ICL
1906A document IM87
parts of Vols. 1-3

CHAPTER 2

PAGING

2.1 Introduction

When multiprogramming techniques are being used with large computer systems, problems arise over core store organisation. Programs are stored both in the core store and in the backing store. As some programs are deleted and new programs are started, there is an interchange of programs between these two locations. Programs are of varying length and a new program will not necessarily fit into the 'gap' in the core store created by a deleted program. If such 'gaps' were left unused, this would limit the number of programs which could be held in the core store and increase the requirement for transfers between the core store and backing store. One method of overcoming this problem is to move programs within the core store to fill up any 'gaps'. This method, used by the Datum and Limit system, is acceptable on small machines but with the larger systems it becomes an impracticably slow process.

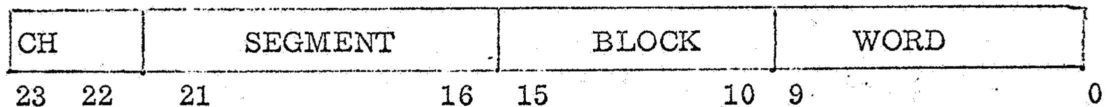
Another problem concerns programs and parts of programs (routines) which can be employed by different users. It may be a complete program for which each user supplies his own set of data or a routine which can be incorporated in various programs. With the Datum and Limit system, a programmer can only refer to addresses within the area allocated to his program and any general routine (pure procedure) must actually be incorporated in this program. It would obviously be economical to keep just one copy of such pure procedures in the core store and allow any user to access them if required.

2.2 The Paging System

2.2.1 Outline

The Paging system was introduced to overcome the problems outlined above and the main features of this system are:

1) All programs are divided into blocks of 1K word length. For convenience, a group of 64 contiguous blocks is termed a segment. The format of an address is thus:



2) The store is divided into pages of 1K word length.

3) Any block of program may be inserted in any available page in the store.

4) A set of Paging tables enable any block starting address to be correlated with the appropriate page starting address. Bits 10-21 of a relative (virtual) address can then be replaced by the corresponding bits of the absolute address. Note that bits 0-9 specify a word within a 1K block or page and are thus the same for relative and absolute addresses.

5) All information is held in the backing store and copied into the core store as required. If a page is written to in the core store, the backing store must be updated before the core store page is overwritten.

6) The organisation of programs in the core store and the transfer of programs between the core store and backing store is controlled by the supervisor programs George 4 and Executive.

2.2.2 Advantages

The advantages claimed for the Paging system are as follows:

1) It results in faster store accesses than the Datum and Limit system.

2) The programmer is not confined to a small continuous range of addresses.

3) It is not necessary to shift the remaining programs in the store to close up the 'gap' created when a program is deleted.

4) Programs of low priority and parts of programs can be held in the backing store.

5) If the program refers to a block in the backing store, this block will automatically be brought into the core store.

6) If a routine is used by more than one program, only one copy of the routine need be kept in the store. Such routines are held in a common area of store (program 0's store area) and are made available to an object program by inserting replacement addresses in the tables for that program

7) The Paging tables may be set up so that one copy of a program can be used simultaneously by two or more users operating with different data.

2.2.3 Organisation

The Paging system is organised by the supervisor programs George 4 and Executive. An obvious requirement is the facility to convert (or translate) a relative address into an absolute address. The software maintains three basic types of table, accessible to the hardware, which permit this. The basic tables are the Program Table, the Segment Tables (one per program) and the Page Tables (nominally 16, 32 or 64 for each program).

Note that on 1906A, the paging hardware is referred to as the Address Translator.

Some other functions of the software are:

1) Allocating pages of core store to a program (quota fixing). This allocation may be based on the program's past requirement for pages.

2) Swapping (new) programs in to the core store and (deleted) programs out to the backing store.

3) Swapping a program's pages in and out of core store (page turning). Note that when a program requires a page from the backing store, this is swapped for one of the program's pages in the core store: i.e. the program's quota of pages in core store remains unchanged.

4) Arranging for certain pages of store to be shared by different programs. A program is allowed access to a shared page by inserting replacement addresses in the basic tables for that program.

To facilitate these operations, the software maintains various other tables which are not accessible by hardware. Some of these tables are described briefly here but for fuller information the reader should refer to the appropriate software manual.

2.3 The basic Paging Tables

2.3.1 The Program Table

The Program Table contains a two-word entry for each program. The format of the entry is:

Word N

SEGMENT TABLE LENGTH (2 bits)	STARTING ADDRESS OF PROGRAM'S SEGMENT TABLE (22 bits)
23 22	21 0

The m.s.two bits indicate the number of entries in the Segment Table:

Bit 23	22	
0	0	Not used (by hardware)
0	1	16 entries
1	0	32 entries
1	1	64 entries

Word N + 1

G register information (and Limit in D & L mode)

When a program is entered, the 172E instruction loads the contents of word N into the Address Translator PD register and the contents of word N + 1 into the G register.

2.3.2 The Segment Table

A program's Segment Table has 16,32 or 64 single word entries. The format of an entry may be:

PAGE TABLE LENGTH (2 bits)	STARTING ADDRESS OF A PAGE TABLE (18 bits)	Not Used	PAGE TABLE NOT IN CORE STORE (1 bit)
23 22	21 4	3 1	0

The m.s. two bits specify the number of entries in the Page Table or replacement.

Bit 23	22	
0	0	Replacement (shared page)
0	1	16 entries
1	0	32 entries
1	1	64 entries

Note that as a Page Table contains at least 16 entries, it is only necessary to record the m.s. 18 bits of the Page Table starting address (bits 0-3 will always be zero). When replacement is specified, the replacement address occupies bits 0-21 of the entry.

2.3.3 The Page Table (Segment Description)

The Page Table actually forms part of a Segment Description (SD) Table. The SD Table contains a two word entry for each page in a segment and the first words of each entry comprise the Page Table. The format of a Page Table entry is:

AVAILABILITY BITS (2 bits)	PAGE STARTING ADDRESS (12 bits)	PERMISSION BITS (3 bits)	Not Used
23 22	21 10	9 7	6 0

The m.s. two bits indicate the availability of the page or replacement:

Bit 23	22	
0	0	Replacement (shared page)
0	1	Page not in core store
1	0	Page in core store and available
1	1	Page in core store but not available

When replacement is specified, the replacement address occupies bits 0-21 of the entry.

The significance of the permission bits is:

Bit 7 = 1	This page may be written to (Write)
Bit 8 = 1	Operands may be read from this page (Read)
Bit 9 = 1	Instructions may be read from this page (Obey)

The other part of the SD Table (i.e. the second word of each entry) is not accessible to the hardware. This part of the table lists the backing store location (or home) of each page together with other information about the page.

2.4 Address Translation

A programmer's relative address may be translated into an absolute address by reference to the three basic tables described in section 2.3. The stages are as follows:

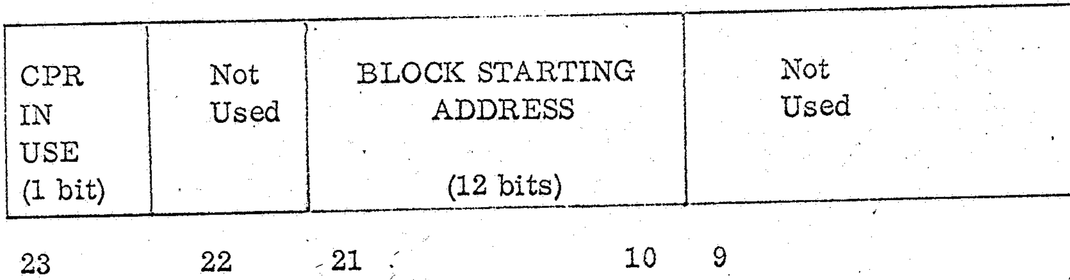
1) The segment part of the relative address (bits 16-21) is added to the Segment Table starting address (in PD) to obtain the address of the relevant entry in the Segment Table. This entry provides the starting address of the appropriate Page Table.

2) The address of the required entry in the Page Table is obtained by OR'ing the block part of the relative address (bits 10-15) with the Page Table starting address (in bit positions 0-5). This entry yields the required page starting address.

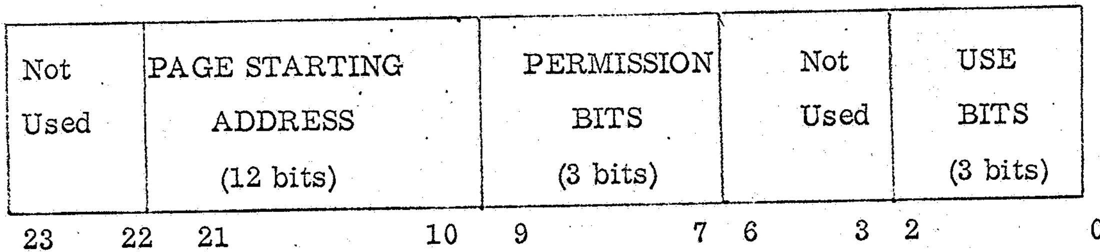
3) Bits 10-21 of the relative address are replaced by the page starting address to form the required absolute address.

2.5 The Current Page Registers (CPR's)

To obviate the necessity of referring to three different tables whenever an address translation is required, the Address Translator is provided with 16 CPR's. These hold the page starting addresses of 16 blocks of program currently in use. A block starting address is stored in one half of the CPR as shown:



The other half of the CPR holds the corresponding page starting address:



Note that bits 7-21 are a copy of the corresponding bits in the Page Table entry. The significance of the Use Bits is:

Bit 0 = 1	Page has been written to
Bit 1 = 1	Page has been read from
Bit 2 = 1	Page has been obeyed.

Associated with each CPR is a lock bit. CPR's may be locked to protect their contents as follows:

a) CPR0, containing block 0 of the program, is loaded and locked when the program is entered. (The first store access loads accumulator X0 from relative address 0). This CPR remains locked until a program change occurs.

b) The CPR containing the current instruction is kept locked.

c) Up to four CPR's may be locked to facilitate the execution of the multi-operand orders 116, 126 and 127.

When an address translation is requested the block part of the relative address is simultaneously compared with the block starting addresses in the 16 CPR's. If equivalence is found it only remains to read the required page starting address from the other half of the CPR. Note that the address can only be used however, if the type of store access required is allowed by the Permission Bits. Also, the type of store access is recorded in the Use Bits.

If equivalence is not found in a CPR, the required page starting address must be traced (as in section 2.4) and loaded into a CPR, together with the block starting address. The CPR to be loaded is selected by a Next Vacant Page (NVP) Counter. This counter will be pointing to the CPR whose contents (if any) are least likely to be required. It is not allowed to point to a locked CPR and it never points to a CPR which has just been loaded. If the CPR selected for loading already contains a store page address, its In Use bit will be set. In this case, the usage of the old page is recorded in a Store Use Table (see section 2.6.1) before the CPR is loaded with the new page address.

2.6 Other Paging Tables

2.6.1 Executive tables

There are two tables in Executive's store area, namely the Store Use Table (SUT) and the Store Lockdown Table (SLOT). SUT may be accessed by the hardware and both tables are accessible by George 4. Each entry in SUT refers to a particular page of store and is half a word long:

Not Used	PROCESSOR NUMBER (Bit significant)	USE BITS (3 bits)	Not Used	PROCESSOR NUMBER (Bit significant)	USE BITS (3 bits)
23	22 – 15	14 – 12	11	10 – 3	2 – 0
	Even page			Odd page	

If a CPR which is In Use is unloaded, its Use Bits are recorded in the appropriate SUT entry. Note that this does not destroy any Use Bits already set in the SUT entry. When a CPR is loaded, the Use Bits in the SUT entry are loaded into the CPR together with the page address. The Processor Number bits cater for multiprocessor configurations. They identify the processor which is using a page of store at any particular time, i.e. the processor which holds the page address in one of its CPR's. The Processor Number is deleted when a CPR is unloaded and written in when the CPR is loaded.

The starting address of the SUT is some fixed number (say T), which is a multiple of 2K. The address of the entry for a page p therefore is T + p/2. Note that when a SUT access is required, p will be the page address inthe CPR which is being unloaded or has just been loaded.

The other Executive Table SLOT also contains a half word entry for each page of store. Part of a SLOT entry is the Lockdown Count for the page. This gives a indication of the peripheral transfers involving the page and other processes which require that this page should be kept in the core store.

2.6.2 George 4 tables

There are two tables in George 4's store area which are used only by George 4. These are the George Lockdown Table (GLOT) and the Page Usage Table (GPUT). In both these tables, an entry refers to a particular page of store. A GLOT entry is half a word long. It contains a Lockdown Count which is referred to when page turning. That is, when a program's page has to be swapped out of the core store, (to make room for a new page) the page selected will not be a page with a non-zero Lockdown Count. The GLOT entry also contains a copy of the Use Bits from SUT.

A GPUT entry is one word long. The entry is significant only if the page is occupied by an object program, in which case the entry contains information about the usage of the page. This information includes:

a) A bit which indicates whether or not the page is occupied by an object program.

b) A decay count of the number of page swaps since the page was last accessed.

c) The number of programs with access to the page. (This number will only be greater than 1 if it is a shared page).

d) A more permanent record of the Use Bits in SUT and GLOT.

CHAPTER 2

ADDRESS TRANSLATOR

2.1 Introduction OD3

The address translator contains the hardware necessary to convert a relative address into an absolute address using either the datum and limit or paging systems. The relative address enters the translator from the instruction unit via the J highway. When the absolute address has been formed it is placed in one or more of the three address registers AA, AY & AW where it is available to address the store. Absolute addresses and literals bypass the translation logic and are loaded directly into the AA register. Literal operands are then conveyed directly to the fixed point unit.

In the paging system the most significant twelve bits of the relative address are compared with the block starting addresses in the sixteen current page registers. If equivalence is found, then the corresponding page starting address is gated out to form the most significant twelve bits of the absolute address. When equivalence is not found the required page starting address is traced by reference to the program, segment and page tables in turn using the Table Search routine. The page starting address and the associated block starting address are then loaded into the current page register indicated by the next vacant page counter. Equivalence will then be found.

In the datum and limit system, the relative address is datumised and the absolute address so obtained is then compared with the limit address.

Operations in the address translator are controlled by the address translator microprogram (OD12).

2.2 Block address registers (PBx) OD3/J2

These sixteen registers which form a part of the less significant word of each current page register hold the starting address of a block of program. Each twelve bit register (PB0010-21 to PB1510-21) is loaded from AA10-21 by strobe NTPB which is produced under the following conditions:-

a) When a current page register is being loaded from store at the end of the table search microprogram.

b) When the contents of an X register are being written into the odd half of a current page register by a 171E instruction.

The data is routed to the appropriate register by the decoded address signals ANVP00-15. The output is taken to the equivalence check logic and also to the TB and PR highways.

2.3 Block address equivalence OD3/G2

The block address in each current page register is compared with the most significant twelve bits of the relative address on the J highway (J10-21). If equivalence is found the signal EQx is generated, providing that the in use bit PBx23 and NEQT are set and the write sequence is not in operation. The signal EQx generates NPTx which gates the page starting address from PPx onto the TP highway.

2.4 Page address registers (PPx) OD3/K2

These sixteen registers, which form part of the more significant word of each current page register, hold the starting address of a 1K page of store containing the block of program specified by the corresponding block starting address. Each twelve bit register (PP0010-21 to PP1510-21) is loaded from AA10-21 by strobe NTPP which is produced by the microprogram at the following times:-

a) When a current page register is being loaded from store at the end of the table search microprogram.

b) When the contents of an X register are being written into the even half of a current page register by a 171E instruction.

The data is routed to the appropriate register by the decoded address signals ANVP00-15. If equivalence has been found the contents of this register are gated onto the TP and PR highways to convert a relative address into an absolute one.

2.5 In use bistables (PBx23) OD3/G2

These sixteen bistable circuits (PB0023 to PB1523) form bit 23 of the less significant word of each current page register. They are used to indicate that the current page register is in use, and must therefore be unloaded before it can be overwritten. The bistable is set by NTPB and ANVPx at the same time as the block address is loaded, and is reset by NRU and ANVPx during the unloading part of the table search sequence.

Bits 00-09, 21 and 22 of the less significant word (PBx) of each current page register are not used.

2.6 Use bistables (PPx00-02) OD3/L2

These sixteen sets of three bistable circuits (PP0000-02 to PP1500-02) form part of the more significant part of the current page registers. They are used to record the types of store access which have taken place while the page of store is defined in the current page register. The significance and logic of each bit is as follows:-

a) Bit 00 indicates that an operand has been written to the page. It is set by NTWU and NPTx for write operands (WOP).

b) Bit 01 indicates that an operand has been read from the page. It is set by NTRU and NPTx for read operands (RQRD).

c) Bit 02 indicates that an instruction has been obeyed from the page. It is set by NTOU and NPTx for instructions (RQI).

All these bits are reset by NRU and ANVPx during the unloading part of the table search sequence.

2.7 Permission bits and decoding (PPx07-09) OD3/L2

Bits 07-09 of the more significant part of each current page register record the type of store access which is permitted for this page of store. The significance of each bit is:-

a)	Bit 07	An operand may write to this page
b)	Bit 08	An operand may be read from this page
c)	Bit 09	An instruction may be read from this page

These three bits are loaded in the same way and at the same time as the page starting address by strobe NTPP and ANVPx.

The bits are decoded with WOP, RQRD and RQI respectively, and the signal PERMIT is only generated if the appropriate permission bit is active.

Bits 03-06, 22 and 23 of the more significant word of each current page register are not used.

2.8 Lock bistables (PLKx) OD3/F2

Each current page register has a bistable circuit associated with it to indicate that it is locked and may not be unloaded and overwritten. The bistable circuit is set by NTLK and NPTx when the contents of the current page register are likely to be required. While it is set the contents of the current page register cannot be unloaded as the next vacant page counter is not allowed to point to a locked register. All the locked bistables except PLK00 are unlocked by NTUN when an instruction address in a new page (RQNP) or a jump address (RQJ) is being translated. All the locked bistables are unlocked by NTUN00 (for PLK00) and NTUN during the part of the 172E and 173E microprogram which unloads all the current page register. Individual lock bistables may be unlocked by NTUK within 40ns of being locked. This mechanism is used to step on the next vacant page counter so that it always points to the least used current page register.

Up to six current page registers may be locked at any time. Register 00, which refers to the first 1K words of the program is locked when the program is entered and remains locked until a program change occurs. The register containing the current instruction address is kept locked, and up to four others (defining transfer areas) may be locked when a multi-operand instruction (116, 126, 127) is in progress.

OD3/E2

2.9 Next vacant page counter (NVP)

Whenever equivalence cannot be found in the paging system, then a current page register must be unloaded (if it is in use) and then loaded with the required block and page addresses. The next vacant page count ensures that the current page register which is unloaded is the one whose contents are least likely to be required. Each time a paging access is made and equivalence is found, the equivalent register is temporarily locked (NTLK) and then unlocked (NTUK) within 40ns. The counter is not allowed to point to a locked register, so if it happens to be pointing to the (locked) equivalent register the signal EQVP is generated and it will be stepped on to the next current page register addre by strobe NTVP. If the following registers are also locked then the counter will be stepped on again until it points to an unlocked register.

The four bit counter is incremented by one each time that strobe NTVP is received, and may be reset to zero by NRVP which is produced during the unload sequence of the 172E and 173E instructions. The four bits are decoded to give the current page register selection signals ANVP00 to ANVP15.

OD3/D2

2.10 Special register address register (APV)

The current page registers may be read from or written to as special registers 1024-1055 using 170E or 171E instructions respectively. The even addresses read the page parts and the odd addresses the block parts of the current page registers. The address is routed via the J highway from the instruction unit to the address translator where the least significant five bits (J00-04) are loaded into the special register address register APV0-4 by strobe NTAPV. This is produced at the beginning of the 170E and 171E address translator microprograms. The least significant bit (APV0) is used by the microprogram to select the appropriate half of the current page register. The other four bits (APV1-4) are decoded to give the current page register number using the same logic which decodes the next vacant page counter address. This only takes place when NAN is set.

OD3/H4, K4, M4

2.11 Fan-in to PR highway

The PR highway is used to convey data from the appropriate part of the current page register to the short code fixed point unit and to the mark in use table held in the PW register. The data is obtained from the less significant word via the TB highway when NBV is set, and from the more significant word via the TP highway when NPV is set. NPV or NBV are set during the 170E instruction which reads half of the contents of a current page register depending on whether the address is even or odd. NPV is also set when the mark in use table is being updated during the table search sequence.

2.12 Mark in use register (PW) OD3/A8

This twenty four bit register holds an updated use table entry while it is being written to store. The word read from the use table in store contains the entries for two pages. Bits 00-11 refer to an odd page and bits 12-23 to an even page. When loading PW from RE the entry referring to the page in the current page register which is not being unloaded is strobed into PW unchanged by strobe NTPW. For the entry which is being updated however, the bit significant processor number (bits 03-10 or 15-22) is inhibited by NUWAM or NUWNZ. Also the use bits from the current page register being unloaded which are available on PR00-02 are ored into PW00-02 or 12-14 with the use bits from the store (RE00-02, or 12-14). The complete pair of use table entries is then written back to store via the PW highway.

When the current page register has subsequently been reloaded it must be marked in use in the use table. The use table entry is again read out to PW and this time the processor number (PROC7-0) is loaded into PW03-10 (NMWAM) or PW15-22 (NMWNZ). The updated entry is again written back to store.

2.13 Segment table base address register (PD) OD3/A5

This register holds the program's segment table base address, and is loaded by a 172E instruction when the program is entered. The address is read from store (word N) and routed to the PD register on the RE highway and is strobed into the register by NTD. Bits 00-05 of PD are routed to the segment table adder for addition of the segment part of the relative address and bits 06-21 go directly to the entries to the T highway. Bits 22 and 23 are used to specify the segment table length as follows:-

Bit 23	22	
0	0	Not used
0	1	16 entries
1	0	32 entries
1	1	64 entries

2.14 Segment table adder OD3/B5

This adder is used to add the segment part of the relative address (on J16-21) to the segment table base address in the PD register. A simplified diagram of one slice of logic is shown in Fig. 2.1 together with the associated truth tables. A block diagram of the carry system is given in Fig. 2.2. It will be seen that the carries are formed in two blocks of three bits each. Carries in each block are formed in parallel but the block carry is in series. The carries from bits 4 to 6 are decoded along with the table length indicator bits (PD22-23) to determine when the segment is out of range (SOOR). The output of the adder is taken to the entry gates to the T highway via the PDS highway.

2.15 Datum address register (DT) OD3/B1

This sixteen bit register (DT06-21) is used to hold the datum address of the program. It is loaded when the program is entered from word N specified by a 172E instruction. The data is routed from store via the RE highway and is strobed into the register by signal NTD. The output is taken to the datum adder except during the first part of a datum and limit 177E instruction (NDLC active).

2.16 Datum adder OD3/C2

The sixteen bit datum adder is used to add the most significant sixteen bits of the relative address on J06-21 to the datum address to give the absolute address, which is sent to the T highway entry gates via the DTS highway. The absolute address is also checked to ensure that it is less than the limit address.

The basic logic of one slice of the adder is shown in Fig. 2.3 together with its associated truth tables. A block diagram showing the carry system is given in Fig. 2.4. The carry system is divided into four blocks of four bits each. Individual carries in the first block are produced in series and in the other three blocks in parallel. The block carries are generated in parallel. If the carry into bit 22 is active (DTCY22) then a reservation fail is signalled (RESFL) because the absolute address is greater than 4M-1.

2.17 Limit address register (LM) OD3/A1

A sixteen bit register (LM06-21) which holds the limit address of the program. It is loaded when the program is entered from word N + 1 specified by a 172E instruction. The data is routed from store via the RE highway and is strobed into the register by signal NTLM. The output is taken to limit check logic and also to the zero detection logic which indicates a zero limit address (4M) by the signal LZERO.

2.18 Limit check OD3/B3

This checks the datumised address to ensure that it is less than the limit address. The limit address in LM is subtracted from the datumised address obtained from the output of the datum adder. The basic logic is shown in Fig. 2.5 together with the relevant truth table. The logic is only used to determine whether the carry into bit 22 is active or not; all other logic is omitted as it is necessary).

If the datumised address is greater than or equal to the limit, this constitutes a reservation fail and it is detected by the absence of carry into bit 22 of the limit checker ($\overline{\text{LMCY22}}$). Note however that if the program is at the top of the store, its limit address (4M) will be represented by all zeros (LZERO) and there will be no carry into bit 22. In this particular case RESFL is not generated.

2.19 Fan-in to T highway

This twenty four bit highway is used to load the absolute address buffers. It receives data from a number of sources as shown below:-

a) From store via the RE highway when slack gate NST is set. This is used by the table search microprogram when the page table base address is being loaded into AA, and also when the page starting address is being loaded into a current page register via the AA register. It is also used in a similar way by a 171E instruction which is writing to half a current page register as a special register.

b) From a fixed number UT11-21 to T11-21 and from a page starting address in a current page register TP11-21 to T00-10 when NUT is set. This is used to obtain the address of the mark in use table entry relating to a particular pair of current page registers. The fixed number UT11-21 is wired in using C3 elements.

c) The segment table address is gated from the segment table adder PDS00-05 to T00-05 and the segment base address register PD06-21 to T06-21 when slack gate NDT is active. This address is used to obtain the appropriate page table base address from store.

d) From J highway J10-15 to T00-05 when slack gate NJPT is active. This is used to gate the page part of the relative address into the bottom six bits of the page table base address which has already been loaded into AA by the table search microprogram. When a page table has only 16 or 32 entries, bits 04-05 of the page table base address may be active. These are preserved by oring AA04-05 with J14-15 when NJPT is set.

e) From the J highway to the T highway on the following occasions:-

   1) Loading the page starting address into a current page register.
   J10-21 gated to T10-21 by NJTLZ and $\overline{\text{NWT}}$.

   2) Loading an absolute address directly from the J highway when the instruction is a literal or has an absolute address.
   J00-05 gated to T00-05 by NJTAF and $\overline{\text{NWT}}$.
   J06-09 gated to T06-09 by NJTGK and $\overline{\text{NWT}}$.
   J10-21 gated to T10-21 by NJTLZ and $\overline{\text{NWT}}$.

   3) The least significant ten bits of a relative address on the J highway are combined with the page starting address from the current page register to give the absolute address in paging mode.
   J00-05 gated to T00-05 to T00-05 by NJTAF and $\overline{\text{NWT}}$.
   J06-09 gated to T06-09 by NJTGK and $\overline{\text{NWT}}$.

4) The least significant six bits of a relative address on the J highway are combined with the datumised address from the datum adder to give the absolute address in datum and limit mode.
J00-05 gated to T00-05 by NJTAF and $\overline{NWT}$.

f) The page starting address from PPx and the TP highway (TP10-21) is gated by NPT & $\overline{NWT}$ to T10-21 to form a complete absolute address with the appropriate parts of the relative address.

g) The datumised address from the output of the datum adder (DTS06-21) is gated by NDDT & $\overline{NWT}$ to form a complete absolute address with the part of the relative address from the J highway.

h) A write operand address held in the AW register is routed to the AA register via the slack gate NWT and the T highway. This slack gate inhibits the gates from the J, PT and DTS highways as these might possibly be set at the same time as NWT.

2.20 Absolute operand address register (AA) — OD3/F7

This twenty four bit register is used to hold the absolute address of an operand until the address is accepted by store. It receives its input from the T highway (see Section 2.19). The bistable circuits are split into two groups; AA00-05 are strobed by NTAAF, and AA06-23 by NTAGZ. Normally both strobes occur simultaneously, but when the page table address is being formed only NTAAF is generated and when the page starting address is being loaded into a current page register only NTAGZ is active. The output of the register is taken to the distributor via the AH highway, to the short code fixed point unit for literal operands, to the current page registers for loading purposes and to the decoding logic. Bits AA00, 22 & 23 are decoded to provide signals for the microprogram. These are as follows:-

a) For a segment table entry; PTAB = AA00 indicates that the page table is not in the core store.

b) For a segment table or page table entry; PREP (AA22-23 are both zero) indicates a replacement address.

c) For a page starting address; PAV (AA22 is zero, AA23 is one) indicates that the page is in the core store and is available.

d) For a page table base address; POOR indicates that the page is out of range. The logic decodes AA22-23 and J14-15 (relative address most significant page bits) to ensure that the page table address will be within the length of the table specified.

2.21 Absolute instruction address register (AY) OD3/H7

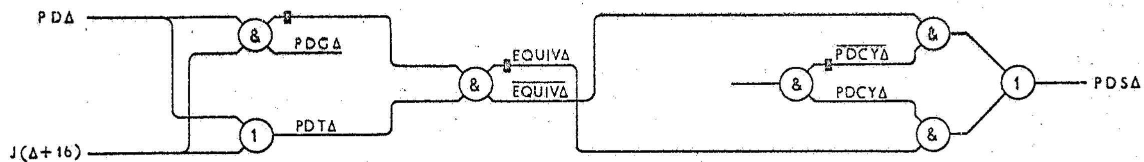
This sixteen bit register holds the most significant part of the current translated instruction address. The least significant part of the address is obtained from the AC register in the instruction unit (see Section 2.23). The address is strobed into the register by NTY from the T highway (T06-21), whenever the instruction address is translated for a sequencer new page request or a successful jump.

2.22 Absolute write operand address register (AW) OD3/K7

A write operand address on the T highway is always loaded into AA, but it is not used at this time and AA is set free. The address is also loaded into the AW register by strobe NTW where it is held until the write cycle is about to take place. The twenty two bit write address is then transferred back to the AA register via the slack gate NWT and the T highway, from where it can be used to address the store.

2.23 Fan-in to AH highway OD3/J8

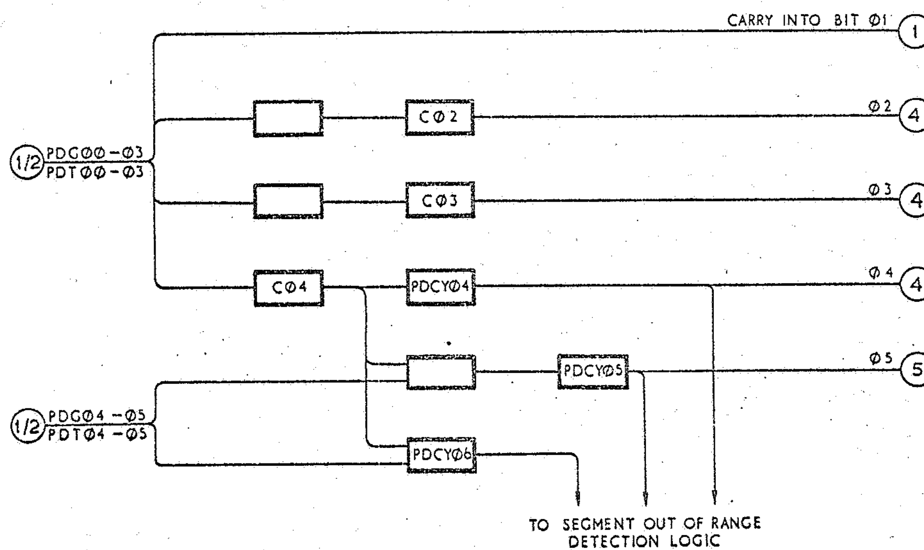
This twenty two bit highway is used to address the store via the distributor. Operand addresses are routed from AA00-21 when slack gate NAAH is set. Instruction addresses are provided from the AY and AC registers when NYAH is active. In paging mode the address is formed from AC00-09 and AY10-21, while in datum and limit mode it is formed from AC00-05 and AY06-21.



PDΔ	J(Δ+16)	PDGΔ	PDTΔ	EQUIVΔ
Ø	Ø	Ø	Ø	1
1	Ø	Ø	1	Ø
Ø	1	Ø	1	Ø
1	1	1	1	1

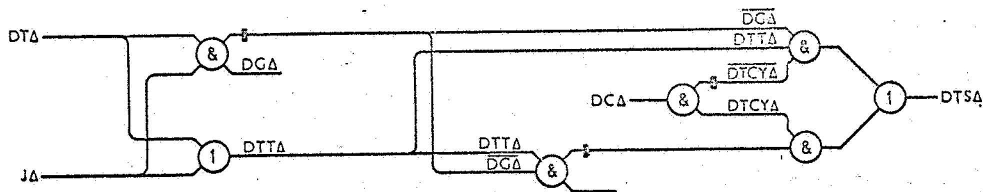
EQUIVΔ	PDCYΔ	PDSΔ
Ø	Ø	1
1	Ø	Ø
Ø	1	Ø
1	1	1

FIG. 2.1 SEGMENT TABLE ADDER BASIC LOGIC



○ NUMBERS IN CIRCLES INDICATE NUMBER OF ELEMENTS USED COUNTING FROM THE PD AND J HIGHWAYS.

FIG. 2.2 SEGMENT TABLE ADDER CARRY SYSTEM



DTΔ	JΔ	DGΔ	DTTΔ
ϕ	ϕ	ϕ	ϕ
1	ϕ	ϕ	1
ϕ	1	ϕ	1
1	1	1	1

DTΔ	JΔ	DCΔ	DTSΔ
ϕ	ϕ	ϕ	ϕ
		1	1
1	ϕ	ϕ	1
		1	ϕ
ϕ	1	ϕ	1
		1	ϕ
1	ϕ	ϕ	ϕ
		1	1

FIG 2.3 DATUM ADDER BASIC LOGIC.

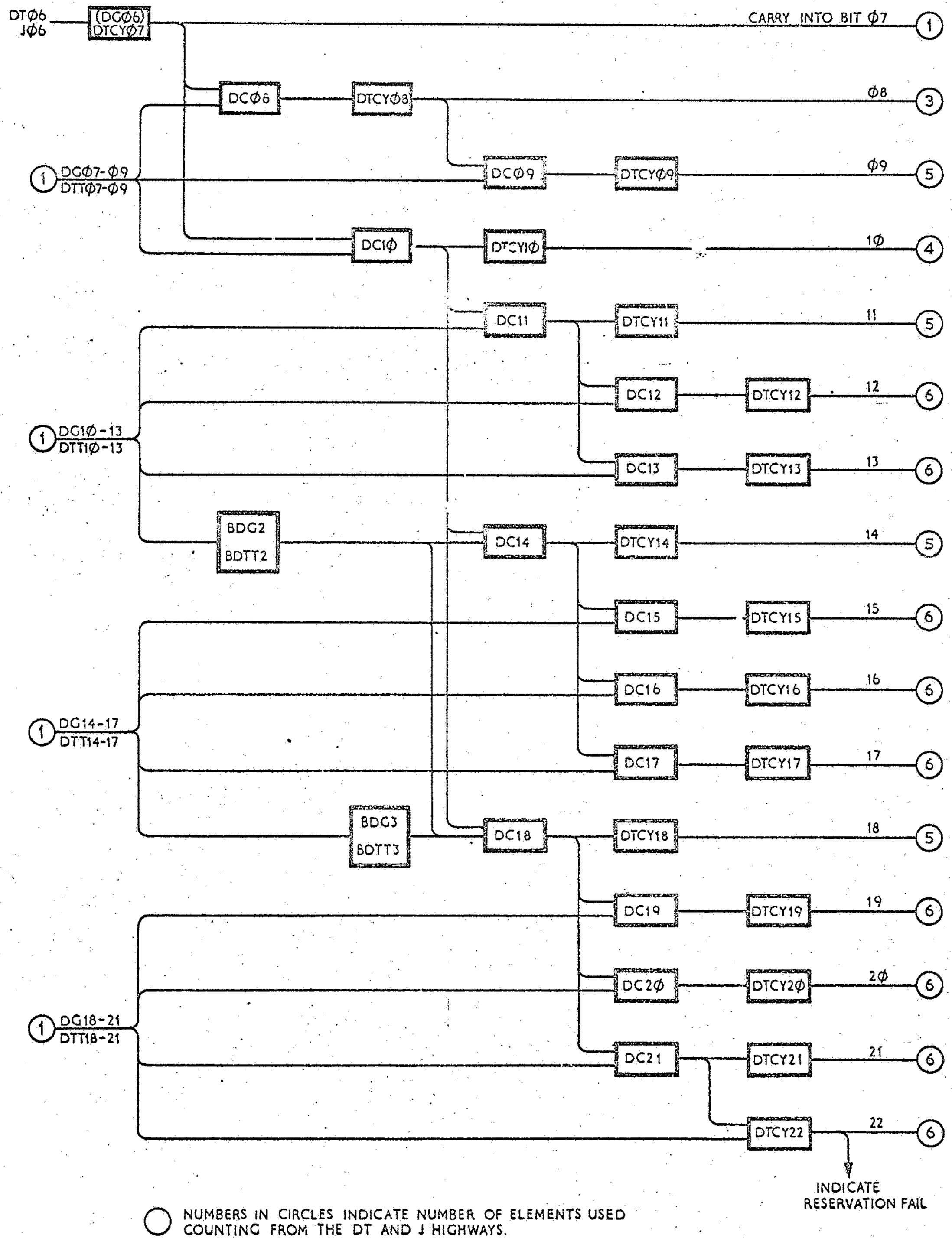
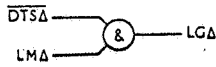
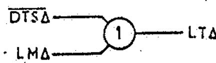


FIG. 2·4 DATUM ADDER BLOCK CARRY SYSTEM.





DTSΔ	LMΔ	LGΔ	LTΔ
Ø	Ø	Ø	1
1	Ø	Ø	Ø
Ø	1	1	1
1	1	Ø	1

FIG 2.5 LIMIT CHECK BASIC LOGIC

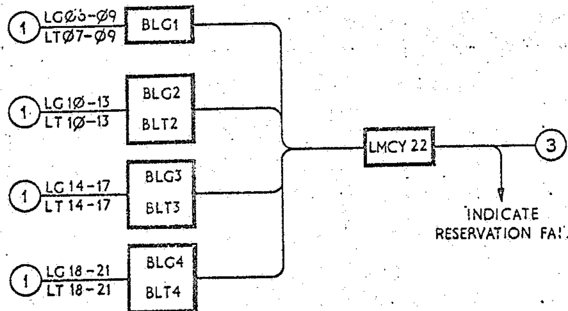


FIG 2.6 LIMIT CHECK CARRY SYSTEM

CHAPTER 2

ADDRESS TRANSLATOR MICROPROGRAM

2.1 Introduction

The addresses presented to the machine from the object programs will not, in a multiprogramming mode of operation, be absolute addresses of the core store, but will be relative to some datum address associated with the particular program. It is the purpose of the address translator to convert these relative addresses to absolute addresses in order that the required entry in the core store can be located.

The 1906A machine offers a choice of either a 'Datum and Limit' system or a 'Paging' system. The principles of these two systems are described in Volume 1 Chapter 2, but a brief summary is included below.

When the datum and limit system is in use, the PD (Program Datum) register will have been loaded with the program's datum by the 172E instruction, when the program was entered. This datum must be added to the relative address to give the absolute value of the address before the relevant entry in the core store can be located. In practice the instruction address is only datumised every sixty four words, providing no branching occurs. All branch addresses and operands are datumised but literals bypass the sequence. There is also a 'reservation' check on the addresses to ensure they fall within the space reserved for the program, which is terminated by the program's 'limit'.

In the paging mode of operation, the relative address is considered in three sections; bits 0-9 as the word, bits 10-15 as the block, and bits 16-21 as the segment, there being 1K words per block. The core store is divided into pages, also 1K words long, and when the program is loaded into the core store, each block of program occupies a page of core store, these pages not necessarily being consecutive. It is the task of the paging unit to keep a record of the pages into which the blocks of program have been loaded, and to locate these blocks and hence the particular words when they are required.

When the paging system is in use, the PD register will have been loaded by the 172E instruction with the starting address of the segment table for that particular program. The required entry of the segment table is given by adding the entry in the PD register to bits 16-21 of the relative address. This entry will then contain the starting address of the page table, the entry therein being calculated by adding bits 10-15 of the relative address to the starting address of the page table. The entry will then give the page starting address, bits 0-9 of the relative address giving the required word within the page.

Using this method, it can be seen that three store cycles would be required to translate the address, which would take up too much time and so 16 hardware registers are introduced to overcome this problem. These double length registers, called the 'current page' registers, hold the block starting address of the program in bits 10-21 of the least significant word, and the corresponding page starting addresses in bits 10-21 of the most significant word. It is therefore only necessary to compare the required block address with the block addresses held in the C.P.R.'s, the corresponding page address can then be read out from the C.P.R. which gave equivalence with the block address. This arrangement thus obviates the necessity of several core store cycles, provided of course the required block address is held in one of the C.P.R.'s If none of the C.P.R.'s hold the required block address, then a table search must be initiated involving several store cycles as explained above. When the required page is found it is loaded into one of the C.P.R's together with the block address. A second equivalence check on the address should then prove successful, and the C.P.R. will be left with the information stored in it to translate any other addresses within the same block range for that program.

2.2 Paging

2.2.1 General

When a program is running in the machine, only certain addresses need to be translated, these being the ones which when the address count is advanced, either cross a page boundary i.e. move out of one page of the core store into another, or, are such that there is a possibility this may happen. The addresses can be broadly divided into three groups, (a) Operands; (b) Instruction Addresses and (c) Branch Addresses, the action of the address translator being different for each group. There are however, within the address translator two sequences which are common to all three groups. These are, the equivalence sequence, and the table search sequence, which are described below and will be referred to in the descriptions of the address groups.

2.2.2 The Equivalence Sequence

As described in Section 2.1 and also in Volume 1 Chapter 2, some of the 16 current page registers will hold information showing the addresses of the pages of core store into which particular blocks of program have been loaded. This is done by holding the starting addresses of the 1K word long page of core store and the starting address of the 1K word long block of program which is loaded into it, in the same C.P.R. as shown in Fig.1

Not Used	Page Starting Address	Permission Bits	Not Used	Use Bits	In Use Bit	Not Used	Program Block Starting Address	Not Used
23 22	21 10	9 7	6 3	2 0	23	22	21 10	9 0

CURRENT PAGE REGISTER FIG. 1

The 'in-use' bit is set when the C.P.R. is loaded, to indicate it is a valid address, for a current program. If it is not set, the page address can be read out, since it would refer to a program which has exited through a Group 17 unload C.P.R. order.

Not all the C.P.R.'s will necessarily have been loaded with addresses relevant to the program which is running, and it may be that not all the blocks of the program have their starting addresses in a C.P.R. As the program proceeds, a C.P.R. will be loaded when a new page is entered and when all 16 C.P.R.'s are full, the C.P.R.'s which were loaded first, excluding any locked C.P.R.'s., will be overwritten with information of a new page. This means that any C.P.R's which are overwritten contain information on the pages least likely to be required again.

It is the function of the Equivalence Sequence to check if any of the C.P.R's hold the starting address of the block of program, within which the address to be translated lies. The m.s. part of the C.P.R. which gave equivalence will then hold the starting address of the page of core store, into which the block of program has been loaded.

The instruction unit microprogram will have gated the address which is to be translated onto the J highway. Bits 10-21 of this address will give the starting address of the block within which the address lies. This block address is checked for equivalence with the block addresses held in the 16 C.P.R.'s. If any one gives equivalence, the corresponding page address (bits 10-21) is read out and used to replace bits 10-21 of the relative address; thus, the block address is replaced by the page address. Bits 0-9 of the relative address are retained unaltered to give the required word within the page of core store. This new address, bits 0-9 from the relative address and bits 10-21 from the page address form the absolute address which can be used to address the core store.

The Address Translator sequence is initiated by the signal TREQ. If TRAN is set, showing a translation is required, the microprogram will be directed by $\overline{DLHK}$ through the Equivalence Sequence. There is a wait for LKF to ensure that the previous translation has left the 'next vacant page' count pointing to an unlocked C.P.R.

The signal NJTAF gates bits 0-9 of the relative address from the J highway onto the T highway (see Diagram 3) and NJTGK gates the block part of the address, bits 10-21 to the equivalence check circuits.

If $\overline{NWT}$ is set, indicating that a write address is not about to be gated onto the T highway, the equivalence check can proceed. If $\overline{NWT}$ is not set, when NEQT is set, bit 21 of the relative address will be inhibited, thus preventing equivalence from being found (O.D.3/H2).

If equivalence is found between bits 10-21 of the relative address and the block address held in one of the C.P.R.'s then, providing the 'in use' bit of that C.P.R. is set, the signal EQx will be made; x being the number of the C.P.R. which gave equivalence. The signal NPTx gates the page address from the C.P.R. which gave equivalence to the TP highway. It is then gated onto the T highway by the signal NPT, produced by the microprogram (OD12/G3).

After equivalence has been found, the absolute address, made up from bits 0-9 of the original relative address and bits 10-21 from the page address of the C.P.R. which gave equivalence is on the T highway. If equivalence was found, the signal EQ will be set. The action of the Address Translator from this point on is dependent upon the type of address to be translated.

If equivalence was not found, then the Table Search Sequence will be entered to load the required page address into a C.P.R. The Equivalence Sequence is then re-entered with TRRQ and if equivalence is still not found, the not equivalence sequence is entered by NEQ (see Chapter 18).

2.2.3 Table Search Sequence

If the equivalence check does not give equivalence from any of the C.P.R.'s., it is necessary for one of the C.P.R.'s to be loaded with new information. The C.P.R. at which the 'next vacant page' count is pointing is used, for this will be unloaded and the information in it is the least likely to be required (see Vol.1 Chapter 2). Before it can be re-loaded, however, a check is made to determine whether the 'in use' bit is set. If the 'in use' bit is set, the 'use table' entry for that block must be updated with the 'use bits' in the C.P.R. before they are destroyed.

If the C.P.R. to which the 'next vacant page' counter is pointing has the 'in use' bit set, the signal ANVPx and PB x 23 (x being the C.P.R. number) will give the signal UVP (O.D. 3/G4).

The table search sequence can be divided into four routines, each being described below.

a) Unload the C.P.R. (S.C.108)

This routine addresses the required one of 16 half word 'use table' entries and reads the contents into the PW register. The contents are then updated with the 'use bits' held in the C.P.R. and then written back to the same address in the core store. The address of the 'use' table is formed in two sections. Bits 11-21 being a fixed number and bits 0-11 being the page address held in the relevant C.P.R. (see Vol.1 Chapter 2).

After the equivalence check has failed to find equivalence, and since the Table Search Sequence has not taken place, $\overline{EQ}$ and $\overline{SRCH}$ will be set. (OD 12/J3).

After setting SRCH, assuming UVP is set, the signals NPV and NUT will be made. NPV & ANVPx will make the signal NPTx (OD 3/H3) which will gate the page address and the use bits of the C.P.R. which is about to be unloaded onto the TP highway. The signal NUT gates the page address from TP 11-21 to T 0-10 and the fixed number from UT 11-21 to T 11-21 to form the 'use table' entry address.

Note: Since the 'use' table entry is only half a word long, TP10 the l.s. bit of the C.P.R. address, is not required to form the 'use table' address It is however, used to select the appropriate half of the word which has been addressed, to give the required entry for the page.

The signal ZUR produces the signals NTAAF and NTAGZ which gate the 'use table' entry address from the 'T' highway into the AA register. ZUR also makes the signals D:ROP and D:NEXT in readiness for the 'read' followed by 'write' cycles. (The signal D:NEXT ensures successive core store cycles for read and write sequences). The PREQ signal is sent to the distributor to initiate the reading of the 'use' table entry from store. NAAH gates the 'use' table entry address onto the AH highway.

If the page address is odd, TP10 will be set, the signal NUWAM being made. If the page address is even, TP10 will not be set so NUWNZ (OD 12/G7) will be made. These signals will direct the 'use' bits from the PR highway to either PW0-2 or PW12-14 which can then be loaded to either the l.s. or m.s. part of the 'use' table address. The original contents of the 'use' table entry are modified by the'use' bits of the C.P.R., thus updating the (half-word) use table entry associated with the block of program. Any of the bits already set in the 'use' table will be left unchanged, but any bits unset may now be set if they have been set in the C.P.R. since the 'use' table was last updated. The sequence then generates NTPW (OD12/G7) which strobes the updated use table entry to the PW register.

The bistable ZUW is set and the strobe ZZUR produced. ZZUR starts the sequence which writes the updated 'use' table back to its original core store location, by re-entering the microprogram sequence at (OD 12/K5). The ZUW bistable generates D:WOP and D:PAG which gives the signal DPW in readiness for the write cycle. A PREQ signal is sent to the distributor, NAAH gating the table address from the AA register to the AH highway. DPW will then gate the contents of the PW register to the core store.

The microprogram then continues by producing NRU (OD 12/H7), which is gated with ANVPx to give NRUx. This signal resets the 'in use' bit PB23 of the C.P.R. being unloaded. NRU is also gated with ANVPx to reset the 'use' bits. The microprogram then sets the signal ZZUW.

b) Read from segment table (S.C.109)

The address of the entry in the segment table which contains the page table start address is formed in the AA register by adding the segment part of the relative address (bits 16.21) to the segment table base address in the PD register, (see Diagram 5).

The 'unload C.P.R. routine' will have produced ZZUW which will cause re-entry to the microprogram sequence at (OD 12/J4). Assuming the segment is not out of range $\overline{\text{SOOR}}$ being set, (see OD 3/B6) the bistables ZSR and NDT will be set.

Bits 16-21 of the relative address on the J highway will be added to bits 0-5 of the segment table starting address in the PD register, the output being PDS0-5. The carry bits PDCY are used to detect whether the segment is out of range.

The signal NDT gates the complete segment table entry address (PDS0-5 and PD6-21) onto the T highway. ZSR will make the signal D:ROP (OD/.K3) to prepare the distributor for reading out the page table start address. The signals NTAAF and NTAGZ made by ZSR and $\overline{\text{ZRSR}}$ (OD 12/K5), will gate the address of the segment table entry into the AA register. This address will then be gated to the address highway by NAAH, the PREQ signal initiating the 'read out' onto the RE highway. When the RDST signal is detected, indicating the information has been read out, the signal NST (OD 12/K7) gates it onto the T highway. It is then gated to the AA register by NTAAF and NTAGZ (OD 12/J7), thus the AA register has been loaded with the page table start address. The microprogram then generates ZZSR assuming it is not a replacement address.

If the address read out is a replacement address (bits 22 and 23 not being set), then this address gives the location of the page table start address. Another store cycle is therefore required.

When the address has been read into the AA register, the signal PREP will be made (decode of bits $\overline{22}$ & $\overline{23}$) thus directing the microprogram to set ZRSR and generate ZZRS. (OD 12/J8). The 'read from page table' sequence is re-entered by ZZRS (OD12/J5) the address of the page table start address being in AA from the previous read cycle. The microprogram then initiates the read cycle as before, loading the required address into AA with NTAAF and NTAGZ (OD 12/J7). This address, from the second read cycle, is then interrogated to determine if it also is a replacement address. Three conditions can then arise.

1) The address in AA is not a replacement address. In this case $\overline{\text{PREP}}$, the second read cycle will end by generating the signal ZZSR in readiness for the 'read from page table sequence'.

2) The address in AA is a replacement address and MULREP is made. In this case ZZRS will again be made thus initiating a further read cycle using the address just loaded into AA. This cycle could be repeated until condition (1) above was attained.

3) The address in AA is a replacement address and $\overline{\text{MULREP}}$ is made. Since multiple replacements are not allowed, a failure condition arises, REPFLS being generated. This signal will cause entry to the not-equivalence sequence when the main microprogram is allowed to proceed (i.e. when TA is made), as $\overline{\text{TEQ}}$ will be set.

c) Read from Page Table (S.C.110)

The page table entry address is formed by using bits 10-15 of the relative address (the block part) as the l.s. five bits of the page address. Bits 6-21 of the page address being the page table start address stored in the AA register.

ZZSR re-enters the microprogram at OD3/H4 and if $\overline{\text{POOR}}$ and $\overline{\text{PTAB}}$ (Page Table ABsent) are set indicating that the address of the page table is not out of range and that the page table is in core store, the signals ZPR and NJPT are made. NJPT gates bits 10-15 of the relative address from the J highway to bits 0-5 of the T highway.

ZPR will make the distributor signal D:ROP (OD 19/M5) and the signal NTAAF which will gate bits 0-5 from the T highway to the AA register, (see Diagram 6). The AA register will then contain the page table entry address and the permission bits. NAAH will gate the address onto the address highway, the PREQ signal initiating the read out from the core store.

When the RDST signal shows the page address has been read out onto the RE highway, NST (OD12/K7) gates it onto the T highway. The signals NTAAF and NTAGZ (OD 12/K7) gate it into the AA register, the microprogram then generating ZZPR provided it is not a replacement address. If it is a replacement address, a sequence similar to that for 'read from segment table' is followed.

d) Load CPR (S.C. 111)

Having now loaded AA with the page address into which the block of program has been loaded, this information along with the block address must now be written to the CPR.

In the following description it is assumed that a multiprocessor system is not being used, hence $\overline{\text{PTIME}}$ and $\overline{\text{ALLOW}}$ will be made.

The sequence is entered by ZZPR making NTPP and setting NJTLZ (OD 12/N5).

NTPP gates the page address and the permission bits from AA7-21 to the C.P.R. bits PP7-21 (OD 3/K1). NJTLZ gates the relative address from the J highway, bits 10-21 onto the T highway, (see Diagram 7). The microprogram then makes the signal NTAGZ which gates the block address from the T highway to bits 10-21 of the AA register, followed by NTPB which gates it from the AA register to the C.P.R. bits PB10-21 and sets the 'in use' bit (OD 3/G2).

The signal TRRQ will cause the equivalence sequence to be re-entered (OD 12/G2) and unless a hardware fault occurs, equivalence should be found.

2.2.4 Instruction Addresses (S.C.105);

When an instruction address is gated onto the J highway from the AC register, (OD 2/J7) as it is a relative address it must be made absolute in the address translator. There are two different cases to be considered.

1) when the address does not lie within the same page as the previous instruction address, or if there is a possibility that this may happen (case 1a).

2) when the address does lie within the same page as the last instruction address.

In the first case considered above, a TREQ signal is required to initiate the translation this being produced from the instruction unit microprogram (OD 11/J6) when the signal NEWP is present. This signal will be produced by any one of three conditions.

Since a page is 1K words long, when the instruction address reaches a multiple of 1024 this is an indication that the instruction address has crossed a page boundary. The signal HCY 10 is produced from the carry bits in the control adder which in turn produces NEWP (OD 4.M5).

An instruction which was part way through the instruction unit may have to be abandoned, because the preceding instruction, which may have written to its store location, must be allowed to complete its action before the following instruction can be processed. WTOC is produced (OD 10/K4) and when the instruction is called again BKGI will be produced which will set NEWP (OD 11/J8).

When an HC 'carry on' type instruction has held up the sequence and the abandoned instruction is recalled, the AY register may not hold the correct page address and so a translation is required. The signal HCCO is produced which sets NEWP. (OD 10/F8).

When a program is entered by the 172E instruction the address from which it is required to start, which will have been loaded into the P register, must be translated. The end of the 172E instruction also therefore produces a TREQ signal. (OD 17, J,K,L, M.).

The TREQ signal will initiate the address translation (OD 12/E1) by entering the equuivalence sequence as described in section 2.2.2. Assuming equivalence is found the translated address will be on the T highway, and provided PERMIT is set showing the type of store access on this page is permitted (OD 12/F3), the sequence will be directed by RQNP to produce NTY. If PERMIT is not set, TNOP will be produced and the non-equivalence routine initiated (OD 3/C2). This signal NTY gates bits 6-21 from the T highway to the AY register, (only bits 10-21 are required for the page address, bits 6-9 being redundant in this case), thus the page address is now in the AY register and will remain there until a new page is entered.

After a wait until the address highway is free, (AHF), the staticiser NYAH is set. The signal INREQ can alternatively enter the sequence at this point (OD12/B1) as explained below. NYAH gates the page part of the address (bits 10-21) from the AY register and the 'word' part of the address (bits 0-9) from the AC register, where the relative address will still be held, to form the absolute address on the address highway.

The PREQ signal will then initiate the distributor to read out the instruction. The signal KIN will cause the instruction pair to be loaded to the E and O registers (or if it is a single instruction to the O register only) when RDST is received, and after a delay of 50ns the signal RHF is set.

If EQ and PERMIT are present, the microprogram is also directed down another path (OD 12/G3) to initiate the setting of the use bits and CPR locking sequences.

If a multioperand transfer order has been obeyed, up to four CPRs may have been locked, it is therefore desirable that these be unlocked after the next instruction address translation is made. All CPRs except CPR0 are therefore unlocked after an instruction address translation. The CPR which now contains the page address for the new instruction block is locked, until a page boundary is crossed. This is necessary because of software considerations, although the hardware would be unaffected if it were not locked.

The RQNP signal directs the microprogram to set NTUN which unlocks all C.P.R.'s except C.P.R.O. RQI will then cause NTOU to be set (OD 12/G5) which will set the 'obeyed' bit in the CPR which gave equivalence (PPO2). NTLK will lock the CPR (OD3/E1) which will produce PLKx which when gated with ANVPx (OD3/F4) makes EQVP if the next vacant page counter is pointing to the C.P.R. which gave equivalence. The signal NTVP will then be made, stepping on the next vacant page counter to an unlocked C.P.R. If the C.P.R. which gave equivalence was not the one that was last loaded, the next vacant page counter will not be pointing to it, therefore EQVP will not be made as it is not necessary to step on the counter. In either case the C.P.R. which gave equivalence will be locked.

In the second case considered, as the address lies within the same page as the previous address, the page part (bits 10-21) will already be held in the AY register. It is therefore not necessary to do a full translation.

The instruction unit will not set NEWP, hence the signal INREQ will be produced (OD11/K6). This will cause entry to the address translator at (OD 12/B1) NYAH will then gate the page part of the address from the AY register and the 'word' part of the address from the AC register to form the absolute instruction address on the address highway. The sequence continues from this point as for the normal translation described earlier.

2.2.5 Operand Address

All operand addresses must be translated, but the store access for read operands is treated separately from write operands.

a) Read operands (S.C.106)

The TREQ signal will cause entry to the equivalence routine and assuming equivalence is found the absolute operand address will be loaded onto the T highway, the signal EQ being set. If the PERMIT signal is set showing it is a page which can be read from, the sequence divides into two branches.

The main path is directed by the signal $\overline{\text{EQCK}}$ and $\overline{\text{RQNP}}$ to await for the AA register to be free (AF) and the signal $\overline{\text{WPROC}}$ to indicate a transfer from the AW register to the AA register is not underway.

The signals NTAAF and NTAGZ gate the operand address from the T highway for the AA register, the signal NTA generating the control signal A:ROP. NTAR is sent to the microprogram to indicate the operand address is in the AA register, and the AA register and the address highway are marked busy by $\overline{\text{AF}}$ and $\overline{\text{AHF}}$. The sequence is then held awaiting the signal APROC from the micro-program which requested the translation (CLA is produced from the 152,153 and 154 orders, resetting APROC if the read cycle is cancelled: see Chapter 16).

When the APROC signal is present, the A:ROP signal directs the routine to set PREQ and NAAH to load the operand address onto the address highway and initiate the distributor 'read out' sequence. The address accepted signal is passed onto the CPU and after a wait for the read highway to come free, NTRH is sent to the microprogram which sent the TREQ signal. The signal KOP is then sent to the microprogram and if it is an 023 instruction, the signal KIN is also set in order that the microprogram can distinguish between an operand and an instruction. The RDST signal indicates the operand has been read out onto the RE highway.

The other branch of the microprogram sequence, is directed by $\overline{\text{RQNI}}$ (OD 19/C3) to set NTRU which sets the 'read' use bit, PP1, in the CPR which gave equivalence (OD3/M1). The sequence continues, generating NTLK which locks the CPR, and produces PLKx (OD3/F3) which when gated with ANVPx (OD3/F4) makes EQVP if the N.V.P. pointer is pointing to the C.P.R. which gave equivalence. After a delay of 25ns, NTUK unlocks the CPR which gives $\overline{\text{EQVP}}$. After a delay of 70ns if EQVP is present, the signal NTVP will step on the next vacant page counter to point to the next C.P.R., hence making this C.P.R. the least likely to be reloaded. If the pointer is still pointing at a locked CPR it will be stepped on again; as EQVP will be set, until an unlocked CPR is found.

If the N.V.P. counter is not pointing to the CPR which gave equivalence there is no necessity to step it on hence EQVP will not be made.

b) Write Operands (S.C.107)

The operand address must be translated, loaded to the AW register and from there copied to the AA register where it can be gated to the address highway in readiness for the write cycle. The address translator however, also copies the translated address to the AA register before the AW register is loaded, this copy being redundant. The AA register is then set free (signal AF) so that it can be used by another sequence if required. This action is referred to as a 'dummy operand' sequence which utilises the logic available.

The first part of the sequence is similar to that of a read operand address. The TREQ signal causes an equivalence check to be made (see Section 2.2.2) and assuming equivalence is found the translated address is gated from the J highway to the T highway by NJTAF and NJTGK (OD3, J,K,M-5) the signal EQ being set. The sequence shifts into two paths at this point.

The other path selected RQNI sends the signal NTWU which sets the 'written to' use bit (PP0) in the C.P.R. which gave equivalence (OD3/M2). The CPR is locked by NTCK (OD3/D4) being unlocked 25ns later. The next vacant page counter is then stepped on by NTVP to point at an unlocked C.P.R. thus ensuring that the C.P.R. which gave equivalence is the least likely to be re-loaded. The details of this logic are described in the 'read operand' description above, section 2.2.5a.

The main path is directed by $\overline{EQCK}$ and $\overline{RQNP}$ to the waiting gate AF and $\overline{APROC}$ (OD12/E4). The signals NTAAF and NTAGZ gate the operand address to the AA register. After a 25ns delay $\overline{RQJ}$ will cause TF to be set (OD 12/D5). This signal which is sent will allow the awaiting WREQ (OD4/L5) to be sent back to the address translator (OD12/A1) to complete the sequence.

Meanwhile the main branch of the sequence having waited for APROC (D5) is directed by $\overline{A:ROP}$ and A:WOP to set AF after 25ns, thus the address in the AA register can be overwritten as it serves no useful purpose. The translated address however is still on the T highway.

The WREQ signal (F4) after the wait for WF to signify that the AW register is free, causes NTW, which gates the address from the T highway to the AW register, (OD12/A1).

Provided AF is set, showing the AA register to be free, and a WPROC signal has been sent by the sequencer, NWT transfers the address in the AW register to the T highway. The signals NTAAF and NTAGZ then gate the address from the T highway to the AA register. When the address highway is free the signal NAAH gates the address onto the address highway and sends a PREQ signal to the distributor. When the 'write' cycle is under-way, the distributor sends the signal AA which is then directed to the instruction unit microprogram. (CLRM is produced from the 150, 151 sequences, resetting WPROC if the write cycle is cancelled see Chapter 16).

Similarly if not equivalence occurs, giving NEQP during a floating point operation, the numbers being stored odd/even, CLW will be produced (OD12/A2). This will effect the address translator sequence the same way as CLRM (OD12/A2) (See Chapter 16).

2.2.6 Branch Instruction (S.C.104)

Note: See also S.C. 103/1,2,3 for first pass of replaced branch instructions of which this sequence is the second pass.

In the case of branch instruction, the branch address must be made absolute, it is then loaded into the AA register, a copy of the page address kept in the AY register for the next group of instruction. For a conditional jump instruction, the absolute address is loaded into the AA register before the decision of whether it is successful or not is made. If it is a successful jump, PASS is set, but if not, $\overline{PASS}$ is set, the store access being abandoned.

The TREQ signal from the sequencer causes entry to the equivalence check routine, when LKF is set from the previous translation (OD12/F2). Assuming equivalence is found, the translated address will be gated to the T highway by NPT, NJTAF (OD12/G3) and after a 10ns delay, TA signal is sent (OD12/C4). When the AA register is free and a write address is not about to be loaded to AA, NTAAF and NTAGZ gate the absolute address on the T highway into AA register and the signal $\overline{AF}$ generated to make the AA register not free (OD12/E5). The sequence is now awaiting the APROC signal to signify a store access can be made.

The other path of the microprogram which branched at OD12/F3 now takes the path selected by RQJ signal (OD12/G4) again having to wait for APROC.

It is at this point that the result of the test as to whether the jump is successful or not is made. The two cases, PASS and $\overline{PASS}$ are now considered separately.

If the jump instruction is not successful, $\overline{\text{PASS}}$ being set, the store access translation is not required. The signals A:RQJ and $\overline{\text{PASS}}$ will direct the sequence down the dummy operand path (OD 12/D6), setting the address highway free and the AA register free (AHF and AF). The translator is also set free (OD 12/D6). The other path of the sequence is directed by $\overline{\text{PASS}}$ to set LKF.

If the jump instruction is successful, PASS being set, the store access to continue. The signals A:RQJ, PASS, TEQ & $\overline{\text{A:ROP}}$ will set NTY (OD 3/D6) which will gate bits (6-21) from the T highway to the AY register, thus leaving a copy of the page address in the AY register for the following instructions. The signals A:RQJ, PASS & TEQ will direct the sequence to make the PREQ signal to initiate a read out of the instruction, NAAH gating the address from the AA register to the address highway. When the distributor sends the address accepted signal AA, this is sent to the CPU.

The sequence will then be directed by $\overline{\text{A:ROP}}$ to set DTAD (OD 12/C7), which loads the branch address to the Maintenance Aid Buffer, AD. The signal NTRH (D8) is sent to the instruction unit microprogram to indicate that an instruction pair will be the next information on the read highways.

The signal KIN to the instruction unit microprogram will load the instruction pair to the E and O registers (or if it is a single instruction to the O register only) via the distributor. The signal RDST will be sent to the address translator, which in turn will send it to the C.P.U. indicating the instruction is about to be read out.

The other path of the sequence which was waiting for APROC will be directed by PASS to follow the procedure as for an instruction address. NTUN will unlock all the CPR's except C.P.R.0; NTOU will set the obeyed bit in the C.P.R. which gave equivalence, NTLK then locking the C.P.R. If this CPR has the next vacant page counter pointing to it, the pointer must be moved on to the next unlocked CPR. EQVP will produce NTVP which will step the counter on to an unlocked CPR.

2.2.7 Replaced Branch Instructions (S.C.103)

With a replaced branch instruction, n is the address of the core store location which contains the branch address. In order to read out the required instruction, two passes through the address translator are required, the first to translate the address of the branch address and the second to translate the branch address.

On entry to the instruction unit at signal TREQ (OD 12/E1), the relative address has been gated from the P highway to the J highway and an equivalence check is carried out (Section 2.2.2).

Assuming equivalence is found, the translated absolute address is gated to the T highway by NPT, NJTAF (OD3/D2). When the AA register is free and a write address is not about to be loaded from AW to AA, the signals NTAAF and NTAGZ gate the translated address to the AA register, $\overline{AF}$ then being generated to mark the AA register not free. The sequence is then awaiting the APROC signal to signify a store access can be made.

The other path of the microprogram which branched at D3 now takes the path selected by the RQJ signal (E3). On receiving this signal, assuming it is a successful branch, NTUN unlocks all the CPR's except CPR0 and as RQRD is present, NTRU marks the 'READ' use bit of the CPR which gave equivalence and although NTLK locks this CPR after a delay of 25ns, it is unlocked by the signal NTUK, as this is not the final required page address.

If the next vacant page counter is pointing to the CPR which gave equivalence, the PLKx signal gated with ANVPx (OD2.J4) will produce EQVP which when gated with NLKK delayed by 70ns will produce NTVP to step on the next vacant page counter. The CPR which gave equivalence is then the last to be reloaded.

If the next vacant page counter is not pointing to the CPR which gave equivalence the counter will not be stepped on as ANVPx will not apply to the same CPR as the PLK signal (OD3/F3).

After the wait for the APROC signal (OD12/D5), the decision will have been made whether or not the branch is successful and if it is so, PASS will be set. If it is not set, the branch test being unsuccessful, the translation is abandoned, the microprogram being directed by A:RQJ and $\overline{PASS}$ (OD12/D6) to set the address highway and the AA register free.

In the case of a successful branch instruction, the signals A:RQJ, PASS and TEQ direct the microprogram to send the PREQ signal to the distributor to initiate the 'read' cycle. NAAH gates the address from the AA register to the address highway The signal A:ROP directs the sequence set $\overline{NAAH}$ and AHF (OD12/E7).

The signals KOP and KIN are sent to the instruction unit (OD14/D1) which will initiate the microprogram to gate the branch address into the fixed point buffer. This address is then pre-modified if necessary and loaded into the P register, from which it is gated onto the T highway. Another TREQ signal is made at this point to cause entry to the address translator for the second pass. The second pass is exactly the same as for a normal branch instruction. PASS always being set (see Section 2.2.6) When the branch address has been translated, the required instruction pair is loaded to the E and O registers, (or if it is a single instruction, to the O register only) a record of the page address being left in the AY register.

2.3 Translator By-pass (S.C.101)

The by-pass directs the translator microprogram to load the number from the J highway to the T highway without translating it (OD 12/E2). It is used when the information on the J highway is a 'Literal' or in some cases as listed in Section 2.3.2 below, when it is an address less than 8 indicating an X register access is required.

2.3.1 Literals

When the TREQ signal is made, the signal ABS will be present (OD 12/F2) which will direct the sequence down the by-pass. The signals NJTAF, NJTGK and NJTLZ gate the contents of the J highway (bits 0-21) to the T highway. If the AA register is free and a write address is not about to be loaded into it (AF and $\overline{\text{WPROC}}$) the signals NTAAF and NTAGZ load the literal to the AA register. After the APROC signal is received, the sequence is directed down the dummy operand path by $\overline{\text{A:WOP}}$ & $\overline{\text{A:ROP}}$ & $\overline{\text{A:INKP}}$, the A register being set free after a delay of 25ns providing the F register is free.

2.3.2 X Register Access

The action of the translator when the address to be translated is less than 8 is dependent upon the origin of the address. The sequence of the translator can be divided into; a) via the normal translation sequence, b) via the by-pass.

The path via the normal translation sequence is used for:

1) HC type instructions excluding the 116, 126 and 127.

2) During the interrupt sequences when the access to words 0-7 of the program is required.

3) Branch and replaced branch instructions.

4) Instruction addresses excluding the 023 instruction.

In all these cases the X register access is made via the distributor, the path through the address translator being the same as for addresses >7.

The path via the by-pass is used for:

1) Reading an operand from an X register; after the translator by-pass, the sequence is the same as for an address >7 with the exception of the fixed point operand. In this case the signal A:XOP will be set, and gated with A:ROP (OD 12/D6) hence the routine will be directed down the dummy operand path after the APROC signal is received. Access to the X registers is via the micro-program and not the distributor for the fixed point operand.

2) Writing to an X register: after the by-pass, the sequence is the same as for writing to an address greater than 7, access to the X register being made via the distributor.

3) During the interrupt sequences when access to the hardware X registers is required, the by-pass is used, the remainder of the sequence being as for a normal 'read operand'. Access to the register is made via the distributor.

4) The 116, 126 and 127 multi-operand transfer instructions. If the addresses are less than 8, when the first and last addresses of the origin and destination are checked, after the by-pass, the signal EQCK directs the sequence to end by setting translator free TF (OD 12/D4).

For the subsequent read and write accesses, the by-pass is used, but after this the sequence continues as for read and write routines with addresses >7, access to the X registers being made via the distributor.

When the TREQ signal is received the signal HDX, when present, will direct the sequence through the by-pass. As for the 'literals', the address will be gated from the J highway to the T highway without translating it. From there on, the sequence will depend upon the origin of the address on the J highway as described above.

2.4 Datum and Limit

In order to make the relative address in the program into an absolute address, the datum for the program must be added. A check is also made that the program size is such that it will be within the set limit. The program is considered in blocks of 64 words, datumising only being necessary when a new block is entered, or when there is a possibility that this may happen. Bits 6-21 of the address give the block address, bits 0-5 indicating the word within the block. It is therefore only necessary to datumise the block part of the address, bits 6-21.

The conditions which give rise to a datumising sequence being required, are the same as those which require an equivalence check in the paging system, with the exception of the instruction address crossing a page boundary. In this case, a datumising sequence is required every 64 words instead of every 1K words the signal HCY6 producing the signal NEWP (OD 11/J8) when the address in the AC register reaches a multiple of 64. The machine is switched to the datum and limit mode by a manually operated switch which sets the signal DLHK.

When a program is running in the datum and limit mode, the datum will have been loaded to DT6-21 (OD 3/C3) by the 172E instruction. Bits 6-21 of the relative address are added to the datum in the datum adder (OD 3/C2) to give the datumised address DT6-21. The TREQ signal will direct the micro-program to be routed by DLHK to set NDDT and NJTAF. These signals gate the datumised address onto the T highway, bit 6-21 from the datum adder, and bits 0-5 from the J highway. From this point on, the sequence proceeds in the same way as for the paging mode of operation, as described in Sections 2.2.4 to 2.2.7.

PREFACE

This memorandum describes a programming system which has been simulated on the ICL 1900 computer, and realized on the experimental Basic Machine at Stevenage. It is the first system to make use of a comprehensive set of data tags, recognised by hardware. The present system design was initiated in 1965, and has been the basis of experimental programmes which have lead to alterations in hardware and system characteristics. As a general rule, however, changes which would invalidate existing programs, or whose effect is predictable, have not been implemented. The existing system is described in sufficient detail in the following pages to allow complete programs to be prepared for running and debugging. Supplementary information will be found in the IBS Operating Manual.

The two previous editions of BLMM 18 are outdated by this memorandum. The work of compiling and editing has been carried out by Miss D.I.S. Meinhart, with contributions from A.L.G. Flanagan, J.K. Iliffe, A.W. Shilling, Mrs. J. Travis, and J.J.L. Williams. We would like to acknowledge the cooperation of users in S.A.P.O. and Imperial College in pointing out defects, and tolerating the ambiguities of earlier efforts.

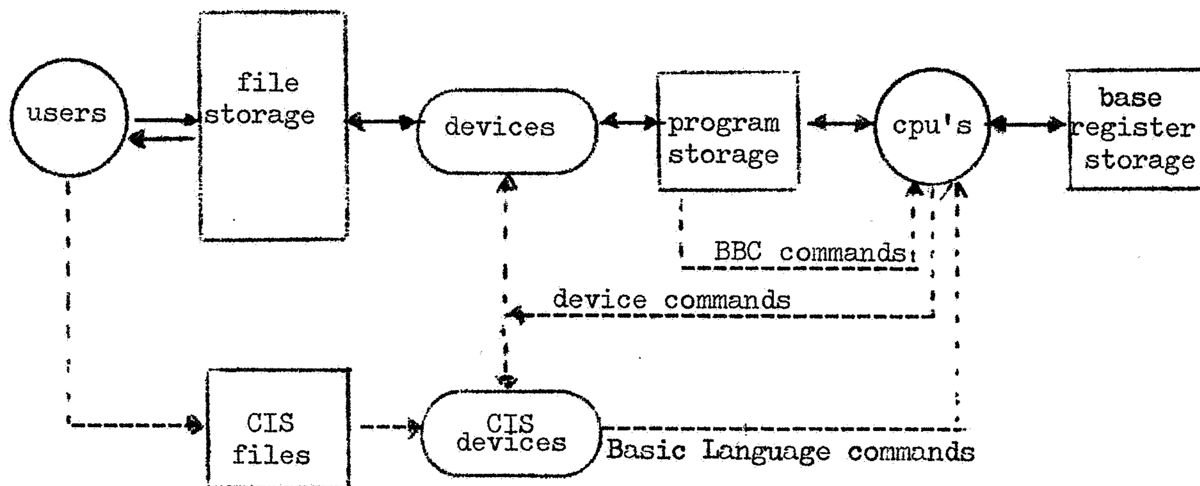
Stevenage, Herts.
31 August 1969.

1. Machine and Data Structures

A Basic Machine consists of an information store and a number of processors. The store can be subdivided into areas of program, base and file storage, which means in practice that the three are addressed in different ways and deliver different units of information on being accessed. A processor is either a central processing unit (cpu), which controls the transfer of information between program and register storage, or a (peripheral) device, which controls transfers between program and file storage.

Information transfers are usually accompanied by a change in the pattern of stored information. In the case of a device, this change must be logically simple, e.g., a parity bit calculation or conversion of a bit pattern to printed character form, but in a cpu more elaborate operations can be carried out, including a range of arithmetic and logical functions of binary quantities, taken either from base or from program storage. A cpu is controlled by a sequence of instructions taken from the program store in the form of binary basic code (BBC) commands, according to rules similar to those for any conventional single sequence calculator. The devices are controlled by device commands issued by cpu's as a result of obeying certain BBC instructions.

The broad picture of the basic system is completed by the presence of a number of users, who have the ability to communicate with parts of file storage, and to control cpu activity by giving commands in the form of Basic Language instructions which are obeyed directly. The whole system, neglecting the multiplicity of persons and physical components, can therefore be sketched as follows:



It can be seen that Basic Language commands in the form of CIS (command input stream) files control the information paths between base and program storage, and indirectly control the paths between program and file storage. Binary basic code can only be generated by putting BL commands into a file and having them translated into binary form, so that all system activity stems from sequences of BL commands. Of course, the origin of a command sequence may be in the text of a program written in another language, which has been through several stages of translation, but all commands must eventually enter the system in the format of Basic Language, and the system can only be fully exploited by issuing commands directly in BL.

It does not follow, however, that every capability of the system is available to the users. It is normally the case that the utility of an installation or group of installations is decreased if each user is allowed to specify exactly where his information should be held in physical storage, or exactly which processors should obey his commands, or exactly when a particular operation should be performed. One of the requirements of an input language is that it should insulate the users as much as possible from changes in the physical components of the system. The main features of Basic Language which make such concealment possible are the symbolic basis of the addressing system, which insulates the user from the properties of different storage devices, and the definition of a process as the act of obeying a sequence of BL commands, which enables not only the configuration of devices and cpu's, but also the exact pattern of binary basic code itself, to be concealed from users. The same principle has been applied wherever possible in the system design, and by a combination of linguistic and hardware devices a high degree of independence of machine coding has been achieved.

The "base" referred to above includes a number of hardware "registers". The register store itself stands squarely on the boundary between exposed and concealed parts of the system. To a user who is primarily interested in file manipulation, the existence of program and register storage is almost irrelevant, except when he wants to describe new procedures. However, with a suitably elaborated language all procedures can be described by operations on the program store, with no explicit reference to registers, and consequently the range of application of the language is widened. In the Basic Machine system it is intended to provide two versions of the input language. In Pure Basic Language the structure of the register store is entirely concealed from the user; in Applied Basic Language a set of registers is made explicit, and the instruction code is oriented towards using the registers in the most effective way. A single compiling program is sufficient to translate programs written in either version of BL into binary basic code.

The purpose of the present document is to describe the grammar and use of Applied Basic Language. In Section 2 the main components of the language are introduced, constants are defined, and the method of table searching which leads to the value of variables is given. The next three sections deal with the three classes of instructions, i.e:

- machine instructions, which are directly obeyed by the cpu;
- system instructions, which are obeyed by calling subroutines permanently held in program storage;
- and assembly directives, which are used to control the process of translating BL commands into BBC.

Section 7 contains an example of a program, illustrating some of the material in Section 2.5. The initial file handling system is described in Section 8. The final Section consists of various tables to which reference is made in the text.

1.1 Storage

The Basic Machine store consists of a core (16K words of 32 bits), a collection of magnetic and paper tapes (some of which are recorded in ICL 1900-compatible mode), and a number of electromagnetic delay line stores. The last named meet the requirement for register storage in Applied Basic Language; program storage occupies most of the core and one or more magnetic tapes; the file store resides on the remaining magnetic tapes and paper tapes, to which must be added line printer output and teleprinter input and output, since the definition of the system embraces all material exchanged with users. The registers are built to meet the specification given below; the program store is considerably different from the core and tape onto which it is mapped, so that special hardware has to be used to maintain the conceptual store structure; the logical organisation of the file store is also markedly different from its physical realization, but since access to the file is handled by interpretive routines the techniques used on the Basic Machine are conventional.

1.1.1 Register Storage

Register storage consists of a number of registers designated "XO", "X1",,,, etc.* Associated with each register is a variable tag describing the register content, the main tag codes being (in hexadecimal form, see Section 2.1) :

tag 0 Binary data
1 Null register
#A Address

The tag value is set when the register is loaded. It is interrogated by hardware in the course of function execution.

Binary data (tag 0) consist of 32-bit words. Throughout the machine description, bits are numbered from zero at the least significant, and a subfield extending from position p to p+q is written

as "bit p(q)". The register bits are therefore numbered from zero to 31. The tag does not form part of the directly accessible information. A conventional set of functions of binary data is available, arithmetic functions treating the register as an integer in two's complement representation, with the sign in bit 31.

A null register (tag 1) is strictly meaningless as an operand, and causes an error to be signalled by most machine functions. The tag can be set to 1 and tested by program, and may be used to lead into special system or user-defined actions.

---

* In the experimental machine there are eight registers.

An address (tag A) points directly to an information unit in core store which is the first member of a sequence of similar elements. The address contains a coded description (D) of the elements and a limit (L) indicating how many more there are, as well as the location number (C) of the first one:

An address cannot be manipulated as binary data by a user, so it is unnecessary to specify the exact format of the register. C is a location number sufficient to cover the actual core store; L is defined to be a 15-bit field, so that the maximum addressable sequence has 32,768 elements; D describes both the arithmetic category of the elements and the way the sequence is represented in core, as discussed in subsection 1.1.2.1. Note that if L=0, the sequence degenerates into a single element, and the address is said to be singular.

The above are the three main categories of register information, each register being able to hold information from any category. Further refinements of data descriptions are made in the definition of machine functions (Section 3).

1.1.2 Program Storage

The program store is conceived as a collection of ordered, finite sets of information, the length of each set being chosen by the programmer to suit the information it contains. Associated with each set is a unique descriptive word known as a codeword, which is formed and stored by a system routine as the result of a store request made by a programmer. Besides giving the length of the set, the codeword indicates its physical starting position and the sort of element it contains.

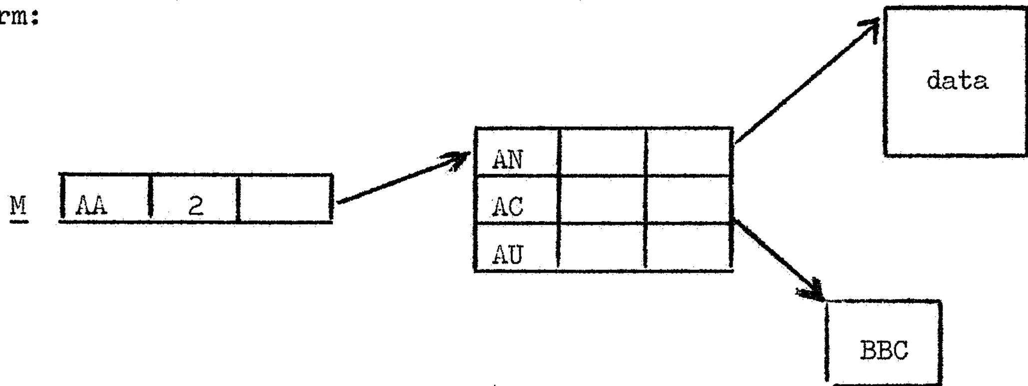
In general the elements of a set are of the same sort, characterised by a numerical coding of type (integer, instruction, codeword, etc.) and size (8,16,32 or 64 bits). It is also possible to describe sets of mixed elements, and in Applied Basic Language one such set, known as the register dump, is associated with each active process in the machine.

By grouping the codewords themselves into sets, and arranging that their codewords appear in other sets, it is readily seen that the program store can assume the structure of a "tree", provided certain conventions are adopted to prevent circularity: such conventions are observed by the store control routines. The program store is thus presented as a hierarchy of information, with codeword sets at the intermediate levels, and instructions or data occurring at the "lowest" level of the hierarchy. The "top" level consists of a set of codewords in fixed locations, describing both permanent and variable system information. One level down from this is a set of codewords describing the process bases, one base being associated with each active process in the machine. The user is given free access only to his own base (by using global names, see p. 17) and to the lower level information which it describes. The machine and system functions do not allow one to progress "up" the hierarchy and so from one base to another: in this way program protection is achieved. There are, however, certain occasions when controlled access to other bases, including the permanent system base, is allowed.

Codewords are classified according to the sort of information they describe, and the way it is represented in physical storage (see p.7). Table 2 lists the classes and the abbreviations used for them in this manual. In general, those system functions which operate on sets of information are strongly dependent on codeword class, while the machine functions, operating on single elements, are less so.

Codewords of classes AN, AC, AA and AR are similar in format to an address, the descriptive part being used to give the codeword class. A codeword or address M referring to a sequence of 15 data elements can be drawn diagrammatically as:

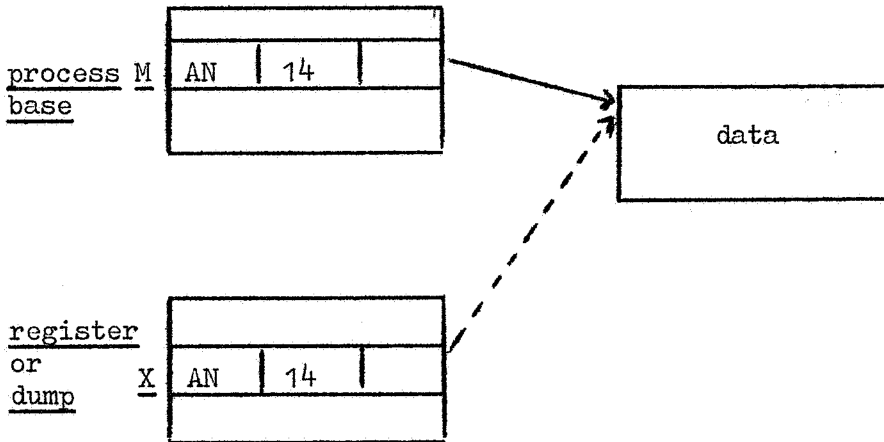
There is no need to write in the location field, since the arrow serves the purpose. A small "tree" consisting of a set of three codewords, one referring to data, another to instructions, and the third undefined, has the form:



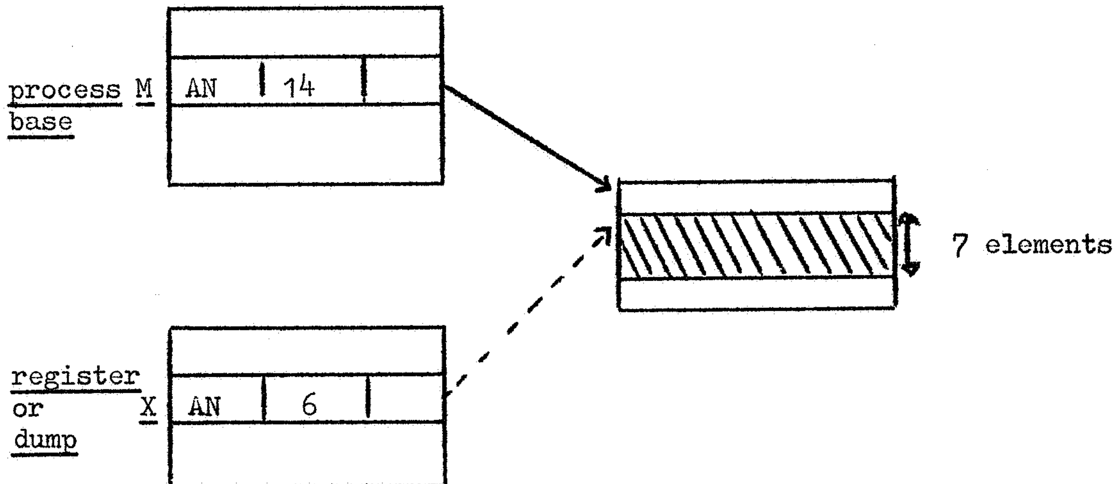
A typical process base consists of from 20 to 100 such structures.

In the above diagrams, it can be seen that the descriptive part of the codewords applies to each element of the sequence they refer to. Such sequences are said to be homogeneous. If the codeword is of class XD, however, each element of the sequence carries its own descriptor, in the form of a tag. Strictly speaking the process base is mixed, since it includes the registers, but since all the other elements are thought of as codewords the registers are given a special mixed sequence, into which they can be stacked or "dumped" at any time to make the registers available for some other purpose. Just one such register dump is associated with each process; it can be accessed either as a nesting store, or by constructing addresses pointing to sequences of elements. No other use is made of class XD sequences in Applied Basic Language.

It follows that any address formed by a process must be held either in register storage or the dump. An address pointing to part of the program store can be formed by loading a register from a codeword, thus producing another pointer to the same data sequence, as in the following diagram:



By operating on the address, the register can be made to point to any subsequence of the original data:



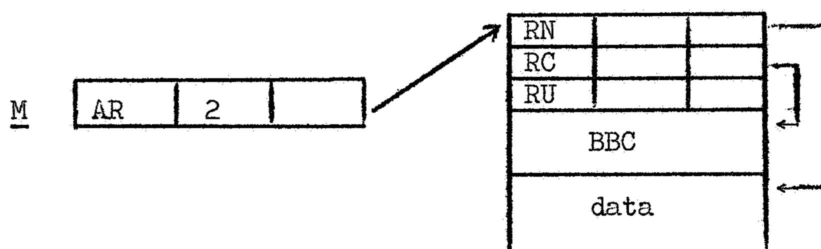
By copying the address into another register and operating on that, addresses referring to overlapping sequences can be constructed, and frequently are. We are thus led to an important distinction between addresses and codewords - that the latter are held in program storage and are responsible for defining the information structure, whereas addresses are formed transiently in registers (and dump) to point at the information to be processed.

It is usual in Basic Language to refer to the sequence defined by a codeword as a set. The operation of loading a register from a codeword is known as codeword evaluation. If the codeword is of class AN, AC or AA evaluation will lead to an address coinciding with the set; class XD cannot arise from BL programs; class SB leads by address substitution to another codeword for which the evaluation is repeated; the remaining classes lead to interpretive actions, as described in later sections.

1.1.2.1 Representation

The position of a set in core store has no relevance to Basic Language instructions, which work exclusively on the structures defined by the process base. If, for purpose of store management, the core assignment has to be reorganised by moving sets within core, or from one storage level to another, all codewords and addresses are updated by the system. The way in which the program store is mapped into core is called its representation. The core store is partitioned into a number of active blocks, holding useful information, between which there may be inactive blocks, which can be made available for use when requested. There are several ways in which structures can be represented, the choice between them depending on how they are used, and particularly on the way the structure changes during a calculation.

In the absolute representation, the location number contained in a codeword can refer to a set anywhere in the core or secondary store system. In the relative representation, the first element of the defined set must be a fixed distance from the codeword. Hence the set cannot be moved without moving the codeword itself; they are bound together in the same block, along with other relatively represented sets. Corresponding to the codeword classes already given, which contain absolute codewords, are classes of relative codewords referring to data, instructions or relative codewords (see Table 2). Diagrammatically, the small tree defined earlier by three absolute codewords can be given in relative form as follows:



The importance of relative sets is that the store management system can ignore their substructure, and deal only with the absolute codeword (M) defining the block in which they appear. The block is constructed, with all appropriate internal cross-references, by the assembly program. The disadvantage of relative sets is that their structure cannot be changed without reassembly. From the machine function point of view, there is no distinction between the two representations.

In both forms, set elements are stored in consecutive core locations (ascending from the first to last element), packed according to element size.

1.1.3 Logical Structure of File Storage

A data file consists of a sequence of data sets which, in program storage, would each be described by a class AN codeword. A structured file consists of a sequence of entities which, in program storage, would be described by any absolute codeword.

The file store is conceived logically as a collection of data files and structured files, to which any user has access in theory, though by convention the construction of file names limits each user or group of users to particular areas of the file store. The file is addressed by giving a file name and a codeword, requesting (by means of a system function) that the codeword be "connected" to the file. If the file can be found, and a suitable peripheral device is available, the device is reserved and the codeword classed as FC. Thereafter, file elements can be selected by "positioning" the device codeword and requesting transfers to or from appropriate elements of the program store. The actual methods of positioning and recording are discussed in Section 8.

Like the program store, the file can be distributed over core, drum, disc, tape, or any suitable medium. The main logical difference between the stores is that program is addressed directly, and the addresses are manipulated by machine functions, whereas a file can only be accessed through a FC codeword, which allows access to one record at a time.

1.2 Control Conventions

The cpu executes Basic Language commands which have been translated into binary basic code. Since there may be more active processes in the system than there are cpu's, the active processes are queued until a cpu is available. In the experimental Basic Machine, a process will normally run to completion, or until held up waiting for service from another part of the system, or until a process placed higher in a priority list joins the queue. To switch from one process to another, the locations holding the addresses of the process base and dump must be reloaded, and the new register contents exchanged for the old.

The experimental binary basic code is a two-address machine code having two formats for instructions. In the first, two registers X, Y and a function F are specified. F is applied to arguments defined by X and Y, the result possibly overwriting the X argument or being stored in XO. In the second format, an integer N is given in place of Y. The first format requires 16 bits, and the second uses 16 or 32, including a field for N. Since the user cannot operate on BBC, details of subfields and the several special formats encountered are not relevant to Basic Language.

A Basic Language command translates into one or more BBC instructions. A sequence of BL commands translates into a sequence of BBC, described by a codeword of class AC or RC. Relative jumps within a sequence are described in BL by jumps to labelled instruction, or by relative line counts, the correct relative jump being computed by the assembly program. Similarly, references to locally stored constants or workspace are made by name in BL and assembled into relative half-word counts. Jumps are also made by addressing a codeword starting from an item in the process base, or from a local label.

The BBC control register is a singular address pointing to the current instruction. No limit value is required since in this context it is meaningless: progression from one instruction to the next is sequential except as determined by the jump instructions. The control register includes the arithmetic condition codes (2 bits) and control mode indicators (3 bits). Condition codes are set by arithmetic and certain other functions. Mode indicators are set as a result of obeying a jump through a codeword. Their use is explained more fully in Section 3.1. The control register may be copied into any register to form a link address, to be used as a jump destination, but since the user is unaware of detailed BBC structure a modified jump is not allowed. Provision is made in BL for constructing sets of labelled jump destinations, which meet the requirement for computed jumps.

During assembly, labels may be stored with the binary code in such a way that, if the code is obeyed in trace mode, the labels of executed instructions will be output, together with other pertinent information, as a monitoring aid. Tracing is therefore conditional on the mode of assembly, which can be changed by re-assembling a segment of code, and on the control mode, which can be varied during execution.

A group of 32 bits, called process status bits, is associated with each process. Their states are referred to as zero (or off) and non-zero (or on). Their main use is in communicating between system and user programs, and they are assigned special significance in this context, e.g., to indicate various monitoring states or to simulate a "hard overflow" condition. The list of current assignments appears as Table 6. A special function (PSB) is used to test and set them.

Executable BL functions are classed either as machine functions, which are assembled as open sequences of BBC instructions, system functions, which are obeyed by executing jumps to system routines permanently held in the program store. The arguments and link of system functions are placed in register storage. The description of each function includes the argument values and tags for which it is defined: if the given arguments do not satisfy the requirements, an error indication is given. Normally this involves issuing a diagnostic report and suspending the program awaiting operator action, but it is also possible for the user to specify error actions and arrange for a process to be restarted without operator intervention. The currently defined error codes and messages are listed in Table 7. The provisions for error monitoring are described in Sections 6 and 8.3.2.

The normal sequencing of a process may be interrupted at any time by a stream of BL commands. Before being interrupted, the process is said to be at normal level of control; afterwards, it is at command level and the source of commands is called the command input stream (CIS) file. At command level, further interruptions are usually queued until the current CIS terminates, although action can be taken in an emergency to regain control of a process. All BL assembly is carried out at command level: the ability to assemble two program segments at different levels of the same process leads to some undesirable complications in language definition. After CIS processing has ended, normal control will be resumed where it left off, unless CIS has specifically asked for a restart to be made from a new entry point. When scanning the cpu queue, the highest priority process at command level is served before any normal level process. The mechanism of interrupting a process is explained in Section 4. At command level, the text can be interpreted in two different ways, as explained in Section 2.3.

The act of initialising the operating system is an operator-controlled procedure which causes the system process to be started, idling at normal level. By means of a CIS interrupt, any process can be requested to start a subordinate parallel process, with a new base and dump. The new process is placed lower than the current one in cpu priority, but in other respects the subordinate process has the same rights to system resources as its progenitor. If a process P is subordinate to Q, then P is given access to part of Q's process base, but the converse is not true. Since all processes are subordinate to the system, they all have access to part of the system base. On a private scale, sharing a data pool can be achieved by starting a common process Q, which in turn starts subordinate processes P1, P2, ... etc., each having access to its own base, to Q's and to the system's. However, the convention has been adopted that although reading and writing of elements in another base is allowed, it is not permissible to effect structural changes except in one's own base, so that the structure of the system store, for example, cannot be changed by one of the user's processes.

1.3 Using the Basic Machine

The effect of the system software is to present the machine to users as a number of parallel processes sharing a common file store, with a restricted form of information-sharing in the program store itself. In the experimental machine both the file and process naming systems are insecure, relying for their effectiveness on locally agreed conventions, though stricter controls could easily be applied.

The first task of the user is to determine the input, output, and working file requirements of his problem. This includes the preparation of paper tape data files for input under his own control or that of the Basic Language assembler, and the definition of output files. Part of the input may also come from previously computed files, library tapes, program dumps, etc.

The next task is to select the number of processes required, and the program structure appropriate to each stage of each process. For most purposes one process is adequate, but it may be possible to make better use of peripheral devices, or to simplify programming, by starting two or more.

New pieces of program are written as segments of Basic Language text, to be translated into binary blocks. The choice of blocks should be guided by the logical structure of a program, since blocks are easily changed when correcting errors or making extensions. From the system viewpoint, blocks of a few hundred words are preferred. The assembly program provides a listing of the input text, together with error messages and a summary of store assignments within a successfully assembled block, for use in diagnosing faults. Binary basic code is not listed. The assembled block is retained in the program store, ready for use without further alteration, or for filing.

The design of a process must provide for its eventual death from natural or other causes, which in turn leads to consideration of methods of process monitoring. In the most direct instance, a process is introduced by giving its name, and time and space requirements; then the first few sets of data and instructions are defined by CIS.

The process is started by specifying an entry point and terminating CIS. Program faults giving rise to illegal argument tags, or illegal addressing operations, or excess of space or time demands, produce diagnostic output (in terms of the program store structure) and then abolish the process. Similarly for irremediable faults in transfer operations.

An alternative method of use is to embed the process in a controlling program which attempts to take automatic action in response to certain failure conditions. Such a program may be specially designed for one process, or it may have general application, for example to the class of programs written to the conventions of a particular programming language. Any failure the control program can't handle may be resolved by operator or on-line user action. The mechanism of control is detailed in Section 4.2.

Finally, having prepared data, instructions, and operating notes, it remains to the user to gain admission to the system. The capacity of the experimental machine is governed by storage space, device, and process handling ability. The user or operator must ensure that the necessary devices are, or soon will be free: otherwise the process may start and then be held up until file connections are made. He will then attempt to start the new process, stating the space requirement: if this cannot be met, or if the system cannot accept a new process, the user is rejected, and he must try again later.

It will be apparent that the basic system follows conventional practice in many particulars, but there are certain fundamental divergences whose effect pervades the system. Amongst them are the following:

(1) Basic Language takes the place of three "languages" which appear in the definition of a conventional system, i.e., the binary machine code, the symbolic assembly language, and the operating commands, by making the first irrelevant and unifying the concepts of the other two.

(2) The instruction and address formats are concealed from the user, who knows these entities by what they "do" and not by what they "are".

(3) Program storage takes the form of a group of trees (whose detailed structure can be varied continuously under program control), instead of a single block of stored words.

(4) The assembly program is permanently available. Program segments can be assembled at any time and incorporated into programs without the use of loading or consolidating routines.

(5) Function interpretation is dependent on argument tag values, and on data descriptions appearing in addresses, as well as explicit function codes taken from the instruction sequence.

This programming manual should preferably not be read before giving some thought to what these interrelated assumptions must entail. Before attempting to run a program on the experimental BLM, reference should be made to the current Operating Manual.

3. Machine Instructions

A machine instruction consists of a machine function identifier (MF), and one or more arguments. The machine function identifier may be followed by an operator which selects one of three possible variants of the function. The arguments are either register names, constants, labels, or compound names; in certain cases a list of arguments is given, separated by commas.

The syntax of machine instructions makes use of the following abbreviations:

X	for any register name
N	for <integer>
S	for <signed integer>
E	for <numeral>
G	for <compound name>
XN	for X or N
XS	for X or S
XP	for a list of register names
SID	for S or <identifier>
GES	for G or E or S
XID	for X or <identifier>

Function Variants

The general format of a machine function with two arguments is :

X MF XS

the X term is the first argument, the XS term is the second argument. In general, the function identified by MF is applied to operands defined by the two arguments, the result replacing the first operand. Certain functions have a three-address variant in which the result is stored in XO; it is written:

X MF' XS

i.e., the function identifier is followed by a prime. Certain functions have a test-only variant which does not store the result, but only sets the condition codes (CC); it is written:

X MF* XS

i.e., the function identifier is followed by an asterisk.

The machine functions fall into four groups:

arithmetic and logical functions (Section 3.2);
addressing functions (Section 3.3);
jump functions (Section 3.4);
tag-independent functions (Section 3.5).

Section 3.1 describes the interpretation of machine instructions and gives details of data representation and handling within the machine.

Sections 3.2 to 3.5 have a common layout:

(i) an alphabetic list of functions described in the section;
(ii) general remarks applicable to all or most functions in the section;
(iii) detailed descriptions of the functions;
(iv) examples.

Coding examples are in Section 3.6 and exercises in Section 3.7.

Table 3 gives a summary of machine functions.

3.1 Interpretation of Machine Instructions

The interpretation of machine instructions is given in terms of the argument values, register contents, and control state at the time of execution. The following headings summarise the meaning given to special groups of digits in addresses, and introduce notations for describing states of interest to programmers. The next two paragraphs refer specifically to the control register: this is a special hardware register used in the sequencing of instructions. (Instructions are normally obeyed in the sequence in which they appear in the program text, except as directed by jump functions.) It contains information about current control mode and condition code states. (See Section 1.2 for control conventions.)

3.1.1 Condition Codes

Condition codes are indicators stored as part of the control register, and made available to the user for testing. They are reset by the result of each arithmetic function to one of four states:

zero result
negative result
positive result
overflow from result

Singly or in combination, these states correspond to the standard arithmetic conditions, which the user can test with the following mnemonic codes:

mnemonic code (CC)	last result
ZE	equal to zero
NZ	not equal to zero
LT	negative
GT	positive
LE	zero or negative
GE	zero or positive
ØV	caused overflow
NV	no overflow

Any of the eight mnemonic codes can be used with a conditional jump (Section 3.4). Because the condition codes are held in the control register, they are stored when a link is set, and take their previous value when the link address is given as destination. When obeying a jump through a codeword the condition codes are set to zero.

3.1.2 Control Modes

There are three monitoring modes, the first two of which cause the control sequence to be broken when certain conditions arise. They are:

Mode 1 monitor on integer overflow;
2 monitor on labels;
3 inhibit warnings from devices.

Each mode is independent and more than one mode can be "on" at the same time.

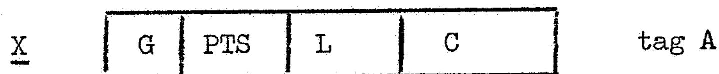
If an integer overflow occurs in mode 1, the result is not stored, and control is transferred to a system monitoring routine. The standard action is to signal error (see Section 6) and await operator action; the alternative action is to turn on a process status bit (see Table 6), then return to the interrupted sequence. For floating point overflow the actions are similar.

If a labelled instruction is encountered in mode 2, and if it has been assembled in trace mode (see Section 5), control is transferred to a trace routine, which generates one or more lines of diagnostic information on the current monitoring file (see Section 6). Normal sequencing is then resumed.

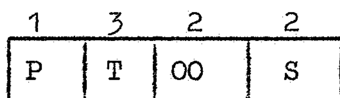
The warning-free mode (3) can be used only by system programmers. It is required for that part of the system dealing with warnings from peripheral devices.

When a jump through a codeword is obeyed, the current control modes ore "or-ed" with modes specified in the codeword. The consequence is that if a control mode in "on", it remains "on" through all codeword-directed jumps; if it is "off", it remains "off" until a jump is taken through a codeword with the mode bits "on". The control mode can therefore be influenced indirectly by setting the bits in codewords (using the MODE process control function of Section 4.2); it can be influenced directly by operating on the control register (using the MON machine function), or by jumping to a link address.

3.1.3 Protection, Type and Size Codes



The descriptive part D of an address or codeword (see Section 1.1.1) is divided into two fields. The first, G, makes the primary distinction between codeword classes described in Section 1 (and Table 2). The second is an 8-bit group, PTS, which repeats some of the discrimination of G, but makes a finer resolution of data types affecting the interpretation of machine functions. The PTS field has the following format:



P is the protection bit; T is a group of three type bits; the next two bits are not used by machine functions; S is a group of two size bits. The recognized type and size codes describe stored elements in the following way:

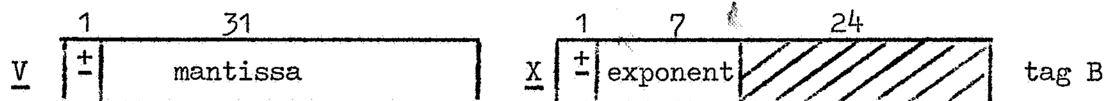
Type	Size	Description
0	0	binary data, 8-bit byte
0	1	binary data, 16-bit half-word
0	2	binary data, 32-bit word
0	3	binary data, 64-bit double word*
1	1	BBC instruction
2	2	codeword
3	3	floating point number, 64-bit
6	2	tagged (dump) element

Note that with the above coding, size is only relevant to data of type 0.

The protection bit, when "on", can be used to prevent writing to any data element (types C and 3).

Machine functions depend primarily on the type of information referred to by an address. A register containing the address of information of type t is said to have "tag A(t)". Thus a control link has tag A(1); a reference to a floating point sequence has tag A(3); reference to codewords is by a tag A(2) register; a dump address has tag A(6); etc. The control register describes information of type 1, size 1.

Machine functions discriminate between integer and floating point data representations. In register storage, integers have tag 0 (see Section 1.1.1); floating point numbers have tag B and are stored in a register pair as follows:



(V-registers are described in Section 3.1.6, floating point numbers in Sections 2.2 and 3.2.)

In program storage, a floating point number occupies a pair of consecutive words, the mantissa being in the first word, the exponent in the second. It is not normally possible to manipulate the components of a floating point number in a register as integers, but by using the PTS addressing function a floating point sequence can be described as an integer sequence, giving access to the separate components.

---

* Only the l.s. 32 bits are processed in the arithmetic and LD functions.

3.1.4 The AutoFetch (A/F) Convention

The arithmetic, and certain addressing functions, require their arguments to specify one or two numerical values. An argument may be a constant, in which case the value is implicit, or a register. The register may hold data (tag 0 or B), be null (tag 1), or contain an address of data (tag A(0) or A(3)). In the last case, the operand retrieved from store is the first in the sequence referred to. Tag 1 will usually, tags A(1) and A(2) always, cause an error action. Binary data of less than word size are automatically adjusted to 32 bits by sign extension to the left. This method of argument interpretation is known as the AutoFetch, or A/F, convention. It means, for example, that an instruction such as:

X1 ADD X2

has four possible interpretations, depending on the tags of X1 and X2:

(a)	X1 = X1 + X2	(both data)
(b)	content (X1) = content (X1) + X2	(address, data)
(c)	X1 = X1 + content (X2)	(data, address)
(d)	content (X1) = content (X1) + content (X2)	(both address)

An interpretation similar to A/F allows the absolute jump command to discriminate between control links in registers and stored codewords.

Note that both the A/F and the A/S (see next paragraph) conventions apply to the depth of one level only, i.e., to addresses of data, but not to addresses of addresses of data.

3.1.5 The AutoStore (A/S) Convention

When transferring data from register to program store, the tag of the register must conform to the type of store element addressed. With type 6 elements, no problem arises, but for other types the following rules apply.

Type 0 elements may receive tag 0 or tag B data. Tag 0 is truncated to the appropriate size, possibly signalling overflow if significant digits are thereby lost. Tag B data are rounded to the nearest integer value and then stored as tag 0. (If the truncated part of the mantissa has a value of exactly one half, rounding is "away from zero".)

Type 3 elements may receive either tag 0 or tag B data. Tag B registers are stored directly. Tag 0 is converted to floating point form before storing.

Examples of rounding and truncation:

fl.pt. number	resulting integer
8.3	8
-8.3	-8
8.8	9
-8.8	-9
8.5	9
-8.5	-9

3.2 Arithmetic and Logical Functions

ADD	Add	NOT	Not
AND	And	NSB	Position of Most Significant "1"
DIV	Divide	OR	Or
MPY	Multiply	SC	Scale
MV	Move	SH	Shift
NEQ	Not Equivalent	SUB	Subtract

3.2.1 General Remarks

All functions set the CC according to the sign and value of the result (see below).

Each argument must define a type 0 or type 3 operand, following the A/F convention. The logical functions (AND, OR, NEQ, NOT) require both operandsto be type 0. The remaining functions (except NSB, MV, SC and SH) handle operands of mixed type by converting both to type 3 for the computation. The function is applied to produce a primary result which is used to set the CC and initiate monitoring action if indicated. (A primary result of type 3 is always in normalized form.) For the test-only variant, the function terminates at this point. For the three-address variant, the primary result is placed in X0. For the two-address form of the instruction, the tag of the first argument determines:

(i) whether conversion to type 0 is needed; if it is, the appropriate rounding and truncation are carried out (see Section 3.1.5). In either case, the result of (i) is the secondary result.

(ii) whether the secondary result is placed directly in the first argument register, or is stored by the A/S convention. Overflow on A/S resets the CC and may cause monitoring.

The resulting type, given in the table below for various combinations of operands, is the type of the secondary result for the two-address form, and the type of the primary result for the three-address variant.

type of operands	type of the result of function application	
	two-address form	three-address variant
both integers	integer	integer
both floating	floating	floating
first integer, second floating	integer	floating
first floating, second integer	floating	floating

Available variants are indicated in parentheses with the formats. Overflow may occur only in the case of ADD, DIV, MPY, MV, SC, SH, and SUB.

The value of S must be such that

$$-32768 \leq \underline{S} \leq 32767$$

unless stated otherwise.

3.2.2 Function Details

ADD: The primary result is the arithmetic sum of the two operands.
The format is X ADD XS (*† variants)
e.g., X3 ADD X4
Overflow may occur.

AND: The primary result is the logical product of the two operands.
The format is X AND XS (*† variants)
e.g., V6 AND #00FF
Both operands must be of type O.

DIV: The primary result is obtained by dividing the first operand by the second.
The format is X DIV XS († variant)
e.g., X0 DIV X2

If both operands are integers, their values v must be such that

$$-(2^{31}-1) \leq v \leq 2^{31}-1$$

A zero divisor causes overflow to be signalled; otherwise, the CC are set by the sign and value of the quotient.
Function forms:

(i) the two-address form: the quotient is sent to X; the remainder is lost whatever the types of the operands;

(ii) the three-address variant: the quotient is sent to X0; if both operands are integers, the remainder is an integer and has the sign of the divisor; it is sent to V0. If one or both operands are floating point numbers, the remainder is lost.

See Example 3 in Section 3.2.3.

MPY: The primary result is the arithmetic product of the two operands.
The format is X MPY XS († variant)
e.g., X5 MPY X5

If both operands are integers, their values v must be such that

$$-(2^{31}-1) \leq v \leq 2^{31}-1$$

and result consists of the 32 l.s. bits of the product.
Overflow may occur.

MV: The primary result is the value of the second argument.
The format is X MV XS (* variant)
e.g., X0 MV #10

If the types or tags of both operands are not the same, the primary result is converted to the type or tag of the first operand. If the first argument defines a floating point operand, the primary result is normalized. Overflow can occur on conversion or on A/S.

There exists a special negated form, in which the value is negated arithmetically.
The format is X MV -X (* variant)
e.g., X3 MV -X4
Overflow can also occur by negation.

NEQ: The primary result is the logical non-equivalence or symmetric difference of the two operands.

```
The format is     X     NEQ    XS                    (*' variants)
          e.g.,   V4    NEQ    X5
```

Both operands must be of type 0.

NOT: The primary result is the logical complement of the second operand.

```
The format is     X     NØT    XS                    (*' variants)
          e.g.,   X6    NØT    X6
```

Both operands must be of type 0.

NSB: The primary result is an integer related to the bit position of the m.s. "1" in the second operand, as given below.

```
The format is     X     NSB    X
          e.g.,   X3    NSB    X5
```

Denote the second operand by y, the bit position of the m.s. "1" in y by m; then the result r is as follows:

y of type 3		$r = 31$
y of type 0:	$y = 0$	$r = 0$
	$y > 0$	$r = 1 + m$
	$y < 0$	y is negated arithmetically, then r computed as for $y > 0$

Although the primary result depends on the second operand only, the first argument must define an operand of type 0 or 3.

OR: The primary result is the logical sum of the two operands.

```
The format is     X     ØR     XS                    (*' variants)
          e.g.,   X7    ØR     #0F0F
```

Both operands must be of type 0.

SC: The primary result is obtained by scaling (shifting arithmetically) the first operand by the amount specified by the second operand.

```
The format is     X     SC     XS                    (*' variants)
          e.g.,   X0    SC     13
```

The second argument must define an integer n such that

$$-64 \leqslant n \leqslant 63$$

The shift is left for n positive, right for n negative.
If n is defined by A/F, only the 8 l.s. bits are taken; of these, the m.s. bit (bit 7) is taken as the sign, bit 6 is ignored, and bits 0(5) give $|n|$.

If the first operand is of type 3, the second operand is added arithmetically to the exponent.
Overflow can occur.

SH: The primary result is obtained by shifting (logically) the first operand by the amount specified by the second operand.
The format is $\underline{X}$ SH $\underline{XS}$ (* ' variants)
e.g., $\underline{X5}$ SH $\underline{-4}$
The second argument must define an integer n such that

$$-64 \leq n \leq 63$$

The shift is left for n positive, right for n negative.
If n is defined by A/F, only the 8 l.s. bits are taken; of these, the m.s. bit (bit 7) is taken as the sign, bit 6 is ignored*, and bits 0(5) give $|n|$.
Both operands must be of type 0.

SUB: The primary result is obtained by subtracting the second operand from the first.
The format is $\underline{X}$ SUB $\underline{XS}$ (* ' variants)
e.g., $\underline{X4}$ SUB $\underline{X0}$
Overflow may occur.

3.2.3 Examples

1. Let X2 contain the integer 105_{10}= 1101001, X3 the integer 71_{10}= 1000111. (Leading zeros in binary numbers have been omitted.)

The following table shows a few independent commands with their primary results. (Within the Basic Machine, the primary result is represented in two's complement.)

command			primary result
X0	MV	-X2	-105_{10}
X2	ADD'	X3	176_{10}
X2	SUB'	X3	34_{10}
X2	AND'	X3	1000001
X2	ØR'	X3	1101111
X2	NEQ'	X3	$\overbrace{0.....}^{25}00101110$
X1	NØT	X3	$\overbrace{1.....}^{25}10111000$
X2	SH'	5	110100100000
X3	SH'	-2	10001
X2	SC	1	210_{10}
X3	MPY'	3	213_{10}
X2	DIV	21	5_{10}
X0	NSB	X3	7_{10}

---

* On the experimental machine, bit 6 is compared with bit 7; if they are the same the shift is logical, but if they are different the shift is arithmetic. In the latter case, overflow can occur.

3.3 Addressing Functions

IND	Index	MOD	Modify
LD	Load	PTS	Change Protection, Type, and Size
LIM	Limit	RMOD	Replace and Modify
MEM	Set Membership	TAG	Set Tag

3.3.1 General Remarks

Only LD, MEM, and TAG* can set the CC. Overflow can occur only with LD (see below). A/F applies only where specified.

Unless restricted otherwise, the values of N and S must be such that

$$0 \leq \underline{N} \leq 32767$$
and $$-32768 \leq \underline{S} \leq 32767$$

In general, an address with tag 2(t) can occur wherever an address with tag A(t) is allowed; it should be kept in mind that a tag 2 address refers only to one element and it should be used accordingly.

Available variants are indicated in parentheses.

3.3.2 Function Details

IND: The primary result is the index of the second argument.
The format is X IND X
e.g., X3 IND X5
The second argument must be an address with tag A(0),A(2),A(3), or A(6). The result has tag 0 and is the index or current limit of the (possibly singular) sequence addressed by the second argument.

LD: The primary result is the value of the second argument, as defined below.
The format is X LD GES
e.g., X0 LD -21 (i)
X2 LD 17 E 2 (ii)
X3 LD X7 (iii)
X4 LD ARR (iv)
The result is placed in the first argument register. CC are set by a tag 0 or tag B result, not by a tag 2 or tag A result. Overflow can occur.

The subcases of GES are (see examples above):
(i) S: the result is the value of S;
(ii) E: the result is the value of E; (note that LD is the only machine function which can have a floating point number as an explicit argument;)
(iii) X: tag 0 or B: the result is copied from X;
tag A(0) or A(3): the result is obtained by A/F from the address in X;
tag 1 or A(1): monitored on execution;
tag 2(n), n=0,1,2,3,6: treated like tag A(n);
tag A(2): the result is the address formed by evaluating the codeword pointed at by X;
tag A(6): the result is a copy of the dump element (the contents of a register pair) pointed at by X;

(iv) G: the result is the address of the sequence denoted by the name G.

There exists a special negated form.
The format is X LD -X
e.g., X3 LD -X4
The result is negated arithmetically; it must have tag O or B.

LIM: The primary result is the address obtained by limiting the address given by the first argument to the index given by the second operand.
The format is X LIM XN (' variant)
e.g., X2 LIM XO
The first argument must have tag A(O), A(2), A(3), or A(6).
The second argument must define an integer (≥0) directly or by A/F. Monitoring occurs if the resulting address is invalid (e.g., if the second operand exceeds the index of the first argument).

MEM: The function tests whether the sequence pointed at by the first argument is part of the sequence pointed at by the second argument.
The format is X MEM X
e.g., XO MEM X2
Both arguments must be addresses with identical PTS codes; let the first argument point at the (possibly singular) sequence T, and let the second argument point at the sequence Z; the primary result is as follows: if T is part (or whole) of Z, then the primary result is the index of the first element of T in Z; if T is not part of Z, the primary result is a negative integer (its value is not significant). Note that MEM is concerned only with addresses of sequences, but not with the values of their elements. The result has tag O and sets the CC in either case.

MOD: The primary result is the address obtained by modifying the address given by the first argument by the value of the second operand.
The format is X MOD XN (' variant)
e.g., X5 MOD' X2
The first argument must have tag A(O), A(2), A(3), or A(6).
The second argument must define an integer (≥0) directly or by A/F. The value of the second operand is used to increment the location contained in the address (taking account of size), and is subtracted from the index. Monitoring occurs if the resulting address is invalid (i.e., if the value of the second operand exceeds the index of the first argument).

PTS: The primary result is the address obtained by adjusting the address of the first argument to conform to the type/size codes given by the second argument.
The format is X PTS N (' variant)
e.g., XO PTS O
X must be an address with tag A(O) or A(3); (if X has tag 2(O) or 2(3), the function fails;) N must have one of the following values:

N	description of set elements
0	unprotected, type 0, size 0
1	" " size 1
2	" " size 2
3	" " size 3
#33	" type 3, size 3
#80	protected, type 0, size 0
#81	" " size 1
#82	" " size 2
#83	" " size 3
#B3	" type 3, size 3

The protection, type, and size codes of the first argument are compared with N; iff all 3 conditions below are satisfied, the function is carried out:

(i) protection state unchanged, or an unprotected set made protected;

(ii) type unchanged, or type 3 changed into type 0;

(iii) size unchanged, or changed from a larger numeric value to a smaller one.

The resulting address has the new protection, type, and size codes (given by N); its index and location are adjusted to the new size.

RMOD: The primary result is obtained by modifying by the second operand the address resulting from evaluation of the first argument, and setting its index to zero.
The format is X RMOD XN (' variant)
e.g., V2 RMOD 18
The first argument must have tag A(2). The second argument must define an integer (⩾0) directly or by A/F. The primary result has tag 2. Monitoring occurs if the resulting address is invalid (e.g., if the second operand exceeds the index of the address obtained from evaluation of the codeword pointed at by the first argument).

TAG: The function sets the tag of the first argument to the value of the second argument.
The format is X TAG N (* variant)
e.g., XO TAG #B
For the two-address form, the value of N must be 0 or #B.
The first argument must have tag 0 or B. If part of a long register is set to tag 0, the complementary part is nulled.
The test variant permits the following values of N: 0,1,2,#A,#B.
The tag of the first argument is compared with N, and the result sets the CC as follows:

tag of 1st argument	N				
	0	1	2	#A	#B
0	ZE	GT	GT	OV	OV
1	GT	ZE	GT	OV	OV
2	GT	GT	ZE	LT	OV
A	OV	OV	LT	ZE	GT
B	OV	OV	OV	GT	ZE

3.4 Jump and Control Functions

J	Jump	MON	Set Monitoring Mode
JL	Jump if Last	PSB	Set Process Status Bit
JNL	Jump if Not Last		

3.4.1 General Remarks

According to their nature, jumps can be described as conditional or unconditional; according to their destination, as relative or absolute.

Conditional jumps depend on the value of the CC or of a specified register, and are taken if the jump condition is satisfied; if it is not satisfied, the next instruction in sequence is obeyed. They must jump to destinations within the current block, expressed as local labels or relative line counts in either direction (see below). A jump is conditional if the first argument is mnemonic code (CC), or if the function is JL or JNL.

Unconditional jumps are always taken. They may be to a destination within the current block, to a link address, or to a codeword pointing to a block of code (AC and RC codewords). They may also plant into a specified register a link address, which is the address of the next instruction in sequence. When a link address is planted, the current control modes and CC settings are saved with it; when a jump is taken to a link, the saved control modes and CC setting are restored. When jumping to a codeword, the current control modes are "or-ed" with those in the codeword to give a new setting, and the current CC setting is cleared.

Relative jumps have SID as the second argument. The value of S must be such that

$$-15 \leqslant \underline{S} \leqslant 63$$

and it is interpreted as a line count relative to the current line. (Comment, erased, or blank lines are not counted.) Thus the first instruction on the line preceding the current one has relative position -1, and the first instruction on the line following the current one has position 1. The relative count 0 refers to the first instruction on the current line. If there are two or more instructions on a line, it is possible to jump only to the first of them. ID is a label which must be defined elsewhere in the current block, either before, on, or after the current line.

Absolute jumps have X, ID, or G as the 2nd argument; G is a compound name defined locally or externally; X may be a codeword (tag A(2)), which describes a block of code, or a link (tag A(1)); ID is an external name.

Function forms: no variants are available; some functions have a special form as given below.

3.4.2 Function Details

J: If the jump is unconditional, the primary result is a jump to the second argument; if conditional, the primary result is a jump to the second argument if the jump condition is satisfied, or to the next instruction in sequence otherwise.

There are five formats:

(i)	CC	J	SID	e.g.,	ZE	J	3
(ii)		J	SID	e.g.,		J	LAB
(iii)	X	J	X	e.g.,	X2	J	X2
(iv)		J	X ID	e.g.,		J	X1 or J LAMP
(v)	X	J	G	e.g.,	X1	J	CONV.3

(i) CC must be one of the mnemonic codes (see Section 3.1.1); conditional, relative;

(ii) unconditional, relative;

(iii) a link is set in the first argument before the jump is taken; unconditional, absolute;

(iv) unconditional, absolute;

(v) this form is recognized by the assembler as equivalent to
X LD G; X J X
a link is set in the first argument; unconditional, absolute.

JL: The primary result is a jump to the second argument if the jump condition is satisfied; otherwise, the next instruction in sequence is taken.
The format is X JL SID
e.g., X3 JL -4

The jump condition and procedure depend on the tag of the first argument. If the first argument is an address with tag A(0), A(2), A(3), or A(6), the jump condition is satisfied if modification by one would lead to modification overflow; otherwise, X is modified by one. If the first argument has tag 0, the jump condition is satisfied if the value of X is $\leq$ 0; otherwise, X is decremented by one.

JNL: The primary result is a jump to the second argument if the jump condition is satisfied; otherwise, the next instruction in sequence is taken.
The format is X JNL SID
e.g., X2 JNL LA2

The jump condition and procedure depend on the tag of the first argument. If the first argument is an address with tag A(0), A(2), A(3), or A(6), the jump condition is satisfied if modification by one would not lead to modification overflow, and modification takes place before the jump is carried out; otherwise, X is unchanged. If the first argument has tag 0, the jump condition is satisfied if the value of X is > 0, and the jump is taken after decrementing the value of X by one; otherwise, X is unchanged.

MON: The current control mode is altered according to the value of the argument.
The format is MON $\underline{N}$
e.g., MON 2

The values of $\underline{N}$ and their meanings are:

-2	turn off label trace
-1	turn off overflow monitor
1	turn on overflow monitor
2	turn on label trace

The monitor codes are not additive.

PSB: The process status bit specified by the first argument is tested and set according to the second argument.
The format is PSB $\underline{N}_1$, $\underline{N}_2$

e.g., PSB 28, 2

The original value of bit $\underline{N}_1$ sets the CC to ZE (bit $\underline{N}_1$=0) or NZ (bit $\underline{N}_1$=1). Bit $\underline{N}_1$ is then possibly reset according to the value of $\underline{N}_2$ as follows:

0	set bit $\underline{N}_1$ to 0 (off)
1	set bit $\underline{N}_1$ to 1 (on)
2	leave bit $\underline{N}_1$ unchanged
3	invert bit $\underline{N}_1$

The value of $\underline{N}_1$ must be between 0 and 31 inclusive. See table 6 for process status bits used by the system; bits 24 through 31 are available to the user as flags.

3.4.3 Examples

1. It is required to write a short segment labelled "FILL" (to be embedded into a large program) to fill a given byte set with a given character. On entry to FILL, X0 points to the first element of the byte set, X1 holds the link, and X2 contains the given character (also a byte). For example, the byte set may be a printing buffer, and the character may be a blank (visible space). On exit from the segment, X0 should point to the last byte filled, X2 should be unchanged. Contents of X3 may be destroyed (overwritten) by the segment.

(i) A less efficient program segment, illustrating the use of several functions:

```
FILL:  X3   IND   X0   ← find index of given byte set
       X0   MV    X2   ← fill one byte with given character
       X3   JL    2    ← decrement index
       X0   JNL   -2   ← keep filling bytes
            J     X1   ← jump to link
```

(ii) A more efficient program segment, not using X3:

```
FILL:  X0   MV    X2;  X0   JNL   0;  J   X1 ← loop until bytes filled
```

2. Let process status bit 25 be a buffer overflow indicator. It is required to write a segment labelled "TBUF" to interrogate the indicator; if it is off, no action is taken and control returns to the link in X1; if it is on, it is turned off and control proceeds to the segment "BUFØ", where corrective measures will be applied.

```
TBUF:   PSB  25, 0          ← turn bit 25 off
        NZ   J   BUFØ       ← indicator was on, go to BUFØ
        J    X1             ← indicator was off, jump to link
```

3. Rewrite the shorter version of FILL (Example 1 above) so that after entering the segment, one label ("FLAB") will be traced once before entering the loop.

```
FILL:   MØN  2              ← turn on label trace
FLAB:   J 1                 ← dummy instruction
XO  MV  X2; XO  JNL  0;  J X1
```

A dummy instruction after "FLAB:" is necessary, because a labelled line must contain at least one instruction.
If we wrote:

```
FILL:   MØN  2
FLAB:   XO  MV  X2; XO  JNL  0;  J X1
```

FLAB would be traced on every pass through the loop. (The use of MON 2 presupposes that the block is assembled in label-tracing mode. See Section 5.)

Since a jump to a link restores the control modes and the CC settings (see Section 3.4.1.), it is not necessary to do "MON - 2" to turn off the label trace before exiting. See also Example 1 in Section 6.4 for further use of MON.

3.5 Tag Independent Functions

CLEA	Clear Register	DUMP	Dump Register
CPY	Copy Register	UNDU	Undump Register

3.5.1 General Remarks

All functions are concerned entirely with register manipulation, and transfers to and from the dump. They treat the register content and its tag as a unit; the tag is not interpreted.

CLEA, DUMP, and UNDU deal only with long register, i.e., with V-X register pairs; they accept only X-registers as arguments.

All functions leave the CC unchanged. No function variants are available.

3.5.2 Function Details

CLEA: The register pairs defined by the arguments are given tag 1.
The format is CLEA XP
e.g., CLEA X6, X5, X7

(There exists also the system function CLEA. See Section 4.1.).

CPY: The content of the register in the second argument is copied to the register in the first argument. The short/long register rules apply (see Section 3.1.6 and Example 1 below).
The format is X CPY X
e.g., V3 CPY X5

DUMP: The register pairs defined by the arguments are copied (with their tags), in the order given, to the top of the dump.
The format is DUMP XP
e.g., DUMP X3, X1, X5, X6, X2
The dumped registers are unchanged.

UNDU: The items from the top of the dump are transferred to the register pairs (with their tags) defined by the arguments, in the order given.
The format is UNDU XP
e.g., UNDU X2

3.5.3. Examples

1. CPY, MV, and LD: the instruction

X2 CPY X0

copies the content (with tag) of X0 into X2 irrespectively of the tag values of both X2 and X0; the instruction

X2 MV X0

checks the tags of X2 and X0; if they define (by A/F) operands appropriate for an arithmetic function, the instruction is obeyed (with A/S if indicated); otherwise monitoring results; the instruction

X2 LD X0

is carried out as follows: the tag of X2 is irrelevant; the tag of X0 is interpreted, and the function is obeyed accordingly (see Section 3.3.2), setting the tag of X2 on completion.

2. CLEA and LD: the instruction

```
CLEA   X3
```

sets the tags of X3 and V3 to 1; the instruction

```
X3  LD  O
```

sets X3 equal to O, and its tag to O. V3 is nulled if the tag of X3 was A or B, and unchanged otherwise.

3. DUMP and UNDU: the register dump works on the last-in-first-out (LIFO) principle. Therefore, to restore a sequence of registers previously dumped by one BL instruction, say

```
DUMP  X3, X1, X5
```

to their original contents, the sequence of names must be reversed when undumping:

```
UNDU  X5, X1, X3
```

6. Error Monitoring and Diagnostics

Diagnostic information can be broadly classified as follows:

(i) errors detected during assembly of program text;

(ii) errors detected whilst obeying program text;

(iii) selective label trace;

(iv) errors detected during peripheral transfers.

The first three are described below in Sections 6.1, 6.2, and 6.3, respectively; the fourth is dealt with in Section 8.2.

6.1 Assembly Monitoring

There are two sorts of error report given by the assembler, namely hard failure, which indicates an error in the input text, and soft warning, which indicates a possible source of programming error, i.e., a point where the assembler has had to take special action to produce executable code.

Each error is reported by a message of the form "assembly error n" (see Figure 6.1 for an example). The value of n, positive for hard failures and negative for soft warnings, describes the error (see Table 8). It would be a good exercise for the reader to determine the reason for each error message in Figure 6.1. If the text is being assembled with listing on (see LIST, Section 5.1), the message is output on the current listing device immediately below the line in question. For soft warnings, more than one message referring to the same line of text may be output. If the text is being assembled with listing inhibited, the message is output on the operator device (normally TW) and, according to the mode of assembly (see Section 2), gives additional information. The message is expanded to either "p assembly error n in direct mode" or "p assembly error n, line l, block b" where p is the process name, b is the name of the block of code and l is the count of the line in question (ignoring blank and comment lines) relative to the beginning of b.

Unlike a soft warning, a hard failure ignores every character following the failed instuction up to the next newline. Subsequent action depends on the mode of assembly. In direct mode the END assembly directive (see Section 5.1) is obeyed. In indirect (block assembly) mode, remaining lines of text up to and including the END of block are scanned so that as many potential errors as possible may be reported, but no further code or constants are stored and the block is left undefined; translation then continues in direct mode. A soft warning does not curtail translation in either mode.

6.2 Error Actions at Run Time

Normal sequencing of a process is interrupted by failures resulting from execution of either machine instructions (see Section 3) or system instructions (see Section 4). Each failure is described by an error code number (see Table 7). By default all failures are diagnosed by the system, which produces an interpreted error report and terminates the failing program sequence. However, for certain types of

machine instruction failure there are two alternatives to the standard error action. The user can either set a flag and continue the interrupted sequence, or specify his own diagnostic action which enables him to continue or not the interrupted sequence (see Section 6.2.1).

If the failing process owns a printer, then the system error report is output on it (Figure 6.2 is a sample printout); otherwise, the name of the failing process and the appropriate error code number are output on the operator device (normally the typewriter). Then the system reserves a printer, outputs a full error report, and releases the printer. The user can also output his own diagnostics on any device he has reserved. (In both cases, the system or user may have to wait for the desired device to become available.)

6.2.1 Machine Instruction Failure

By default all such failures (see Table 7 (i)) are interpreted by the system, and for certain failures (error code nos. 1,2,7,8) this is the only method of interpretation. For the remainder (error code nos. 3, 4,5,6,9,10) alternative diagnostic action is conditional on the setting of specific process status bits (see Section 1.2 and Table 6).

If PSB 21 is on, control is transferred to the user's own routine EXER, which must be defined as a single-entry block of code (EXER SET #11). EXER is entered by a CALL from the system and the user has access to parameters in the dump which give information about the cause and register contents at the time of failure. The system ensures that any failures which occur within the user's routine give rise to standard (system) error action. Exit from EXER is via a RET command, its argument specifying the way in which control is to continue, i.e., either repeat the failing instruction, or continue from next instruction in the interrupted sequence, or return to the system for the standard error action, or terminate the failing program sequence. If the user's error routine is no longer required PSB 21 should be turned off.

If any of PSB 8 to 14 (with the exception of PSB 10) are on, system interpretation of the corresponding failure is curtailed and control continues from the next instruction in the interrupted sequence after setting a PSB flag.

If neither of these alternatives is required a standard error report is output on the printer and the failing program sequence terminated.

During normal program sequencing, instructions which give rise to integer overflow are not monitored; instead, condition codes are set and can be tested by the programmer. However, if the program is being obeyed in mode 1 (see Section 3.1.2), which can be set or reset using the MON function (see Section 3.4) or the MODE function (see Section 4.2), integer overflow will be monitored and can be interpreted by any of the alternatives decribed above. In contrast, floating point overflow will always be monitored.

Example 1 in Section 6.4 illustrates the use of EXER and MON in integer overflow handling.

6.2.2 System Instruction Failure

Failures of this nature can only be monitored by the system and are independent of the process status bits. A shortened version of the standard error report (described in next Section) is output on the printer and the failing program sequence terminated.

6.2.3 System Error Monitoring

For all failures monitored by the system, an error report is output which contains information concerning the type of failure, the name of the failing process, and the contents of all registers which have been stacked in the dump since the program sequence was initiated. If this sequence was initiated by a STRT (see Section 4.2) in a user's routine, control returns to the instruction following the STRT, with the error code number stored in the register X2. This gives the user another opportunity to define his own error actions. He can either supplement the standard error report, or restart the failing program sequence after taking corrective action, or initiate a completely independent routine. However, if the failing program sequence was initiated by a CALL (see Section 4.2) or J (see Section 3.4) instruction in a user's routine, or NTRY (see Section 4.2), control returns to the system start point which suspends the failing process waiting for operator action. Alternatively, if the failing program sequence was initiated by a STRT, CALL or J at command level, control returns to the system, which terminates the CIS file.

For machine instruction failures, additional information is output: the contents of each register, the condition codes (see Section 3.1.1) and control mode (see Section 3.1.2) settings at the time of failure, as well as the failing instruction and its location within the block. For system function failures, the top 8 registers in the dump correspond to X7,X6,X5,X4,X3,X2,X1,X0 at the time of entry to the system function, including the link back to the user's routine (see Section 4.2.1).

Each register pair is fully interpreted according to the tag and type of the information contained. Figure 6.2 is an example of a diagnostic printout caused by a machine instruction failure. For registers which address data sets, the value of each element is printed in hexadecimal on successive line(s). At the beginning of such lines, the index in the data of the first element on the line is printed in decimal. For byte sets only, the set elements are printed in character form as well. For registers which address codewords and codeword sets, the codeword class is given. For registers which address one data item, the number is printed in decimal together with a description of its size. Registers which address code are described as links; registers followed by an asterisk denote marked links. Registers which contain data give the number in both decimal and hexadecimal. Note that registers which address data sets, codeword sets, data, code or codewords within a structured block of code, have the index path enclosed in parentheses. For all such registers, except links, the index given is not the true index, but the half-word index relative to the beginning of the named block. (The true index of an address is expressed in terms of the type and size of the item being addressed.) For relative byte data sets only, a "+" sign indicates the least significant byte in the half-word.

6.3 Label Trace

During assembly, selected labels can be stored within a block of binary code. Each such label, provided that the code is being executed in trace mode, will interrupt the normal sequencing of the program text (before the instruction(s) on the labelled line have been obeyed). The user has access to the name of the label and current contents of the registers, and can direct that the interrupted sequence be resumed or suspended after possibly outputting diagnostic comments. Alternatively, he must direct the system to output a standard trace report, given a suitably defined device, and resume normal sequencing. Tracing is therefore conditional on the trace assembly mode, the trace control mode, and the user's trace definition routine (if present).

The trace assembly mode is defined by the TRAC assembly directive (see Section 5.1).

The trace control mode is set or reset using the MON instruction (see Section 3.4) and/or the MODE function (see Section 4.2).

The user's trace definition routine, TRDF, is a process-based routine which monitors every label traced. TRDF must be defined as a single-entry block of code (TRDF SET #11), otherwise a system failure will result (e.c.no.18). TRDF is entered from a system routine via CALL. Exit from TRDF is via a RET command, its argument specifying whether or not a standard trace report is required before resuming normal sequencing.

6.3.1 Selective Tracing

The TRAC assembly directive determines which labels may possibly be traced, and the format of such trace. (The currently available formats are described in the Operating Manual.) Labels assembled in "TRAC 0" mode cannot be traced without reassembling the complete routine. The MON instruction (assembly time only) and the MODE function and TRDF routine determine which out of these possible labels will actually be traced and under what conditions.

For example, if the user requires a particular subroutine, which can be called by numerous other routines, always to be traced, then the MODE function should be used to set the label-tracing mode. However, if the user requires that subroutine to be traced only when used by some of his other routines, then the MON function should be used. (Example 2 in Section 6.4 illustrates the application of MODE and MON in selective tracing; Example 3 in Section 3.4.3 also deals with MON.) The MON instruction is also useful for conditional tracing within program loops, but the conditions must be defined at assembly time. Instead, the whole loop or routine could be obeyed in trace mode, and TRDF used to define trace conditions, bypass specific labels, or alter the trace report format. For instance, TRDF is particularly useful for debugging without reassembly of program blocks, and Example 1 in Section 6.4 illustrates its use in integer overflow handling.

TABLE 2 : CODEWORD CLASSES	
AA	Absolute set of absolute codewords
AC	Absolute set of code
AN	Absolute set of data
AR	Absolute set of relative codewords
AU	Absolute undefined codeword
BS	Backing store block
DC	Device codeword
DZ	Delayed ZERO codeword
FC	File codeword
LO	Lockout codeword
PC	Process codeword
RC	Relative set of code
RN	Relative set of data
RR	Relative set of relative codewords
RU	Relative undefined codeword
SB	Substitution codeword
XD	Register dump

The classes are grouped as follows:

Absolute: AA,AC,AN,AR,AU
Relative: RC,RN,RR,RU
Escape: BS,DC,DZ,FC,LO,PC
Substitution: SB
Register dump: XD

TABLE 3: MACHINE FUNCTIONS

Mnemonic	Variants	Primary Result	CC	Description
ADD	'*	x+y	S	page 32
AND	'*	x&y	N	32
CLEA		clear register pair		45
CPY		copy register Y		45
DIV	'	x/y	S	32
DUMP		dump register pair		45
IND		index (Y)		36
J		control jump	S(1)	41
JL		jump if last		42
JNL		jump if not last		42
LD	-	Y!	S(2)	36
LIM	'	X↑y		37
MEM		X in Y (no A/S)	S	37
MOD	'	X'y		37
MON		control monitor		43
MPY	'	xy	S	32
MV	-*	y	S	32
NEQ	'*	$x \neq y$	N	33
NOT	'*	not y	N	33
NSB		x=signif(y)	S	33
OR	'*	x or y	N	33
PSB		set process status bit N	S	43
PTS	'	change PTS code in X		37
RMOD	'	X.y		38
SC	'*	$x2^{y}$	S	33
SH	'*	x shift y	N(3)	34
SUB	'*	x-y	S	34
TAG	*	set tag X to N	S(4)	38
UNDU		undump register pair		45

Notes:

CC: S indicates CC set by result
N indicates CC set, but truncation overflow ignored
(1) CC set only on jumps to links or codewords
(2) CC set only if result has tag O or B
(3) N if shift, S if scale
(4) CC set only by the test variant

Arguments:
X first argument register
Y second argument register
x first operand, evaluated by A/F
y second operand, evaluated by A/F, or constant

TABLE 4: SYSTEM FUNCTIONS						
Mnemonic	Registers Overwritten* with Arguments				Section	Page No.
	X0	X2	X3	X4		
ABOL					4.2	61
ATLE	+	+			4.1	54
CALL		+	(1)		4.2	61
CLEA		+(2)			4.1,8.4	54,99
DIS		(1)			8.4	99
=	+	+			4.1	55
EXCH	+	+			4.1	56
FREE		+			4.2	61
GIVE		+			4.2	61
HALT					4.2	61
HEAD		+	+		8.4	100
MASK	+	+	+		8.4	100
MODE	+	+			4.2,8.4	62,100
NTRY		+			4.2	62
POSN	+	(1)			8.4	100
RESV	+	+			8.4	101
RET		(1)			4.2	62
RHEA	+	+	(1)		8.4	101
RPLY	(3)	+			8.4	102
SET	+	+	(1)		4.1	56
STOP		(1)			4.2	62
STRT		+	(1)		4.2	62
STRU	+	+			8.4	102
TASK	+	+	(1)		4.2	62
TFER	+	+	(1)	(1)	8.4	103
USER	+	+			4.2	63
ZERO	+	+	(1)		4.1	57

Notes:

- * X1 always overwritten with link
- \+ always overwritten
- (1) overwritten if argument present
- (2) arguments processed serially
- (3) overwritten with result

If an argument is X, and identical with the expected parameter position, the register is unaffected by the calling sequence.

TABLE 6 : PROCESS STATUS BITS	
0	Integer overflow
1	Exponent overflow
2	
3	Store protection violation
4	Illegal argument tags
5	Modification overflow
6	Synchronization fault
7	
8	Integer overflow control: If off, signal error; On, set PSB(0)
9	Exponent overflow control:If off, signal error; On, set PSB(1)
10	
11	Protection violation: If off, signal error; On, set PSB(3)
12	Illegal tag control: If off, signal error; On, set PSB(4)
13	Modification overflow: If off, signal error; On, set PSB(5)
14	Synchronization fault: If off, signal error; On, set PSB(6)
15	
16	On: Full GTAB and STAB available to the system process
17	
18	On: List input text on monitor device during assembly
19	
20	On: Machine or system function error has occurred
21	On: Obeying user's EXER
22	Used by diagnostics routine
23	Assembler use

TABLE 7 : ERRORS REPORTED AT RUN-TIME

Code	Error
(i)	Machine Instruction Failures
0	
1	Illegal function or variant (assembly failure)
2	Access to device or undefined codeword (DV,AU,RU)
3	Store protection violation
4	Illegal tags
5	Modification overflow
6	Synchronisation error
7	Dump overflow
8	Undefined jump destination or incomplete code
9	Integer overflow
10	Exponent overflow
(ii)	System Instruction Failures
11	Direct store allowance exceeded
12	
13	
14	Invalid file name
15	System failure
16	Time allowance exceeded
17	
18	Illegal system function arguments
19	No room for user
20	Attempt to redefine unowned structure
21	Monitor device required
22	
23	Illegal operating command
24	System function used at incorrect level
25	
26	

Contains B and associated arithmetic logic.

A typical instruction, ACC+ A (say), would enter the CPU via the IBU from the local store. It would be decoded in the PROP and as the operand is not a descriptor, the operand would be passed directly to Acc. If the operand were a descriptor, ACC+ A{B} (say), PROP would pass SEOP the contents of A and SEOP would request a word from SAC (the address of which would be obtained by adding the Origin field of A to B.) SAC would then return the word and SEOP format it before passing it to Acc.

In order to speed up the system, PROP contains a "Name Store" (Aspinall, Kinniment and Edwards, 1968). Whenever PROP needs an operand, it checks in the Name Store before doing a store access. If the operand has been used previously, the Name Store will hold its address and its value. An associative access takes 80nSec instead of 800nSec if a memory access is required. Also, PROP is organised as a "pipeline" (Ibbett, 1972) which means that it is processing more than one instruction at a time, in a serial manner, with each stage in the processing ending in the operand being stored in a register.

4. BLM "Codewords"

4.1 Concepts

BLM codewords are of the form;

Where G	TAG (4 bits) - defines the type of the codeword. If G=0, this codeword is binary data. If G=1, it is "null." An attempt to access this word will cause an interrupt. If G=%A, the codeword is a descriptor. This field is outside the normal word length, and access to it requires a special order.
P	PROTECTION (1 bit) - defines the access permission. If P=1, Read Only access to this codeword.
T	TYPE (3 bits) - defines the type of the codeword. If the codeword describes binary data, T=0. If it points to the start of a sequence of machine orders,

T=1. If T=2, this codeword points to other codewords. T=1 implies that the next level in the tree contains floating point numbers. If T=6, the codeword points to a "dump." A dump is a sequence of words whose tags are not checked when accessed.

S SIZE (2-bits) - define the size of the elements. This field is coded as $\log_2(\text{Size}) - 3$.

L LIMIT (15 bits) - The length of codeword sequences defined at the next level.

C ORIGIN (14 bits) - The starting address of the next level.

In BLM, all the store is assumed to be tree structured. The root of the tree is assumed to point to processes which are, in effect, multiprogrammed jobs. Each process may have associated with it, or may generate in the course of its execution, any number and depth of sub-processes. Even code sequences are regarded as the leaves of sub-trees:individual orders may not be directly accessed without attaching a label to them.

Every process has a base and a dump. The base is a set of eight registers (labelled X0,...,X7) and is used either as general registers or to hold codewords. A dump is a sequence of un-tagged locations and is organised as a stack.

Instructions are dependent on the tags of their operands because of the AutoFetch and AutoStore conventions. AutoFetch means that if a base element has tag "A," then the operand will be interpreted as an address from which to extract the operand. AutoStore means that an operand is interpreted either as the destination of the data or as the address of the destination.

4.2 Order Code

As BLM has two levels of order code, it is important to distinguish between the two. BBC (Binary Basic Code) is the machine language; which is, unfortunately, not described in any of the published papers on BLM. BL (Basic Language) is an auto-code (in the sense that certain BL orders generate more than one BBC order) and it is this language that is now presented.

BL orders are of the form;

N1 F N2

Where N1 is the first argument. It may be either a base element

or a condition code indicator. (see below)

F is the function. Functions are divided into four classes;

1) arithmetic and logical functions.
2) addressing functions.
3) jump functions.
4) tag-independent functions.

N2 is the second argument. It may be either a base element or a signed integer.

As with the explanation of the MU5 order code, fragments of code will be presented after a subset of the orders has been given.

CLASS 1: X ADD X

The second operand is added to the first.

X MV X

The second operand is moved to the first.

CLASS 2: X IND X

The second operand must be a codeword. The first operand then contains the Limit field of the second.

X LD X

The second operand is moved to the first. AutoStore is not applied to the first operand.

X LIM X

If the first operand is a codeword, its Limit field is set to the second operand.

X MOD X

The second operand is added to the Origin field, and subtracted from the Limit field,of the first operand. Bound checking is performed.

X TAG X

This can be used either for changing the tag of the first operand to the value of the second or for checking the value of the tag. If it is used for checking, the result is returned via the CC's (Condition Codes.)

CLASS 3: N1 J N2

The first operand is a CC, it may be;

GE
GT

IR (Invalid Result)

LE

LT

NZ (Non-zero)

UN (Unconditional)

VR (Valid Result)

ZE (Zero)

The second operand is either a signed integer or a label. An integer is interpreted as a relative line-count.

If the condition is true, control is passed to the location specified by the second operand.

X JL N2

The jump condition and procedure depends on the tag of X. If the tag is %A, then X is modified by one and if this would result in Bound overflow, a jump is made and the modification is not performed. If the tag is 0, then a jump is made if X ≤ 0 or, if not, X is decremented by one.

X JNL N2

If the tag of X is %A, X modified by one and if this does not lead to Bound overflow, a jump is made; otherwise, X is unchanged. If the tag is 0, a jump is made if X > 0 and after decrementing X; othewise, X is unchanged.

CLASS 4: X CPY X1

This copies X1 to X without the AutoFetch or AutoStore conventions.

Examples

1) To access array elements using Iliffe Vectors.

If base element X6 contains the root codeword, then the following code could be used:

```
XO  CPY  X6   (move codeword to work register)
XO  MOD  I1   (modify to select next sub-tree)
XO  LD   XO   (load next codeword)
XO  MOD  I2   (2nd. subscript)
XO  LD   XO
....
```

```
        X7  LD   XO  (move element to X7)
2)   To find the sum of a vector 100 elements long.
     If X6 contains a codeword for the vector, then;
        X1  MV   O   (X1 has tag O, used for holding sum)
        XO  CPY X6
   L:   X1  ADD XO  (AutoFetch on 2nd. operand)
        XO  JNL L   (will cycle to end of vector)
```

4.3 Implementation

There is very little published information on the hardware of the BLM; see for example (ICL, 1969). It appeared to have been implemented using micro-programming, and as a simulator on a ICL 1900 machine.

However, it is interesting to consider what sort of problems would have to be solved to implement BLM with a performance similar to that on MU5.

Two points strike one immediately;

1) The use of base elements as general registers is unfortunate. In order to overlap the execution times of orders, it becomes necessary to introduce some form of register "scoreboard." (Thornton, 1964)
2) Although the use of tagged storage is very useful, it makes deocding of instructions more difficult, and may cause problems with gating.

However, some of the MU5 techniques are applicable; the IBU and PROP (apart from the difficulty mentioned above) could be used on BLM.

Assuming that the core store of BLM is conventional; that is, consisting of n equal length words each identified by a location, then the tree structure suggested by Iliffe is difficult to create and modify during the running of the program. Further, since code sequences are included in the structure, and since the locations of leaves are unknown, it becomes difficult to avoid crossing page boundarys in a program with a fair number of control transfers.

5. Comparison between MU5 and BLM

The principal difference between MU5 and BLM is that where BLM

treats codewords as central to its structure, MU5 descriptors are included simply to ease the burden of accessing data structures. BLM assumes that programmers understand structure accessing, hence the AutoFetch and AutoStore conventions: MU5 helps programmers by forcing them to be explicit in their structure accessing. In a sense, this reflects the difference in level between the two machines.

The use of tagged storage in BLM is of dubious value considering the extra hardware needed, but as it eliminates the need for the large order code of MU5 it is perhaps justified.

A comparison of the size of code compiled is difficult; both machines have features that would appear to introduce "break-points" into the code used to solve large problems. However, it would appear that the MU5 needs an average of 32 bits per subscript for an array access, whereas BLM needs 64 bits. However, in the second example (see above) BLM needs 128 bits, whereas MU5 needs 176 bits. Again, BLM does more orders in the subscripting example and less in the summing example. But, whereas BLM can take advantage of the special nature of that example, MU5 is forced to use a more general technique.

6. Conclusion

This paper has presented subsets of MU5 and BLM order codes and attempted a comparison between the two machines. It has not been possible to compare the hardware of the machines, as no data is available on BLM.

REFERENCES

T. Kilburn, D. Morris, J.S. Rohl and F.H. Sumner (1968)
"A System Design Proposal"
IFIP 68.

J.K. Iliffe (1972)
"Basic Machine Principles" 2nd Edition.
MacDonald Computer Monographs.

J.K. Iliffe (1961)
"The use of the Genie system in numerical calculations"
Annual Review in Advanced Programming. Vol. 2. p.1-28.

D. Aspinall, D.J. Kinniment and D.B.G. Edwards, (1968)
"Associative memories in large computer systems"
Information Processing 68. Vol. 2.

R.N. Ibbett (1972)
"The MU5 instruction pipeline"
Computer Journal. Vol.15, No. 1. p.42-50.

International Computers Ltd. (1969)
User Manual for BLM

J. D. Thornton (1964)
"The design of a computer — Control Data 6600."

```
MBER = 5 IN PROCESS SYST

ENTS AT FAILURE
.,NULL,.....                                              X0   MNDV   ....CODEWORD....   DC
.,NULL,.....                                              X1   BLOK(,13)   ......LINK......
|'0↑24  ....DATA SET....  PROTECTED BYTES
#3   #5   #7   #9   #B   #D   #F   #11   #13   #15   #17   #19   #1B   #1D   #1F   #21   #2
#26  #3A
-/AB D E FZ
.,NULL,.....                                              X3   TABL,0,2,13              273096758
21'0↑11  ....DATA SET....  WORDS
#17D  #FFFFFFFD          #0          #B        #1234      #12345      #123456     #1234567
EDC  #FFFFFFED
.,NULL,.....                                              X5   NVAL(,2)   ....CODEWORD....  RC
-2044985336    #861C0008                                  X6                  12              #C
,0,31'0↑5  ..CODEWORD SET..  AU

        CODE(,75)         X4 MOD X6

STARTING AT TOP OF DUMP
(|'85*,↑6)   ....DATA SET....  BYTES
#32  #29  #27  #28  #34  #19

,0,1,1,0                  4660  HWORDS                    X    TABL,0,1,0,6                    58
.,NULL,.....                                              X    ......NULL......
|'0↑150  ....DATA SET....  BYTES
#10  #10  #34  #28  #25  #32  #25  #10  #21  #32  #25  #10  #1   #2   #0   #10  #2
#34  #25  #32  #33  #10  #2F  #2E  #10  #34  #28  #29  #33  #10  #2C  #29  #2E  #2
#25  #32  #25  #10  #21  #32  #25  #10  #1   #2   #0   #10  #23  #28  #21  #32  #2
#10  #2F  #2E  #10  #34  #28  #29  #33  #10  #2C  #29  #2E  #25  #10  #10  #10  #3
#21  #32  #25  #10  #1   #2   #0   #10  #23  #28  #21  #32  #21  #23  #34  #25  #3
#34  #28  #29  #33  #10  #2C  #29  #2E  #25  #21  #22  #23  #24  #25  #5A  #21  #2
#23  #5A  #21  #22  #5A  #21  #5A  #0
20 CHARACTERS ON THIS LINE   THERE ARE 120 CHARACTERS ON THIS LINE   THERE ARE 120



.,NULL,.....                                              X    ......NULL......
,11'5↑22  ....DATA SET....  HWORDS
 #FFFD    #0     #B     #D   #FF51    #4F     #22     #11     #8     #4     #2
 #FFF8  #FFF3  #FFEB  #FFDE  #FFC9  #FFA7
|'4↑18  ....DATA SET....  FLT PT
E200000 #EF      #61A80000 #F2       #7A120000 #F5       #464B4000 #F9       #5F5E100
7E10DE6 #31      #4EE2D6D3 #4C       #66666666 #DE       #51EB851E #DB       #4189374
3E2D624 #D1      #431BDE83 #CE       #6BAFCA6B #CA       #55E63B88 #C7       #734ACA5
    #0 #80
.,NULL,.....                                              X    NTAB,3             125B    3   FLT P
    75B    3        #4B000000 #EB
           0               #0                             X*   BLOK(,26)   ......LINK......
.,NULL,.....                                              X    NAME   ....CODEWORD....   AC
           2               #2                             X    STAS(,10)   ....CODEWORD....   RC
           0               #0                             X                  15              #F
```