UNIVERSITY OF MANCHESTER

DEPARTMENT OF COMPUTER SCIENCE

MU5 BASIC PROGRAMMING MANUAL

This document is an edited recreation of a copy of the MU5 Basic Programming Manual dating from 1975, parts of which date back to 1972 (at least). The original manuscript appears to be lost and the editor's copy, like those held in the special collections of the University of Manchester library, is in rather poor condition. The original was produced on a manual typewriter, and whereas this version was created using Latex and xfig, it has been designed to appear similar to the original. Some typographical and grammatical errors in the original have been corrected and some of the layout has been altered to improve self-consistency. A copy of a later (1978) edition of the Manual also exists, in a different typeface. The 1978 edition contains some corrections and updates but also includes some new errors. This reconstruction draws on both versions in an attempt to produce a more accurate description of MU5 as seen by its programmers. Any remaining errors are the fault of the editor.

Editor's notes and other additional material are shown in blue. Publication of this document is by kind permission of the University of Manchester School of Computer Science.

Thanks are due to Rob Jarratt for his careful proof reading but mainly for having inspired this reconstruction through his work on creating a software emulator for MU5[1].

Roland Ibbett
June 2017

Roland Ibbett is an Emeritus Professor of Computer Science, University of Edinburgh and formerly Reader in Computer Science at the University of Manchester. In 1979 he was co-author, with the late Professor Derrick Morris, of the "The MU5 Computer System", published by The Macmillan Press.

[1] https://robs-old-computers.com/projects/mu5/

CONTENTS

Chapter 1

1.1 Introduction

The organisation of the machine[2] is reflected in its order code which is essentially of the form:-

F N

where F defines the function and N the operand. There are four classes of orders:-

Computational orders

B-orders

Structure accessing and store-to-store orders

Organisational orders

In the computational orders, which are distinguished by a 1 in digit 0, the instruction is divided thus:-

1	cr	f	N
	2	4	9

The cr bits define one of four types of arithmetic:-

signed fixed-point

unsigned fixed-point

decimal

floating point

In MU5 the signed fixed-point operations use a 32-bit register X, while the unsigned fixed-point, floating-point and decimal operations share a common 64-bit register A. However, the structure of the instruction code allows for four separate registers. The f bits define the operation to be performed and N defines the operand. Computational orders are of the single address type (e.g. A = operand, A + operand).

[2] see: APPENDIX I. For more information go to
http://www.cs.manchester.ac.uk/about-us/history/mu5/
http://ethw.org/The_University_of_Manchester_MU5_Computer_System

The B-orders operate on the modifier register B. They have the form:-

```
| 001 |     f     |               N                |
----------------------------------------------------
            4                     9
```

The functions provided correspond to those which operate on X but the division orders are not implemented in MU5.

In the structure addressing and store-to-store orders, it is convenient to think of the instruction being divided in the same way:-

```
| 0 | 1 | d |     f     |               N                |
----------------------------------------------------------
                  4                     9
```

However, the two values of d give a total of 32 possible functions; some of these are used for manipulating registers in the secondary operand unit, which is closely associated with all store-to-store operations.

In the organisational orders, the instruction is divided thus:-

```
| 000 |     f'     |               N'                |
------------------------------------------------------
   3        6                      7
```

The cr bits are zero and the 6 f' bits define both the organisational register and the operation to be performed. The organisational orders are mainly concerned with control transfers and the manipulation of organisational registers.

An operand is specified by N (or N') and is independent of the function. An operand may be a literal, a 'named operand' (more simply, a 'name'), or a secondary operand; various internal registers (X, B, etc.) may also be addressed as operands.

1.2 Summary of the Order Code

This section summarises the overall pattern of the order code[3]. The detail is given in later sections as shown below. Some functions in MU5 differ from the general form overleaf, which should be taken only as a statement of the general characteristics.

References	Chapter
Computational Orders	3, 4
B register	3
Accumulators	4
Structure Accessing and Store to Store Orders	5
Organisational Orders	6
Operand Accessing	2
Literals	2.3
Variables	2.4
Internal Register Operands	2.5
Stacked Operands	2.6
Privileged Operands	2.7
Secondary Operands	2.8
Descriptor Types 0 - 3	2.10 - 2.13
Internal Registers	
B, BOD	3
AEX	4
MS	6.2
NB, XNB, SF, (SN)	2.2
CO	6.3
D	2.2
XD, DOD, DT, XDT	5
BN	6.5
Z	2.5

[3] Appendix II shows the version of the order code used by the design engineers.

Computational and Store–to–Store Orders

f	STS/D		B	XS	AU	ADC	AFL
0	XDO =	DO =	=	X =	AOD =	DUMMY	= (32)
1	XD =	D =	= (–1)	DUMMY	DUMMY	AEX =	= (64)
2	STACK	D *=	*=	X *=	AOD *=	AEX *=	*=
3	XD =>	D =>	=>	X =>	AOD =>	AEX =>	=>
4	XDB =	DB =	+	+	A +	DUMMY	+
5	XCHK	MDR	—	—	A—	DUMMY	—
6	SMOD	MOD	*	*	*	DUMMY	*
7	XMOD	RMOD	/	/	DUMMY	DUMMY	/
8	SLGC	BLGC	≢	≢	A ≢	DUMMY	A ≢
9	SMVB	BMVB	V	V	A V	DUMMY	A V
10	DUMMY	BMVE	↑	↑ ARITH	A↑ LOG	A ↑	A↑ CIRC
11	SMVF	SMVF	&	&	A &	DUMMY	A &
12	TALU	DUMMY	⊖	⊖	A ⊖	AOD COMP	⊖
13	DUMMY	BSCN	COMP	COMP	A COMP	COMP	COMP
14	SCMP	BCMP	CINC	AEX=CONVX	DUMMY	UNPACK	AEX=CONVA
15	SUB1	SUB2	⊘	⊘	DUMMY	DUMMY	⊘

3	4	3	6
cr	f	k	n

k1 = 0 or k = 0 – LITERAL n is 6–bit signed literal
k = 1 – IR n defines internal register, P.T.O. –>
k = 2 – V32 Operand is accessed directly at
k = 3 – V64 (NB) + unsigned n; n is scaled for V32
k = 4 – S[B] Operand is accessed via a
k = 5 – S[B] descriptor at (NB) + n, using
k = 6 – S[0] B or O as an index
k1 = 1 or k = 7 ———— K (Extended Operand) P.T.O. –>

3	6	1	6
0	f'	k1	n

Organisational Orders

0	–>	EXIT	DUMMY	DUMMY
4	JUMP	RETURN	DUMMY	DUMMY
8	XC0	XC1	XC2	XC3
12	XC4	XC5	XC6	STACKLINK
16	MS =	DL =	SPM	SET LINK
20	XNB =	SN =	XNB +	XNB =>
24	SF =	SF +	SF = NB +	SF =>
28	NB =	NB = SF +	NB +	NB =>
32	= 0	≠ 0	≥0	< 0
36	≤ 0	> 0	OVERFLOW	Bn
40	= 0	≠ 0	≥0	< 0
44	≤ 0	> 0	OVERFLOW	Bn
48	0	Bn & X	$\overline{\text{Bn}}$ & X	X
52	Bn & $\overline{\text{X}}$	Bn	Bn ≢ X	Bn V X
56	$\overline{\text{Bn}}$ & $\overline{\text{X}}$	Bn ≡ X	$\overline{\text{Bn}}$	$\overline{\text{Bn}}$ V X
60	$\overline{\text{X}}$	Bn V $\overline{\text{X}}$	$\overline{\text{Bn}}$ V $\overline{\text{X}}$	1

Denote test result by T (= 0 for NO, = 1 for YES)
The operand specifies the way in which BN is set

BN = 0	BN & T	BN /& T	BN = T
BN &/ T	DUMMY	BN ≢ T	BN V T
BN /&/ T	BN ≡ T	BN /	BN / V T
BN = / T	BN V / T	BN / V / T	BN = 1

Internal Register Operands

The n bits define the internal register to be used.

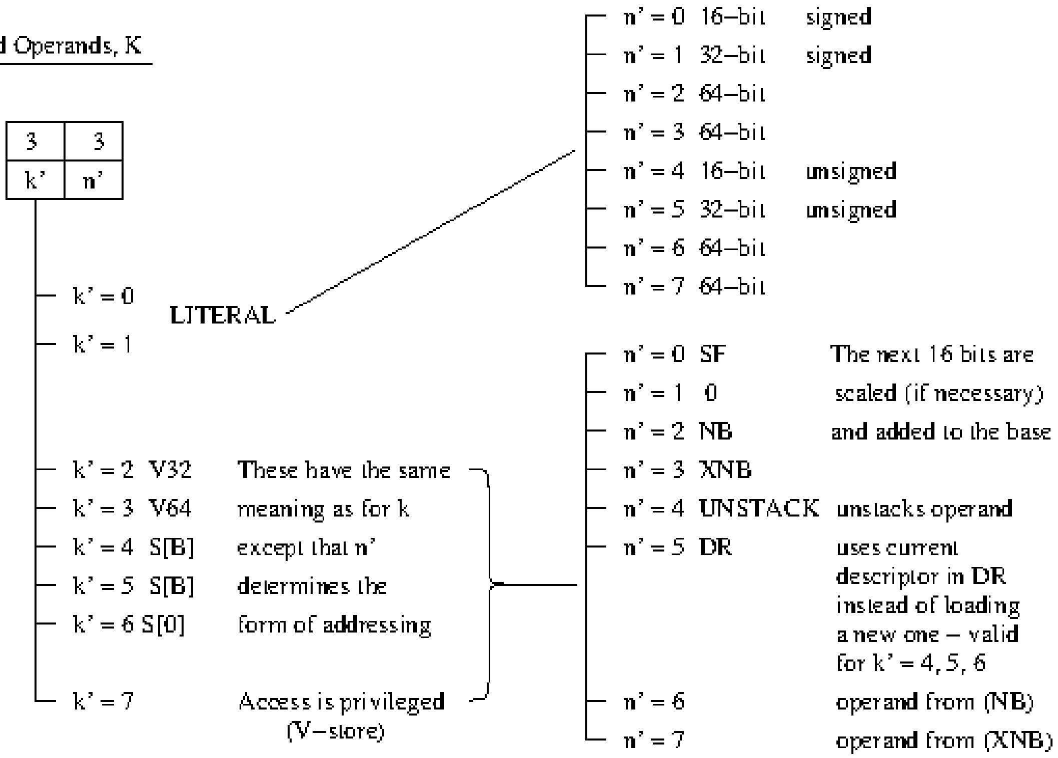
	← 16 →	← 16 →	← 16 →	← 16 →
0	MS	NB	CO	
1			XNB	
2			SN	NB
3			SN	SF
4			BN	
5				
6				
7				

16	D	
17	XD	
18		DT
19		XDT
20		DOD
21		
22		
23		

32		B
33		BOD
34	Z	
35		
36	BOD	B
37		
38		
39		

	← 64 →
48	AEX
49	
50	
51	
52	
53	
54	
55	

Extended Operands, K



3	3
k'	n'

k' = 0
k' = 1 LITERAL

- n' = 0 16–bit signed
- n' = 1 32–bit signed
- n' = 2 64–bit
- n' = 3 64–bit
- n' = 4 16–bit unsigned
- n' = 5 32–bit unsigned
- n' = 6 64–bit
- n' = 7 64–bit

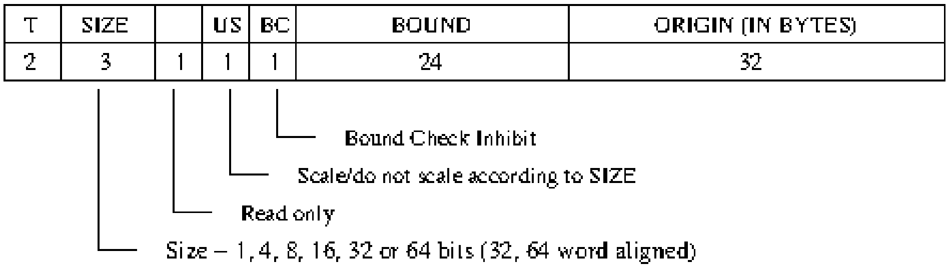
k' = 2 V32
k' = 3 V64
k' = 4 S[B]
k' = 5 S[B]
k' = 6 S[0]
These have the same meaning as for k except that n' determines the form of addressing

k' = 7 Access is privileged (V–store)

- n' = 0 SF The next 16 bits are scaled (if necessary) and added to the base
- n' = 1 0
- n' = 2 NB
- n' = 3 XNB
- n' = 4 UNSTACK unstacks operand
- n' = 5 DR uses current descriptor in DR instead of loading a new one – valid for k' = 4, 5, 6
- n' = 6 operand from (NB)
- n' = 7 operand from (XNB)

Descriptor Formats

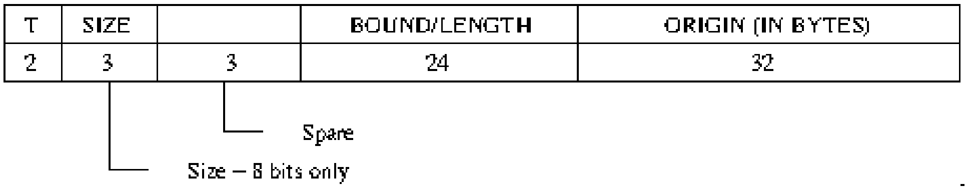
Type 0 - General Vector



T	SIZE		US	BC	BOUND	ORIGIN (IN BYTES)
2	3	1	1	1	24	32

Bound Check Inhibit
Scale/do not scale according to SIZE
Read only
Size – 1, 4, 8, 16, 32 or 64 bits (32, 64 word aligned)

Type 1 - General String

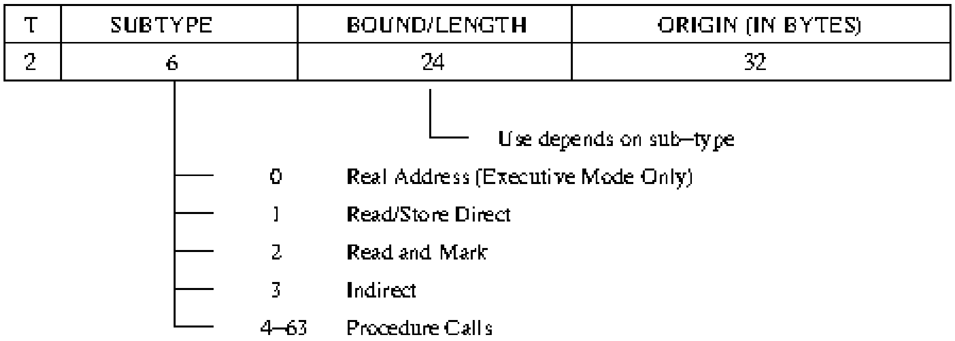


T	SIZE		BOUND/LENGTH	ORIGIN (IN BYTES)
2	3	3	24	32

Type 2 - Address Vector

Format identical with Type 0

Type 3 - Miscellaneous Sub-types



T	SUBTYPE	BOUND/LENGTH	ORIGIN (IN BYTES)
2	6	24	32

Chapter 2 Operand Accessing

2.1 Introduction

The operands for all orders are transferred from source to destination via a highway which is 64 bits wide. For a fetch order, the operand part of the order defines how the highway is loaded and the function part defines the destination (and the operation to be performed at the destination). For a store order, the function part defines how the highway is loaded and the operand part defines the destination.

The function part of an order is described in Chapters 3 - 6; this chapter describes the operand part. With a few exceptions, which will be mentioned when they arise, any function part may be combined with any operand part, so that the two parts may conveniently be described separately.

Operands may be of various sizes up to a maximum of 64 bits. If the operand is less than 64 bits long, then it is loaded on to (or taken from) the least significant end of the highway. On a fetch order, the remaining bits of the highway are set to zero (except for literal operands - see Section 2.3). On a store order, the remaining bits are truncated; for secondary operands only, the truncated bits are checked for zeros.

In addition to various sizes of operand, there are various kinds of operand:-

literals A literal is specified directly as part of the order, e.g. ‘X + 1’ would add 1 to the (original) fixed-point accumulator.

variables A variable is the value in a store location whose address is specified by a base register and the displacement from the base.

internal registers the value in most of the internal registers (B, NB, etc.) can be specified as an operand, e.g. ‘X = NB’ loads the value in NB into the fixed-point accumulator.

stacked operands Operands can be sent to and from a hardware implemented stack working on a last-in first-out basis, e.g. ‘A* STACK’ multiplies the floating-point accumulator by the top operand on the stack and removes the operand from the stack.

privileged operands These can only be accessed in Executive mode; they are described in Chapter 8.

secondary operands A special mechanism is provided for accessing secondary operands, i.e. operands contained in some data structure. The operand part of the order specifies a data descriptor and a modifier. The data descriptor is a 64-bit animal specified as a variable or stacked operand and is combined with the modifier in D to produce the size and address of the secondary operand. For example, consider the orders ‘B = 3, D = FRED, A + D[B]’ where FRED is a descriptor at address (NB = 5). The action would be to load 3 into the modifier register B, then send the descriptor at address NB + 5 to DR, modify by the value in B (i.e. 3) to give the size and address of the secondary operand and finally add this operand to the floating-point accumulator.

2.2 Internal Registers Relevant to Operand Accessing

Section 1.2 contains a complete list of the internal registers with references to their descriptions. In this section, only those registers relevant to operand accessing are described.

The Name Segment Number SN

The name segment number SN is 16 bits long. The two most significant bits are permanently zero, and the remaining 14 bits define the segment currently being used for names in a program. Any segment (0, 1, 2, .. 2↑14 - 1) may be used for this purpose; it is conventional to use segment 0 whenever possible. The value contained in SN may only be altered by calling an executive procedure.

00	NAME SEGMENT
2	14

The Name Base Register NB

The name base register NB is 16 bits long. The most significant 15 bits hold the address of any 64-bit word in the name segment SN; the least significant bit is permanently zero. When NB is the base register for an operand access, then it is added to the displacement (the name) to give the address of the operand within the segment SN. If the addition overflows out of the segment, there will be an interrupt.

NB is designed to be the base register for local names in a procedure; its value will usually only be changed on entry and exit.

Orders which alter NB are described in Sections 6.2 and 6.3

64-bit word address	0
15	1

The Stack Front Register SF

The format of the stack front register SF is identical with that of NB. SF can be used as a base register in the same way as NB. However, the space in front of SF (i.e. at addresses > SF) must not be accessed in this way; interrupt routines use the area in front of SF as working space, so these locations are liable to change at any time.

The stack is designed both to provide temporary working space within a procedure (e.g. for evaluating arithmetic expressions) and also the space required for procedure calls (see Section 6.4). Certain orders, e.g. STACK B, cause an operand to be stacked - SF is advanced by 2 (32-bit) words and the operand is stored at the 64-bit word specified by the new value of SF. These operands may be unstacked by specifying the STACK as the operand part of an order, e.g. 'A = STACK' - the 64-bit word at SF is loaded on to the highway and SF is decreased by 2. Note that all unstacked operands are assumed to be 64 bits long.

Orders that alter SF are described in Sections 3.2, 4.4, 6.2 and 6.3.

64-bit word address	0
15	1

The Extra Name Base XNB

The extra name base register, XNB, is 32 bits long. Bits 2-30 hold the address of a 64-bit word anywhere in the virtual memory; bits 0, 1 and 31 are permanently zero. XNB is used as a base register in the same way as NB and SF, except that the operand is in the segment defined by the top half of XNB (instead of SN). Note that the addition of the name must not overflow out of this segment, or there will be an interrupt.

XNB is designed to be a base register for non-local names used in a procedure, and will often change its value in a procedure. In many programs, the top half of XNB will be zero (like SN).

Orders that alter XNB are described in Section 6.2.

00	segment	64-bit word address	0
2	14	15	1

The Data Descriptor Register D

The data descriptor register, D, is 64 bits long and is used to hold the descriptors required for accessing data structures. The operand part of an order which accesses a data structure specifies a descriptor and a modifier. The descriptor is loaded into D and then combined with the modifier to define the size and address of the particular structure element required.

Details of the descriptor types and the mechanisms for accessing secondary operands are given in Sections 2.8 and 2.10 - 2.13.

D also plays a major part in the operation of the store-to-store orders.
D manipulation orders are described, with the store-to-store orders, in Chapter 5.

2.3 Literal Operands

A literal operand appears directly as part of the order; if a literal is specified as the operand part of a store order, there will be an interrupt.

There are several alternatives:-

(a)	6-bit	signed
(b)	16-bit	unsigned
(c)	16-bit	signed
(d)	32-bit	unsigned
(e)	32-bit	signed
(f)	64-bit	

The literals are copied to the least significant end of the highway. The remaining bits of the highway are set to zeros for unsigned literals and to copies of the sign bit for signed literals. The precise format of orders containing literals is unexpected.

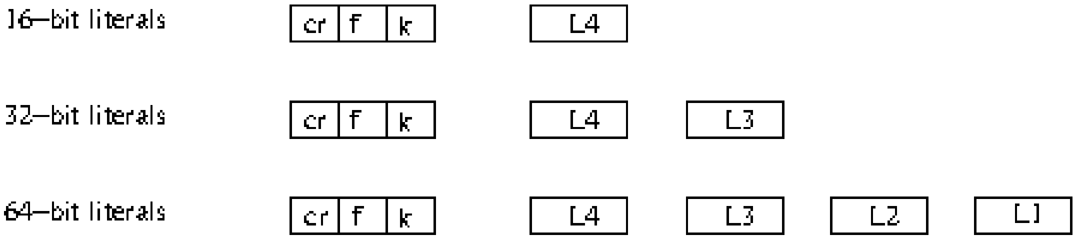
Let L1 denote 16 bits loaded on to highway bits 0-15.

Let L2 denote 16 bits loaded on to highway bits 16-31.

Let L3 denote 16 bits loaded on to highway bits 32-47.

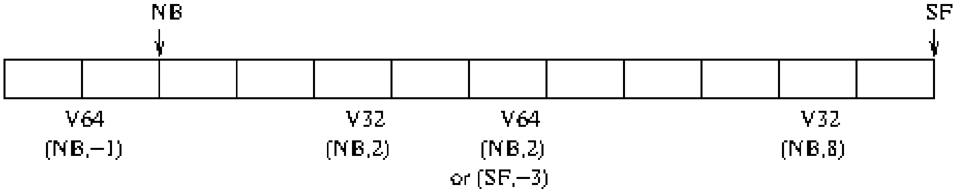
Let L4 denote 16 bits loaded on to highway bits 48-63.

Then the orders appear as follows:-



2.4 Variable Operands

There are two kinds of variable, V32 and V64, of sizes 32 bits and 64 bits respectively. The operand part of the order specifies the kind of order and also defines its name and base. NB, XNB, SF or O may be used as its base (it is convenient to consider O to be a base register which always contains 0). The name is the distance of the variable from the base counting in units equal to the variable size. Some examples are shown below - the diagram represents a section of the virtual store marked out in 32-bit words:-



V64 name are in the range $-2\uparrow 15 \le \text{name} < 2\uparrow 15$

V32 names are in the range $0 \le \text{name} < 2\uparrow 16$

To calculate the address of the operand, the name is scaled (if necessary) and added to the base. If this addition overflows out of the base segment, there will be an interrupt. If NB, SF or O is used as the base, then the variable is taken from the name segment (SN); if XNB is used as the base, then the most significant half of XNB defines the segment. In short instructions the 6-bit displacement, n, is always unsigned, i.e. $0 \le n < 64$.

N.B. If XNB points to a segment which is not the name segment, operands relative to XNB may not be used with the following functions, XNB =>, NB =>, SF =>, SETLINK.

Note that the organisational commands, input/output and CTL use words in the name segment. Thus when running under the operating system, 32-bit words:-

0 - 15 should not be used when writing in XPL

0 - 96 should not be used when using the Autocode machine.

2.5 Internal Register Operands

Any internal register may be specified as the operand for a fetch order; a store order may write to most internal registers, except those within the primary operand unit, i.e. MS, NB, CO, XNB, SN, SF, BN. A table listing all the registers or combinations of registers that can be accessed in this way is given in Section 1.2; note that only complete lines may be accessed, for example, SF cannot be read by itself but only in combination with SN.

Internal register operands may only be used with computational and store-to-store orders, not with organisational orders.

The Internal Register Z is a dummy operand which is written to as a means of suppressing overlap until the order is complete.

2.6 Stacked Operands

When the operand part of the order specifies STACK, the 64-bit word at SF is loaded on to the highway and then SF is decreased by 2. Note that all operands coming from STACK are 64 bits long; this does not mean that only 64-bit operands may be sent to the stack - shorter operands will be extended by zeros on the way.

[A store order specifying STACK will store the operand at SF and decrease SF by 2. This is not a sensible order but it is allowed.]

2.7 Privileged Operands

Privileged operands are used by the Executive to hold system control information. They can only be accessed in Executive mode and are of no interest to the ordinary programmer.

Access can be made in two ways. In the first case, a base register and a name are specified as for a variable operand; the size is always 64 bits and the address is calculated exactly as for a V64 variable. However, access is made not to the virtual store of the program but to the local V-store (i.e. the address is interpreted as a local V-store address). In the second case, the operand part of the order is STACK; this action is exactly the same as for other stacked operands but SF is now interpreted as a 64-bit word address in the local V-store. (The local V-store is described in Chapter 8.)

2.8 Secondary Operands

For a secondary operand, the operand part of the order specifies a 64-bit descriptor and a modifier. Normally, the descriptor specifies the type and origin (i.e. the starting address) of the data structure containing the secondary operand and the modifier defines which particular operand is required. For example, if A is a descriptor specifying a vector of 32-bit elements, then the orders 'B = 25; X = A[B]' would load the 25th element (counting from zero) of A into the fixed-point accumulator.

Descriptors can define vectors or strings of elements of various sizes; miscellaneous special types are also provided. The different types of descriptor are defined in Sections 2.10 - 2.13.

A descriptor may be specified in the same way as a variable or stacked operand; it is always 64 bits long and is loaded into the D register. Alternatively, the operand part of the order may specify that the descriptor is already in D; this avoids unnecessary loading into D if the same descriptor is used for consecutive secondary operands.

The modifier used is normally B or O (i.e. no modifier). However there are special functions (see Chapter 5) which allow any operand to be used as a modifier and also cause a special type of modification. All modifiers are interpreted as signed 32-bit integers.

When access is made via certain types of descriptor (e.g. vectors) it is possible to check automatically that the modifier (if any) lies in the range $0 \leq$ modifier $<$ bound. The bound is held in bits 8-31 of the descriptor.

N.B. A secondary operand may not be used in conjunction with the following functions: D =>, XD =>, XNB =>, NB =>, SF =>, SETLINK.

2.9 Length of the Orders

An order may be 16, 32, 48 or 80 bits long. It will be 16 bits long when:-

(a) Operand is 6-bit literal or internal register

(b) Operand is variable or secondary; base register is NB and $0 \leq \text{name} \leq 63$; function part is computational or store-to-store

(c) Operand is variable, privileged or secondary from STACK

An order will be 48 bits long if the operand is a 32-bit literal and will be 80 bits long if the operand is a 64-bit literal. In all other cases an order will be 32 bits long.

2.10 Type 0 - Vector Descriptors

Type 0 descriptors are used for vectors of elements of size 1, 4, 8, 16, 32 or 64 bits. The descriptor defines the origin of the vector, the element size and an upper bound for the modifier (i.e. the number of elements in the vector). The format is :-

T	SIZE	RO	US	BC	BOUND	ORIGIN IN BYTES
2	3	1	1	1	24	32

T = 0 Defines type 0.

SIZE Defines the element size as 1, 4, 8, 16, 32 or 64 bits. (Coded as follows:-
000 = 1 bit; 010 = 4 bits; 011 = 8 bits; 100 = 16 bits;
101 = 32 bits; 110 = 64 bits.)

RO If RO = 1, descriptor is read only and any attempt to use it for writing will cause an interrupt.

US If US = 0, then the modifier is scaled before being added to the origin - for 1-bit elements the modifier is shifted down 3 bits, for 4-bit elements down 1 bit, for 8-bit elements none, for 16-bit elements up 1 bit, for 32-bit elements up two bits, for 64-bit elements up 3 bits.
If US = 1, the modifier is not scaled.

BC If BC = 1, then there is no bound check.

BOUND An upper bound for the modifier. If the bound check bit BCH in DOD (see Chapter 5) is set to 0 and BC = 0, then the modifier (if any) must lie in the range $0 \leq \text{modifier} < \text{BOUND}$, otherwise there will be an interrupt (see Section 5.2).

ORIGIN The origin defines the base address of the vector; it is always a 32-bit byte address. For 16-bit vectors, the least significant bit of the modified address is ignored, so that all elements start at a 16-bit word boundary. For 32 and 64-bit vectors, the two least significant bits are ignored, so elements start on a 32-bit word boundary. Note that vectors of 1-bit and 4-bit elements must start on a byte boundary.

Action: When an access is made, the modifier (if any) is scaled (according to SIZE and US) and added to the origin to give the address of the required element. Provided there is no bound check fail, the element is accessed. On a fetch order, it is loaded on to the highway (operands < 64 bits long are loaded at the least significant end and the remaining bits are zeroed). On a store order, the highway is stored at the element; there will be an interrupt of any non-zero bit is truncated (see Section 5.2).

2.11 Type 1 - String Descriptors

Type 1 descriptors are used for 8-bit elements. The descriptor defines the origin and length of the string. The format is:-

T	SIZE				LENGTH	ORIGIN IN BYTES
2	3	1	1	1	24	32

T = 1 Defines type 1.

SIZE Must define the element size as 8 bits (011), else an interrupt will occur.

LENGTH Defines the number of elements in the addressed string.

ORIGIN Defines a base address, as in type 0.

Action: The modifier (if any) is added to the origin to give the address of the start of the string. LENGTH defines the length of the string, i.e. the number of elements in the string. There is no bound checking.

The final operand is a string of 8-bit elements. In store-to-store orders the whole string will be used as the operand (see Chapter 5). In computational orders, the operand can be at most 64 bits long; if the string is less than 64 bits, then it is zero filled for fetch, truncated with zero-checking for store; if the string is longer than 64 bits, just the first 64 bits of the string are loaded on to or stored from the highway.

2.12 Type 2 - Descriptor Descriptors

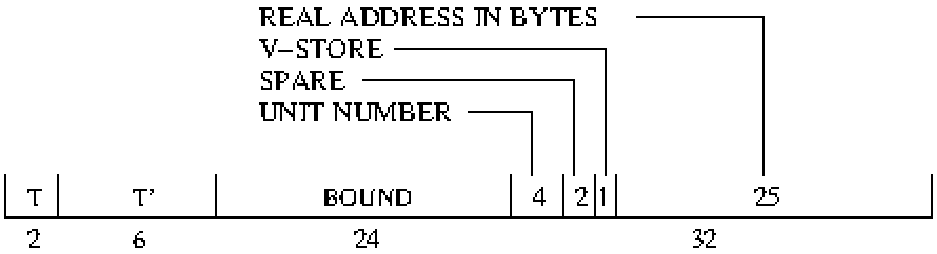
Type 2 descriptors are identical with type 0 descriptors (except that T = 2 instead of 0). [It's not clear whether type 2 included the RO bit.]

T	SIZE	RO	US	BC	BOUND	ORIGIN IN BYTES
2	3	1	1	1	24	32

It is conventional to use type 2 descriptors to address vectors containing descriptors; type 0 is used for vectors containing data.

2.13 Type 3 - Miscellaneous Descriptors

Type 3.0 Real Address



T, T' = 3, 0 define type 3.0

BOUND Upper bound for modifier as in type 0.

ORIGIN Contains the real store address (the physical address, not the virtual store address) of a 64-bit operand. The three least significant bits are ignored[a].

Action: The operand is accessed in the same way as a type 0 64-bit element. The modifier is always scaled and bound checked if bit BCHI in DOD is set to 0. Type 3.0 descriptors may only be used in Executive mode.

Type 3.1 Read/Store Direct

T	T'	BOUND	ORIGIN IN BYTES
2	6	24	32

T, T' = 3, 1 defines type 3.1

BOUND Upper bound for modifier as in type 0.

ORIGIN Defines a 64-bit word address; the three least significant bits are ignored.

[a] The V-Store bit should really have been labelled Vx-Store. The V-Store and Vx-Store (Chapters 8 & 9) are (almost) completely separate and are separately addressed.

Action: Access is made in exactly the same way as for a type 0 64-bit element (assuming US = BC = 0, so that the modifier is scaled and bound-checked if bit BCHI in DOD is set to 0. Note that the word lies on a 64-bit word boundary.

The accessing mechanism for this descriptor bypasses all operand buffers and always accesses the real store corresponding to the defined virtual address. This type of access is needed in some executive procedures.

Type 3.2 Read and Mark

T	T’	BOUND	ORIGIN IN BYTES
2	6	24	32

T, T’ = 3, 2 define type 3.2

BOUND Upper bound for modifier as in type 0.

ORIGIN Defines a 64-bit word address;
the three least significant bits are ignored.

Action: Access is made in exactly the same way as for type 3.1 descriptors, bypassing the operand buffers. In addition, for a fetch order, the value of the 64-bit word in the store is finally set to zero.

Type 3.3 Indirect

T	T’	X	ORIGIN IN BYTES
2	6	24	32

T, T’ = 3, 3 define type 3.3

X Unused

ORIGIN Defines a 32-bit word address; the three least significant bits are ignored.

Action: The 64-bit element at the origin address is loaded into D and then interpreted according to its type. The new descriptor may be indirect, in which case the whole process is repeated. If a modifier is specified, modification takes place at the final (not indirect) stage.

Type 3.4 - 3.7 Procedure Call

T	SIZE	T'	X	ORIGIN IN BYTES
2	3	3	24	32

T, SIZE, T' = 3, 5, 4 - 7 define the procedure call type.

X Upper bound for procedure call vector which must have 32-bit elements.

ORIGIN Contains the address of the procedure call vector. The two least significant bits of the origin field are ignored.

Action: When an attempt is made to access the operand, the hardware forces a procedure call[a] to the address held in the first 32-bit word of the vector, with the return link including the 'D set' bit pointing to the instruction attempting to make the access. The origin is not modified (even if a modifier is specified by the operand part of the order).

One example of the use of the procedure call descriptor is an implementation of an Algol parameter call by name. If the corresponding actual parameter is a simple variable, then the parameter descriptor can be a normal type 0 descriptor. But if the actual parameter is an expression, then the descriptor will be a procedure call to code which evaluates the expression. The value will be stored in some suitable store location and D replaced by a type 0 descriptor pointing to it; finally, the 'D set' bit in the stored link (c.f. Section 6.2) is set and an EXIT obeyed. The order causing the procedure call will be re-obeyed - the 'D set' bit prevents reloading of D and defines that the current value of D describes the required operand. (The 'D set' bit is automatically reset to 0.)

Note that a procedure call descriptor may be modified; the modification will take place when the order is re-obeyed after exit from the procedure.

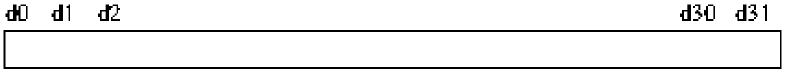
[a] This involves executing two hard-wired instructions held in the Instruction Buffer Unit:-
STACKLINK
JUMP D[0].

Chapter 3 The B-arithmetic

3.1

There is a separate 32-bit B-arithmetic unit which operates on the modifier register B. Although B is used mainly for modification, it is also used for some of the simpler integer arithmetic, for example, i = i + 1.

The bits in B are numbered from 0 on the left hand (most significant end).



The operand connection to the B-arithmetic unit is from the least significant 32 bits of the highway (bits 32-63).

The B-arithmetic unit performs signed 2's complement arithmetic. Thus B may take values in the range -2↑31 to 2↑31 - 1. If, after any arithmetic operation, the true result is outside this range, the overflow bit is set. The overflow bit and a bit which is used to inhibit the interrupt resulting from overflow are digits 5 and 0 of BOD. Thus digit 5 of BOD is set to a one if overflow occurs and the interrupt will be inhibited if digit 0 is also set to one. All other digits of BOD are not significant.

3.2 The B-Instructions

The order code provides for 16 B-functions. Only 14 of these are implemented on MU5 and the rest are dummy instructions. The instructions are:-

LOAD (=)

Load B from the least significant 32 bits of the highway.

LOAD & DECREMENT (=')

Load B from the least significant 32 bits of the highway then subtract 1. If an overflow occurs, digit 5 of BOD is set.

STACK & LOAD (*=)

The stack front register (SF) is first advanced by 2. The contents of B are placed on the highway as for a store order (see below). This is then sent to the 64-bit word whose address is specified by the new value of SF. Finally, the operand is loaded into B as in the load order (see above).

STORE (=>)

The content of B is placed on the least significant 32 bits (bits 32-63) of the highway and zeros are placed on the most significant 32 bits (0-31). The operand specifies the destination of this information.

ADD (+)

The operand is added to B, leaving the result in B. If an overflow occurs, then digit 5 of BOD is set.

SUBTRACT (-)

The operand is subtracted from B, leaving the result in B. If an overflow occurs, then digit 5 of BOD is set.

MULTIPLY (*)

B is multiplied by the operand to produce a 32-bit result which is the least significant 32 bits of the true 64-bit signed answer. If the true product has more than 32 significant bits, then B contains the least significant 32 bits of the true answer and digit 5 of BOD is set.

DIVIDE (/) A dummy instruction

NON-EQUIVALENCE($\not\equiv$)

B and the operand are non-equivalenced to produce a result in B.

OR (V)

B and the operand are or'ed to produce a result in B.

AND (&)

B and the operand are and'ed to produce a result in B.

SHIFT ($\uparrow$)

B will be shifted arithmetically (left) by the number of places specified by the signed integer in digits 57-63 of the operand. If overflow occurs, digit 5 of BOD is set.

COMPARE (COMP)

The operand is subtracted from B. Bits T1 and T2 of the test register are set from the result of the subtraction (see Section 6.4). A true result is always generated and no overflow may occur. The overflow bit in BOD is copied to bit T0 of the test register[a]. The contents of B are not altered.

REVERSE SUBTRACT ($\ominus$)

B is subtracted from the operand leaving the result in B. If an overflow occurs, then digit 5 of BOD is set.

COMPARE & INCREMENT (CINC)

A compare operation is performed (see above) then B is incremented by 1. If B overflow occurs as a result of being incremented, then digit 5 of BOD is set after the compare operation has been completed.

REVERSE DIVIDE ($\oslash$) A dummy instruction.

[a] This should have said "... no overflow interrupt may occur, i.e. if the result overflows, bit T0 in the test register is set instead of digit 5 in BOD."

Chapter 4 Accumulator Arithmetic

4.1 The Accumulator and its Associated Registers

The function code contains a set of 16 functions for each of the following kinds of arithmetic:-

fixed point signed
fixed point unsigned
floating point
decimal

In MU5 there are two associated registers:-

X, which is used by the signed fixed point orders,

and

A, which is used by the unsigned fixed point, floating point and decimal orders.

Each accumulator register is conceptually 64 bits long but digits 0-31 of X will not exist on MU5. There are two other visible 64-bit registers in the arithmetic unit, namely AOD and AEX. The bits of AOD are concerned mainly with interrupts whereas AEX (the accumulator extension register) serves to hold the least significant part of double length results. Because the accumulator ‘A’ is shared, the load and store functions would be the same in the fixed point unsigned, decimal and floating point instruction sets. Therefore, the load and store functions in the fixed point unsigned set are made to operate on AOD and those in the decimal set on AEX.

It is convenient to consider the operand for each function to be the 64-bit accumulator input buffer AIB. Thus the operation of the accumulator functions will be described by reference to the registers:-

A, X, AOD, AEX, AIB

4.2 Allocation of Digits in AOD

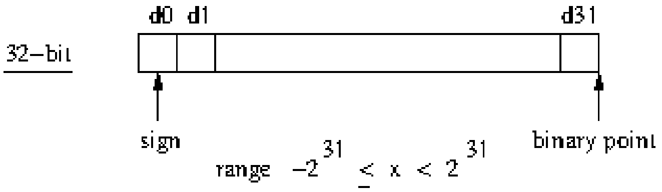
digit	
51	Operand size (0/1 meaning 32/64 bits)
52	Inhibit floating point overflow interrupt
53	Inhibit floating point underflow interrupt
54	Inhibit fixed point overflow interrupt
55	Inhibit decimal overflow interrupt
56	Inhibit zero divide interrupt
57	Floating point overflow indicator
58	Floating point underflow indicator
59	Fixed point overflow indicator
60	Decimal overflow indicator
61	Zero divide indicator
62	Inhibit rounding
63	Double length $\underline{+}$

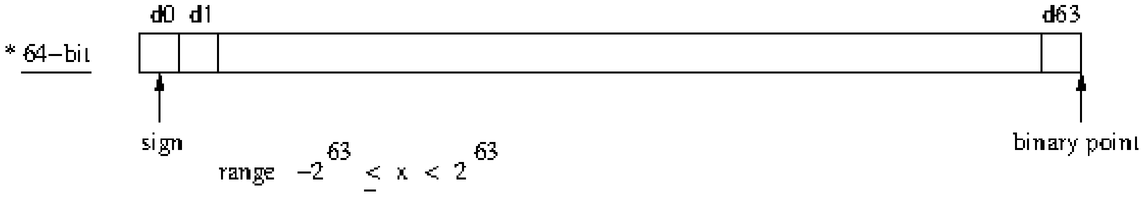
4.3 Formats for Arithmetic Data

The formats marked with an asterisk are software concepts only and have no significance in the hardware.

(a) Fixed-point signed

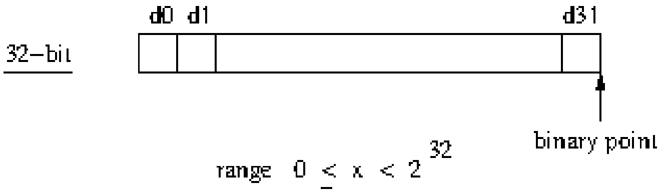
Data is signed binary, held in 2's complement form. For multiplication and division, the binary point is at the least significant end, i.e. data is interpreted as an integer.

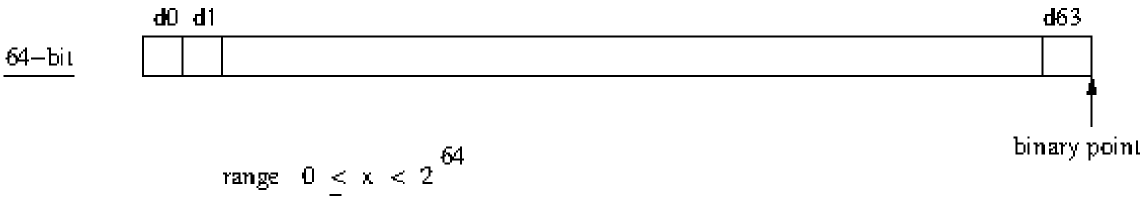




(b) Fixed-point unsigned

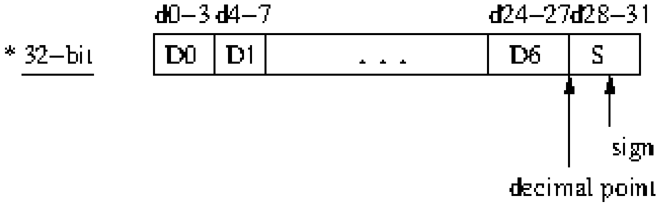
Data is unsigned binary. For multiplication and division, the binary point is at the least significant end, i.e. data is interpreted as an integer.



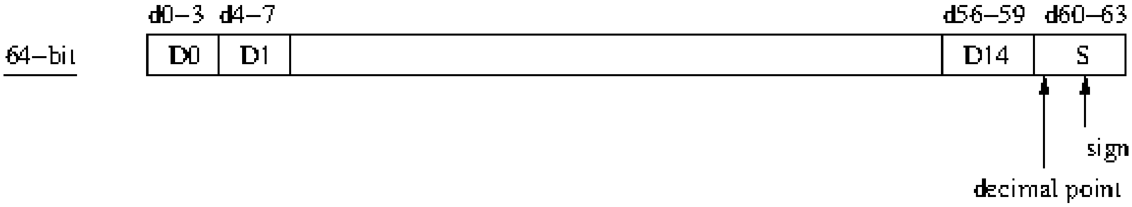


(c) Decimal

Data is in sign-modulus form. The modulus consists of 7 or 15 decimal digits occupying 4 bits each and the sign occupies 4 bits at the least significant end. Each decimal digit is coded in binary (0 ≡ 0000, 1 ≡ 0001 . . . 9 ≡ 1001); the sign code 1101 means -ve, all other combinations mean +ve (1111 is preferred). For multiplication and division, the decimal point is at the least significant end, i.e. data is interpreted as an integer.



range $-10^7 < x < 10^7$



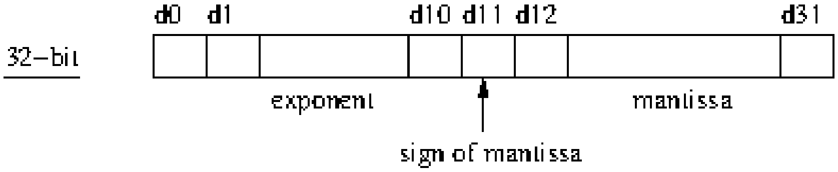
range $-10^{15} < x < 10^{15}$

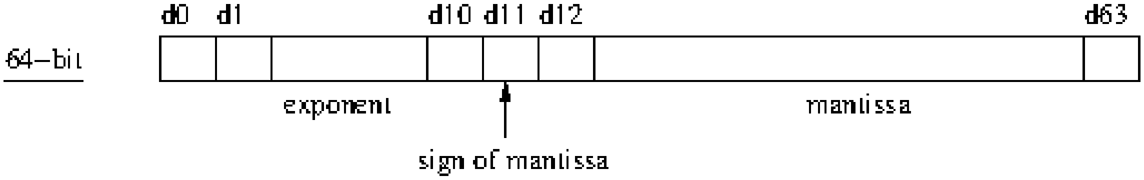
(d) Floating-point

Da ta is stored as a 2's complement mantissa m with an 11-bit exponent e stored at the most significant end. The most significant bit of m gives the sign of m and the interpretation of m assumes a binary point after the sign digit. The exponent has the base 16 and is interpreted as an unsigned 11-bit integer plus 1024, i.e.

00000000000	->	-1024
00000000001	->	-1023
	.	
	.	
10000000000	->	0
	.	
	.	
11111111111	->	1023

This code has been chosen so that floating-point zero has all bits = 0.





4.4. The Signed Fixed Point Accumulator Orders

The arithmetic functions in this set assume X and the operand to be signed integers.

LOAD (=)

Copy digits 32-63 from AIB to X.

LOAD DOUBLE (=') Dummy instruction.

STACK & LOAD (*=)

Stack X in digits 32-63 of the next free 64-bit word on the stack, making digits 0-31 in this word zero. Then operate as for LOAD.

STORE (=>)

Copy X to digits 32-63 of the highway and zeros to digits 0-31 of the highway.

ADD (+)

Digits 32-63 of AIB are added to X and the result is returned to X. If the addition overflows, digit 59 of AOD is set. In this case the result in X will be the least significant 32 bits of a 32-bit answer.

SUBTRACT (-)

Digits 32-63 of AIB are subtracted from X and the result is returned to X. If the result overflows, digit 59 of AOD is set.

MULTIPLY (*)

X is multiplied by digits 32-63 of AIB to form a signed single length result in X. If the result overflows, digit 59 of AOD is set and the result is the least significant bits of the 64-bit answer.

DIVIDE (/)

X is divided by digits 32-63 of AIB to form a quotient in X, which will be rounded down. If the divisor is zero, then digit 61 of AOD is set and X will be unaltered.

NON-EQUIVALENCE(≢)

The logical non-equivalence of digits 32-63 of AIB with X replaces X.

OR (V)

The logical or of digits 32-63 of AIB with X replaces X.

SHIFT (↑)

X will be shifted arithmetically (left) by the number of places specified by the signed integer in digits 58-63 of AIB. Digit 59 of AOD will be set if the result overflows.

AND (&)

The logical and of digits 32-63 of AIB with X replaces X.

REVERSE SUBTRACT (⊖)

X is subtracted from digits 32-63 of AIB and the result is stored in X. If overflow occurs, digit 59 of AOD is set.

COMPARE (COMP)

The operand in digits 32-63 of AIB is subtracted from X. Both are treated as signed integers. Bits T1 and T2 of the test register are set from the result of the subtraction. Note that a true result is generated and no overflow may occur. Bit 59 V bit 61 of AOD is copied to bit T0 of the test register[a]. The content of X is not altered.

CONVERT (CONV)

The only conversion function implemented in the 'X' set is the conversion from integer to floating. The standardised floating result is left in AEX.

REVERSE DIVIDE (⊘)

Except that digits 32-63 of AIB are divided by X, this function operates as for DIVIDE.

[a] This should have said "… no overflow interrupt may occur, i.e. if bit 59 or 61 would have been set, the logical OR of the inputs to these bits is formed in the hardware and used to set T0."

4.5. The Unsigned Fixed Point Accumulator Orders

The arithmetic functions in this set assume the least significant 32 bits of A and the operand in digits 32-63 of AIB to be 32-bit unsigned integers. They return a 64-bit signed result to A. If the most significant 32 bits of A or AIB are initially non-zero, they are set to zero prior to arithmetic.

LOAD (=)

Copy digits 51-63 from AIB to AOD.

Note: floating point LOAD DOUBLE is the correct order to use in conjunction with unsigned arithmetic.

LOAD DOUBLE (=') Dummy instruction.

STACK & LOAD (*=)

AOD is stacked and loaded

STORE (=>)

Copy digits 51-63 of AOD to the highway, setting the other digits of the highway to zero.

ADD (+)

Digits 32-63 of AIB are added to digits 32-63 of A and the result is stored in digits 0-63 of A.

SUBTRACT (-)

Digits 32-63 of AIB are subtracted from digits 32-63 of A and the result is stored in digits 0-63 of A.

MULTIPLY (*)

Digits 32-63 of A are multiplied by digits 32-63 of AIB to form a 64-bit product which is stored in A.

DIVIDE (/) Dummy instruction.

NON-EQUIVALENCE(≢)

Digits 32-63 of A are non-equivalenced with digits 32-63 of AIB and the result is stored in digits 32-63 of A. Digits 0-31 are set to zero.

OR (V)

Digits 32-63 of A are or'ed with digits 32-63 of AIB and the result is stored in digits 32-63 of A. Digits 0-31 are set to zero.

SHIFT (↑)

A is shifted logically (left) by the number of places specified by the signed integer in digits 57-63 of AIB. This order operates on all 64 bits of A.

AND (&)

Digits 32-63 of A are and'ed with digits 32-63 of AIB and the result is stored in digits 32-to 63 of A. Digits 0-31 are set to zero.

REVERSE SUBTRACT (⊖)

Digits 32-63 of A are subtracted from digits 32-63 of AIB and the result is stored in digits 0-to 63 of A.

COMPARE (COMP)

As for COMP in the signed fixed point set, except that the comparison applies to digits 32-63 of A and is on an unsigned basis. T0 of the test register is set to zero.

REVERSE DIVIDE (⊘) Dummy instruction.

4.6. The Decimal Mode Accumulator Orders

LOAD (=) Dummy instruction.

LOAD DOUBLE (=')

Load AEX from AIB.

STACK & LOAD (*=)

Stack and load AEX.

STORE (=>)

Store AEX.

ADD (+) Dummy instruction.

SUBTRACT (-) Dummy instruction.

MULTIPLY (*) Dummy instruction.

DIVIDE (/) Dummy instruction.

NON-EQUIVALENCE(≢) Dummy instruction.

OR (V) Dummy instruction.

SHIFT (↑)

Digits 59-63 of AIB are interpreted as a signed binary integer which specifies the number of decimal places by which A is to be shifted (left). The shift is logical over digits 0 to 59. Digits 60-63 are unaltered. If a left shift overflows, digit 60 of AOD is set.

AND (&) Dummy instruction.

COMPARE AOD

Digits 51-63 of AIB are and'ed with digits 51-63 of AOD. The overflow digit of the test register will be set to 0/1 depending upon the result being non-zero/zero.

COMPARE (COMP)

A is interpreted as a decimal number according to the formats in Section 4.3. Bit T2 of the test register is set as the sign (bits 60-63) of A. The logical & of A and AIB is formed and bit T1 of the test register is set according as the result is = or ≠ to zero. Bit T0 of the test register is set to bit 60 of AOD (decimal overflow)[a].

UNPACK

This instruction sets AEX (32-59) = AIB (32-59) and AEX (60-63) = AIB (60-63) V A (0-3). AEX (0-31) are unaltered. It also shifts digits 0-59 of A four places left. Digits 56-59 of A are set zero and digits 60-63 are unaltered.

REVERSE DIVIDE (⊘) Dummy instruction.

[a] As in the case of other COMP orders, an overflow is recorded in test bit T0 rather than in AOD.

4.7. The Floating Point Accumulator Orders

For some of the floating-point arithmetic instructions, A and AEX are regarded as a double-length result register, A holding the most significant half and AEX the least significant half. Both will have the format shown in Section 4.3.

In all instructions AIB will form the 64-bit operand if digit 51 of AOD is one. If AOD is zero, digits 32-63 of AIB will form the most significant part of the 64-bit operand of which the other half is zero.

The operation of the floating-point instructions is dependent upon the setting of digits 62 and 63 of AOD. Digit 62 is the inhibit rounding digit. Rounding is performed by forcing 1 into digit 63 of A if the mantissa of AEX is non-zero. Digit 63 is set to select the special double-length versions of add, subtract and reverse subtract and is ignored by all other operations.

LOAD SINGLE (=)

First, digit 51 of AOD is set to zero, then digits 32-63 of AIB are copied to digits 0-31 of A. Digits 32-63 of A are cleared.

LOAD DOUBLE (=')

First, digit 51 of AOD is set to a one, then AIB is copied to A.

STACK & LOAD (*=)

A (or digits 0-31 of A) is stacked, then A is loaded as in = / ='if digit 51 of AOD is 0/1. Digit 51 of AOD is unaltered.

STORE (=>)

A (or digits 0-31 of A, as for X) is stored depending on whether digit 51 of AOD is 1 or 0.

ADD (+)

The operand from AIB is added to A. First the exponents of A and AIB are compared and the exponent field of A is replaced with the larger. The mantissa associated with the smaller exponent is then shifted right by the number of hexadecimal places given by the exponent difference. Also the digits that are shifted out are placed in the mantissa field of AEX, the rest of AEX being cleared. The mantissa field of A is set to the sum of the mantissa fields of A and AIB (one of which may have been shifted). The normalisation shifts that follow apply across the mantissa fields of both A and AEX, with a maximum shift of 13 hexadecimal places. If both mantissa fields are zero, a standard floating point zero is generated (Section 4.3). The exponent of A and AEX are both set to the exponent of the double-length result and rounding is performed as described above. When digit 63 of AOD is set and A and AEX contain a double-length number smaller than AIB, a correct unrounded double-length result will be formed. If either of the above cause exponent overflow/underflow, digit 57/58 of AOD will be set.

SUBTRACT (-)

The operand from the highway is subtracted from A. The operation proceeds in the same general way as ADD. However, if the number to be subtracted is smaller, it is negated and then the add operation is performed.

MULTIPLY (*)

A is multiplied by the operand to give a double length result in A and AEX. This result is standardised and AEX exponent is set as above. Rounding will occur if digit 62 of AOD is not set. On exponent overflow/underflow, digit 57/58 of AOD is set

DIVIDE (/)

A is divided by the operand to give a single-length standardised (possibly rounded) result in A. If the divisor is zero, digit 61 of AOD is set or if exponent overflow or underflow occurs, digit 57/58 of AOD is set.

NON EQUIVALENCE (≢)

The result of combining A with AIB with logical ≢ is returned to A.

OR (V)

The result of combining A with AIB with logical V is returned to A.

SHIFT (↑A)

A is shifted circularly (left) by the number of places specified by digits 58-63 of AIB.

AND (&)

The result of combining A with AIB with logical & is returned to A.

REVERSE SUBTRACT(⊖)

This is the same as SUBTRACT, except that A is subtracted from the operand.

COMPARE (COMP)

A is compared with the operand in AIB and the test register is set. Both are assumed to be floating-point numbers. T0 of the test register is set if any of bits 57, 58, 61 are (i.e. would have been) set.

CONVERT (CONV)

The only conversion function provided in the floating-point set is one that converts the integer part of A to a signed fixed-point number, leaving the result in AEX. If the result is too big, digit 58 of AOD is set.

REVERSE DIVIDE (⊘)

This is the same as DIVIDE, except that the operand is divided by A.

Chapter 5 Structure Accessing and Store to Store Orders

5.1 Introduction

This chapter defines the registers D, XD and DOD (Section 5.2) and describes the orders associated with the secondary operand unit. The orders fall into three classes:-

(a) Register manipulation	(Section 5.3)
(a) Structure access	(Section 5.4)
(a) Store to Store	(Section 5.5)

The register manipulation orders are concerned with loading and storing the registers D and XD. The structure access orders are concerned with modifying descriptors and accessing elements of data structures. The store to store orders enable operations to be carried out on strings of bytes of any length, e.g. moving one string to another or comparing strings. The registers D and XD are used to hold the descriptors of the strings.

The STACK order is described in Section 5.3, chiefly because it isn't described anywhere else.

This chapter assumes the reader is familiar with the different types of descriptor defined in Sections 2.10 - 2.13.

5.2. Internal Registers in the Secondary Operand Unit (SEOP)

The two main registers in the SEOP are D and XD. They are both 64 bits long and are used to hold descriptors, so they have type, bound and origin fields as shown below:-

```
            d0                                                d63
  D         | TYPE |   BOUND (DB)    |   ORIGIN (DO)         |
            ------------------------------------------------
                 8          24                  32
  XD        | TYPE |   BOUND (XDB)   |   ORIGIN (XDO)        |
            ------------------------------------------------
                 8          24                  32
```

The following notation is used for the various parts of D, XD:-

DO	the origin field of D	(d32-63)
XDO	the origin field of XD	(d32-63)
DB	the bound field of D	(d8-31)
XDB	the bound field of XD	(d8-31)
DT	the top half of D	(d0-31)
XDT	the top half of XD	(d0-31)

The only other register in SEOP that can be used by the programmer is DOD; DOD, D, XD, DT AND XDT may all be read from or written to as internal register operands. DOD is a 32-bit register that contains the interrupt and interrupt inhibit bits for SEOP as follows:-

d31	XCH	XCHK digit
d30	ITS	Illegal Type/Size
d29	EMS	Executive mode Subtype used in non-executive mode
d28	SSS	Short Source String in store to store order
d27	NZT	Non-Zero Truncation when storing secondary operand
d26	BCH	Bound Check Fail during secondary operand access
d25	SSSI	SSS Interrupt Inhibit
d24	NZTI	NZT Interrupt Inhibit
d23	BCHI	BCH Interrupt Inhibit
d22		Read only interrupt, attempt to write using type 0 descriptor with read only bit set.

ITS and EMS will always cause an interrupt. SSS, NZT or BCH will cause an interrupt unless SSSI, NZTI or BCHI respectively, are set.

5.3. D and XD Manipulation Orders and STACK

STACK Stack the operand (advance SF by 2, then store operand at new SF).

DO = Load the origin of D from bits 32-63 of the operand.
Bits 0-31 of D are unaltered.

D = Load D from bits 0-63 of the operand.

D *= Stack D (advance SF by 2, then store operand at new SF).
Then load D from bits 0-63 of the operand.

D => Store D in bits 0-63 of the operand.

DB = Load the bound of D (bits (8-31) from bits 40-63 of the operand.
The rest of D is unaltered.

XDO =
XD =
XD =>
XDB =
As for DO =, D =, D =>, DB = but operate on XD instead of D.

Note: Some of these orders may be used with secondary operands.
For S[B] and S[0] operands, the effect is as follows:-

(a) DO =, D =, DB =
D will first be loaded with the S operand descriptor; then the secondary operand will be accessed and will replace the whole or part of the new value of D.

(b) D *=
The original contents of D will be stacked before the S descriptor is loaded into D.

(c) XDO = , XD =, XDB =
Work as expected.

The orders D => and XD => may not be combined with any secondary operand (S[B], S[0], D[B] or D[0]).

5.4. Structure Access Orders

[None of these orders may be used with secondary operands[4].]

MOD Uses bits 32-63 of the operand as a signed integer modifier for the descriptor in D. The modifier is added to the origin field (after scaling if US = 0 in type 0 or 2 descriptors) and subtracted from the bound field. A bound check interrupt will occur unless $0 \leq$ modifier $<$ bound (assuming bound checking is not inhibited). The bound check applies to descriptors of types 0, 1, 2, 3.0, 3.1 and 3.2. For type 3.3, the indirectly addressed descriptor is loaded into D before the modification takes place. Similarly for type 3.4 - 3.31, the procedure is called first.

XMOD Exactly the same as MOD except that it works on XD instead of D. (Types 3.3, 3.4 - 3.31 are illegal.)

SMOD As for MOD, but DB is unaltered and there is no bound check.

MDR Equivalent to MOD followed by a D = D[0].

RMOD Bits 0-31 of the operand are loaded into bits 0-31 of D.
Bits 32-63 of the operand are added to bits 32-63 of D.

XCHK If $0 \leq$ operand bits 32-63 $<$ XDB, then bit 31 of DOD is set to 1, otherwise it is set to 0.

[4] It might have been helpful for there have been a reminder here that 1-bit and 4-bit vectors must start on a byte boundary, meaning that if MOD, etc. are used with these vectors, they can only take operands that are multiples of 8.

SUB1 A complicated order that works as follows:-

```
XD = operand   (bits 0-63)
D = 0          (clear all bits of D)
B - XD[0]      (B - operand addressed by XD)
B * XD[1]
DB = XD[2]
MOD B
XMOD 3
```

XD must be a vector descriptor (type 0 or 2) addressing 32-bit elements.

SUB2 Omits the first two steps of SUB1:-

```
B - XD[0]
B * XD[1]
DB = XD[2]
MOD B
XMOD 3
```

Use of structure access orders

MOD (and XMOD) can be used for constructing substrings of larger strings, e.g. ‘D = S; MOD I; DB = L’ creates a descriptor for the string of length L starting at the Ith byte of the string S. MOD can also be used to step through a vector, since ‘D = V; MOD 1’ creates a descriptor to a vector consisting of all but the first element of V.

MDR can be used for moving through a list structure or for creating arrays via an Iliffe vector.

RMOD is used for ‘reverse modification’. This is useful when used in combination with the ‘dope vector’ orders SUB1 and SUB2 described below. It can also be used to make data structures relocatable.

XCHK is a special order that is used to check for overlapping strings. The only dangerous case is when the start of the destination string (for a move or logical store to store order) lies within the source string. So the idea is to put the source string descriptor in XD and then use XCHK with operand = destination origin - source origin.

Dope Vector Orders

The SUB1 and SUB2 orders are used for accessing arrays via dope vectors. For a general array:-

$$X[l1 : u1, l2 : u2, \qquad ln : un]$$

we want to access X[i1, i2, . . . in]. The address can be expressed in the form:-

$$X0 + (i1 - l1)*m1 + (i2 - l2)*m2 + . . . + (in - ln)*mn$$

where m1, m2, . . mn are suitable multipliers; in addition we must have l1 ≤ i1 ≤ u1, l2 ≤ i2 ≤ u2, .. ln ≤ in ≤ un. When the array is declared, a dope vector is created that contains a triple of 32-bit elements for each dimension of the array; a triple consists of the lower bound l, the multiplier m and a checking value c. For the array X above, the dope vector will look like:-

l1	m1	c1	l2	m2	c2	. . .	ln	mn	cn

A descriptor X' is created which points to this vector. To access the element, the appropriate sequence is:-

B = i1	1st subscript
SUB1 X'	Load XD with dope vector descriptor and clear D.
	Subtract l1 from B, multiply by m1, check result is in range
	0 ≤ B ≤ c1 and add to DO.
	(Hence DO = (i1 - l1)*m1 and l1 ≤ i1 ≤ u1).
B = i2	2nd subscript
SUB2	DO + (i2 -l2)*m2 l2 ≤ i2 ≤ u2
.	
.	
B = in	nth subscript
SUB2	DO + (in - ln)*mn ln ≤ in ≤ un

There are now two ways of accessing the element itself:-

RMOD X	or	B = DO
A = D[0]		A = X[B]

[The checking values c are clearly (u1 - l1 + 1)*m1.]

5.5. Store to Store Orders

The store to store orders fall into three main classes: string-string, byte-string and table-string orders; there is also one special table look-up order. The string-string orders operate on a source string and a destination string; operations are provided that move (i.e. copy), compare and logically combine the strings. The byte-string orders use a byte and a destination string; they are the same as string-string orders in which the source string consists of the specified byte repeated as often as necessary. The table-string orders make it possible to translate the characters of a string into a different code specified in a table or to check a string to see if it contains any of the characters specified in a table. The table look-up order scans the table for a particular element.

The MASK

For all the store to store orders except table look-up (TALU), bits 48-55 of the operand are an eight-bit MASK. In each byte processed by the order, bits corresponding to 1's in the mask are ignored; when any byte is used in an operation, the corresponding bits are taken to be zeros, and if a byte is put into store, the corresponding bits in the store are unaltered. For example, if MASK = 11000011, then a move order will only change bits 2-6 of the bytes in the destination string.

String-String Orders

For all string-string orders, XD contains the source string descriptor and D the destination string descriptor. The descriptors must have type 0, 1 or 2 and element size 8 bits, otherwise there will be an ITS interrupt. The operand defines the MASK (see above) and a FILLER. The FILLER is not in fact used for all the orders.

```
                 d48                    d63
OPERAND          |   MASK   |  FILLER    |
                       8          8
```

SMVB Moves one byte from source to destination. If the source string is a null string, move FILLER to destination. If DB = 0 there will be a BCH interrupt. Updates DO, DB, XDO, XDB.

```
[(if DB = 0  then BCH interrupt
if XDB = 0  then (FILLER => D[0]; MOD 1)
            else (XDO[0] => D[0]; XMOD 1)]
```

*SMVE Moves source to destination. If source is shorter than destination there will be an SSS interrupt; but note that this will simply terminate the operation if the inhibit bit SSSI is set. Updates DO, DB, XDO, XDB.

```
[Li: if DB ≠ 0 then (if XDB = 0 then SSS interrupt
else (XD[0] => D[0]; MOD 1; XMOD 1); -> L1)]
```

SMVF Moves source to destination. If the source runs out, then uses FILLER for remainder of destination. Updates DO, DB, XDO, XDB.

```
[ L1: if DB ≠ 0 then (SMVB OPERAND; -> L1)]
```

* See note below under Table-String Orders

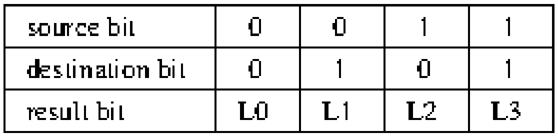
SCMP Compares source and destination strings looking for inequality in two corresponding bytes. If source runs out, FILLER is used. The test register is set = 0 if no inequality is found, > 0 if source byte > destination byte, < 0 if source byte < destination byte; for comparison purposes, the bytes are treated as unsigned integers. DO, DB, XDO, XDB are updated.

```
[L1: if DB = 0 then (T = '='; -> L4);
     if XDB = 0 then (if FILLER ≠ D[0] then -> L2 else
                          (MOD 1; -> L1))
     else if XD[0] ≠ D[0] then -> L3 else (MOD 1;
                          XMOD 1; -> L1);
L2: if FILLER < D[0] then T = '<' else T = '>' ; -> L4 ;
L3: if XD[0] < D[0] then T = '<' else T = '>' ;
L4:                       ]
```

SLGC Source and destination are logically combined and the result stored in destination. The logical operation is the same for each bit of each byte and is defined by bits 44-47 of the operand.

```
 d44 d45 d46 d47
| L0 | L1 | L2 | L3 |
```

The result of the operation is defined by the table below:-



source bit	0	0	1	1
destination bit	0	1	0	1
result bit	L0	L1	L2	L3

If the source string runs out, there will be an SSS interrupt.

```
[L1: if DB ≠ 0 then if XDB = 0 then SSS interrupt;
(XD[0] lgc D[0] => D[0]; MOD 1; XMOD 1; -> L1)]
```

Byte-String Orders

The byte-string orders are the same as the string-string orders except that the source consists of copies of the BYTE specified in operand bits 56-63. The MASK appears as usual in operand bits 48-55. D contains the destination string descriptor which must have type 0, 1 or 2 and element size 8 bits, otherwise there will be an ITS interrupt; XD is not used.

```
                   d48                 d63
OPERAND            |  MASK   |  FILLER  |
                   ----------------------
                        8         8
```

BMVB Moves one byte to destination.

BMVE Moves BYTEs to destination until full.

BSCN Scans destination looking for a byte = BYTE. The test register is set = 0 if an equality is found, < 0 otherwise. DO and DB are updated.

```
[L1: if DB = 0 then (T = ‘<’; -> L2);
     if BYTE = D[0] then (T = ‘=’; -> L2) else (MOD1; -> L1);
L2:                  ]
```

BCMP Scans destination looking for a byte ≠ BYTE. The test register is set as for SCMP.

BLGC Combines BYTE with destination string using logical operation defined by operand bits 44-47 as for SLGC. Result is stored in destination.

Table-String Orders

For both the table-string orders, XD contains a string descriptor which must have type 0, 1 or 2 and element size 8 bits. D may contain any descriptor. The operand contains no information other than the MASK, specified as usual in bits 48-55.

*TRNS Each byte of the XD string is processed in turn. First, it is used as a modifier for the descriptor in D to access a secondary operand. Then the least significant 8 bits of the secondary operand replace the original byte. There may be a BCH interrupt during the D access. D will usually contain a byte vector descriptor.

```
[L1: if XDB ≠ 0 then (D[XD[0]] => XD[0]; XMOD 1; -> L1)]
```

*TCHK Each byte of the XD string is accessed as above and used as a modifier for the descriptor in D. If the least significant bit of the secondary operand is a 1, then the operation is terminated with BN = 0. If no 1 is found for the whole of the XD string, then BN = 1. D will usually contain a bit vector descriptor.

```
[L1: if XDB = 0 then BN = 1 else
          (if l.s. bit of D[XD[0]] = 0 then (XMOD 1; -> L1)
          else BN = 0)]
```

* TRNS, TCHK, SMVE These orders are not commissioned[a]. If an attempt is made to execute any of them the effect will be that of a DUMMY order, except that an interrupt may occur if the wrong type, size or length has been specified as described above and TCHK will set the Test Register in an unspecified manner.

[a] i.e. in the particular MU5 processor built at the University of Manchester to execute this order code.

Table Look-Up

TALU This order enables a fast scan to be made for an element equal to the operand. D contains a descriptor that defines the table - origin in DO, length in DB. The length is expressed in byte units. The descriptor must have type 0 or 2 and element size 32 bits. XDO contains a MASK that is used in exactly the same way as the mask in the other store to store orders; bits corresponding to 1's in the MASK are ignored. The least significant 32 bits of the operand are compared with each element of the table in turn for equality (under control of MASK). If no equality is found, then the operation terminates with the test register set > 0 and the descriptor in D updated (DB = 0, DO points after the end of table). If inequality is found, then the test register is set = 0 and the descriptor in D will point to the element found, with the bound field updated.

```
[L1: if DB = 0 then T = '>' else
 (if operand ≠ D[0] then (MOD 1; -> L1) else T = '=')]
```

N.B. TALU takes operands directly from store.

Chapter 6 Organisational Orders

6.1 Introduction

The format for the organisational orders is shown below.

```
| cr = 0 | f'  |     N'    |
-----------------------------
     3       6         7
```

The cr bits are zero and the f' bits define the function to be performed. N' defines the operand for the order as described in Chapter 2; the only kind of operand that cannot be addressed is an internal register.

The organisational orders fall into the following groups:-

(a) Register operations - orders manipulating NB, XNB, SF, MS.

(b) Control transfers and procedure call orders.

(c) Conditional control transfers.

(d) Boolean orders - operating on the 1-bit Boolean register BN.

(e) Special orders.

6.2 Register Operations

NB, SF and XNB Orders

The registers NB, SF and XNB are defined in Section 2.2 and may be regarded as unsigned registers. In the following orders, it should be remembered that the least significant digit of each register is permanently zero, so that the least significant bit of the operand will have no effect.

NB = Load NB from bits 48-63 of the operand.
SF = Load SF from bits 48-63 of the operand.
XNB = Load XNB from bits 48-63 of the operand.

NB + Add operand bits 48-63 to NB; interrupt on segment overflow.
SF + As for NB +, but add to SF.
XNB + Add operand bits 48-63 to the least significant 16 bits of XNB;
interrupt on segment overflow (carry into top half of XNB is not allowed).

SF = NB + Add NB to operand bits 48-63 and store result in SF;
interrupt on segment overflow.
NB = SF + Add SF to operand bits 48-63 and store result in NB;
interrupt on segment overflow.

For each of the orders NB+, SF+, XNB+, SF = NB+ and NB = SF+, the base register (or registers) involved are unsigned integers but the operand is a signed 16-bit integer. The result must be in the range 0 ≤ result ≤ 2↑16 or there will be a segment overflow interrupt.

NB => The name segment number SN is stored at bits 32-47 and NB at bits 48-63 of the operand. The operand may not be a secondary operand.
SF => As for NB =>, but store SN and SF.
XNB => As for NB =>, but store (all 32 bits of) XNB.
SN = Load SN from bits 32-47 of the operand [bits 32-33 of SN remain = 0].
This order only alters SN if in Executive mode.

The Machine Status Register MS

The machine status register contains 16 bits of system information numbered MS0 - MS15. MS8 - MS15 are concerned with the interrupt organisation and can only be set in Executive mode; they are described in Chapter 7. MS0 is the ‘D set’ bit whose use is explained under the procedure call descriptor in Section 2.13. MS2 & MS3 are used in conjunction with the System Performance Monitor (Section 9.8). MS4 - MS7 are the test bits T0, T1, T2 and the Boolean BN, described in Sections 6.4 and 6.5.

	0	1	2 3	4	5	6	7	
MS	DS		SPM	T0	T1	T2	BN	EXECUTIVE
	1	1	2	1	1	1	1	8
		↑						
		Inhibit Program Faults						

MS = This order sets various bits of MS to 0 or 1 depending upon bits 32-63 of the operand.

Let $0 \leq i \leq 7$. Then

(a) if operand bit $(56 + i) = 0$, $MS(8 + i)$ is unaltered

(b) if operand bit $(56 + i) = 1$, $MS(8 + i)$ is set to operand bit $(48 + i)$

(c) if operand bit $(40 + i) = 0$, $MS(i)$ is unaltered

(d) if operand bit $(40 + i) = 1$, $MS(i)$ is is set to operand bit $(32 + i)$

If not in Executive mode, MS8 - MS15 are unchanged and only (c) and (d) above apply.

Example: The order MS = 00010111 11001111

would set MS8 = MS9 = MS12 = 0, MS13 = MS14 = MS15 = 1

and leave MS10 and MS 11 unaltered.

MS is also altered by the EXIT and RETURN functions (Section 6.3). When any order altering MS causes the By-pass CPRs[5] digit to be altered, an Acc => Z instruction must precede it in order that all store accesses will be completed before the CPRs are turned on or off. Care must also be exercised in turning the Name Store or Level 0 bit on or off, and when altering bits 12 and 13.

[5] The CPRs are the Current Page Registers in the Store Access Control Unit (see Appendix I)

6.3 Control Transfers and Procedure Calls

The link is a 64-bit register with format:-

MS	NB	CO
16	16	32

MS and NB are defined in Sections 6.2 and 2.2 respectively. CO is the 16-bit word address of the order currently being obeyed; the most significant bit of CO is always zero.

–> Relative jump; bits 32-63 of the highway are taken to be a signed 2's complement integer and are added to CO. An attempt to transfer control across a segment boundary will cause an interrupt.

JUMP: Absolute jump; CO is loaded from bits 33-63 of the highway.

STACKLINK CO and bits 32-63 of the highway are added as for the relative jump and the result, together with MS and NB is stored. Symbolically:-

STACK [MS, NB, CO + operand]

The addition CO + operand may give overflow as for –>.

RETURN The operand part of this order must specify the STACK. The order sets SF = NB and then unstacks the link. Symbolically:-

SF = NB
[MS, NB, CO] = [SF]
SF - 2

Bits 8-15 of MS are only reset if in Executive mode. If any operand other than STACK is specified, then the order is exactly the same as EXIT. (Note that if the operand specifies that the STACK is to be used as a descriptor, then SF is reset as above.)

SETLINK The link is stored at the address specified by the operand. Symbolically:-

[OPERAND] =[MS, NB, CO]

The operand may not be a secondary operand.

EXIT The link is reset from the operand. Symbolically:-

[MS, NB, CO] = [OPERAND]

Bits 8-15 of MS are only reset if in Executive mode.

The following example illustrates how STACKLINK and RETURN can be used to call a procedure P with three parameters A1, A2, A3. Note that procedure calls implemented in the compilers are slightly more complicated (see The MU5 Compiler Writers Manual). Before the call, the stack will be:-

```
   |    |    |    |    |    |    |    |    |    |
_______________________________________________________
                  ↑
                  SF
```

The call will look like:-

```
          STACKLINK L1
          STACK A1
          STACK A2
          STACK A3
          JUMP P
     L1:
```

so that after the call the stack looks like:-

```
   |    |    |    |LINK|   A1   |   A2   |  A3   |    |    |
_______________________________________________________________
                                             ↑
                                             SF
```

The procedure itself will contain orders:-

```
          PROCEDURE P
          NB = SF - 6     to set NB for use as a base in the procedure
          SF + n          for the local names of the procedure
          RETURN
```

The order NB = SF - 6 sets NB -> LINK in the stack; SF + n advances SF.

```
   |    |    |    |LINK|   A1   |   A2   |  A3   |    - - -    |    |
_______________________________________________________________________
                   ↑                                          ↑
                   NB                                         SF
```

The RETURN will reset MS, NB, CO from the LINK in the store and return SF to its original position.

6.4 Conditional Control Transfers

The test bits T0, T1, T2 are bits 4, 5 and 6 of the machine status register MS (see Section 6.2). They are set by the computational orders COMP and CINC (see Chapters 3 and 4) and by some of the store to store orders. The significance of these bits is generally as follows:-

T0	set to 1 if overflow
T1	set to 0 if result =0, i if result ≠ 0
T2	set to 0 if result ≥ 0, 1 if result < 0

A set of seven orders is provided that cause a relative jump if MS is suitably set. If the test succeeds, then the jump is carried out in exactly the same way as for –>.

IF = 0, –>	jump if T1 = 0
IF ≠ 0, –>	jump if T1 = 1
IF ≥ 0, –>	jump if T1 = 0 or T2 = 0
IF < 0, –>	jump if T2 = 1
IF ≤ 0, –>	jump if T1 = 0 or T2 = 1
IF > 0, –>	jump if T1 = 1 and T2 = 0
IF OVERFLOW, –>	jump if T0 = 1

There is an eighth conditional jump order that may be used to test the BOOLEAN, BN, described in the next section.

IF BN, –>	jump if BN = 1

6.5 Boolean Orders

The Boolean, BN, is bit 7 of the machine status register MS. There are two kinds of order that set BN. The first kind combines BN with the result of a test and uses the operand to define what logical operation to perform; the second kind combines BN directly with the operand.

The first kind of order tests MS in one of 8 ways to produce a result R equal to 0 or 1.

= 0	R = 1	if T1 = 0,	0 otherwise
≠ 0	R = 1	if T1 = 1,	0 otherwise
≥ 0	R = 1	if T1 = 0 or T2 = 0,	0 otherwise
< 0	R = 1	if T1 = 1,	0 otherwise
≤ 0	R = 1	if T1 = 0 or T2 = 1,	0 otherwise
> 0	R = 1	if T1 = 1 and T2 = 0,	0 otherwise
OVFLOW	R = 1	if T0 = 0,	0 otherwise
BN	R = 1	if BN = 1,	0 otherwise

Bits 59-63 of the operand define the way in which this result R is to be combined with BN as follows:-

0000	BN = 0	set BN = 0
0001	BN &	and BN with R
0010	BN /&	invert BN, then and with R
0011	BN =	load BN with R
0100	BN &/	and with inverse of R
0101	BN = BN	dummy order
0011	BN ≢	not equivalence with R
0111	BN V	or with R
1000	BN/&/	invert BN, then and with inverse of R
1001	BN ≡	equivalence with R
1010	BN/	invert BN
1011	BN/V (implies)	invert BN, then or with R
1100	BN =/	load BN with inverse of R
1101	BN V /	or with inverse of R
1110	BN/V/	invert BN, then or with inverse of R
1111	BN = 1	set BN = 1

The second kind of order uses 4 of the f' bits to specify the function as above and takes the operand R from the least significant bit of the highway.

6.6 Special Orders

XC0-6 Stack the operand and jump to segment 8193, 32-bit word locations 0-6 respectively[a].

DL = The 32 Display Lamps on the Engineers' Console (See Appendix III) are set equal to bits 32-63 of the operand. The Display Lamps may also be written to as a V-line (see Chapter 8).

SPM This function is for use with the System Performance Monitor associated with the MU5 Computer Complex[b].

[a] These functions set the Executive Mode bit in MS, as noted in Section 7.2.

[b] *i.e.* it does not affect the MU5 processor in any way, it simply sends a pulse to the SPM.

Chapter 7 The Interrupt System

7.1 The Interrupt Structure

There are eight types of interrupt divided into two groups of four, the System interrupts and Process based interrupts. The system interrupts are concerned with activities external to the current process (e.g. peripheral control). The process based interrupts occur as a result of specific actions in the current process. The interrupts are shown below, each associated with a three bit interrupt number.

SYSTEM INTERRUPTS	000	System Error
	001	CPR Non-Equivalence
	010	Exchange
	011	Peripheral Window
PROCESS BASED INTERRUPTS	100	Instruction Count Zero
	101	Illegal Orders
	110	Program Faults
	111	Software Interrupt

When interrupts occur simultaneously, the first to be dealt with is the one with the smallest interrupt number.

When an interrupt occurs, the hardware stops what it is doing and enters an interrupt sequence. During this entry sequence the 'Interrupt Entry Bit' is set. This allows the sequence to run in a special non-interruptible mode of operation described by the table in Section 7.2.

The first action of this sequence is to retain the state of the machine in a compact form to allow a straightforward return to the current process after dealing with the interrupt. This is achieved by storing a 64-bit link word. The format of this link is the same as that of the control register consisting of 16-bit Machine Status register[6] (16 bits), Name Base register (16 bits) and the 32-bit control address.

In addition to retaining this link, the interrupt sequence also transfers control to the appropriate interrupt procedure. This control transfer is achieved by resetting

[6] In the original version of Chapter 7, this register was called the Machine State register, which is inconsistent with earlier chapters, so it has been corrected herein to avoid confusion.

the 64 bits (MS, NB, CO) from the second half of the Ith double word entry of a table (I is the interrupt number). The first word of this entry is used to hold the stored link. This table is 16 x 64 bits long and starts at word 16 of the first of the common segments (segment 8K)[7].

When a CPR ≢ interrupt occurs the link will point to the next instruction to be obeyed but the previous instruction may not be complete. Incomplete instructions are held in the OBS buffer and the CPR ≢ interrupt routine must preserve (and restore) this buffer before using any instructions which could alter its content. On other interrupt entries, all instructions up to the one addressed by the link will be complete.

Unserviced interrupts may be read in PROP V-line 26.

Before discussing the interrupts in detail, it is necessary to describe the Machine Status register.

[7] It would have been helpful if this description had referred to the mechanism used to implement this sequence *i.e.* the use of two hard-wired instructions:-

SETLINK	System V-Store (k' = 7) Base = 0 Name = 2*I
EXIT	System V-Store (k' = 7) Base = 0 Name = 2*I + 1

with System V-Store addresses being mapped to segment 8192.

7.2 The Machine Status Register

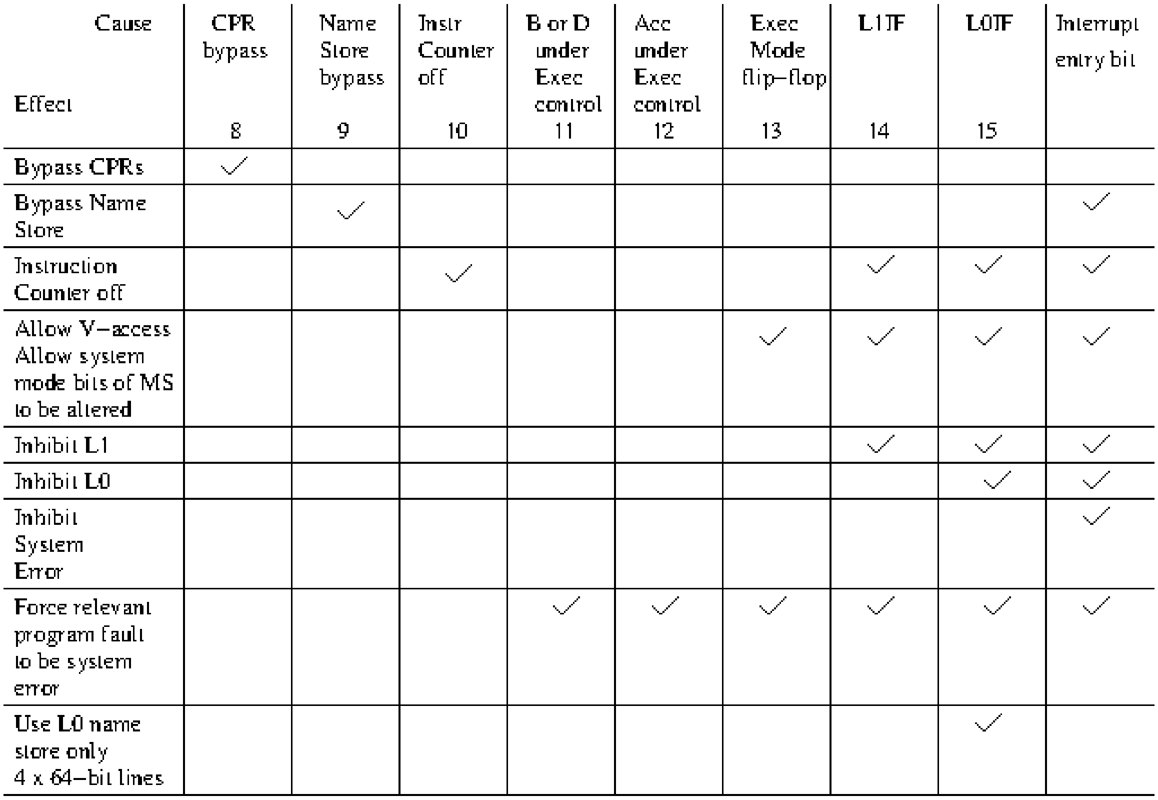
The machine status register (MS) is 16 bits long and only bits 0, 1, and 4-7 may be directly altered by user programs; bits 4-7 are test register bits.

Bits 2 & 3 & the l.s. 8 bits are known as the system mode bits and they may be altered if the Executive mode bit is set. This bit is set by interrupt entry or by the functions $XC_0 \ldots XC_6$ (see Section 6.6). MS is arranged as follows:-

BIT	
0	Force D[ ] instead of S[ ]
1	Inhibit program fault interrupts (A, B, D, etc.)
2	System Performance Monitor
3	0 - Runs in mode set on console
	1 - Run CPU on NO OVERLAP if MS02 = 1
4	Overflow
5	≠
6	-ve
7	Boolean
8	Bypass CPRs
9	Bypass Name Store
10	Instruction Counter Inhibit
11	B and D faults to System Error in Exec mode
12	A faults to System Error in Exec mode
13	Executive mode flip-flop
14	Level 1 Interrupt flip-flop (L1IF)
15	Level 0 Interrupt flip-flop (L0IF)

The use of bits 0, 1 and 4-7 is described in Section 6.2. The use of bits 2 and 3 is described in Section 9.8.

Effects of System Mode Bits



Cause Effect	CPR bypass 8	Name Store bypass 9	Instr Counter off 10	B or D under Exec control 11	Acc under Exec control 12	Exec Mode flip–flop 13	L1IF 14	L0IF 15	Interrupt entry bit
Bypass CPRs	✓								
Bypass Name Store		✓							✓
Instruction Counter off			✓				✓	✓	✓
Allow V–access Allow system mode bits of MS to be altered						✓	✓	✓	✓
Inhibit L1							✓	✓	✓
Inhibit L0								✓	✓
Inhibit System Error									✓
Force relevant program fault to be system error				✓	✓	✓	✓	✓	✓
Use L0 name store only 4 x 64–bit lines								✓	

B or D interrupts will be forced as system errors if bit 11 is set and an ACC interrupt will be forced as a system error if bit 12 is set. All other program faults will be forced as system errors if bit 13, 14 or 15 or the Interrupt bit is set.
(N.B. The Name Store is bypassed when the interrupt entry bit is set since access to common segments cannot be made via the name store.)

L0 Name Store

In L0 mode (bit 15 set) the 8 32-bit lines of Name Store are used as fast registers. The hardware interprets only the bottom 3 bits of a ‘name type’ operand address and maps it into this name store. No store accesses occur. These fast registers may be used as 8 x 32-bit names or 4 x 64-bit names.

7.3 System Interrupts (Level 0)

The procedures that service these interrupts must be written so as not to cause any other Level 0 system interrupts, for example, the Peripheral Window Interrupt procedure should not cause a CPR Non Equivalence Interrupt. This requires that a few CPRs are permanently allocated to cover the program and working store used by these Level 0 Interrupt procedures [in fact there were 4 - see Section 8.6]. If a CPR ≢ interrupt occurs while the Level 0 Interrupt flip-flop is set, then a System Error Interrupt is caused. This also occurs if an illegal hardware function is executed while the Level 0 Interrupt flip-flop is set. In this sense the System Error Interrupt differs from the other System Interrupts. However, once it does occur, it cannot recur until the System Error Status register is reset or unless the Engineers 'Interrupt' is pressed.

The System Error Interrupt

A System Error Interrupt is caused by a hardware or Executive failure. The System Error Status register shown below can pinpoint the exact cause. The software action is to perform error diagnosis and restart normal operations if possible.

bit	error indication
48	Engineers Interrupt (Console - forces CPR bypass)
49	Early Warning Power Failure
50	SAC Parity
51	Name Store Multiple Equivalence
52	OBS Multiple Equivalence
53	CPR Multiple Equivalence
54	Spare
55	IBU Multiple Equivalence
56	B or D error & (MS11)
57	Acc error & (MS12)
58	Illegal function & (L0IF + L1IF + EXEC)
59	Name adder overflow & (L0IF + L1IF + EXEC)
60	Control adder overflow & (L0IF + L1IF + EXEC)
61	CPR exec illegal
62	CPR ≢ & (L0IF)
63	Spare

CPR Non-Equivalence Interrupt

This interrupt can be produced by a user program or by an Executive mode procedure. It occurs when an attempt is made to access an address that does not lie within the address field defined by the contents of the CPRs. if the required address lies in the local store, the procedure will free a CPR and allocate it to the page containing the address. Control is then returned to the interrupted process.

If the page containing the required address is not in local store, then the procedure will locate the page and organise its transfer to local store. In this case control is not returned to the interrupted process (which is halted awaiting the termination of the transfer) and a process change may occur.

Exchange Interrupt

This interrupt is set by the Block Transfer Unit on completion (or termination) of a Core to Core Transfer[8].

Peripheral Window Interrupt

This interrupt is caused by an external device (e.g. peripheral processor, drum) writing to the Peripheral Window V-line or by an interrupt from the Console (e.g. TTY, CLOCK). The two are distinguished by bits 61 and 60 in V-line %011A, the first indicating peripheral window. The information sent to this 32-bit V-line consists of a sending unit number and a message. The Peripheral Window procedure must queue up this message for subsequent processing.

Writing to the Peripheral Window V-line sets it not busy in readiness for another message. The Peripheral Window procedure runs with interrupts inhibited, so a subsequent message won't be acknowledged until the procedure is exited.

This interrupt is also entered for console interrupts (Teletype and clock). PROP V-line 26 contains the cause of the interrupt.

[8] This terminology is a hangover from the days of Atlas, which had a ferrite core main store. In the case of MU5, the first level of backing store was a ferrite core Mass Store but the Local Store was built using plated-wire technology. Also, the required page might have been on the second level of backing store, the Fixed-head (magnetic) Disc Store (see Appendix I.)

7.4 Process Based Interrupts (Level 1)

The Instruction Counter Zero Interrupt

This interrupt is set when the instruction counter becomes zero. It may be inhibited by the 'instruction counter inhibit' bit being set in the Machine Status Register.

The Illegal Order Interrupt

This interrupt is caused by program fault conditions detected by the hardware; these conditions set bits 48-53 in the program fault status V-line (see next section).

The Program Fault Interrupt

This interrupt is also caused by program fault conditions detected in the hardware that set bits 56-58[9] in the program fault status V-line.

The following table relates the assignment of bits in the Program Fault Interrupt Status V-line to specific fault conditions:-

V-line bit	condition
48	Illegal function & $\overline{\text{L0IF + L1IF + EM}}$
49	Name Adder overflow & $\overline{\text{L0IF + L1IF + EM}}$
50	Control Adder overflow & $\overline{\text{L0IF + L1IF + EM}}$
51	Illegal V store access
52	CPR $\overline{\text{EXEC}}$ illegal (illegal access via CPRs when not in Exec mode)
53	Parity
54	System Performance Monitor (see Section 9.8)
55	Spare
56	B fault & $\overline{\text{MS1}}$
57	D fault & $\overline{\text{MS1}}$
58	Acc fault & $\overline{\text{MS1}}$

When a program fault occurs, the hardware may not wait until the arithmetic and control units have finished their current operation(s), so that multiple fault conditions may not be completely recorded in the Program Status register.

The Software Interrupt

This interrupt occurs in user mode only when the software interrupt bit is set (see Section 8.3).

[9] After the SPM was built, bit 54 was included in this set.

Chapter 8 The V-Store

8.1 Introduction

The V-store contains hardware registers used to control and/or diagnose parts of the MU5 processor. The V-store does not include the Internal Registers, which are addressed by a different mechanism. The V-store is not generally accessible to a user program but is accessible from within MU5 to Executive, interrupt routines (Section 7.2) and certain control or hardware diagnostic programs.

The figure in Section 1.2 shows the instruction format required to obtain a V-store operand.

The V-store is divided into 128 blocks of 256 lines each. Each line is normally a 64-bit quantity but many of the lines will contain less than 64 bits, in which case the bits will appear right justified in a 64-bit word. The bits in a V-line are numbered as they would appear on a 64-bit wide highway. The following table gives the allocation of block numbers to section of the V-store in the central part of the machine.

BLOCK NO.	V-STORE TYPE
0	SYSTEM V-STORE (S8192)
1	PROP V-STORE
2	OBS V-STORE
3	CONSOLE V-STORE
4	SAC V-STORE
5	IBU V-STORE
6	PERIPHERAL V-STORE
7	PARITY V-STORE

8.2 System V-store (S8192)

These 256 V-line addresses are mapped into the first 512 x 32-bit words in segment 8192, the first of the common segments, by the PROP before using them to access store. This gives the executive a simple means of communicating between processes and approximates to the Atlas working store.

Starting at the 32nd 32-bit word of segment 8192 are 8 pairs of new and old links used by the interrupt entry sequence.

V-line	Virtual Address (S8912)	Name	size
(Decimal)	Hex specifying		
64-bit	32-bit boundaries		
boundaries			
16	20	System Error Old Link	64
	22	System Error Entry Link	64
18	24	CPR ≢ Old Link	64
	26	CPR ≢ Entry Link	64
20	28	Exchange Old Link	64
	2A	Exchange Entry Link	64
22	2C	Peripheral Window Old Link	64
	2E	Peripheral Window Entry Link	64
24	30	Instruction Count Old Link	64
	32	Instruction Count Entry Link	64
26	34	Illegal Order Old Link	64
	36	Illegal Order Entry Link	64
28	38	Program Fault Old Link	64
	3A	Program Fault Entry Link	64
30	3C	Software Old Link	64
	3E	Software Entry Link	64

8.3 Primary Operand Unit V-store (Block 1)

The following is a list of the registers in the V-store of the Primary Operand Unit, giving size, type of access and references to more detailed descriptions of usage and construction:-

Address (Decimal) 64-bit boundaries	Name	Size	Access
0	PROGRAM FAULT STATUS	16	R/W=Reset

This line records fault reasons for the Program Fault Interrupt (bits 56-58), the Illegal Order Interrupt (bits 48-53) and the System Performance Monitor (bit 54) (Section 7.4).

1	SYSTEM ERROR STATUS	16	R/W=Reset

This line contains the flip-flops used to record system error conditions for the System Error Interrupt (Section 7.3).

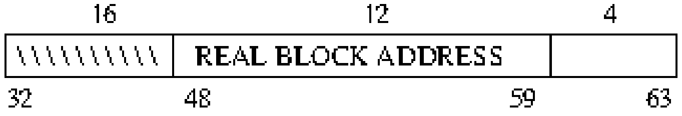
2	PROCESS NUMBER	4	R/W

This line contains the number of the current process and is used to generate the 'P' part of a virtual address. Writing to this line clears the JUMP TRACE in the Instruction Buffer Unit by resetting the Valid bits for each line (Section 8.7).

3	INSTRUCTION COUNTER	16	R/W

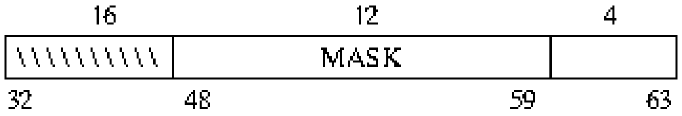
This contains the number of instructions remaining to be executed before the Instruction Counter Interrupt occurs (i.e. 64K machine instructions/Instruction Count Interrupt.) The counter may be stopped by setting MS10.

8	SEARCH ADDRESS	12	W



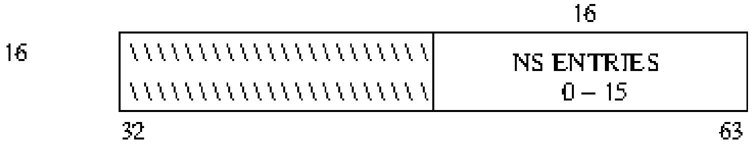
Bits 48-59 specify the block address of a 16 x 32-bit word block in the name segment of the process specified in PROCESS NUMBER (line 2). The line number is ignored. Writing to this line causes an associative search of the name store using the SEARCH MASK (line 9) which must have been set up previously. If a line of the specified block exists in the name store, the test register is set non-zero.

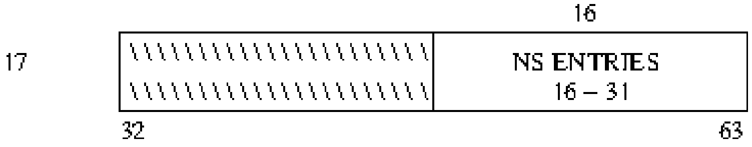
9 SEARCH MASK 12 W



The mask operates on the search (block) address. A ‘1’ specifies that the bit is to be ignored.

16/17 NS LINE COPY 16+16 R
(%10/%11) These lines contain a bit significant indicator showing which name store entry received the last valid name store access.



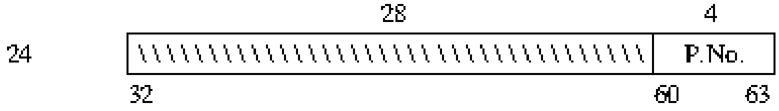


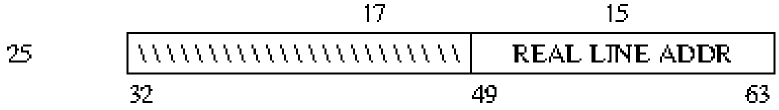
These lines have only READ access. However, writing to these addresses results in hardware action in the PROP V-store without disturbing NS LINE COPY.

Writing to the address line 16 causes LINE POINTER to be reset. This is a hardware pointer to an entry in the name store. Resetting sets the pointer to the first entry in the name store. The pointer can only be altered by reading line 24 which causes it to be incremented by 1 (modulo 28). Writing to address line 17 causes all entries in the name store to be marked unused and unaltered. The core copies of any existing entries are not updated.

24/25 NS NEXT LINE VIRTUAL ADDRESS 4+15 R

(%18/%19) These lines contain the virtual address in the associative name store pointed to by the LINE POINTER.





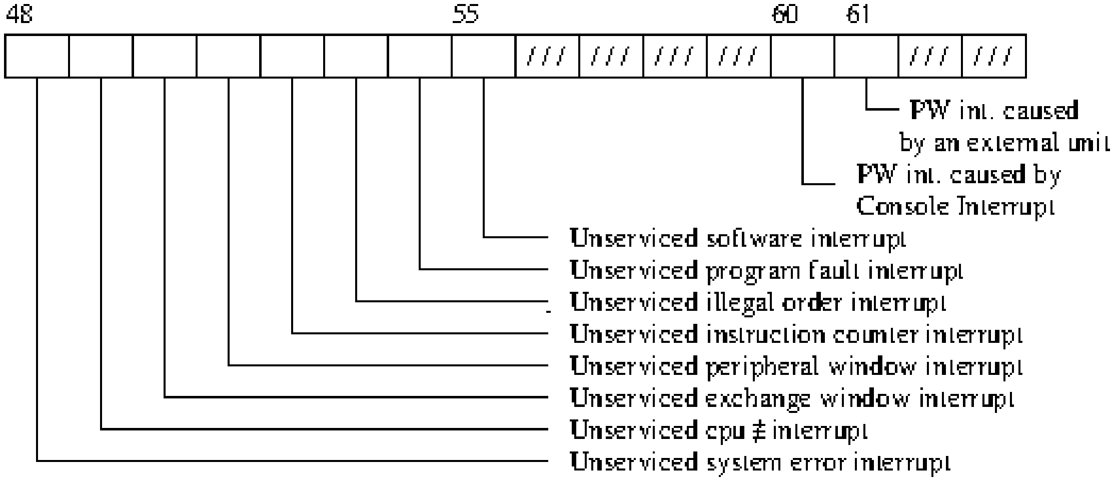
The normal virtual address format contains a segment number. Here the segment number is implied. It is the name segment of the process given by line 24. The line address in line 25 references a 64-bit word boundary in the name segment.

Reading line 24 causes the name store LINE POINTER to be incremented (cyclic modulo 28).

Although access to these line is limited to READ only, writing to the address line 24 causes special action without the contents of line 24 being disturbed. Writing to line 24 causes the name store to be purged throughout and the line pointer to be reset.

26 DISPLAY LAMPS 32 R/W

(%1A) Writing sets the engineers' display lamps. When read, the following bits have the meaning:-


48
55
60
61
///
///
///
///
///
///
PW int. caused by an external unit
PW int. caused by Console Interrupt
Unserviced software interrupt
Unserviced program fault interrupt
Unserviced illegal order interrupt
Unserviced instruction counter interrupt
Unserviced peripheral window interrupt
Unserviced exchange window interrupt
Unserviced cpu ‡ interrupt
Unserviced system error interrupt

27 SOFTWARE INTERRUPT 1 R/W
(%1B) Bit 63 of this line is set by the software trapping mechanism and by process-based interrupts (Section 7.1). It only causes an interrupt in user mode.

8.4 The OBS V-store (Block 2)

The following short description of the main features of the OBS system will clarify points in OBS V-store control.

The OBS contains an operand store similar to the PROP name store but which is not restricted to dealing with only name segment operands (names). All operands that are not names and all operands in accumulator orders (names included) are buffered in the OBS operand store. (This means that the OBS has a name store of its own and the PROP name store only buffers names associated with ‘non-accumulator’ orders.)

An entry in the operand store consists of virtual address and contents; this means an operand may be altered in the operand store without the main store version having been updated. The update only takes place when the operand is displaced by a new operand request. This replacement is normally cyclic, depending on other activities in the OBS.

A request to the OBS consists of a function and its operand. All accumulator orders are passed to OBS which queues the function part and if necessary buffers the operand. The Acc queue has a maximum of six entries, each of which references an entry in the operand store. Operands referenced by the Acc queue are avoided by the operand replacement mechanism described above.

A ‘non-accumulator’ order sent to the OBS bypasses the queuing mechanism and may be dealt with out of program sequence provided there is no possibility of a clash between its operand and those referenced by the Acc queue.

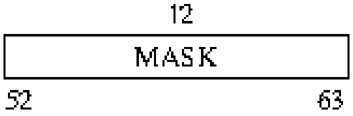
When an Acc order causes a CPR ≢, processing of the Acc queue is halted. The leading entry is responsible for the ≢ and the remaining functions can only be processed sensibly when the ≢ has been serviced.

The following is a description of the OBS V-lines.

Address (Decimal)	Name	Size	Access
0	OBS.CLEAR		
	OBS.PURGE	1	W

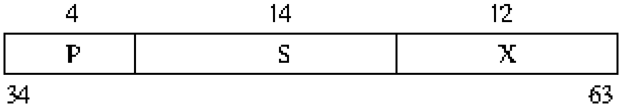
Writing a ‘0’ to bit 63 causes the OBS to be CLEARED. This means all ‘altered’ operands in the OBS operand store are written back to main store. They are also retained in the operand store and reset to ‘unaltered’. Writing a ‘1’ to bit 63 causes the OBS to be purged. This means all ‘altered” operands are written back as in the CLEAR but in addition all lines in the operand store are set empty, i.e. the OBS operand store is left in a RESET state.

1	(unassigned)		
2	OBS.MASK	12	W



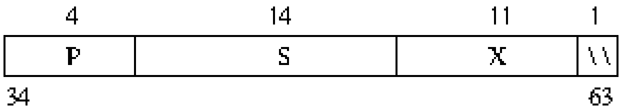
This is used to mask the X field of the OBS.FIND line (see below). A ‘1’ in the mask causes the corresponding bit in the Find operation to be ignored.

3	OBS.FIND	30	R/W

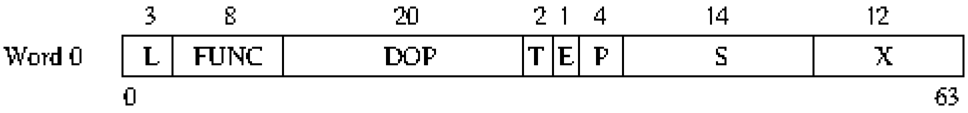


This address defines an 8 x 64-bit block boundary. Writing to this V-line initiates a masked associative search of the OBS operand store (see also line 2). If association equivalence occurs, bit 63 of this V-line is set to a ‘1’; otherwise it is set ‘0’. If association equivalence occurs, the test register is set non-zero, otherwise the test register is set equal to zero.

4 OBS.DUMP 29 W



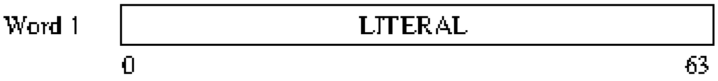
Writing to this line causes the functions in the Acc queue and their operands to be written to the specified 16 x 64-bit block. All useful information in the operand store is retained (i.e. the lines are not all RESET) but the Acc queue is returned to the 'empty' state. Each dumped entry consists of 2 x 64-bit words.



This contains the Acc function (FUNC) and some hardware information (DOP). The operand virtual address is P, S, X, L and the mode at the time of access is in E (executive). The T bits specify the type of operand:-

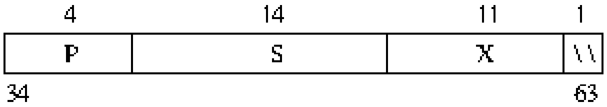
Type no	OPERAND
0	Invalid (Empty entry)
1	Literal
2	Vector
3	Name

If the operand is a literal, the virtual address field is irrelevant and the literal is held in the second word of the entry.



Note that when the operand is not a literal type, word 1 is irrelevant.

5 OBS.UNDUMP 29 W



The above address specifies the 16 x 64-bit block from which the OBS is to be reloaded. Writing to this V-line firstly causes a CLEAR operation to be performed (see line 0). Then the Acc queue is retrieved. Operands (other than literals) associated with the new Acc queue consist only of a virtual address part (see line 4 - OBS.DUMP). Any such operand which does not exist in the operand store at this time is now inserted in an 'unfilled' state, i.e. its 'contents' are not accessed from main store at this stage (see line 6 - OBS.RESTART).

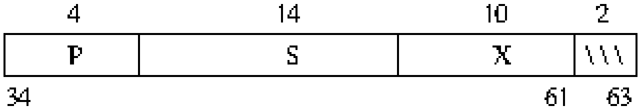
No OBS orders may be executed between the Undump and Restart/Exit orders.

6 OBS.RESTART - W

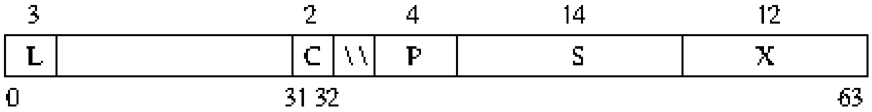
An UNDUMP operation can leave the operand store with 'unfilled' operands. Such operands have to be filled by causing accesses to main store before normal operation may continue. Writing to this V-line causes the above action to be initiated when the next EXIT or RETURN order is obeyed. No order that makes use of the OBS must appear during this period. The restart sequence forces all its operand accesses in executive mode. This means that if a parity occurs at this time, a System Error will be generated.

7 (Unassigned)

8 OBS.INSPECT 28 R/W



Writing to this line causes all virtual addresses in the operand store to be written to the specified 32 x 64-bit block of store. Each virtual address will have the following format:-



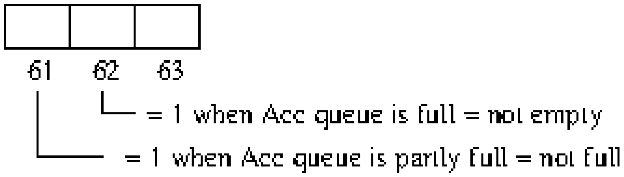
P, S, X and L are described above (Line 4).

The C bits have the following meaning:-

bit 31 = 1 means 'in use'

bit 32 =1 means 'referenced' from the Acc queue'

Reading this V-line yields the following information:-


61
62
63
= 1 when Acc queue is full = not empty
= 1 when Acc queue is partly full = not full

NOTES

OBS V-store addresses should not be used with organisational functions. In REMOTE mode, if PROP is sending orders to OBS, any attempt to access OBS V-store from a PPU through Exchange may produce spurious results.

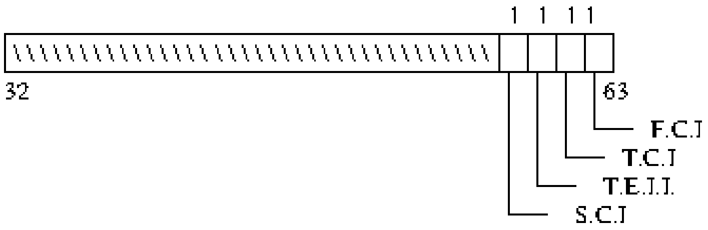
15 OBS.RESET

(%F) Writing to this V-line causes a reset of the OBS Buffer Store. Lines will be set 'not in use' and unaltered. No updating of the main store will take place.

8.5 Control Console V-store (Block 3) [c.f. Appendix III]

The console V-lines are as follows:-

Address	Name	Size	Access
(Decimal)			
64-bit boundaries			
0	CONSOLE INTERRUPT	4	R/W



Each of these is '1' when set.

F.C.I is the fast clock interrupt (currently 1/100 sec).

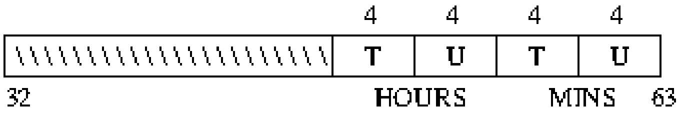
T.C.I is the teletype character interrupt

T.E.I.I. is the teletype external incident interrupt.

S.C.I is the slow clock interrupt (currently 1 sec).

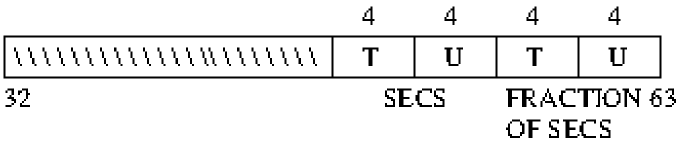
External incidents are 'accept', 'cancel' and 'input request' (see Line 7). Writing to this line cannot cause the T.E.I.I bit to be reset. This can only be achieved by resetting line 7. Writing a '1' to bit 62 resets bit 62. Writing a '1' to bit 63 resets bits 60 and 63.

2	TIME UPPER	13	R



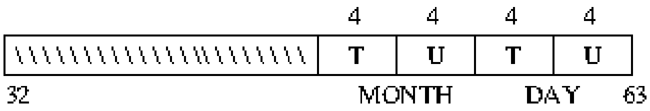
Hours and minutes appear in binary coded decimal in tens and units as shown.

3	TIME LOWER	15	R



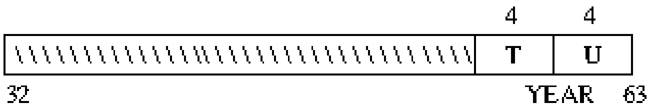
Seconds and fractions of seconds in b.c.d. as shown. This line is staticised by reading TIME.UPPER.

4 DATE LOWER 11 R



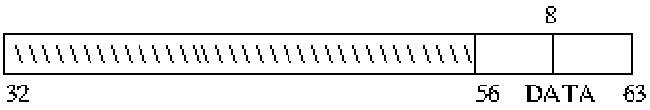
Months and days appear in binary coded decimal in tens and units as shown.

5 DATE UPPER/HOOTER 8 R



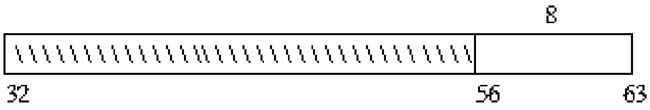
The year is in b.c.d. Although this line has read only access, writing to bit 63 at this address will operate the hooter [The hooter was a loudspeaker - writing a 0/1 moved the diaphragm in/out].

6 TELETYPE DATA 8 R/W



To output a character, TELETYPE CONTROL must be in output mode, then writing to this line starts the transfer. On reading, BIT 56 is the character parity

7 TELETYPE CONTROL 8 R/W



Each bit has individual significance and is ‘1’ when set active.

Digit	
56	PRINT ON/OFF (OFF = ‘1’)
57	CANCEL INSTRUCTION
58	Input/Output Teletype (‘1’ for input)
59	‘Input Request’

60 ‘Accept’

61 ‘Cancel Message’

62 Teletype Start

63 Teletype online

When the TTY is online, bits 59, 60, 61 will cause an interrupt.

ON LINE (Peripheral)

PRINT	$\overline{\text{LIT}}$	LIT	CANCEL MESS	ACCEPT MESS	INPUT REQUEST

OFF LINE (Instruction Source)

PRINT	LIT	$\overline{\text{LIT}}$	CANCEL INST		MUST BE LIT

Depress ’OFF LINE’

10 MODE SWITCHES 16 R

(%A) The least significant 8 bits and the most significant 4 bits of this line specify various modes of operation when set to ‘1’.

Digit

48-55 These are used for switching the stacks of the Local and Mass stores off-line.

56 Level 0.

57 Inhibit Clock Interrupt 1.

58 Inhibit Interrupts.

59 No overlapping of instructions.

60 Bypass Name Store.

61 Inhibit Clock Interrupt 0.

62 Allow Exchange Resets.

63 Reset Parity.

11 ENGINEERS HANDSWITHCHES 16 R

(%B) All 16 bits (48-63) are used to control hardware diagnostic programs and error recovery procedures.

12 (%C)	ENGINEERS CONTROL SWITCHES	10	R

These appear in the 10 l.s. bits of the line and have the following significance. In 'Auto' all bits are zero.

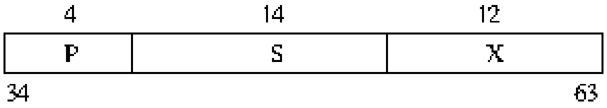
Digit	
54	Remote OFF/ON (Chapter 9)
55	Reset
56	Interrupt
57	Single Shot
58	KCs (i.e. not TEST)
59	STEP (i.e. not AUTO)
60	Increment OFF
61	PREPULSE ON
62	HANDKEYS for instruction source (i.e. not TELETYPE)
63	Instruction buffer/manual instruction

N.B. The above description of the Console V-store is only true as long as the REMOTE switch is OFF (digit 54 line 12). If REMOTE is ON, lines 10, 11 and 12 have their access permission increased from READ only to READ/WRITE with the exception of bits 54 and 62 of line 12 which remain READ only.

8.6 SAC V-store (Block 4)

The following is a list of registers in the V-store of the SAC unit. The list gives the address size, access and references to more detailed descriptions and constructions:-

Address (Decimal) 64-bit boundaries	Name	Size	Access
0	CPR SEARCH	30	W

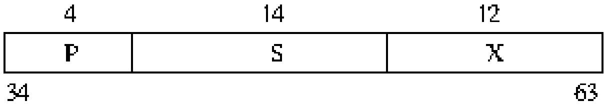


The use of this line is described later under line 5, the CPR FIND vector.

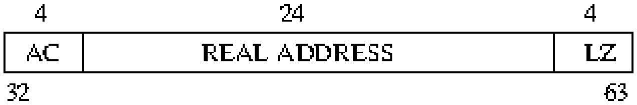
1 CPR NUMBER 5 W

This contains the 5-bit CPR number (0-31).

2 CPR VA 30 R/W



3 CPR RA 32 R/W



The 4 LZ bits represent the page sizes 64K - 16 words as the values 12-0 respectively. The AC bits are the access control on the page (shown below).

bit 32 = 0	Executive Mode only
= 1	Any
bit 33	READ Permission
bit 34	WRITE Permission
bit 35	OBEY Permission

Lines 1-3 are used for reading from and writing to one of the 32 CPRs[10]. Each of these is conceptually divided into Virtual Address part and Read Address part of the format illustrated in CPR VA and CPR RA. Writing to either CPR VA or CPR RA initiates a write operation to the VA half or RA half of the CPR specified by the contents of CPR NUMBER. Similarly, reading from CPR VA or CPR RA initiates a read from the VA or RA half of the CPR specified. A restriction involved in loading a CPR is that the Real Address part must be written to immediately before the Virtual Address part. This restriction does not apply when reading from a CPR.

[10] CPRs 28-31 were permanently allocated to Level 0 Interrupt procedures; the hardware of CPR 31 was modified to allow it to map 1 Mword pages.

4 CPR IGNORE 32 R/W

This consists of 32 bits, each of which corresponds to a CPR and when set to ‘1’ means that the CPR is empty. This vector of bits is ordered so that the most significant refers to CPR 0.

5 CPR FIND 32 R/W

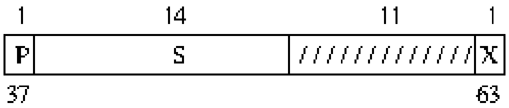
This has the same format as CPR IGNORE but is used in conjunction with the CPR SEARCH (line 0) and CPR FIND MASK (line 9) in an equivalence search through all the CPR registers. The CPR FIND MASK line specifies which bits in the PSX part of the virtual half of the CPRs are not used in the equivalence check and CPR SEARCH specifies the required bit pattern. Writing to CPR SEARCH initiates the operation. Each CPR that causes equivalence has a ‘1’ ORed into its corresponding bit in the CPR FIND line.

6 CPR ALTERED 32 R/W

7 CPR REFERENCED 32 R/W

Lines 6 and 7 are vectors of the Altered and References bits. These lines have the same formats as lines 4 and 5. An attempted access via a CPR causes a bit to be set in line 7 (& 6 if write access) even on access violation. Writing to a CPR (line 2) resets its bit in lines 4 - 7.

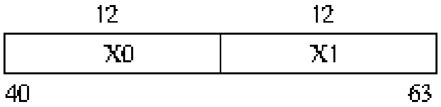
9 CPR FIND MASK 16 W



The use of this line is described under line 5 CPR FIND. The Find mechanism operates over each bit of the segment field whereas the P and X fields both merely have a ‘do’ or ‘do not’ single bit specification. A ‘1’ means do not search on this bit.

11 CPR X FIELD 24 R

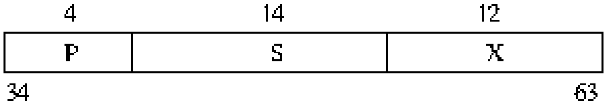
(%B)



Reading this register initiates a read from the Virtual Address half of the CPR specified by CPR number. This line is required for engineering purposes and is fully described in the MU5 Hardware Manual, Chapter 6.

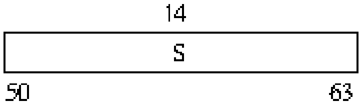
16 CPR NOT EQUIVALENCE PSX 30 R/W=Reset

(%10)



17 CPR NOT EQUIVALENCE S 14 R

(%11)

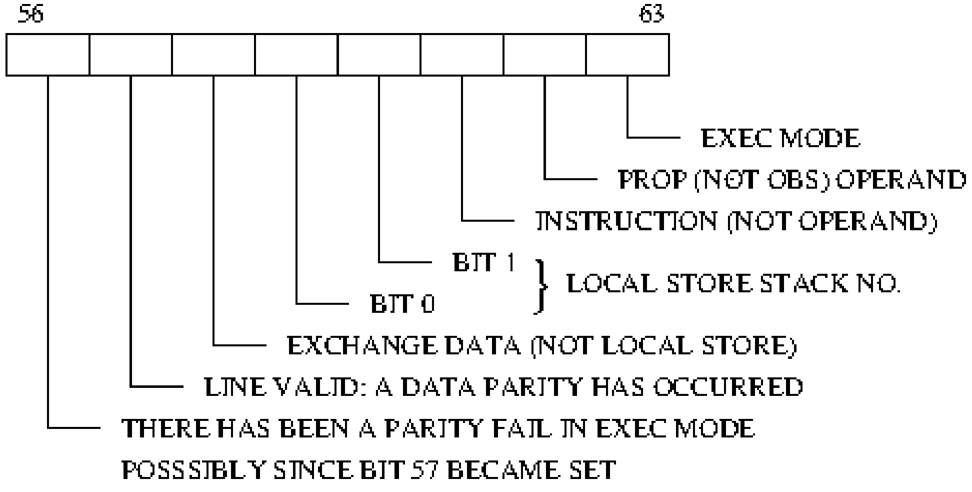


Line 16 holds the virtual address of the 16-word block that contains a line address that is to be presented to the CPRs for equivalence. Line 17 holds the segment field of this address. When a CPR Not Equivalence occurs, these lines remain set although further addressing though the CPRs can take place.

The action of writing to the PSX line returns both to their normal state. Any CPR ≢ interrupts that occur during the CPR ≢ interrupt procedure will be monitored as systems errors. If these occur before the PSX line has been reset, there will be no information in the PSX about the address causing the system error CPR ≢.

20 SAC PARITY 8 R/W=Reset bit 57

(%14)

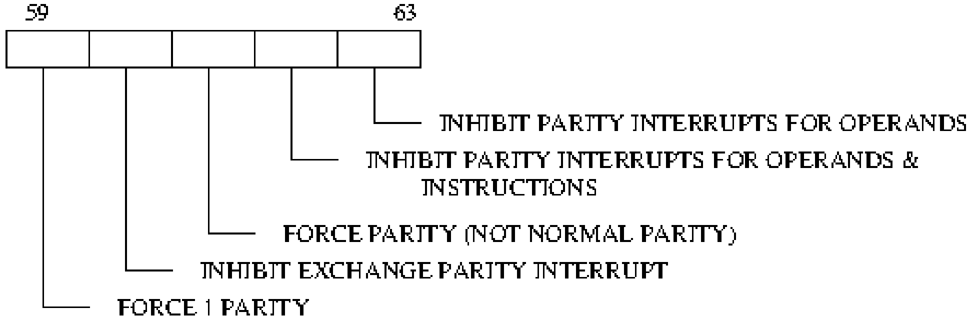

56
63
EXEC MODE
PROP (NOT OBS) OPERAND
INSTRUCTION (NOT OPERAND)
BIT 1
BIT 0
LOCAL STORE STACK NO.
EXCHANGE DATA (NOT LOCAL STORE)
LINE VALID: A DATA PARITY HAS OCCURRED
THERE HAS BEEN A PARITY FAIL IN EXEC MODE
POSSSIBLY SINCE BIT 57 BECAME SET

Bits 56 and 57 are reset by:– GENERAL RESET
WRITING TO THIS LINE
WRITING TO LINE 23
WRITINGTO BLOCK 7 LINE 1

A data parity error locks out bits 57-63 until reset.
Bit 56 records the occurrence of all exec. parity fails.

21 SAC MODE 5 R/W

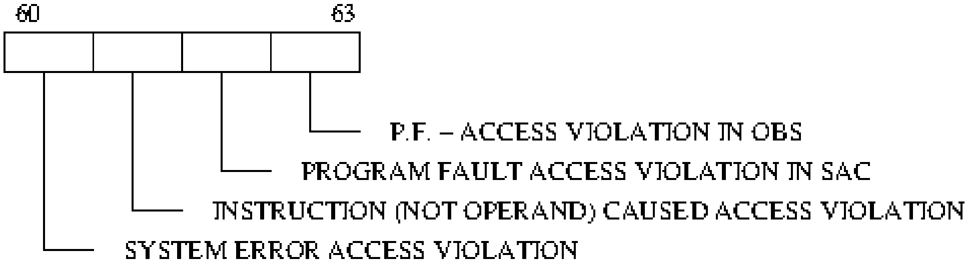
(%15)


59
63
INHIBIT PARITY INTERRUPTS FOR OPERANDS
INHIBIT PARITY INTERRUPTS FOR OPERANDS & INSTRUCTIONS
FORCE PARITY (NOT NORMAL PARITY)
INHIBIT EXCHANGE PARITY INTERRUPT
FORCE 1 PARITY

Interrupt inhibits do not inhibit the setting of the parity fail bits in lines 20 and 23. All bits in this line are cleared by general reset.

22 ACCESS VIOLATION 3 R/W=Reset

(%16) Bits 61 & 63


60
63
P.F. – ACCESS VIOLATION IN OBS
PROGRAM FAULT ACCESS VIOLATION IN SAC
INSTRUCTION (NOT OPERAND) CAUSED ACCESS VIOLATION
SYSTEM ERROR ACCESS VIOLATION

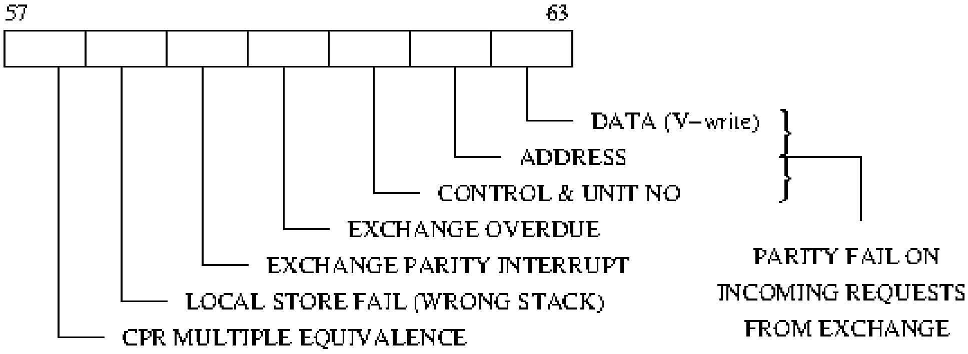
Bit 61 is only valid when bit 62 is set and refers to program faults only

Bits 60, 62 and 63 are reset by:- General reset

Writing to this line

23 SYSTEM ERROR INTERRUPTS 7 R/W = reset

(%17) (see below)


57
63
DATA (V–write)
ADDRESS
CONTROL & UNIT NO
EXCHANGE OVERDUE
EXCHANGE PARITY INTERRUPT
LOCAL STORE FAIL (WRONG STACK)
CPR MULTIPLE EQUIVALENCE
PARITY FAIL ON INCOMING REQUESTS FROM EXCHANGE

These bits are all set independently of on another, as their respective faults are detected. Writing to this line resets all bits except bit 58 and also resets line 20 and block 7 line 1. Bit 58 detects a hardware fault that cannot be cleared by software and can only be reset by general reset. General reset resets all bits. Writing to block 7 line 1 clears the bottom 4 bits

24 UNIT STATUS R

(%18) 1 - 1905E OPERATIONAL (Bit 63)

2 - EXCHANGE OPERATIONAL (Bit 62)

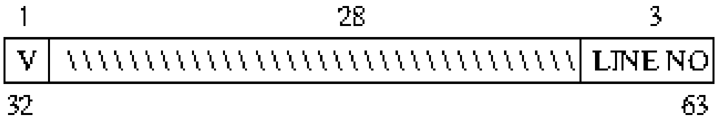
25 1905E INTERRUPT W

(%19) Writing to this line causes an interrupt signal to be sent to the 1905E.

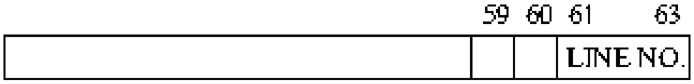
8.7 The IBU V-store (Block 5)

The instruction buffer unit maintains a record of the eight most recent control transfers in the form of a ‘jump from’ address with a ‘jump to’ address. This table is known as the JUMP TRACE and software communicates with it by means of the IBU V-lines. The JUMP TRACE is not maintained in any interrupt mode. Since V-store access can only be obtained in these modes, this ensures that the IBU V-store is used sensibly when the JUMP TRACE is static.

Address	Name	Size	Access
64-bit boundaries			
0	FILL-POINTER	4	R/W

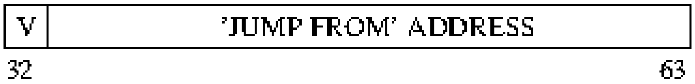


The line points to the entry in the JUMP TRACE that is the next to be filled. When the hardware fills an entry, the FILL-POINTER is incremented by 1 (modulo 8). Reading this line gives the format shown above. The Valid bit (bit 32) indicates whether that entry has been filled by the hardware since the last process change (see Section 8.3, PROP V-store). When writing to this line, the format is as follows:-



Bit 59 causes the FILL-POINTER to be written to when bit 60 (Trace On) is zero i.e. the Trace is switched off. To switch the Trace on, the line must be written to again with bit 60 set to a one or a general reset given.

1 JUMP FROM 32 R



This line contains the 'JUMP FROM' address (the address of the last 16-bit section of the JUMP instruction) in the entry in the JUMP TRACE at the FILL-POINTER. It can only be read if bit 60 in line 0 has been set to zero.

N.B. The 'JUMP TO' addresses in the JUMP TRACE cannot be read as V-store.

8.8 Peripheral Window V-store (Block 6)

Address	Name	Size	Access
0	MESSAGE WINDOW	32	R/W=Reset

This register belongs to the MU5 V-store but in addition may have information written to it from other units in the system. The writing of this information causes an interrupt in MU5. The information may be read but any attempt to write to the line will merely cause the line to be set not busy.

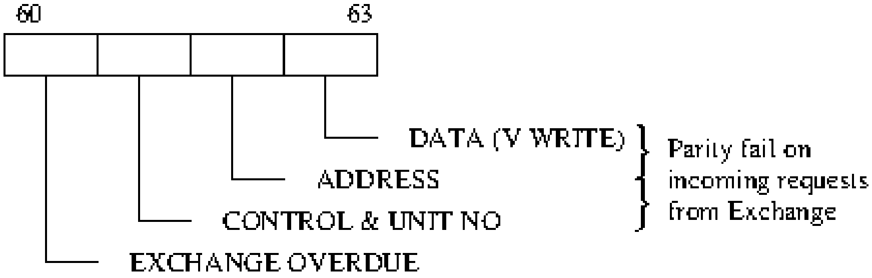
8.9 Parity V-store (Block 7)

Address	Name	Size	Access
(Decimal) 64-bit boundaries			
0	MU5 RIPF	1	R/W

This provides a means of inhibiting further requests from MU5 to Exchange (see also Section 9.2).

General reset resets this to zero.

1	EXCHANGE REQUEST PARITY	4	R/W=Reset



Writing to this line resets all bits and resets the l.s. 4 bits of Block 4 line 20.

This line is reset by:- General reset
Writing to this line
Writing to Block 4 line 23.

This line is duplicated in Block 4 line 20.

Chapter 9 The Vx-Store

9.1 Introduction

The Vx-store consists of registers that control and/or diagnose system hardware and which are communicated with from MU5 and other units in the system. It defines the means of communication between MU5 and the other units of the system. MU5 may only access Vx-store when in executive mode or any interrupt mode. Access is achieved by a real address in a real address descriptor or by CPR bypass. Below is a list of the system Vx-stores:-

9.2 MU5 Vx
9.3 Disc (Drum) Vx
9.4 Block Transfer unit Vx
9.5 1905E Vx
9.6 Local Store Vx
9.7 Mass Store Vx
9.8 System Performance Monitor Vx

N.B. A problem exists when writing to the Vx-store and then changing the status of the machine. As far as MU5 is concerned, a write order is complete when it is accepted by the store access control and it is possible for a large number of instructions to be obeyed before the order is actually executed. A software interlock must be applied in cases where this could cause trouble, e.g. writing to reset a BTU interrupt and then releasing the interrupt flip-flops in Machine Status may result in a second BTU interrupt, that will apparently vanish and may look like a message interrupt in the worst case. A similar problem could arise with parity interrupts and local store fail soft.

There are many ways of preventing such an interlock and two such ways are illustrated:-

a)	=> Vx-store	b)	=> Vx-store	
	Bn = same Vx-store		B ⊘ same Vx store	
	(destroys Bn)		B ⊘ 0	(innocuous)

A and X orders do not provide a satisfactory interlock.

9.2 The MU5 Vx-store

The normal MU5 Vx-store consists of the following:-

The Peripheral Window (Block 6, Line 0)

This line falls into the Vx category because it is the means by which other units in the system communicate with MU5. These have write only access and use this to put information into the 32-bit line. This causes an interrupt in MU5 which allows the information to be read.

The Parity V-store (Block 7)

MU5 RIPF is bit 63 of line 0 of MU5 V-store block 7. It has read/write access from all units in the system including MU5. When any bit in the Exchange Vx line UPF becomes set (see Section 9.4), Exchange sends a signal to every unit in the system. This is ignored if the unit has the manual ‘ignore parity’ switch set. If not, and the RIPF bit is set to ‘1’, then no further accesses are permitted from MU5 to Exchange until appropriate action is taken.

Remote V-store access

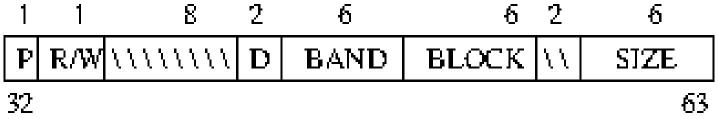
In normal circumstances this is a complete description of MU5 Vx-store. However, if the REMOTE switch is set in MU5 Console V-store (line 12 bit 54, see Section 8.5), then blocks 2-5 of MU5 V-store become available is Vx-store in addition to the two lines just described. The MU5 V-store is completely described in Chapter 8. All V lines in block 2, 3 and 5 become treated as Vx lines and with the same access as before. However, block 3, the Console V-store (Section 8.5) does have some changes on access. Lines 10, 11 and 12 as V-store have only READ access. The permissible access as Vx lines is READ/WRITE with the exception of bits 54 and 55 of line 12 which stay READ only.

Access to Peripheral Window and MU5 RIPF is unaltered on REMOTE.

9.3 The Disc (Drum) Vx-store

The Disc Vx lines exist in one block (Block 0) which consists of 32 lines of 64 bits.

Address (64-bit word)	Name	Size	Access
0	DISC ADDRESS	22	R/W



1	1	8	2	6	6	2	6
P	R/W	\\\\\\\\\	D	BAND	BLOCK	\\	SIZE
32							63

P is the internal read request bit and if set (i.e. = ‘1’) overrides bit 33. (Therefore normally a zero.)

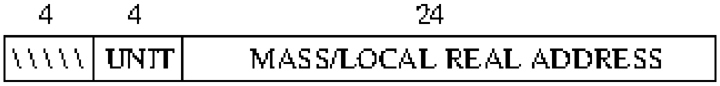
R/W specifies whether reading from or writing to the disc (‘1’ = READ.)

D specifies the disc number 0-3. Each has 64 bands containing 37 blocks of 256 words (32 bits + 4 parity bits).

SIZE specifies the number of block requested for transfer. Writing to this line initiates a disc transfer.

Note The block and size digits are updated during a transfer.

1	STORE ADDRESS	28	R/W



4	4	24
\\\\\	UNIT	MASS/LOCAL REAL ADDRESS

The 28 bits specify the real address in what is the receiving or sending unit. Hardware ignores the l.s. eight bits of the address; it is assumed to point to at least a 256 word block boundary (minimum transfer size). This line is altered during a transfer.

2	DISC STATUS	32	(see below)

The description below is of the status bits when set = ‘1’

Digit

32 - Decode the rest of the status line. This is examined by software on completion of each transfer. If set, some further action is required. This bit is reset by writing a ‘1’ to it.

33 - Decode Vx line 7.

This indicates an operator’s request to go onto SELF TEST. Further information about the request is held in line 7.

34-41 - Eight bits reserved for discs 2 & 3 having the same significance as bits 42-49.

42 - Disc 1 absent. Set manually to indicate disc is off-line, i.e. cannot be read from or written to by CPU.

43-44 - Spare

45 - Disc 1 on Self Test.

46 - Disc 0 absent.

47-48 - Spare

49 - Disc 0 on Self Test.

50 - Illegal request to the disc.

51-52 - When input parity error occurs these bits define whether it occurred in data, address or control information.

53 - (PF0). Input parity error. This causes a bit to be set in the Exchange Vx line UPF.

Resetting both bits is achieved by writing a ‘1’ to this bit.

54 - Bound locked out (see line 5).

55 - Data late (hardware error).

56 - Column parity error (internal to disc).

57 - Row parity error (internal to disc).

58 - Ignore parity fault - applies to ‘input parity error’ (bit 53) only

59 - End transfer. ‘1’ = ENDED.

60-63 - Disc unit number (= 0).

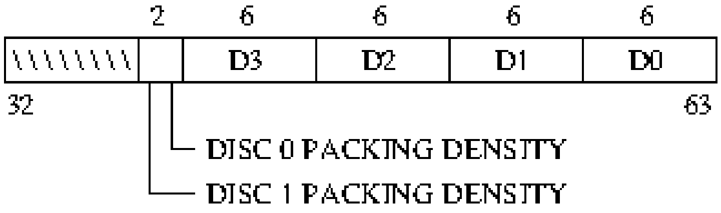
Access

All bits of STATUS can be read. Only bits 32, 50, 53, 58 and 59 can be written to. Each is reset by writing a ‘1’ to its bit position.

WARNING

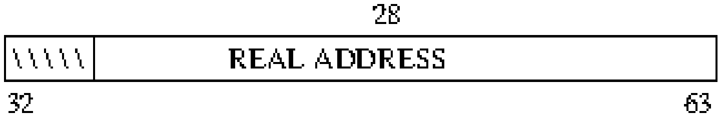
The fault bits in this line are only valid when written to the address in line 4. Some of them are reset by the disc straight after the transfer completes.

3 CURRENT POSITIONS 24 R



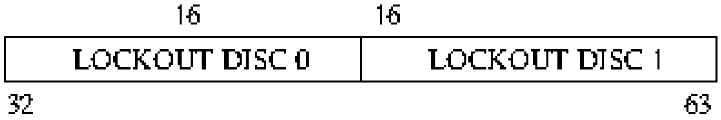
This line gives the current positions of each of the discs and also records which packing density is current (‘0‘ = HALF P.D., ‘1’ = FULL P.D.).

4 COMPLETE ADDRESS 28 R/W



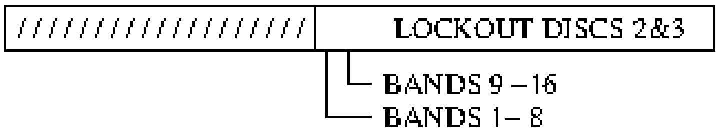
This line holds the address to be written to on disc transfer complete. The information written is the STATUS line 2. This address must not be a disc address.

5 LOCKOUT 01 32 R

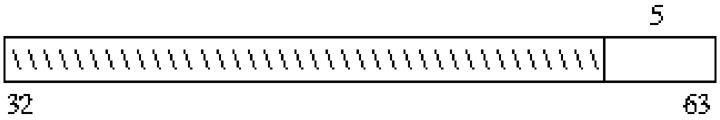


This contains 16 lockout switches for each of discs 0 and 1. These are set manually. Each bit locks out 4 bands.

6 LOCKOUT 23 32 R



7 REQUEST SELF TEST 5 (see below)



Digit

59-60 - Reserved for Request Self Test on discs 2 and 3.

61 - ‘Request self test’ on disc 1. (‘Request self test’ is set manually.)

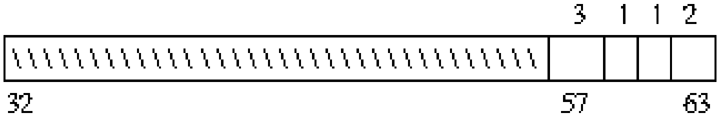
62 - ‘Request self test’ on disc 0.

63 - ‘CPU permission to self test’.

Access

All bits can be READ. Only bit 63 can be written to.

8 SELF TEST COMMAND 7 R/W



Digit

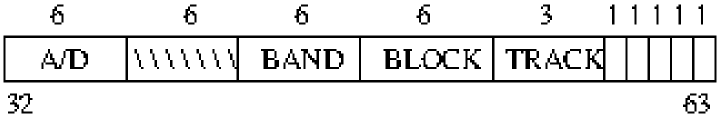
57-59 - Margins

60 - A/D required

61 - Self Test

62-63 - Disc number

9 SELF TEST STATE 26 R



A/D holds the A/D conversion value. The current BAND, BLOCK and TRACK are maintained.

Digit

59 - Max/Min Signal.

60 - Print A/D

61 - Surface Error

62 - Address Error

63 - Self Phasing Error

9.4 The BTU Vx-store

The Block Transfer Unit is designed to perform autonomous block transfers between six possible stores in the system (2 mass stores and 4 local stores). Up to 4 block transfers may be specified concurrently and these are carried out on an equal priority, time-shared basis.

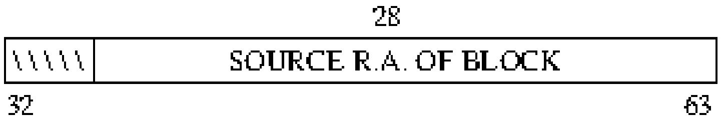
A block transfer from mass to local, for instance, is carried out one word at time through the Exchange. The word to be transferred is buffered in the BTU before being sent on to the local store. Thus the transfer mass → local actually consists of two transfers:-

Mass → BTU followed by
BTU → Local

A block transfer has 4 controlling V-lines associated with it. There are 4 sets of these V lines allowing 4 concurrent transfers. Each is situated in one of the 4 block addresses 0-3 of this unit's address field. A block of BTU Vx-store consists of 32 lines of 64 bits. Within blocks 0-3 the Vx lines have the following significance:-

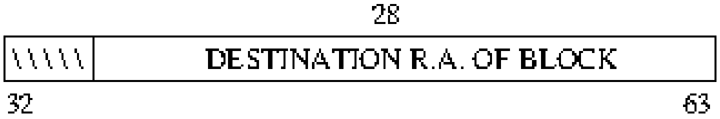
BLOCKS 0-3 (Transfer control Vx lines)

Address (Decimal) 64-bit boundaries	Name	Size	Access
0	SOURCE ADDR	32	R/W



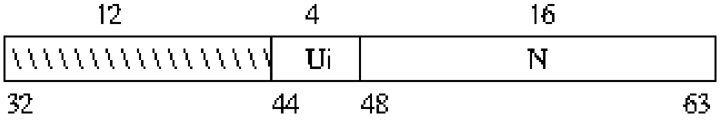
The hardware interprets this address as referring to a boundary that is a multiple of the block size obtained by rounding the transfer size up to the nearest power of 2. In addition, the least significant address bits are always interpreted by the hardware as zero (16 word minimum boundary). A transfer of all zeros (null transfer) is achieved when bit 41 of the source R.A. is set to 1 and the unit no (bits 36-39) are 9 (i.e. local).

1 DESTINATION ADDRESS 28 R/W



This address is interpreted by hardware in the same way as the source address.

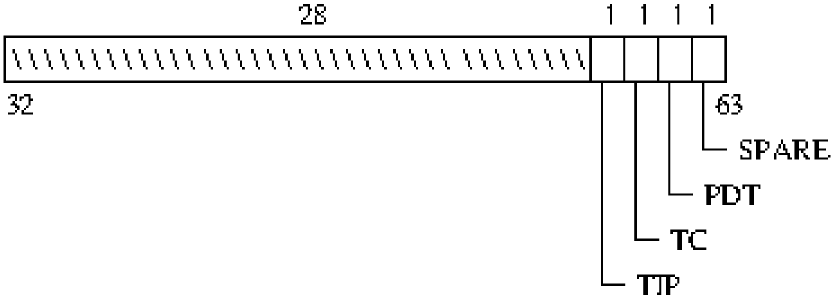
2 SIZE 20 R/W



At the start of a block transfer N specifies the transfer size as 2 less than the number of 32-bit words due for transfer. (N/2 will always be odd.) The maximum transfer size is 64K and the minimum theoretical size is 2 words.

The transfer is carried out from the final word of the block backwards to the first. Each time a 64-bit word is transferred, N is decremented by 2. On completion of the transfer N will be -2. Ui is the number of the unit which is to be interrupted on completion of the transfer.

3 TRANSFER STATUS 4 R/W



Bit 60 is the Transfer in Progress bit. Setting this bit initiates a block transfer. It may be reset by software to terminate the transfer midway. Hardware resets this bit on transfer complete (successful or not).

Bit 61 is the Transfer Complete bit. Hardware sets this bit on completion (successful or not) of the block transfer. This is what causes the Block Transfer Complete interrupt in Ui. The interrupt may be turned off by resetting this bit. Bit 62 is the Parity during Transfer bit. When set, this bit indicates that the current transfer has been terminated by hardware because of a parity fault (see BTU Block 4 - Parity Vx lines. Bit 63 is a spare fault bit.

BLOCK 4 (Parity Management)

0 PFO 1 R=Reset

Bit 63 of this line is set if a parity error is detected on address or control bits sent to the BTU from the Exchange. This signal is passed back to the Exchange and sets a bit in the UPF line in the Exchange Vx-store (see BTU Vx-store Block 5 line 2). Both these bits are reset by reading the PFI bit only,

1 BTU RIPF 1 R/W

Bit 63 of this line is a means of inhibiting further requests from the BTU to Exchange. When any bit in the UPF Vx line (see block 5 line 2) becomes set as a result of some parity fault, Exchange sends a signal to each unit in the system. When this signal appears in the BTU, provided parity interrupts are uninhibited, the RIPF line, if set, will stop further Exchange requests.

2 TRANSFER COMPLETE 4 R

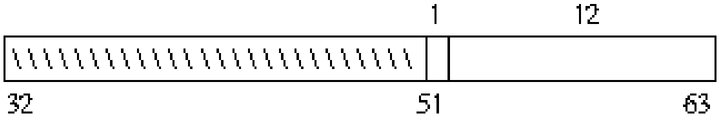
Bits 56-59 contain the transfer complete bits for channels 0-3 respectively. N.B. Early morning reset sets these bits to zero.

BLOCK 5 (Exchange Vx Lines)

The Exchange does not have the status of a unit and its Vx lines are addressed via the BTU. The Exchange Vx lines constitute block 5 of the BTU Vx-store.

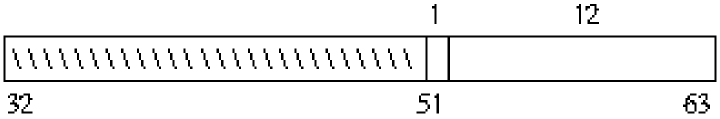
0 SUPF 13 R/W=Reset

This line indicates sending unit parity fail. For each transfer through Exchange, the data, address and control information is checked for correct parity. If the check fails, a bit is set in SUPF corresponding to the unit that sent the information.



Bits 52-63 correspond to wrong parity from units 0-11 respectively. Thus Exchange does not stop the transfer if a parity failure is detected, but merely notes which unit was responsible for it. Reading this line causes it to be cleared. Bit 51 is known as CAP. If set, it indicates a control or address parity.

1 UPF 13 R



Bit 51 of this line is set if any of the bits of SUPF (line 0) are set. It represents any parity fail on information entering Exchange. Bits 52-63 represent any parity signal sent to Exchange from units 0-11 respectively. All units parity check information coming from Exchange. Any failure causes a bit to be set in the unit and a signal returned to Exchange which sets the appropriate bit in UPF.

Bits 52-63 of UPF are reset by resetting the parity fail bit in the appropriate unit. Bit 51 is reset by reading SUPF.

9.5 The 1905E Vx-store

This Vx-store consists of 4 lines in block 0.

Address (Decimal) 64-bit boundaries	Name	Size	Access
0	VXINT	1	W

Writing to this line interrupts the 1905E at the next Normal Mode instruction fetch time (the value of bit 63 is irrelevant). Writing to this line sets bit 2^{20} in the 1905E's internal V-line (known as SR129).

1	5ERIPF	1	W

Writing a '0' to bit 63 of this line resets the 5ERIPF flip-flop (thus allowing requests through the Exchange in the event of a parity fail). Writing a '1' to bit 63 sets 5ERIPF (thus inhibiting requests in the event of a parity fail). 5ERIPF appears as bit 2^{21} in the 1905E's internal V-line 129.

2	RPS	1	W

Writing to this V-line resets the 1905E's PFO flip-flop (the value of bit 63 during writing is irrelevant). PFO, which indicates a parity fail detected by the 1905E on information received via Exchange, also appears as bit 2^{18} in the 1905E's internal V-line 129.

3	Spare	1	W

Writing to this line causes no action.

Notes

a) Writing to higher Vx addresses causes the address to be decoded modulo 4.

b) Incoming Read and Read-and-Mark requests are ignored by the 1905E, except that 'Buffer Free' signals are returned to Exchange. Note that the 'Store Free' flip-flop does not exist in the Exchange for the 1905E.

a) In order to aid system development, there exist further signals between the 1905E and MU5. The following is a list of the relevant bits in the 1905E's internal V-line 129.

digit in V-line 129	Meaning
2^1	Advance Warning Power Failure (similar to bit 49 in MU5 System Error V-line).
2^2	MU5's Remote switch on/off.
2^3	Exchange operable/inoperable (similar to MU5 V-line 24 in block 4).
2^{15}	Allow/Inhibit MU5 communication (a manual switch on the 1905E).
2^{16}	Unit fail (similar to the Exchange Overdue signal at bit 60, MU5 V-line 23 in block 4).
2^{17}	DINTO (Diagnostic Interrupt Outwards); this is set by writing to MU5 V-line 25 in block 4.
2^{22}	BTU End-of-Transfer interrupt.

The 1905E also sends a signal that appears in bit 63 of MU5 V-line 24 block 4, to indicate that the 1905E is operational. Finally, the 1905E software may produce a signal DINTI (Diagnostic Interrupt Inwards) that interrupts MU5 at Level 0 and appears as bit 54 in the System Error V-line (as yet not implemented).

Further description of the 1905E/MU5 interface may be found in the relevant 1905E documentation[11]).

[11] It's unlikely that this documentation still exists.

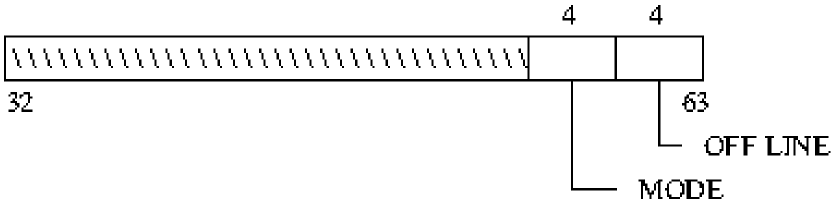
9.6 The Local Store Vx-store

This Vx-store consists of 5 lines in block 0.

Address (Decimal) 64-bit boundaries	Name	Size	Access
0	PFO	4	R/W=Reset

Bit 63 is set when a parity failure occurs in incoming addresses or control bits from Exchange. This results in a bit being set in Exchange V line UPF. Both bits are reset by writing to PFO.

8	FAILSOFT	8	R/W



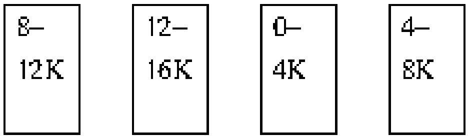
Digit	
56-59	These bits can be set to specify a failsoft mode (see below).
60-63	These bits are set manually to indicate that a stack (0-3 respectively) is OFF LINE. Writing to these bits has no effect.

Bits 56-59 of Line 8

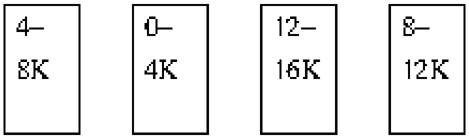
These may be set to 11 meaningful numbers. Each number corresponds to a mode of operation of the Local Store, as below:-

NUMBER	MODE	0	1	2	3
0	Normal – all 4 stacks interleaved in order	0 1	2 3	4 5	6 7
4	Non–interleaved – each stack contains 4K of sequential addresses, in order	0– 4K	4– 8K	8– 12K	12– 16K

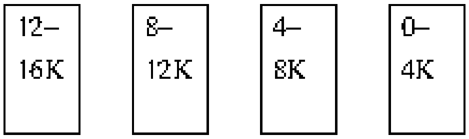
5 Non–interleaved – each stack contains 4K of sequential addresses, in order



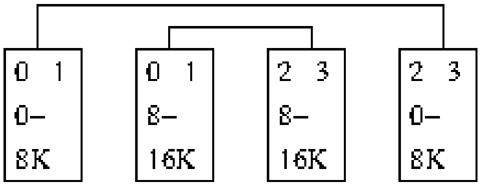
6 Non–interleaved – each stack contains 4K of sequential addresses, in order



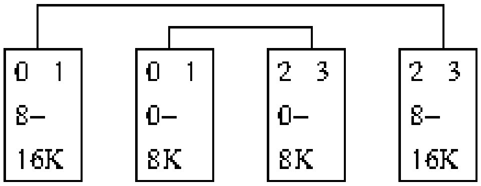
7 Non–interleaved – each stack contains 4K of sequential addresses, in order



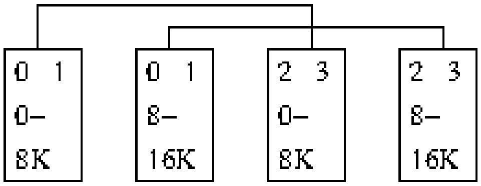
8 Interleaved in pairs, stacks 0 & 3 and 1 & 2, addressed in order



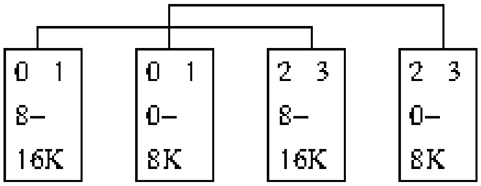
9 Interleaved in pairs, stacks 1 & 2 and 0 & 3, addressed in order



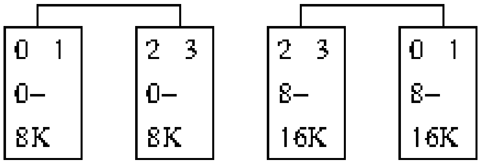
10 Interleaved in pairs, stacks 0 & 2 and 1 & 3, addressed in order



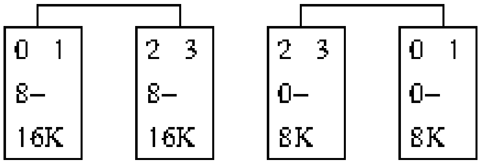
11 Interleaved in pairs, stacks 1 & 3 and 0 & 2, addresssed in order



12 Interleaved in pairs, stacks 0 & 1 and £ & 2, addressed in order



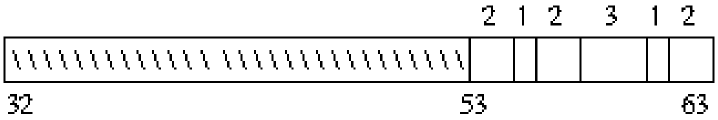
13 Interleaved in pairs, stacks 3 & 2 and 0 & 1, addressed in order



Modes 1, 2 and 3 also give normal interleaving.

Suitable precautions must be taken to ensure that the setting of this line is not changed unless the Local Store is completely inactive.

16 SELF TEST CONTROL 11 R/W
(%10)



Digit

53-54 Margins - these bit specify three states as follows;-

00 - Normal

01 - Normal

10 - Inverse of console switches

11 - Obey console switches

55 End action - ‘0’ - Continuous self test

‘1’ - One cycle of the stack

56-57 Fixed Address Bit

These bits specify the range of the 64-bit addresses to be tested on this stack.

00 - All 64-bit words

01 - Only the odd 64-bit words

10 - Only the even 64-bit words

58-60 Pattern for testing

61 Function

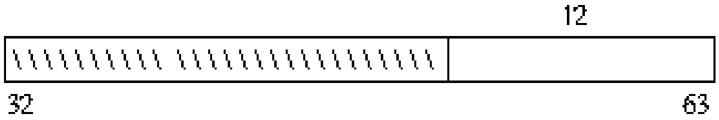
‘0’ - Clear Write

‘1’ - Read Restore

62-63 Stack on test

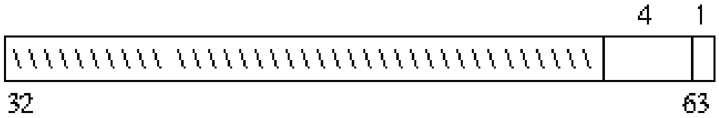
24 READ ADDRESS 12 R

(%1B)



This allows the current self test address to be read (e.g. after stop on error). The address is in the form of 12 bits referencing a 64-bit word in the stack on test. The stack in question must already be on self-test.

32 PST 5 See below



(%20)

Digit

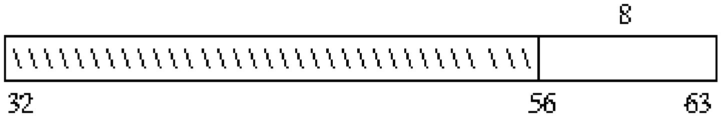
58-62 READ ONLY Fault bits - one per stack

63 R/W - this bit initiates and terminates self-test when set = 0.

9.7 The Mass Store Vx-store

Address (Decimal) 64-bit boundaries	Name	Size	Access
0	POWER STACK 0	8	R/W
1	POWER STACK 1	8	R/W
2	POWER STACK 2	8	R/W
3	POWER STACK 3	8	R/W

These first 4 Vx lines define the operation of power supplies in the stacks 0-3 with the following format:-

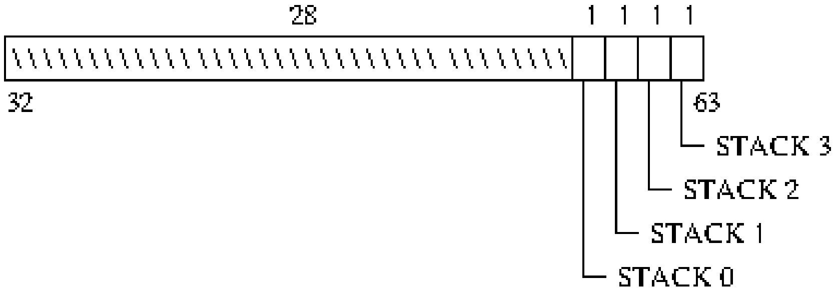


Digit	
56-57	Power Supply 0
58-59	Power Supply 1
60-61	Power Supply 2
62-63	Power Supply 3

Each stack has 4 power supplies and the digits above define whether these have to work at nominal values or at increased or reduced margins.

Value	
00	Nominal
01	Reduced margin
11	Increased margin

4 OFF LINE STACKS 4 R



These bits when set to ‘1’ define a stack to be OFF LINE. This is achieved manually.

5 WORKING STACKS 4 R/W

This line has the same format as line 4. Each bit when set specifies a stack to be working normally. Reset to put a stack on test.

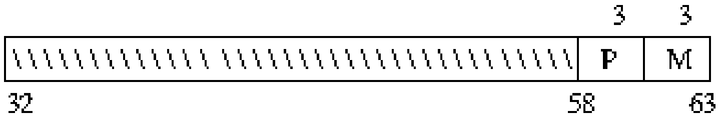
6 STACKS ON TEST 2 R/W

Bits 62 and 63 of this line define which of stacks 0-3 is on self test.

7 ON TEST INDICATOR 1 R/W

Setting bit 63 initiates self test on the stack specified in line 6. To reset testing after a stoppage, e.g. stop on error, OTI must first be reset to zero, then set back to ‘1’.

8 SELF TEST CONTROL 6 R/W



Before putting any stack on self test, patterns (58-60) and operating modes (61-63) can be written to this line.

9 END ACTION 3 R/W

Bit 61 is SEQ and specifies only one cycle of all addresses of the stack on test ('1') or continuous cycling ('0').

Bit 62 is ST and for the stack on test specifies 'stop at end of current store cycle' ('1').

Bit 63 is CON and specifies (for each stack on test) 'stop on error' ('0'). To continue after stop on error this bit can be reset to '1' then back to '0' again.

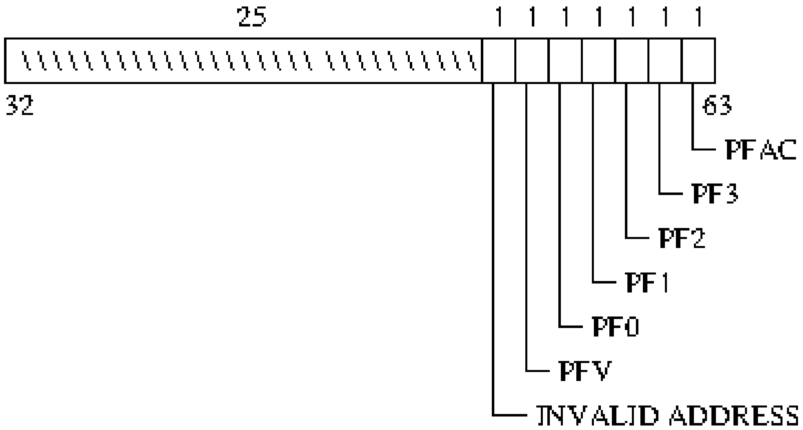
19 TRANSFER ADDRESS 17 R
(%A)

This line holds the address at which an error occurred when on self test.

11 TRANSFER DATA OUT 36 R
(%B)

This line may be read to obtain 9 bits, the data involved in an error on self test. Data is regarded as 36 bits (32 data bits + 4 parity bits). This is conceptually divided into 4 sets of 9 bits. When line 11 is read, line 17 defines which set of 9 bits is to be read.

12 PARITY FAIL 7 R=Reset
(%C)



Digit	PARITIES
57 (INVAL)	INVALID ADDRESS sent to interface i.e. too high for number of stacks working
58 (PFV)	V-store I/P data
59 (PF0)	Stack 0 I/P or O/P data
60 (PF1)	Stack 1 I/P or O/P data
61 (PF2)	Stack 2 I/P or O/P data
62 (PF3)	Stack 3 I/P or O/P data
63 (PFAC)	Address or Control bits from Exchange

If any of these bits is set, a signal is sent to Exchange and to the Engineers' Door and Console.

13 (%D)	STACK 0 STATUS	3	R
13 (%E)	STACK 1 STATUS	3	R
13 (%F)	STACK 2 STATUS	3	R
13 (%10)	STACK 3 STATUS	3	R

The formats of these 4 lines are all the same.

Digit	
61	Power Supply on margin
62	Stack on self test
63	Error during self test

17 (%11)	TRANSFER PART	2	R/W

This specifies which section of the data or address word is to be read when reading TDO or TA lines 24 and 26.

'0' refers to the most significant section.

'3' refers to the least significant section.

9.8 The System Performance Monitor Vx-store

The System Performance Monitor (SPM) was a bespoke system built as part of the MU5 Project to allow hardware monitoring of activity within the MU5 Processor. Cables attached to various registers and logic signals in the Processor were connected as inputs to the SPM where they could simply be counted or could be recorded in the form of a histogram showing, for example, the relative number of occasions on which a given event occurred 'n' times between occurrences of some other event. As well as a set of fast counters and a data (histogram) store, the SPM included a title store (to enable histograms to be labelled) and its own visual display unit on which the contents of its store(s) could be displayed. The SPM could be controlled externally via its V-line and MS2 in MU5. MS2 controlled recording in the SPM, *i.e.* the SPM only recorded data when MS2 was set to 1, allowing selective recording of specific processes such as benchmark programs. With MS2 = 1, setting MS3 caused instructions to be executed one at a time, rather than being pipelined. The SPM stores could be accessed by other units in the MU5 Complex to allow large amounts of data to be accumulated and printed.

System Performance Monitor Real Store

This consists of 3 blocks of store addressed as follows:-

Address (64-bit word)	Name	Size	Access
0	SPM.DATA.STORE	512 x 16 bits	R/W

To read/write from this store, bits 48-63 of the highway are used.

512	SPM.FAST.COUNTERS	16 x 32 bits	R

To read these counters, bits 48-63 of the highway are used. The more significant half of a fast counter is at address 512 + 7 + 16*N, the least significant half is at address 512 + 15 + 16*N where N is the fast counter required.

768	SPM.TITLE.STORE	256 x 7 bits	R/W

To read/write from this store, bits 48-55 of the highway are used.

System Performance Monitor V-line

This line is accessed by using a real address descriptor (Type 3.0), with V-store specified in the origin. The real address part specified is irrelevant. The top 32 bits of the V-line are write only (if read, they return 0). The bottom 32 bits are read only.

Digit		
0	Enable bits 1, 2, 3	
1	RC1	Resets and initiates histogram logic
2	RCVAL	Overrides validation lines for histogram input pulses
3	INTOFF	Reset Interrupt
4	Enable bits 5, 6 7	
5	RCNT1	Initiates fast counters
6	RCNTVAL	Overrides validation lines for fast counters
7	RCNTR	Clears fast counters
8	Enable bits 9, 11	
9	RAI0	Identifies the activity required for software monitoring
10		Not used
11	RAI1	Used as for RAI0
12	Enable bits 13, 14, 15	
13	RIMA	} Control the input mode on the
14	RIMB	} two histogram input channels
15	RDYW	Allows Dwell Histogram logic to continue upon overflow of Y counter
16	Enable bits 17, 18, 19	
17	RSF**4	Set the scale factor on the Prescaler
18	RSF**2	"
19	RSF**1	"
20	Enable bits 21-31	
21		Not used

22		Writing to this bit causes an interrupt (see bit 60)
23	RSAC	Allows CPU to write to or read the real store via Exchange
24	RSAD	Initiates the display of the store contents on the VDU
25	RSAIH	Interval Histogram mode
26	RSAIDH	Increment/Decrement Histogram mode
27	RSARCH	Clears Title store
28	RSADH	Dwell Histogram mode
29	RSADAD	Direct Addressing mode
30	RSARD	Clears Data store
27 & 30		Clears Title store and Data store
28 & 29		Snap-shot mode
NOTE:		Except for those pairs noted, only one bit in the above group should be set at a time
31	PFLR	Reset parity fail indicator
32-59		Not used
60	RINT	An interrupt has been forced by writing to bit 22
61	RM*C	Monitor is in Manual mode
62	RIGNEX	Monitor has ignored Exchange request this causes an interrupt
63	GIGNEX	Monitor is ignoring Exchange except for V-reads i.e. Monitor is in one of the following modes:- MANUAL, DISPLAY, CLEAR-STORE

Interrupts	Three conditions cause an SPM interrupt:-
STORE FULL	The SPM has accumulated all the data it can handle and must be dumped.
RIGNEX	(see bit 62)
RINT	(see bit 60)

Chapter 10 The Basic Programming Language - XPL[12]

Table of Contents

[12] The XPL compiler originally ran on both MU5 and the ICL 1905E but only the MU5 version was used once MU5 was fully commissioned.

10.1 The Metalanguage

Modified BNF (Backus Naur Form) is used to define any XPL syntactic element.

The modifications to the BNF are as follows:

(1) In the order of alternatives and of elements within alternatives:-

a. Any alternative which is a stem of another comes after it.

b. If any one alternative is a special case of another, it must come first.

c. In recursive definitions, there must be at least one left-most element not recursive.

(2) Metalinguistic Bracketing:-

Several alternatives may be specified as an element of another by enclosing them in square brackets.

10.2 Program & Statements

```
<XPL.PROGRAM> ::= <PROGRAM.OF.A.SEGMENT>[<XPL.PROGRAM>|<NIL>]
```

```
<PROGRAM.OF.A.SEGMENT> ::=    *SEGMENT<SP><SEGMENT.NO><NL>
                              BEGIN <NL>
                              <program> <NL>
                              END <NL>
                              *END OF SEGMENT <NL>
```

The <program> consists of a number of statements and hence:-

```
<program> ::= <STATEMENT>[<PROGRAM>|<NIL>]
```

The statements are:-

```
<STATEMENT> ::=    <LABEL>|
                   <LABEL><SEP>|
                   <TABLE><SEP>|
                   <TEXT><SEP>|
                   <BLOCK><SEP>|
                   <DECLARATIVE><SEP>|
                   <INSTRUCTION><SEP>|
                   <SPECIAL.DIRECTIVE.STATEMENT><SEP>|
                   <SEP>
```

where <sep>, a separator is:-

```
<SEP> ::= <NL>|<COMMENT>
```

and where <comment> commences with two colons and terminates with a newline. For line continuation purposes π <NL> is ignored.

The seven types of XPL statement are explained in the following sections.

10.3 Names, Literals & Labels

Since the basic operands in the language are names and literal, it is convenient to define them first.

(1) Names & Literals

```
<LITERAL>              ::= <DECIMAL>|
                           %<HEX.DIGITS>|
                           "<CHARACTER.STRING>"|
                           <NAME>|
                           <DR.LIT>
<DECIMAL>              ::= [+|-|<NIL>]<INTEGER>[.<INTEGER>|<NIL>]
<INTEGER>              ::= <DECIMAL.DIGIT>[<INTEGER>|<NIL>]
<HEX.DIGITS>           ::= [<HEX>|<HEX>(<INTEGER>)]
                           [<HEX.DIGITS>|<NIL>]
<HEX>                  ::= 0|1|2|3|4|5|6|7|8|9|A|B|C|D|E|F
<CHARACTER.STRING>     ::= [<CHARACTER>|<HEX.PAIR>]
                           [<CHARACTER.STRING>|<NIL>]
<HEX.PAIR>             ::= <VB><HEX><HEX><VB>
<VB>                   ::= a vertical bar
<NAME>                 ::= <LETTER>|<LETTER><NAME.SYMBOLS>
<NAME.SYMBOLS>         ::= [<LETTER>|<DECIMAL.DIGIT>|.]
                           [<NAME.SYMBOLS>|<NIL>]
<DR.LIT>               ::= D<QUA><OCT><OCT>/[<INTEGER>|<NAME>]
                           /[<INTEGER>.<INTEGER>[.<INTEGER>|<NIL>]|
                           <NAME>]
<QUA>                  ::= 0|1|2|3
<OCT>                  ::= 0|1|2|3|4|5|6|7
```

(2) Labels

A label takes the form of a name followed by a colon, i.e.

<LABEL> :: = <NAME>:

Any number of labels may precede an instruction; references to labels are optimised by XPL.

(3) Further Notes on Literals

There are five different types of literal, each of which is explained below. The XPL complier will attempt to optimise their lengths.

a. DECIMAL NUMBER (not currently implemented on MU5) in which a decimal point implies that the number be coded as a floating-point literal. All decimal numbers are signed.

Examples:-

Signed fixed point:-

2	::	6-bit
51	::	16-bit
+51	::	16-bit (the positive sign is optional)
327680	::	32-bit

Floating-point (all 64-bit):-

1.0	::	1. will be faulted
-0.23	::	-.23 will be faulted

b. BINARY LITERAL is a % (percent) sign followed by a string of hexadecimal digits. Right justification is adopted.

Examples:-	%1F	::	6-bit signed
	%2F	::	16-bit unsigned (6-bit is impossible, since
		::	the sign bit will be propagated)
	%F903	::	16-bit unsigned
	%4F903	::	32-bit unsigned
	%800000000004F903	::	64-bit
	%80(10)4F903	::	gives the same literal as in the above
		::	example

c. CHARACTER LITERAL is a string of characters enclosed by single characters that represent double quotes. Hexadecimal pairs can be used to represent characters not available on the input device. A compiled literal is packed eight bits per character (in ISO code) and right justified.

Examples:-

```
"ABCDE"            ::  Double quotes represented by " on most
                   ::  Flexowriters.
"ABCDE|0A|XYZ"     ::  The hexadecimal pair |0A| is compiled
                   ::  as a newline character.
```

d. NAME LITERAL can either be a name declared as a literal or the name of a label, in which case the absolute address is used.

Examples:-

```
MAX.VALUE          = 100
MASK               = %FOF
ENTRY.POINT        = "START"
NINE               = 9

BASE:
```

e. DR LITERAL is a descriptor literal

In XPL, the bound field can either be a previously defined name or an integer, whereas the origin field can be either a name or ‘SEGMENT.WORD.BYTE’, which specifies the address of the start of the string (in the absence of the ‘.BYTE’, the byte position is taken to be zero).

Examples:-

```
D033/19/8196.74.3
                  :: This is a vector descriptor (type 0), defining a string of 8-bit
                  :: elements. Modifier is not scaled and there is no bound check.
                  :: There are 19 elements, starting at byte position 3 of line 74 in
                  :: segment 8196.

D260/NINE/BASE
                  :: This is a descriptor (type 2), defining a string of 64-bit
                  :: elements. Modifier is scaled and there is a bound check. There
                  :: are 'NINE' elements, starting at the label 'BASE'. (The name
                  :: literal 'NINE' must be declared prior to this DR literal,
                  :: whereas the label 'BASE' can be a forward reference.
```

N.B. When a vector is accessed by using a name corresponding to a descriptor literal, two orders will be compiled, e.g.

```
A = DESCRIPTOR.1[B]
```

will be compiled into:-

```
D = DESCRIPTOR.1
A = D[B]
```

10.4 Tables & Texts

Table is used to plant literals within the compiled code, whereas text is used to plant a string of symbols

(1) Tables

<TABLE> ::= DATAVEC<SP><NAME>(<LENGTH>)
<NL><LIT.LIST><NL><END

where <NAME> ::= a literal descriptor set up to access the content of the TABLE.

<LENGTH> ::= an <INTEGER> specifying the length of the literals in the TABLE. (The length can be 1, 4, 8, 16, 32 or 64 bits).

<LIT.LIST> ::= <LIT.LINE><NL><LIT.LIST>|<LIT.LINE>

<LIT.LINE> ::= <LIT.ITEMS>[<LB><INT><RB>|<NULL>]
where <integer> indicates the number of times that the preceding items on this line are to be repeated. Nested repetitions are not allowed.

Example:-

```
DATAVEC TABLE.1 (64)
          99
          %14A76F02F
          "AB"
          -1,[5]                :: -1 to be planted 5 times
          4,FRED,[2]            :: 4, FRED to be planted twice
          0
END
```

(2) Texts

<TEXT> ::= DATASTR<SP><NAME>"<CHARACTER.STRING>"

where <NAME> ::= a string descriptor set up to access the
<CHARACTER.STRING>

Example:-

```
DATASTR CAPTION.1 "**FIX LINEPRINTER**"
```

10.5 Blocks

```
<BLOCK>  ::= [BEGIN|PROC<SP><PROC.NAME>]
             [(<LABEL.LIST>)|<NIL>]<NL>
             <program>
             END
```

```
where   <PROC.NAME>       ::= the <NAME> of the procedure.
        <LABEL.LIST>      ::= <NAME>[,<LABEL.LIST>|<NIL>]
```

Basically, a block consists of two declaratives, BEGIN and END, and serves to define the scope of labels. Declarations other than labels have a global scope equivalent to that of a forward reference. Redefinitions of names within this global scope are not allowed. Blocks can be nested to any depth.

If PROC is used instead of BEGIN, a jump instruction is planted by XPL to jump round the procedure.

If the BEGIN is followed by some names of labels, e.g. BEGIN (L31, L32), then entries to that block can be made to these labels from the enclosing block.

Example:-

```
BEGIN
          L1:  – – –
               -> L31
               – – –
               -> L32
               – – –
               BEGIN (L31, L32)      :: implying L31 and L32 are in the
                                     :: same level as L1 an L2.
                          – – –
                    L31:  – – –
                          – – –
                    L32:  – – –
                          – – –
               END
          L2: – – –
END
```

10.6 Declaratives

There are two types of declarative, both of which give the name a global scope, namely:-

(a) variable declaration <VAR.DEC>
(b) literal declaration <LIT.DEC>

<DECLARATIVE> ::= <VAR.DEC>|<LIT.DEC>

(1) Variable Declarations

The variable declarations assign a name to a displacement relative to a base, which can be NB, XNB, SF, 0 or STK (for accessing the stack).

```
<VAR.DEC>          ::= V[32|64|V]/[NB|XNB|SF|0|STK]<VAR.SPEC>
<VAR.SPEC>         ::= <NAME>:<DISPLACEMENT>[,<VAR.SPEC>|<NIL>]
<DISPLACEMENT>     ::= [-|<NIL>]<INTEGER>|%<HEX.DIGITS>
```

A name declaration must start with a V, followed by the size of the variable. The size is either 32 or 64-bit, or a V indicating that the variable will be used in privileged mode to access a V-store location.

Example

```
V32/NB FRED:3            :: FRED can be found three 32-bit words away
                         :: from NB and is of size 32 bits.

V32/STK TOP.32.bits:0 :: The displacement must be zero with STK.
```

Notes

1. The hardware of the machine treats all V-store as 64-bit quantities. This means that VV quantities are equivalent to V64 quantities.

2. The positions of variables in MU5 are governed by a base address contained in one of the registers NB, XNB, SF and a displacement. It should be noted that the value of the Base address and the value of the displacement cannot be freely interchanged as may have been expected.

   Thus:-

   ```
           V64/NB FRED.1:6        :: sets up a displacement of 6
           NB = 0                 :: and a Base of 0
   ```

   and:-

   ```
           V64/NB FRED.2:0        :: sets up a displacement of 0
           NB = 6                 :: and a Base of 6
   ```

   will not access the same location in the store. This is because the base registers always count in units of 32 bits whereas the displacement counts in units of the size of the variable, in this case 64 bits. The example therefore says that zero units of 32 bits plus 6 units of 64 bits is not the same as zero units of 64 bits and 6 units of 32 bits.

(2) Literal Declarations

A literal declaration assigns a value to a name. When the name appears as an operand, its value will be coded as a literal in the instruction. (As already shown in (3) of Section 10.3, XPL will optimise the lengths of the literals).

```
<LIT.DECL>   ::= L/<LIT.SPEC>
<LIT.SPEC>   ::= <<NAME>=<LITERAL>[,<LIT.SPEC>|<NIL>]
```

Examples:–

```
L/        MAX.VALUE = 100
L/        MASK = %44420F
L/        NAME.STRING = "ACCUMULATOR"
```

N.B. a. A number of declarations can be placed on a line, e.g.

```
V32/SF    VAR.0:0, VAR.1:1, VAR.2:2, VAR.3:3
L/        MAX.VALUE = 100, MIN.VALUE = -100
```

b. Newline or a comment would terminate the sequence, hence declarations requiring more than a single line would have to be written as:

```
V32/SF VAR.0:0, VAR.1:1, VAR.2:2 VAR.3:3
V32/SF VAR.4:4, VAR.5:5
```

10.7 Instructions

There are four types of instruction, namely:

(1) computational <COMPUT>,

(2) store to store <STS>,

(3) organisational <ORG> and

(4) conditional <CONDIT>.

Each type of instruction is dealt with separately. However, it is convenient to define the syntax of an operand first.

```
<OPERAND> :: = <SIMPLE.OPERAND>|<NAME><LB>[B|O]<RB>
where      <SIMPLE.OPERAND> ::= <NAME>|<LITERAL>
           <LB> ::= left square bracket
           <RB> ::= right square bracket
```

(1) Computational Instructions

```
<COMPUT>    ::= [<B.ORD>|<X.ORD>|<A.ORD>|<AOD.ORD>|<AEX.ORD>]
                <OPERAND>
<B.ORD>     ::= B[=|='|*=|=>|+|-|*|≢|V|<=|&|-:|COMP|CINC]
<X.ORD>     ::= [XS|X][=|*=|=>|+|-|*|/|≢|V|<=|&|-:|COMP|CONV|/:]
<A.ORD>     ::= [AFL|A][=|='|*=|=>|+|-|*|/|≢|V|<=|&|-:|COMP|CONV|/:] |
                [AU|AX][+|-|*|/|≢|V|<=|&|-:|COMP] |
                [ADC|AD][<=|COMP|CONV]
<AOD.ORD>   ::= AOD[=|*=|=>|COMP]
<AEX.ORD>   ::= AEX[=|*=|=>]
```

N.B. Where the operator -: is reverse subtract
and /: is reverse divide

(2) Store to Store Instructions

This group of orders uses the secondary operand unit; they operate on strings.

```
<STS>     ::= <FN.1><OPERAND> |
              <FN.2><SIMPLE.OPERAND> |
              SUB1 <NAME> |
              SUB2


<FN.1>    ::= D=|D*=|DO=|XD=|XDO=|STACK
<FN.2>    ::= D=>|XD=>|DB=|XDB=|
              MOD|RMOD|SMOD|XMOD|MDR|XCHK|
              BMVE|BMVB|BCMP|BLGC|BSCN|
              SMVE|SMVB|SCMP|SLGC|SMVF|TALU|TCHK|TRNS
```

(3) Organisational Instructions

This group of orders defines internal register operations.

```
<ORG>          ::=  RETURN |
                    [EXIT|JUMP|XJUMP|STKLINK|]<OPERAND> |
                    SETLINK<SIMPLE.OPERAND> |
                    <MS.ORD> |
                    <XC.ORD> |
                    <SF.ORD> |
                    <NB.ORD> |
                    <XNB.ORD> |
                    <MISC.ORD> |
                    <DUMMY.ORD> |
<MS.ORD>       ::=  MS = <OPERAND>
<XC.ORD>       ::=  [XC0|XC1|XC2|XC3|XC4|XC5|XC6]<OPERAND>
<SF.ORD>       ::=  SF[=|+|=NB+]<OPERAND> |
                    SF => <SIMPLE.OPERAND>
<NB.ORD>       ::=  NB[=|+|=SF+]<OPERAND> |
                    NB => <SIMPLE.OPERAND>
<XNB.ORD>      ::=  XNB[=|+]<OPERAND> |
                    XNB => <SIMPLE.OPERAND>
<MISC.ORD>     ::=  [SN =|DL =|SPM =]<OPERAND>
<DUMMY.ORD>    ::=  D[1|2|3|4]
                       (for coding up dummy organisational orders)
```

The XJUMP order will search the Common procedure Name List for the name and plant an absolute jump to it. The STKLINK order plants a 64-bit operand.

(3) Conditional Instructions

This group of orders deals with control transfers and the setting of the BOOLEAN, BN.

```
<CONDIT>        ::=  <JUMP.SPEC><NAME> |
                     IF <COND>, <JUMP.SPEC><NAME> |
                     BN <B.FN> IF <COND> |
                     BN <B.FN><OPERAND>
<COND>          ::=  =0|≠0|<0|≤0|>0|≥0 |
                     OV|BN
<BN.FN>         ::=  / |
                     ≡|≢ |
                     = |=/ |
                     &|&/|/&|/&/ |
                     V|V/|/V|/V/
<JUMP.SPEC>     ::=  [<LONG>|<SHORT>|<NULL>] –>
<LONG>          ::= +
<SHORT>         ::= −
```

The jump instructions (–>) are relative; XPL will compile the optimum code.

As indicated by the + or – preceding the jump, either a long (32-bit) operand or a short (6-bit) operand is assumed. The default option of NULL will result in a 16-bit operand.

10.8 Procedure Call Facilities

There are 6 types of procedure call available:-

CALL	:: Plants a relative jump
ACALL	:: Plants an absolute jump
XCALL	:: Plants an absolute jump
	:: to the specified LIBRARY procedure.
	:: (plants for 16-bit operand)

```
CALL <PROC.NAME>(<LINK>, <PARAMETERS>)
```

Where <PARAMETERS> are any operands permitted after the instruction STACK.

The above expression will be compiled into:-

```
STKLINK <LINK>
STACK PARAMETER.1
STACK PARAMETER.2
      .
      .
STACK PARAMETER.N
–> <PROC.NAME>
```

ENTER	:: Plants a relative jump
AENTER (or SCALL)	:: Plants an absolute jump
XENTER	:: Plants an absolute jump
	:: to the library procedure.

```
ENTER <PROC.NAME>(<PARAMETERS>)
```

This expression will be compiled into:-

```
            STKLINK L1 (6-bit operand planted)
            STACK PARAMETER.1
            STACK PARAMETER.2
                  .
                  .
            STACK PARAMETER.N
JUMP <PROC.NAME>
L1:
```

10.9 Special Directive Statements

*SEGMENT<SP><EXECUTE.SEG.NO>[,<COMPILE.SEG.NO><NL>|<NL>]

where <EXECUTE.SEG.NO> has a value of -1 or 1 to $2^{14}-1$
and <COMPILE.SEG.NO> has a value of 1 to $2^{13}-1$

:: <EXECUTE.SEG.NO> specifies the segment
:: in which the code is to be executed.
:: <COMPILE.SEG.NO> specifies the segment
:: in which the code is to be compiled. If this is
:: unspecified, the compiler will select the next
:: available segment.
:: If <EXECUTE.SEG.NO> is equal to -1, the
:: compiler will select the execution and
:: compilation segment numbers.

*LINE<SP><LINE.NO><NL>

:: Specifies the line (in 16-bit quantities) within
:: the segment at which the next instruction is
:: to be planted.

or

*LINE<SP>.<SP>+<LINE.NO><NL> :: Adds the operand to the current line number.

*END

:: Prints a list of unmatched references on the
:: CTL currently selected output stream, sets
:: the information and returns.

*PRINT ON, *PRINT OFF

:: Control the listing of the program text and
:: compiled code.

*NL<SP><NLADDR>

:: <NLADDR> is the address in 32-bit words
:: of the library name list.

*N

:: Normally the compiler removes any relative
:: jumps to the next order. The first time the
:: directive is encountered, it turns off this
:: optimisation. The next time, it turns the
:: optimisation on, etc.

*MAP ON, *MAP OFF :: Control the printing of the compile map.

10.10 Alternative Punching Conventions for the VDUs

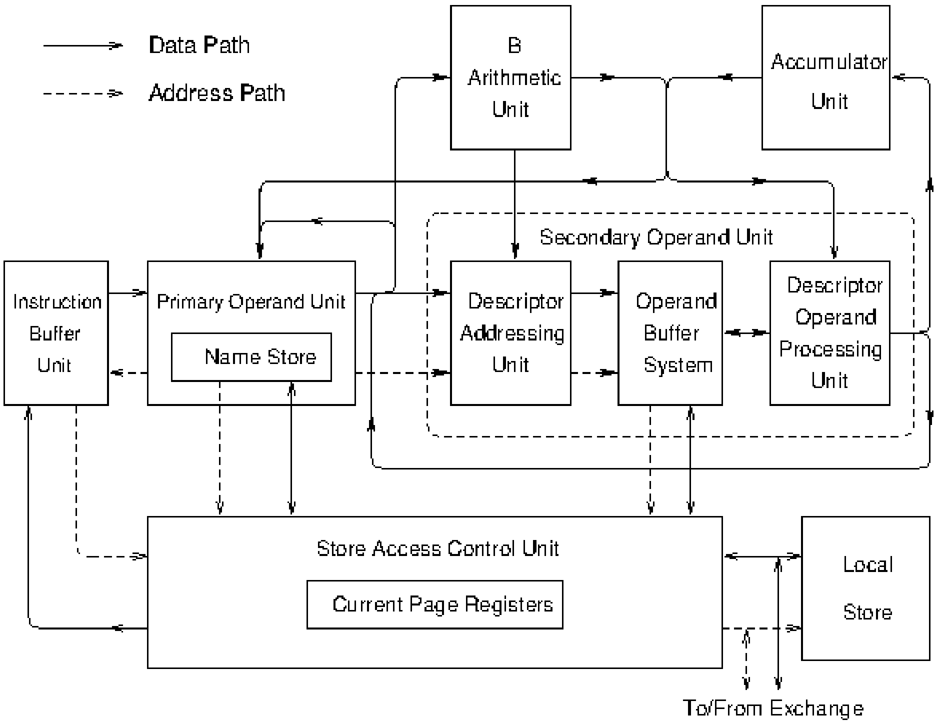
		Paper Tape	VDU
(1)	Not Equal	≠	/=
(2)	Equivalent	$\underline{=}$	−=
(3)	Not Equivalent	$\underline{\neq}$	−/=
(4)	Greater Than or Equal To	$\underline{>}$	>=
(5)	Less Than or Equal To	$\underline{<}$	=<

10.11 The Test Bits in MS

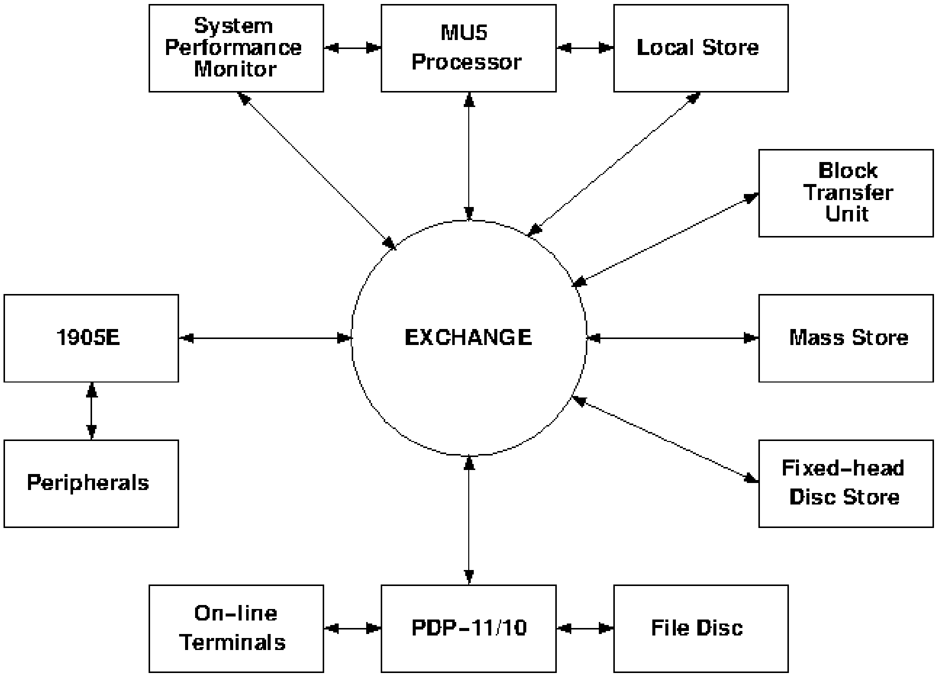
A table giving the test bit combinations in MS, corresponding to the conditional orders in XPL.

Condition	Test Bits
=	T1/
=/	T1
≥	T1/ V T2/
<	T2
≤	T1/ V T2
>	T1 & T2/
OV	T0
BN	BN

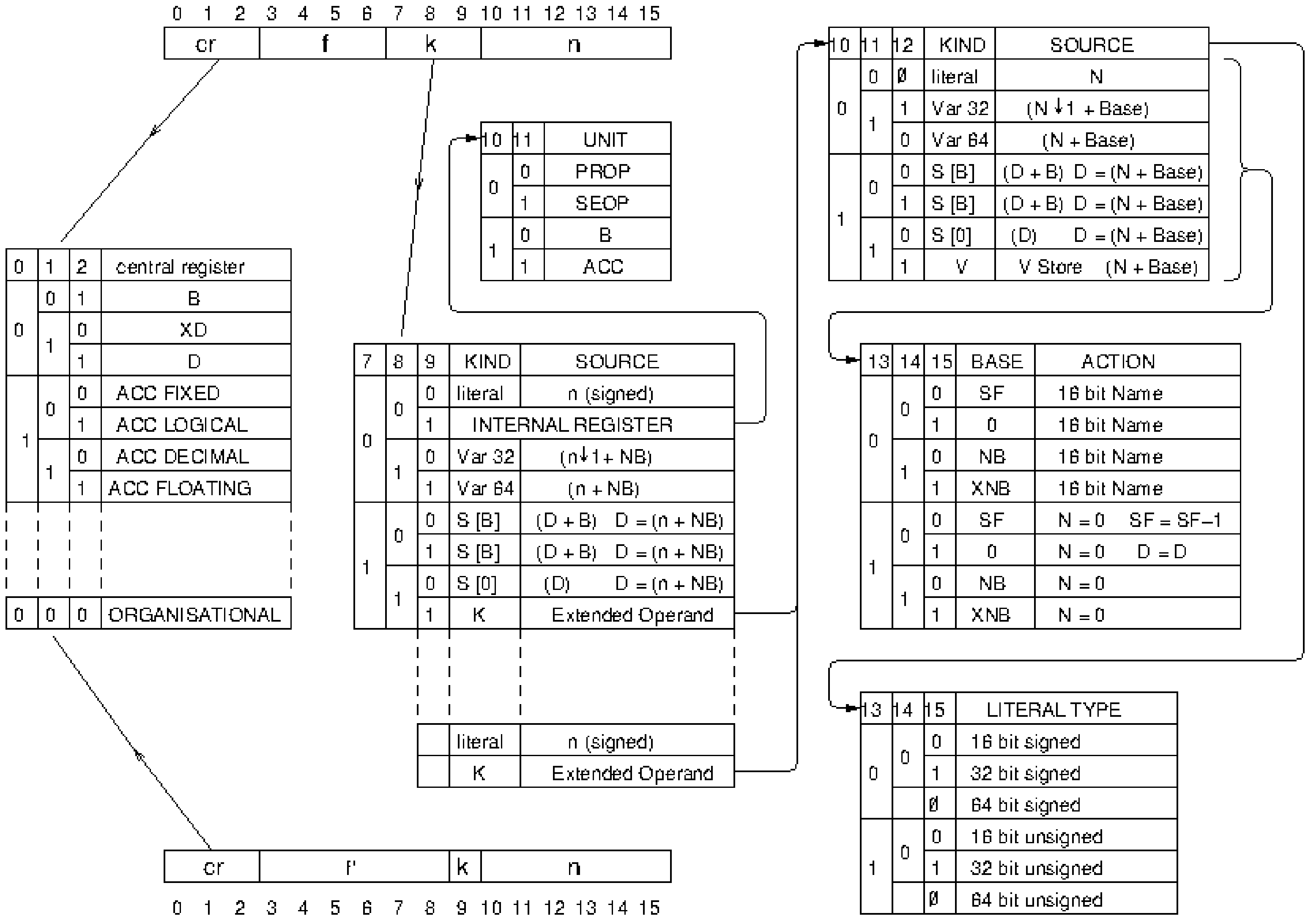
Appendix I MU5 Block Diagrams



The MU5 Processor



The MU5 Computer Complex



0 1 2	3 4 5 6	7 8 9	10 11 12 13 14 15
cr	f	k	n

0	1	2	central register
0	0	1	B
	1	0	XD
		1	D
1	0	0	ACC FIXED
		1	ACC LOGICAL
	1	0	ACC DECIMAL
		1	ACC FLOATING
0	0	0	ORGANISATIONAL

10	11	UNIT
0	0	PROP
	1	SEOP
1	0	B
	1	ACC

7	8	9	KIND	SOURCE
0	0	0	literal	n (signed)
		1	INTERNAL REGISTER	
	1	0	Var 32	(n↓1+ NB)
		1	Var 64	(n + NB)
1	0	0	S [B]	(D + B) D = (n + NB)
		1	S [B]	(D + B) D = (n + NB)
	1	0	S [0]	(D) D = (n + NB)
		1	K	Extended Operand
			literal	n (signed)
			K	Extended Operand

10	11	12	KIND	SOURCE
0	0	Ø	literal	N
	1	1	Var 32	(N ↓1 + Base)
		0	Var 64	(N + Base)
1	0	0	S [B]	(D + B) D = (N + Base)
		1	S [B]	(D + B) D = (N + Base)
	1	0	S [0]	(D) D = (N + Base)
		1	V	V Store (N + Base)

13	14	15	BASE	ACTION
0	0	0	SF	16 bit Name
		1	0	16 bit Name
	1	0	NB	16 bit Name
		1	XNB	16 bit Name
1	0	0	SF	N = 0 SF = SF–1
		1	0	N = 0 D = D
	1	0	NB	N = 0
		1	XNB	N = 0

13	14	15	LITERAL TYPE
0	0	0	16 bit signed
		1	32 bit signed
		Ø	64 bit signed
1	0	0	16 bit unsigned
		1	32 bit unsigned
		Ø	64 bit unsigned

cr	f'	k	n
0 1 2	3 4 5 6 7 8	9	10 11 12 13 14 15

MU5 INSTRUCTION SET – OPERANDS

14.1.74
RNI

Reproduced by RNI Jan 2017

				001	010	011	100	101	110	111
3	4	5	6	B	STS/D		XS	AU	ADC	AFL
0	0	0	0	=	XDO =	DO =	X =	AOD =	DUMMY	= (32)
			1	= (−1)	XD =	D =	DUMMY	DUMMY	AEX =	= (64)
		1	0	*=	STACK	D *=	X *=	AOD *=	AEX *=	*=
			1	=>	XD =>	D =>	X =>	AOD =>	AEX =>	=>
	1	0	0	+	XDB =	DB =	+	A +	DUMMY	+
			1	—	XCHK	MDR	—	A—	DUMMY	—
		1	0	*	SMOD	MOD	*	*	DUMMY	*
			1	/	XMOD	RMOD	/	DUMMY	DUMMY	/
1	0	0	0	≢	SLGC	BLGC	≢	A ≢	DUMMY	A ≢
			1	V	SMVB	BMVB	V	A V	DUMMY	A V
		1	0	↑	DUMMY	BMVE	↑ ARITH	A ↑ LOG	A ↑	A ↑ CIRC
			1	&	SMVF	SMVF	&	A &	DUMMY	A &
		0	0	⊖	TALU	DUMMY	⊖	A ⊖	AOD COMP	⊖
			1	COMP	DUMMY	BSCN	COMP	A COMP	COMP	COMP
		1	0	CINC	SCMP	BCMP	AEX=CONVX	DUMMY	UNPACK	AEX=CONVA
			1	⊘	SUB1	SUB2	⊘	DUMMY	DUMMY	⊘

f BITS

0	0	1	1	7	
0	1	0	1	8	
–>	EXIT	DUMMY	DUMMY		
JUMP	RETURN	DUMMY	DUMMY		
XC0	XC1	XC2	XC3		
XC4	XC5	XC6	STACKLINK		
MS =	DL =	SPM	SET LINK		
XNB =	SN =	XNB +	XNB =>		
SF =	SF +	SF = NB +	SF =>		
NB =	NB = SF +	NB +	NB =>		
= 0	≠ 0	≥ 0	< 0		See note 1
≤ 0	> 0	OVERFLOW	Bn		
= 0	≠ 0	≥ 0	< 0		See note 2
≤ 0	> 0	OVERFLOW	Bn		
0	Bn & X	$\overline{\text{Bn}}$ & X	X		See note 3
Bn & $\overline{\text{X}}$	Bn	Bn ≢ X	Bn V X		
$\overline{\text{Bn}}$ & $\overline{\text{X}}$	Bn ≡ X	$\overline{\text{Bn}}$	$\overline{\text{Bn}}$ V X		
$\overline{\text{X}}$	Bn V $\overline{\text{X}}$	$\overline{\text{Bn}}$ V $\overline{\text{X}}$	1		

f' BITS

12	13	14	15	PROP				SEOP		B		ACC
0	0	0	0	MS	NB	CO		DR		BOD	B	AEX
			1			XNB		XDR			BOD	
		1	0			SN	NB		DT	Z		
			1			SN	SF		XDT			
	1	0	0			BN			DOD			
			1									
1	1	1	1									

NOTES:

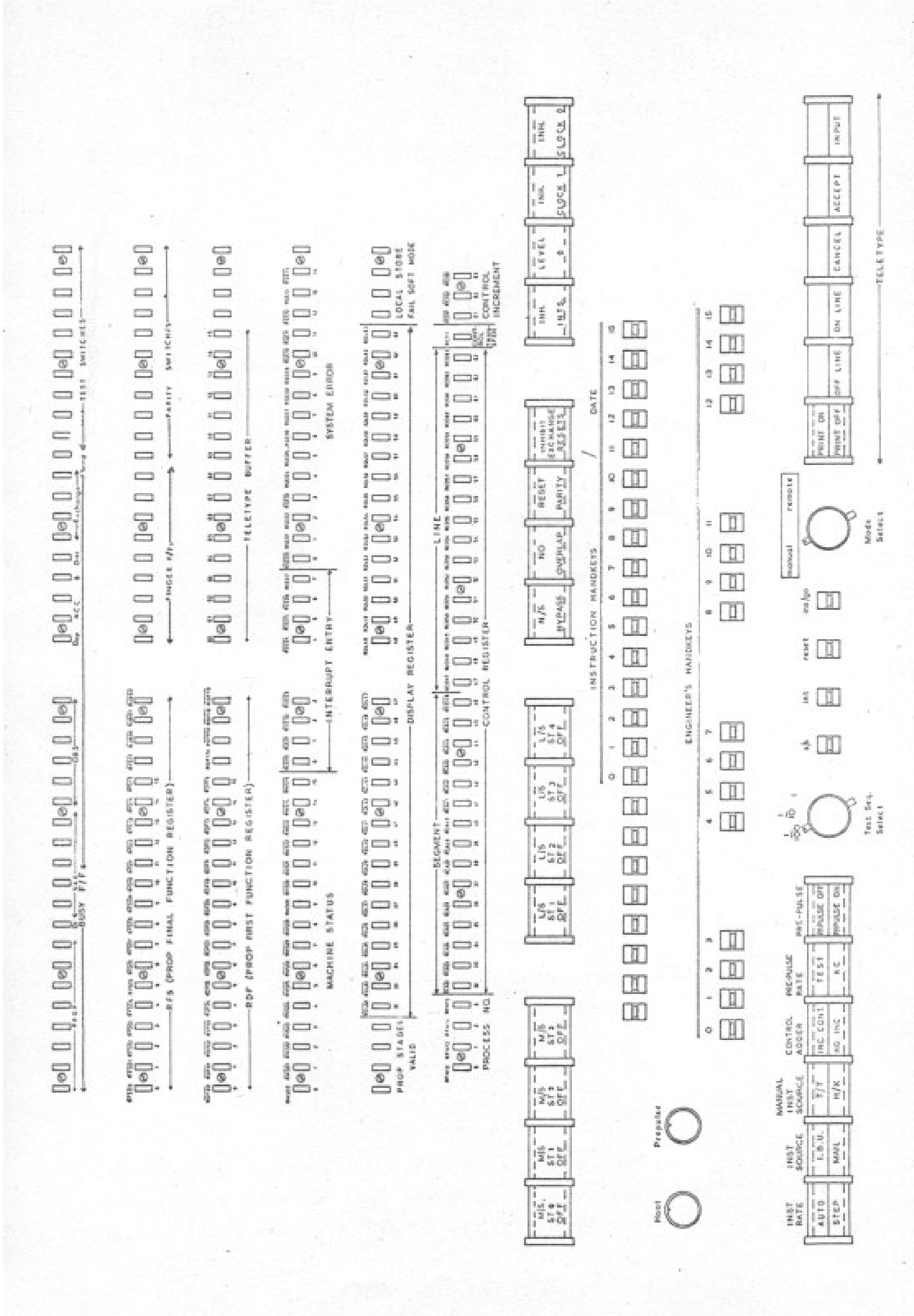
1. –> if TEST
2. Set Bn if TEST (fn given by operand)
3. Set Bn (X = l.s. bit of operand)

14.1.74
RNI

Reproduced by RNI Jan 2017

MU5 INSTRUCTION SET – FUNCTIONS AND INTERNAL REGISTERS

Appendix III Engineer's Console Front Panel



I / FIG 3 CONSOLE FRONT PANEL