operand addressing errors. Literal operand (see 12.8/12.1).

Significant part of operand truncated (see 12.3/6.0)

6 CONTROL AND JUMP INSTRUCTIONS

6.1 Jump (J)#1A (see fig 26)

Operand length	:	32 bits (see note below)
Description	:	The operand increments or overwrites PC causing a jump to occur.
CC	:	Unaltered
Program errors	:	Operand addressing errors for jump instruction. (see 12.6/10.0)

NOTE:- PC, the program counter. holds the address of the current instruction and it is incremented for each instruction. When a jump instruction is encountered in a program, PC is overwritten with the operand, which is the jump address. Fig 26. demonstrates a use of this instruction. The stack is unaltered by the jump.

The operand length should be 32 bits. If the operand is addressed indirectly via a type 0 or type 2 descriptor, the addressed item in store may be 32 or 64 bits long. In either case it is treated simply as an instruction address and not as a descriptor - if it is 64 bits long, then the most significant 32 bits are ignored.

Fig 26 JUMP (J)

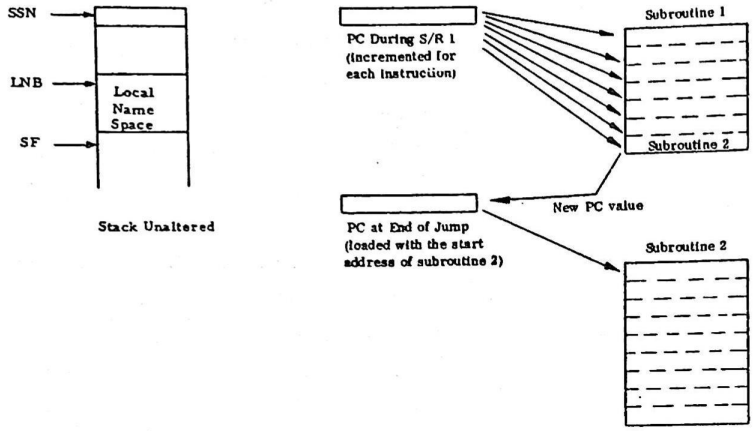
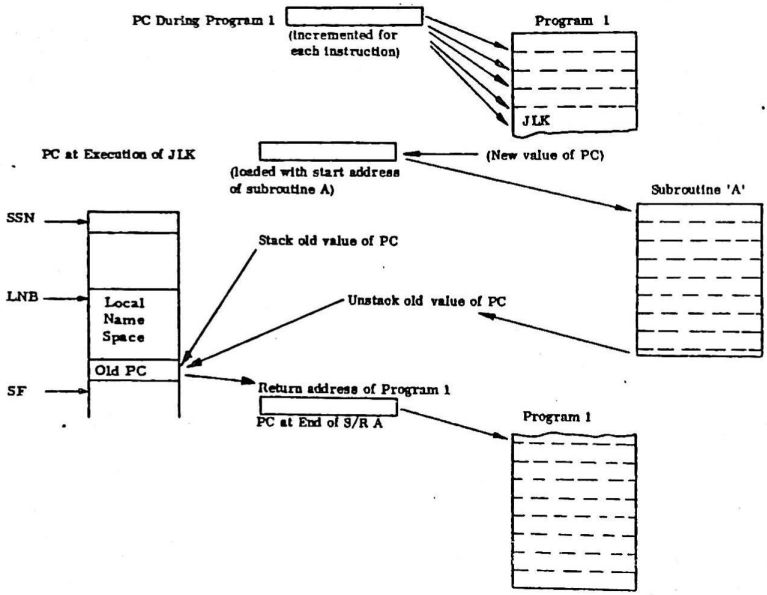


Fig 27 JUMP AND LINK (JLK)



6.2 Jump and Link (JLK) # 1C (see fig 27)

Operand length : 32 bits. The note under (6.1) applies

Description : The updated contents of PC are stacked as a 32 bit byte address (i.e. with a zero bit concatenated); SF is incremented by 1. The operand increments or overwrites PC causing a jump to occur. If the operand is the top-of-stack item, the updated PC and the operand are effectively swopped.

CC : Unaltered.

Program errors : Operand addressing errors for jump instruction (see 12.6/10.0).

This instruction can be used to access a subroutine (Fig 27). The address of the jump is stored at TOS and provides the link back to the original program. When we have finished with the subroutine the link is removed from TOS and overwrites PC. This provides the return to the original program.

6.3 Jump on Condition Code (JCC) # 02

Operand length : 32 bits. The note under 6.1 applies.

Description : If the bits of the mask field M are M_0, M_1, M_2 & M_3 and if the current condition code setting is i, then operate as for the Jump instruction if, and only if, $M_i = 1$; otherwise proceed to the next instruction in sequence. Alternative condition code settings may be tested by making more than one bit of M non-zero.

CC : Unaltered.

Program errors : Operand addressing errors for jump instruction (see 12.6/10.0).

This instruction uses the CC (Condition Code) Register which is set as a result of tests carried out on operands or registers. The

tertiary format is used for the instruction:-

JCC location, mask

where location is the destination of the jump. The mask field consists of 4 bits (M_0, M_1, M_2, M_3) which occupy bits 7-10 of the instruction (see Fig. 13) representing the four possible values of CC.

If M_0 is set to a one and CC = 0 then the instruction will cause a jump to the operand location.
If M_1 is set to a one and CC = 1 again a jump will occur. Similarly for M_2 and M_3.

JCC can be used in conjunction with orders such as CPB (compare B).

CPB is used to set CC as follows:-

If B = operand, CC = 0
B < operand, CC = 1
B > operand, CC = 2

so by using the instructions together:-

e.g. CPB FRED
JCC JIM, # C (mask field expressed in hexadecimal C = 1100 i.e. $M_0 = 1$, $M_1 = 1$, $M_2 = 0$, $M_3 = 0$) a jump to location will result if CC = 0 or CC = 1 i.e. if B ≤ FRED.

Certain mnemonics can be used in the mask:-

E (equal to) = 8 (1000)
L (less than) = 4 (0100)
G (greater than) = 2 (0010)

Thus JCC TOM, E will produce a jump to TOM if CC = 0.

6.4 Jump on Arithmetic True (JAT) # 04

Jump on Arithmetic False (JAF) # 06

These instructions use the tertiary format described in chapter 1.

Operand length : 32 bits. The note under (6.1) applies.

Description : These instructions test the contents of ACC, regarded as a floating-point, fixed point or decimal number, of DR, or of B, for one of the conditions specified by the mask field M, and a jump occurs (operand increments or overwrites PC) if the specified condition is true (first version) or untrue (second version). Otherwise the next instruction in sequence is obeyed.

ACC, DR, B and OV are unaltered.

CC : Unaltered

Program errors : Operand addressing errors for jump instruction (see 12.6/10.0).

The table of mask conditions is shown in fig 28.

e.g. JAT ALAN, # 4

will produce a jump to location ALAN if the ACC regarded as a fixed point number is zero. Note that JAF ALAN, # 4 will jump to ALAN if ACC $\neq 0$.

JAT and JAF do not use the CC register (unlike JCC).

6.5 Decrement B and Jump if Non Zero (DEBJ) # 24

Operand length : 32 bits. The note under (6.1) applies.

Description : 1 is subtracted from B. If the result is non-zero a jump is made, the operand incrementing or overwriting PC. If the result is zero no jump occurs and the next instruction in sequence is obeyed. In either case the decremented value is left in B. If B originally contained -2^{31} OV is set, $2^{31} - 1$ is left in B,

Fig 28 JUMP ON ARITHMETIC TRUE (JAT) (# 04)
JUMP ON ARITHMETIC FALSE (JAF) (# 06)

MASK		
0	ACC = 0	Floating Point Mode
1	ACC > 0	
2	ACC < 0	
3	Undefined	
4	ACC = 0	Fixed Point Mode (Undefined if ACS = 3)
5	ACC > 0	
6	ACC < 0	
7	Undefined	
8	ACC = 0	Decimal Mode
9	ACC > 0	
A	ACC < 0	
B	DR Length (bits 8-31) = 0	
C	B = 0	
D	B > 0	
E	B < 0	
F	OV Set	

JAT ALAN,# 4 will produce a jump to location ALAN if the ACC (regarded as a fixed point quantity) = 0.

JAF ALAN,# 4 will jump to ALAN if ACC $\neq$ 0.

Note JAT, JAF do not use CONDITION CODE.

and interrupt occurs unless the condition is masked; otherwise OV is cleared.
If the operand form uses B, the jump location is undefined.

CC : Unaltered

Program errors : Operand addressing errors for jump instruction B overflow (unless masked) (see 12.1).

6.6 Out (OUT) #3C

Operand length : 32 bits.

Description : This instruction causes a class 9 interrupt to occur. The operand is left as the 32 bit interrupt parameter on the new stack. ACC, B, and (unless operand access is indirect) DR are unaltered.

CC : (Dumped value) unaltered.

Program errors : Operand addressing errors.

6.7 Idle (IDLE) # 4E

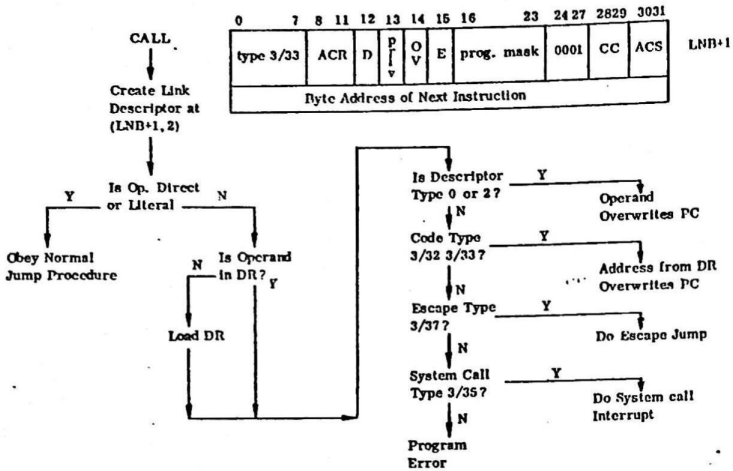
Operand length : Not applicable. Literal must be specified.

Description : This instruction causes instruction sequencing to be suspended until an interrupt (of any class) occurs.
The value of PC dumped on interrupt points to the next instruction in sequence. The instruction makes no reference to store or registers if the operand is a literal.

CC : Unaltered

Program errors : Only universal types listed in section 12.10..

Fig 29 CALL (CALL) (# 1E)



6.8 Call (CALL) # 1E (see fig 29)

Operand length : 32 bits.

Note: If the operand is addressed indirectly via a type 0 or type 2 descriptor, the addressed item in store may be 32 or 64 bits long. In either case it is treated simply as an instruction address, not a descriptor (so neither Call nor Escape mechanisms can be invoked) and, if 64 bits long, its more significant bits are ignored. If the operand is addressed indirectly via a code descriptor, the address in bits 32-63 of the code descriptor is the operand. If the code descriptor is unbounded a microcode routine may be entered when the jump is made.

Description : This instruction is used to enter procedures. A link descriptor specifying the location to return to on exit is generated and loaded into (LNB+1) and (LNB+2). The operand increments or overwrites PC causing a jump to occur. If SF ⩽ LNB + 2, the link descriptor is not stored and the instruction terminates with a program error interrupt.

The link descriptor is of unbounded Code Type and consists of:-

In (LNB + 2);

The byte address of the next instruction (i.e. the length of the Call instruction added to the contents of PC, with a zero bit concatenated at the less significant end).

In (LNB + 1);

Bits 0 - 7	: 11100001 (Type 3, sub-type 33)
Bits 8 - 11	: ACR
Bit 12	: D
Bit 13	: PRIV
Bit 14	: OV
Bit 15	: E
Bits 16-23	: Program mask
Bits 24-26	: Zero
Bit 27	: 1
Bits 28, 29	: CC
Bits 30, 31	: ACS

If the address form is indirect the descriptor, which is left in DR, may be one of the following types:

a) 0 or 2 : no special action

b) Code : the address in the descriptor itself, possibly modified, overwrites PC.

c) System Call : An interrupt is performed. If the descriptor is modified, the modifier is accessed but no modification takes place (eg, if the modifier is TOS, SF will be decremented).

d) Escape : An escape action is performed (LNB+1, +2) will be undefined.

If the operand is accessed directly from (LNB +2) or via a descriptor in (LNB+1) the result is undefined. If the operand is accessed indirectly, system software may intervene to decode a system call descriptor and in this case the contents of ACC, B and XNB must be regarded as undefined.

CC : Unaltered.

Program errors : Operand addressing errors for Call instruction. (see 12.6/10.2)

SF $\leqslant$ LNB + 2 (see 12.8/12.5).

NOTE: See section 4.8 for description of Precall Instruction.

6.9 Exit (EXIT) # 38 (see fig 30)

Operand length : 32 bits

Description : This instruction is used to return from procedures and after non-stack-switching interrupts. The stack is returned to its status quo and a jump is made as specified by the link descriptor. Fields of the link descriptor may be used to overwrite parts of PSR as specified by bits in the operand.

The link descriptor is extracted from (LNB+1, LNB+2). It may only be of types Code, System Call, or Escape. If SF $\leq$ LNB + 2, or the descriptor is not one of these types, the instruction terminates with a program error interrupt. If the link descriptor is System Call, PSR and PC are copied into DR, in the form of a link descriptor, and the system call interrupt exception condition routine is entered.

If the link descriptor is Code, DR is not altered but various fields in PSR are altered as follows:

- if the value of bits 8-11 of (LNB+1) is not less than ACR, bits 8-11 of (LNB+1) overwrite ACR; else program error interrupt.
- if the value of bit 13 of (LNB + 1) is not greater than PRIV, bit 13 of (LNB+1) overwrites PRIV; else program error interrupt.
-if operand bit 25 = 1, bits 16-23 of (LNB+1) overwrite PM.
-if operand bit 26 = 1, bits 28-29 of (LNB+1) overwrite CC.
-if operand bit 27 = 1, bits 30-31 of (LNB+1) overwrite ACS (program error interrupt if attempt is made to set ACS = 1).

6.9
contd.

- if operand bit 28 = 1, bit 12 of (LNB + 1) overwrites D.
- if operand bit 29 = 1, bit 14 of (LNB + 1) overwrites OV (this does not cause an overflow interrupt).
- bit 15 of (LNB +1) overwrites E

Other operand bits are ignored (reserved). The operand may only be a 7-bit literal. To restore the stack status quo, the contents of the LNB register are transferred to SF, and provided that bits 0-13 = SSN and bits 14-29 < SF, bits 14-29 of the contents of the locations now pointed at by SF are transferred to LNB. If these conditions are not satisfied LNB is unaltered.
The address from the code descriptor (ex-(LNB + 2)) overwrites PC.
Finally bit 31 of LNB as stacked (i.e. SF) is tested to Control the 'collapse' of stack, and if the bit 31 is one, a jump is made to the address in PC.
In emulating machines, if E=1 and EM has a locally valid value, emulate alien code.
If the new ACR is larger than the previous value, and the EP bit is set in SSR, an EP interrupt will occur, the next instruction is executed, unless masked.

When EXIT is used to return from procedures, system software may intervene to decode a system call descriptor and in this case the contents of DR and XNB must be regarded as undefined.

CC : Unaltered if operand bit 26 = 0. If operand bit 26 = 1, and the link descriptor is of Code type,

Fig 30 Exit (EXIT) # 38

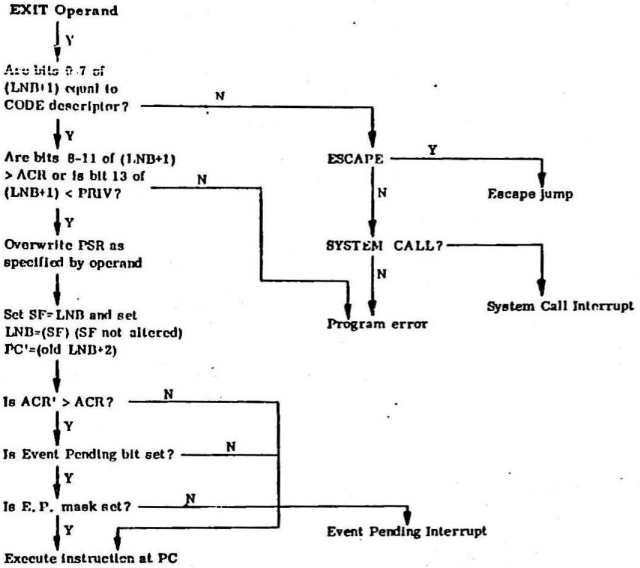
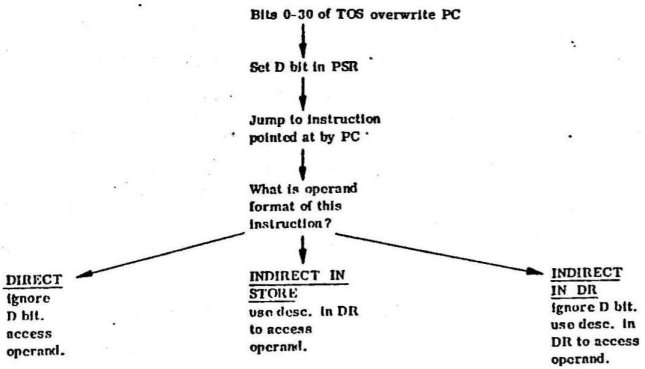


Fig 31 Escape Exit (ESEX) # 3A



6.9 contd.

CC takes value specified in bits 28, 29 of (LNB + 1).

Program errors : Incorrect operand type (must be 7-bit literal) (see 12.8/12.2).

SF ⩽ LNB + 2 (see 12.8/12.5)

Link descriptor not Code, System Call or Escape (see 12.6/10.3)

New PRIV > old PRIV
New ACR < old ACR
(see 12.5/9.5)

Bits 0-13 of (LNB) ≠ SSN (see 12.4/8.2)

Bits 14-29 of (LNB) > new SF (see 12.4/8.3)

Attempt to set ACS = 0 (see 12.9/13.7)

(Emulating machines; PEI 14) New E = 1 and EM = 0 or invalid value

6.10 Escape Exit (ESEX) # 3A (see fig 31)

Operand length : Not applicable. Literal must be specified.

Description : The operand field is ignored. Bits 0-30 of the word at the top of the stack overwrite PC; bit 31 is ignored. SF is decremented by 1 word. The 'D' bit is set in PSR, and a jump is made to the instruction pointed at by PC. If that instruction accesses the store indirectly, via a descriptor located in the store, the effect of the D bit will be to prevent that descriptor from being used; instead the descriptor already in DR (assumed to have been placed there by the escape routine) will be used. If the instruction specifies modification it will take place. E.g. if the operand format is MIT, it will be interpreted as MD. If indirect access via DR, or direct access, is specified, the D bit of PSR is ignored. E.g. if the operand format is DT it will be interpreted literally. The D bit

6.10
contd.

of PSR is cleared by the instruction so that its effect is limited to the first instruction after Escape Exit. That instruction will usually be the one which originally triggered the escape mechanism, re-executed; note that in the first example above, TOS will have been accessed (to obtain the Escape descriptor) before the escape action, in the second example TOS is not accessed until after Escape exit since Escape descriptors are unmodified.

CC : Unaltered

Program errors : Only universal types listed in section 12.10.

7 B INSTRUCTIONS

This chapter deals with instructions which affect the B register. This is a 32 bit register, thus all instructions must have a 32-bit operand.

7.1 Load B (LB) # 7A

Operand length : 32 bits

Description : The operand is loaded to B. OV is cleared. The previous contents of B may be used as a modifier in fetching the operand.

CC : Unaltered

Program errors : Operand addressing errors

7.2 Stack and Load B (SLB) # 52

Operand length : 32 bits

Description : The contents of B are copied to an intermediate register, and the operand is loaded to B. OV is cleared. The contents of the intermediate register are stacked, causing SF to be incremented by 1 word.
The intermediate register ensures that the operand forms T, DT, IT and MIT are valid.
The previous contents of B may be used as a modifier in fetching this operand.

CC : Unaltered;

Program errors : Operand addressing errors.

7.3 Store B (STB) # 5A

Operand length : 32 bits

Description : The contents of B are stored in the operand location; they may be used as a modifier in accessing the latter. B is unaltered.

CC : Unaltered.

Program errors : Operand addressing errors
Literal operand (see 12.8/12.1)
Non zero bits of stored item truncated (see 12.3/6.0).

NOTE: The effect on OV if B is the operand of ST B is undefined.

7.4 Add to B (ADB) #20
Subtract from B (SBB)#22
Multiply B (MYB)#2A

Operand length : 32 bits

Description : The arithmetic operation indicated is performed between the operand and the contents of B (which may be used as a modifier in accessing the operand). Both are treated as signed 32-bit integers. The least significant 32 bits of the resulting sum, difference or product is left in B.
If overflow occurs, i.e. the sum, difference or product is less than -2^{31} or greater than $2^{32}-1$, OV is set; otherwise OV is cleared. Overflow will also cause interrupt to occur if not masked.

CC : Unaltered.

Program errors : Operand addressing errors. B overflow (unless masked) (see 12.1).

7.5 Compare B (CPB)#26

Operand length : 32 bits

Description : The contents of B are compared with the operand, both being regarded as signed integers. The result of the comparison is indicated in CC. B and OV are unaltered.
Comparisons are performed exactly as for 32-

bit fixed point compare (9.14).

The contents of B may be used as a modifier in accessing the operand.

CC : 0 B = operand
1 B < operand
2 B > operand
3 Not used

Program errors : Operand addressing errors.

7.6 Compare and Increment B (CPIB) # 2E

Operand length : 32 bits

Description : The action of the instruction is identical to that of Compare B (7.5), with the addition that after the comparison 1 is added to the contents of B. OV is set if this causes B to overflow (i.e. if the contents of B go from $2^{31}-1$ to -2^{31}); interrupt will occur if this condition is not masked. OV is cleared if overflow does not occur.

The original contents of B may be used as a modifier in accessing the operand.

CC : 0, B (original contents) = operand
1, B (original contents) < operand
2, B (original contents) > operand
3, Not Used

Program errors : Operand addressing errors. B overflow (unless masked) (see 12.1).

7.7 Dope Vectors

The final instruction in this chapter is DOPE VECTOR MULTIPLY. However, before this is tackled, it would be opportune to discuss arrays. Imagine we have a series of locations, say JOE (0-9) and wish to operate on them successively then the B register instructions are ideal. The register can act as a modifier for accessing these

locations. For a single dimensional array as in fig 32, this is easy enough. To access ED1 we use a modifier of one, etc.

For two-dimensional arrays the value of the modifier is obtained by adding the first subscript to the product of the second subscript and the number of elements in the first dimension.

So for F12 in the array F (0 : 3, 0 : 2), the modifier required is 1 + (2 x 4) = 9. Checking fig 32 you will see that a modifier of 9 will enable us to access F12.

7.8 3D Arrays

The process becomes more complicated for 3-D arrays. Fig 32 also demonstrates the principles involved. A worked example is shown, using the formula given in the diagram - the modifier needed to access element I211 is 18. If you check the array you will find that this is quite correct.

The formula given becomes more complicated to use when there are arrays whose lower bounds are not zero. Dope vectors considerably simplify the calculation of modifiers in this situation.

Consider the array Z (5-9) which is shown in fig 33. For each dimension of the array, there is a dope vector - this is a set of 3 words, i.e. three 32-bit fields x, y, z, which give information about the array. There will be a descriptor pointing to the start of the array - calculation of the modifier will enable the correct element to be accessed.

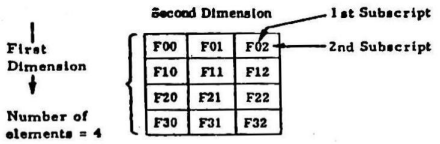
SINGLE DIMENSIONAL ARRAY

ED (0:4)

ED0	ED1	ED2	ED3	ED4

TWO DIMENSIONAL ARRAY

F (0:3, 0:2)



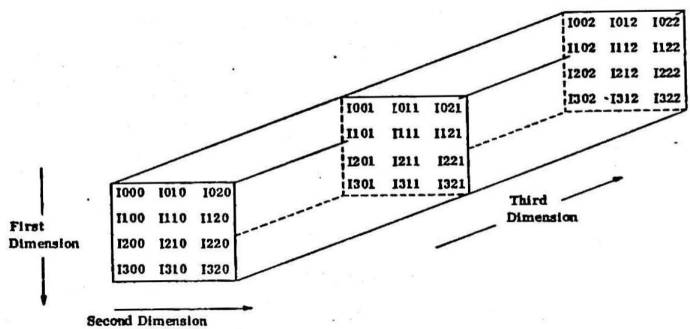
F00	F10	F20	F30	F01	F11	F21	F31	F02	F12	F22	F32

Modifier value M = first subscript + (second subscript x number of elements in first dimension)

For F12: M = 1 + (2 x 4) = 9.

THREE DIMENSIONAL ARRAY

I (0:3, 0:2, 0:2)



I000	I100	I200	I300	I010	I110	I210	I310	I020	I120	I220	I320	I001	I101	I201	I301	I011	I111	I211	I311	I021
I121	I221	I321	I002	I102	I202	I302	I012	I112	I212	I312	I022	I122	I222	I322						

M = first subscript + (second subscript x number of elements in first dimension) + (third subscript x number of elements in second dimension x number of elements in first dimension)

For I211 M = 2 + (1 x 4) + (1 x 3 x 4) = 2 + 4 + 12 = 18.

FIG 32 ARRAYS

7.8
Cont. x is the lower subscript limit for that dimension. y is the number of elements in the previous dimensions. Note that for a single dimensional array or for the first dimension of a multi-dimensional array, y = 1, z is the number of elements up to the end of that dimension, including previous dimensions.
Incidentally for a multi-dimensional array the value of z for a particular dimension is of course, the same as the value of y for the next dimension.

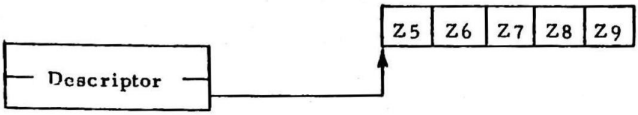
7.9 Dope Vector Multiply (VMY) # 2C (see fig 34)

Operand length : 32 bits

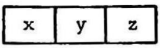
Description : DR must contain a type 0 descriptor, with size code 32 bits and USC and BCI = 0 (or an interrupt occurs).
The contents of the word pointed at by this descriptor and of the next two words are referred to below as x, y and z.
The action of the instruction is to evaluate the expression (i-x)y, where (i = operand) whose l.s. 32 bits are left in B. As each of x, y and z are accessed the address in DR is incremented by 4 bytes and the bound decreased by 1 (Bound Check interrupt occurs if this changes the bound field contents from 0 to all 1's). Thus at the end of the instruction the address will have been increased by 12 bytes and the bound decreased by 3. Indirect addressing forms are not permitted. Interrupts occur if any of the following conditions are not satisfied (unless Bound Check is masked).

$0 \leq (i - x) < 2^{31}$ (i and x signed integers)
$0 \leq (i - x)\, y < z$
OV is cleared.

Fig 33 DOPE VECTORS



DOPE VECTORS are sets of three words which hold information about ARRAYS.



x = lower subscript limit for that dimension.

y = number of elements in previous dimensions (for a 1-D array or the first dimension of a multi-dimensional array, y = 1).

z = number of elements up to the end of that dimension (including previous dimensions).

In the above example, x = 5

y = 1

z = 5

Fig 34 DOPE VECTOR MULTIPLY (VMY)

The operation of this instruction is to calculate (i-x)y and check that the result is less than z, where x, y, z are the Dope vectors. The result is left in the B register.

To calculate the modifier required for Z8 in Fig 33, use the format:-

VMY 8

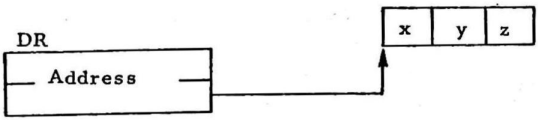
where i = operand = 8.

The Dope Vectors are x=5, y=1, and z=5.

Therefore (i-x)y = (8-5). 1=3.

This is the result left in the B register.

The result = 3 is less than z which is equal to 5.



7.9
Contd.

CC	:	Unaltered.	
Program errors	:	Operand addressing errors Indirect address form (see 12.8/12.2) Incorrect type and size code of descriptor in DR (see 12.6/10.4).	
$i < x$		unless)	(see 12.2/5.3)
$i - x \geq 2^{31}$		bound)	(see 12.2/5.2)
$y < 0$		check is)	(see 12.2/5.4)
$z < 0$		masked)	(see 12.2/5.5)
$(i - x)\ y \geq z$ (includes $(i-x)\ y \geq 2^{31}$)		)	(see 12.2/5.6)
Bound check on x, y or z		)	(see 12.2/5.7)

Example 1

Say we wish to calculate the modifier to access z8 in the array z (5-9).

This can be done using the Dope Vector Multiply instructions as follows:-

VMY 8

where the operand i = 8.

Thus (i - x) y = (8 - 5). 1 = 3, which is left in the B register. The result is checked to be less than z (=5).

We can see that a modifier of three will enable us to access Z8.

The descriptor register must contain a type 0 descriptor with size code 32-bits - the address field of the descriptor points at x. This information is summarised in fig. 34.

The modifier required for a multi-dimensional array is the sum of the individual modifier results.

i.e. for a 3-D array the modifier is:-

$$(i_1 - x_1)\ y_1 + (i_2 - x_2)\ y_2 + (i_3 - x_3)\ y_3$$

Example 2

Fig 35 shows another worked example - here the modifier is calculated to access FRED 36 in the array FRED (2:3, 5:6). First of all the dope vectors for the array have to be evaluated.

For the first dimension x = 2, y = 1, z = 2. Moving on to the second dimension x = 5 (being the lower limit for that dimension), y = 2 (the number of elements in the previous dimension), z = 4 (total number of elements in the array).

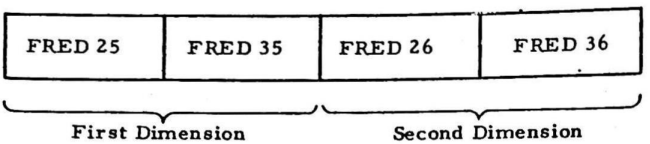
A descriptor pointing to x_1 (x for the first dimension) is loaded into DR. A dope vector multiply using an operand of 3 calculates the modifier for the first dimension. The result, which is 1, is left by the instruction in the B-register - this is copied to the top of stack. Now the procedure is repeated for the second dimension; a dope vector multiply using the operand of 6 leaves the result of 2 in the B register. The contents of TOS, containing the modifier for the first dimension, are added to B to give the final modifier required.

Example 3

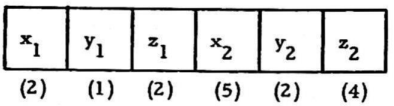
Fig 36 illustrates an example of VMY used to calculate the modifier in a 3-D array. The element required is JOE (5, -1, 10) in the array JOE (3:9, -2:10, -20:20). The dope vectors are calculated and DR is loaded with the descriptor pointing to x_1. Three dope vector multiply operations are carried out, results of which are stacked and finally added together to produce the modifier.

Fig 35 DOPE VECTOR EVALUATION

Calculate the modifier required to access FRED 36 in the array FRED (2:3, 5:6).



The Dope Vector for the above example is:-



To calculate M for FRED 36:-

LD operand pointing to x_1

VMY 3	B' = (3-2). 1 = 1	Checks that B' < z1 (first dimension)
STB TOS	B' = 1, TOS = 1	
VMY 6	B' = (6-5). 2 = 2	Checks that B' < z2 (second dimension)
ADB TOS	B' = 3 = modifier	

Fig 36 VECTOR MULTIPLY FOR A 3-D ARRAY

Array JOE (3:9, -2:10, -20:20)

Find modifier to access JOE (5, -1, 10)

$x_1 = 3$; $y_1 = 1$; $z_1 = 7$

$x_2 = -2$; $y_2 = 7$; $z_2 = 7 \times 13 = 91$

$x_3 = -20$; $y_3 = 91$; $z_3 = 7 \times 13 \times 41 = 3731$

LD operand		
VMY 5	B' = (5-3). 1 = 2	Checks that B' < 7
STB T	B' = 2, TOS = 2	
VMY -1	B' = (-1 + 2). 7 = 7	Checks that B' < 91
STB T	B' = 7, TOS = 7	
VMY 10	B' = (10 + 20). 91 = 2730	Checks that B' < 3731
ADB T	B' = 2737, TOS = 2	
ADB T	B' = 2739 = modifier	

8. DR INSTRUCTIONS.

8.1 Load DR (LD) # 78

Operand length : 64 bits

Description : The operand is transferred to DR. If it is accessed indirectly the operand, rather than the descriptor used to access it, is left in DR. CC is set to indicate the type of descriptor loaded.

CC :
0 Type 0 descriptor loaded
1 Type 1 descriptor loaded
2 Type 2 descriptor loaded
3 Type 3 descriptor loaded

Program errors : Operand addressing errors.

Fig 37 shows two examples of an LD; the first with a directly accessed operand, the second with an indirectly accessed operand.

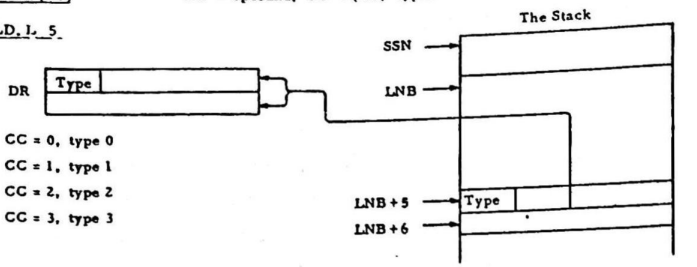
8.2 Load Relative (LDRL) #70 (see fig 38)

Operand length : 64 bits

Description : The operand is transferred to DR, with its least significant 32 bits (the address field) augmented by the value of its own byte address. Carry out of the address field resulting from this addition is ignored. Thus, if the operand (the item in store) starts in byte x and is a descriptor pointing at location y, that descriptor is left in DR with its address adjusted to point at location (x+y) x need not be a multiple of 4.

Fig 37 LOAD DR (LD) DR' = operand, CC' = (DR)' type.

i) LD, I, 5



ii) LD, IL 5

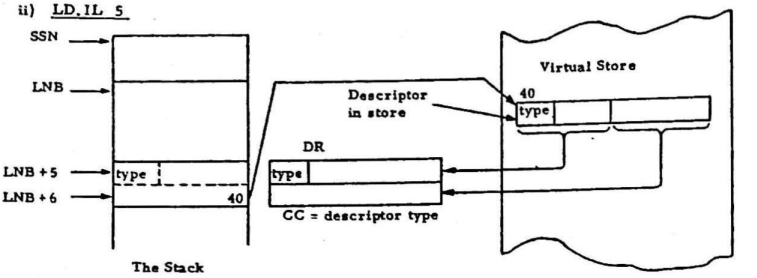
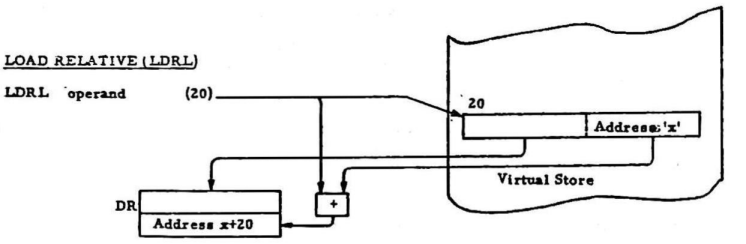
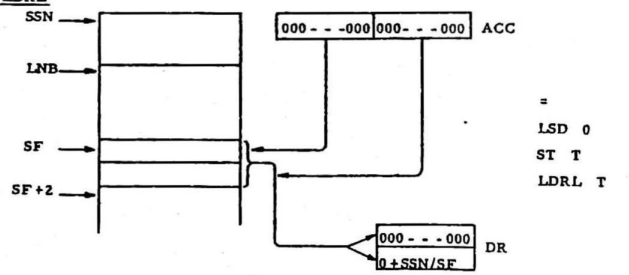


Fig 38 LOAD RELATIVE (LDRL)



Use of LDRL



=
LSD 0
ST T
LDRL T

If accessed indirectly, the operand, rather than the descriptor used to access it, is left in DR. Literal operands and operands in registers are not permitted. CC is set to indicate the type of descriptor loaded.

CC : 0 Type 0 descriptor loaded
1 Type 1 descriptor loaded
2 Type 2 descriptor loaded
3 Type 3 descriptor loaded

Program errors : Operand addressing errors
Invalid address forms (see 12.8/12.2)

8.3 Stack and Load DR (SLD) # 50

Operand length : 64 bits

Description : The contents of DR are copied to an intermediate register. The operand is loaded to DR, and the contents of the intermediate register are stacked, causing SF to be incremented by 2 words. The intermediate register ensures that the operand forms T, DT, IT and MIT are valid.

If accessed indirectly, the operand, rather than the descriptor used to load it, is left in DR. CC is set to indicate the type of descriptor loaded.

CC : 0 Type 0 descriptor loaded
1 Type 1 descriptor loaded
2 Type 2 descriptor loaded
3 Type 3 descriptor loaded

Program errors : Operand addressing errors.

8.4 Store DR (STD) # 58

Operand length : 64 bits

Description : The contents of DR are stored in the operand location. Indirect address forms are not permitted.

CC : Unaltered.

Program errors : Operand addressing errors
Literal operand (see 12.8/12.1)
Significant part of operand truncated. (see 12.3/6.0)
Indirect address form (see 12.8/12.2)

8.5 Load Address (LDA) # 72

Operand length : 32 bits

Description : The operand is loaded to the less significant 32 bits of DR. The more significant 32 bits are unaltered unless an indirect form is used, in which case they will be replaced by the corresponding bits of the descriptor used to access the operand.

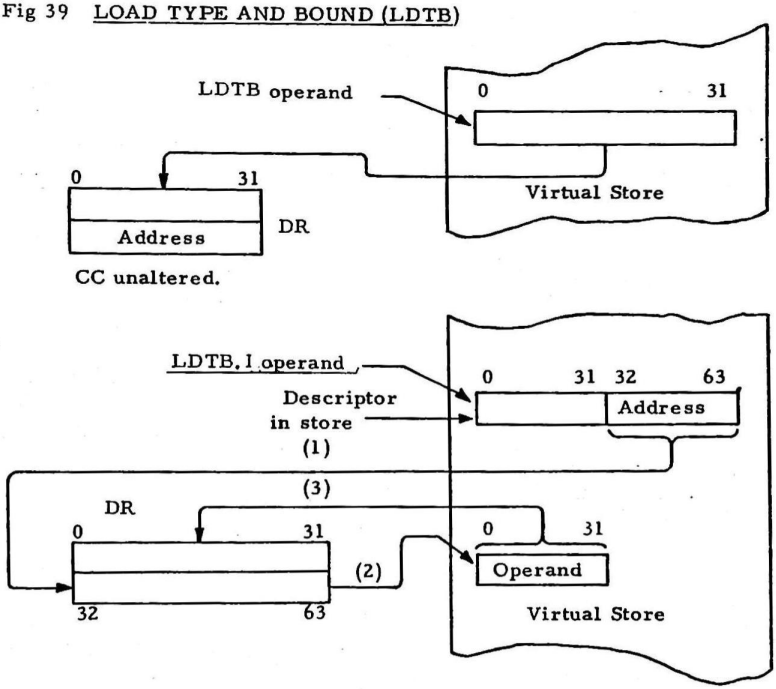
CC : Unaltered

Program errors : Operand addressing errors.

8.6 Load Type and Bound (LDTB) # 74 (see fig 39)

Operand length	:	32 bits
Description	:	The operand is loaded to the more significant 32 bits of DR. The less significant 32 bits are unaltered unless an indirect address form is used, in which case they will be replaced by the address (unmodified) from the descriptor used to access the operand.
CC	:	Unaltered.
Program errors	:	Operand addressing errors.

Fig 39 LOAD TYPE AND BOUND (LDTB)



8.7 Load Bound (LDB) (see fig 40)

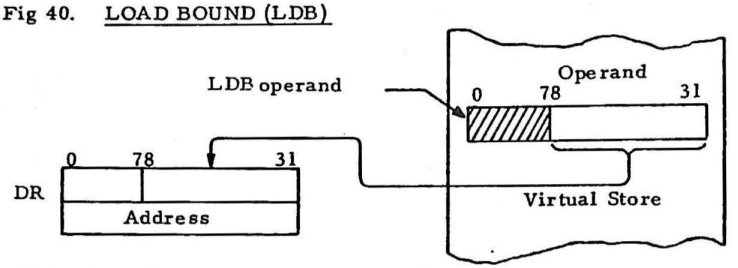
Operand length : 32 bits

Description : The least significant 24 bits of the operand are loaded to bits 8-31 of DR. The remaining bits of DR are unaltered unless an indirect address form is used, in which case they will be replaced by the corresponding bits (address field unmodified) of the descriptor used to access the operand. Bits 0-7 of the operand are ignored.

CC : Unaltered.

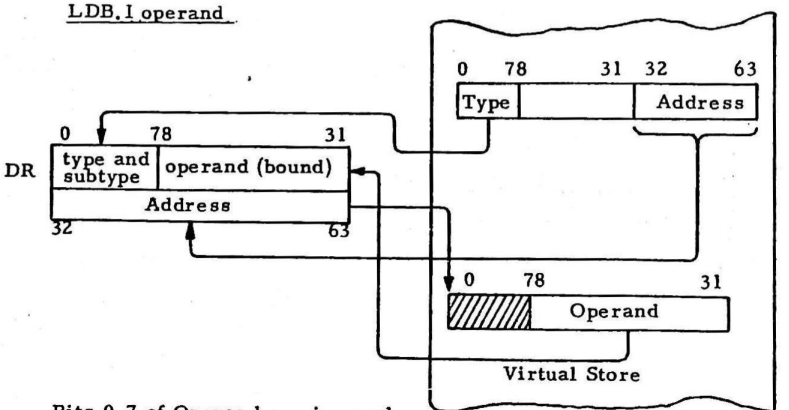
Program errors : Operand addressing errors.

Fig 40. LOAD BOUND (LDB)



CC unaltered.

LDB.I operand



Bits 0-7 of Operand are ignored.

8.8 Modify DR (MODD) #16 (see fig 41)

Operand length : 32 bits

Description : If DR contains a Vector, String or Code descriptor (ie type 0 with valid size code, type 1, type 2 or type 3; subtypes 32 or 33), the operand is added to the address field of DR and subtracted from the bound/length field, carry out of the address field is ignored and bits 0-7 of DR are unaltered (see Notes below).

If DR contains an Escape descriptor, the escape mechanism is invoked (so that the required descriptor may be substituted in DR before being modified).

If the descriptor in DR is type 0 with an invalid code, or System Call, or type 3 with an undefined subtype number, a program error interrupt occurs.
Indirect address forms are not permitted.

Notes: a) If the descriptor is type 0 or 2 and USC is not set, or type 3 subtype 32 or 33, the operand is scaled appropriately before addition to the address field. If the descriptor is type 0 with size code 0, the least significant 3 bits of the operand are ignored because of the scaling operation.

b) If the operand, regarded as unsigned i.e. positive, is not less than the original contents of the bound/length field, only the least significant 24 bits of the difference are left in that field. In such

cases, if the descriptor is type 0 or 2 with BCI not set, or type 3 subtype 32 (bounded code) a program error condition (Bound Check, maskable) is generated. This does not apply to String descriptors.

CC : Unaltered.

Program errors : Operand address errors
Indirect address form (see 12.8/12.2)
Bound significant, and≤operand (see 12.2/5.1)
Descriptor is System Call (see 12.6/10.11)
Descriptor is invalid (see 12.6/10.9 or 10)

8.9 Increment Address (INCA) #14

Operand length : 32 bits

Description : The operand is added to bits 32-63 of DR. Bits 0-31 of DR are unaltered. Indirect address forms are not permitted.

CC : Unaltered.

Program errors : Operand addressing errors
Indirect address form (see 12.8/12.2)

Fig. 41 MODIFY DR (MODD)

MODD operand

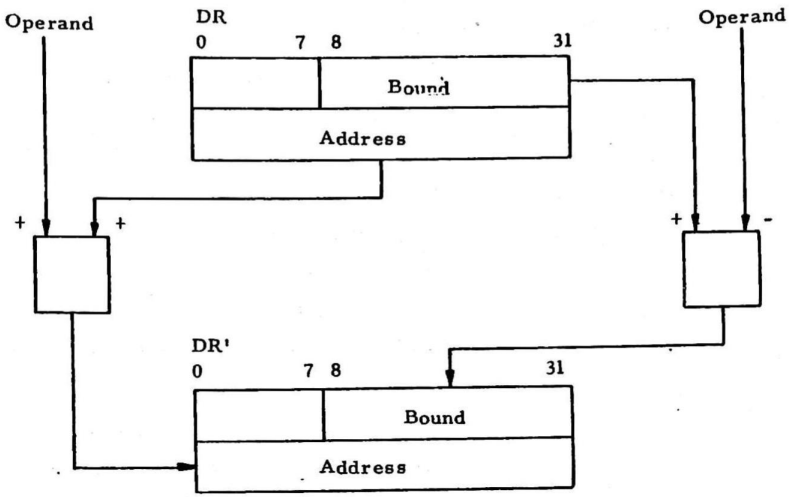
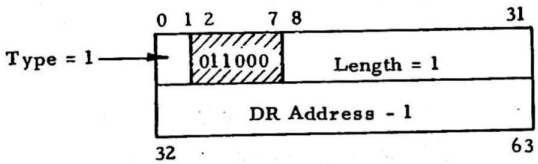


Fig. 42 START SIGNIFICANCE (SIG) (#28)

If CC = 0, Create the following descriptor and load into operand location, then set CC = 1



If CC ≠ 0 no action

8.10 Start Significance (SIG) # 28 (see fig 42)

Operand length : 64 bits

Description : If CC = 0, a descriptor is created and stored in the operand location, and CC is set to 1. The descriptor is made up as follows:

Bits 0, 1 (type) = 1
Bits 2-7 = 011000
Bits 8-31 (length) = 1
Bits 32-63 (address) = 1 less than contents of address field.

If CC ≠ 0, no action is performed.
Indirect address forms are not permitted.

Note: This instruction is designed for use in conjunction with 'Suppress and Unpack' (10.12).

CC :
0 Not Used
1 CC originally 0, and descriptor stored or CC originally 1.
2 CC originally 2.
3 CC originally 3.

Program errors : Operand addressing errors
Literal operand (see 12.8/12.1)
Indirect operand form (see 12.8/12.2)
Significant part of operand truncated (see 12.3/6.0)

8.11 Validate Address (VAL) #10

Operand length : 32 bits

Description (AML0) : This instruction is designed to investigate whether a descriptor provided to a called procedure is valid at the access level of the caller. The descriptor is assumed to be in DR, and to be Type 0, Type 1 or Type 2. If type 0 or 2 the descriptor is assumed to be bounded. Bits 8-11 of the operand are interpreted as the access control key (normally held in ACR) of the caller; the remaining operand bits are ignored. If the descriptor in DR is 'invalid', condition code 3 is set, and the instruction terminates. Invalidity includes any of the following:

a) Descriptor is type 0 or 2, and BCI is set
b) Descriptor is type 3
c) Descriptor is type 0 and has invalid size code
d) Descriptor (type 0,1 or 2) has zero in bound/length field.

If the descriptor is valid, the address of the last word or byte in the field pointed at by the descriptor is calculated (without altering DR). This address is calculated from the address of the first byte as follows:

Type 0: Add (Bound-1) (scaled if USC = 0), then if size = 64 bits add 4 bytes, if size = 128 bits add 12 bytes (word alignment assumed).

Type 1: Add-(length-1)

Type 2: As for type 0, with size = 64 bits.

If the address thus calculated has a different segment number from the initial address, or if it has the same segment number but lies beyond the upper limit of that segment, or if it has the same segment number as SSN but is not less than SSN + LNB, CC is set to 3 (in the last case, if it is less than SSN + LNB, and the initial address is also in the stack segment, CC is set to 0). Otherwise CC is set to indicate whether read or write access to that segment, at the access level given by the operand, is permitted.

If access is not permitted CC is set to 3. The second word of the segment table entry is ignored. Indirect address forms are not permitted. DR is unaltered. ACR is unaltered.

CC :

0 Read and write access permitted at specified level

1 Read access permitted, write inhibited

2 Read access inhibited, write permitted

3 Descriptor invalid, or field crosses segment boundary, or neither read nor write permitted (includes case of invalid segment number).

Program errors : Operand addressing errors

Indirect address form (see 12.8/12.2)

Description (AML1) : The descriptor is taken from DR and must be type 0, 1 or 2 or type 3; subtype 63. If type 0 or 2,the descriptor must be bounded. Bits 8-11 of the operand are interpreted as the access control key (normally held in ACR) of the caller; the remaining bits are ignored.

The descriptor is first checked for 'Type Null' which is defined to be Type 3, subtype 63, and if so CC is set to 2 and the instruction terminates.

The descriptor is then checked for validity. If 'invalid', CC is set to 3 and the instruction terminates.
'Invalid' includes any of the following:

(a) Type 0 or 2 and BCI set
(b) Type 3, subtype 63
(c) Type 0 and invalid size code 1, 2 or 4
(d) Bound / length field is zero
(e) Field invalid, i.e. all or part of the field does not exist.

Implementation of check (e):

The address of the last byte or word in the field pointed at by the descriptor is calculated (without altering DR) from the address of the first byte as follows:
Type 0 or 2 - Add (bound - 1), scaled if USC = 0, then add 4 bytes if size code 6, add 12 bytes if size code 7 (word alignment assumed).

Type 1 - Add (length - 1)

The address in the descriptor points at an initial segment, the calculated address points at a final segment and there may be 'intermediate' segments in-between.

The end address is checked to be less than the segment limit of the final segment or less than the SSN + LNB if initial segment is the same as SSN. If the final segment is different from the initial segment then the initial and intermediate segments are checked that they are of maximum size (bits 14-24 of STE are all one's)

Finally, the access keys of the initial, intermediate and final segments are checked against the access level given in the operand. If the addressed segment is the same as SSN, CC is set to 0.
If read access is not permitted in any segment, the CC is set to 3. If write access is not permitted in any segment then CC is set to 1. Otherwise CC is set to 0.
Indirect forms are not permitted. DR and ACR are unaltered.

CC : 0 - Read and write access permitted at specified level

1 - Read access permitted, write access inhibited.

2 - Descriptor is type 3, subtype 63 (Null)

3 - Descriptor invalid OR read access inhibited.

Program Errors : As for AML0

9 COMPUTATIONAL FUNCTIONS

The computational functions perform arithmetic operations on the accumulator. Instructions are provided which can deal with floating-point, fixed-point, logical and decimal data. Some functions e.g. add, subtract will perform essentially the same operation and differ only in the way they interpret the data format.

9.1 Summary of Computational Instructions

Fig 43 summarises the computational functions available on 2900. Floating point operations are preceded by the letter R (real), fixed-point by I (integer), decimal by D and logical by U (unsigned).

All the instructions have primary operand formats. With most of the instructions, the operand interacts with the contents of ACC and the result is left in ACC. A few functions are provided for converting data from one format to another. Note that fixed-point and logical operations with ACS = 128 are not permitted.

9.2 Floating Point Data Format

The format for a 32-bit, 64-bit or 128-bit floating-point number is shown in fig 44. A 32-bit number uses the most significant 32 bits and a 64-bit number the most significant 64 bits. A 128-bit number uses bits 0-127 excluding bits 64-71, i.e. bit 72 is a continuation from bit 63.

A 2900 floating-point number can be represented in so-called sign and modulus format:

$SF \times 16^{(c-64)}$

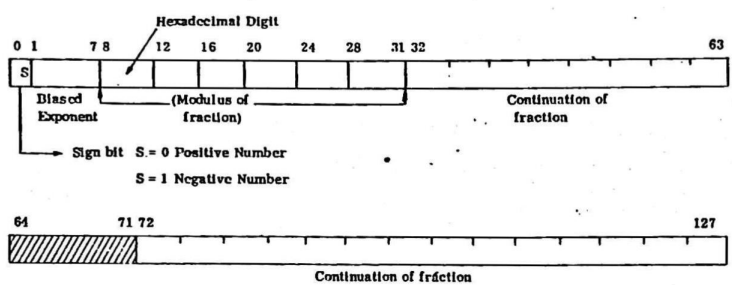
where S is the sign bit for the whole number. It is zero for a positive number, a one for a negative number.

F is the unsigned fractional part of the number. The binary point of this fraction lies to the left of bit 8, so that F ranges in value from 0

Fig 43 SUMMARY OF COMPUTATIONAL FUNCTIONS

Second Digit.	First Digit. 8	9	A	B	C	D	E	F
0					UAD	DAD	IAD	RAD
2					USB	DSB	ISB	RSB
4					URSB	DRSB	IRSB	RRSB
6					UCP	DCP	ICP	RCP
8			FLT	FIX	USH	DSH	ISH	RSC
A	AND	DDV	IDV	RDV	ROT	DMY	IMY	RMY
C	OR	DRDV	IRDV	RRDV	SHS	DMYD	IMYD	RMYD
E	NEQ	DMDV	IMDV	RDVD	SHZ	CBIN	CDEC	-

Fig 44 FLOATING POINT FORMAT



Bits 1 - 7 contain biased exponent, C, in the range 0 - 127; this represents a true exponents biased by + 64.

Number = SF x $16^{(C-64)}$

to just less than 1. F is expressed as a series of hexadecimal digits. Thus it has 6, 14 or 28 hex. digits depending on whether the format is 32, 64 or 128 bits.

C is characteristic - or biased exponent - in fact, it is a true hexadecimal exponent biased by +64. That is the reason why the characteristic has to be corrected by subtracting 64.

Bits 64-71 are ignored when reading data, however when producing results, bit 64 is made identical to bit 0, the sign bit and bits 65-71 contain the biased exponent value of bits 72-127, i.e. a value 14 less than that in bits 1-7. If the characteristic in bits 1-7 is less than 14, then a negative value would result in bits 65-71. In this situation, the difference is made positive by adding 128. Note for true zero, all 128 bits are made zero.

Unless otherwise stated, the results of all floating-point operations are normalised - this implies that the first hexadecimal digit of F is non-zero. A number is normalised by shifting F left by the appropriate number of hexadecimal places and subtracting this number from the characteristic.

The use of normalised operands is recommended since greater precision will be achieved than by the use of non-normalised numbers. However, it is perfectly acceptable to use non-normalised numbers.

Floating-point underflow occurs if a result cannot be expressed in normalised form with a true exponent greater than -65. In this situation the result is made true zero, and an interrupt will occur unless floating-point underflow is masked.

Floating-point overflow occurs when a normalised result requires a true exponent larger than +63. The overflow bit OV is set, and unless floating-point overflow is masked an interrupt will occur. For either case the result is given with a normalised fraction and characteristic 128 less than it should be.

Arithmetic results are truncated i.e. rounded towards zero. Precision is achieved in most cases by the use of 'intermediate fractions' which are allowed one more hexadecimal digit than appears in the result fraction. This is referred to as a 'guard digit'. When normalised operands are used a single guard digit is sufficient to ensure that the error in the result of any single arithmetic operation does not exceed unity in the least significant digit of the normalised result.

9.3 Floating Point Instructions

Floating Add (RAD) #F0
Floating Subtract (RSB) #F2

Operand length	:	ACS
Description	:	The operand is added to, or subtracted from, the contents of ACC, and the normalised result is left in ACC. ACS may be 32, 64 or 128 bits. Overflow or underflow may occur as described in 9.2. OV is cleared if overflow does not occur. Floating-point subtraction is performed by inverting the sign bit (only) of the operand and then following the rules for floating-point addition.
CC	:	Unaltered.
Program errors	:	Operand addressing errors Floating overflow (unless masked) (see 12.1) Floating underflow (unless masked) (see 12.1)

Floating-point addition is performed in three stages:- (see fig 45)

(1) The fractional part of the number with smaller characteristic is shifted down logically by the number of hexadecimal places which is the difference of the characteristics. The digit left in the position immediately to the right of that originally occupied by the least significant digit of the fraction is retained as 'guard digit' (and the unshifted fraction is extended with a zero in the corresponding position), but all the other digits shifted off are lost (effectively, treated as zeros).

If the characteristics were equal both fractions are extended with zero guard digits.

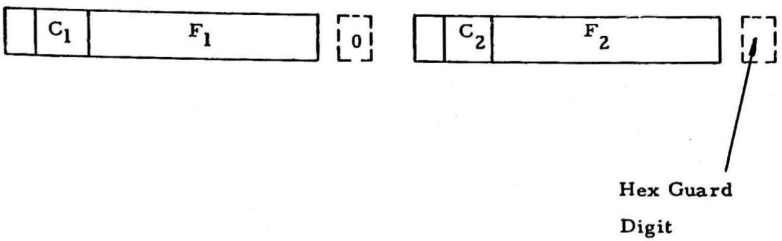
The above procedure is still carried out even if the number with larger characteristic has a zero fraction (this will not occur with normalised operands). In the case where the number with smaller characteristic has a zero fraction, shifting may however be suppressed (and the next stage omitted) without affecting the result.

(2) The two signed fractions, including their guard digits, are added algebraically to form an intermediate sum in sign-and-modulus form. The intermediate sum has an associated sign bit, a possible carry bit and, including the guard digit, 7, 15 or 29 hexadecimal digits.

(3) The intermediate sum is normalised to generate the final result. The characteristic initially associated with the intermediate sum is the larger of the two original characteristics. Normalisation proceeds as follows:-

- if all digits and the carry bit of the intermediate sum are zero, a true zero result is generated.
- if the carry bit is non-zero, the intermediate sum is shifted one hexadecimal place to the right (generating a 1 in the most significant hexadecimal digit position), and its carry bit and guard digit are removed, to form the fractional part of the result. 1 is added to the characteristic (this may cause overflow as described in 9.2) to form the result characteristic. The sign bit of the result is that associated with the intermediate sum.
- if the carry bit is zero, but one or more digits of the intermediate sum are non-zero, the latter is shifted left until the most significant hexadecimal digit is non-zero. Following each hexadecimal shift, zero is inserted in the guard digit position and 1 is subtracted from the characteristic. Should this cause the characteristic to become negative, underflow occurs as described in 9.2 (and a true zero result is generated). If the characteristic does not become negative, the result comprises the sign bit associated with the intermediate sum, the final characteristic and the normalised intermediate sum with the carry bit and guard digit removed.
- if ACS = 128 bits, bits 64-71 of the result are generated as described in 9.2.

Fig 45 RULES FOR FLOATING ADDITION



1. The fractional part with the smaller characteristic (C_2 in this case) is shifted right by (C_1 - C_2) hex. places, thus converting the numbers to the same exponent. Note the last digit shifted off the r.h. end is retained as a guard digit. The unshifted number is effectively extended on the right with a zero.

2. The two signed fractions are added together algebraically to form an intermediate sum in sign and modulus form. The intermediate sum has an associated sign bit, a possible carry bit, and including the guard digit, 7, 15 or 29 hex. digits.

3. The intermediate sum is normalised.

9.4 Floating Reverse Subtract (RRSB) # F4

Operand length : ACS

Description : This is exactly the same as Floating Subtract (9.3) except that the difference left in ACC is formed by subtracting the original contents of ACC from the operand. Effectively the sign bit (only) of the original contents of ACC is inverted and the operand is then added.

CC : Unaltered.

Program errors : Operand addressing errors
Floating overflow (unless masked) (see 12.1)
Floating underflow (unless masked) (see 12.1)

9.5 Floating Compare (RCP) # F6

Operand length : ACS

Description : The operand is compared algebraically with the contents of ACC, and CC is set to indicate the result of the comparison. ACC and OV are not altered. ACS may be 32, 64 or 128 bits. Overflow and underflow do not occur.

The comparison is performed effectively by carrying out the first two stages of the Floating Subtract operation (9.3) and examining the intermediate sum. Equality is indicated if and only if the carry bit and all digits (including the guard digit) of the latter are zero. Otherwise the setting of CC depends on the sign bit associated with the intermediate sum.

Note that, when the number with the larger

characteristic is not normalised, equality may be indicated according to the above rules even though the numbers are not equal in value; but that in cases where subtraction would produce a zero result because of underflow, inequality is always indicated.

CC : 0 Equality (Intermediate sum zero)
1 ACC < operand
2 ACC > operand
3 Not Used

Program errors : Operand addressing errors.

9.6 Scale (RSC) #F8

Operand length : 32 bits

Description : The signed fixed point integer (i) in the least significant 8 bits of the operand is added to the characteristic of the floating-point number in ACC, which is then normalised.
The contents of ACC are thus effectively multiplied by 16^i, where $-128 \leq i \leq +127$.
The remaining bits of the operand are ignored. ACS may be 32, 64 or 128 bits.

If the fractional part of the number is zero a true zero result is generated. If it is non-zero, and if after adding i and subtracting the amount of normalising shift, the characteristic exceeds 127, overflow occurs as described in 9.2. Similarly, if the fraction is non-zero, and after adding i, or subsequently after subtracting the amount of normalising shift, the characteristic

becomes a negative, underflow occurs and a true zero result is generated. OV is cleared if overflow does not occur.

If ACS = 128 bits, bits 64 - 71 of the result are generated as described in 9.2.

This instruction may be used with a zero operand to normalise any floating-point number.

CC : Unaltered.

Program errors : Operand addressing errors
Floating overflow (unless masked)(see 12.1)
Floating underflow (unless masked) (see 12.1)

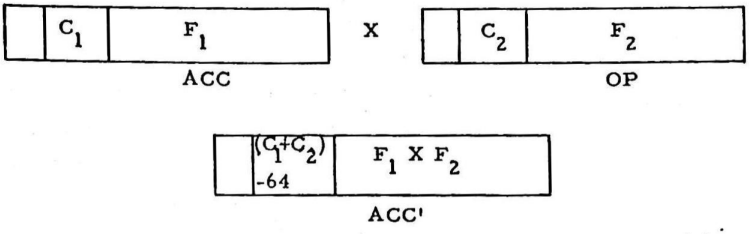
9.7 Floating Multiply (RMY) #FA (see fig 46)

Operand length : ACS

Description : The normalised product of the contents of ACC and of the operand is left in ACC. ACS may be 32, 64 or 128 bits.

If the fractional part of either multiplier is zero, a true zero result is generated, and neither overflow nor underflow can occur. If neither of the fractional parts is zero, the fractional part of the result is that which would be produced by forming the true, double-length, product of the fractions associating with it a characteristic which is the sum of the original characteristics, minus 64, normalising the product and then truncating it to half length. Overflow or underflow occur, as described in 9.2, if

Fig 46 FLOATING MULTIPLY (RMY)
FLOATING MULTIPLY DOUBLE (RMYD)



$C_1 = c_1 + 64$
$C_2 = c_2 + 64$
} where C_1 and C_2 are true exponents.

$(C_1 + C_2) - 64 = (c_1 + 64 + c_2 + 64) - 64 = (c_1 + c_2) + 64 = E + 64$

This is the biased exponent sum.

RMYD doubles original ACS

RMY leaves ACS unchanged and truncates the double-length answer to ACS length.

and only if the final characteristic exceeds 127 or is negative; in the latter case a true zero result is generated. When the result is not zero the sign bit is determined by the rules of algebra.

The desired result can be obtained by pre-normalising the multiplier and multiplicand fractions (ignoring overflow or underflow at this stage) and retaining for normalisation only the most significant 7, 15 or 29 hexa-decimal digits of their product; after normalisation, which in this case will involve at most one left shift, the guard digit is removed.

OV is cleared if overflow does not occur.
If ACS = 128 bits, bits 64 - 71 of the result are generated as described in 9.2.

CC : Unaltered.

Program errors : Operand addressing errors
Floating overflow (unless masked) (see 12.1)
Floating underflow (unless masked) (see 12.1)

9.8 Floating Multiply Double (RMYD) #FC

Operand length : ACS

Description : The contents of ACC are multiplied by the operand, each having the length specified by ACS, and their double-length normalised product is left in ACC. For this operation ACS must be 32 or 64 bits, and ACS is doubled on completion.
Except that the true product of the fractions

is not truncated (and the result is therefore exact) this operation is otherwise identical to Floating Multiply (9.7).

CC : Unaltered.

Program errors : Operand addressing errors
Floating overflow (unless masked) (see 12.1)
Floating underflow (unless masked) (see 12.1)
ACS = 128 bits (see 12.9/13.5)

9.9 Floating Divide (RDV) # BA (see fig 47)

Operand length : ACS

Description : The contents of ACC are divided by the operand, and the normalised quotient left in ACC. ACS may be 32, 64 or 128 bits. No remainder is preserved.

If the divisor fraction is zero, the result in ACC is undefined, but OV is cleared; the Zero Divide interrupt occurs, unless masked. If the divisor fraction is non-zero underflow or overflow may occur as described in 9.2, unless the dividend fraction is zero, in which case a true zero result is generated.

In order to ensure that non-normalised operands can be used the dividend and divisor fractions are first normalised (ignoring overflow or underflow at this stage); after normalisation the latter may be scaled one place up to ensure that the quotient fraction does not overflow. In this

Fig 47 FLOATING DIVIDE (RDV) ACC' = ACC/Op

FLOATING REVERSE DIVIDE (RRDV) ACC' = Op/ACC

FLOATING DIVIDE DOUBLE (RDVD) ACC' = ACC/Op; ACS' = $\frac{1}{2}$ ACS

(1) (2)

If (1) is being divided by (2):

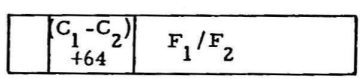
$C_1 = c_1 + 64$

$C_2 = c_2 + 64$

$(C_1 - C_2) + 64 = (c_1 + 64 - c_2 - 64) + 64 = (c_1 - c_2) + 64 = E + 64$

Again we have a correctly biased exponent.

Thus, answer is

ACC' 

case not more than one left shift will be required to normalise the quotient and therefore should be developed to 7, 15 or 29 digits as appropriate and the guard digit dropped after normalisation. The characteristic associated with the quotient fraction before normalisation is the difference of the characteristics of the normalised dividend and divisor, plus 64, (65 if the divisor fraction is scaled up). Overflow or underflow occurs as described in 9.2 if and only if the final characteristic exceeds 127 or is negative; in the latter case a true zero result is generated. When the result is not zero the sign bit is determined by the rules of algebra.

OV is cleared if overflow does not occur.

If ACS = 128 bits, bits 64 - 71 of the result are generated as described in 9.2.

CC : Unaltered.

Program errors : Operand addressing errors
Zero divide (unless masked) (see 12.1)
Floating overflow (unless masked) (see 12.1)
Floating underflow (unless masked) (see 12.1)

9.10 Floating Reverse Divide (RRDV) # BC

Operand length : ACS

Description : This operation is identical to Floating Divide (9.9) except that the quotient left in ACC is formed by dividing the operand by the

contents of ACC.

CC : Unaltered

Program errors : Operand addressing errors
Zero divide (unless masked) (see 12.1)
Floating overflow (unless masked) (see 12.1)
Floating underflow (unless masked) (see 12.1)

9.11 Floating Divide Double (RDVD) # BE

Operand length : Half ACS

Description : This operation requires ACS = 64 or 128 bits, and in the course of execution halves ACS. The contents of ACC are divided by the operand, the latter being half the size of ACC. ACS is halved and the normalised quotient (whose size accords with the new value of ACS) is left in ACC.

The operation is otherwise identical to Floating Divide (9.9) except that the dividend fraction is longer. All digits of the latter participate.

CC : Unaltered

Program errors : Operand addressing errors
Zero divide (unless masked) (see 12.1)
Floating overflow (unless masked) (see 12.1)
Floating underflow (unless masked) (see 12.1)
ACS = 32 bits (see 12.9/13.3)

9.12 Fixed-Point Format (see fig 48)

Fixed-point numbers are represented as 32-bit or 64-bit signed integers. The sign convention is 2's complement, bit 0 being the sign bit. The binary point is assumed to be to the right of the least significant bit. Thus the contributions to the value of a 32-bit operand made by 1's in different positions are as follows:

Bit 0	-2^{31}
Bit 1	$+2^{30}$
Bit 2	$+2^{29}$
Bit 31	+1

The largest positive number representable is $(2^{31} - 1)$ and the largest negative number is -2^{31}.

For a 64-bit number the contributions are:

Bit 0	-2^{63}
Bit 1	$+2^{62}$
Bit 2	$+2^{61}$
Bit 62	$+2^{1}$
Bit 63	+1

The largest positive and negative numbers representable in this format are $(2^{63} - 1)$ and -2^{63}, respectively.

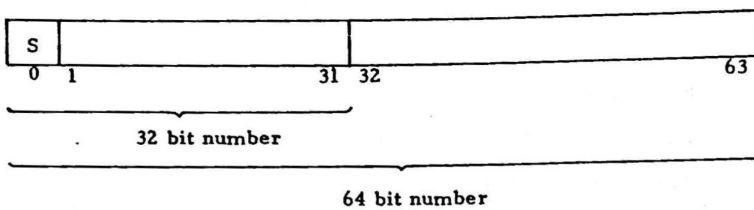
Fixed-point results which exceed capacity cause fixed-point overflow to occur. OV is set, and if the condition is not masked, interrupt ensues.

By observing suitable conventions regarding the positioning of the binary point, fixed-point instructions may be used to operate on fractions as well as integers.

Fixed-point numbers more than 64 bits long may be operated on by splitting them into 32-bit portions and using those, singly or in pairs, as operands for the instructions: Add Logical (9.22) Subtract Logical (9.23). Multiply Double (9.17) and Remainder Divide (9.20).

When multiplication and division are involved it will generally be necessary to hold only 31 bits of the number in each 31-bit portion, with a dummy (zero) sign bit. A pair of consecutive 32-bit portions in this form can be converted to the normal 64-bit number format by shifting bits 0-31 into bit positions 1-32 and vice-versa.

Fig 48 FIXED-POINT FORMAT



S is Sign Bit (0 for Positive, 1 for Negative)
Number is represented in Two's Complement form.
For a Positive Number: this is in Normal Binary Format which is Right Aligned and with bit 0 equal to zero.
A Negative Number can be formed by taking a Positive Number, inverting all the bits and adding a 1 to the L.S. bit position.
This gives the Two's Complement Form for a Negative Number.
Bit 0 will always be 1 for a valid Negative Number.

9.13 Fixed-Point Instructions

Add (IAD) # E0

Subtract (ISB) # E2

Reverse Subtract (IRSB) # E4

Operand length : ACS

Description : The operand is added to, or subtracted from, the contents of ACC, and the result left in ACC. ACS = 32 or 64 bits is assumed.

Fixed-point overflow occurs if the result lies outside the range of representable numbers (see 9.12); when it occurs OV is set and, if not masked, an interrupt ensues. The result left in ACC in this case is the least significant 32 or 64 bits of the true sum or difference. Overflow can only occur when adding numbers with like signs or subtracting numbers with opposite signs. OV is cleared if overflow does not occur. Reverse Subtract is exactly the same as Subtract, except that the difference left in ACC is formed by subtracting the original contents of ACC from the operand.

CC : Unaltered.

Program errors : Operand addressing errors
Fixed overflow (unless masked) (see 12.1)
ACS = 128 bits (see 12.9/13.0)

9.14 Compare (ICP) #E6

Operand length : ACS

Description : The operand is compared algebraically with the contents of ACC, and CC is set to indicate the result of the comparison. ACC, and OV are not altered.

Equality is indicated if all bits are equal.

ACC < operand is indicated if and when conducting a left-to-right scan of the bits of both, the first non-equivalent pair of bits occurs when the ACC bit concerned is a 1 (if bit 0) or 0 (any subsequent bit), and ACC > operand in the same circumstances when the ACC bit concerned takes the opposite values.

ACS = 32 or 64 bits is assumed.

CC :
0 Equality
1 ACC < operand
2 ACC > operand
3 Not Used

Program errors : Operand addressing errors

ACS = 128 bits (see 12.9/13.0)

9.15 Arithmetic Shift (ISH) #E8

Operand length : 32 bits

Description : The contents of ACC are effectively multiplied by 2^i where i the signed integer specified in the least significant 7 bits of the operand.

Other bits of the operand are ignored.

A positive value of i represents a leftward shift, in which case zeros are inserted at the least significant end of ACC and OV is set if the contents of bit 0 of ACC change at any time during shifting - this will cause an interrupt if the condition is not masked. If bit 0 does not change OV is cleared.

A negative value of i indicates a rightward shift. In this case the sign bit is propagated by leaving bit 0 unchanged during the shifting. Bits shifted off the right of ACC are lost, but CC is used to indicate what they were. OV is cleared by a rightward shift. ACS = 32 or 64 bits is assumed.

CC : 0 $i < 0$; all bits shifted off the right of ACC were 0's.

1 $i < 0$; last bit shifted off the right of ACC = 0 (some 1's)

2 $i < 0$; last bit shifted off the right of ACC = 1

3 $i \geqslant 0$

Program errors : Operand addressing errors
Fixed overflow (unless masked) (see 12.1)
ACS = 128 bits (see 12.9/13.0)

9.16 Multiply (IMY) * EA

Operand length : ACS

Description : The contents of ACC and the operand, both signed integers, are multiplied, and the least significant 32 or 64 bits of their product left in ACC. Overflow occurs if the result exceeds capacity (see 9.2). In this case OV is set and, unless fixed-point

overflow is masked, interrupt ensues.
Otherwise OV is cleared.
ACS = 32 or 64 bits assumed.

CC : Unaltered

Program errors : Operand addressing errors
Fixed overflow (unless masked) (see 12.1)
ACS = 128 bits (see 12.9/13.0)

9.17 Multiply Double (IMYD) # EC

Operand length : 32 bits (= ACS)

Description : This operation expects ACS = 32 bits and leaves ACS = 64 bits on completion. The 64 bit product of the contents of ACC and the operand is left in ACC. OV is cleared. CC is set to indicate the signs of the original contents of ACC and the operand in case unsigned arithmetic has to be implemented by software.

CC :
0 ACC and operand both positive
1 ACC positive, operand negative
2 ACC negative, operand positive
3 ACC and operand both negative

Program errors : Operand addressing errors
ACS = 64 or 128 bits (see 12.9/13.5)

9.18 Divide (IDV) # AA (see fig 49)

Operand length : ACS

Description : The contents of ACC are divided by the operand and the unrounded quotient left in ACC, all three being signed integers. The rules for determining the quotient are as for Remainder Divide (9.20), i.e. quotient times divisor is numerically not greater than dividend.

If the divisor is zero OV is cleared but the quotient is undefined, and interrupt occurs unless masked.

Overflow will occur, causing OV to be set and interrupt to ensue, unless masked, if -2^{31} or -2^{63} is divided by -1, ACC is unaltered. If overflow does not occur OV is cleared.

ACS = 32 or 64 bits is assumed.

CC : Unaltered.

Program errors : Operand addressing errors
Zero divide (unless masked) (see 12.1)
Fixed overflow (unless masked) see(12.1)
ACS = 128 bits (see 12.9/13.0)

Fig 49	DIVIDE (iDV) (#AA)	ACC' = ACC/Op
	REVERSE DIVIDE (IRDV) (#AC)	ACC' = Op/ACC
	REMAINDER DIVIDE (MDV) (#AE)	ACC' = ACC/Op; TOS = Remainder set CC

CC = 0 Remainder = 0 or Remainder > 0, divisor > 0
1 Remainder > 0, divisor < 0
2 Remainder < 0, divisor > 0
3 Remainder < 0, divisor < 0

9.19 Reverse Divide (IRDV) #AC

Operand length : ACS

Description : This operation is identical to Divide (9.18) except that the operand is divided by the contents of ACC rather than the other way round. If overflow occurs ACC will contain -2^{31} or -2^{63}, depending on ACS.
ACS = 32 or 64 bits is assumed.

CC : Unaltered

Program errors : Operand addressing errors
Zero divide (unless masked) (see 12.1)
Fixed overflow (unless masked) (see 12.1)
ACS = 128 bits (see 12.9/13.0)

9.20 Remainder Divide (IMDV) #AE

Operand length : ACS

Description : The quantity in ACC is divided by the operand, the quotient is left in ACC and the remainder is stacked. All these quantities are signed integers with length ACS; as a result of stacking the remainder SF is incremented.
ACS = 32 or 64 bits is assumed.

The remainder is numerically less than the divisor and, if non-zero, has the same sign as the dividend. CC is set to facilitate obtaining a remainder which obeys the rules for the PL/1 'Mod' function.

If the operand is zero the quotient, the remainder (which is stacked) and the setting of CC are undefined, but OV is cleared, and the zero divide interrupt ensues, unless masked. Overflow will occur if -2^{31} or -2^{63} (according to ACS) is divided by -1. This will cause OV to be set and interrupt to occur, unless masked. ACC is unaltered, an undefined remainder is stacked, and the setting of CC is undefined. Otherwise OV is cleared.

CC : 0 Remainder zero, or remainder > 0, divisor > 0
1 Remainder > 0, divisor < 0
2 Remainder < 0, divisor > 0
3 Remainder < 0, divisor < 0

Program errors : Operand addressing errors
Zero divide (unless masked) (see 12.1)
Fixed overflow (unless masked) (see 12.1)
ACS = 128 bits (see 12.9/13.0)

9.21 Logical Format

Logical operations on 32- or 64- bit items treat them either as strings of bits with no numerical significance, or as unsigned (i.e. positive) fixed-point numbers. In this case the contributions of individual bits are as described in 9.12 except that bit 0, if non-zero contributes $+2^{31}$ (32-bit format) or $+2^{63}$ (64-bit format). Overflow cannot occur with logical operations.

9.22 Logical Instructions

Logical Add (UAD) # C0

Operand length : ACS (=32 bits)

Description : The effect of this instruction is to leave in ACC the least significant 32 bits of the sum of the operand and the original contents of ACC, both regarded as unsigned integers. OV is cleared. CC is set to indicate whether or not carry occurred out of ACC bit 0.
The operation is only defined for ACS = 32 bits
If this operation is performed on words with dummy (zero) sign bits (i.e. portions of multiple length quantities) they should be left shifted to remove those bits.

CC : 0 No Carry
1 Carry
2 Not Used
3 Not Used

Program errors : Operand addressing errors
ACS = 64 or 128 bits (see 12.9/13.1)

9.23 Logical Subtract (USB) # C2

Operand length : ACS

Description : This operation is identical to Logical Add (9.22) except that the 2's complement of the operand is added to the contents of ACC. OV is cleared. ACS = 32 bits is assumed.

CC : 0 No carry (indicates 'borrow' into bit 0 in performing subtraction.).
1 Carry (indicates no 'borrow'). Includes the case where complementing causes

carry, ie. operand = 0).

2 Not Used

3 Not Used

Program errors : Operand addressing errors

ACS = 64 or 128 bits (see 12.9/13.1)

9.24 Logical Reverse Subtract (URSB) # C4

Operand length : ACS (= 32 bits)

Description : This operation is identical to Logical Subtract (9.23) except that the result is formed by adding the 2's complement of the original contents of ACC to the operand. OV is cleared. ACS = 32 bits is assumed.

CC : 0 No carry (indicates 'borrow')

1 Carry (indicates no 'borrow', includes the case where ACC was 0, so complementing causes carry).

2 Not Used

3 Not Used

Program errors : Operand addressing errors

ACS = 64 or 128 bits (see 12.9/13.1)

9.25 Logical Compare (UCP) # C6

Operand length : ACS

Description : The operand is compared with the contents of ACC, CC being set to indicate the result of the comparison.

Equality implies equivalence in every bit position. ACC < operand is indicated if, when scanning the bits of both from left to right, the first non-equivalence pair of bits occur when the ACC bit concerned is a 0. ACC > operand where it is a 1.

ACC and OV are unaltered.

ACS = 32 or 64 bits is assumed.

CC : 0 Equality
1 ACC < operand
2 ACC > operand
3 Not Used

Program errors : Operand addressing errors
ACS = 128 bits (see 12.9/13.0)

9.26 Logical Shift (USH) # C8

Operand length : 32 bits

Description : The least significant 7 bits of the operand are treated as a signed integer specifying the number of places of left (positive) or right (negative) shift applied to the contents of ACC. Zeros are inserted in the least or most significant bit of ACC as shifting proceeds. OV is cleared.
The remaining operand bits are ignored.
ACS = 32 or 64 bits is assumed.

CC : Unaltered.

Program errors : Operand addressing errors
ACS = 128 bits (see 12.9/13.0)

9.27 And (AND) # 8A
Or (OR) # 8C
NEQ # 8E

Operand length : ACS

Description : Each bit in ACC is replaced by a new bit generated from its original value and the corresponding bit in the operand, as follows:

Original ACC bit	Operand bit	Result Bits		
		AND	OR	NOT EQUIV.
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

OV is cleared.

ACS = 32 or 64 bits is assumed.

CC : Unaltered

Program errors : Operand addressing errors

ACS = 128 bits (see 12.9/13.0)

9.28 Rotate (ROT) # CA

Operand length : 32 bits

Description : The contents of ACC are shifted left by the number of binary places specified by the operand, interpreted as an unsigned integer, in such a way that each bit shifted off the left-hand end of ACC (bit 0) is re-inserted at the right-hand end (bit 31 where ACS = 32 bits; bit 63 where ACS = 64 bits).

OV is cleared.

ACS = 32 or 64 bits is assumed.

When ACS = 32 bits, bits 0-26 of the operand (bits 0-25 when ACS = 64 bits) do not affect the result, but may influence the time taken by the instruction, so it is recommended as a programming rule that when the operand is a literal, operand bits 25 and 26 should be the same as bit 27. Similarly, when ACS = 64 bits, bit 25 should be the same as bit 26. Deviation from this rule will not lead to any error.

CC : Unaltered

Program errors : Operand addressing errors.
ACS = 128 bits (see 12.9/13.0)

9.29 Shift 32 Bits (SHS) #CC

Operand length : 32 bits

Description : The least significant 32 bits of the contents of ACC are shifted logically in exactly the same way as for Logical Shift (9.26); in fact if ACS = 32 bits the instructions are identical. If ACS = 64 bits, the more significant 32 bits of ACC are unaltered. Zeros are inserted in the least (leftward shift) or most (rightward shift) significant bit position of the 32 bits shifted as shifting proceeds. OV is cleared.
ACS = 32 or 64 bits is assumed.

CC : Unaltered

Program errors : Operand addressing errors
ACS = 128 bits (see 12.9/13.0)

9.30 Shift While Zero (SHZ) #CE

Operand length : 32 bits

Description : The contents of ACC, if non-zero, are shifted logically leftward until bit 0 is a 1. The number of binary places shifted is stored as a 32 bit positive integer in the operand location. OV is cleared.
If ACC is zero, a zero is stored. This condition may be detected by testing ACC for zero after the operation.
ACS = 32 or 64 bits is assumed.

CC : Unaltered

Program errors : Operand addressing errors

Literal operand (see 12.8/12.1)
Non-zero bits of stored item truncated
(see 12.3/6.0)
ACS = 128 bits (see 12.9/13.0)

9.31 Decimal Data Format (see fig. 50)

Decimal numbers are held in the accumulator or in store in packed decimal format. See fig. 50. Each digit is represented in a hexadecimal form so that two digits are held to a byte. The 4 least significant bits of a decimal number contain the sign of the whole number - the values for positive and negative sign code are given in the diagram.

Thus a decimal number in this form always contains an odd number of digits, i.e. 7, 15 or 31 bits corresponding to ACC sizes 32, 64 or 128. Results generated by decimal operations have sign codes as follows:

#C for positive numbers, #D for negative numbers.

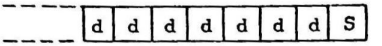
The only exceptions to this rule are DSH (Decimal Shift), SUPK (Suppress and unpack) and PK (pack). Zero results will have sign code #C except possibly after one of the above (exception) instructions or following an instruction which causes overflow.

Overflow will occur if the numeric part of the result is too large for the accumulator - OV will be set and unless decimal overflow is masked an interrupt will occur.

Incidentally the values of numeric digits in the operands are not checked, and the results of decimal operations are undefined when digits in the range #A - #F are present in the operands.

Fig 50 Decimal Data Format

Decimal Number	Hex	Decimal Number	Hex
0	0000	5	0101
1	0001	6	0110
2	0010	7	0111
3	0011	8	1000
4	0100	9	1001



Sign Code

Positive 1010 (#A), 1100 (#C), 1110 (#E)
1111 (#F)

Negative 1011 (#B), 1101 (#D)

All other values are undefined.

ACC Size	No. of Dec. Digits	Bit Positions of Sign Code
32	7	28 - 31
64	15	60 - 63
128	31	124 - 127

9.32 Decimal Instructions

Decimal Add (DAD) # D0

Decimal Subtract (DSB) # D2

Operand length : ACS

Description : The operand is added to, or subtracted from the contents of ACC. The subtraction operation is equivalent to changing the sign of the operand and adding.
If overflow occurs the result is correctly represented (including the sign digit) apart from the overflowed digit. OV is set and interrupt occurs unless a decimal overflow is masked.
OV is cleared if overflow does not occur.

CC : Unaltered.

Program errors : Operand addressing errors
Decimal overflow (unless masked) (see 12.1)

9.33 Decimal Reverse Subtract (DRSB) # D4

Operand length : ACS

Description : As for Decimal subtract (9.32) except that the difference left in ACC is formed by subtracting the original contents from the operand.

CC : Unaltered.

Program errors : Operand addressing errors
Decimal overflow (unless masked) (see 12.1)

9.34 Decimal Compare (DCP) # D6

Operand length : ACS

Description : The operand is compared with the contents of ACC. ACC and OV are unaltered. CC is set to indicate the result of the comparison. Equality is indicated if the operand is

identical in every numeric digit with the contents of ACC, and the sign digits are both positive or both negative (positive sign digits do not necessarily have the same bit representation), or if the sign digits are opposite but all the numeric digits of both are zeros. If the signs are opposite and the numeric digits are not all zeros, ACC < operand is indicated if the ACC sign is negative, ACC > operand otherwise. If the sign digits are equivalent ACC < operand is indicated if, when conducting a left-to-right scan of the numeric digits of both operands, the smaller digit of the first unequal pair is in ACC (positive sign) or in the operand (negative sign); otherwise ACC > operand is indicated. There is no check that the numeric digits lie in the range 0-9.

CC : 0 Equality
1 ACC < operand
2 ACC > operand
3 Not Used

Program errors : Operand addressing errors

9.35 Decimal Shift (DSH) # D8

Operand length : 32 bits

Description : The least significant 7 bits of the operand specify the amount by which the contents of ACC are to be shifted. This amount is interpreted as a signed integer (i) in the range -64 to +63. Other bits of the operand are ignored. If the integer is positive (=i) all but the least significant 4 bits of ACC are shifted 4i binary places to the left. The

sign digit is unaltered. If any of the bits shifted off the leftmost end of ACC are 1's, OV is set and interrupt occurs unless decimal overflow is masked. OV is cleared. if all the bits shifted off are zeros. Zero bits are inserted in the bit position adjacent to the sign digit as shifting proceeds. If the amount of shift is negative (= -i) all except the least significant 4 bits of ACC are shifted 4i binary places to the right. OV is cleared. The sign digit is unaltered. Bits shifted out of the position to the left of the sign are lost. Zeros are inserted at the most significant end of ACC.

CC : Unaltered

Program errors : Operand addressing errors
Decimal overflow (unless masked) (see 12.1)

9.36 Decimal Multiply (DMY) #DA

Operand length : ACS

Description : The product of the contents of ACC and the operand is left in ACC. If the product exceeds ACC capacity the result is undefined; OV is set and interrupt occurs unless decimal overflow is masked.
Otherwise OV is cleared.

CC : Unaltered

Program errors : Operand addressing errors
Decimal overflow (unless masked) (see 12.1)

9.37 Decimal Multiply Double (DMYD) #DC

Operand length : ACS

Description : The double-length product of the contents of ACC and the operand is left in ACC, ACS

being doubled in the course of the operation. OV is cleared. ACS = 128 bits is not permitted.

CC : Unaltered.

Program errors : Operand addressing errors
ACS = 128 bits (see 12.9/13.5)

9.38 Decimal Divide (DDV) #9A

Operand length : ACS

Description : The contents of ACC are divided by the operand and the unrounded quotient left in ACC. The rules for determining the quotient are those for Decimal Remainder Divide (9.40). OV is cleared.
If the divisor is zero (i.e. all its numeric digits are zeros) the result is undefined (but OV is cleared), and the zero divide interrupt occurs unless masked.

CC : Unaltered.

Program errors : Operand addressing errors
Zero divide (unless masked) (see 12.1)

9.39 Decimal Reverse Divide (DRDV) #9C

Operand length : ACS

Description : The operation is exactly as for Divide (9.38) except that the operand is the dividend and the contents of ACC the divisor.

CC : Unaltered.

Program errors : Operand addressing errors
Zero divide (unless masked) (see 12.1)

9.40 Decimal Remainder Divide (DMDV) # 9E

Operand length : ACS

Description : The contents of ACC are divided by the operand; the quotient is left in ACC, and the remainder is stacked, causing SF to be incremented. All these quantities are of length ACS. OV is cleared.

The quotient value is such as to produce a remainder which is either zero, or of the same sign as the dividend and numerically less than the divisor.

CC is set to facilitate the evaluation of the PL/1 'Mod' function. If the divisor is zero (all its numeric digits are zero) the quotient, the remainder (which is stacked) and the setting of CC are all undefined (but OV is cleared), and the zero divide interrupt occurs unless masked.

CC :
0 Remainder zero, or remainder > 0, divisor > 0
1 Remainder > 0, divisor < 0
2 Remainder < 0, divisor > 0
3 Remainder < 0, divisor < 0

Program errors : Operand addressing errors
Zero divide (unless masked) (see 12.1)

Miscellaneous Instructions

Four functions are provided to allow conversion between floating-point and fixed - point formats and fixed - point and decimal formats.

9.41 Fix (FIX) * B8 (see fig 51)

This instruction converts data from floating - point to fixed - point format. Prior to the instruction ACC is loaded with the floating - point number to be converted.

Operand length : 32 bits

Description : The exponent, and sign and fraction, of the floating - point number in ACC are separated; the former is adjusted and stored as a 32 bit signed integer in the operand location, the latter is left in ACC as a signed (2's complement) integer. If the number in ACC had a zero fraction, a zero exponent is stored. ACS may be 32, 64 or 128 bits. If it was 128 bits ACS is halved and the least significant 14 digits of the fraction are lost, CC being set to indicate the nature of the lost bits.

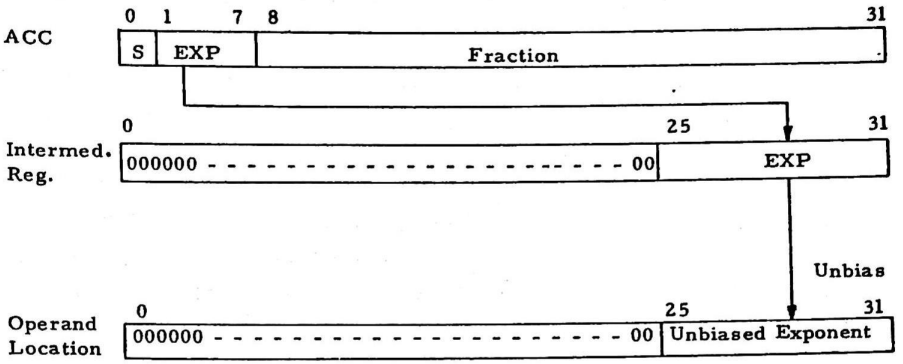
OV is cleared.

If the fraction (all 6, 14 or 28 digits) is zero the action of the instruction is to clear ACC and OV, halve ACS if it was 128 bits, and store a 32 bit zero word in the operand location. CC is set to 0.

If the fraction is non-zero, bits 1 - 7 of ACC are stored at the least significant end of a 32 bit intermediate register, whose remaining bits are zeros. The exponent is unbiased, and the fraction effectively converted to an (unsigned) integer, by subtracting 70 (ACS = 32 bits) or 78 (ACS = 64 or 128 bits) from the quantity in the intermediate location which is

Fig 51 Fix (FIX) #B8

If fraction is non-zero e.g. 32 bit ACC



The exponent is unbiased by subtracting 70 (ACS = 32 bits)
78 (ACS = 64, 128 bits)

from the quantity in the intermediate register.

If bit 0 of ACC = 0, bits 1-7 are made zeros.

If bit 0 of ACC = 1, fraction is negated, bits 0-7 of ACC are made 1's.

CC = 0 No non-zero bits lost.

1. (ACS = 128 bits) lost portion $< \frac{1}{2}$
2. (ACS = 128 bits) lost portion $\geq \frac{1}{2}$
3. Not Used.

then stored in the operand location.
If bit 0 of ACC is 0 bits 1-7 are made zeros.
If bit 0 of ACC is 1, the fraction (effectively including bits 72 - 127 if ACS = 128 bits; although these bits are subsequently discarded they effect the value of bit 63 and of CC) is negated, and bits 0-7 of ACC made 1's. If ACS = 32 or 64 bits CC is set to 0. If ACS = 128 bits, CC is set to 2 if bit 72 of ACC (after negation, if any) is non-zero, to 1 if bit 72 is zero but bits 73-127 are not all zeros, to 0 if bits 72 - 127 are all zeros; the more significant 64 bits of ACC overwrite the less significant 64 and ACS is set to 64 bits. OV is cleared.

CC : 0 No non-zero bits lost
1 (ACS = 128 bits) lost portion $< \frac{1}{2}$
2 (ACS = 128 bits) lost portion $\geqslant \frac{1}{2}$
3 Not Used

Program errors : Operand addressing errors
Literal operand (see 12.8/12.1)
Non-zero bits of stored item truncated (see 12.3/6.0)

9.42 Float (FLT) #A8 (see fig 52)

Operand length : 32 bits

Description : The 32 or 64-bit signed integer in ACC is combined with the exponent value specified by the operand to form a normalised floating-point number in ACC. ACS (32 or 64 bits) is doubled. The least significant 8 bits of the operand specify a signed integer which is the hexadecimal exponent associated with the integer in ACC.

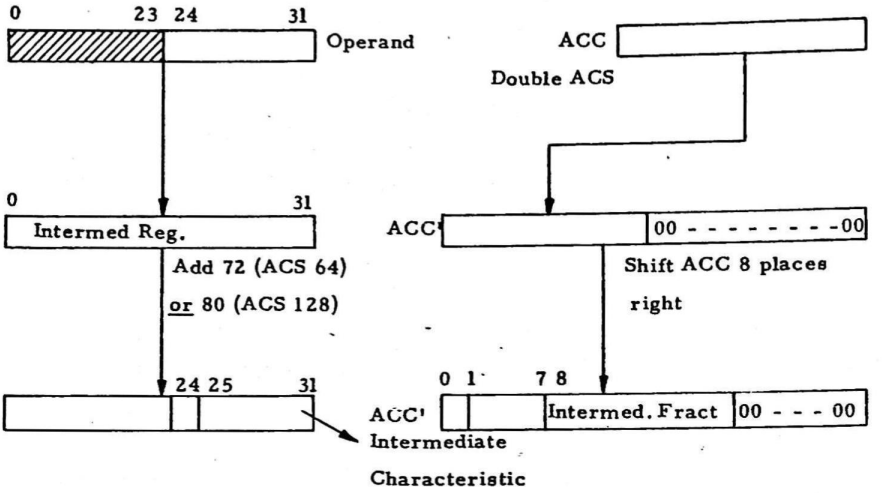
Other operand bits are ignored.
If the contents of ACC are zero the action of the instruction is to double ACS, extending ACC with another zero word or double word. The value of the operand is immaterial.
OV is cleared. If the contents of ACC are non-zero, the action is effectively as follows (though hardware may not follow these steps precisely):

- ACS is doubled, the previous contents of ACC now being placed in the more significant half and the less significant half made zero.
- The least significant 8 bits of the operand are placed in an intermediate register. The value of the most significant of these 8 bits is recorded for future reference. The quantity in the register is incremented by 72 (ACS = 64 bits) or 80 (ACS = 128 bits) - this is now the 'intermediate characteristic'.
- The contents of ACC are shifted right arithmetically 8 binary places. If bit 0 of ACC was originally a 1, the contents of ACC are negated (to form the modulus of the fraction) and bit 0 of ACC is made 1. Bits 8 onward now form the 'intermediate fraction'; bit 0 is the sign bit.
- The intermediate fraction is now normalised by shifting it up one hexadecimal place (4 bits) at a time until bits 8-11 are not all zeros. 1 is subtracted from the intermediate characteristic for every hexadecimal shift.
- The least significant 7 bits of the intermediate characteristic overwrite bits 1-7 of ACC. If ACS = 128 bits, the contents of the

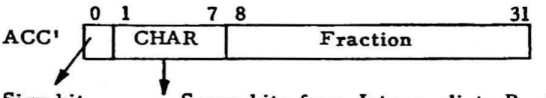
Fig 52 Float (FLT) #A8 converts fixed point number in ACC into floating point format.

For ACC = 0 ACS' = 2 ACS : ACC' = 0

For ACC ≠ 0



The intermediate fraction is normalised.



Sign bit

Seven bits from Intermediate Register (Intermediate Characteristic).

least significant 64 bits of ACC are shifted 8 binary places to the right and a copy of the characteristic in bits 1-7, minus 14 (or plus 114, if the latter is less than 14) inserted in bits 65-71 of ACC. Bit 64 is made the same as bit 0. OV is cleared.
- The most significant bit of the 8-bit intermediate characteristic should be 0. If it is a 1, underflow or overflow occur as described in 9.2, depending on whether the corresponding bit of the original operand was 1 or 0, respectively.
If underflow occurs ACC is made 0. If overflow occurs OV is set.

CC : Unaltered

Program errors : Operand addressing errors
Floating overflow (unless masked) (see 12.1)
Floating underflow (unless masked) (see 12.1)
ACS = 128 bits (see 12.9/13.2)

9.43 Convert To Binary (CBIN) # DE

Operand length : Not applicable. Literal must be specified.

Description : The operand is ignored. The signed integer in ACC is converted from the packed decimal representation to fixed-point binary representation, with 2's complement sign convention. If ACS = 128 bits it is halved; in this case overflow may occur if the number lies outside the range -2^{63} to $+(2^{63}-1)$, inclusive, whereupon OV is set, and interrupt occurs unless fixed-point overflow is masked. Otherwise OV is cleared.

CC : Unaltered

Program errors : Fixed overflow (unless masked) (see 12.1)

9.44 Convert to Decimal (CDEC) # EE

Operand length : Not applicable. Literal must be specified.

Description : The contents of ACC, a signed fixed-point integer, are converted to standard decimal form in ACC (see 9.31).
ACS is doubled. OV is cleared.
ACS = 128 bits is not permitted.

CC : Unaltered

Program errors : ACS = 128 bits.

10. STORE TO STORE FUNCTIONS.

10.1 Strings

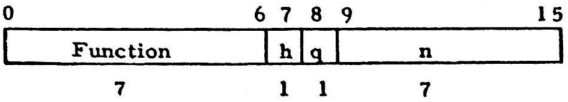
A versatile repertoire of store-to-store functions is provided on 2900. As was mentioned in the 'instruction formats' chapter, these are specifically provided for the commercial data processing establishments (generally the moving or manipulation of data from one part of store to another possibly allowing editing during the move).

Store-to-store operations take place between a string of bytes, referred to as the (DR) string, described by a string or byte-vector descriptor held in DR (usually, but not always, the 'destination string') and (in most cases) a second string described by a string or byte-vector descriptor held in ACC (referred to as the (ACC) string, and usually the 'source string'). In some cases (ACC) is not involved, and a string consisting of copies of a byte specified as a literal in the instruction may be used as source string, or alternatively a source byte may be taken from B; in three cases the (DR) string interacts directly with the contents of ACC itself.

Byte-vector and string descriptors are checked to ensure their size fields contain 011. The length of a string described by a byte-vector is in this context defined by the contents of the bound field.

The number of bytes involved in a store-to-store operation (referred to as L) is specified either explicitly in the instruction format, or in the length field of the descriptor in DR. Instructions are 16 or 32 bits long, and use the secondary format described in chapter 1.

The format of the first 16 bits is as follows:-



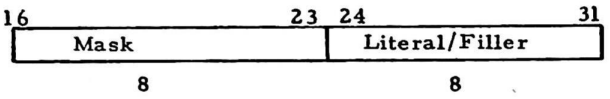
h = 0 : L = n + 1 ($1 \leq L \leq 128$)

h = 1 : L = (length field of DR)($0 \leq L \leq 2^{24}$) (n field reserved)

q = 0 : 16-bit instruction

q = 1 : 32-bit instruction

When the instruction is 32 bits long, the second 16 bits specify a Mask byte and a Literal or Filler byte, thus



In the course of each operation, the (DR) string is processed from left to right, one byte at a time, and for each byte processed the address field is incremented by 1 and the bound field decremented by 1. When the (ACC) string is similarly processed from left to right (this is not the case for all instructions) the descriptor in ACC is updated likewise (and OV is cleared).

When the (ACC) string is processed left to right, and it contains less than L bytes, it is effectively extended on the right with copies of the filler byte. If there is no filler byte, i.e. the instruction is 16 bits long, an interrupt occurs. When the filler byte is used, the descriptor in ACC is left with zero in its length field and the original contents of the length field added into the address field. If the ACC

length field is initially zero, the filler is used immediately, and the address field is ignored. If L = 0 the instruction will be interpreted as a null operation, the contents of the (ACC) and (DR) strings being ignored and unaltered (virtual store interrupt cannot occur).

10.2 Checks

All store-to-store operations include certain standard checks; interrupt occurs if any check fails. These checks (referred to in the instruction description) are:

1) DR must contain a type 0 or type 1 descriptor with size Code 3, or an escape descriptor, else see 12.6/10.6.

2) When L = n + 1 (literal), L $\leqslant$ (length field of DR) else see 12.7/11.0.

3) When ACC contains a descriptor, it must be of the correct type (type 0 or 1 with size code 3, except for Table check and Table translate), and ACS = 64 bits (for some instructions (ACC) is only used with the 16-bit instruction format), else see 12.6/10.7 or 8.

4) When the (ACC) string is processed left to right, and the instruction is 16 bits long, (length field of descriptor in ACC) $\geqslant$ L, else see 12.7/11.2.

Most store-to-store operations can be interrupted in mid-flight, and resumed after the interruption. As well as (DR) and, where relevant, (ACC) being updated in the course of the operation, additional facilities are provided enabling the instructions to be thus resumed.

The results of store-to-store operations which necessitate changing the contents of store locations are undefined if the two strings involved overlap, unless otherwise specified.

For the purposes of determining overlap the (DR) string is taken to be L bytes long, and the ACC string's length is the lesser of L and the length specified by the descriptor in ACC.

When the source information is taken from B, the contents of B are unaltered by the instruction. Fields of B ignored by the instruction are spare.

Note that these instructions only shift BYTES - there are no word shifts. It is also important that the source and destination string descriptors be loaded prior to the operation.

For short instructions, the format is:

Function h, n,

where h indicates where the number of bytes involved is to be specified explicitly, (h = 0) as n + 1, or implicitly, (h = 1) as the length field of DR.

For long instructions, the format is:

Function h, n, m, lit/filler,

where m is the mask byte and there is also a literal/filler byte. The use of these bytes is demonstrated for the move instruction, although the rules hold true for other instructions in the series. The following table (Fig. 53) summarises whether the mask and literal/filler bytes in B or the second half of a long instruction are used by each store to store instruction.

Fig 53. VALIDITY OF MASK AND LITERAL/FILLER BYTES

Instruction Mnemonic	16-bit Instruction		32-bit Instruction		Description in Section
	B(16-23)	B(24-31)	Inst. (16-23)	Inst. (24-31)	
MV	not used	not used	used	used if L> (ACC) length	10.3
MVL	used	used	used	used	10.5
SWEQ	used	used	used	used	10.6
SWNE	used	used	used	used	10.7
CPS	not used	not used	used	used if L> (ACC) length	10.8
ANDS ORS NEQS	not used	not used	ignored (reserved)	used	10.9
CHOV	not used	not used	ignored (reserved)	ignored (reserved)	10.11
PK					10.13
SUPK	not used	used if CC = 0	ignored (reserved)	used if CC = 0	10.14
TTR TCH	not used	not used	ignored (reserved)	ignored (reserved)	10.16 10.17
INS	used if CC ≠ 0	used if CC = 0	used if CC ≠ 0	used if CC = 0	10.18

10.3 Move (MV) # B2 (see fig 54)

Description : The (ACC) string overwrites the (DR) string. In the 16-bit case the (ACC) string must be at least L bytes long. In the 32-bit case the (ACC) string, if shorter than L bytes, is effectively extended with copies of the filler byte; and only those bits of each (DR) string byte which correspond to 0's in the mask byte are altered (to the corresponding bits of the appropriate (ACC) string byte or the filler byte).

The result is undefined if the (DR) string overlaps the (ACC) string on the right- i.e if the first byte of the (DR) string coincides with any one of the 2nd to nth bytes of the (ACC) string, where 'n ' is the lesser of L and the length of the (ACC) string. Otherwise the fields may overlap in any way and the correct result is obtained. If L = 0, and none of the other checks fail, a null operation is performed, leaving ACC and DR unaltered.

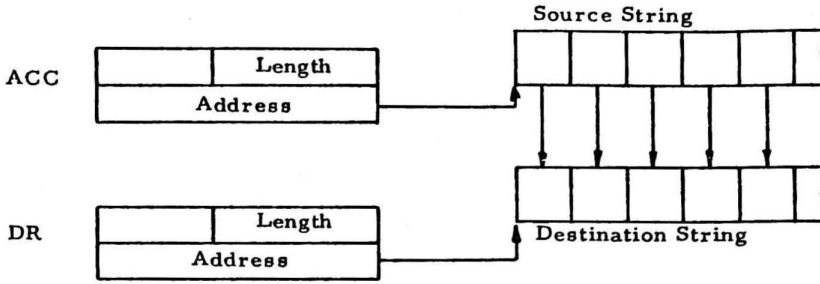
CC : Unaltered.

Program errors : Any failures of standard checks 1 - 4 (see 10.1)

10.4 Move Example

Fig 55 shows a worked example of a move instruction. The (ACC) string is 4 bytes long and the (DR) string is 6 bytes long - thus the first four bytes are taken from the (ACC) string, the remainder from the filler byte. The mask byte only allows the most significant two and least significant four bits of each byte to be propagated.

Fig 54. MOVE (MV)



If we wish to move 6 bytes, the instruction can be written as either:

MV 0, 5 (i.e. L = 6 = n-1 therefore n = 5) the compiler subtracting 1 automatically.

or MV 1, 0 which specifies the length of the transfer by the length field of the descriptor in DR.

Fig 55. EXAMPLE OF MOVE

Suppose we wish to transfer four bytes except bits 2 and three of each byte, which remain unchanged. Let us suppose further that; ACC string (source) contains four bytes: AA, 76, 24, EF; DR string (destination) contains six bytes (all zeros); and the fifth byte of the destination string is to be set to all ones except those bits masked out.

So to mask out bits 2 and 3 a mask:

00110000 = X'30' is constructed.

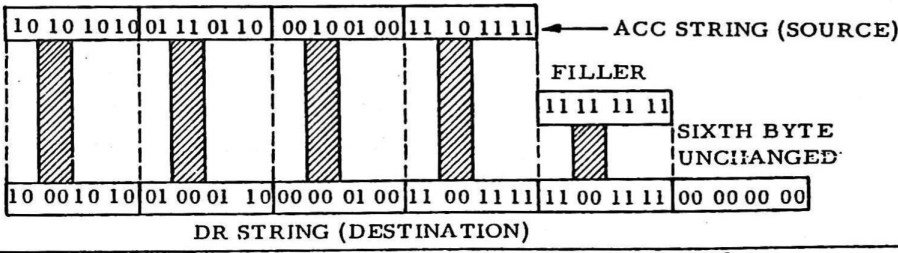
For the fifth byte a filler of all ones:

11111111 = X 'FF' is constructed.

The secondary order in SFL to achieve this is:

MV.N 5, X'30', X'FF'

Diagramatically it can be shown as:



10.5 Move Literal (MVL) #B0

Description : The unmasked bits of the source byte overwrite the corresponding bits of each byte of the (DR) string. In the 16-bit case, the source byte is the contents of bits 24-31 of B, and the mask byte the contents of bits 16-23. In the 32-bit case, the source byte is the literal byte. Only those bits of each (DR) string byte which correspond to 0's in the mask byte are altered (to the corresponding bits of the source byte). If L = 0, and none of the other checks fail, a null operation is performed, leaving DR unaltered.

CC : Unaltered.

Program errors : Any failures of standard checks 1 and 2, (see 10.1).

10.6 Scan While Equal (SWEQ) #A0 (see fig. 56)

Description : In the 16-bit case, the reference byte is the contents of bits 24-31 of B, and the mask byte the contents of bits 16-23. In the 32-bit case the reference byte is the literal byte.

Each byte in the (DR) string, working from left to right, is compared with the reference byte. Only those bits in each byte, including the reference byte, which correspond to 0's in the mask byte, are compared.

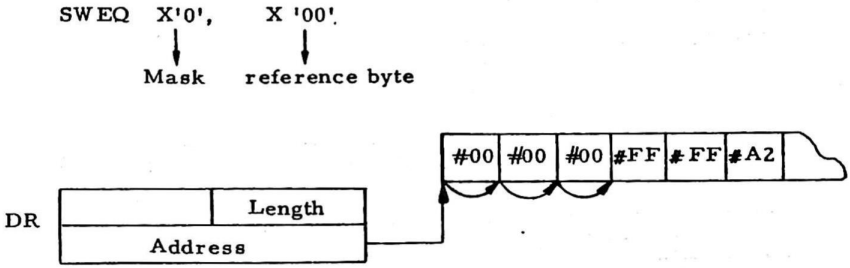
The operation stops when the (DR) string is exhausted, or when inequality is detected, whichever occurs first; in the latter case (DR) will finish pointing to the first byte in the (DR) string which is not equal to the reference byte.

CC is set to indicate whether or not inequality was found, and if so whether (the unmasked portion of) the (DR) byte was less than or greater than the (unmasked portion of) the reference byte.
If L = 0, and none of the other checks fail, a null operation is performed, leaving DR unaltered, but leaving CC = 0.

CC : 0 Inequality not found
1 Not Used
2 (DR) string byte > reference byte (unmasked portions)
3 (DR) string byte < reference byte (unmasked portions)

Program errors : Any failures of standard checks 1 and 2, (See 10.1).

Fig.56 Scan While Equal (SWEQ)



Reference byte is contents of bits 24-31 B register (for 16 bit instruction) or contents of literal part of instruction (for 32 bit instruction).

In this example at the end of the instruction (DR) will point to the first byte containing #FF and CC will be set to 2.

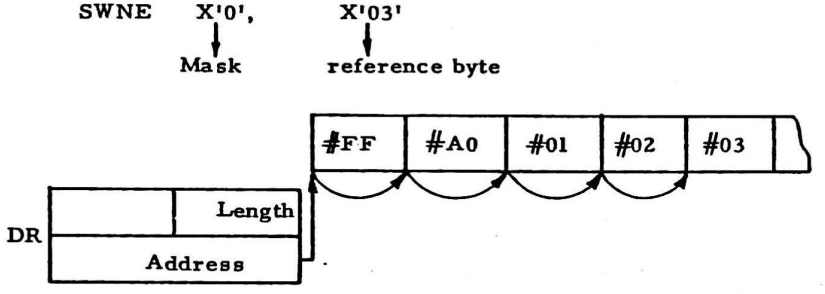
10.7 Scan While Not Equal (SWNE) #A2 (see fig. 57)

Description : This operation is similar to Scan while equal (10.6) except that the operation terminates when the unmasked portion of a (DR) byte equals the unmasked portion of the reference byte, or after L bytes. If L = 0 and none of the other checks fail, a null operation is performed, leaving DR unaltered, but leaving CC = 0.

CC :
0 Equality not found
1 (DR) string byte - reference byte (unmasked portions)
2 Not Used
3 Not Used

Program errors : Any failures of standard checks 1 and 2 (see 10.1).

Fig 57 SCAN WHILE NOT EQUAL (SWNE)



Reference byte is contents of bits 24-31 B Register (16-bit instruction) or contents of literal part of instruction (32 bit instruction)

In this example at the end of the instruction (DR) will point to the byte containing #03 and CC will be set to 1.

10.8 Compare Strings (CPS) A4 (See fig. 58)

Description : Successive bytes of the (ACC) and (DR) strings are compared - i.e. the first byte of one with the first byte of the other, and so on - until an unequal pair is found, or L bytes have been compared equal. In the 16-bit case the (ACC) string must be at least L bytes long. In the 32-bit case comparison is only applied to those bits which correspond to 0's in the mask byte; and if the (ACC) string is less than L bytes long, it is effectively extended (if necessary) with copies of the filler byte.

CC is set to indicate the result of the comparison. Where inequality is found, (DR) will finish pointing to the first (DR) string byte which compared unequal, and (ACC) likewise unless the (ACC) string has already expired.

If L = 0, and none of the other checks fail, a null operation is performed, leaving ACC and DR unaltered, but leaving CC = 0.

CC :

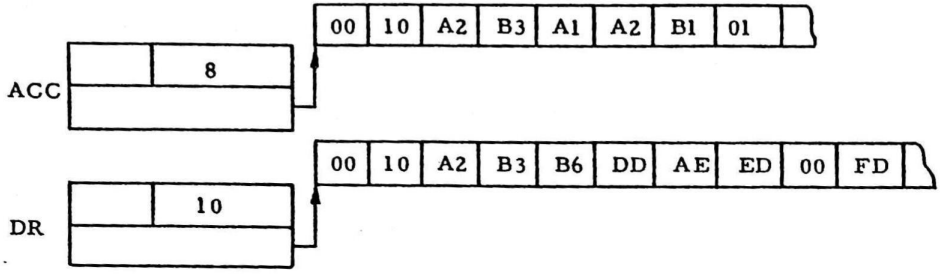
0 Inequality not found
1 Not Used
2 (DR) String byte > (ACC) String byte (unmasked portions)
3 (DR) String byte < (ACC) String byte (unmasked portions)

Program errors : Any failures of standard checks 1-4. (see 10.1).

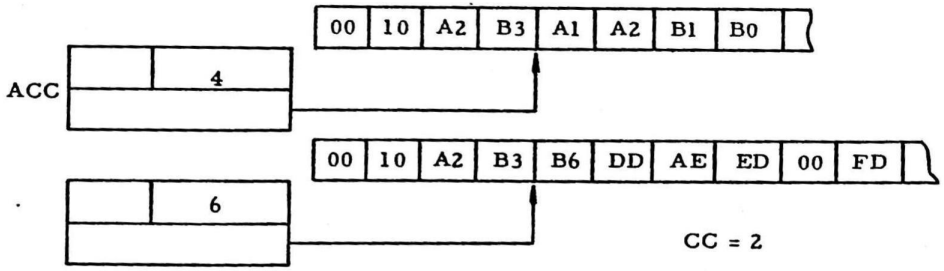
Fig 58 COMPARE STRING (CPS)

CPS.N 6

Before execution:



After execution:



10.9 And Strings (ANDS #82

Or Strings (ORS) #84

NEQ Strings (NEQS) #86

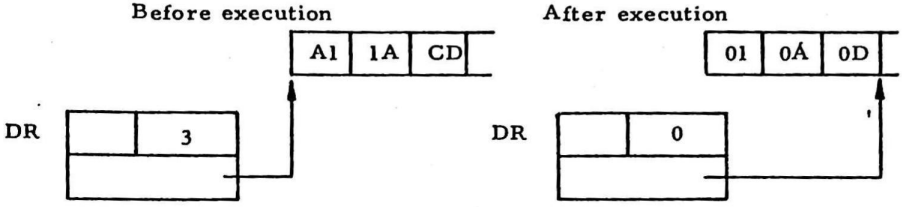
Description : See Fig. 59. Each byte of the (DR) string is replaced by the result of performing the appropriate logical operation between itself and the corresponding byte of the source string.
In the 16-bit format the source string is defined as the (ACC) string. In the 32-bit format ACC is not used and the source string is L copies of the literal byte. The mask byte is ignored (reserved).
If L = 0, and none of the other checks fail, a null operation is performed, leaving ACC and DR unaltered.

CC : Unaltered.

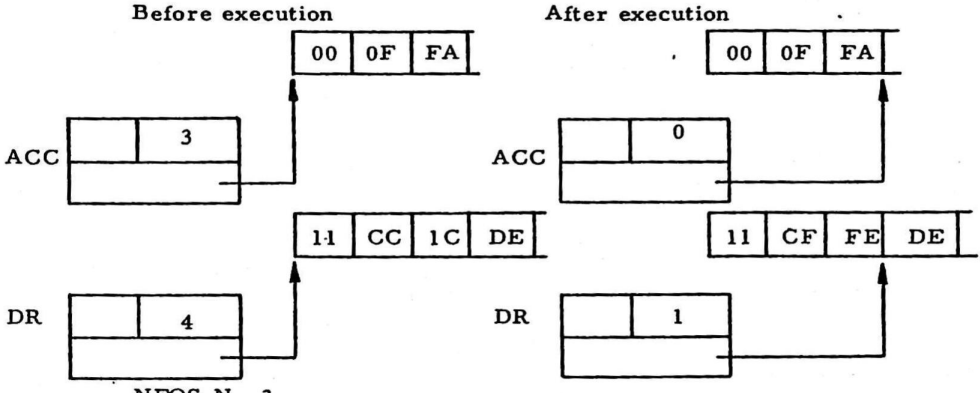
Program errors : Any failures of standard checks 1-4 (see 10.1).

Fig. 59 AND STRINGS (ANDS)
OR STRINGS (ORS)
NEQ STRINGS (NEQS)

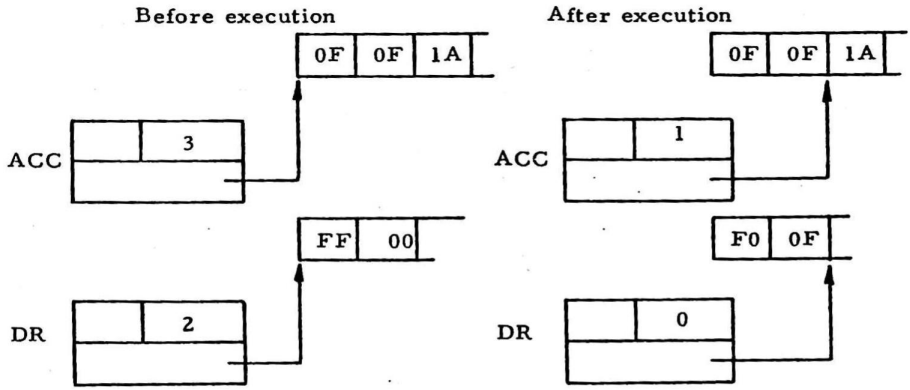
ANDS X'0', X'0F'



ORS.N 3



NEQS.N 3

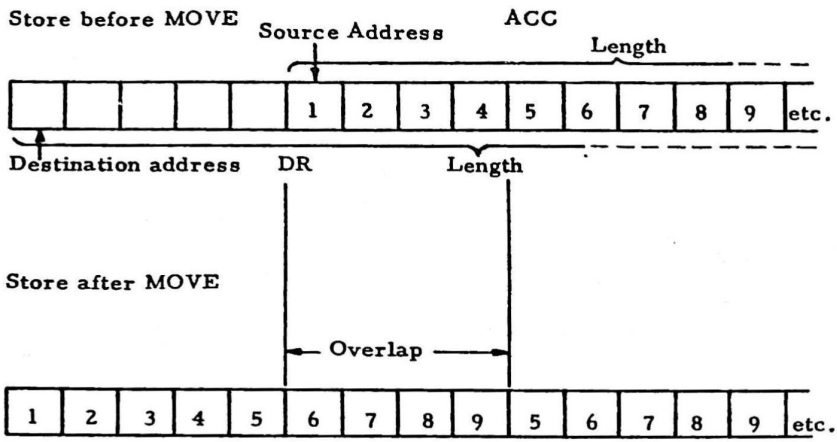


10.10 String Overlaps

If the address of a source string in store coincides with the address of the destination string then chaos could be caused by an instruction such as a MOVE. However, some overlap conditions could be tolerated. See Fig. 60 (a), (b) and (c).

Fig. 60 OVERLAP CONDITIONS

Fig. 60(a) Overlap on the LEFT MV.N 9

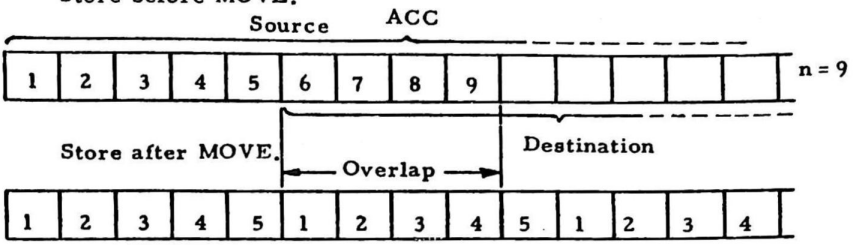


Although the overlapped locations are now corrupted (as a future source) the MOVE itself has been successful i.e. 9 Bytes have been correctly moved from the address in ACC to the address in DR.

∴ overlap on the left is acceptable.

Fig. 60 (b) Overlap on RIGHT MVN.9

Store before MOVE.



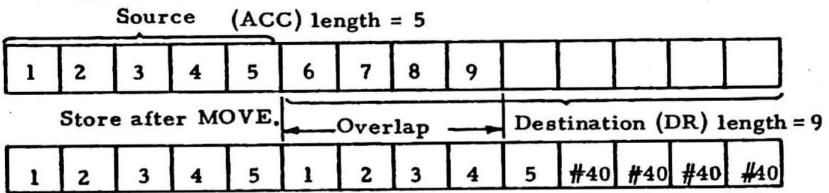
Not only are the overlapped locations corrupted (as a future source) but the MOVE itself has been unsuccessful i.e. Only 5 bytes have been correctly moved. Bytes 6 - 9 were overwritten before they could be moved. This is termed a corrupting overlap.

∴ Overlap on the RIGHT is unacceptable.

Note:- In the above example if the Source (ACC) String length was 5 or less, and the 32-bit format was used (with Mask and Filler specified) then the overlap would be acceptable, as shown below.

Fig. 60 (c) No effective overlap MV X'00', X'40'

Store before MOVE.



The item defined by (ACC) has been correctly moved and the remainder correctly filled.

For the purposes of MV and CHOV:

'n' in example 2 is 9 (L < Source (ACC) String length)

'n' in example 3 is 5 (Source (ACC) String length < L)

10.11 Check Overlap (CHOV) #B4

Description : This instruction tests whether the (ACC) and (DR) strings overlap, and if so whether or not the starting address of the (DR) string is greater than that of the (ACC) string. For the purpose of testing for overlap, the length of the (DR) string is L, and the length of the (ACC) string is the lesser of L and the contents of the (ACC) length field. The latter is only permitted to be less than L if the 32-bit instruction format is used, in which case the mask and filler bytes are ignored (reserved); if the length of the (ACC) string is zero, no overlap is indicated. CC is set to indicate the type of overlap. ACC and DR are unaltered. If L = 0, CC is set = 0.

CC :
0 No overlap
1 Corruption of source string when overlap on LEFT.
2 Corruption of destination string.

If the (ACC) string length < (DR) string length, the mask and filler bytes are ignored. Thus CC values 0, 1 indicate permissible overlap, while a value of 2 indicates corrupting overlap.

Program errors : Any failure of standard checks 1-4 (see 10.1).

CHOV Flowchart

L = No of bytes to be moved

Defined by:

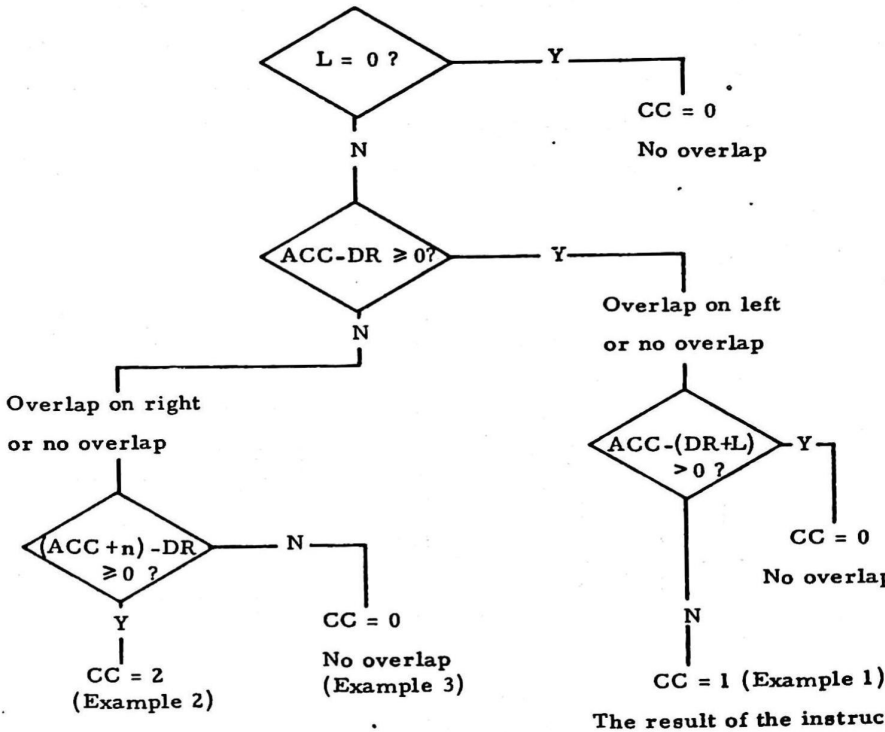
If n = 0 Literal field + 1

If n = 1 Length field of DR (destination)

n = The lesser of L and the length of the (ACC) Source String.

ACC = Address of 1st Source byte.

DR = Address of 1st destination byte.



The result of the instruction will be incorrect and the Source String will be corrupted.

The result of the instruction will be correct but the Source String will be corrupted.

10.12 Decimal Numbers

Decimal numbers can be represented in unpacked form by either the ISO format or the EBCDIC (Extended Binary Coded Decimal Interchange Code) format. In both formats each decimal digit occupies a byte, since the numeric is preceded by a zone code. Numbers in ISO have a zone code of #3, in EBCDIC # F - see fig 61.

Both formats are usable but they are wasteful as one byte represents a single character or number, so they are converted to packed decimal format (which was described in chapter 9 computational functions) where each decimal digit occupies 4 bits. This conversion is called PACKING and the reverse operation is UNPACKING.

Fig 61. DECIMAL NUMBERS

10.13 Pack (PK) # 90

This instruction converts the zone/numeric formats into a packed decimal format leaving the result in the accumulator. Prior to the instruction, a descriptor pointing to the string of unpacked numbers is loaded into DR. (Fig. 62).

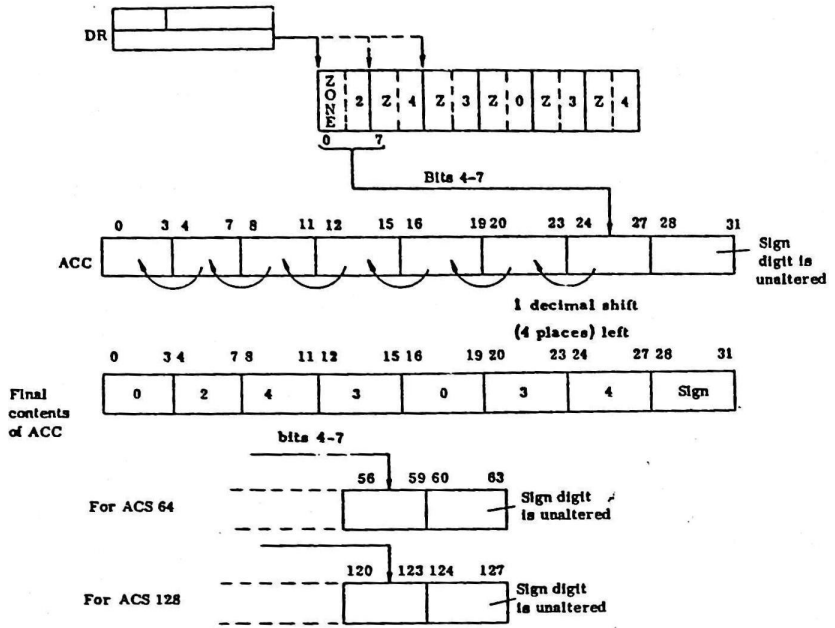
Description : For each (DR) string byte, working from left to right, the contents of ACC are shifted decimally left by 1 place and the least significant 4 bits of the byte inserted in the space thus created next to the sign digit of ACC. ACS may be 32, 64 or 128 bits. OV is set if any non-zero bits are shifted off the top of ACC, and in this case decimal overflow interrupt will occur unless masked. If no non-zero bits are shifted off OV is cleared.
The sign digit is unaltered.

If L = 0, and none of the other checks fail, a null operation is performed, leaving ACC and DR unaltered; OV is cleared.
Either 16 or 32 bit forms may be specified but for 32 bits the mask and literal bytes are ignored (reserved).
The results of the instruction is undefined if L $\geq$ 128.

CC : Unaltered.

Program errors : Any failures of standard checks 1 and 2 (see 10.1).
Decimal overflow (unless masked) (see 12.1).

Fig 62 PACK (PK) (#90) Example with ACS 32.



10.14 Suppress and Unpack (SUPK) # 94 (see fig 63)

Description : Successive digits of the decimal number in ACC are unpacked, each digit generating a byte which overwrites the next position in the (DR) string. The value of the inserted byte depends on the value of the leftmost (unpacked) digit of ACC, and on the setting of CC. After each (DR) string byte has been overwritten, ACC is decimally shifted up one place to remove the unpacked digit, and CC is updated. The sign digit is not shifted or altered.

The inserted byte is either a copy of the literal byte (bits 24-31 of B if the 16-bit instruction format is used), or is formed by prefixing bits 0-3 of ACC (the unpacked digit) with a numeric zone code. In the action table below, these two alternatives are referred to as 'insert literal' and 'insert digit' respectively. In the latter case, the zone code is binary 0011 if the ISO mode bit in SSR is 1, binary 1111 if it is 0.

	CC = 0	CC ≠ 0
Unpacked digit = 0	Insert literal CC unaltered	Insert digit CC unaltered
Unpacked digit ≠ 0	Insert digit Set CC = 2 *Stack descriptor See Overleaf	Insert digit Set CC = 2

*If CC = 0 and the digit being unpacked is non-zero, a type 1 descriptor is generated and stacked, which has 1 in its length field and whose address field contains 1 less than the current address in DR - i.e. it points to the byte immediately to the left of the position in the (DR) string where the digit is inserted. This causes SF to be incremented by two words. (A similar effect is achieved by the Start significance instruction.)

The operation terminates after unpacking L digits. If after unpacking the last digit, CC = 2 or 3, the sign digit of ACC is inspected, and CC is set to 2 or 3 depending on whether the sign is positive or negative. If CC = 0 or 1 it is unaltered and the sign is ignored. If L = 0, and none of the other checks fail, a null operation is performed leaving ACC, CC and DR unaltered.

OV is cleared. ACS may be 32, 64 or 128 bits.

With the 32-bit instruction format the mask byte is ignored (reserved)

The result of the instruction is undefined if $L \geqslant 128$.

Notes: (a) After a number has been completely unpacked ACC will contain no non-zero digits, and only by testing CC can it be ascertained whether the number was positive, negative or zero.

(b) If the descriptor which is stacked when the first non-zero digit is

unpacked is not used for sign insertion (e.g.) it must be removed from the stack anyway.

CC : 0 CC was 0, and all digits unpacked were 0's.

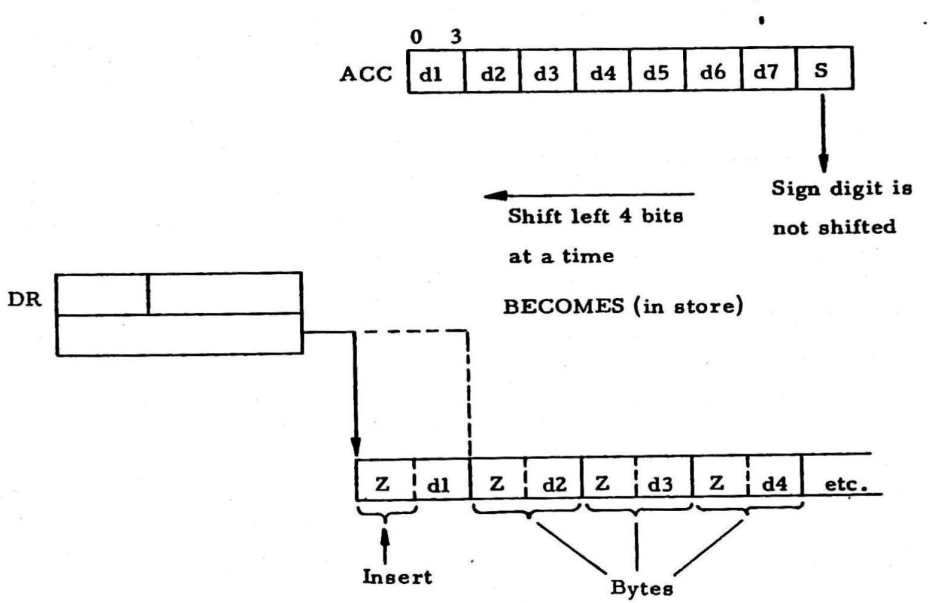
1 CC was 1, and all digits unpacked were 0's.

2 CC was 2 or 3, or some non-zero digits unpacked. Sign positive.

3 CC was 2 or 3, or some non-zero digits unpacked. Sign negative.

Program errors : Any failures of standard checks 1 and 2. (see 10.1).

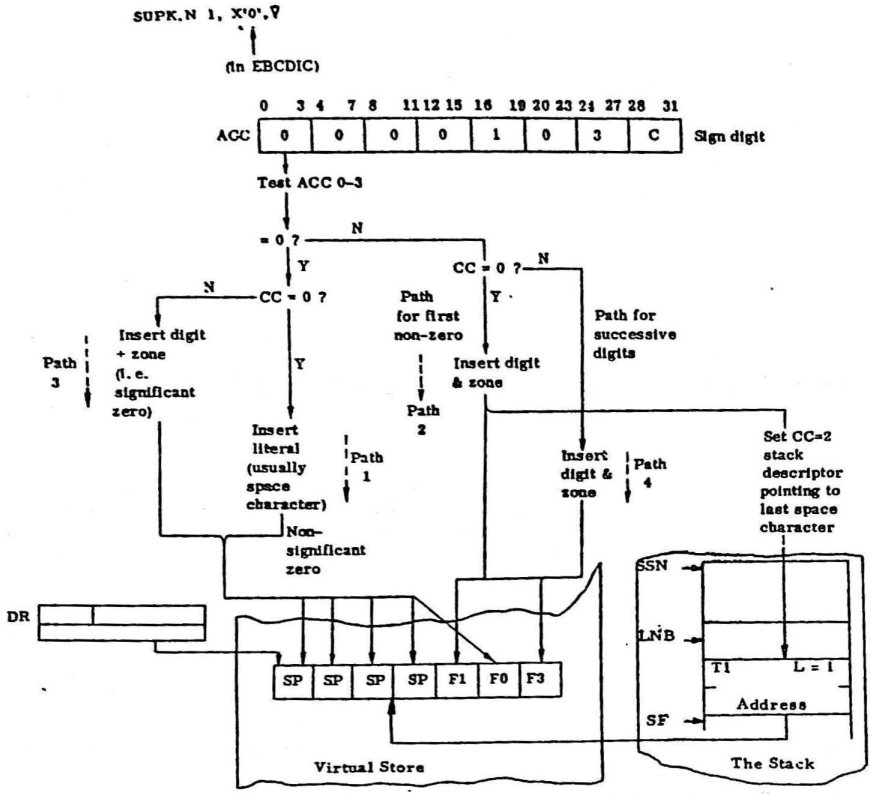
Fig. 63 SUPPRESS AND UNPACK (SUPK) (#94) BASIC OPERATION



0011 (i.e. #3) for ISO

1111 (i.e. #F) for EBCDIC

Fig. 64. SUPPRESS AND UNPACK (SUPK) DETAILED OPERATION



10.15 Detailed Operation

Fig 64 shows the mechanism of the instruction in more detail. Bits 0-3 of ACC are examined - if they are zero, then CC is checked. This should be zero if no significant zeroes have been encountered. In this situation a literal (usually a space character) is inserted (i.e. path 1 is followed) DR is updated and ACC is shifted down 4 bits. The procedure is repeated until the first non-zero character is reached.

ACC is tested, found to be non-zero, CC is examined - this is still zero, so path 2 is followed. The digit is inserted, prefixed by the zone code. Then condition code is set to 2 and a descriptor is stacked pointing to the last space character.

The next packed digit is accessed, in the example shown, this is a zero - However it is a significant zero. CC is examined. This is now non-zero, since it was set to a value of two on the last loop, so path 3 is taken. The zero prefixed by the zone code is inserted into store.

The next digit examined is 3 (i.e. non-zero), CC $\neq$ 0 so path 4 is followed - the digit and zone-code are inserted. The operation terminates after L digits have been unpacked. If CC = 2 or 3, the sign digit of ACC is inspected at this stage and CC is set to 2 or 3 depending on whether the sign was positive or negative respectively.

Incidentally the mode bit in SSR determines the zone code to be inserted - if it is a one, then the ISO code# 3 is used. For a zero, the EBCDIC code # F is inserted.

Summarising the CC settings of the SUPPRESS & UNPACK instruction:

CC = 0, CC was zero and all the digits unpacked were zeros (path 1)

CC = 1 CC was originally a one and that all digits unpacked were zeros (path 3)

CC = 2 Either CC was originally two (path 4) or that some non-zero digits were unpacked (path 2). Sign was positive.

CC = 3 Either CC was three, (path 4) or some non-zero digits were unpacked (path 2). Sign was negative.

10.16 Table Translate (TTR) # A6 (see fig 65)

Description : The descriptor in ACC points to a translation table. The descriptor must be of type 0, with size code 8 bits, and a valid bound; USC and BCI must not be set. Each byte in the (DR) string is replaced by a translation byte, obtained by using the byte as a modifier for the base address in ACC to access the required translation byte. Thus if the byte address in ACC is A, and the value of a (DR) string byte is 11011011 (=219), the latter is replaced by the contents of byte location A +219. An interrupt occurs if the value of any (DR) string byte (219 in the above example) is not less than the bound field of the descriptor in ACC. (ACC) is unaltered. If L = 0, and none of the other checks fail, a null operation is performed, leaving DR unaltered. Either 16 or 32-bit forms may be specified but for 32 bits the mask and literal bytes are ignored (reserved). If Program Mask bit 5 (Bound Check) is set, the table is assumed to be 256 bytes long.

CC : Unaltered.

Program errors : Any failures of standard checks 1-3 (see 10.1)
(DR) string byte ≥ bound field of descriptor in ACC. (see 12.2/5.8)

Fig 65 TABLE TRANSLATE (TTR) (#A6)

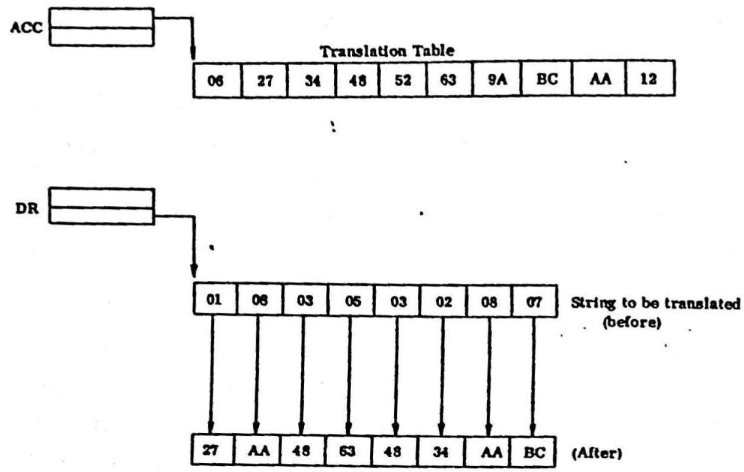
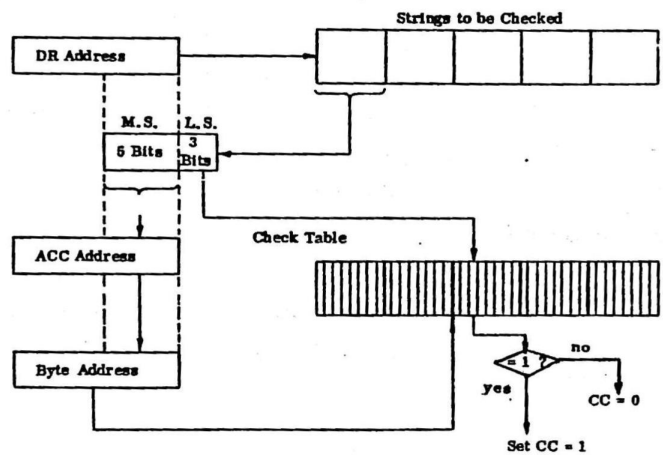


Fig 66 TABLE CHECK (TCH) (#80)



Note:- The check table is normally 32 bytes x 8 bits long. Max. 256 bits.

The first byte to be translated in the example shown is # 01 - using this value as a modifier the byte containing # 27 is accessed. This value replaces # 01 in the (DR) string. The next byte in the (DR) string is # 08 - using it as a modifier, the value # AA replaces # 08 in the (DR) string.

A use of this instruction would be for conversion between the EBCDIC and ISO character codes.

10.17 Table Check (TCH) # 80 (see fig 66)

Description : The descriptor in ACC points to a table of check bits. This descriptor must be of type 0, with size code 1 bit, and a valid bound; USC and BCI must be zero. Successive bytes in the (DR) string are checked against this table in the following way; the more significant 5 bits of the byte are used as a modifier of the address in the (ACC) descriptor to reference a byte in the table, and the least significant 3 bits to refer to one of the bits (numbered 0-7) in that byte. Thus, if the value of the byte is 10010101 (=8,x 18 + 5) and the byte address in ACC is A, the check-bit is bit 5 of byte (A + 18). If the value of the byte is not less than the bound field of the descriptor in ACC, an interrupt occurs. Processing of the (DR) string, from left to right, continues until the string is exhausted, or a byte whose check bit is 1 is found; in the latter case (DR) is left pointing to that byte. CC is used to indicate the reason

for termination. (ACC) is unaltered. The (DR) string is unaltered.

16 - or 32-bit instruction forms are permitted. In the 32-bit form, the mask and literal bytes are ignored (reserved). If L = 0, and none of the other checks fail, a null operation is performed, leaving DR unaltered, but leaving CC = 0.

If program Mask bit 5 (Bound Check) is set, the table is assumed to be 32 bytes long.

CC : 0 No non-zero check bit found
1 Non-zero check bit found
2 Not Used
3 Not Used

Program errors : Any failures of standard checks 1-3. (see 10.1)
(DR) string byte > bound field of descriptor in ACC. (see 12.2/5.8)

10.18 Conditional Insert (INS) #92

Description : This instruction is similar to Move literal (10.5) in that the source byte overwrites successive bytes of the (DR) string. However the source byte is defined differently, as follows:

CC = 0 Literal (bits 24-31 of B, if 16-bit format)

CC ≠ 0 Mask (bits 16-23 of B, if 16-bit format)

Each (DR) string byte is completely over-written.

If L = 0, and none of the other checks fail, a null operation is performed, leaving DR unaltered.

This instruction is intended for use with

		Suppress & Unpack (10.14).
CC	:	Unaltered.
Program errors	:	Any failures of standard checks 1 and 2. (See 10.1).

Note that three of the instructions in the following section (TEST, SET and CLR) are illegal at AML0.

In addition, these instructions disappeared from some later documentation, and may not have been implemented at all.

11 MISCELLANEOUS FUNCTIONS

This section covers the instructions not dealt with elsewhere. The first two instructions are used for bridging purposes between 1900 series and 2900 machines. They convert 6 bit data to or from 8 bit data.

11.1 Compress ACC (COMA)#98

Expand ACC (EXPA)#88 (see fig. 67)

These instructions use the primary format described in Chapter 1.

Operand length : Not applicable, literal must be specified.

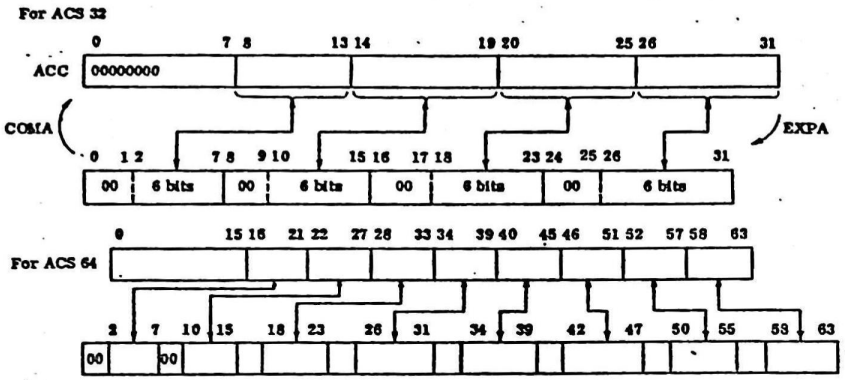
Description : These instructions require ACS = 32 or 64 bits. They convert the contents of ACC between an unpacked and a packed form by manipulation of fields as follows:

Packed Form (ACS = 32)	Packed Form (ACS = 64)	Unpacked Form
Bits 8 - 13	Bits 16 - 21	Bits 2 - 7
14 - 19	22 - 27	10 - 15
20 - 25	28 - 33	18 - 23
26 - 31	34 - 39	26 - 31
	40 - 45	34 - 39
	46 - 51	42 - 47
	52 - 57	50 - 55
	58 - 63	58 - 63

Compress ACC converts from unpacked form to packed form, ignoring the original contents of bits 0, 1, 8, 9, etc. of ACC, and generating zeros in bits 0-7 or 0-15, depending on ACS.

Expand ACC converts from packed form to unpacked form, ignoring the original contents of bits 0-7 or 0-15, and generating zeros in bits 0, 1, 8, 9, etc. ACS is unchanged. OV is cleared.

Fig. 67 COMPRESS ACC (COMA) (#98) and EXPAND ACC (EXPA) (#88)



CC : Unaltered.

Program errors : ACS = 128 bits (see 12.9/13.8)

11.2 Increment and Test (INCT) #56

Test and Decrement (TDEC) #54

Operand length : 32 bits

Description : Increment and test causes 1 to be added to the operand in store, and CC to be set according to the value of the result of that addition. Between reading the original operand value and replacing it by the incremented value, access to the operand location is prevented by hardware.

Test and decrement causes CC to be set according to the original value of the operand, and 1 to be subtracted from it. Between reading the original operand value and replacing it by the decremented value, access to the operand location is prevented by hardware.

In both cases the operand is effectively incremented or decremented in situ; ACC and OV are unaltered and overflow is ignored.

The following restrictions apply:
a) Access to the operand is forced by hardware to bypass slave storage.
b) The operand must be located in the store rather than a register. Direct TOS and (PC+N) operand forms are not permitted.
c) If the operand is accessed indirectly it may only be via a vector (type 0) descriptor with size code 32 bits.

If slave storage is present in the processor these instructions are also required to clear the operand slave store of items from segments marked non-slaved (NS) in either segment table. The segment containing the operand need not be cleared from slave stores unless marked NS.

Note: The operand value will usually be interpreted as the number of other processes waiting to use a shared resource with -1 indicating 'available'.

CC : 0 Operand zero
1 Operand > 0
2 Operand < -1
3 Operand = -1
(here 'operand' refers to the final value for Increment and test, the original value for Test and decrement).

Program errors : Operand addressing errors
Incorrect descriptor type (see 6/10.13)
Incorrect operand types; Literal, IS, TOS, B, (PC+N) (See 12.8/12.2)

11.3 Activate (ACT) # 3E (see fig. 68)

Operand length : 128 bits

Description : The first two words of the operand are loaded to LSTB. Bits 14-30 of the first word, and 0-3 and 29-31 of the second word are ignored (spare). The third word is ignored (spare). Bits 0-13 of the fourth word contain the new value of SSN; bits 14-31 are ignored (spare). After loading LSTB the action of the instruction is to generate the base address of segment (SSN + 1) (effectively by concatenating the bit pattern 10...0 with bits 0-12 of the fourth word of the operand; if the segment number is in the range 0-8191, the new LSTB is used to translate it) and unload the contents of words 0-15 of that segment (the 'process state') to the appropriate OCP registers, the reverse of the stack-switching interrupt process. In emulating machines, the E and M bits of the new PSR and SSR will be examined after unloading words 0-7, and subsequent action will depend on their values.

Virtual addressing mode is assumed throughout. Methods of changing addressing mode are implementation-defined. If there is disagreement between the values of SSN specified by the operand and in words 0 and 4 of the process state the result is undefined. A system error interrupt may occur if an attempt is made to load an odd segment number to SSN.

The new process defined by the undumped process state is entered at the instruction specified by PC, qualified if necessary by the setting of II. In emulating machines, if E = 1 in new PSR, and EM in new SSR has a locally valid value, alien code is executed.

If bit 31 of the first word of the operand is a 1, and if the EP interrupt mask bit is not set in the new SSR, an Event Pending interrupt occurs on resumption of the process. This may occur before the registers have been completely undumped and, if II is set, it will occur before attempting completion of the uncompleted instruction. The EP bit in SSR is ignored and is not cleared.

PSTB is not altered. Execution of Activate may cause the new value of IC to be decremented.

The instruction requires PRIV = 1 to be executed.

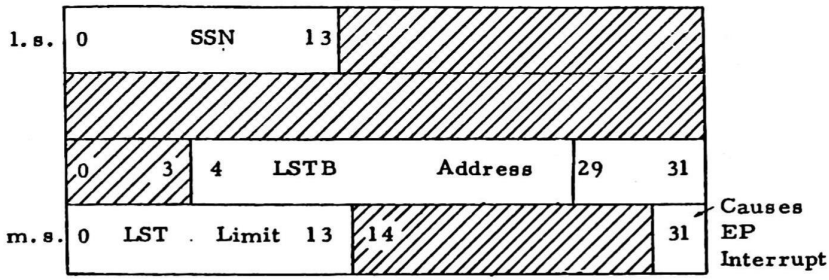
CC : As specified in new PSR

Program errors : Operand addressing errors
Privilege (see 12.5/9.6)
(Emulating machines; PEI 14). New E = 1 and EM = 0 or invalid value.

(Note: Masking of program error interrupts is controlled by the program and interrupt mask bits in PSR and SSR at the beginning of the instruction. The occurrence of an unmasked program error during the initial stage of Activate will prevent the loading of LSTB, SSN and the other processor registers (i.e. effectively it is the old process, not the new one, being interrupted). However, in the case of an error in switching to emulation, implementation defined action will occur.

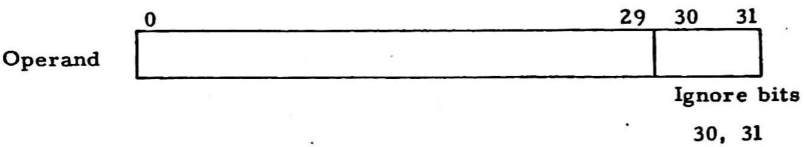
Fig 68 ACTIVATE (ACT) (#3E)

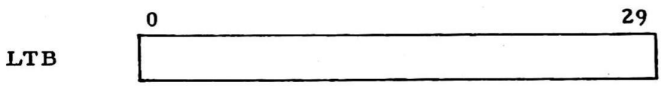
Operand of Activate Instruction



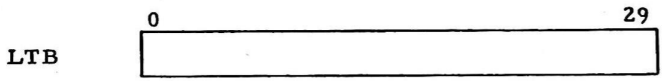
Generates Base Address of (SSN + 1) by concatenating SSN bits 0 - 12 with the bit pattern 100.....00 (Note SSN is always even.

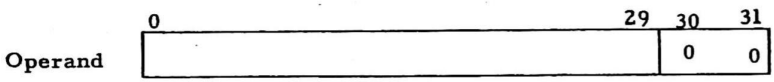
Fig 69 LOAD CTB (LCT) #30





STORE CTB (STCT) #36





11.4 Load CTB (LCT) # 30 (see fig 69)

Operand length	:	32 bits
Description	:	Bits 0-29 of the operand are loaded to CTB. Bits 30, 31 are ignored.
CC	:	Unaltered.
Program errors	:	Operand addressing errors

Store CTB (STCT) # 36

Operand length	:	32 bits
Description	:	The contents of CTB, expanded to a 32-bit byte address by concatenating two zero bits on the right, is stored.
CC	:	Unaltered.
Program errors	:	Operand addressing errors Literal operand (see 12.8/12.1) Non-zero bits of stored item truncated (see 12.3/6.0)

11.5 Test (TEST) #08

This function is illegal in AML0

Operand length : 32 bits

Description : The operand is fetched and used to set CC. The ACC and OV are unaltered.

CC settings :
0 Operand = 0
1 Operand > 0
2 Operand < - 1
3 Operand = - 1

Program Errors : Operand addressing error.

Note: If the operand is accessed indirectly via a vector descriptor size 6 or 7 (64 or 128 bits) only the l.s. 32 bits are used.

11.6 Clear (CLR) #0C
Set (SET) #0E

These functions are illegal in AML0

Operand length : 32 bits

Description : These instructions are provided for clearing or setting store locations without affecting the contents of the accumulator. In effect, a pseudo 32 bit accumulator is created, containing all-zeros if CLR or all-ones if SET. The contents of this ACC are stored in the operand location which is thus set to all zeros or all-ones.

CC : Unaltered.

Program Errors : Operand addressing errors.
Literal operand
Size error

Notes: With function SET

1) Use of a byte string descriptor with a length of 1-3 bytes will cause a size error (non-zero bits truncated).

2) Bit string operand formats should have the S field set to 1, otherwise a size error will occur.

3) Use with a vector descriptor for a double or a quadruple word item will only set the l.s. 32 bits to all-ones.

12 PROGRAM ERRORS

12.1 Interrupt Parameter

Each Program Error is associated with an interrupt parameter. See fig. 69.

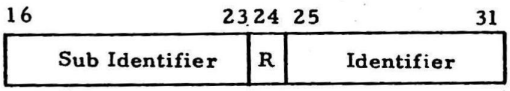
P.E.I. numbers are assigned as follows:-

Code 0 Floating point overflow
1 Floating point underflow
2 Fixed point overflow
3 Decimal overflow
4 Zero divide
5 Bound check
6 Size
7 B overflow
8 Stack
9 Privilege
10 Descriptor
11 String
12 Instruction
13 Accumulator
14 ESR errors (emulating machines only).

Codes 0 to 4 and 7 have no sub-identifiers and therefore bits 16-23 are all 1's for these program errors.

The program errors are now listed in Identifier and Sub-identifier order, except for cases where no sub-identifiers exist (0 - 4 & 7).

Fig. 69. Program Error Interrupt Parameter



Bits 16-23 = Sub Identifier, implementation defined (= all 1's if no sub-identifier).

Bit 24 = Restart bit. If bit 24=1 indicates restart is not possible.

Bits 25-31 = P.E.I. (Program Error Identifier). Range defined.

12.2 Bound

5.0 Descriptor bound check-modifier (unsigned) too large.

5.1 Modify DR-operand (unsigned) $\geqslant$ DR bound.

5.2 Dope vector multiply: (Index-lower bound) overflows.

5.3 Dope vector multiply: (Index-lower bound) negative.

5.4 Dope vector multiply: Multiplier negative.

5.5 Dope vector multiply: Upper bound negative.

5.6 Dope vector multiply: Displacement (product) $\geqslant$ upper bound. or $\geqslant 2^{31}$.

5.7 Dope vector multiply: DR bound goes negative.

5.8 Table check, translate, (DR) string byte too large.

12.3 Size (Location too small for operand)

6.0 Significant part of operand truncated.

6.1 Store to register, descriptor size code (types 0 or 2) > operand length (except for jumps).

12.4 Stack (Stack register operation check)

8.0 Unstacking operation makes SF $\leqslant$ LNB (SF unaltered).

8.1 Undefined.

8.2 Load LNB and Exit, bits 0-13 of new LNB $\neq$ SSN (LNB unaltered).

8.3 Load LNB and Exit, new LNB $\geqslant$ SF

8.4 Raise LNB, new LNB $>$ SF (LNB unaltered).

8.5 Raise LNB, new LNB < old LNB (LNB unaltered).

8.6 Adjust SF, new SF $\leqslant$ LNB (SF unaltered).

8.7 Adjust SF, segment overflow (SF unaltered).

12.5 Privilege (A program error leads to an attempt to use a resource which the current level of privilege does not justify).

9.0 Read protection fail with ACR

9.1 Write protection fail with ACR.

9.2 Execute when execute permission bit not set.

9.3 Use of image store without privilege permission.

9.4 Use of non-existent image store (e.g. address does not exist, or read to write only IS or write to read only IS).

9.5 ACR < old ACR or PRIV > old PRIV on Exit.

9.6 Activate executed without privilege permission.

12.6 Descriptor

10.0 Jump descriptor not type 0, size 32 or 64, type 2, escape code,

10.1 Descriptor for normal operand access not type 0, 1, 2 or escape type.

10.2 Call descriptor is not type 0, size 32 or 64, type 2, code, escape or System Call.

10.3 Link descriptor for Exit is not code, escape, or system call.

10.4 Descriptor in DR not type 0, size 32 for Dope Vector multiply.

10.5 Length in type 1 descriptor used by primary format instruction is zero or exceeds operand length.

10.6 DR descriptor is not type 0 or 1 with size code 3, or escape, for store-to-store operation.

10.7 ACC descriptor not type 0 or 1 with size code 3, for some store-to-store operations.

10.8 ACC descriptor not type 0, size 1 or 8, for Table check and Table translate, respectively.

10.9 Size code incorrect in type 0 descriptor.

10.10 Descriptor sub-type undefined.

10.11 Modifier DR with System Call descriptor in DR.

10.12 (Not Assigned)

10.13 Incorrect descriptor used for semaphore instruction.

10.14 Reserved for emulating machines.

12.7 String

11.0 L > DR bound.

11.1 (Not Assigned)

11.2 L > ACC bound for 16 bit form of Move, Compare, And, Or and Not equivalent strings. Check overlap.

12.8 Instruction

12.0 Instruction function is illegal or unassigned.

12.1 Store to literal.

12.2 Indirect address form for certain functions (e.g. Increment & test, Modify DR, Load relative).

12.3 Relative jump attempts to alter segment number in PC.

12.4 Unassigned operand address forms k" = 7 and k' = 1 or k" = 1 (AML0 only).

12.5 Item addressed in stack segment lies above TOS.

12.6 Item addressed by PC +N lies outside current code segment.

12.7 Normal update of PC attempts to alter segment number.

12.8 Reserved for emulating machines.

12.9 Accumulator (ACC incompatible with instruction)

13.0 ACS 128 bits and fixed point or logical. (except Multiply Double).

13.1 ACS 64 bits and Add/Subtract logical.

13.2 AGS = 128 bits and Float.

13.3 ACS = 32 bits and Floating divide double.

13.4 ACS = 128/32 bits and Load/Store upper half.

13.5 ACS = 128 bits, or 64 if fixed, and Multiply double.

13.6 ACS ≠ 64 bits for store-to-store instructions involving descriptor in ACC.

13.7 Modify PSR or Exit attempts to set ACS = 0.

13.8 ACS = 128 bits and Compress/Expand ACC.

12.10 Operand Addressing Errors

The phrase 'operand addressing errors' is used throughout to cover errors of the following types:

Bound check (class 0)

Size (class **1)

Privilege (classes 0, **1, 3, 4)

Descriptor (classes *0, **1, *2, **5, 9, 10)

Instruction (classes *3, 4, 5, 6)

* JUMPS ONLY, ** NOT JUMPS

Other program errors are listed explicitly with each instruction description, except for errors which are checked as part of the

APPENDIX 1

LIST OF INSTRUCTIONS IN MNEMONIC ALPHABETICAL ORDER

Mnemonic	Name	Section	Function Code (Hex)
ACT	Activate	11.3	3E
ADB	Add to B	7.4	20
AND	And	9.27	8A
ANDS	And strings	10.9	82
ASF	Adjust SF	4.6	6E
CALL	Call	6.8	1E
CBIN	Convert to binary	9.43	DE
CDEC	Convert to decimal	9.44	EE
CHOV	Check overlap	10.11	B4
CLR	Clear	11.6	0C
COMA	Compress ACC	11.1	98
CPB	Compare B	7.5	26
CPIB	Compare & increment	7.6	2E
CPS	Compare strings	10.8	A4
CPSR	Copy PSR	5.11	34
CYD	Copy DR	5.6	12
DAD	Decimal Add	9.32	D0
DCP	Decimal compare	9.34	D6
DDV	Decimal divide	9.38	9A
DEBJ	Decrement B & jump if non-zero	6.5	24
DMDV	Decimal remainder divide	9.40	9E
DMY	Decimal multiply	9.36	DA
DMYD	Decimal multiply double	9.37	DC
DRDV	Decimal reverse divide	9.39	9C
DRSB	Decimal reverse subtract	9.33	D4
DSB	Decimal subtract	9.32	D2
DSH	Decimal shift	9.35	D8
ESEX	Escape exit	6.10	3A
EXIT	Exit	6.9	38
EXPA	Expand ACC	11.1	88
FIX	Fix	9.41	B8

Mnemonic	Name	Section	Function Code (Hex)
FLT	Float	9.42	A8
IAD	Integer add	9.13	E0
ICP	Integer compare	9.14	E6
IDLE	Idle	6.7	4E
IDV	Integer divide	9.18	AA
IMDV	Integer remainder divide	9.20	AE
IMY	Integer multiply	9.16	EA
IMYD	Integer multiply double	9.17	EC
INCA	Increment address	8.9	14
INCT	Increment & test	11.2	56
INS	Conditional insert	10.18	92
IRDV	Integer reverse divide	9.19	AC
IRSB	Integer reverse subtract	9.13	E4
ISB	Integer subtract	9.13	E2
ISH	Arithmetic shift	9.15	E8
J	Jump	6.1	1A
JAF	Jump on arith. - condition false	6.4	06
JAT	Jump on arith. - condition true	6.4	04
JCC	Jump on CC	6.3	02
JLK	Jump and link	6.2	1C
L	Load	5.1	60
LB	Load B	7.1	7A
LCT	Load CTB	11.4	30
LD	Load DR	8.1	78
LDA	Load address	8.5	72
LDB	Load bound	8.7	76
LDRL	Load relative	8.2	70
LDTB	Load type & bound	8.6	74
LLN	Load LNB	4.2	7C
LSD	Set ACS 64 & load	5.2	64

Mnemonic	Name	Section	Function Code (Hex)
LSQ	Set ACS 128 & load	5.2	66
LSS	Set ACS 32 & load	5.2	62
LUH	Load upper half	5.4	6A
LXN	Load XNB	4.5	7E
MODD	Modify DR	8.8	16
MPSR	Modify PSR	5.10	32
MV	Move	10.3	B2
MVL	Move literal	10.5	B0
MYB	Multiply B	7.4	2A
NEQ	Not equivalence	9.27	8E
NEQS	Not equivalence strings	10.9	86
OR	Or	9.27	8C
ORS	Or strings	10.9	84
OUT	Out	6.6	3C
PK	Pack	10.13	90
PRCL	Precall	4.8	18
RAD	Floating add	9.3	F0
RALN	Raise LNB	4.4	6C
RCP	Floating compare	9.5	F6
RDV	Floating divide	9.9	BA
RDVD	Floating divide double	9.11	BE
RMY	Floating multiply	9.7	FA
RMYD	Floating multiply double	9.8	FC
ROT	Rotate	9.28	CA
RRDV	Floating reverse divide	9.10	BC
RRSB	Floating reverse subtract	9.4	F4
RRTC	Read real-time clock	5.7	68
RSB	Floating subtract	9.3	F2
RSC	Scale	9.6	F8
SBB	Subtract B	7.4	22
SET	Set	11.6	0E

Mnemonic	Name	Section	Function Code (Hex)
SHS	Shift 32 bits	9.29	CC
SHZ	Shift while zero	9.30	CE
SIG	Start significance	8.10	28
SL	Stack & load	5.8	40
SLB	Stack & load B	7.2	52
SLD	Stack & load DR	8.3	50
SLSD	Stack, set ACS 64 & load	5.9	44
SLSQ	Stack, set ACS 128 & load	5.9	46
SLSS	Stack, set ACS 32 & load	5.9	42
ST	Store	5.3	48
STB	Store B	7.3	5A
STCT	Store CTB	11.4	36
STD	Store DR	8.4	58
STLN	Store LNB	4.3	5C
STSF	Store SF	4.7	5E
STUH	Store upper half	5.5	4A
STXN	Store XNB	4.5	4C
SUPK	Suppress and unpack	10.14	94
SWEQ	Scan while equal	10.6	A0
SWNE	Scan while unequal	10.7	A2
TCH	Table check	10.17	80
TDEC	Test & decrement	11.2	54
TEST	Test	11.5	08
TTR	Table translate	10.16	A6
UAD	Logical add	9.22	C0
UCP	Logical compare	9.25	C6
URSB	Logical reverse subtract	9.24	C4
USB	Logical subtract	9.23	C2
USH	Logical shift	9.26	C8
VAL	Validate address	8.11	10
VMY	Dope vector multiply	7.9	2C

APPENDIX 2

Functional Grouping of 2900 Orders

JUMPS

Mnem	Hex Code	Type	
J	1A	P	Jump unconditionally. PC'=PC + LIT or OP_{0-30}
DEBJ	24	P	Decrement B & Jump if non-zero. B'=B-1, jump if B'≠ 0
JCC	02	T	Jump on CC. PC'=PC + LIT or OP_{0-30} if bit (CC) of M=1

M =	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
JCC	-	3	2	2	1	1	1	1	0	0	0	0	0	0	0	A
can				3		3	2	2		3	2	2	1	1	1	N
be								3				3		3	2	Y

JAT	04	T	Jump on arithmetic true. jump if 'M' is true.
JAF	06	T	Jump on arithmetic false. jump if 'M' is false.

Where with floating M=0 ACC=0, M=1 ACC > 0, M=2 ACC < 0
with fixed M=4 ACC=0, M=5 ACC > 0, M=6 ACC < 0
with decimal M=8 ACC=0, M=9 ACC > 0, M=10 ACC<0
with B M=12, B=0, M=13, B > 0, M=14, B < 0
and M=11, DR BOUND =0, M=15, OV=1

JLK	1C	P	Jump & link. TOS'=PC, PC'=PC + LIT or OP_{0-30}
CALL	1E	P	Procedure call. (L+1)' = descriptor of PC and jump
EXIT	38	P	Procedure exit. stack returned to status quo, jump to link
ESEX	3A	P	Escape exit, PC'=TOS, D'=1 with descriptor in DR

DR INSTRUCTIONS

LD	78	P	Load DR. DR'=OP_{0-63}, CC'= descriptor type
LDRL	70	P	Load relative. DR'=OP_{0-63} + OP address, CC'=descriptor type
LDA	72	P	Load address. DR'_{32-63} =OP_{0-31}
LDTB	74	P	Load type and bound. DR'_{0-31} =OP_{0-31}
LDB	76	P	Load bound. DR'_{8-31} =OP_{8-31}
SLD	50	P	Stack & load DR. TOS'=DR, DR'=OP_{0-63}, CC'=descriptor type
STD	58	P	Store DR. OP'_{0-63} = DR

INCA	14	P	Increment address. $DR'_{32-63}=DR_{32-63}+OP_{0-31}$
MODD	16	P	Modify DR. $DR'_{8-31}=DR_{8-31}-OP_{8-31}$, $DR'_{32-63}=DR_{32-63}+OP_{0-31}$ scaled.
VAL	10	P	Validate address. check validity of DR by OP_{8-11} as ACR OK CC'=0, read only CC'=1, write only CC'=2 invalid CC'=3.
SIG	28	P	Start significance. If CC'=0, OP_{0-63}= type 1 descriptor of length = 1. address = address -1 and CC set to 1. If CC ≠ 0, do nothing

B INSTRUCTIONS

LB	7A	P	Load B. $B'=OP_{0-31}$, OV'=0
SLB	52	P	Stack & Load B. TOS'=B, $B'=OP_{0-31}$, OV'=0
STB	5A	P	Store B. $OP'_{0-31}=B$, OV'=0
ADB	20	P	Add to B. $B'=B+OP_{0-31}$, OV'=overflow
SBB	22	P	Subtract from B. $B'=B-OP_{0-31}$, OV'=overflow
MYB	2A	P	Multiply B. $B'=B.\ OP_{0-31}$, OV'=overflow
CPB	26	P	Compare B. $B=OP_{0-31}$, CC'=0, B < OP, CC'=1, B > OP, CC'=2
CPIB	2E	P	Compare B increment B. as CPB & B'=B + 1
VMY	2C	P	Dope vector multiply. $B'=(OP_{0-31}-x).\ y$, check $0 \leq B' < z$ & $0 < (OP - x) < 2^{31}$, x, y, z = (DR), (DR + 1), (DR + 2).

STACK CONTROL

LLN	7C	P	Load LNB. $LNB'=OP_{14-29}$, OP_{0-13} must equal SSN
STLN	5C	P	Store LNB. OP'_{0-31} = SSN/LNB
RALN	6C	P	Raise LNB. $LNB'=SF-OP_{0-31}$
ASF	6E	P	Adjust SF. $SF+OP_{0-31}$, OP may be -ve
LXN	7E	P	Load XNB. $XNB'=OP_{0-29}$
STSF	5E	P	Store SF. OP'_{0-31} = SSN/SF
STXN	4C	P	Store XNB. OP'=XNB
PRCL	18	P	Precall, Stack SSN + LNB & ASF

SUNDRIES

OUT	3C	P	Out. Cause class 9 interrupt with OP_{0-31} as parameter.
IDLE	4E	P	Idle. Suspend instruction sequencing until interrupt occurs
ACT	3E	P	Activate. OP_{0-63} = LSTB; OP_{64-95} is spare; OP_{96-110} = SSN. Load OCP registers with contents of SSN + 1 and start process.
LCT	30	P	Load CTB. CTB' = OP_{0-29}
STCT	36	P	Store CTB. OP' = LCB
* TEST	08	P	Test operand. ACC & OV unaltered.
* CLR	0C	P	Clear store location. ACC unaltered.
* SET	0E	P	Set store location. ACC unaltered.

STRING INSTRUCTIONS

SWEQ	A0	S	Scan while equal. LIT ≠ (DR) stg, not found CC'=0 (DR) > ref. CC'=2. (DR) < ref. CC'=3.
SWNE	A2	S	Scan while unequal, LIT ≡ (DR) stg, not found CC'=0 else 1
CPS	A4	S	Compare strings. (DR) = (ACC), CC'=0 | (DR) > (ACC), CC'=2 | (DR) < (ACC), CC=3
MV	B2	S	Move. (DR) string = (ACC) string
CHOV	B4	S	Check overlap. none, CC'=0 | (ACC) > (DR), CC'=1 | (ACC) < (DR), CC'=2
MVL	B0	S	Move literal. (DR)' string = literal
TCH	80	S	Table check. check (DR) stgs with bit in (ACC) stg and stop if bit = 1 setting CC'=1, else CC'=0
TTR	A6	S	Table translate. (DR) string is translated by (ACC) stg.
PK	90	S	Pack. ACC' = (DR) string, packed. OV = 0
SUPK	94	S	Suppress & unpack. (DR)' string = ACC unpacked, CC is set
INS	92	S	Conditional insrt. (DR)' string = LIT (CC=0) or mask (CC≠0)
ANDS	82	S	And string. (DR)' stg. = (SR) stg. & (ACC) stg. or literal
ORS	84	S	Or string. (DR)' stg = (DR) stg. v (ACC) stg. or literal
NEQS	86	S	Not equivalent string. (DR)' stg = (DR) stg ≠ (ACC) stg. or literal.

SEMAPHORE INSTRUCTIONS

INCT 56 P Increment and test OP'=OP + 1

TDEC 54 P Test and decrement OP'=OP - 1
OP=0, CC=0 | OP>0, CC=1 | OP< -1, CC=2 | OP= -1, CC=3
(OP=final value for INCT and original value for TDEC).

ACC INSTRUCTIONS

L 60 P Load. $ACC'=OP_{ACS}$, $OV'=0$

LSS 62 P Set ACS 32 & load. ACS'=32, $ACC'=OP_{0-31}$, $OV'=0$

LSD 64 P Set ACS 64 & load. ACS=64, $ACC'=OP_{0-63}$, $OV'=0$

LSQ 66 P Set ACS 128 & load. ACS=128, $ACC'=OP_{0-127}$, $OV'=0$

LUH 6A P Load upper half, ACS'=2ACS, ms $\frac{1}{2}$ of $ACC'=OP_{ACS}$, $OV'=0$

CYD 12 P Copy DR. ACS'=64, ACC'=DR, OV'=0

SL 40 P Stack & load. $TOS'_{ACS}=ACC$, $ACC'=OP_{ACS}$, $OV'=0$

SLSS 42 P Stack, set ACS 32 & load. TOS_{ACS} = ACC, ACS'=32, $ACC=OP_{0-31}$, $OV'=0$

SLSD 44 P Stack, set ACS 64. & load. TOS_{ACS}=ACC, ACS'=64, $ACC=OP_{0-63}$, $OV'=0$

SLSQ 46 P Stack, set ACS 128 & load. TOS_{ACS}=ACC, ACS'=128, $ACC=OP_{0-127}$, OV=0

ST 48 P Store. OP_{ACS}=ACC, OV'=0

STUH 4A P Store upper half. $OP'_{\frac{1}{2}ACS}$= ms $\frac{1}{2}$ of ACC, ACS'=$\frac{1}{2}$ ACS OV'=0

MPSR 32 P Modify PSR. If $OP_{27}=1$, $ACS'=OP_{30-31}$ If $OP_{26}=1$, $CC'=OP_{28-29}$ If $OP_{24}=1$, PM'=1's at OP_{16-23}=1's or $OP_{25}=1$, PM'=0's at OP_{16-23}=0's.

CPSR 34 P Copy PSR. OP_{0-15}=0 | OP_{16-23}= PM' | OP_{24-27}=1110 OP_{28-29}=CC' | OP_{30-31}=ACS'.

RRTC 68 P Read real time clock. ACC'_{0-63} = clock output.

LOGICAL (WORD)

UAD C0 P Logical add. ACC'=ACC + OP_{0-31}, CC'=carry, OV'=0

USB C2 P Logic subtract. AC'=ACC - OP_{0-31}, CC'=borrow, OV'=0

URSB C4 P Logical reverse subtract. ACC'=OP_{0-31} - ACC, CC'= borrow, OV'=0

UCP C6 P Logical compare. ACC=OP_{ACS}, CC'=0 | ACC < OP, CC'=1 ACC > OP, CC=2

USH C8 P Logical shift. Shift ACC left OP_{25-31} places or right if OP-ve, OV'=0

ROT CA P Rotate. Rotate ACC left OP_{25-31} places or right if OP-ve, OV'=0

SHS CC P Shift 32 bits. Shift ACC_{0-31} left OP_{25-31} places or right if OP-ve, OV'=0

SHZ CE P Shift while zero. Shift ACC left until ACC, ≠ 0, OP'= no. of places, OV'=0

AND 8A P And. ACC'=ACC & OP_{ACS}, OV'=0

OR 8C P Or. ACC'=ACC v OP_{ACS}, OV'=0

NEQ 8E P Not equivalent. ACC'=ACC ≠ OP_{ACS}, OV'=0

DECIMAL

DAD D0 P Decimal add. ACC'=ACC + OP_{ACS}, OV'=overflow

DSB D2 P Decimal subtract. ACC'=ACC - OP_{ACS}, OV'= overflow

DRSB D4 P Decimal reverse subtract. ACC'=OP_{ACS} - ACC, OV'=overflow

DMY DA P Decimal multiply. ACC'=ACC. OP_{ACS}, OV'= overflow

DMYD DC P Decimal multiply double. ACS'=2ACS, ACC'=ACC. OP_{ACS}, OV'=0

DDV 9A P Decimal divide. ACC'=ACC/OP_{ACS}, OV'=0

DMDV 9E P Decimal remainder divide. ACC'=ACC/OP_{ACS}, TOS_{ACS}= rem. OV'=0, rem=0 or rem > 0 & div > 0, CC'=0, | rem > 0 & div < 0, CC'=1, | rem < 0 & div > 0. CC'=2, | rem < 0 & div < 0, CC'=3

DRDV 9G P Decimal reverse divide. ACC'=OP_{ACS}/ACC, OV'=0

DCP D6 P Decimal compare. ACC=OP_{ACS}, CC'=0 | ACC < OP, CC=1 | ACC > OP, CC=2

DSH D8 P Decimal shift. Shift ACC left 4.OP places or right if -ve, OV'= overflow

CBIN DE P Convert to binary. ACC' (binary) = ACC (pk decimal), OV' = overflow

CDEC EE P Convert to decimal. ACS'=2 ACS, ACC' (dec) = ACC (bin), OV'=0

FIXED POINT (INTEGER)

IAD E0 P Add. ACC'=ACC + OP_{ACS}, OV' = overflow

ISB E2 P Subtract. ACC'=ACC - OP_{ACS}, OV' = overflow

IRSB E4 P Reverse. subtract. ACC'=OP_{ACS} - ACC, OV'= overflow

IMY EA P Multiply. AC'=ACC. OP_{ACS}, OV'= overflow

IMYD EC P Multiply double. ACS'=2ACS, ACC'=ACC. OP_{ACS}, OV'=0 ACC +ve, OP +ve, CC'=0, | ACC +ve, OP -ve, CC'=1, ACC -ve, OP +ve, CC'=2 | both -ve, CC=3

IDV AA P Divide. ACC'=ACC/OP_{ACS}, OV'= overflow

IRDV AC P Reverse divide. ACC'=OP_{ACS}/ACC, OV'= overflow

IMDV AE P Remainder divide. ACC'=ACC/OP_{ACS}, TOS'= rem, 'CC' as DMDV

ICP E6 P Compare. ACC=OP_{ACS}, CC'=0, | ACC < OP, CC'=1, ACC > OP, CC'=2

ISH E8 P Arithmetic shift. ACC'=ACC. 2^i, i = OP_{26-31}.
OV'= overflow i +ve, CC'=3 | i -ve, all last bits =0, CC'=0 | Last bit 0, CC'=1 | Last bit 1, CC'=2

FLOATING POINT (REAL)

RAD F0 P Floating add. ACC'=ACC + OP_{ACS}, OV'= overflow

RSB F2 P Floating subtract. ACC'=ACC - OP_{ACS}, OV'= overflow

RRSB F4 P Floating reverse subtract. ACC'=OP_{ACS} -ACC, OV'= overflow

RMY FA P Floating multiply. ACC'=ACC. OP_{ACS}, OV'= overflow

RMYD FC P Floating multiply double. ACS'=2ACS, ACC=ACC.OP_{ACS}, OV'= overflow

RDV BA P Floating divide. ACC'=ACC/OP_{ACS}, OV'= overflow

RRDV BC P Floating reverse divide. ACC'=OP_{ACS}/ACC, OV'= overflow

RDVD BE P Floating divide double. ACS'=2ACS, ACC'=ACC/OP_{ACS}, OV'= overflow

RCP F6 P Floating compare. ACC=OP_{ACS}, CC'=0 | ACC < OP, CC'=1 | ACC > OP, CC=2

RSC F8 P Scale. ACC'=ACC. 16^i. i=OP_{22-31}, OV'= overflow

FLT A8 P Float. ACS'=2 ACS, ACC'$_{FP}$=ACC int. i=OP_{22-31}(biased) OV'= overflow

FIX B8 P Fix. ACS'=$\frac{1}{2}$ ACS, ACC' int =ACC_{FP}, OP'=ACC exp (unbiased) OV'=0

EMULATION

COMA 98 P Compress ACC ACC' (6 bit chars) = ACC (8 bit bytes)

EXPA 88 P Expand. ACC ACC' (8 bit bytes) = ACC (6 bit chars)

* Instruction illegal for AML0.