# ICL

## Product specification

**PSD** 2.5.1

Sheet 1

Issue 6/0

Holder

Copy no.

Distribution Centre to insert name and copy number of holder

**Title**  PRIMITIVE LEVEL INTERFACE

| | | | |
|---|---|---|---|
| Prepared by: | various | Technical authority: | RS&P, SCD |
| Location: | | Name: | J. E. Thompson |
| Enquiries to: | J. E. Thompson | Location: | BRA 01 |
| Location: | BRA 01 | Issuing authority: | PDG Documentation |
| | | Location: | BRA 01 |

Status **Red**

**Instructions to holders**
(The PSD is incomplete until these instructions have been carried out)

| Issue | Sheet | Instructions | Replaces | Signature of Technical Authority | Signature of Issuing Authority | Date of Issue |
|---|---|---|---|---|---|---|
| 6/0 | 1 - 147 | Re-issue | 5/6, CP36 CP37 | *J.E.T.* | *[signature]* | 27.3.80 |

| | | | PSD | 2.5.1 |
|---|---|---|---|---|
| **ICL** | **Product specification** | **Company restricted** | **Issue** | 6/0 |
| | | | **Sheet** | 2 |

0         **DOCUMENT CONTROL**

0.1      **Contents List**

**ICL**

**Product specification**

**Company restricted**

**PSD** 2.5.1

Issue 6/0

Sheet 4

Appendix 1: Alphabetical list of instructions
Appendix 2: Summary of additional facilities at AML1
Appendix 3: List of microcode routines

## 0.2 Status Record Details

Nil

## 0.3 Change proposals approvals list

All approval centres

## 0.4 Document cross-references

| Ref | PSD | Issue | Title |
|-----|-----|-------|-------|
| [1] | 4.2 series | | Processor specifications, pts I &II |
| [2] | 2.5.2 | 2/4 | PI for Peripheral Controllers |
| [3] | | | |
| [4] | | | Current KERNEL documentation |
| [5] | 2.4.1 | 4/0 | ICL/CDC Compatible Trunk Interface |
| [6] | 2.3.8 | 1/3 | Primitive level system communication and control |
| [7] | 2.5.5, 2.5.6, etc. | | Primitive interfaces for Emulators |

## 0.5 Document predecessors

Previous issues of this document (see also section 1.2.2)

## 0.6 Document acceptances

All the contents of this document have either been agreed by representatives of CDD and SPD at resolution meetings, or circulated on paper and discussed and agreed formally before publication in this document.

## 0.7 Changes forecast

The whole of this document will be brought up to date, to reflect the changes that have occured during the previous three years.
The addition of such items as vectored interrupts, slaving facilities and new descriptor types will be included.

# 1    GENERAL

## 1.1   Scope

This document defines the range standard for the primitive level interface for P1, P2, P3 and P4.  The contents of subsequent sections are outlined below:

### Section 2:   Concepts and conventions

Describes the significance of the 'primitive level' referred to in the title of the specification, and its relationship to Alice and to other elements in the range structure.  Certain new concepts, defined in detail in later sections, are outlined, with the aim of making those sections easier to read.

### Section 3:   Registers and Image store

The information on which the processor operates directly is held, outside the main store, in a well-defined set of registers.  This section defines those registers, dividing them into two classes: visible and invisible. The former are those whose contents are directly concerned in, and in certain cases may be altered by, the sequencing and execution of instructions at a non-privileged level.  The latter play a similar part at more highly privileged levels.

The 'Image store' mechanism which provides access to these and other registers, e.g. for diagnostic purposes, is also defined.

### Section 4:   Virtual store addressing and protection

Defines the virtual store accessing mechanism (which provides inter-process protection), and the in-process store protection mechanism.

### Section 5:   Instruction sequencing

Defines the way instructions are normally sequenced, and the rules for updating the Program Counter register; the procedure entry and exit mechanisms, including the way they exploit the stack; and the interrupt mechanism and conditions which lead to interrupts.

### Section 6:   Instruction and descriptor formats

Defines the formats of 16-bit and 32-bit instructions, and of all the different types of descriptor.  The rules whereby instructions access operands are given.

### Section 7:   Exception conditions

Classifies the various interrupt conditions which arise out of program execution.

**ICL**

**Product specification**

**Company restricted**

**PSD**   2.5.1

Issue     6/0

Sheet     6

**Section 8:   Instruction descriptions**

Defines the action of each instruction, including the error conditions which they may generate.

**Section 9:   Privileged operations**

Defines certain functions which may be implemented either as special instructions or by using other instructions to access privileged image store locations.

**Appendix 1:   List of instructions**

An alphabetical index giving function codes and section references.

**Appendix 2:**

A summary of additional features at AML1.

## 1.2  Introduction

### 1.2.1 Purpose of document

This document describes the range standard primitive features common to P1-P4, and indicates which features are implementation-defined. It does not define, for example, which instructions are expected to be extracoded on which models, nor does it go into detail on any implementation-defined aspects. For these the appropriate processor specifications (1) should be consulted. Nor is this document concerned with Input-Output facilities; these are defined in (2). Features used by microprogrammed emulators are included.

System components such as SAC, SMAC, etc. are defined in the processor specifications.

### 1.2.2 History

After the publication of NRSD 1.1.6, Issue 1/0 (20.10.69), which is totally superseded by the present document, the primitive interface was completely revised as part of the process of convergence with Manchester University. The revised primitive interface, as it applied to P3 and P4, was described in NWB 101-108. NWB 103-108, respectively, corresponded to sections 3-8 of this document.

Issue 1 of NWB 103-108 appeared at the end of March, 1970. A resolution meeting for these documents, chaired by the Deputy Manager, NRO, was held on 1st April, 1970, and on 9th-10th April, 1970 a further meeting with representatives of CEO(NW) and DPEO. As a result of these meetings Issues 2 of NWB 103-108 were produced at the end of April, 1970. At the same time NWB 109 was produced which described the points at which P1 and P2 (as originally planned) diverged from P3 and P4.

Issue 1/0 of this document, dated 29.5.70, was effectively a merger of NWB 103-108, Issue 2, with NWB 109, to produce a range standard for P1-P4.

Issue 1/1 (1.7.70) contained numerous small corrections aimed at tightening up and clarifying the specification, and some more significant changes.

Issue 2/0 was the outcome of a mini-resolution meeting held on 15.10.70, attended by members of NRO, SPO, CEO(NW) and DPEO.

Issue 2/1 (23.2.71) incorporated more detailed descriptions of the image store, privileged functions, real-time clock, and other changes, circulated prior to the end of 1970.

Issue 2/2 (24.3.71) was the outcome of a meeting with representatives of DPEO, CEO (NW) and SPO held on 10.2.71.

Issue 2/3 (26.5.71) incorporated changes circulated in April 1971.

Issue 2/4 (8.7.71) changed the status of the document to RED.

Issue 2/5 (10.9.71) resulted as a consequence of the policy decisions made by the New Range Working Party on 17th June 1971. The format of floating-point numbers was changed to that used in IBM 360 and 370 Computers (and System 4), and the EBCDIC internal character code was adopted as standard. Changes to the format of IT and IC were also incorporated. (CP2)

Issue 3/0 (2.12.71) contained cross-referencing between sections 7 and 8, several minor changes for clarification and for consistency with other documents, and some additional notes involving hardware changes. The real-time clock was also changed. (CP3)

Issue 3/1 (18.2.72) contained a change to the Validate Address instruction to incorporate adequate checks on stack references. (CP4/5)

Issue 3/2 (2.10.72) contained changes to improve efficiency of software particularly that written in S3, by reducing certain commonly-occurring overheads. (CP6)

Issue 3/3 (19.2.73) clarified certain details in line with hardware implementation. (CP7)

Issue 4/0 (11.7.73) introduced "null" string operations (L=0), removed ACR checks on the stack segment, described the features required by emulating machines, and included other minor amendments and clarifications (CP8).

Issue 4/1 (2/11/73) made slight changes to IS access rules and introduced the processor type number (IS block 0, line 16) and the bit distinguishing external CPU register IS addresses from SAC or trunk addresses (CP10, superseding CP9).

Issue 4/2 (31.1.74) qualified the absence of ACR checking on stack segment, and specified less stringent check on EM bits in SSR when switching to emulation mode.

Issue 4/3 incorporated the following changes:-

(a) ST, STUH to clear OV if operand is B.
(b) LSS to be permitted to use image store operand forms.
(c) ASF to leave SF unadjusted if VSI occurs.
(d) Checks on accesses above SF in stack segment to be implementation option.
(e) Stack segement not to be used for I/O (paging excepted).
(f) Actions of PK, SUPK undefined if more than 127 bytes involved.
(g) IC may be decremented "during" rather than "at termination of" each instruction (less precision implied).
(h) Space to be available in main store for the interrupt parameters on a stack switching interrupt.

Issue 4/4 corrected a typographical error.

Issue 4/5 incorporated the following change:-

Descriptors created by SIG and SUPK to have bits 2 - 7 defined.

The following changes were introduced in Issue 5/0.

(a) A new register called Linkage Table Base (LTB) is added and associated operand forms in place SSN.
(b) Three new instructions are added
    (i)   Load LTB  (LLT)
    (ii)  Store XNB (STXN)
    (iii) Store LTB (STLT)
(c) The Diagnose instruction (DIAG) is deleted.
(d) A new instruction Pre-call (PRCL) is added.

Issue 5/1 introduces the following changes:-
(a) Deletion of COM and EXP
(b) Clarifications to DEBJ, JAT, JAF, TTR, TCH.
(c) Stack segments must not be marked as NS.

Issue 5/2 introduced the following changes.
(a) Miscellaneous clarifications.
(b) 'Linkage Table Base' change to 'Cross-reference Table Base'.
(c) Description hardware System Call decode added.
(d) Need for Read access permission for Block 0 of Image Store deleted.
(e) Semaphore descriptors introduced and INCT and TDEC modified.
(f) EXIT modified to place link in DR.
(g) Microcode Descriptor introduced.
(h) Addition.

Issue 5/3 introduced the following changes
(a) Typographical corrections to Issue 5/2.
(b) Addition of Bit String operand form.
(c) Addition of OBS (Operate on Bit String) instruction.
Issue 5/4 introduced the following changes

Applicable to AML0 (and AML 1)

(a) Correction of minor errors from previous issues

(b) Appendix 3 is added listing microcode descriptor numbers

(c) Event pending interrupts (during ACTIVATE) are defined to be taken before completing an instruction if II is set.

Applicable to AML1 only

(a) Read Real Time Clock - alternative format

(b) System Call Count maintained by system call decode routine

(c) P Space Check - Number of parameters checked by system call decode routine.

(d) Modified VALIDATE instruction

(e) Extension of use of bit string operands

(f) OBS replaced by three functions - Test, Clear and Set

(g) Operand form (B+N) introduced

(h) Vector descriptors, size code 4 (=16 bits) introduced

(i) 'Signed' descriptors introduced

Issue 5/5 introduced the following changes:

(a) Definition of Block 2 Image Store for SCU1 systems.

(b) Preferred values of Block 0, sub-block 60, Image Store.

(c) Revisions to 'Processor Characteristics' Image Store, Block 0, Line 16.

(d) Relaxed requirements for Slave Store clearing.

(e) Miscellaneous corrections and clarifications.

Issue 5/6 introduced the following changes:

(a) Clarification of AML1.

(b) Removal of Bit String and TEST CLR and SET.

(c) Amplification of setting of USE bits.

(d) Miscellaneous corrections and clarifications.

Issue 6 introduce the following changes:

(a) A complete retyping of this document on a 7700 Information Processor has been made. This incorporates minor editorial corrections only.

(b) CP35 (Image Store Changes) has been added.

(c) CP36 changes of a minor nature to bring the document nearer to its final state. These include slaving conditions in TCH & TTR, protection of slaves, addition of 128 bit image store area and a restriction to one, in the indirection of Global Segment Tables.

(d) CP37 restoration of infrastructure characteristics in image store word 16

## 2 CONCEPTS AND CONVENTIONS

### 2.1 The primitive level

The 'primitive level' is the lowest level at which a description of a New Range Processor as a machine which sequences and executes instructions can be given. It corresponds to the hardware instruction code. Most users will program the system at one of two levels higher than the primitive level, i.e. either at a 'high level (using a high level language, e.g. PL/1), or at the Alice Level which is intermediate between 'high' and 'primitive' levels (Alice = Translator Target Language - see [3]). Alice has many features of an Assembly code but has greater significance in that, as its name implies, it is the language into which high level languages compile, and also, although it is not really in one-to-one correspondence with any primitive level instruction code, it is sufficiently close to that level to be implemented efficiently on any current or projected New Range processor.
The primitive level interface described in this document defines the facilities available to a programmer operating at the primitive level. These facilities are provided on P1, P2, P3 and P4. The description speaks of instructions, registers, and store, but these should not be assumed to mirror the hardware too precisely - some of the instructions described may be implemented on some machines by extracode, rather than directly in hardware; and since store addressing is virtual the composition and allocation of physical main storage is hidden, e.g. some 'registers' may be held therein. In other words, this is the description of an interface and does not attempt to define how the facilities are implemented in hardware on the other side of the interface.

### 2.2 New Concepts

#### 2.2.1 Virtual machines and processes

Unless explicitly described as real addresses, all store addresses are virtual, i.e. they refer to a conceptual 'virtual store' which is mapped onto the real store, rather than directly to the real store. A virtual address is converted by hardware to a real address (of whose value the user is ignorant) in the act of accessing the store.

The dynamic relocatability implicit in the virtual-to-real address conversion means that the allocation of real store can be optimised by the supervisory software which controls the mapping of the virtual store.

A virtual address is 32 bits long, and is the address of an 8-bit byte. The virtual store available to any user thus appears to be up to $2^{32}$ bytes in size. This virtual store is divided into 16384 segments each consisting of up to $2^{18}$ bytes; the segment number in any virtual address occupies the most significant 14 bits. Conversion of virtual to real addresses is performed using 'segment tables' held in the real store; two registers called the Local Segment Table Base register (LSTBR) and the Public Segment Table Base register (PSTBR) are provided to hold the real base addresses of the segment tables for segments 0-8191 ('local' segments) and 8192-16383 ('public' segments), respectively. These registers also hold limit values giving the largest permitted segment numbers in the ranges 0-8191 and 8192-16383, respectively, which may be

used. Each segment is also individually limited in size and its permitted modes of access controlled, by values held in the segment tables.

A program having a single instruction stream (i.e. not capable of being multiprogrammed with itself), which makes continuous and exclusive use of the procesor registers while it is being executed (i.e. their contents are preserved on interruption and restored when execution of the program is resumed), and which refers to a virtual store defined by LSTBR and PSTBR as described above, is called a process. The environment for a process is called a virtual machine. A virtual machine may support more than one process, each having a separate stack within the virtual store. One operating system may provide several virtual machines and by convention they all use the same public segments, i.e. the contents of PSTB are the same for all the processes controlled by one operating system. Changes to PSTB necessitate re-initialisation of peripheral controllers by software (see [2]). Different virtual machines will, by definition, use different values of LSTB. However local segments may be shared between them. Such shared segments, which may be numbered differently in the different virtual stores to which they are common, are controlled by a conceptual 'global segment table'.

The system of address conversion and store protection is described in detail in Section 4.

### 2.2.2 The Stack

The instruction code at the primitive level is based on the use of a last-in, first out stack, held in the virtual store. Each process has a stack. This stack is defined by three registers, as follows:

<u>SSN (Stack Segment Number - 14 bits)</u> SSN contains the number of the segment in which the stack is held. The stack 'base' is at location 0 of this segment, and the stack 'expands' through increasing addresses. The contents of SSN can only be changed by the action of supervisory software - the stack cannot 'overflow' from one segment to the next.

<u>SF (Stack Front - 16 bits)</u> The stack is 32 bits (1 word) wide - i.e. it expands and contracts in steps of one or more words. The contents of SF (n,say) indicate the number of words in the stack that are occupied; these are words 0-(n-1) of the stack segment, so that SF may be regarded as a pointer, relative to the base of the stack, to the first unoccupied word in the stack. The instruction code allows for 'reverse Polish' operations to be carried out which either remove the top item from the stack, or place a new item at the top of the stack; these operations automatically cause the contents of SF to be decremented or incremented by the appropriate number of words. SF may also be altered by an instruction which specifies a quantity to be added to it or subtracted from it - thus a specified number of locations may be added to, or deleted from, the stack, in a single operation.

<u>LNB (Local Name Base - 16 bits)</u> As well as being used for reverse Polish operations, the stack exists to provide each procedure (subroutine) with its own working area, or 'local name space'. When a procedure is called, a local name space is created for it at the top of the stack. This space is deleted on exit from the procedure.

Nested procedure calls will cause a succession of local name spaces to be built up on the stack, which will be deleted on a 'last in, first out ' basis as the procedures end.

The base of the current local name space is pointed to by LNB; i.e. LNB contains the word address, relative to the base of the stack, of the first location in the local name space. The quantity in LNB is always less than in SF.

The instruction code provides facilities for addressing items in the local name space relative to LNB. The value in LNB may be altered to point to a location a specified number of words below the top of the stack, and it may also be stored away at the top of the stack or elsewhere in the virtual store.

The sequence of events when entering procedures is illustrated in Fig. 0.
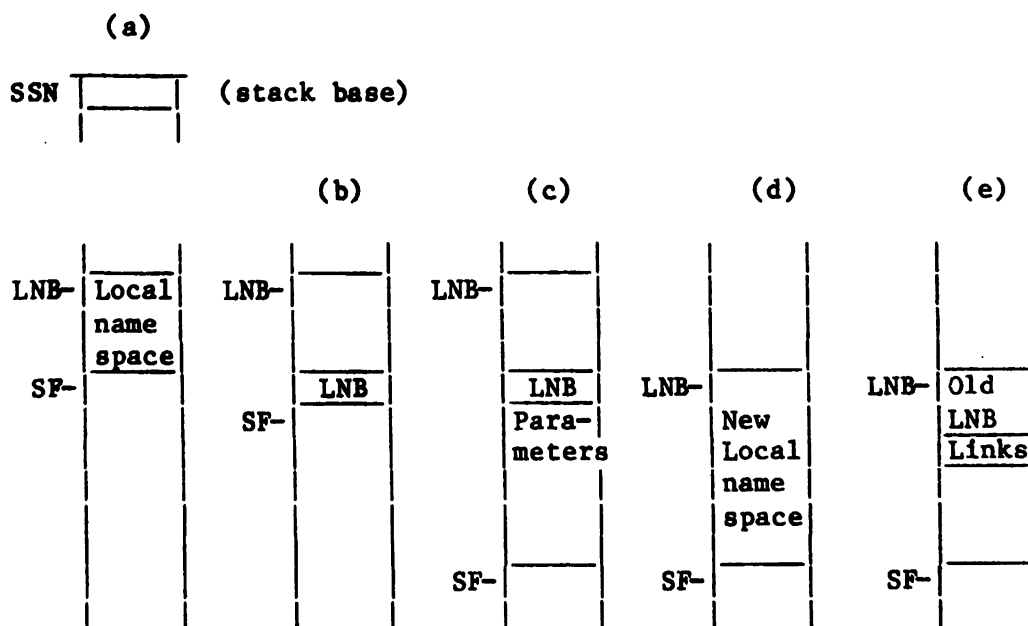


Fig.0

(a)  Shows the state of the stack just before the process starts to enter the new procedure.

(b)  LNB is stored at the top of the stack, causing SF to be incremented by 1 word. LNB is still unaltered.

(c)  Parameters are stored at the top of stack (leaving at least 2 words vacant next to the word where LNB was stored). This is accompanied by a further increase in SF.

(d)  LNB is altered (specifying a displacement from SF) to point to the location where its previous value was stored. A new local name space is thus created, which contains the old LNB and the parameters.

(e)  The 'Call' instruction is executed. This causes linkage information including the return address, to be placed in the two words adjacent to the location now pointed at by LNB, before control is transferred to the procedure. The procedure may now extend the local name space further by loading more items to the top of the stack, at the same time incrementing SF.

When the 'Exit' instruction is executed, the stack is returned to its
status quo at (a), and control is transferred to the return address.
Further information on the use of the stack and its associated registers
is given in Sections 3, 6 and 8.


## 2.2.3  Descriptors

The instruction code makes extensive use of 'descriptors' for indirect
addressing.  A descriptor is a 64-bit entity which formally describes an
item of information in the store.  The second word of the descriptor
contains the base address of the item described.  The first word contains
information relating to the unit size of the item, the number of units it
contains, whether modifiers added to the item's address should be scaled
or not, etc.

Full information on descriptors is given in Section 6.

## 2.3  Conventions

(a)  The contents of a register are usually referred to by the name of the
     register itself, thus SF, LNB for the contents of the registers SF
     and LNB.
(b)  The contents of the location whose virtual address is x are given by
     (x); e.g. (LNB + 1).
(c)  The contents of locations addressed by a descriptor in location x is
     indicated by ((x)); e.g. ((LNB+3)).
(d)  When a register or location has n bits, they are numbered 0 (leftmost
     bit) to (n-1) (rightmost bit).  Thus ACC (0-31) indicates the
     leftmost 32 bits of (the contents of) ACC.
     The same applies to fields within registers.
(e)  An address (also the register or field containing it) is said to be
     "word aligned" when it is a word number rather than a byte number;
     i.e. such an address is multiplied by 4 to convert it to a byte
     address.  "Halfword aligned", etc., are similarly defined (mutatis
     mutandis).


## 2.4  Ignored Fields

Where fields in registers, image store locations, or data or instruction
formats are ignored, they are described in one of three ways:

### (a)  Reserved

The field is currently ignored by hardware but may be used in the future.
Where there is a program control over the contents of the field, the only
information which a program should store there is a pattern of bits which,
if the field is later exploited by hardware, will ensure that the program
still works in the same way.  This pattern should be assumed to be 'all
zeroes' except where otherwise stated.

(b)   Unused (applies to fields in registers and image store only)

The field is unused by hardware and would only be used to allow for the expansion of adjacent fields; it may not be present at all, and the retrievability of information stored into it is undefined. Effectively the field is 'reserved' but the value stored in it by programs is immaterial.

(c)   Spare

The field is not, and will not, be used by hardware:  it may be used by programs to store and retrieve any pattern of bits, subject to the usual protection and privilege rules.

Note:   A field which is used by hardware in some circumstances may be 'reserved' or 'spare' in others.  The context will make this clear.

## 2.5  Architectural Mod Levels

The concept of Architectural Mod Levels (AML) is introduced to allow for progressive enhancements to the primitive level interface without instantaneously impacting the total population of 2900 systems. The basic AML is AML0 and it is a rule that AMLs are backwards compatible such that software written for systems at AMLn will run on systems at AMLn+1:  All new or enhanced software designed to exploit AML1 features will continue to run on AML0 machines.

The AML of systems can be read from line 16 of Block 0 of the Image Store. The sections of this specification which are affected by variations between AMLs are indicated in bold type.

A summary of additional facilities are given in Appendix 2.

## 3 REGISTERS AND IMAGE STORE

The 'image store' comprises the totality of addressable hardware registers. It gives the appearance of being a consecutively-addressed set of 32-bit locations. Image store locations may be accessed as such using suitable instruction operand formats, but require privilege to be accessed in this way.

Two particular subsets of hardware registers are described separately below. These are:

(a) 'Visible' registers — those registers accessible without privilege, i.e. without using the image store operand format.

(b) 'Invisible' registers — those registers accessed with privilege which define the virtual machine environment.

Some of these registers are shorter than 32 bits - in which case they may share image store location with other registers - and some longer, in which case they are spread over more than one image store location.

### 3.1 Visible Registers

The list of visible registers is, with mnemonics:

| | | |
|---|---|---|
| PC | Program counter | 31 bits |
| SF | Stack front pointer | 16 bits |
| LNB | Local name base pointer | 16 bits |
| XNB | Extra name base pointer | 30 bits |
| CTB | Cross-reference table base pointer | 30 bits |
| ACC | Accumulator | 32, 64 & 128 bits |
| B | Index accumulator | 32 bits |
| DR | Descriptor register | 64 bits |
| ACS | ACC size | 2 bits |
| OV | Overflow | 1 bit |
| CC | Condition code | 2 bits |
| PM | Program mask | 8 bits |
| RTC | Real-time clock | 64 bits |

#### 3.1.1 Program counter

This register points to the start of the current instruction and is half word aligned. It occupies bits 0-30 of an image store location: bit 31 is unused.

#### 3.1.2 Stack Front pointer

SF points to the word which is in front of the top of the stack. It is word aligned and may only point within the stack segment. Any references not below SF are invalid and the results will be unpredictable. When an item is stacked or unstacked SF is incremented or decremented by the

**ICL**

**Product specification**

**Company restricted**

**PSD** 2.5.1

Issue 6/0

Sheet 16

length in words of the item. Items are stacked most significant word first so that SF points to the word in front of their significant word, i.e. if the l.s. word is at address N, SF contains N+1. SF occupies bits 14-29 of an image store location; bits 30, 31 are unused. SF is concatenated with the invisible register SSN (which contains the stack segment number) for addressing the stack.

### 3.1.3 Local Name Base

LNB points to a word in the stack which is, by convention, the base of the local name space. It is word aligned and may only point within the stack. It has associated load instructions. It occupies bits 14-29 of an image store location; bits 30, 31 are unused. LNB is concatenated with the invisible register SSN (which contains the stack segment number) for addressing the local name space.

### 3.1.4 Extra Name Base

XNB may point to a word in the stack which will be, generally, the base of some addressing space. By convention it is used for back tracking lexical levels. It may also be used as a base address for directly addressed off-stack areas. It is word-aligned. It occupies bits 0-29 of an image store location; bits 30, 31 are unused.

### 3.1.5 Cross-reference Table Base

CTB may be used in the same way as XNB. By convention it is used to point to the base of a procedure linkage table. It is word aligned. It occupies bits 0-29 of an image store location, bits 30, 31 are unused.

### 3.1.6 Accumulator

The accumulator register is 128 bits long; its 32 bit portions are called A0 (most significant), A1, A2 and A3. The currently visible part of the accumulator register is referred to as ACC and may be 32, 64 or 128 bits long, as determined by ACS (3.1.9). ACC is right aligned in the accumulator register so that ACC (0-31) occupies A3, A2 or A0 depending on whether ACS-32, 64 or 128 bits respectively. Most arithmetic operations occur in ACC. A0-A3 occupy four consecutive image store locations.

### 3.1.7 Index accumulator

B is a 32 bit register whose contents are assumed to be a signed integer. Its main use is for descriptor address modification. Positive or negative overflow out of B during arithmetic operations on it sets OV and generates a maskable interrupt condition, It occupies one image store location.

### 3.1.8 Descriptor Register

The DR register holds a descriptor in standard descriptor format or the result of an explicit DR instruction. Any indirect address access places the descriptor used, in DR during the course of an instruction. The descriptor is in unmodified form unless explicitly changed by the instruction or changed by the operation of a store to store instruction. DR occupies two image store locations.

(The next four registers all form part of PSR in the image store).

### 3.1.9 ACC size

For all operations which involve the contents of ACC the conceptual size of ACC is given by ACS as follows:

        ACS = 0 Invalid
              1 32 bits
              2 64 bits
              3 128 bits

The contents of ACC are right-justified in the 128 bit accumulator register. The rules concerning size of operands are given in section 6.3.1. When ACS is increased the item in ACC is extended on the left with zeroes - i.e. the current contents of ACC are regarded as a bit pattern. If the change makes the ACC smaller then the item in ACC is truncated on the left with the remaining bits unchanged. Inspection of the truncated portion of ACC will not yield predictable results.

### 3.1.10 Overflow (OV)

This marker is set when the current operation in ACC (as qualified by ACS and the type of arithmetic operation) or B overflows. The marker may be inspected by the two Jump on arithmetic instructions. The marker is only and always set or cleared by instructions which change the contents of ACC or B. A "store" operation with B as operand will clear OV.
The conditions which set OV are:-
        Floating overflow
        Fixed overflow
        Decimal overflow
        B overflow

### 3.1.11 Condition code (CC)

This two bit marker is set by various instructions which produce transitory conditions which may be tested later. The marker remains set until the next instruction defined as setting it occurs, as defined in section 8.

### 3.1.12 Program mask

If the mask bit corresponding to a defined interrupt is 1 then when the interrupt occurs the interrrupt is reset and operation continues as though the interrupt had not occurred. The program mask is alterable by program by the 'Modify PSR' instruction.
        Bit 0 Floating overflow
            1 Floating underflow
            2 Fixed overflow
            3 Decimal overflow
            4 Zero divide
            5 Bound check overflow
            6 Size
            7 B overflow

### 3.1.13 Real-time clock (RTC)

The visible portion of RTC consists of two 32-bit registers, X and Y. The instruction 'Read real-time clock' (section 8.1.3.11) causes X to be stored in bits 0-31 of ACC, Y in bits 32-63. In a multiprocessor system X and Y are effectively common to all processors.

Register Y is a hardware binary counter with counting from any low order bit to give resolution of 2 microseconds at bit 31 (+ or - 0.01%). Register X may be maintained by software and is a continuation of Y with bit 31 duplicating bit 0 of Y.

An 'External' interrupt is generated whenever carry out of bit 1 of Y occurs. The carry is added to bit 0 in the normal way. Carry out of bit 0 is ignored. The interrupt remains pending if interrupts are masked. Software is expected to update X. Should a Read real-time clock instruction occur before the interrupt is serviced the value stored will show bits 31 and 32 of ACC not equivalent and one must be added to bit 31 to correct the value. To produce a true 64-bit number bits 33-63 must be shifted up one place. It is a requirement that the registers are not cleared by IPL reset, and that successive accesses to the real time clock using the RRTC instruction generate unique times. Apart from the Read real-time clock instruction, the X and Y registers can be read or written by the normal image store action.

A third, invisible, register, Z (up to 5 bits) may be set by an image store instruction to a value n in the range 0-31 (upper limit depending on counting frequency in register Y). This will have the effect of causing an additional external interrupt to occur when bit n of Y changes from 0 to 1; except when n=0 when this interrupt does not occur. However the regular overflow interrupt when bit 1 changes from 1 to 0 occurs regardless of the setting of Z.
See hardware Specifications for explicit definition of RTC Control on Multiprocessor Systems.

## 3.2 'Invisible' registers

This list of invisible registers is, with mnemonics:

| | | | | | |
|-----|-----|-----|-----|-----|-----|
| SSR | System status register | 1 | image | store | location |
| PSR | Program status register | 1 | " | " | " |
| LSTB | Local segment table base | 2 | " | " | " |
| PSTB | Public segment table base | 2 | " | " | " |
| IT | Interval timer | 1 | " | " | " |
| IC | Instruction counter | 1 | " | " | " |
| RTCZ | Periodic interrupt control (see section 3.1.13) | 1 | " | " | " |
| SSN | Stack segment number | 1 | " | " | " |

### 3.2.1 SSR

The system status register may be set by image store command if privileged. It has the fields:

| | | |
|-----|-----|-----|
| II | Bit 0 | Abnormal termination, instruction incomplete |
| RAM | Bit 1 | Real addressing mode if 1 bypassing segment tables |
| | Bits 2,3 | Reserved |
| PI | Bits 4,5 | Processor Identifier |
| EP | Bit 6 | Event pending |
| DGW | Bit 7 | Diagnostic write |
| ISR | Bit 8 | Image store read |
| ISO | Bit 9 | ISO numeric mode if 1 (see 8.3.3.10) |
| | Bits 10,11 | Reserved |
| EM | Bits 12-15 | Emulation Mode. Non-zero value indicates alien code to be emulated: |

            0001 - 1900 code
            0010 - System 4 code
     Other values to be assigned.
     Reserved on non-emulating machines.

Interrupt Mask (Interrupt masked if bit = 1)

| | |
|-----|-----|
| Bits 16 - 19 | Reserved |
| 20 | Instruction counter |
| 21 | Event pending |
| 22 | Extracode |
| 23 | Out |
| 24 | System call |
| 25 | Program error |
| 26 | Interval timer |
| 27 | Virtual store |
| 28 | Peripheral |
| 29 | Multiprocessor |
| 30 | External |
| 31 | System error |

## 3.2.2 PSR

The program status register may be set by interrupts, by privileged image store command or in part by the non-privileged Exit or Modify PSR instructions.

| | Bits 0-7 | Unused |
| --- | --- | --- |
| ACR | Bits 8-11 | Access Control Register (see section 4.2.2) |
| D | Bit 12 | Set by escape exit, reset by next instruction. If 1, use descriptor already in DR instead of from store. |
| PRIV | Bit 13 | Privilege (see section 3.4) |
| OV | Bit 14 | Overflow (see section 3.1.10) |
| E | Bit 15 | If 1, emulate alien code indicated by bits 12-15 of SSR. Reserved on non-emulating machines. |
| PM | Bit 16-23 | Program mask (see section 3.1.12) |
| | Bits 24-27 | Reserved. |
| CC | Bits 28,29 | Condition code (see section 3.1.11) |
| ACS | Bits 30,31 | ACC size (see section 3.1.9) |

When a Store instruction with image store operand or an interrupt is used to overwrite PSR, the following points should be noted:

1)  The effect of storing 0 into ACS is undefined
2)  If ACS is increased the new portion of ACC may not be cleared as occurs when a non-privileged instruction is used to perform this function. However the existing portion will not be altered. A reduction of ACS will work as defined in section 3.1.9.

## 3.2.3 LSTB

The local segment table base may be set by privileged image store command. Its use is defined in Section 4.1 and it occupies two image store locations.

| LSTB0 : | Bits 0-13 | Segment number limit (bit 0 = 0) |
| --- | --- | --- |
| | Bits 14-31 | Unused |
| LSTB1 : | Bits 0-3 | Unused |
| | Bits 4-28 | Double word aligned segment table base (real) address |
| | Bits 29-31 | Unused |

## 3.2.4 PSTB

The public version of 3.2.3 (bit 0 of PSTB0 = 1).

## 3.2.5 IT

The interval timer contains a 24 bit logical counter occupying bits 8-31 of an image store location, and a guard bit at bit 7. Bits 0-6 are unused. The counter counts down once every n microseconds ($1 \leqslant n \leqslant 16$); n may be obtained from the hardware manual and will usually, though not necessarily, be a power of two.

When bit 8 changes from 0 to 1, the guard bit is set, counting continues and a stack switching interrupt is attempted which if masked or not taken because of the simultaneous occurence of an interrupt with higher priority (section 5.3.7), remains pending (recorded by the guard bit): otherwise the guard bit is cleared and the rules for asynchronous interrupts are obeyed (section 5.3). The register may be read or written to by privileged image access; if writing sets the guard bit an interrupt is attempted.

### 3.2.6 IC

The instruction counter, if provided, is a 24 bit logical counter occupying bits 8-31 of an image store location, and a guard bit at bit 7. Bits 0-6 are unused.

The mechanism is the same as that for IT except that counting is by subtracting one during the execution of each instruction, and the interrupt, if it occurs is non-stack switching (section 5.3). Instructions which are restarted as a consequence of VSI's or automatic retries may decrement IC more than once.

The action of clearing the guard bit by hardware as the interrupt occurs prevents repeated interruption.

### 3.2.7 SSN

This register contains the stack segment number and is concatenated with SF and LNB for addressing in the local name space. It may only be changed with privilege or by the interrupt stack switching sequence or the hardware return from such an interrupt.

The contents of this register are constrained to be even. Any attempt to load it with an odd number leads to a System error interrupt. It occupies bits 0-13 of both the SF and LNB image store locations.

## 3.3  IMAGE STORE

3.3.1 The image store mechanism is a method of addressing hardware registers in the system.  It provides a method of performing operations at a hardware level which for reasons of privilege, timing or the level of control could not be performed at a software level.

The image store cannot be defined in its entirety as a range mechanism. Remarks are therefore made in the text following about range definition status.

The image store mechanism in particular allows system components to communicate with each other.

Here the 32 bit address form (address in B) is used to denote which 'External' register is being used.  A register is defined as any hardware source or destination of data (up to 32 bits in length) which is used to control or report part of the state of a system component. Each such register is allocated an address which is unique throughout the range.  By this means, it is hoped that compatible procedures between P1, P2, P3 and P4 may be evolved.  Where registers are common to all systems, then the same address will apply throughout.  An example of this might be 'Send Channel Flag to SAC on port number 2, trunk number 3'.  Here the address would be the 'System Control' register of the appropriate trunk 40230800 and the data sent to that address would be 00000001, both numbers quoted being hexadecimal (see [2] and [5]).

The addressing structure is more fully described in 3.3.3 below, and the data associated with the various registers in 3.5.

3.3.2 The image store is addressed by using the instruction address forms:

a)  Image store N, address = 18 bits of instruction literal N, extended with zeroes on the left.

b)  Image store B, address = bits 0-31 of B register.

Some image store locations are always cleared when read; others can be read in two ways, read or read and clear, depending on the image store address used.

Image store locations are normally 32 bits wide; however on some systems it is necessary to define image locations that are 64 or 128 bits wide.

Image store operands should only be used with LSS, LB, STB, CPB, L(ACS=32) or ST(ACS=32) functions for accessing 32 bit wide image store, or with LSD (or LSQ as appropriate) or L(ACS=64/128) or ST(ACS=64/128) functions for accessing 64/128 bit wide image store. The function used (read or write) must be appropriate to the type of accessibility (read, read/write, write, read and clear) of the image store location and the process must have sufficient privilege as given by the following table. The action is undefined if functions other than those listed above are used, or if the operand length of the function is not equal to the size of the image store location.

| PRIV | DGW | ISR | READ | WRITE | READ & CLEAR |
|------|-----|-----|------|-------|--------------|
| 0 | 0 | 0 | N (*1) | N | N |
| 0 | 0 | 1 | A | N | (*5) |
| 0 | 1 | 0 | N (*1) | DA | N |
| 0 | 1 | 1 | A | DA | DA |
| 1 | 0 | 0 | A | ) A or N | A |
| 1 | 0 | 1 | A | ) (*6) | A |
| 1 | 1 | 0 | A | ) A or DA | A |
| 1 | 1 | 1 | A | ) (*6) | A |

where A = allowed, N = not allowed
and DA = A for the diagnostic registers whose diagnostic
allow bit is set.  Otherwise N

NOTE:
* *1) Read access to Block 0 lines 0-16 is permitted without read access permission.
* 2) The CPU always checks write permission (PRIV = 1 or DGW = 1) before issuing a 'Write' command.
* 3) The CPU always checks read permission (PRIV = 1 or ISR = 1) before issuing a 'Read' or a 'Read and Clear' Command.
* 4) When PSR is addressed using LSS/IS.1, the new ACC size is obtained.
* *5) Read and Clear access to IS locations not having a Diagnostic Allow bit set is only permitted with PRIV = 1. When PRIV = 0, permission for Read and Clear access to IS locations having a Diagnostic Allow bit set can only be guaranteed if ISR = 1 and DGW = 1.
* *6) Write access to certain registers - specifically including the Trunk address register, IS address 40PT00XX - is only permitted if DA is set and DGW = 1, irrespective of PRIV. Such registers are only altered by test programs, and accidental alteration by Supervisor could be disastrous.
* 7) The CPU reports the permission fails in notes 2 and 3 above as Program Error type 9 subtype 3 (7.4.2.9.3).
* 8) Failure of other checks on the Image Store acces may be reported as Program Error type 9 subtype 4 (see 7.4.2.9.4).

3.3.3 Bits 0-2 of a 32 bit image store address indicate block number.  The remaining bits are interpreted according to the block number.

Literal image store addresses are limited to block 0.

Block 0:  CPU registers.  Other address bits as follows:

|       |                        |
|-------|------------------------|
| 3 - 15 | Reserved (must be zero) |
| 16 - 23 | Sub-block number |
| 24 - 31 | Register or line number |

Registers in block 0, sub-block 00 have range-defined addresses and include the visible registers (excluding RTC) and invisible registers (excluding LSTB, PSTB, RTC) described in sections 3.1, 3.2 (see section 3.5.1).

Registers in block 0, sub-block 60 are not range-defined
However, a list of preferred values of addresses is given in
section 3.5.1.

**Block 1: Reserved**

**Block 2: External registers.** Other address bits as follows :
SMAC based systems.

| Register | 0 | 4 | 8 | 12 | 16 | 20 | 24 | 28 |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| SMAC | 010X | 11XX | SMAC No | | SMAC reg address | | | |
| Store | 010X | 10XX | SMAC No | | Store mod | | Store reg | |
| SAC | 010X | 010X | Port | | SAC register address | | | |
| Trunk | 010X | 000X | Port | Trunk | Trunk reg address | | | |
| CPU | 010X | 0X1X | Port | see hardware specifications | | | | |
| | 3 | 7 | 11 | 15 | 19 | 23 | 27 | 31 |

X = Reserved bit

When addressing a SAC, Trunk or external CPU register, SMAC number is not
required since each is connected to the same port number on all SMACs. In
these cases, incorrect specification of bit 6, which distinguishes CPUs
from SACs or trunks, may lead to a Program Error (see 7.4.2.9.4)

Refer to processor specifications [1] for definitions of SMAC, SAC, port
etc.

SCU1 based systems

| 0 | 3 | 4 | 9 | 10 | 31 |
| --- | --- | --- | --- | --- | --- |
| 0100 | | DMN | | Reg Address | |

Bits 4 - 9 DMN, Destination Module Number Bits
Bits 10 - 31 address of a 32 bit image store location (not
a byte address)

**Block 3: 64 bit wide external registers.**

Block 3 is only available on SCU1 systems. The image store locations are
defined to be 64 bits wide. The format of address is

| 0 | 3 | 4 | 9 | 10 | 28 | 29 | 30 | 31 |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0110 | | DMN | | Reg Address | | 0 | | |

Bit 4-9    DMN Destination Module Number
Bit 10-28  Address of a 64 bit image store loaction
Bit 29     Reserved (must be zero)

Block 4   128 bit wide External Registers

Access to this block is only available to those systems in which the
infrastructure defined in PSD 2.5.11 is applicable.

Format of the address in B

    0-2    Block Number
    3-9    Reserved (set to zero)
    10-27  Address
    28-31  Zero

Address 1 of this block is for the purpose of sending messages along the
L64 highway from the OCP.  This method will not be normally used by the
supervisor which has access via the unit table, but is provided
specifically for those processors implementing their sub-supervisor
functions by use of 2.5.1 facilities.

The action of an image store access to this location is interpreted in two
ways, depending on the instruction used.

a)  Load using B (L IS/B)
    Send message in Acc to L2 highway,
    and await "data" response

b)  Store using B (ST IS/B)
    Send message in Acc to L2 highway,
    and await "ACK" reponse.

These two forms will normally be preceded by an LSQ (set Acc length to 128
bits and load data to Acc).

Notes
a)  The data sent from Acc to the highway is modified
    so that the source module number (SMN) is that of the
    OCP generating the message.  Similarily, transmission
    Functions are set as follows.

        TF0  set to 1 (command)
        TF1  as in data
        TF2  set to 0 (steering normal)

b)  Access to this block other than as described is indeterminate.

Blocks 5-7 : Reserved

### 3.4 Privilege

The privilege bits (PRIV, DGW, ISR) for image store access are held in PSR (13) and SSR (7-8) respectively. Use of these bits is described in sections 3.3.2 and 9. (Bit = 1 means privilege exists).

The privilege is checked against an access lock as described in section 3.3.2 at the time of access, there being a lock held conceptually with each image store location. Incorrect access to an image store location (wrong privilege, non existent location, read to write only or write to read only) is detected at this time and leads to a program error interrupt. On some processors, diagnostic parts of image store may be selected by writing to other image store locations.

The value of PRIV can only be altered in one of four ways:

a) by interrupt (including explicit System call)
b) by writing to PSR while in the most privileged mode
c) by Exit instruction
d) by Activate instruction

Action (a) gives more privilege, actions (b), (c), and (d) give less.

DGW and ISR may be altered in one of three ways:

a) by a stack-switching interrupt
b) by writing to SSR while in the most privileged mode
c) by Activate instruction

### 3.5 Image store map

### 3.5.1 Block 0

The range-defined locations in block 0 correspond to the interrupt dumping order for lines 0 - 1, and are:
(Lines 0-16 can be read without access permission.)

| Line | Register | Access |
|------|----------|--------|
| 0 | SSN/LNB | Read only |
| 1 | PSR | Read/write, write privilege if PRIV (PSR (13)) = 1 |
| 2 | PC | Read only |
| 3 | SSR | Read/write, write privilege as for PSR |
| 4 | SSN/SF | Read only |
| 5 | IT | Read/write, write privilege as for PSR |
| 6 | IC | Read/write, write privilege as for PSR |
| 7 | CTB | Read only |
| 8 | XNB | Read only |
| 9 | B | Read only |
| 10 | DR0 ) | Read only, DR0 = DR(0-31) |
| 11 | DR1 ) | DR1 = DR(32-63) |
| 12 | A0 ) | Read only, A0 = leftmost 32 bits, |
| 13 | A1 ) | A3 = rightmost 32 bits of |
| 14 | A2 ) | accumulator register |
| 15 | A3 ) | (not ACC) |
| 16 | Processor Characteristics | Read only. |

|  |  |
|--|--|
| Bits 0-7 | Processor Functional Module Number (SCU1 or SCU2 systems) |
| Bits 8-15 | Infrastructure Type |
|  | 00 SMAC/SAC |
|  | 01 SCU1 |
|  | 02 SCU2 |
|  | 03 To FF reserved |
| Bits 16-23 | Architectural Mod Level. |
| Bits 24-31 | Processor Type. |

| Bits 24-31 | Processor Type. |
|------------|------------------|
| 10 | 2950 |
| 20 | 2960 |
| 21 | 2956 |
| 28 | (S2+) |
| 30 | 2970 |
| 31 | (S3S) |
| 32 | (S3M) |
| 41 | 2976 |
| 40 | 2980 |
| 43 | 2982 |

Block 0, Sub-block 60

The following addresses are the preferred values which should be used on new designs of OCP, where applicable. They are intended to correspond to the values used on 2950, 2960 and 2970.

| Hex Addr | Register or Use | Access | Notes |
| --- | --- | --- | --- |
| 6000 | LSTB0 | R/W | |
| 6001 | LSTB1 | R/W | |
| 6002 | PSTB0 | R/W | |
| 6003 | PSTB1 | R/W | |
| 6004 | Reserved | | |
| 6005 | Reserved | | |
| 6006 | Hand keys | R | |
| 6007 | OCP Properties | R | |
| 6008 | Hooter Control | | |
| 6009 | Broad cast System error | W | 1 |
| 600A | System Interrupt | R/W | 1 |
| 600B | Reserved | | |
| 600C | RTC -Register X | R/W | |
| 600D | RTC - Register Y | R/W | |
| 600E | RTC - Register Z | W | |
| 600F | Clock module No | R/W | |
| 6010 | Reserved | | |
| 6011 | OCP Module Control (Bit 0 - Clear Address translation slaves) | R/W | 2 |
| 6012 | Load Microprogram | W | |
| 6013 | Cause System Error | W | |
| 6014 | Register Load | W | |
| 6015 | Reserved | | |
| 6016 | Display Register 1 | W | 2 |
| 6017 | Display Register 2 | W | 2 |
| 6018 | Stop OCP | W | |
| 6019 | Request Hamming Log | R/W | 3 |
| 601A | Enable Microcode Int.buffer | R/W | 3 |
| 601B | Clear specific CPR | W | |
| 601C | Cause SSN + 1 Dump | W | |
| 601D | Report OCP Failure | W | |
| 601E | Set Interrupt Destination Address | R/W | 3 |
| 601F | Set Coupler Function | W | 3 |
| 6020 | OCP Function Module No. | R | 3 |
| 6021 | Property Code of Module | R | 3 |
| 6022 | Store Blocks on Line | R | 3 |
| 6023 | Reserved | | |
| 6024 | SCU Property | R | 3 |
| 6025-60FF | Not defined | | |

For further details see the relevant OCP specification.

Notes.
1. These locations are for SMAC/SAC systems.
2. Use of these locations varies between 2970 and 2960 based systems.
3. These locations are for use by SCU1 based systems.

**ICL**

**Product specification**

**Company restricted**

**PSD** 2.5.1

Issue 6/0

Sheet 29

Block 0, Sub-block 63 .
Sub-block 63 is used by 2950,2956,2960 for multiprocessor communication

| Hex Address | Register or Use | Access | Notes |
|---|---|---|---|
| 6300 | Null | R | |
| 6301 | System Error Parameter | R | 4 |
| 6302 | System Error Interrupt | R | 4 |
| 6303 | Multiprocessor interrupt | R | 4 |

4. These locations are not used by 2950 or 2956.

### 3.5.2 Block 2

(Only SAC and Trunk register addresses range defined)

| ADDRESS (HEX) | CONTENTS | DETAILS | |
|---|---|---|---|
| | SAC | BITS | MEANING |
| 44PX00XX | Peripheral Interrupt | 0-15 | 'Normal' Trunk Flags |
| 44PX01XX | External Interrupt | 0-15 | 'External incident Trunk Flags |
| 44PX02XX | System Interrupt | 0-15 | 'Immediate action' Trunk Flags |
| | | 31 | SAC detected |
| 44PX04XX | Status | | |
| 40PT00XX | TRUNK Address (Read or write - see note 4 of 3.3.2) | 0-3 4-5 6-31 | Reserved (Byte lines) Spare Real Word Address |
| 40PT08XX | Control (Read and clear* or write) | 30-31 | CFA 0-1 (see [5]) |
| 40PT09XX | Function (Read only) | 28-31 | FFB0-3 (see [5]) |
| 40PT0CXX | System Status (Read and clear) | | |
| 40PT0DXX | Diagnostic Status (Read or write) | | |
| 40PT0EXX | Image Store Diagnose (Read or write) | 0-31 | Data for diagnostic mode |

P = Port number     *Read and clear checks are enforced on this
T = Trunk number     register for the sake of additional protection.
X = Reserved     Bits 30-31 are not actually reset on a Read.

## 4    VIRTUAL STORE ADDRESSING AND PROTECTION

### 4.1  Segment and Page Tables

4.1.1 The store is addressed from a 32 bit virtual address which is referenced to the virtual memory of the process using the address.

| Segment | Word | Byte |
|---|---|---|
| 0      13 | 14      29 | 30    31 |

The address translation hardware converts virtual addresses to 28-bit real byte addresses, which have the following format:

| RSN | Word | Byte |
|---|---|---|
| 0    1 | 2      25 | 26    27 |
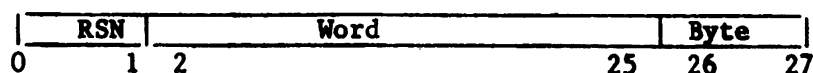
RSN = Remote Store Number.  Local (i.e. non-remote) storage may have RSN = 0, and vice-versa.

The real address is derived by referencing the virtual word-byte address to the base, or the page table, of the segment given in the first 14 bits. The segments are divided into two classes - those in the range 0-8191 are local to the process although they may be shared with other processes and those in the range 8192 up which are public.

4.1.2 In order to find the base, or page table, of the segment named, access is made by hardware to the segment table via a segment table base register (STBR).  A process has access to two STBRs, which are hardware registers, one for local (i.e. 'private') segments and one for public segments, the selection being made on the segment number used.  STBR format:

STB0

| Segment no. limit |
|---|
| 0      13 |

STB1

| Table base pointer (double word aligned real address) |
|---|
| 4      28 |
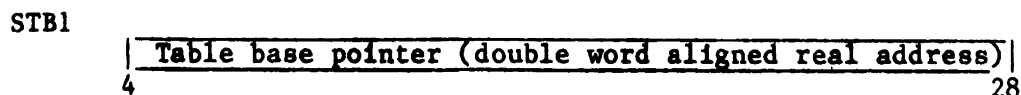
In order to find the entry for the required segment, the segment number being accessed is checked less than or equal to the limit (else virtual store interrupt).  Bits 1-13 of the segment number are then added to the base pointer and the result is used as a real double word aligned address to access the entry in the segment table giving the base of the required segment or of its page table.  The segment table is therefore a list of double word entries, double word aligned, indexed from the base by bits 1-13 of the segment number.

4.1.3 A segment table entry has the following form:

| 0 | 1 | 2 | 3 ... 11 | 12 | 13 | 14 ... 24 | 25 ... 31 |
| --- | --- | --- | --- | --- | --- | --- | --- |
| SP | P | NS | APF | SP | | Segment Limit | SP |
| A | I | USE | Real Address | | | | SP | F |
| 32 | 33 | 34 | 35 36 | | | 60 | 61 62 63 |

SP : Spare - usable by software

A : 0 - Segment unavailable
   1 - Segment available

> Note: If A = 0 and F = 0 the real address field is ignored by CPU and peripheral controller hardware, and may be treated as 'spare' by software, subject to the general proviso that undefined effects may result if alterations are made while there is a copy of the table entry in the Address Translation slave store - see section 4.3. See also [2].

P : 0 - Non-paged segment
   1 - Paged segment

I : 0 - Segment not shared; real address points directly at base of segment (P = 0) or page table base (P = 1)
   1 - Segment shared; real address points at Global segment table entry.

NS : 0 - Segment may be freely 'slaved'
   1 - In certain circumstances items in this segment must be removed, or excluded, from slave stores (see Section 4.3).
Segments used for stacks must not be marked NS and the OCP may assume without checking that the Stack segment has NS=0.

Use :

This field is ignored (spare) if I = 1 or P = 1.

Note: It is not set by I/O Controllers.
If I = 0 and P = 0 the bits indicate segment usage as follows:

Bit 34 - set to 1 by hardware when any location in the segment is referenced (read or write)
Bit 35 - set to 1 by hardware when any location in the segment is written to.

Hardware is permitted to set the "Referenced" bit if it prefetches any code or data from the segment.

At AML1 hardware is permitted to set the "use" field for the operands of an instruction which is started but not completed.

APF: Access Protection Field. This field must always contain meaningful information, even when A = 0. See section 4.2.2.

**Real Address:** (see section 4.1.1): Real addresses occupy bits 36 onwards (i.e. RSN is in bits 36,37) and point either to 128 byte boundaries (if I = 0 and P = 0), or to 8 byte boundaries (if I = 1 or P = 1). The real address field is accordingly truncated on the right, the least significant bits being spare and available for software use; in the former case bits 57 - 62 are spare, in the latter case 61 - 62.

**Segment limit:** bits 14 - 24 contain the maximum value which is permitted in the corresponding field (i.e. bits 14 - 24) of any virtual address referencing this segment. This allows for 128 - byte boundaries, which is the case with unpaged segments (P = 0). In order to cater for 1024 - byte page boundaries, bits 22 - 24 of this field must contain 111 when P = 1, otherwise undefined end effects may occur.

**F:** 'Fixed' bit = 1 indicates to peripheral controllers (see [2]) that the segment is wholly or partially available for I/O.

4.1.4 Having accessed the required segment table entry, the hardware has to check that the Access Protection Field permits access of the required type (see Section 4.2.2. - if access is not permitted a program error interrupt occurs), that bits 14 - 24 of the virtual address do not exceed the segment limit and that A = 1 (otherwise virtual store interrupt).

Subsequent action depends on the setting of I and P.

If I = 1 the second word of the segment table entry is interpreted thus:

| A | 1 | SP | Real address - double word aligned | SP | F |
|---|---|----|------------------------------------|----|----|
| 32 | 33 | 34  35 | 36                              60 | 61  62 | 63 |

The (Global) segment table entry pointed to by the double-word-aligned real address in bits 36 - 60 of the original segment table entry is referenced. This indirection process continues until a segment table entry with I = 0 is found. If any of the segment table entries thus referenced has A = 0 a virtual store interrupt occurs. The first word (bits 0 - 31) of each table entry referenced after the first is ignored; effectively, the segment table entry used to access the required item in virtual store is formed by combining the first word of the original segment table entry with the second word of the last one (which has I=0), the indirection process being omitted if I=0 in the original entry.
Only one level of indirection of this type is permitted; the effect of violating this rule is undefined.

4.1.5 If P = 0 the second word of the final segment table entry is interpreted thus:

| A | 0 | USE | Real address - 128 byte aligned | SP | F |
|---|---|-----|----------------------------------|----|----|
| 32 | 33 | 34  35 | 36                           56 | 57 | 62  63 |

The 128-byte-aligned real address is the segment datum, and is added to the word-byte number (bits 14 - 31; bit 24 aligned with bit 56 of the datum) of the original virtual address to produce the required real address. The USE bits in this case are interpreted as described in 4.1.3 (and are the only segment table field altered by hardware).

4.1.6 If P = 1, the second word of the final segment table entry is interpreted thus:

| A | 0 | SP | Real address - double word aligned | SP | F |
|---|---|----|-----|----|---|
| 32 | 33 | 34 35 36 | | 60 61 | 62 63 |

The real address points at the base of the page table for the segment. The page table contains one 32 bit entry for each page, and starts on a double-word (8-byte) boundary.

Bits 14-31 of the original virtual address are conceptually subdivided as follows:

| Segment | Page | Line | Byte |
|---------|------|------|------|
| 0       13 | 14      21 | 22      29 | 30      31 |

To form the real address of the required page table entry, the page number (bits 14 - 21), regarded as a number of words, is added to the double-word aligned page table base address.

The format of a page table entry is as follows:

| A | SP | USE | Real address - 1024 byte aligned | SP | F |
|---|----|----|-----|----|---|
| 0 | 1 | 2  3 | 4                                21 | 22 | 30 | 31 |

A : 0 - Page unavailable (causes virtual store interrupt to occur)
    1 - Page available

> Note: If A = 0 and F = 0 the remaining fields are ignored by CPU and peripheral controller hardware, and may be treated as 'spare' by software subject to the same proviso that applies to segment table entries (4.1.3.)

F :     Fixed bit, used by peripheral controller. See [2].
USE :   These bits indicate page usage in exactly the same way as the corresponding bits in the second word of segment table entries do for non-paged segments.

> Bit 2 - set to 1 by hardware when any location in the page is referenced (read or write)
> Bit 3 - set to 1 by hardware when any location in the page is written to.

The USE bits are the only page table field altered by hardware. These bits are not set by I/O Controllers. Hardware is permitted to set the "referenced" bit if it prefetches any code or data from the page.

At AML1 Hardware is permitted to set the use field for operands of an instruction which is started but not completed.

The 1024 - byte aligned real address is the page base, and is concatenated with the line-byte number (bits 22-31) of the original virtual address to produce the required real address.

**ICL**

**Product specification**

**Company restricted**

**PSD** 2.5.1

Issue 6/0

Sheet 35

## 4.2 Protection

4.2.1 A process is seen as having a certain entitlement to system resources and protection exists to confine the process to that level of entitlement. At the level the process may make access to store, a check is made by comparing the process level with the segment access protection (see this section). The process may require facilities at a lower level and a check must be made that it is entitled to access that lower level and that the parameters it passes are valid at its current level of entitlement. These latter points are covered part by hardware, part by software. The transition between levels carries hardware checking (see section 5) followed by software checks at the lower level. The checking of parameters is software initiated but aided by the hardware Validate instruction (section 4.2.3). Finally, since addresses derive largely from the stack, checks are made that the stack registers are handled reasonably although the security of the stack lies mostly in the way in which it is generated.
NOTE: Lower Level means more trustworthy, nearer to the Kernel.

4.2.2 Segment protection occurs in that on access to store the access control register contents (ACR) and the mode of access (execute, read, write) are compared with the access permission fields of the segment. Access is permitted (else program exception) if:

**Read** and t1 not less than ACR. t1 is read access key. (Note: Read access to current code segment for items addressed as (PC + N) (see section 6) is not checked)

**Write** and t2 not less than ACR. t2 is write access key.

Read or write accesses by a process to its stack, via any type of operand address, or as implicit in the operation of certain instructions, are not checked if SSN is less that 8192. A local segment may therefore have values of t1 and t2 less than the ACR of a process which uses it for its stack, and hence be protected from possible corruption by other processes in the same virtual machine (Section 2.2.1). Programmers should assume, however, that accesses to the stack are ACR-checked if SSN is not less than 8192, though this check may also be waived in some circumstances.

When ACR checking is applied via Address Translation Slave Store (Section 4.3), the absence of checks on a local segment may persist, even after a stack-switching interrupt of the process whose stack it contained, until the next execution of "Activate" has cleared the relevant slave stores.

**Execute** and t3 = 1. t3 is execute permission bit.

Hardware implementors may assume that t1 is not less than t2. An additional check is made for the execute access. A relative jump must not alter the segment number in PC (else program error). The result of writing into the segment currently being executed is unpredictable.

After writing to segments not currently being executed, care must be taken to clear those instruction slave stores which may contain items from many segments when the written segment is subsequently executed.
Where t1 = APF (5-8), t2 = APF (1-4), t3 = APF (0).

**ICL**

**Product specification**

**Company restricted**

**PSD**    2.5.1

Issue    6/0

Sheet    36

4.2.3 The Validate instruction (see section 8.1.5.9) is used by a lower level to check the validity of parameters passed from a higher level. The instruction takes the ACR value put on the stack at call time and compares it with the value of APF for the segment to which the descriptor being validated points. A condition code is set on the comparison.

4.2.4 The stack segment is protected by the way in which it is generated and also by minor checks upon stack register operations.

4.2.4.1 The stack segment number can only be altered by a hardware checked mechanism - interrupt or activate. SF and LNB must be in the same segment as stack base since they share the same SSN register. Operations on SF and LNB cannot alter the segment number, i.e. the stack cannot overflow its segment (else program error).

N.B. The segment limit for the stack segment must not have the maximum value - i.e. the segment size must not exceed 255 x 1024 bytes (paged) or 2047 x 128 bytes (unpaged). The results of violating this rule are undefined.

4.2.4.2 Manipulation of the stack for procedural use occurs largely under hardware mechanisms - the call, exit and escape exit mechanisms.

4.2.4.3 Checks apply to those instructions which do set stack pointers.

> e.g. Load LNB : Check new value less than SF
> Raise LNB: Check new value less than SF
> Adjust SF: Check new value greater than LNB

4.2.4.4 Any access to the locations between the current top of stack and the stack segment limit (except in the course of increasing SF) is considered incorrect because the contents of those locations are unpredictable. Hardware checks (e.g. on accesses using the operand forms (LNB+n) and (LNB+N) may be implemented.

4.2.4.5 On a stack segment, input-output operations (paging excepted) must not be performed at any time when that segment may be written to by an OCP operation. For other segments, a similar restriction may be implementation-defined.

## 4.3 Slave Stores

4.3.1 Parts of the main store may be 'slaved' to improve effective performance - i.e. the contents of various locations may be held in fast-access registers outside the main store. The choice of locations will vary dynamically in the course of program execution.

It is expected that slaving as implemented in New Range processors will adhere to the following principles:

(a) The slave stores will be associatively addressed using the virtual, rather than the real, addresses of the corresponding main store locations.

(b) Writing to store will always imply that the main store location concerned is overwritten (and possibly a slave store associated with it). However reading from store will frequently bypass the main store altogether, if the item concerned is currently 'slaved'. (see section 4.3.2).

One implication of these principles is that great care must be taken to avoid ambiguity in cases when two or more different locations in main store have the same virtual address (because they belong to different virtual machines), or where items in shared segments are referred to by more than one virtual address - a danger in the first case being that an item with a certain virtual address might be left in a slave store and later referred to there in error by another process which uses the same virtual address to reference a totally different item; in the second case a process might read an out-of-date item from a slave store, because another process had updated the main store location currently associated with that slave store but through using a different virtual address had not caused the slave store to be updated. A similar problem may arise as a result of peripheral input transfers which cause store locations, some of which may be slaved, to be overwritten, without updating the appropriate slave stores.

The action to ensure such ambiguities do not occur must be taken by hardware, and consists of clearing appropriate slave stores on specified occasions, e.g. before permitting a process to access the store locations which have been overwritten by a peripheral input transfer. It is also a rule that where segments are shared between processes, or where the same item is referred to by two different virtual addresses, and are updated, those segments must be marked 'non-slaved', and the processes referring to them must use the semaphore instructions. A segment used as a Stack segment cannot be used for this purpose.

### 4.3.2 Delayed Write-Through

Writing to store will always imply that the mainstore location is overwritten. However, this may be delayed, if there is an associated slave store, until the occurance of certain trigger actions. An exception to the above, occurs when writing to the Stack segment, in that items which were written but are subsequently above Stack Front, at the time of the trigger action, may be discarded.

The delayed write-through must be completed by those actions which cause the associated slave store to be cleared, namely:- Load PSTB, Activate, Semaphore Descriptor access, Clear Address Translation Slave Stores, Stack-switching interrupts (Stack Slaves only).

The data transferred into mainstore during write-through will always consist of an integral number of bytes. Write-through will not be applied to any byte unless at least one bit has been written to it since the last trigger action..

If during a write-through of any store location from slave stores a hardware error occurs which is reported by a System Error Interrupt, then the Interrupt Identifier must contain the value of ACR that was in use at the time of the most recent instruction that stored an operand in that location.

The above defines the legal requirements, but in practice it will be undesirable to delay all write-throughs until the occurance of the trigger actions. In general, it is reasonable to delay write-through to the address space between LNB and SF until the trigger actions occur. Writes to other areas may be delayed, but the OCP should be organised to 'trickle write' updated slave store locations to main store. This also applies to items below LNB after a CALL instruction. The trickle write must be completed by the trigger actions.

4.3.3 The slave stores are classified in one of the four following categories:-

1. Instruction Slave Store - Contains items from segments that have been fetched for the purpose of execution.

2. Stack Slave Store - Contains items from one segment only, the segment being the Stack segment defined by the SSN register.

3. Operand Slave Store - Contains items from all other segments.

4. Address Translation Slave Store - Contains items from Segement Tables and Page Tables.

4.3.4 The slave stores are cleared selectively by any one of the following actions:-

1. **Load PSTB** - These are instructions which, using Image store operand forms, store the contents of ACC into the two halves of the Public Segment Table Base Register (PSTB).

2. **Activate CPU Process** - This is an instruction which loads LSTB and all the registers necessary for starting/restarting a process from the segment SSN+1 (see Section 9)

3. **INCT and TDEC instructions** - Modify and test a specified store location without letting any other CPU or peripheral processor access the location until the completion of the instruction (see Section 8.1.2.7)

4. **Clear Address Translation Slave Store** - This clears the Address Translation Slave store of a processor and halts it. This is achieved by

**ICL**

**Product specification**

**Company restricted**

**PSD**   2.5.1

Issue   6/0

Sheet   39

writing to the internal image store location (block 0) sub-block 60 line 11 of the required processor with bit 0 = 1. (Note: see processor specifications for implementation deviations and method of writing to image store of other OCPs).

4.3.5 The action of each instruction on the slave store is now described.

    1.  <u>Load PSTB</u>  (these actions performed when either half of PSTB is altered)

Instruction Slave Store is cleared of all items referenced by segment Numbers 8192-16383.

Stack Slave Store is cleared completely if the segment Number of the stack is in the range 8192-16383.

Operand Slave Store is cleared of all items referenced by segment Numbers 8192-16383.

Address Translation Slave Store is cleared of all items translating the segment Numbers 8192-16383.

    2.  <u>Activate CPU process</u>
If the new LSTB is different to the old LSTB, then the Instruction Stack, Operand and Address Translation slave store are cleared of items referenced by segments 0-8191.

The Operand slave store is cleared of items in segments 8192-16383 if the segment is marked NS.

    3.  <u>AMLO</u>
<u>INCT and TDEC with all operands</u>

or  <u>AML1</u>
<u>INCT and TDEC with semaphore descriptors only, if a switch bit is set; otherwise all operands.</u>

Instruction Slave Store is not required to be cleared. The Stack Slave store is not required to be cleared. Operand Slave Store is cleared of items from segments marked 'Non-slaved' (NS) in the Segment Table. Address Translation Slave Store is not required to be cleared.

    4.  <u>Clear Address Translation Slave Store</u>

Instruction, Stack and Operand Slave Stores of the specified processor are not required to be cleared. This instruction causes the Address Translation Slave Store of the specified processor to be cleared completely. The specified processor remains halted until it receives a Restart signal from the processor which halted it; it will then continue at the instruction it was suspended on. These facilities are essential for multi-processor systems.

4.3.6 Slave Stores whose contents have been invalidated by a peripheral input transfer will normally be cleared by the Activate instruction which

restarts a process waiting to access the store area concerned. Thus during the transfer various slave stores may contain information which is wholly or partly invalid.

Hardware should therefore avoid "writing back" the contents of a slave store to main store, which may be expedient when a process legitimately overwrites part of a slaved location, unless certain that the rest of that location has not been altered by a peripheral transfer since being copied to the slave store. In practice this implies that "writing back" should be restricted to the stack segment, which, if the rule in 4.2.4.5 is observed, will not be involved in I/O activity, or access by another OCP.

4.3.7 It may be necessary to clear slave stores on other occasions, depending on how they are used. For instance when an item read from store is loaded to a slave store it should not be necessary to check its segment APF against ACR when it is subsequently read again from the slave store, provided ACR has not been increased before the second access. An increase in ACR (e.g. accompanying Exit) may therefore necessitate the clearance of those slave stores on access to which ACR is not checked. The stack slave need not be cleared on such occasions since in effect ACR checking is not applied to the stack segment. (see section 4.2.2)

To increase ACR, programmers should only use Exit, Activate, or an explicit write to PSR (Image Store block 0, line 1) and must ensure that the overwriting of PSR which is part of either the standard interrupt mechanism (see Fig. 3 in section 5), or non-standard hardware implementations of the System Call mechanism which do not clear slave stores, does not cause ACR to be increased.

Note that when ACR is decreased, locations which were previously accessible only for reading (and which may be already "slaved") may also become accessible for writing.

## 5  INSTRUCTION SEQUENCE

### 5.1  Normal Sequencing and Jumps

Instructions are processed sequentially until a jump instruction, procedure Call, Exit, Activate, escape or interrupt action is met. The address of the current instruction is held in the program counter (PC) register. Instructions are half word aligned so that addresses held in PC have the format:

| Current code segment | | Current half word | |
| --- | --- | --- | --- |
| 0 | 13 | 14 | 30 |
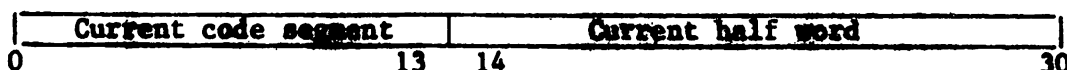
When PC is read (e.g. by Call) or written to (e.g. by Exit) the copy in store has an extra 'bit 31' concatenated so it represents a 32-bit byte address.

PC is automatically updated in the course of each instruction, to point to the next instruction in sequence. Updating in this way is not permitted to cross the upper boundary of the current code segment; nor is an instruction permitted to lie across this boundary.

The contents of PC may be altered to point to an instruction which is not the next in store sequence by a jump (including Call) instruction, relative or absolute, or by Exit, Activate, escape or interrupt.

In certain machines, instructions overlap and this leads to situations in which they may be abandoned, e.g. a jump in a previous instruction is successful. No changes in main store will occur due to the partial implementation of an instruction, however the USE field may be set in the segment or page tables see sections 4.1.3 and 4.1.6.

A relative jump is one whose operand is a literal, and the literal is regarded as a signed integer half word displacement from the address of the current instruction. The target instruction must lie within the current segment. An absolute jump is one whose operand is a 32-bit quantity, addressed directly or indirectly, which is interpreted as a byte address and overwrites PC. A change of segment is allowed. The other mechanisms by which PC may be altered are detailed in the following sections.

### 5.2  Procedure Entry and Exit

5.2.1 Three types of procedure entry are provided:

(a) procedure using the same name space (Jump and Link)
(b) binding procedure using the same name space (escape)
(c) procedure with new name space and extra name access to current space (Call).

The Jump and link instruction provides for case (a) by stacking the address of the next instruction at (SF) before jumping. This link may be transferred to PC by a Jump instruction specifying TOS as operand.

**ICL**

**Product
specification**

**Company
restricted**

**PSD** 2.5.1

**Issue** 6/0

**Sheet** 42

5.2.2 The escape mechanism occurs when an escape descriptor is found or placed
in DR when executing Modify DR or accessing store during instruction
execution. (See Fig. 2). By use of the escape mechanism (which is
automation recognition of the descriptor type) and the Escape exit
instruction it is possible to return to execute the original instruction
with the D bit set in PSR to indicate that the operand is to be accessed
via the descriptor already in DR rather than as indicated by the
instruction address field. The D bit is reset at the end of the
instruction.

The escape mechanism causes a transfer of control to the address contained
in the location pointed at by DR after the PC address of the current
instruction has been stacked as a 32 bit item. The type bits in DR are
changed to give a word vector descriptor with BCI and UBC set so that
parameters for the routine are addressable via DR. At the end of the
escape routine DR may be set with a suitable descriptor and, by use of the
Escape exit instruction, the original instruction may be repeated using
the descriptor supplied by the escape routine. There are other methods of
return since an escape descriptor may occur on jump or data access or
procedure call or exit. If the Escape exit is used after resolution of an
escape descriptor found on a normal procedure exit, it will not have been
possible to pass a parameter in DR for the normal exit.

5.2.3 The procedure call mechanism (see Fig 1) always creates a new local name
space and this is done in two stages.

There is a precall sequence of instructions which stores the old LNB as a
link for procedure exit, creates space on stack front for the PC link and
for a descriptor to linkage tables for the new procedure and then stacks
parameters for the new procedure. LNB is then raised to the beginning of
this new name space, local to the new procedure. "If the Pre-call
instruction is used, the start of the new name space is automatically
aligned so that two-word items can be organised to coincide with two-word
boundaries in the stack segment".

The Call instruction then puts link information for the return to the old
procedure in a standard place in the new name space at (LNB+1, +2)
consisting of the updated PC address, links from PSR, and a descriptor
type. At this point a decision is made as to whether the Call is a system
call or a procedure call. The above operations are standard to both and
the decision is made on the type of descriptor accessed as the Call
operand if indirection is involved. The choice of type is conventionally
made by the loader rather than Alice so that procedure calls are made by
name at the Alice level in order to defer decisions about type of call and
level of execution to the loader; though by suitable choice of operands
Alice can take the choice out of the loader's hands. If the operand is
accessed directly the Call is a procedure call.

If the descriptor left in DR is code, vector or descriptor type a jump is
made to a new procedure. If the descriptor is code type the jump
destination address is the address in DR plus the modifier, if any; if
type 0 or 2 it is the contents of the indirectly addressed location.

If the descriptor is a system call descriptor then a system call interrupt is made.

Of the visible registers only DR, PC and (if used for operand access) SF are affected by the actual Call instruction. However the contents of other registers may be overwritten by the software executed immediately following a System Call interrupt (see [4]).

5.2.4 The Call instruction is used in four ways (see [4]):

Normal-call    with code descriptor or vector descriptor
Inward-call    with system call descriptor
Outward-call   with system call descriptor
Task-call      with system call descriptor

The normal call is to a procedure of the same status (PSR remains unchanged). The return required is therefore of the same sort, i.e. without change of ACR or PRIV.

The inward call is to a more trustworthy procedure i.e. with lower ACR and/or PRIV =1. In order to decrease ACR or set PRIV a hardware interrupt mechanism must be invoked. The system call descriptor in DR contains parameters to index two system call tables. The return can be normal as the original values of ACR and PRIV can be picked up and reloaded with the Exit mechanism.

The outward call is to a less trustworthy procedure. It cannot be a normal call since it is not desired to pass over the same ACR and PRIV and because a special mechanism (inward Return) must be invoked for the return. The return must therefore also be by system call (see [4]).

The task call is used to implement PL/I tasking; return is by another task call.

5.2.5 The Exit instruction is therefore used in two ways: exit with code descriptor - return for normal and inward calls; exit with system call descriptor - return for outward calls.

Figure 2 shows the Exit instruction. For exit using a code descriptor any of ACC, B or DR may be used to pass parameters since the Exit instruction does not alter these registers. However, if the link descriptor on exit was a system call descriptor, decoding software intervenes, corrupting both DR and XNB, so in this case the contents of these registers must be considered undefined (see [4]). The checks on ACR and PRIV are necessary because these are reloaded from the stack and may have been overwritten in error at a level of privilege which would write into the stack, but not into PSR.

**ICL**

**Product
specification**

**Company
restricted**

**PSD**    2.5.1

Issue     6/0

Sheet     44

The values of ACR and PRIV are loaded automatically on exit, but only if
the comparisons of the stacked values with those already in PSR are
satisfactory.  In an emulating machine, E is always restored by the Exit
instruction (or its alien equivalent) and, if different from its previous
value, will cause a switch to or from alien code.  If the other fields of
PSR (PM, CC, ACS, OV) are to be restored on return then this may be done
either on exit by bits in the operand or prior to exit by 'Modify PSR'
instruction (whose operand can be (LNB+1)) which requires no privilege.
Restoration of PSR may be necessary following an interrupt.

<u>Precall Sequence</u> (implemented by software)   (FIG 1)

Stack is at old SF and old LNB.

1.  Store LNB in (SF)   (SF=SF+1)

2.  Adjust SF by 4 words to allocate local name space for link descriptor
and Procedure Linkage Table descriptor, (in case required)

3.  Stack parameters (P words)

4.  Raise LNB (P+5) (LNB=SF-P-5)

"Note.  Steps 1 and 2 can be implemented by the Precall instruction
(PRCL), which will (prior to stacking LNB) increment SF by 0 or 1, so that
LNB is stored in an odd addressed stack location.  The least significant
bit of the stored value of LNB is set to 0 or 1 accordingly".

```
SSN   | ───────────────────── |
      |                       |
Old  LNB─|                       |
      |                       |
      |                       |
      |           .           |
      |                       |
LNB   |                       |
(old SF)|───── Old  SSN/LNB ─────|
      |────────| PSR  ─────────|
      |───────── PC ──────────|
      | PLT descriptor space  |
(LNB+5) |───────────────────── |
      |                       |
      |───── Parameters (P words) ─────|
SF    |                       |
```
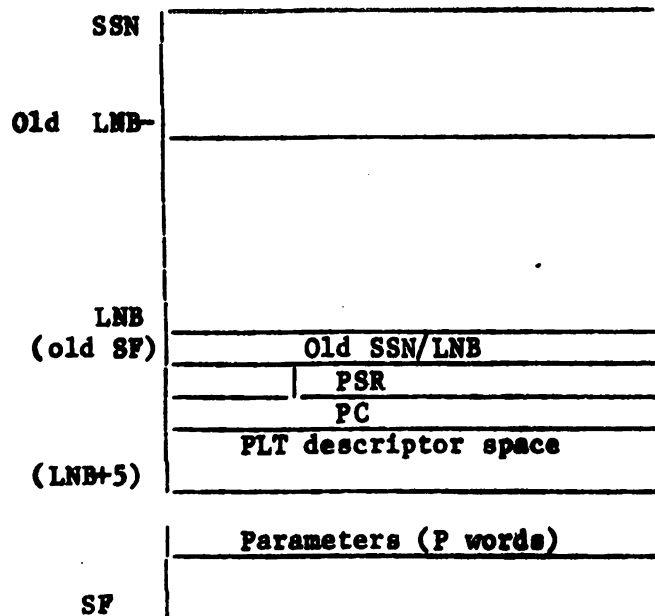
## Call Instruction (implemented by hardware)

1. Load PC, CC, OV, Program mask, ACS, ACR, PRIV, E, and descriptor type to (LNB+1), +2) (but see step 6)

2. If operand is literal or direct obey normal jump procedure, END.

3. If operand indirect, load descriptor to DR (if not already there).

4. If descriptor is type 0 or type 2, operand overwrites PC. END.

5. If descriptor is code type, address from DR (plus modifier if any) overwrites PC. END.

6. If descriptor is escape type do escape jump. (LNB+1, +2) undefined.

7. (Else) do program error interrupt.

Note. ACC, B, DR unaltered for direct procedure calls, else ACC, B unaltered, and DR contains descriptor (Refers to Call instruction only, but see section 5.2.3).


Format of (LNB+1, +2)

| 0 | 7 | 8 | 11 | 12 | 13 | 14 | 15 | 16 | 23 | 24 | 26 | 27 | 28 | 29 | 30 | 31 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| (LNB+1) | DESCRIP-TOR TYPE (3/33) | | ACR | | D | PRIV | OV | E | | PM | | 0 | | 1 | CC | | ACS |
| (LNB+2) | | | | | Address from PC | | | | | | | | | | | |

PC address is that of instruction after the CAll

## Exit Instruction (FIG 2)

Stack as at end of call instruction.

1. Examine bits 0-7 of (LNB+1)

2. If code descriptor type go to 6.

3. If Escape descriptor type transfer (LNB+2) to DR and do escape jump.

4. If system call type copy PSR and PC to DR, and go to Step 16 of Figure 5.

5. (Else) do program error interrupt.

6. Compare bits 8-11, 13 of (LNB+1) with values of ACR and PRIV in hardware registers. If of greater privilege or lower protection go to 5 else load ACR and PRIV.

7. Overwrite other fields of PSR as specified by operand. Overwrite E.

8. SF = LNB

9. LNB = (SF) (SF not altered)     Note: Sequence 1,2,6-11 does not change DR

10. Set PC = (old LNB+2).

10A SF = SF -1 if Bit 31 of (SF) = 1

11. If ACR increased in Step 6, and EP bit set in SSR, interrupt (unless EP masked) otherwise execute instruction at PC. If E = 1, and EM has correct value, emulate.


## Escape Jump

An escape jump occurs in the middle of an instruction with DR = escape descriptor.

1. Stack PC (SF = SF+1) (PC is address of current instruction)

2. Set PC = Contents of 32 bit location addressed by DR.

3. Set DR(0-7) to type 0 (vector) descriptor, size 32, unscaled bound check inhibited.

## Escape Exit

1. Unstack PC (SF = SF-1; PC = (SF))

2. Set D bit in PSR.  END

Note:  DR is unchanged.

## Escape Descriptor

| Address |
|---------|

→

| Addr. of escape routine |
|-------------------------|
| Parameters for escape routine |
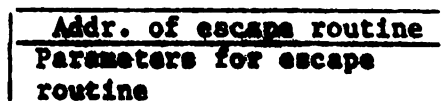
## 5.3 Interrupt mechanism

5.3.1 The interrupt system allows a change of state within a virtual machine by either a forced procedure call using the existing stack or by the use of a new stack. Interrupt conditions may arise asynchronously with current processing, when they will cause interruption at the earliest opportunity either between instructions or, in the case of long instructions, in mid-instruction; or they may arise synchronously out of the operation of the current instruction. In the latter case the instruction may be completed normally or abnormally, or terminated in such a way that it can be restarted and interruption then occurs (except in the case of program errors on pipeline processors, when, in order to bring the pipeline to rest in an orderly way, the interrupt may be deferred until further instructions have been obeyed). Interrupts may be masked by registers in PSR and SSR in which case they may be ignored, remain pending, or cause system errors.

Interrupts occur in one of 12 classes:

| Interrupt class | Priority | Masking Rule | Synchronous Asynchronous | Stack Switched /Not Switched | SW Notes N |
|---|---|---|---|---|---|
| 1. System error | 1 (if Asynchronous) | - | A/S | SW | a,d,e |
| 2. External | 2 | 2 | A | SW | |
| 3. Multiprocessor | 3 | 2 | A | SW | |
| 4. Peripheral | 4 | 2 | A | SW | |
| 5. Virtual store | | 1 | S | SW | a |
| 6. Interval timer | 5 | 2A | A | SW | c |
| 7. Program error | | 1 | S | SW | a,b |
| 8. System call | | 1 | S | N | a |
| 9. Out | | 1 | S | SW | a |
| 10. Extracodes | | 1 | S | N | a |
| 11. Event pending | | 3 | S | SW | a |
| 12. Instruction counter | 6 | 2A | A | N | |

Priority          1 is high

Masking rules:

1. If masked, treated as system error

2. If masked, remains pending to system
   See section 5.3.7

2A If masked, remains pending to process
   See section 5.3.7.

3. If masked, ignored

Notes: (a) Only one of these conditions can arise from the action of current instruction (see 5.3.7, 5.3.8.)

(b) Computational conditions are ignored if masked by program mask bits in PSR, otherwise treated as program errors.

(c) Other time clock interrupts may be admitted in peripheral or external classes, e.g. real-time clock.

(d) Some system error conditions are synchronous, some asynchronous.

(e) The masking rules for system error interrupts are machine dependent.

5.3.2. A definition of each of the above interrupt classes is given here or in some other section of the primitive interface definition.

System errors – hardware errors defined in hardware manual, interrupts 5,7,8,9 and 10 occurring and being masked.

External – interrupts from devices not having a connection to store including real-time clock (e.g. 360 type "Write control" facility). See hardware manual.

Multiprocessor– Interrupts between processors sharing the same store e.g. one processor telling another to reschedule. Similar to 360 remote intitial program load. See hardware manual.

Peripheral – Interrupts from peripheral controllers via SAC. See image store. (Section 3).

Virtual Store – Access to non-available segment or page or outside virtual memory (but not segment protection (ACR) faults which are program errors). See section 4.

Timer – Interrupt when interval timer guard bit non-zero See section 3.2.5

Program Error – Interrupt due to bound check, illegal instruction etc. See section 7.

System Call    - Interrupt due to system call descriptor on Call or Exit.
See Section 7.

Out           - A means of causing an interrupt by software.

Extracode     - Interrupt to obey by software an assigned function not
available in hardware.

Event Pending - Interrupt before the next instruction if:

(a) When executing Exit (Section 8.1.2.9) with a code
descriptor causing ACR to be increased, the EP bit
set in SSR; or

(b) When executing Activate (Section 9.2.2), bit 31 of
Word 0 of the operand is a 1, if II is set, the
interrupt will occur before attempting completion of
the incomplete instruction.
When executing Activate the EP bit in SSR is ignored;
masking depends on the mask bit in the new SSR.

The interruption does not clear the EP bit in SSR.

Instruction    - Interrupt when instruction counter guard bit non-zero.
Counter        PC points to the successor to the last instruction
completed, i.e. counted.

5.3.3 Masking of interrupts occurs via parts of PSR and SSR and, since these registers are of significance in interrupts, their formats are given here as well as in Section 3.

| | | ACR | D | P R I V | O V | E | PROGRAM MASK | | CC | ACS |
|---|---|---|---|---|---|---|---|---|---|---|
PSR

0    7 8   11 12 13 14 15 16   23 24   27 28 29 30 31

| | I I | R A M | | PI | EP | D G W | I S R | I S O | | EM | INTERRUPT MASK |
|---|---|---|---|---|---|---|---|---|---|---|---|
SSR

0    1   2 3   4   5   6   7   8   9   10 11 12   15 16   31
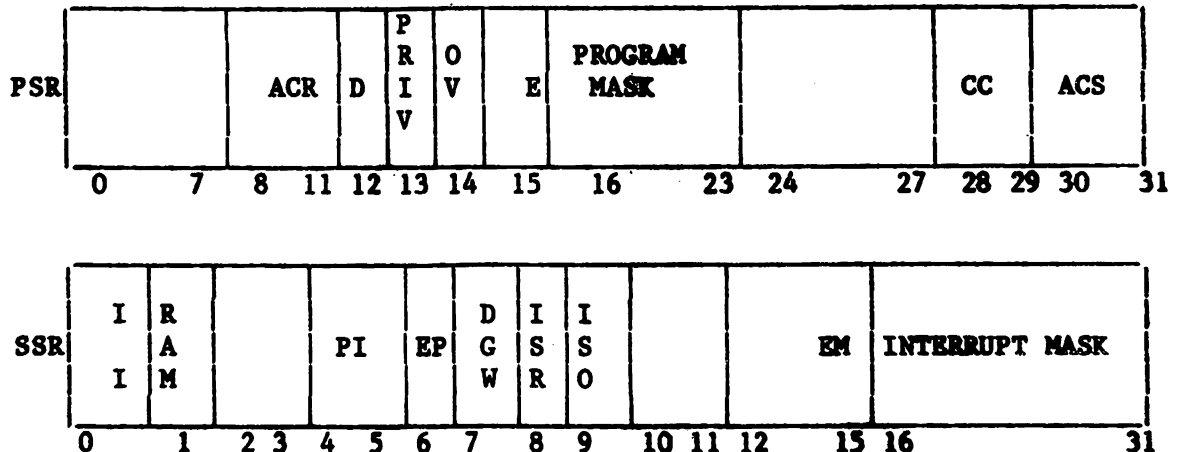
| ACR | Store access control register |
|---|---|
| D | Next operand is DR (escape exit) |
| OV | ACC or B overflow |
| E | Emulate |
| ACS | ACC size (32/64/128) |
| CC | Condition code (0/1/2/3) |
| PRIV | Privilege for accessing hardware registers (see section 3.3 et.seq) |
| RAM | Real address mode |
| EM | Emulation mode |
| II | Instruction Incomplete |
| PI | Processor Identifier |
| EP | Event Pending (See 5.3.2) |
| DGW | Diagnostic write |
| ISR | Image store read |
| ISO | ISO numeric mode |
| EM | Emulation mode |

The interrupt mask has one bit as a mask for each of the 12 classes of interrupt. Interrupt is masked if the bit = 1.

The program mask has 1 bit for each of a number of computational conditions.

Bit 0  Floating point overflow
    1  Floating point underflow
    2  Fixed overflow
    3  Decimal overflow
    4  Zero divide
    5  Bound check interrupt
    6  Size
    7  B overflow

The program mask can be altered by the non-privileged Modify PSR instruction. When one of these conditions arises, the instruction is

completed as described in Section 7.5 (the manner of completion may depend on the setting of the mask bit, and some results may be undefined); if the mask bit is 1, the condition is ignored, but if the mask bit is 0 a program error interrupt occurs.

5.3.4 Interruption is a hardware mechanism. The action of an interrupt is to terminate the current state in such a way that it can later be restarted at the same point and under the same conditions; and to initiate a new state at a point and in an environment suitable to the class of interrupt with sufficient interrupt parameters to allow processing to proceed.

The manner of termination depends on the state to be initiated. Each class of interrupt has an entry in a table call the Interrupt Steering table (IST). In a multiprocessor system each processor has its own IST. Each IST entry contains values to be loaded to hardware registers for the interrupt state to be initiated. The tables are in fixed segments, numbers 8192 onwards, and the entry for interrupt class n is at words.

| | |
|---|---|
| 8(n-1) | SSN/LNB new stack segment if stack switched |
| 8(n-1)+1 | PSR load primarily for change of ACR and PRIV |
| 8(n-1)+2 | PC address of new instruction to be obeyed |
| 8(n-1)+3 | SSR load primarily for interrupt mask |
| 8(n-1)+4 | SSN/SF defines top of new stack |
| 8(n-1)+5 | IT to time hardware - intimate software |
| 8(n-1)+6 | IC (if provided). |
| 8(n-1)+7 | CTB defines new Cross-reference Table pointer. |

The hardware must check first whether the stack is to be switched or not in order to determine the method of terminating the interrupted procedure. For system call interrupts, a different format of IST is used. This is described in section 5.3.15. The consequences of having inconsistent values of SSN in words 8(n-1) and 8(n-1)+4 are undefined.

5.3.5 When the stack is not switched then the interrupt is a forced procedure call to a more trustworthy procedure, like the system call. The return will therefore be a normal procedure exit (also restoring the non-privileged fields of PSR) and the stack must contain the same parameters as for a system or procedure call. Steps 1-7 of figure 3 achieve this. These parameters are already on the stack for a system call interrupt which therefore misses them out. The parameter for system call is in the descriptor in DR. The new environment for the interrupt decoder is provided by loading PC and PSR. The sequence requires that IST is locked in core. Space must be available for the parameters in front of the SF. If this space is not available the sequence is halted and a virtual store interrupt is begun.
(Format Fig. 4).

5.3.6 When the stack is switched then a new procedure is begun on a different stack and the present stack must be preserved for restart. This is achieved by dumping the registers in the bottom locations of segment (SSN+1) (format Fig. 4). The use and size of the dump area outside 16 locations is given in the hardware manual. It is therefore a *requirement that these locations be in core when a procedure is active. The new stack whose segment number is indicated by the first word of the IST entry is

now activated by creating a name space and by putting parameters for the interrupt in this space. It is therefore a *requirement that this stack has sufficient space in front of SF for these parameters, and that the locations they will occupy are present in main store.

* If the requirement is not met a System error interrupt is attempted. Should this fail implementation - defined error actions are invoked.

In order to restart a procedure which has been dumped a normal Exit will not do. The privileged Activate instruction (section 9) is used.

5.3.7 Two interrupts may, in practice or conceptually, occur at the same time. The method of dealing with interrupts ensures that only one interrupt is recorded and handled in general. Where an instruction would produce multiple (synchronous) interrupts then only the first one to occur in hardware terms is taken. In effect the other interrupts do not occur because the operation terminates on recognition of the first interrupt.

An instruction can therefore only produce one interrupt. It is still possible for multiple interrupts to occur by the conincidence of an asychronous interrupt. In these cases the asynchronous interrupt remains pending and the synchronous one is taken. Where multiple asynchronous interrupts only occur then that one with highest priority is taken first, the others remain pending.

The pending status may be recorded by the hardware in an implementation defined image store location, in which case the interrupt is held pending to the system (e.g. a peripheral interrupt); or it may be recorded in a guard bit in the associated register, in which case the interrupt is held pending to a particular process (e.g. interval timer interrupts).

Interrupts in Classes 2,3, and 4 (External, Multiprocessor, Peripheral) usually arise from occurrences outside the interrupted processor, and in a multiprocessor system may affect more than one processor. Normally System Error interrupts (Class 1) will only affect the processor concerned but in multiprocessor systems the possibility of more widespread effects is not ruled out. Other classes of interrupt will only affect a single processor - in the case of IT or IC conditions (Classes 6 and 12) held pending, the processor interrupted will be the one in which the process concerned is running when the removal of the mask bit is detected.

<u>Interrupt Accepted (class n)</u>  (Figure 3)

1. If n = 1 - 7, 9, 11 go to 11.

2. If n = 8, go to Step 1, Figure 5.

3. Check 7 words available in front of SF. If not do virtual store interrupt.

4. Stack SSN/LNB (bits 30, 31 = 0).

5. Stack PC as code descriptor, bits 8-31 as in Fig 1.

6. Advance SF by 2 words and then stack parameter (1 word).

7. Set LNB = SF - 6.

8. Load bits 0-29 of (8(n-1)+1) to bits 0-29 of PSR (i.e. ACS not altered but PM is overwritten).

9. Load (8(n-1)+2) to PC.

10. Obey instruction at PC. END.

11. Dump registers to words 0-15 of segment (SSN+1), and, if II=1, additional information, as required by Activate, to succeeding locations

12. Save old SSN.

13. Load (8(n-1)) to SSN and LNB

14. Do actions at 8,9, (but alter ACS when overwriting PSR)

15. Load (8(n-1)+3) to SSR

16. Load (8(n-1)+4) to SF

17. Load (8(n-1)+5) to interval timer .

18. Load (8(n-1)+6) to IC

18A Load (8(n-1)+7) to CTB

19. Stack interrupt parameter

20. If n=7, store address of erring instruction in word 16 of old segment (SSN+1)

21 Stack old SSN (in bits 0-13; bits 14-31 = 0).

22. Obey instruction at PC. END.

| Stack Switching interrupts | Non-stack-switching interrupts |
|---|---|
| n = 1 System error | n = 8 System call |
| 2 External | 10 Extracodes |
| 3 Multiprocessor | 12 Instruction counter |
| 4 Peripheral | |
| 5 Virtual store | |
| 6 Interval timer | |
| 7 Program error | |
| 9 Out | |
| 11 Event pending | |

## STACK FORMAT - STACK NOT SWITCHED

SSN ---|

LNB----|

| Desc. type | PSR |
|---|---|
| PC | |
| Unused | |
| Unused | |
| Parameter | |

SF----|

---

## STACK FORMAT - STACK SWITCHED

New SSN-|

LNB----|

| |
|---|
| Parameter |
| Previous $SSNx2^{18}$ |
| |

SF-----|

Hardware dump area (words 0-17 of segment (SSN+1))

Old (SSN+1)/

<u>FIG 4</u>

| | | |
|---|---|---|
| 0 | SSN | LNB |
| 1 | PSR | |
| 2 | PC | |
| 3 | SSR | |
| 4 | SSN | SF |
| 5 | IT | |
| 6 | IC | |
| 7 | CTB | |
| 8 | XNB | |
| 9 | B | |
| 10 | DRO | |
| 11 | DR1 | |
| 12 | AO | |
| 13 | A1 | |
| 14 | A2 | |
| 15 | A3 | |
| 16 | * | |
| | Implementation - dependent information | |

\* Address of erring instruction (program error interrupt only. Otherwise undefined).

5.3.8 For a virtual store interrupt it is necessary to switch stacks since a further virtual store interrupt due to stack expansion could not be allowed. The faulting address is found at TOS with an identifier.

If stack expansion does cause a virtual store interrupt condition on a non-stack-switching interrupt (except system call) then the interrupt is converted to a virtual store interrupt. This condition cannot arise in connection with other interrupts - classes 1-7, 9 and 11 switch stack and on a system call the stack does not expand further. The source interrupt class is recorded for handling after dealing with the virtual store condition if stack expansion is then permitted and an interrupt identifier is stored in the hardware dump area at the time of interrupt conversion.

5.3.9 Except in the case of Extracode (class 10) interrupts, the value of PC stored on interrupt is the address of the instruction to which control must be returned after dealing with the interrupt. For Extracode interrupts, PC indicates the address of the instruction causing the interrupt, and software is required to update PC prior to returning to the interrupted process after dealing with this interrupt.

When interruption occurs in mid-instruction, as in the case of a virtual store condition arising in the middle of a store-to-store instruction, the II bit in SSRR is set to indicate that on resumption of the process the instruction indicated by the dumped PC is not started from scratch. II may also be set to indicate incompletion of the interrupt mechanism in the cases where non-stack-switching interrupts are converted to virtual store interrupts, or to indicate another outstanding interrupt condition in conjunction with a program error interrupt (see 5.3.10 below). II is inspected by the Activate instruction, and if II is set it is assumed that additional implementation-defined information, used by Activate, is held in segment (SSN + 1). When II is not set the instruction indicated by PC is executed ab initio.

In the case of the virtual store interrupt caused by a jump to an address in a non-available segment or beyond the segment limit, PC may point either to the jump instruction (probably with II set) or to the destination instruction - this is a matter of implementation.

Mechanisms will be particular to each machine of the range and will be given in the hardware manual since each design may be different. Thus there are the possibilities of complete checking (where necessary) at the beginning of an instruction so that partially completed instructions never occur, or of initialising after a virtual store interrupt condition in some cases (e.g. multi-word writes) but not in others (e.g. store/store) or of making all such conditions cause partially completed instructions. The matter is linked with hardware reloading of the address translation slave store (section 4.3).

5.3.10 The precision with which interrupts are handled is allowed to be variable over the range. Where an instruction would produce multiple interrupts then the hardware defines which interrupt is actually taken. In order to allow for the properties of pipe-lined processors, a program error interrupt may occur a few instructions later than the one which caused the error, because parts of the pipeline may be executing later instructions at the time the error is detected. On pipelined processors

the pipeline will come to rest in an orderly manner, i.e. all instructions up to the point of interruption will be completed, and the dumped PC will indicate the instruction at which the process should resume. II may be set if interruption occurred in mid-instruction, e.g. if in the course of coming to rest after detection of a program error somewhere in the pipeline, the processor starts to execute a string instruction which meets a virtual store interrupt condition part-way through, the pipeline will be closed down, and the program error interrupt will occur, at that point; if the process is subsequently re-activated the virtual store interrupt will be taken.

The address of the instruction causing the error will be placed in word 16 of the dump (segment (SSN +1)); the contents of this word are undefined for other stack-switching interrupts.

In coming to rest after detecting a program error the processor may execute jump instructions, but will not go beyond a Call instruction. If for some reason the pipeline cannot be brought to rest in an orderly manner (e.g. because another program error condition is detected - in this case the parameters will refer to the first error) a bit will be set in the interrupt parameters to indicate that the process cannot be resumed at the instruction indicated by PC.

The method of completing the instruction on which the actual error occurred, when not defined elsewhere in this document, is implementation-dependent. The instruction is always completed, so even on non-pipelined processors the address in PC will not be the same as that of the erring instruction. Floating-point underflow is the typical case in which the result is completely, and sensibly, defined, and in which the interrupt may be required only to log the occurrence without affecting the process.

5.3.11 A parameter is stacked, on the new stack if stacks are switched or else on the old stack, giving further information about the interrupt.

| | |
|---|---|
| System error | 32 bits of hardware identifiers - to be defined |
| External | Implementation defined. Also distinguishes RTC interrupts. |
| Multiprocessor | ) Module number (see [6] in image store |
| | ) address format (see section |
| Peripheral | ) 3.3.3) |
| Virtual store | See section 7.3.2 |
| Interval timer | Undefined. |
| Program error | Program error identifier - 2 bytes. See section 7.4.1 |
| System call | None, parameter in DR. |
| Out | 32-bit operand of Out instruction. |

**ICL**

**Product specification**

**Company restricted**

**PSD** 2.5.1

Issue 6/0

Sheet 57

Extracode                 Parameters are machine dependent.

Event pending             Undefined.

Instruction Counter       Value of IC at interrupt.

5.3.12 The hardware interrupt mechanism accessess IST, segment (SSN + 1), PSR, SSR, etc., with effective ACR=0 and PRIV=1, i.e. these segments and registers are protected against erroneous access but the protection mechanism cannot inhibit the interrupt action.

5.3.13 The interrupt mechanisms defined in this document are not required to clear slave stores, and programmers must therefore ensure that they are not used directly to increase ACR (see section 4.3).

5.3.14 When emulating alien code, if a stack-switching or non-stack-switching interrupt (as opposed to an emulated interrupt) loads a new PSR having E=0, a switch to NR mode occurs.

5.3.15 System Call interrupts are handled differently to other interrupts. The System Call Descriptor (Type 3 subtype 35) contains two parameters, i and j which are used to access the $j^{th}$ entry of the $i^{th}$ System Call Table (SCT) which contains the new value of PSR and either the start address of the called procedure or a pointer to a location containing the start address of the called procedure. This entry also contains parameters defining the maximum ACR level of the calling procedure and the minimum number of stacked parameters.

The format of the system call descriptor is

```
     0          7 8              31
    |  11100011  |        i        |
    |            |        j         |
     32                          63
```
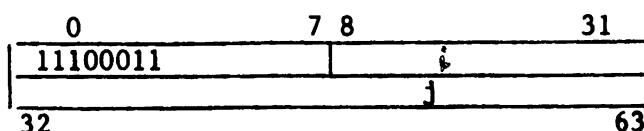
where i is the entry in the SCT Index
      j is the entry in the $i^{th}$ SCT

The format of the IST entry for System Call (words 56 to 63) is

```
              0       7 8          29, 30/31
Word 56 (0)  |//////////////////////////////////|
     57 (1)  |///////////|   New PSR   |///////|  Used for exception
     58 (2)  |        New PC                    |  conditions
  59-61 (3-5)|//////////////////////////////////|
             |//////////////////////////////////|
     62 (6)  |  00110000  |         L            |  Descriptor Defining
     63 (7)  |            A            |///////|   SCT Index
```
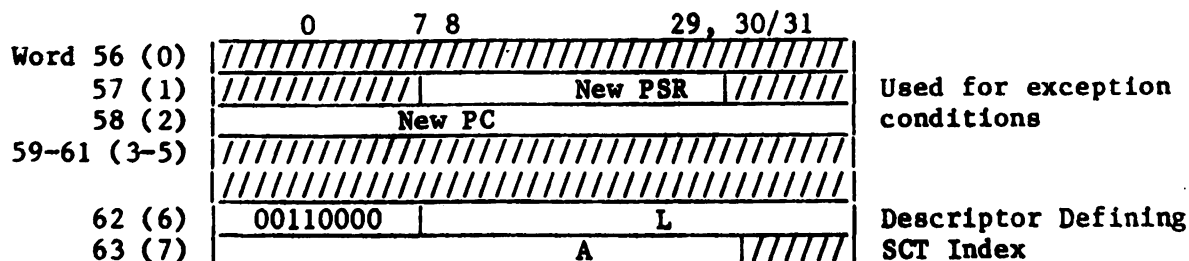
The shaded areas are not used.

Where A is the base word address of the SCT Index.
      L is the number of extries (double words) in the SCT Index.
Words 1 & 2 give the entry point for the exception condition routine

The SCT Index is a table of descriptors, each defining a System Call Table. The Format of each entry in the SCT Index is :-

```
                0       7 8                              31
word A+21      | 00111000 |              M                |
word A+21+1    |               B              | ////// |
```

Where B is the base word address of the $i^{th}$ SCT. M is the number of entries in the $i^{th}$ SCT.

The System Call Table contains 4-word entries, each defining the entry conditions for a procedure. The format of each entry is

```
                0       7 8      11 12   24   31
word B+4j+1    | 1 |//////| K      |   Count    |
     B+4j+1    |   P    |   New PSR       |    |
     B+4j+2    | Type   |        N             |   ) Descriptor
     B+4j+3    |              C               |   )
```

Where Bit 0 of the first word = 1 if hardware action required, and K is the System Call Access Key of the called procedure. (To be checked against ACR of the calling procedure).

The 2nd and 3rd words form a descriptor which is either a Code descriptor pointing at the called procedure or a Vector descriptor pointing at a table, whose entry is a pointer to the called procedure.

Count is a 20 bit binary number which is incremented by 1 each time the System Call is used. This is implemented at AML1. The field is ignored at AML0.

P is the minimum difference in words between SF and LNB. This is implemented at AML1. The field is ignored at AML0.

The steps of the decode routine are given in Figure 5.

System Call Decode Routine (figure 5)

It is assumed that a double word register, S, exists to hold intermediate descriptors. Other letters refer to formats defined in 5.3.15.

1.  Load words 6 and 7 of System Call Entry in IST to Reg. S.
    (IST words 62 and 63).

2.  Check that descriptor in S is type 0, size 2 words and that USC, BCI
    are not set. If fail go to step 16.

3.  Check that i is less than than L.  If fail go to step 16.

4.  Load descriptor in the $i^{th}$ entry of the SCT Index to Reg. S.
    (Words A+21 and A+21+1).

5.  Check that descriptor in S is type 0, size 4 words and that USC,
    BCI are not set.  If fail go to step 16.

6. Check that j is less than M. If fail go to step 16.

7. Load the first word of the $j^{th}$ entry in the SCT to Reg S.
   (Word B+4j).

8. Check that hardware action is permitted (Bit 0 of S=1) and that
   current ACR is not less than K. If fail go to step 16.

8a Add 1 to bits 12-31 of Reg S and write them back to first word
   of the $j^{th}$ entry in the SCT (words B+4j) (not AMLO)

9. Check that SF-LNB is not less than P. If fail go to step 16. (not
   AMLO)

10. Overwrite PSR (bits 8-29) with the new value in second word of jth
    entry in the SCT (bits 8-29 of B+4j+1).

11. Load next two words of $j^{th}$ entry in the SCT to DR.
    (Words B+4j+2 and B+4j+3).

12. If descriptor in DR is type 0, size 1 word: load PC from location
    addressed by C. Obey instruction at PC. END.

13. If descriptor in DR is type 0, size 1 word; load PC from
    location addressed by C. Obey instruction at PC. END.

14. If descriptor in DR is type 0, size 2 words or type 2; load PC
    from location addressed by C+1. Obey instruction at PC. END.

15. Initiate System Error (descriptor incorrect type) END.

16. Overwrite PSR (bits 8-29) with bits 8-29 of word 1 of System
    Call entry in the IST. (IST word 57)

17. Load PC from word 2 of System Call entry of IST.
    (IST word 58).

18. Obey instruction at PC. END.

# 6 INSTRUCTION AND DESCRIPTOR FORMATS

## 6.1 Instruction Formats

Instructions are either 16 or 32 bits in length. There are three formats: primary, secondary (store-to-store) and tertiary. In all formats the first 7 bits specify the function code, the remaining 9 or 25 bits the operand. The format is determined from the function code, and the instruction length from the operand part of the instruction.

```
16 bit  |    Function    |    Operand        |
              7                9
```

```
32 bit  |    Function    |    Operand             |
              7                25
```
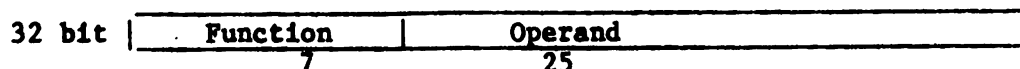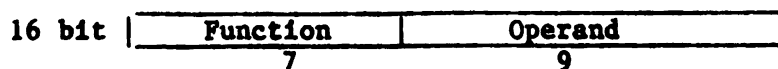
In general, instruction may have any of the types of operand allowed by the format class to which they belong, though for particular functions there may be restrictions which are listed under the individual instruction descriptions. Some instructions do not use the operand, though the operand field determines the instruction length. In such cases literal operand (zero) must be specified.

### Function Decode

The 7-bit function number is decoded in such a way as to provide:

104 primary format instructions (48 computational and 56 miscellaneous).
 16 secondary format instructions (store-to-store).
  8 tertiary format instructions (jumps).
 (see table of allotted function codes in Appendix 1)

### Operand decode

Primary format:

```
16 bit        |  k  |      n        |    k = 0,1,2
                 2          7
```
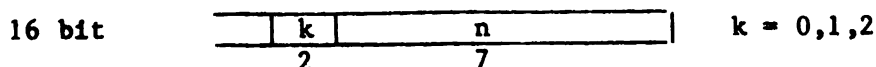
#### k operand

0   n ( 7 bit signed literal)
1   (LNB+n) (Direct access to local name space)
                (n unsigned)
2   ((LNB+n)) (Indirect access via descriptor in
                local name space) (n unsigned)
3   implies further decode

```
16 bit       |  k=3 | k' | k" |    *    |   k" = 6,7
                 2     2    3      2
```
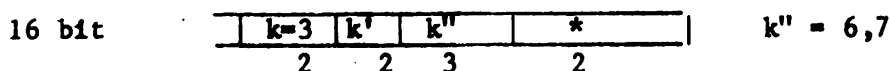
* = Reserved

32 bit     | k=3 | k' | k" | N |     k" = 0-5

    2    2    3     18

Operands for different values of k', k".

| | Direct | Indirect | | |
|---|---|---|---|---|
| k'= | 0 | 1 (Desc. in DR, modified) | 2 (Desc. in store) | 3 (Desc. in store, modified) |
| k"=0 | N(signed literal) | (DR+N) | *(IS location N) | *(IS location B) |
| 1 | Unassigned | | | |
| 2 | (LNB+N) | (DR+(LNB+N)) | ((LNB+N)) | ((LNB+N)+B) |
| 3 | (XNB+N) | (DR+(XNB+N)) | ((XNB+N)) | ((XNB+N)+B) |
| 4 | (PCO+N) | (DR+(PO+N))) | ((PO+N)) | ((PO+N)+B) |
| 5 | (CTB+N) | (DR+(CTB+N)) | ((CTB+N)) | ((CTB+N)+B) |
| ( 6 | TOS | (DR+TOS) | (TOS) | (TOS+B) |
| ( 7 | B | *(B+N) ** | (DR) | (DR+B) |

16 bit (left of rows ( 6, ( 7)

*Classed as Direct Address Form
(**) 32 bit format. This operand form is unassigned at AMLO

## Notes

(i)     Unassigned operand forms cause program error interrupt.

(ii)     PC contains address of current instruction.

(iii)     B=Contents of B register, TOS=Top-of-Stack item

(iv)     ((LNB+N)+B), (TOS+B) and (DR+B) indicate items pointed to by descriptors held respectively in LNB+N, in the top 2 words of stack, and in DR; '+B' indicates that the address in the descriptor in each case is modified by B.

(v)     N is unsigned (positive) except when a literal, or when added to PC (k"=4)

(vi)     For jump instructions the operand, interpreted as a byte address whose least significant bit is ignored, overwrites PC, except when it is a literal in which case it is treated as a signed quantity (number of half-words) and added to the contents of PC.

For the 'Call' instruction the indirect forms may use Code descriptors, in which case the address from the descriptor itself, modified if the instruction form specifies it, overwrites PC.

The use of System Call (by the 'Call' and 'Exit' instructions) and Escape descriptors (by any instruction) is described in Section 5.

Operand for k" = 1

Secondary format (used for store-to-store operations):

16 bit

| | h | q=0 | n | |
|---|---|---|---|---|
| | 1 | 1 | 7 | |

32 bit

| | h | q=1 | n | Mask | Literal/Filler |
|---|---|---|---|---|---|
| | 1 | 1 | 7 | 8 | 8 |
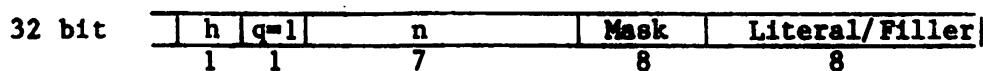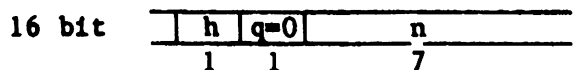
Interpretation of h :

h = 0  Number of bytes = n + 1
h = 1  Number of bytes = Length of destination string

Interpretation of q :

q = 0  16 - bit instruction
q = 1  32 - bit instruction

For a detailed explanation of how these fields are used, refer to section 8.3 on store-to-store operations.

Tertiary format (used only for conditional jump instructions)

16 bit

| | M | k''' | 00 | |
|---|---|---|---|---|
| | 4 | 3 | 2 | |

32 bit

| | M | k''' | N | |
|---|---|---|---|---|
| | 4 | 3 | 18 | |
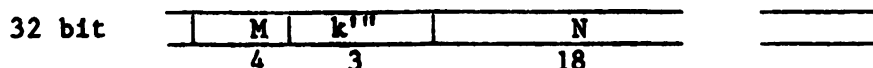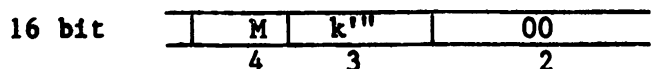
k''' provides the following operand types:

| | | | |
|---|---|---|---|
| 0. N (literal) | 1. (DR+N) | ) | |
| 2. (LNB+N) | 3. (XNB+N) | ) | 32 bit |
| 4. (PO+N) | 5. (CTB+N) | ) | |
| 6. (DR) | 7. (DR+B) | 16 bit | |

(see note (vi) above)

M = 4 bit mask field (see description of Jump on CC and Jump on Arithmetic instructions in section 8.1.2)

## 6.2 Descriptor Formats

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

Descriptor types are distinguished by their more significant 32 bits. The general form of descriptor is as follows:

| 2 | 3 | 1 | 1 | 1 | 24 |
|------|------|---|-----|-----|--------------|
| Type | Size | S | USC | BCI | Bound/Length |
| Byte Address (32 bits) | | | | | |

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

Type 0 = Vector descriptors
Size     The size of the addressed item in store. Permitted sizes, and the corresponding size codes, are as follows:

| Size (bits) | Code | |
|-------------|------|---------|
| 1 | 0 | |
| 8 | 3 | |
| 16 | 4 | not AMLO |
| 32 | 5 | |
| 64 | 6 | |
| 128 | 7 | |

When the size is 32, 64 or 128 bits, the two least significant bits of the byte address, after modification, if any, are ignored – i.e. 1, 2, and 4 word items are made to start on word boundaries. Use of other values will cause program errors as indicated in Section 8.
When size is 16 bits, the least significant bit of the byte address, after modification, is ignored.
        Size code 4 is not available at AMLO.

S        Signed. If set to 1, then

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

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

The S field is ignored at AMLO.

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

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

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

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

Type 1  = String descriptors

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

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

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

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

Type 2  = Descriptor descriptors

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

Type 3   - Miscellaneous
         Bits 2-7 (Size, S, USC, and BCI) define a subtype number.

Subtypes (numbered decimally)

32,33 Code (Bounded, Unbounded)

        Code descriptors may be used to point to the destination
instructions of Jump, Call and Exit instructions. Bits 32-63 contain the
byte address of the first byte of the destination instruction - bit 63 is
ingored as instructions are halfword aligned. Any modifier added to the
address is multiplied by 2 before addition. If sub-type 32, bits 8-31
contain a bound which is used to check the modifier, if any, in exactly
the same way as for Type 0 and Type 2 descriptors. If sub-type 33, bits
8-31 may contain the identity of a microcode subroutine which may be
entered after PC is set. If the microcode subroutine does not exist or
if an error is encountered within the routines a jump is made to the
instruction addressed by PC. A description of the defined microcode
subroutines is given in Appendix 3.

35 System Call

        Bits 8-31 usually contain an entry displacement to index a System
Call Index Table. Bits 32-63 usually contain an entry displacement to
Index the System Call Table indicated by the descriptor accessed for the
System Call Index Table. System Call descriptors are only used by the Call
and Exit instructions - in the latter case as 'link descriptor'. Their use
is described in section 5 and in [4].

37 Escape

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

40,41 Semaphores (Bounded, Unbounded).

        Semaphore descriptors are used to point to semaphore locations in
store. The format is similar to a Type 0, size code 5 descriptor and the
modification rules are the same. The descriptor is restricted to use with
INCT and TDEC instructions. The effect of use with other instructions is
undefined (but a program error is preferred). The descriptor must not be
used to access the stack segment.
        Access to the word pointed at by the descriptor is forced by
hardware to bypass slave storage and is implemented by a Read Hold, Write
Hold combination so as to prevent access to the store location while the
word is modified.
        If slave storage is present, use of this descriptor must clear
the operand slave store of items from segments marked non-slaved (NS).

**63 Null**

     A null descriptor is used to provide the NIL option when a descriptor is used as an optional parameter for a procedure. It is detected by the VALIDATE instruction and is invalid with all other instruction. This facility is part of XL1.

6.3   Operand Addressing and Alignment

6.3.1 General Principles

6.3.1.1 Operand Length (Primary and Tertiary Formats)

The 'operand length' of an instruction is the number of bits of the
operand required by the instruction for its operation.  The operand length
is defined in a number of different ways, depending upon the particular
instruction.

a)   Explicitly defined, e.g. 32 bits for Scale, Load B etc.

b)   Implicitly defined equal to ACS for most instruction operating on ACC,
e.g. Add, Subtract.

c)   Implicitly defined equal to half ACS for Floating Divide Double
(RDVD), Store Upper Half (STUH).

d)   Not applicable for instructions such as IDLE, in which case a literal
must be used.

The operand length is not necessarily the same as the length of the
addressed item, in particular when the addressed item is accessed via a
descriptor or is the contents of Register B.  When there is a difference,
extension or truncation will be necessary or not allowed.

The five possibilities are:

a)   Z - Extend with left hand zeros.

b)   S - Sign extend with left hand zeros or ones.

c)   TZ  Truncate on the left and check for all zros.

d)   TS  Truncate on the left and check that all the truncated bits are the
same as the most significant bit of the non-truncated data.

e)   X - Addressed item too large.

In the case of e), or if the check fails in case c) or d), a size
interrupt occurs and the operation is suppressed.  If the interrupt
condition is masked, the operation is not suppressed and any truncated
bits are ignored.

For instructions which read from the addressed item the rules are:

| Addressed Item | Action | | | |
| --- | --- | --- | --- | --- |
| | > Operand length | | < Operand length | |
| | AMLO | AML1 | AMLO | AML1 |
| Register B | Not applicable | | Z | Z |
| String descriptor | * | * | Z+ | Z+ |
| Vector descriptor, S=0 | X | X | Z | Z |
| Vector descriptor, S=1 | X | X | Z | S |

*Causes a non-maskable program error (descriptor) interrupt.
+Except that in the special case of a zero length value, a non-maskable program error interrupt, (descriptor) occurs.


For instructions which write to the addressed item the rules are:

| Addressed Item | Action | | | |
| --- | --- | --- | --- | --- |
| | > Operand length | | < Operand length | |
| | AMLO | AML1 | AMLO | AML1 |
| Register B | Not applicable | | TZ | TZ |
| String descriptor | * | * | TZ | TZ |
| Vector descriptor, S=0 | Z | Z | TZ | TZ |
| Vector descriptor, S=1 | Z | S | TZ | TS |

*Causes a non-maskable program error (description) interrupt.

The exception to the above rules occurs in the case of Jump-type instruction, where the operand, which over-writes PC, is conceptually 32 bits long. When the operand is specified indirectly via a descriptor, its length in store is permitted to be 32 or 64 bits, and in the latter case the least significant 32 bits overwrite PC, the more significant 32 bits are ignored. Type 1 descriptors are not permitted.
When accessing Block 0 or 2 of the Image store, the operand length must be 32 bits; otherwise the action is undefined. When accessing Block 3 of the Image store, the operand length must be 64 bits; otherwise the action is undefined.
The operand length required by each instruction is listed with the instruction description. The operand lengths for store-to-store instructions are specified in the secondary instruction format (see 6.1 and 8.3).


### 6.3.1.2 Addressing rule

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

### 6.3.1.3 Word alignment

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

> TOS
> (LNB+n), (LNB+N)
> (PC+N)
> (XNB+N)
> (CTB+N)
> (B+N)

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

```
TOS, (LNB+n), (LNB+N) )  : SF,LNB, XNB and CTB contain word
(XNB+N)                )    aligned addresses
(CTB+N)                )
(PC+N): the least significant bit of the sum is ignored.
(B+N): the two least significant bits of the sum are ignored.
```

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

### 6.3.1.4 Justification in registers

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

### 6.3.2 Primary and tertiary format operands

### 6.3.2.1 Literals

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

### 6.3.2.2 Image Store

The operand forms (IS location N) and (IS location B) cause the 32 bit image store location indicated by N, or by the contents of the B register (see 3.3) to be accessed. They must only be used with the functions listed in section 3.3.2, and the operand length must be 32 bits; otherwise the action is undefined.

### 6.3.2.3 Top of Stack

The operand TOS causes the item at the top of the stack (of length = operand length) to be used as operand, and SF to be decremented by the operand length in words. Program error interrupt occurs if this causes SF to become not greater than LNB.
For store type instructions the result is stored as a new top-of-stack item causing SF to be incremented. If storing the result violates the stack segment limit a virtual store interrupt will occur.

### 6.3.2.4 B

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

### 6.3.2.5 Directly-accessed items in store

For the operand forms

     LNB+n
     LNB+N
     XNB+N
     CTB+N
      PC+N
       B+N

the address of the operand is formed by adding N (or n) to the appropriate pointer location, to form a byte address which is a multiple of 4. Length of item accessed = operand length. The rules for checking this addition vary from one form to another, as shown below:-

(LNB+n), (LNB+N)    n extended with zeroes; m.s. 2 bits of N must be zeroes. No carry out of LNB permitted.

(XNB+N), (CTB+N)    No check - N extended with zeroes.

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

(B+N)    No check - N is extended with zeros and is regarded as a word displacement. The two LS bits of B are ignored.

Failure of any check causes interruption.

### 6.3.2.6 Indirectly accessed items in store

The operand is accessed via a descriptor at the specified location

There are two cases: in one case the descriptor is accessed like a directly-accessed operand (i.e. as described in 6.3.2.3 and 6.3.2.5 above, but of length = 64 bits) - the descriptor may be modified by the contents of B - and in the other case the descriptor is already in DR and may be modified by a directly accessed quantity. In both cases the descriptor is left in DR, unmodified, after use. Unless otherwise stated, the descriptor may be of types 0,1,2 or Escape for any primary or tertiary format non-jump instructions. For jump instructions (incuding call) the same rule applies except that type 1 descriptors are not allowed, and the size code in a type 0 descriptor may only be 32 or 64 bits. Additionally, jump instructions are allowed to use descriptors (type 3 subtype 33). For a 'Call' instruction 'Code or 'System Call' types are also permitted. If the descriptor is of escape type a jump out of sequence occurs as described in Section 5.

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

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

The checks of section 6.3.2.5 apply when the directly-accessed item is used as a descriptor or modifier. The rules for accessing and aligning operands are given in 6.3.1 Modifiers or descriptors taken from the top of the stack (using the forms given by k" = 6 in 6.1) cause SF to be decremented by 1 or 2 words, respectively.

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

### 6.3.3 Secondary Format

The secondary format is only used by store-to-store functions. ACC may contain a source descriptor and DR contains the destination string descriptor (an Escape descriptor may be used in place of the latter). Further details are given in Section 8.3.

### 6.3.4 Full Segments

Where the segment limit of 256k. bytes is too small to provide sufficient space for the data, overflow is allowed into a new segment with the next higher segment number. The lower segment limit in this case must be set to the maximum, i.e. bits 14 to 24, all ones. Direct and indirect operands are allowed to cross the boundaries of such full segments. Exceptions to the above are:-

Code segments.   These may exist in more than one segment but the program counter (PC) may not be updated in the course of a normal instruction to cross such a segment boundary (see 5.1).

Stack segments.   Restricted to one segment (i.e. one segment less than one page, if paged or one segment less 128 bytes if unpaged).

# 7 EXCEPTION CONDITIONS

## 7.1 Categories

Exception conditions occur in one of three categories and an occurrence within a category leads to a corresponding interrupt unless masked. They are:

    System errors
    Virtual Store conditions
    Program errors

Each exception condition produces an exception identifier, the 32 bit parameter which is stacked as an interrupt identifier.

## 7.2 System errors

A system error occurs when the system hardware or software fails,. It is therefore linked to some form of restart mechanism and, since this will to some extent be design implementation defined, the definition of the precise mechanism is implementation dependent. However the following causes of system error interrupt are defined:

1. Virtual store condition on 'Activate' (dump area inaccessible)
2. Virtual store condition on interrupt (IST inaccessible)
3. Virtual store condition on stack-switching interrupt (dump area in (old) segment (SSN+1) inaccessible).
4. Virtual store condition on stack-switching interrupt (top of new stack inaccessible for loading parameters)
5. Masked interrupt in classes 5,7,8,9,10
6. Hardware errors detected by hardware checking mechanisms (e.g. parity)
7. Segment table format error.
8. Real address out of range.
9. Attempt to load odd number to SSN.

Errors which occur in the use of that part of image store which accesses different physical units will be handled by fault responses rather than interrupt (similar to IO device failures).

The parameter returned at the time of interrupt should indicate the cause of interrupt but is implementation dependent.

Some of these conditions (e.g. 2,3,4) may themselves inhibit the normal system error interrupt mechanism. The subsequent action in this case is machine dependent.

## 7.3 Virtual store conditions

7.3.1 A virtual store condition occurs when access to the virtual/real translation table does not produce a real address. The following causes are identified:

   0 Segment number greater than limit.
   1 Segment not available (A = 0 in segment table entry)
   2 Segment limit exceeded (non paged segments)
   3 Page number greater than limit
   4 Page not available (A = 0 in page table entry)

Checks 1-4 are carried out after ACR against APF checks (see 4.2.2) which may result in a Program Error.

7.3.2 The format for the 32 bit parameter returned upon virtual store interrupt is:

bits 0-24    Most significant 25 bits of virtual address
bits 25,26   OX (X undefined) Write access attempted
               10 Read access attempted
               11 Execute access attempted
bit 27        0
bit 28        1 if caused during interrupt sequence (II set)
bits 29-31   cause for interrupt, categories as in 7.3.1.

7.3.3 Additional information may be dumped in segment (SSN + 1) (hardware dump area) to an implementation defined format to indicate partially completed operations which must be resumed at the point where a condition occurred.
In certain circumstances (see 7.2 above) virtual store conditions will give rise to system error interrupts.

## 7.4 Program errors

7.4.1 Each program error is associated with an interrupt parameter.
The parameter has three parts
a)  A bit (bit 24) which if = 1 indicates restart is not possible
(see section 5.3.10)
b)  A program error identifier (PEI), range defined (bits 25-31)
c)  A sub identifier, implementation defined (bits 16-23)
(if possible, the class numbers in the following sections should
be used).

PEI numbers are assigned as follows:
Code 0 Floating point overflow
1 Floating point underflow
2 Fixed pint overflow
3 Decimal overflow
4 Zero divide
5 Bound check
6 Size
7 B overflow
8 Stack
9 Privilege
10 Descriptor
11 String
12 Instruction
13 Accumulator
14 ESR errors (emulating machines only) (see[7])

7.4.2 Errors in each category are now classified. Class numbers do not
necessarily correspond to sub-identifiers, which are implementation-
defined and which may not be generated at all for some categories.  In
such cases bits 16-23 of the parameter will be all 1's.

7.4.2.0 Floating point overflow.  Detailed in section 8, no sub-identifiers.

7.4.2.1 Floating point underflow. Detailed in section 8, no sub-identifiers.

7.4.2.2 Fixed point overflow.    Detailed in section 8, no sub-identifiers.

7.4.2.3 Decimal overflow.        Detailed in section 8, no sub-identifiers.

7.4.2.4 Zero divide.             Detailed in section 8, no sub-identifiers.

7.4.2.5 Bound and check.         Detailed in section 6 and 8, classes:-

0 Descriptor bound check-modifier (unsigned) too large
1 Modify DR-operand (unsigned) greater than or equal DR bound
2 Dope vector multiply. (Index-lower bound) oveflows
3 "         "         "    (Index lower bound) negative
4 "         "         "    Multiplier negative.
5 "         "         "    Upper bound negative
6 "         "         "    Displacement (product) greater than or equal to
$2^{31}$, or upper bound.
7 "         "         "    DR bound goes negative
8 Table check, translate, (DR) string byte too large.

**7.4.2.6** Size. Location too small for operand.
- 0 Significant part of operand truncated (See section 6.3.1.1)
- 1 Store to register, descriptor size code
  (type 0 or 2) greater than operand length (except for jumps)

**7.4.2.7** B overflow. Detailed in section 8; no sub-identifiers

**7.4.2.8** Stack. Stack register operation check
- 0 Unstacking operation makes SF less than or equal to LNB (SF unaltered)
- 1 Undefined.
- 2 Load LNB and Exit, bits 0-13 of new LNB not equal to SSN (LNB unaltered)
- 3 Load LNB and Exit, new LNB greater than or equal to SF
- 4 Raise LNB, new LNB greater than or equal to SF (LNB unaltered)
- 5 Raise LNB, new LNB less than old LNB (LNB unaltered)
- 6 Adjust SF, new SF less than or equal to LNB (SF unaltered)
- 7 Adjust SF, segment overflow (SF unaltered)

**7.4.2.9** Privilege. A program error leads to an attempt to use a resource which the current level of privilege does not justify.
- 0 Read protection fail with ACR
- 1 Write protection fail with ACR
- 2 Execute when execute permission bit not set
- 3 Use of image store without privilege permission
- 4 Use of non-existent image store (e.g. address does not exist, or read to write only IS or write to read only IS)
- 5 ACR less than old ACR or PRIV greater than old PRIV on Exit
- 6 Diagnose or Activate executed without privilege permission

**7.4.2.10** Descriptor
- 0 Jump descriptor not type 0, size 32 or 64, type 2, or escape type or code type
- 1 Descriptor for normal operand access not type 0,1,2 or escape type
- 2 Call descriptor is not type 0, size 32 or 64, type 2 code, escape or system call
- 3 Link descriptor for Exit is not code, escape, or system call
- 4 Descriptor in DR not type 0, size 32 for Dope Vector multiply
- 5 Length in type 1 descriptor used by primary format instruction is zero or exceeds operand length.
- 6 DR descriptor is not type 0 or 1 with size code 3, or escape, for store-to-store operation.
- 7 ACC descriptor not type 0 or 1 with size code 3, for some store-to-store operations.
- 8 ACC descriptor not type 0, size 1 or 8, for Table check and Table translate, respectively.
- 9 Size code incorrect in type 0 descriptor.
- 10 Descriptor sub-type undefined.
- 11 Modify DR with System Call descriptor in DR.
- 12 Not used.
- 13 Incorrect descriptor used for semaphore instruction
- 14 Reserved for emulating machines (see[7])

### 7.4.2.11 String

0 L greater than DR bound
1 Not used.
2 L greater than ACC bound for 16 bit form of Move, Compare, And, Or and Not equivalent strings, Check overlap.

### 7.4.2.12 Instruction

0 Instruction function is illegal or unassigned
1 Store to literal.
2 Incorrect address form for certain functions (e.g. Increment and test, Modify DR, Load relative).
3 Relative jump attempts to alter segment number in PC.
4 Unassigned operand address forms: k"=1, k"=7 and k'=1
5 Item addressed in stack segment lies above TOS
6 Item addressed by PC + N lies outside current code segment.
7 Normal update of PC attempts to alter segment number.
8 Reserved for emulating machines (see[7])

### 7.4.2.13 Accumulator ACC incompatible with instruction

0 ACS = 128 bits and fixed point or logical. (except Multiply Double)
1 ACS = 64 bits and Add/Subtract logical.
2 ACS = 128 bits and Float
3 ACS = 32 bits and Floating divide double.
4 ACS = 128/32 bits and Load/Store upper half.
5 ACS = 128 bits, or 64 if fixed, and Multiply double.
6 ACS ≠ 64 bits for store-to-store instruction involving descriptor in ACC
7 Modify PSR or Exit attempts to set ACS = 0
8 ACS = 128 bits and Compress/Expand ACC

### 7.4.3 Operand addressing errors

The phrase 'operand addressing errors is used throughout section 8 to cover errors of the following types:

Bound check (class 0)
Size (class **1)
Privilege (classes, 0,** 1,3,4)
Descriptor (classes *0, **1, *2, **5,9,10)
Instruction (classes *3, 4, 5, 6)

| * Jumps only |
| --- |
| **Not jumps |

Other program errors are listed explicitly with each instruction description, except for errors which are checked as part of the instruction sequencing process, or apply to all instructions.
e.g.:
Stack (class 0)
Privilege (class 2)
Instruction (classes 0,7)

## 7.5 Program Mask

The format of the program mask is given in Section 3. If a particular bit is set and the corresponding interrupt condition occurs, it is ignored.

The results left in ACC or B etc, by arithmetic operations when Floating overflow, Floating underflow, Fixed overflow, Decimal overflow, B overflow or Zero divide conditions occur are defined in Section 8. These are unaffected by PM, which in these cases only determines whether or not these conditions cause interruption, and similarly when a Bound check condition arises from the Dope Vector multiply instruction, (classes 2 -6 only).

However, when a Bound check condition arises in modifying the address in a descriptor, during operand access, and is masked the effect is as if BCI were set in the descriptor. If unmasked the condition causes the fetch or store operation to be suppressed and interrupt occurs.

Similarly if an unmasked Size condition occurs the operation is suppressed an interrupt occurs. If the condition is masked, and it occurs on a store-to-register operation, the additional portion of the item in store is ignored; if a register-to-store operation, the truncated non-zero bits are treated as zeroes.

Masked program error conditions do not remain pending i.e. clearing a PM bit does not cause the corresponding interrupt to occur if the condition arose while it was masked - even if it is one of the overflow conditions and OV is set.

## 7.6 State of registers, etc. after program error

The manner of completion of an instruction which causes a program error interrupt and the exact point in the instruction sequence at which interrupt occurs, may be inconsistent between different models, except where explicitly specified, and subject to the following rules:

a) Registers and store locations (insofar as these can be defined) which the erring instruction would not have altered in normal circumstances will be unaffected.

b) With one exception (Privilege, class 2 - execute, when execute permission bit not set - following a jump or Exit instruction ) the address left in word 16 of segment (SSN+1) is that of the erring instruction. In the exceptional case this address may point to the jump or Exit instruction or to the destination instruction.

# 8    INSTRUCTION DESCRIPTIONS

## 8.1    Miscellaneous functions

### 8.1.1 List of instructions

By "Miscellaneous functions" is meant those instructions which do not come under the headings of floating-point, fixed-point, logical or decimal operations in the accumulator (but including operations on B), store-to-store operations, or privileged operations. All instructions use the primary or tertiary formats described in 6.1; rules for operand access are given in 6.3.

These instructions comprise:

**Control and Jump Instructions**

| | |
|---|---|
| Load LNB | Jump and Link |
| Load XNB | Jump |
| Raise LNB | Decrement B and jump if non-zero |
| Adjust SF | Jump on CC |
| Store LNB | Jump on arithmetic-condition true |
| Store SF | Jump on arithmetic condition false |
| Increment and Test | Escape exit |
| Test and Decrement | Out |
| Call | Idle |
| Exit | Store XNB |
| Load CTB | Pre-call |
| Store CTB | |

**ACC Instructions**

| | |
|---|---|
| Modify PSR | Stack and Load |
| Copy PSR | Load |
| Set ACS 32 and load | Store |
| Set ACS 64 and load | Load upper half |
| Set ACS 128 and load | Store upper half |
| Stack set ACS 32 and load | Copy DR |
| Stack, set ACS 64 and load | Read real-time clock |
| Stack, set ACS 128 and load | |

**B Instructions**

| | |
|---|---|
| Load B | Multiply B |
| Stack and load B | Compare B |
| Store B | Compare and increment B |
| Add to B | Dope vector multiply |
| Subtract from B | |

**DR Instructions**

| | |
|---|---|
| Load DR | Load bound |
| Stack and load DR | Modify DR |
| Store DR | Validate address |
| Load relative | Increment address |
| Load address | Start significance |
| Load type and bound | |

## 8.1.2 Control and Jump Instructions

### 8.1.2.1 Load LNB (LLN)          Function Code : 7C

Operand length    : 32 bits
Description       : Bits 14 - 29 of the operand are loaded to LNB. Bits 30,31 are ignored. Bits 0 - 13 are checked equal to SSN. The new value of LNB is checked to be less than SF. LNB is unaltered if these checks are not satisfied.
CC                : Unaltered
Program errors    : Operand addressing errors
                    Bits 0 - 13 not equal to SSN (see 7.4.2.8.2)
                    Bits 14 - 29 greater than or equal to SF (see 7.4.2.8.3)

### 8.1.2.2 Load XNB (LXN)          Function Code  :  7E

Operand length    : 32 bits
Description        : Bits 0 - 29 of the operand are loaded to XNB. Bits 30, 31 are ignored.
CC                 : Unaltered
Program errors     : Operand addressing errors

### 8.1.2.3 Raise LNB (RALN)          Function Code  :  6C

Operand length      : 32 bits

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

CC                   : Unaltered

Program errors       : Operand addressing errors
                       Operand less than or equal to 0 (see 7.4.2.8.4)
                       Operand greater than SF - +LNB (see 7.4.2.8.5.)

### 8.1.2.4 Adjust SF (ASF)          Function Code  :  6E

Operand length      : 32 bits

Description          : The operand, regarded as a signed number (of words), is added to the word address in SF. Bits 0 - 15 of the operand must be all the same and must equal the carry out of the most significant bit of SF when performing the sum, i.e. segment overflow is not permitted. The result must be greater than LNB. SF is not adjusted if these checks are not satisfied. New stack locations are not cleared. If the operand involves TOS, SF is decremented before being adjusted. If the location pointed at by SF after adjustment, lies beyond the stack segment limit, or lies in a page which is not available in main store, a virtual condition occurrs, as if that location had been accessed. In this case SF is not adjusted, but the adjusted address must be left in the VSI parameter. Bits 25 and 26 of the VSI parameter (section 7.3.2) may indicate that either a read or write access was attempted.

CC                          : Unaltered

Program errors      : Operand addressing errors
                              New SF less than LNB (see 7.4.2.8.6)
                              Operand too large (segment overflow)
                              (see 7.4.2.8.7)

## 8.1.2.5 Store LNB (STLN)                          Function Code  :  5C

Operand length      : 32 bits

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

CC                          : Unaltered

Program errors      : Operand addressing errors
                              Literal operand (see 7.4.2.12.1)
                              Non-zero bits of stored item truncated.
                              (see 7.4.2.6.0)

## 8.1.2.6 Store SF  (STSF)                          Function Code  :  5E

Operand length      : 32 bits

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

CC                          : Unaltered

Program errors      : Operand addressing errors.
                              Literal operand (see 7.4.2.12.1)
                              Non-zero bits of stored item truncated (see 7.4.2.6.0)

## 8.1.2.7 Increment & Test (INCT)                   Function Code  :  56
          Test & Decrement (TDEC)                    Function Code  :  54

Operand length      : 32 bits

Description          : The prime use of these instructions is to implement
semaphores and interlocks.

The following restrictions apply to operand formats.
a) The operand must be located in the store rather than a register.
   Direct TOS and (PC+N) operand forms are not permitted.

b) If the operand is accessed indirectly it must be via a vector
   (type 0) descriptor with size code 32 bits or a semaphore (type 3,
   subtype 40,41) descriptor.

**ICL**

**Product specification**

**Company restricted**

**PSD** 2.5.1

Issue 6/0

Sheet 82

The Condition Code is set from the final value of the Increment and Test, and from the original value for Test and Decrement. INCT and TDEC set the accumulator to the original value of the operand. ACS is set to 32 bits. OV is unaltered.

INCT adds 1 to the operand. Overflow is ignored.
TDEC subtracts 1 from the operand. Overflow is ignored.

However the operand is accessed it is fetched directly from store and any existing slave store copy is updated; and access to the operand location by other units is prevented while the operand is modified. If slave storage is present in the processor (see section 4.3) INCT and TDEC are also required to clear the operand slave store of items from segments marked non-slaved (NS) in either segment table. The segment containing the operand need not be cleared from slave stores unless marked NS.

At AML1 the requirements in the above paragraph are restricted to access via Semaphore Desriptors if a switch bit is set in the image store.

CC: 0   Operand = 0
    1   Operand greater than  0
    2   Operand less than -1
    3   Operand = 1
(Here 'operand' refers to the final value for INCT and original value for TDEC.)
Program errors:   Operand addressing errors.
                  Incorrect descriptor type (see 7.4.2.10.13)
                  Incorrect operand types: Literal IS, TOS, B, (PO+N).
                  See 7.4.2.13.2)

### 8.1.2.8 Call (CALL)                      Function code :   1E

Operand length  : 32 bits
Note: If the operand is addressed indirectly via a type zero or type 2 descriptor, the addressed item in store may be 32 or 64 bits long. In either case it is treated simply as an instruction address, not a descriptor ( so neither System Call nor Escape mechanisms can be invoked), and if 64 bits long, its more significant bits are ignored. If the operand is addressed indirectly via a Code Descriptor, the address in bits 32-63 of the Code Descriptor is the operand. If the Code Descriptor is unbounded, a microcode routine may be entered when the jump is made.
Description: This instruction is used to enter procedures. A link descriptor specifying the location to return to on exit is generated and loaded into (LNB+1), (LNB+2). The operand increments or overwrites PC causing a jump to occur.

If SF less than or equal to LNB+2, the link descriptor is not stored and the instruction terminates with a program error interrupt.

The link descriptor is of unbounded Code Type and consists of:
In(LNB+2):

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

**ICL**

**Product specification**

**Company restricted**

**PSD** 2.5.1

**Issue** 6/0

**Sheet** 83

```
In(LNB+1);
        Bits 0-7    : 11100001 (Type3, subtype 33)
        Bits 8-11   : ACR
        Bit 12      : D  this is always stored as zero
        Bit 13      : PRIV
        Bit 14      : OV
        Bit 15      : E
        Bit 16-23   : Program mask
        Bits 24-26  : Zero
        Bit 27      : 1
        Bit 28,29   : CC
        Bits 30,31  : ACS
```

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

a) 0 or 2      : no special action

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

c) System Call   : An interrupt is performed (see section5 ). If the descriptor is modified, the modifier is accessed but no modification takes place, (e.g. if the modifer is TOS, SZF will be decremented).

d) Escape      : An escape action is performed (see Section 5.)(LNB+1,+2) will be undefined

If the operand is accessed directly from (LNB+2) or via a descriptor in (LNB+1) the result is undefined. If the operand is accessed indirectly, system software may intervene to decode a system call descriptor and in this case the contents of ACC, B and XNB must be regarded as undefined (see [4]).

CC      : Unaltered

Program errors   : Operand addressing errors for call instruction. (see 7.4.2.10.2.) SF not greater than LNB+2 (see 7.4.2.12.5).

### 8.1.2.9 Exit (EXIT)                 Function Code : 38

Operand length    : 32 bits

Description       : This instruction is used to return from procedures and after non-stack-switching interrupts. The stack is returned to its status quo and a jump is made as specified by the link descriptor. Fields of the link descriptor may be used to overwrite parts of PSR as specified by bits in the operand.
The link descriptor is extracted from (LNB+1, LNB+2). It may only be of types Code, System Call or Escape. If SF not greater than LNB+2, or the

descriptor is not one of these types, the instruction terminates with a program error interrupt.

If the link descriptor is System call, PSR and PC are copied into DR (in the form of a link descriptor - see section 8.1.2.8) and the System Call interrupt exception condition routine is entered (see section 5).

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

- if the value of bits 8-11 of (LNB+1) is not less than ACR, bits 8-11 of (LNB+1) overwrite ACR else program error interrupt.
- If the value of bit 13 of (LNB+1) is not greater than PRIV, bit 13 of (LNB+1) overwrites PRIV; else program error interrupt.
- if operand bit 25=1, bits 16-23 of (LNB+1) overwrite PM
- if operand bit 26=1, bits 28-29 of (LNB+1) overwrite CC
- if operand bit 27=1, bits 30-31 of (LNB+1) overwrite ACS (a program error interrupt if attempt is made to set ACS=0)
- if operand bit 28=1, bit 12 of (LNB+1) overwrites D
- if operand bit 29=1, bit 14 of (LNB+1) overwrites OV (this does not cause an overflow interrupt)
- bit 15 of (LNB+1) overwrites E

Other operand bits are ignored (reserved). The operand may only be a 7-bit literal.

To restore the stack status quo, the contents of the LNB register are transferred to SF, and, provided that bits 0-13 of (SF) = SSN, bits 14-29 of (SF) are transferred to LNB (otherwise LNB is unaltered). If bits 14-29 of (SF) are not less than bits 14-29 of SF, a program error is generated and the contents of LNB are undefined. The address from the code descriptor (ex;(LNB+2)) overwrites PC. Finally SF is decremented by 1 if the least significant bit (Bit 31) of (SF) is 1 and a jump is made to the address in PC. In emulating machines, if E=1 and EM has a locally valid value, emulate alien code. If the new ACR is larger than the previous value, and the EP bit is set in SSR, an EP interrupt will occur before the next instruction is executed unless masked.

When Exit is used to return from procedures, system software may intervene to decode a system call descriptor and in this case the contents of DR and XNB must be regarded as defined (see [4])

CC                 : Unaltered if operand bit 26 = 0
                     If operand bit 26=1 and the link descriptor is of Code type, CC takes value specified in bits 28,29 of (LNB+1)

Program Errors: Incorrect operand type (must be 7-bit literal) (see 7.4.2.12.2). SF not greater than LNB+2 (see 7.4.2.12.5.) Link descriptor not Code. System Call or Escape (see 7.4.2.10.3)
New PRIV greater than old PRIV ) (see 7.4.2.9.5)
New ACR less than old ACR      )
Bits 0-13 of (LNB) not equal SSN (see 7.4.2.8.2)
Bits 14-29 of (LNB) not less than new SF (see 7.4.2.8.3)
Attempting to set ACS=0 (see 7.4.2.13.7)
(Emulating machines: PEI 14) New E=1 and EM=0 or invalid value.

### 8.1.2.10 Jump and Link (JLK)        Function Code : 1C

Operand length : 32 bits. The note under CALL (8.1.2.8) applies

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

CC : Unaltered

Program Errors : Operand addressing errors for jump instruction. (see 7.4.2.10.0).

### 8.1.2.11 Jump (J)        Function Code : 1A

Operand length : 32 bits. The note under CALL (8.1.2.8) applies.

Description : The operand increments or overwrites PC causing a jump to occur.

CC : Unaltered

Program Errors : Operand addressing errors for jump instruction. (see 7.4.2.10.0)

**8.1.2.12 Decrement B & Jump If Non-Zero (DEBJ)**          **Function Code : 24**

Operand Length : 32 bits. The note under CALL (8.1.2.8) applies

Description : 1 is subtracted from B. If the result is non-zero a
jump is made, the operand incrementing or overwriting PC. If the result
is zero no jump occurs and the next instruction in sequence is obeyed. In
either case the decremented value is left in B.
If B originally contained -231 OV is set, 231-1 is left in B, and
interrupt occurs unless the condition is masked; otherwise OV is cleared.
If the operand forms uses B, the jump location is undefined.

CC               :   Unaltered

Program Errors   :   Operand addressing errors for jump instruction
                     B overflow (unless masked). (see 7.4.2.7)

**8.1.2.13  Jump on CC (JCC)**                             **Function Code   : 02**

This instruction uses the tertiary format described in 6.1.

Operand Length : 32 bits. The note under CALL (8.1.2.8) applies.

Description : If the bits of the mask field M are M0, M1, M2, and M3,
and if the current condition code setting is 1, then operate as for the
Jump instruction if, and only if, M1 = 1; otherwise proceed to the next
instruction in sequence. Alternative condition code settings may be
tested by making more than one bit of M non-zero.

CC               : Unaltered

Program Errors   : Operand addressing errors for jump instruction.

**8.1.2.14 Jump on Arithmetic - Condition True (JAT)**     **Function Code  :  04**

         **Jump on Arithmetic - Condition False (JAF)**    **Function Code  :  06**

These instructions use the tertiary format described in 6.1.

Operand length : 32 bits. The note under CALL (8.1.2.8) applies.

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

Conditions

   (floating point)
   M = 0   ACC = 0 (Bits *8-31/32-63/72-127 all zero)
       1   ACC > 0 (Bit 0=0, bits *8-31/32-63/72-127 not all zero)
       2   ACC < 0 (Bit 0=1, bits *8-31/32-63/72-127 not all zero)
       3   Undefined

(fixed point)
    4   ACC = 0 (All bits zero)                                    )Undefined
    5   ACC > 0 (Bit 0=0, remaining bits not all zero              )if ACS=3
    6   ACC < 0 (Bit 0=1)                                          )
    7   Undefined

(decimal)
    8   ACC = 0 (Bits *0-27/28-59/60-123 all zeroes)
    9   ACC > 0 (Bits *0-27/28-59/60-123 not all zeroes,
                least significant 4 bits = 1011 or 1101)
   10   ACC < 0 (Bits *0-27/28-59/60-123 not all zeroes,
                least significant 4 bits = 1011 or 1101)

(DR)
   11   Length = 0 (DR bits 8-31 zero)

(B)
   12   B = 0 (Bits 0-31 all zero)
   13   B > 0 (Bit 0=0, bits 1-31 not all zero)
   14   B < 0 (Bit 0=1)
   15   OV set

        *8-31/32-63/72-127 implies that bits 8-31 of ACC are always
checked, that bits 32-63 are checked as well if ACS = 64 or 128 bits,
also bits 72-127 if ACS = 128 bits.

CC                  : Unaltered

Program errors : Operand addressing errors for jump instruction.

### 8.1.2.15 Escape Exit (ESEX)                    Function Code : 3A

Operand length      : Not applicable. Literal must be specified.

Description         : The operand field is ignored. Bits 0-30 of the word
at the top of the stack overwrites PC; bit 31 is ingored.  SF is
decremented by 1 word. The 'D' bit is set in PSR, and a jump is made to
the instruction pointed at by PC. If that instruction accesses the store
indirectly, via a descriptor located in the store, the effect of the D bit
will be to prevent that descriptor from being used; instead the descriptor
already in DR (assumed to have been placed there by the escape routine)
will be used. If the instruction specifies modification, it will take
place. e.g. if the operand format is (TOS + B), it will be interpreted as
(DR + B).  If indirect access via DR, or direct access, is specified, the
D bit is ignored. E.g. if the operand format is (DR + TOS) it will be
interpreted literally.  The D bit is cleared by the instruction so that
its effect is limited to the first instruction after Escape exit.  That
instruction will usually be the one which originally triggered the escape
mechanism, re-executed; note that in the first example above, TOS will
have been accessed (to obtain the Escape descriptor) before the escape
action, in the second example TOS is not accessed until after Escape exit
since Escape descriptors are unmodified.

CC : Unaltered

Program errors : Only universal types listed in section 7.4.3

### 8.1.2.16 Out (OUT)                                          Function Code : 3C

Operand length : 32 bits

Description : This instruction causes a class 9 interrupt (See Section5) to occur. The operand is left as the 32 bit interrupt parameter on the new stack. ACC, B, and (unless operand access is indirect) DR are unaltered.

CC : (Dumped value) unaltered

Program errors : Operand addressing errors

### 8.1.2.17 Idle (IDLE)                                          Function Code : 4E

Operand length : Not applicable. Literal must be specified.

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

CC : Unaltered.

Program errors : Any universal types listed in section 7.4.3.

### 8.1.2.18 Load CTB (LCT)                                          Function Code : 30

Operand length : 32 bits

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

CC : Unaltered.

Program errors : Operand addressing errors

### 8.1.2.19 Store XNB (STXN)                                          Function Code : 4C

Operand length : 32 bits

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

CC : Unaltered

Program errors : Operand addressing errors
Literal operand (see 7.4.2.12.1)
Non-zero bits of stored item truncated,
(see 7.4.2.6.0).

8.1.2.20 Store CTB(STCT)                                  Function Code  :  36

    Operand length      :  32 bits

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

    CC                  :  Unaltered

    Program errors      :  Operand addressing errors.
                       Literal operand (see 7.4.2.12.1)
                       Non-zero bits of stored item truncated.
                       (See 7.4.2.6.0)

8.1.2.21 Pre-call (PRCL)                                  Function Code  :  18

    Operand length      :  32 bits
    Description         :
        a)  The operand is fetched.  The operand must be a 7 bit literal.
        b)  If SF is even (Bit 15=0), SF is incremented by 1.
        c)  The contents of LNB are expanded to a 32-bit byte address by
concatenating the contents of SSN on the left and 2 zero bits on the
right. Bit 31 is then set to 1 if SF was incremented in (b) and the result
is stacked. (SF is incremented by 1)
        d)  The action Adjust SF (ASF) is now followed as described in
8.1.2.4. (The operand is added to SF).

    CC                  :  Unaltered

    Program errors      :  Operand addressing errors.
                       New SF less than  LNB (see 7.4.2.8.5)
                       Operand too large (see 7.4.2.8.7)
                       Incorrect operand type (must be 7 bit literal)
                       (see 7.4.2.12.2)

8.1.3.  ACC Instructions

8.1.3.1 Modify PSR  (MPSR)                                Function Code  :  32

    Operand length      :  32 bits

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

| 16 | 23 | 24 | 27 | 28 | 29 | 30 | 31 |
|---|---|---|---|---|---|---|---|
| PM | | | | CC | | ACS | |

    - if bit 27 is 1, bits 30 and 31 overwrite ACS.  (See section 3.1.9
Program error if attempt is made to set ACS =0).
      If bit 27 is 0, ACS is unaltered and bits 30 and 31 may take any
value.
    - if bit 26 is 1, CC is set to the value in bits 28 and 29.  Otherwise,
CC is unaltered and bits 28 and 29 may take any value.

- if bit 24 is 1, bits of the Program Mask which correspond to 1's in operand bits 16-23 are made 1's; otherwise they are unaltered.
- if bit 25 is 1, bits of the Program Mask which correspond to 0's in operand bits 16-23 are made 0's; otherwise they are unaltered.
- bits 0-15 of the operand are ignored and may take any value.

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

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

Program errors     :  Operand addressing errors
                      Attempt to set ACS = 0 (see 7.4.2.13.7)

### 8.1.3.2 Copy PSR (CPSR)                          Function Code  :  34

Operand length     :  32 bits

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

                      Bits  0 - 15  0's
                      Bits 16 - 23  PM
                      Bits 24 - 27  1110
                      Bits 28,  29  CC
                      Bits 30,  31  ACS

Subsequent use of this operand by 'Modify PSR' (8.1.3.1) causes PM and CC to be overwritten, but not ACS, unless bit 27 is made 1.

CC                 :  Unaltered

Program errors     :  Operand addressing errors
                      Literal operand (see 7.4.2.12.1)
                      Significant part of operand truncated
                              (see 7.4.2.6.0)


### 8.1.3.3. Set ACS 32 & Load  (LSS)          Function Code  :  62
####         Set ACS 64 & Load  (LSD)          Function Code  :  64
####         Set ACS 128 & Load (LSQ)          Function Code  :  66

Operand length     :  New value of ACS

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

CC                 :  Unaltered

Program errors     :  Operand addressing errors

8.1.3.4 Stack, Set ACS 32 & Load (SLSS)    Function Code : 42
      Stack, Set ACS 64 & Load (SLSD)    Function Code : 44
      Stack, Set ACS 128 & Load (SLSQ)    Function Code : 46

    Operand length    : New value of ACS

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

    CC    : Unaltered

    Program errors    : Operand addressing errors

8.1.3.5. Stack & Load (SL)    Function Code : 40

    Operand length    : ACS

    Description    : The contents of ACC are copied to an intermediate register. The operand is loaded to ACC, and the contents of the intermediate register are stacked, causing SF to be incremented by ACS.
    The intermediate register ensures that the operand forms TOS, (DR+TOS), (TOS) and (TOS + B) are valid.
    OV is cleared

    CC    : Unaltered

    Program errors    : Operand addressing errors

8.1.3.6. Load (L)    Function Code : 60

    Operand length    : ACS

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

    CC    : Unaltered

    Program errors    : Operand addressing errors

8.1.3.7. Store (ST)    Function Code : 48

    Operand length    : ACS

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

CC   : Unaltered

Program errors   : Operand addressing errors
Literal operand (see 7.4.2.12.1)
Significant part of operand truncated
(see 7.4.2.6.0)

### 8.1.3.8 Load Upper Half (LUH)   Function Code : 6A

Operand length   : ACS

Description : ACS is doubled and the operand is loaded to the upper half of ACC. The lower half of ACC is unaltered. OV is cleared. ACS = 128 bits is not permitted.

CC   : Unaltered

Program errors : Operand addressing errors
ACS = 128 bits (see 7.4.2.13.4)

### 8.1.3.9 Store Upper Half(STUH)   Function Code : 4A

Operand length   : Half ACS

Description : The contents of the more significant half of ACC are stored in the operand location. ACS is halved. The lower half of ACC is unaltered. ACS = 32 bits is not permitted.

CC   : Unaltered

Program errors : Operand addressing errors.
Literal operand (see 7.4.2.12.1)
ACS = 32 bits (see 7.4.2.13.4)
Significant part of operand truncated
(see 7.4.2.6.0)

### 8.1.3.10 Copy DR (CYD)   Function Code : 12

Operand length   : Not applicable. Literal must be specified.

Description : The contents of DR are copied to ACC. ACS is set to 64 bits. OV is cleared. DR is unaltered.

CC   : Unaltered

Program errors : Only universal types listed in section 7.4.3

### 8.1.3.11 Read Real Time Clock (RRTC)   Function Code : 68

Operand length   : 1 bit. Literal must be specified.

Description : ACS is set to 64 bits and the value of the hardware real-time clock (see 3.1.13) is loaded to ACC as follows:

i) For OCPs at AML0
   The X register is loaded into bits 0-31 of ACC
   The Y register is loaded into bits 32-63 of ACC

ii) For OCPs at AML1
   a) if the operand value = 0, ACC is loaded as above
   b) if the operand value = 1
      Bits 0-62 of ACC are set to the true binary value of the Real Time Clock such that bit 62 of ACC is equivalent to 2 usec of real time. The algorithm is

$$ACC = 2^{32}(X_{0-31} + (X_{31} \oplus Y_0)) + 2 \cdot Y_{1-31}$$

note: in both cases the OV register is cleared.

CC                  : Unaltered

Program errors      : Only universal types listed in section 7.4.3.

## 8.1.4. B Instructions

### 8.1.4.1 Load B (LB)                     Function Code : 7A

Operand length      : 32 bits

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

CC                  : Unaltered

Program errors      : Operand addressing errors

### 8.1.4.2 Stack & Load B (SLB)           Function Code : 52

Operand length      : 32 bits

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

CC                  : Unaltered

Program errors      : Operand addressing errors

### 8.1.4.3 Store B (STB)                  Function Code : 5A

Operand length      : 32 bits

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

**ICL**

**Product specification**

**Company restricted**

**PSD**    2.5.1

Issue    6/0

Sheet    94

CC       :   Unaltered

Program errors     :   Operand addressing errors
                     Literal operand (see 7.4.2.12.1)
                     Significant part of operand truncated. (see
                     7.4.2.6.0)
NOTE:   The effect on OV if B is the operand of STB is undefined.

### 8.1.4.4 Add to B   (ADB)                      Function Code   :   20

    Subtract From B   (SBB)                  Function Code   :   22

    Multiply B    (MYB)                     Function Code   :   2A

Operand length     :   32 bits

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

CC             :   Unaltered

Program errors      :   Operand addressing errors
                      B overflow (unless masked)(see 7.4.2.7)

### 8.1.4.5. Compare B   (CPB)                  Function Code :   26

Operand length     :   32 bits

Description          :   The contents of B are compared with the operand,
both being regarded as signed integers. The result of the comparison is
indicated in CC.
B and OV are unaltered.
Comparisons are performed exactly as for 32-bit fixed-point Compare
(8.2.4.3)
The contents of B may be used as a modifier in accessing the operand.

CC             :   0    B = operand
                      1    B less than operand
                      2    B greater than operand
                      3    Not used
Program errors      :   Operand addressing errors

### 8.1.4.6 Compare & Increment B (CPIB)        Function Code   : 2E

Operand length     :   32 bits

Description          :   The action of the instruction is identical to that
of Compare B (8.1.4.5), with the addition that after the comparison 1 is
added to the contents of B. OV is set if this causes B to overflow (i.e.

if the contents of B go from $2^{31}-1$ to $-2^{31}$); interrupt will occur if this condition is not masked. OV is cleared if overflow does not occur. The original contents of B may be used as a modifier in accessing the operand.

CC  : 0  B (original contents) = operand
1  B (original contents) less than operand
2  B (original contents) greater than operand
3  Not used

Program errors  : Operand addressing errors
B overflow (unless masked)  (see 7.4.2.7)

### 8.1.4.7 Dope Vector Multiply (VMY)  Function Code  :  2C

Operand length  : 32 bits

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

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

CC  : Unaltered

Program error  : Operand addressing errors
Indirect address form (see 7.4.2.12.2)
Incorrect type and size code of descriptor in DR
(see 7.4.2.10.4)

| | | | |
|---|---|---|---|
| i less than x | ) | | (see 7.4.2.5.3) |
| i-x not less than $2^{31}$ | ) | | (see 7.4.2.5.2) |
| | )Unless | | |
| y less than 0 | )Bound | | (see 7.4.2.5.4) |
| z less than 0 | )check is | | (see 7.4.2.5.5) |
| (i-x)y not less than Z, | )masked | | |
| includes   (i-x)y not less than $2^{31}$ )) | | | (see 7.4.2.5.6) |
| Bound check on x, y or z | ) | | (see 7.4.2.5.7) |

8.1.5  DR instructions

8.1.5.1 Load DR  (LD)                                   Function Code  :  78

Operand length      :  64 bits

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

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

Program errors      :  Operand addressing errors

8.1.5.2  Stack & Load  (SLD)                            Function Code  :  50

Operand length      :  64 bits

Description         :  The contents of DR are copied to an intermediate
register.  The operand is loaded to DR, and the contents of the
intermediate register are stacked, causing SF to be incremented by 2
words. The intermediate register ensures that the operand forms TOS,
(DR+TOS), (TOS) and (TOS+B) are valid.
If accessed indirectly, the operand, rather that the descriptor used to
load it, is left in DR.  CC is set to indicate the type of descriptor
loaded.

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

Program errors      :  Operand addressing errors.

8.1.5.3 Store DR (STD)                                  Function Code  :  58

Operand length      :  62 bits

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

CC                  :  Unaltered

Program errors      :  Operand addressing errors
                       Literal operand (see 7.4.2.12.1)
                       Significant part of operand truncated  (see
                       7.4.2.6.0)
                       Indirect address form  (see 7.4.2.12.2)

**8.1.5.4  Load Relative  (LDRL)**                          Function Code  :  70

Operand length        :  64 bits

Description        :  The operand is transferred to DR, with its least
significant 32 bits (the address field) augmented by the value of its own
byte address.  Carry out of the address field resulting from this addition
is ignored.  Thus, if the operand (the item in store) starts in byte x and
is a descriptor pointing at location y, that descriptor is left in DR with
its address adjusted to point at location (x-y).  x need not be a multiple
of 4.  If accessed indirectly, the operand, rather than the descriptor
used to access it, is left in DR.  Literal operands and operands in
registers are not permitted.  CC is set to indicate the type of descriptor
loaded.

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

Program errors        :  Operand addressing errors
                         Invalid address forms (see 7.4.2.12.2)

**8.1.5.5.  Load Address (LDA)**                          Function Code  :  72

Operand length        :  32 bits

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

CC                    :  Unaltered

Program errors        :  Operand addressing errors

**8.1.5.6.  Load Type and Bound (LDTB)**                          Function Code  :  74

Operand length        :  32 bits

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

CC                    :  Unaltered

Program errors        :  Operand addressing errors.

**8.1.5.7.  Load Bound (LDB)**                          Function Code  :  76

Operand length        :  32 bits

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

CC : Unaltered

Program errors : Operand addressing errors

### 8.1.5.8 Modify DR (MODD)                              Function Code : 16

Operand length : 32 bits

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

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

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

Notes: a) If the descriptor is type 0 or 2 and USC is not set, or type 3 subtype 32 or 33, the operand is scaled appropriately before addition to the address field. If the descriptor is type 0 with size code 0, the least significant 3 bits of the oeprand are ignored because of the scaling operation.
b) If the operand, regarded as unsigned, i.e. positive, is not less than the original contents of the bound/length field only the least significant 24 bits of the difference are left in that field. In such cases, if the descriptor is type 0 or 2 with BCI not set, or type 3 subtype 32 (bounded code) a program error condition (Bound Check, maskable) is generated.
This does not apply to String descriptors

CC : Unaltered

Program errors : Operand addressing errors
Indirect address form (see 7.4.2.12.2)
Bound significant, and not greater than operand (see 7.4.2.5.1)
Descriptor is System Call (se 7.4.2.10.11)
Descriptor is invalid (see 7.4.2.10.9 or 10)

8.1.5.9  Validate Address  (VAL)                    **Function Code  :  10**

The following applies to AMLO

Operand length      :  32 bits

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

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

If the descriptor is valid, the address of the last word or byte in the
field pointed at by the descriptor is calculated (without altering DR).
This address is calculated from the address of the first byte as follows:
Type 0:  Add (Bound-1) (scaled if USC = 0) then if size = 64 add 4 bytes;
if size = 128 bits, add 12 bytes (word alignment assumed).
Type 1:  Add (Length-1)
Type 2:  As for type 0, with size = 64 bits.
If the address thus calculated has a different segment number from the
initial address, or if it has the same segment number but lies beyond the
upper limit of that segment, or if it has the same segment number as SSN
but is not less than SSN + LNB, CC is set to 3 (in the last case, if it is
less than SSN + LNB, and the initial address is also in the stack segment,
CC is set to 0).
Otherwise CC is set to indicate whether read or write access to that
segment, at the access level given by the operand, is permitted.  If
access is not permitted CC is set to 3.  The second word of the segment
table entry is ingored.  Indirect address forms are not permitted.  DR is
unaltered.
ACR is unaltered

CC                    :  0    Read and write access permitted at specified
                              level
                         1    Read access permitted, write inhibited
                         2    Read access inhibited, write permitted
                         3    Descriptor invalid, or field crosses segment
                              boundary, or neither read nor write permitted
                              (includes case of invalid segment number).

Program errors        :  Operand addressing errors
                         Indirect address form (see 7.4.2.12.2)

The following applies to AML1.

Operand length    :  32 bits

Description       :  This instruction is designed to investigate whether
a descriptor provided to a called procedure is valid at the access level
of the caller. The descriptor is taken from DR and must be Type 0, 1 or 2
or Type 3, subtype 63.
If type 0 or 2 the descriptor must be bounded. Bits 8-11 of the operand
are interpreted as the access control key (normally held in ACR) of the
caller; the remaining operand bits are ignored.
The descriptor is first checked for 'Type Null' which is defined to be
Type 3, subtype 63, and if so CC is set to 2 and the instruction
terminates.
The descriptor is then checked for validity. If 'invalid', CC is set to 3
and the instruction terminates. 'Invalid' includes any of the following

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

Check (e) is implemented as follows:-
The address of the last byte or word in the field pointed at by the
descriptor is calculated (without altering DR) from the address of the
first byte as follows
Type 0 or 2: Add (Bound-1) scaled if USC=0, then add 4 bytes if size
code=6, add 12 bytes if size code=7 (word alignment assumed).
Type 1:  Add (Length-1)
The address in the descriptor points at an 'initial' segment, the
calculated address points at a 'final' segement and there may be
'intermediate' segments between the above.

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

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

Indirect forms are not permitted. DR and ACR are unaltered.

CC                :  0    Read and Write access permitted at specified
                          level
                   1    Read access permitted, write access inhibited
                   2    Descriptor is type 3, subtype 63 (Null)
                   3    Descriptor invalid OR read access inhibited

**ICL**

**Product specification**

**Company restricted**

**PSD** 2.5.1

Issue 6/0

Sheet 101

Program errors : Operand addressing errors
Indirect address forms (see 7.4.2.12.2)

### 8.1.5.10 Increment Address (INCA)      Function Code : 14

Operand length : 32 bits

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

CC : Unaltered

Program errors : Operand addressing errors
Indirect address form (see 7.4.2.12.2)

### 8.1.5.11 Start Significance (SIG)      Function Code : 28

Operand length : 64 bits

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

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

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

Note: This instruction is designed for use in conjunction with 'Suppress
and Unpack' (8.3.3.10)

CC : 0   Not used
     1   CC originally 0, and descriptor stored; or CC
        originally 1
     2   CC originally 2
     3   CC originally 3

Program errors : Operand addressing errors
Literal operand (see 7.4.2.12.1)
Indirect operand form (see 7.4.2.12.2)
Significant part of operand truncated (see
7.4.2.6.0)

## 8.2 Computational Functions

### 8.2.1 List of Instructions

By 'computational functions' is meant those instructions which perform floating-point, fixed point, logical and decimal arithmetic operations in the accumulator. With most of these operations the operand interacts with the contents of ACC and the result is left in ACC.

All instructions use the primary format described in 6.1. Rules for operand access are given in 6.3.

Several functions (e.g. Add, Subtract) are common to the different arithmetic types, i.e. they perform essentially the same operation and differ only in the way they interpret the data format. This may be reflected in the function decoding.

A few special functions are provided for converting data from one format to another (these are listed under the arithmetic type of the source data).

Fixed-point and logical operations with ACS - 128 bits are not provided. Further restrictions are mentioned under individual instructions.

Instructions common to several types:

| | Floating | Fixed | Logical | Decimal |
| --- | --- | --- | --- | --- |
| Add | X | X | X | X |
| Subtract | X | X | X | X |
| Reverse subtract | X | X | X | X |
| Compare | X | X | X | X |
| Shift (scale) | X | X | X | X |
| Multiply | X | X | - | X |
| Divide | X | X | - | X |
| Reverse divide | X | X | - | X |
| Remainder divide | - | X | - | X |
| Divide double | X | - | - | - |
| Multiply double | X | X | - | X |

(X provided    - Not provided)

Additional floating-point instructions:
    Fix

Additional fixed-point instructions:
    Convert to decimal
    Float

Additional logical instructions:
    And
    Or
    Not equivalent
    Rotate
    Shift 32 bits
    Shift while zero

Additional decimal instructions
Convert to binary

## 8.2.2. Data Formats

### 8.2.2.1 Floating-point format

The format for a 32-bit, 64-bit, or 128-bit floating-point number is as follows:

| S | Biased exponent | (Modulus of) Fraction | Continuation of fraction |
| --- | --- | --- | --- |
| 0 1 | 7 8 | 31 32 | 63 |

| ///////////////////// | Continuation of fraction |
| --- | --- |
| 64           71 72 | 127 |
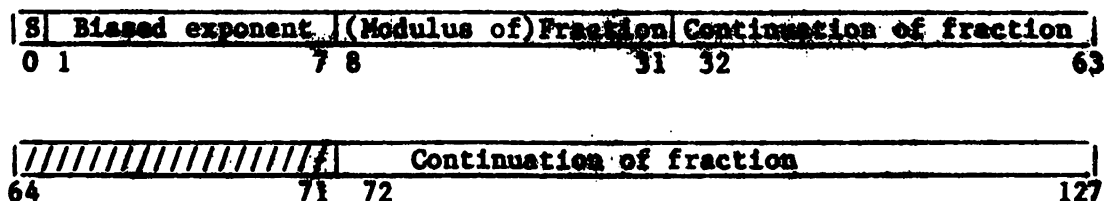
This is also the format used in IBM 360 and 370 computers. Bit 0 contains the sign bit, S, of the number; 0 for positive numbers, 1 for negative numbers.

Bits 1-7 contain a 'characteristic' (IBM terminology) or biased exponent, C, in the range 0-127. This represents a true hexadecimal exponent biased by +64.

Bits 8 onward (excluding bits 64-71 in the 128 bit case; bits 63 and 72 are effectively adjacent) contain an unsigned fraction, F. The binary point of this fraction lies to the left of bit 8, so F ranges in value from 0 to just less than 1.

F has 6, 14 or 28 hexadecimal digits according to whether the format is 32-bit, 64-bit or 128-bit.

If S = 0 the value of the number is $F \times 16^{c-64}$
If S = 1 the value of the number is $-F \times 16^{c-64}$

Thus to change the sign of a number only S need be changed. This sign convention is called sign and modulus, and differes from that used in fixed-point arithmetic.

The contents of bits 64-71 of a 128-bit number are ignored in data; however in results (except when the result is 'true zero' in which case all 128 bits are zeroes) bit 64 is made a copy of the sign bit (bit 0) and bits 65-71 are made to contain a value 14 less than the characteristic in bits 1-7. If the latter is less than 14, 128 is added to the difference to make it positive.

The results of all floating point arithmetic operations, except where otherwise stated, are normalised. This implies that F, if non-zero, is made to lie in the range

$$\frac{1}{16} \leqslant F < 1$$

So that the first hexadecimal digit of F is non-zero  Normalisation is

achieved by shifting F to the left by the requisite number of hexadecimal places and subtracting this number from the characteristic. If F is zero, normalisation is achieved by making all 32, 64 or 128 bits zeroes ('True zero').

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

Non-normalised numbers are acceptable as operands for all floating-point operations, and will give correct results, albeit with less precision than might have been the case with normalsied operands. For this reason the use of normalised operands is recommended. A non-normalised number with zero fraction may have either or both of S and C non-zero and will be operated on as described in the individual instruction descriptions.

When a result cannot be expressed in normalised form with a true exponent greater than $- 65$ i.e. it would require a negative characteristic, underflow occurs and the result is made true zero, and unless the condition is masked an interrupt ensues.

When a normalised result requires a true exponent larger than $+63$, floating point overflow occurs. OV is set, and if this condition is not masked an interrupt ensues. In either case the result is given with a normalised fraction and a characteristic 128 less than it should be.

### 8.2.2.2 Fixed point format

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

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

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

For a 64-bit number the contributions are:

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

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

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

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

Fixed-point numbers more than 64 bits long may be operated on by splitting them into 32-bit portions and using those, singly or in pairs, as operands for the instructions. Add logical (8.2.5.1), Subtract logical (8.2.6.2), Multiply double (8.2.4.5) and Remainder dividle (8.2.4.6).

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

### 8.2.2.3 Logical Formats

Logical operations on 32 or 64 bit items treat them either as strings of bits with no numerical significance, or as unsigned (i.e. positive) fixed-point numbers. In this case the contributions of individual bits are as described in 8.2.2.2 except that bit 0, if non-zero, contributes $+2^{31}$ (32-bit format) or $+2^{63}$ (64-bit format). Overflow cannot occur with logical operations.

### 8.2.2.4 Decimal format

Decimal numbers in the accumulator or store occupy a string of consecutive bytes. The less significant 4 bits of the rightmost byte contain the sign of the number, as follows:
1011 or 1101 - negative
1010, 1100, 1110, 1111 - positive
any other value - undefined.

The more significant 4 bits of the rightmost byte contain the least significant digit of the number, in binary-coded-decimal form, and the remaining digits of the number, in 4-bit binary-coded-decimal form, are held two to a byte in the remaining bytes, each digit being one decimal place more significant than the one held in the adjacent half byte on the right. All decimal numbers are integers. Thus:

| Most Significant | | | | Least Significant | | |
|---|---|---|---|---|---|---|
| Digit | Digit | Digit | | Digit | Digit | Sign |
| 4 | 4 | 4 | | 4 | 4 | 4 |
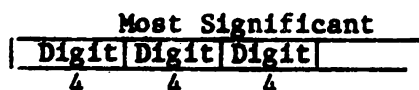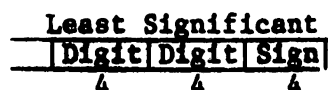
Leftmost byte                          Rightmost byte

Thus a decimal number in this form always contains an odd number of digits. Decimal numbers in the accumulator have 7, 15, or 31 digits, corresponding to ACC sizes 32, 64 or 128 bits.

**ICL**

**Product specification**

**Company restricted**

**PSD** 2.5.1

Issue 6/0

Sheet 106

Results generated by decimal operations have sign codes as follows:

Negative 1101
Positive 1100

This does not apply to DSH, SUPK or PK (see instruction descriptions). Zero results will have sign code 1100 except possibly after one of these operations or following overflow which occurs when the numerical part of the result is too large for the accumulator. This will cause OV to be set and interrupt to occur unless the decimal overflow condition is masked.

The values of the numeric digits in operands are not checked, and the results of decimal operations are undefined when non-numeric digits (i.e. in the range 1010 - 1111) are present in operands. However operands in the correct form will always give results in the correct form.

### 8.2.3 Floating-point instructions

#### 8.2.3.1 Floating ADD (RAD)           Function Code : F0
       Floating Subtract (RSB)           Function Code : F2

Operand length      : ACS

Description             : The operand is added to, or subtracted from the contents of ACC, and the normalised result is left in ACC. ACS may be 32, 64 or 128 bits.
Overflow or underflow may occur as described in 8.2.2.1. OV is cleared if overflow does not occur.
Floating-point subtraction is performed by inverting the sign bit (only) of the operand and then following the rules for floating-point addition, below.

Floating-point addition is performed in three stages:-

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

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

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

2) The two signed fractions, including their guard digitis, are added algebraically to form an intermediate sum in sign-and-modulus form. The intermediate sum has an associated sign bit, a possible carry bit and,

including the guard digit, 7, 15 or 29 hexadecimal digits.

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

- If all digits and the carry bit of the intermediate sum are zero, a true zero result is generated.

- If the carry bit is non-zero, the intermediate sum is shifted one hexadecimal place to the right (generating a 1 in the most significant hexadecimal digit position), and its carry bit and guard digit are removed, to form the fractional part of the result. 1 is added to the characteristic (this may cause overflow as described in 8.2.2.1) to form the result characteristic. The sign bit of the result is that associated with the intermediate sum.

- If the carry bit is zero, but one or more digits of the intermediate sum are non-zero, the latter is shifted left until the most significant hexadecimal digit is non-zero. Following each hexadecimal shift, zero is inserted in the guard digit position and 1 is subtracted from the characteristic. Should this cause the characteristic to become negative, underflow occurs as described in 8.2.2.1 ( and a true zero result is generated.) If the characteristic does not become negative, the result comprises the sign bit associated with the intermediate sum, the final characteristic and the normalised intermediate sum with the carry bit and guard digit removed.

- If ACS = 128 bits, bits 64-71 of the result are generated as described in 8.2.2.1.

        CC                  : Unaltered

        Program errors : Operand addressing errors
                          Floating overfow (unless masked)(see 7.4.2.0)
                          Floating underflow (unless masked)(see 7.4.2.1)

## 8.2.3.2 Floating Reverse Subtract (RRSB)                Function Code  :  F4

        Operand length      :  ACS

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

        CC                  :  Unaltered

        Program errors      :  Operand addressing errors
                               Floating overflow (unless masked)(see 7.4.2.0)
                               Floating underflow (unless masked)(see 7.4.2.1)

### 8.2.3.3. Floating Compare (RCP)                    Function Code : F6

Operand length     : ACS

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

The comparison is performed effectively by carrying out the first two stages of the Floating subtract operation (8.2.3.1) and examining the intermediate sum. Equality is indicated if and only if the carry bit and all digits (including the guard digit) of the latter are zero. Otherwise the setting of CC depends on the sign bit associated with the intermediate sum.

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

CC                 : 0 Equality (Intermediate sum zero)
                     1 ACC less than operand
                     2 ACC greater than operand
                     3 Not used

Program errors     : Operand addressing errors

### 8.2.3.4 Scale (RSC)                               Function Code : F8

Operand length     : 32 bits

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

If the fractional part of the number is zero, a true zero result is generated. If it is non-zero, and if after adding i and subtracting the amount of normalising shift, the characteristic exceeds 127, overflow occurs as described in 8.2.2.1. Similarly, if the fraction is non-zero, and after adding i, or subsequently after subtracting the amount of normalising shift, the characteristic becomes negative, underflow occurs and a true zero result is generated. OV is cleared if overflow does not occur.

If ACS = 128 bits, bits 64-71 of the result are generated as described in 8.2.2.1.

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

CC                              : Unaltered

Program errors                  : Operand addressing errors
                                  Floating overflow (unless masked)(see 7.4.2.0)
                                  Floating underflow (unless masked)(see 7.4.2.1.)

### 8.2.3.5 Floating Multiply (RMY)                    Function Code : FA

Operand length      : ACS

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

If the fractional part of either mulitplier or mulitplicand is zero, a
true zero is generated, and neither overflow nor underflow can occur.  If
neither of the fractional parts is zero, the fractional part of the result
if that which would be produced by forming the true, double-length,
product of the fractions, associating with it a characteristic which is
the sum of the original characteristics, minus 64, normalising the product
and then truncating it to half length.  Overflow or underflow occur, as
described in 8.2.2.1, if and only if the _final_ characteristic exceeds 127
or is negative; in the latter case a true zero result is generated.  When
the result is not zero the sign bit is determined by the rules of algebra.

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

OV is cleared if overflow does not occur.

If ACS = 128 bits, bits 64-71 of the result are generated as described in
8.2.2.1.

CC                              : Unaltered

Program errors                  : Operand addressing errors
                                  Floating overflow (unless masked)(se 7.4.2.0)
                                  Floating underflow (unless masked)(see 7.4.2.1)

### 8.2.3.6 Floating Divide (RDV)                       Function Code : BA

Operand length      : ACS

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

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

In order to ensure that non-normalised operands can be used the dividend and divisor fractions are first normalised ignoring overflow or underflow at this stage); after normalisation the latter may be scaled one place up to ensure that the quotient fraction does not overflow. In this case not more than one left shift will be required to normalise the quotient and the quotient should therefore be developed to 7, 15 or 29 digits as appropriate and the guard digit dropped after normalisation.

The characteristic associated with the quotient fraction before normalisation is the difference of the characteristics of the normalised dividend and divisor, plus 64, (65 if the divisor fraction is scaled up). Overflow or underflow occur as described in 8.2.2.1 if and only if the final characteristic exceeds 127 or is negative; in the latter case a true zero result is generated. When the result is not zero the sign bit is determined by the rules of algebra.

OV is cleared if overflow does not occur.

If ACS = 128 bits, bits 64-71 of the result are generated as described in 8.2.2.1.

CC                          : Unaltered

Program errors              : Operand addressing errors
                              Zero divide (unless masked)(see 7.4.2.4)
                              Floating overflow (unless masked)(see 7.4.2.0)
                              Floating underflow (unless masked)(see 7.4.2.1)

### 8.2.3.7 Floating Reverse Divide (RRDV)                     Function Code   :  BC

Operand length         : ACS

Description            : This operation is identical to Floating divide (8.2.3.6) except that the quotient left in ACC is formed by dividing the operand by the contents of ACC.

CC                     : Unaltered

Program errors         : Operand addressing errors
                         Zero divide (unless masked)(see 7.4.2.4)
                         Floating overflow (unless masked)(see 7.4.2.0)
                         Floating underflow (unless masked)(see 7.4.2.1)

### 8.2.3.8 Floating Divide Double (RDVD)                      Function Code   :  BE

Operand length         : Half ACS

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

The operation is otherwise identical to Floating divide (8.2.3.6) except that the dividend fraction is longer. All digits of the latter participate.

CC                          :   Unaltered

Program errors              :   Zero divide (unless masked)(see 7.4.2.4)
                                Floating overflow (unless masked)(see 7.4.2.0)
                                Floating underflow (unless masked)(see 7.4.2.1)
                                ACS = 32 bits. (see 7.4.2.13.3)

### 8.2.3.9. Floating Multiply Double (RMYD)          Function Code  :  FC

Operand length        :   ACS

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

Except that the true product of the fractions is not truncated (and the
result is therefore exact) this operation is otherwsie identical to
Floating mulitply (8.2.3.5)

CC                          :   Unaltered

Program errors              :   Operand addressing errors
                                Floating overflow (unless masked) (see7.4.2.0)
                                Floating underflow (unless masked)(see 7.4.2.1)
                                ACS = 128 bits (see 7.4.2.13.5)

### 8.2.3.10 FIX (FIX)                                 Function Code  :  B8

Operand length        :   32 bits

Description           :   The exponent, and sign and fraction, of the
floating-point number in ACC are separated; the former is adjusted and
stored as a 32 bit signed integer in the operand location, the latter left
in ACC as a signed (2's complement) integer. If the number in ACC had a
zero fraction, zero exponent is stored.

ACS may be 32, 64 or 128 bits. If it was 128 bits ACS is halved and the
least significant 14 digits of the fraction are lost, CC being set to
indicate the nature of the lost bits.

OV is cleared.

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

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

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

CC                          :  0  No non-zero bits lost
                               1  (ACS = 128 bits) lost portion less than 1/2
                               2  (ACS = 128 bits) lost portion not less than 1/2
                               3  Not used

Program errors              :  Operand addressing errors
                               Literal operand (see 7.4.2.12.1)
                               Significant part of operand truncated
                               (see 7.4.2.6.0)

## 8.2.4 Fixed-point instructions

### 8.2.4.1 ADD (IAD)                          Function Code  :  E0
         SUBTRACT (ISB)                        Function Code  :  E2

Operand length          :  ACS

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

Fixed point overflow occurs if the result lies outside the range of representable numbers (see 8.2.2.2); when it occurs OV is set and, if not masked, an interrupt ensues. The result left in ACC in this case is the least significant 32 or 64 bits of the true sum or difference. Overflow can only occur when adding numbers with like signs or subtracting numbers with opposite signs. OV is cleared if overflow does not occur.

CC                      :  Unaltered

Program errors          :  Operand addressing errors
                           Fixed overflow (unless masked)(see 7.4.2.2)
                           ACS = 128 bits (see 7.4.2.13.0)

### 8.2.4.2 Reverse Subtract (IRSB)                  Function Code    E4

Operand length          :  ACS

Description             :  Exactly as for Subtract (8.2.4.1) except that the difference left in ACC is formed by subtracting the original contents of ACC from the operand.

CC                         :  Unaltered

Program errors             :  Operand addressing errors
                              Fixed overflow (unless masked)(see 7.4.2.2)
                              ACS = 128 bits (see 7.4.2.12.0)

### 8.2.4.3 Compare (ICP)                    Function Code  :  E6

Operand length     :  ACS

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

Equality is indicated if all bits are equal. ACC less than the operand is
indicated if, when conducting a left-to-right scan of the bits of both, the
first non-equivalent pair of bits occur when the ACC bit concerned is a 1
(if bit 0) or 0 (any subsequent bit), and ACC greater than the operand in
the same circumstances when the ACC bit concerned takes the opposite
values.

ACS = 32 or 64 bits is assumed.

CC                 :  0   Equality
                      1   ACC less than operand
                      2   ACC greater than operand
                      3   Not used

Program errors     :  Operand addressing errors
                      ACS = 128 bits (see 7.4.2.13.0)

### 8.2.4.4 Arithmetic Shift (ISH)              Function Code  :  E8

Operand length     :  32 bits

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

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

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

**ICL**

**Product specification**

**Company restricted**

**PSD** 2.5.1

Issue 6/0

Sheet 114

CC                        :  0  i less than 0; all bits shifted off the
                                right of ACC were 0's.
                          1  i less than 0; last bit shifted off the
                             right of ACC = 0 (some 1's)
                          2  i less than 0; last bit shifted off the right
                             of ACC = 1
                          3  i not less than 0

Program errors           :  Operand addressing errors
                             Fixed overflow (unless masked)(see 7.4.2.2)
                             ACS = 128bits (see 7.4.2.12.0)

### 8.2.4.5  Multiply (IMY)                      Function Code  :  EA

Operand length           :  ACS

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

ACS = 32 or 64 bits assumed.
CC                       :  Unaltered
Program errors           :  Operand addressing errors
                             Fixed overflow (unless masked)(see 7.4.2.2)
                             ACS = 128 bits (see 7.4.2.13.0)

### 8.2.4.6 Divide (IDV)  Function Code : AA

Operand length : ACS

Description : The contents of ACC are divided by the operand and the unrounded quotient left in ACC, all three being signed integers. The rules for determining the quotient are as for Remainder Divide (8.2.4.8), i.e. quotient times divisor is numerically not greater than dividend.

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

ACS = 32 or 64 bits assumed.

CC : Unaltered

Program errors : Operand addressing errors
Zero divide (unless masked)(see 7.4.2.4.)
Fixed overflow (unless masked)(see 7.4.2.2)
ACS = 128 bits (see 7.4.2.13.0)

### 8.2.4.7 Reverse Divide (IRDV)  Function Code : AC

Operand length : ACS

Description : This operation is identical to Divide (8.2.4.6) except that the operand is divided by the contents of ACC rather than the other way round. If overflow occurs ACC will contain $-2^{31}$ or $-2^{63}$, depending on ACS.

ACS = 2 or 64 bits is assumed.

CC : Unaltered

Program errors : Operand addressing errors
Zero divide (unless masked)(see 7.4.2.4.)
Fixed overflow (unless masked)(see 7.4.2.2.)
ACS = 128 bits (see 7.4.2.13.0)

### 8.2.4.8 Remainder Divide (IMDV)  Function Code : AE

Operand length : ACS

Description : The quantity in ACC is divided by the operand, the quotient is left in ACC and the remainder is stacked All these quantities are signed integers with length ACS; as a result of stacking the remainder SF is incremented.
ACS = 32 or 64 bits is assumed.
The remainder is numerically less than the divisor and, if non-zero, has the same sign as the dividend. CC is set to facilitate obtaining a remainder which obeys the rules for the PL/1 'Mod' function.

**ICL**

**Product specification**

**Company restricted**

**PSD** 2.5.1

Issue 6/0

Sheet 116

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

| | | |
|---|---|---|
| CC | : | 0 Remainder zero, or remainder greater than 0, divisor greater than 0 |
| | | 1 Remainder greater than 0, divisor less than 0 |
| | | 2 Remainder less than 0, divisor greater than 0 |
| | | 3 Remainder less than 0, divisor less than 0 |

| | | |
|---|---|---|
| Program errors | : | Operand addressing errors |
| | | Zero divide (unless masked) (see 7.4.2.4) |
| | | Fixed overflow (unless masked)(see 7.4.2.2) |
| | | ACS = 128 bits (7.4.2.13.0.) |

### 8.2.4.9 Multiply Double  (IMYD)                    Function Code : EC

Operand length      : 32 bits (=ACS)

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

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

| | | |
|---|---|---|
| Program errors | : | Operand addressing errors |
| | | ACS = 64 bits or 128 bits (see 7.4.2.13.5) |

### 8.2.4.10 Convert to Decimal (CDEC)                    Function Code : EE

Operand length      : Not applicable. Literal must be specified

Description      : The contents of ACC, a signed fixed-point integer, are converted to standard decimal form in ACC (see 8.2.2.4). ACS is doubled. OV is cleared.
ACS = 128 bits is not permitted.

CC                    : Unaltered

Program errors      : ACS = 128 bits

8.2.4.11 Float (FLT)                                    Function Code : AS

Operand length        : 32 bits

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

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

- ACS is doubled, the previous  contents of ACC now being placed in the
more significant half and the less significant half made zero.

- The least significant 8 bits of the operand are placed in an
intermediate register.  The value of the most significant of these 8 bits
is recorded for future reference.  The quantity in the register is
incremented by 72 (ACS = 64 bits) or 80 (ACS = 128bits) - this is now the
'intermediate characteristic'.

- The contents of ACC are shifted right arithmetically 8 binary places. If
bit 0 of ACC was originally a 1, the contents of ACC are negated (to form
the modulus of the fraction ) and bit 0 of ACC is made 1.  Bits 8 onward
now form the 'intermediate fraction'; bit 0 is the sign bit.

- The intermediate fraction is now normalised by shifting it up one
hexadecimal place (4 bits) at a time until bits 8-11 are not all zeroes. 1
is subtracted from the intermediate characteristic for every hexadecimal
shift.

- The least significant 7 bits of the intermediate chartacteristic
overwrite bits 1-7 of ACC.  If CS = 128 bits, the contents of the least
significant 64 bits of ACC are shifted 8 binary places to the right and a
copy of the characteristic in bits 1-7 minus 14 (or plus 114, if the
latter is less than 14) inserted in bits 65-71 of ACC.  Bit 63 is made the
same as bit 0.  OV is cleared.

- The most significant bit of the 8-bit intermediate characteristic should
be 0.  If it is a 1, underflow or overflow occur as described in 8.2.2.1,
depending on whether the corresponding bit of the original operand was 1
or 0, respectively.

If underflow occurs ACC is made 0.  If overflow occurs OV is set.

CC                    : Unaltered

Program errors        : Operand addressing errors
                        Floating overflow (unless masked)(see 7.4.2.0)

Floating underflow (unless masked)(see 7.4.2.13)
ACS = 128 bits (see 7.4.2.13.2)


## 8.2.5 Logical Instructions

### 8.2.5.1 Logical ADD (UAD)                                    Function Code : CO

Operand length      : 32 bits

Description         : The effect of this instruction is to leave in ACC
the least significant 32 bits of the sum of the operand and the original
contents of ACC, both regarded as unsigned integers. OV is cleared. CC is
set to indicate whether or not carry occurred out of ACC bit 0.

The operation is only defined for ACS = 32 bits.

If this operation is performed on words with dummy (zero) sign bits (i.e.
portions of multiple length quantities) they should be left shifted to
remove those bits.

CC                  : 0   No carry
                      1   Carry
                      2   Not used
                      3   Not used

Program errors      : Operand addressing errors
                      ACS = 64 or 128 bits (see 7.4.2.13.1)

### 8.2.5.2. Logical Subtract (USB)                             Function Code : C2

Operand length      : 32 bit

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

CC                  : 0   No carry indicates 'borrow' into bit 0 in
                          performing subtraction)
                      1   Carry (indicates no 'borrow'.  Includes the
                          case where complementing causes carry,
                          i.e. operand = 0)
                      2   Not used
                      3   Not used

Program errors      : Operand addressing errors
                      ACS = 64 or 128 bits (see 7.4.2.13.1)


### 8.2.5.3 Logical Reverse Subtract (URSB)                     Function Code : C4

Operand length      : 32 bit

**ICL**

**Product specification**

**Company restricted**

**PSD**    2.5.1

Issue    6/0

Sheet    119

Description        : This operation is identical to Logical Subtract (8.2.5.2.) except that the result is formed by adding the 2's complement of the original contents of ACC to the operand. OV is cleared. ACS = 32 bits is assumed.

CC                : 0    No carry (indicates 'borrow')
                     1    Carry (indicates no 'borrow'; includes the case where ACC was 0, so complementing causes carry)
                     2    Not used
                     3    Not used

Program errors     : Operand addressing errors
                         ACS = 62 or 128 bits (7.4.2.13.1)

### 8.2.5.4. Logical Compare (UCP)                 Function Code   :   C6

Operand length     : ACS

Description        : The operand is compared with the contents of ACC, CC being set to indicate the result of the comparison. Equality implies equivalence in every bit position. ACC less than operand is indicated if, when scanning the bits of both from left to right, the first non-equivalent pair of bits occurs when the ACC bit concerned is a 0, ACC greater than operand where it is a 1.

ACC and OV are unaltered.

ACS = 32 or 64 bits is assumed.

CC               . : 0    Equality
                     1    ACC less than operand
                     2    ACC greater than operand
                     3    Not used

Programming errors   : Operand addressing errors
                         ACS = 128 bits (see 7.4.2.13.0)

### 8.2.5.5 Logical Shift (USH)                   Function Code   :   C8

Operand length     : 32 bits

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

The remaining operand bits are ignored.
ACS = 32 or 64 bits is assumed.

CC                 : Unaltered

Program errors     : Operand addressing errors
                         ACS = 128 bits (see 7.4.2.13.0)

8.2.5.6 And (AND)                    Function Code  :  8A
       Or  (OR)                     Function Code  :  8C
       Not Equivalent (NEQ)         Function Code  :  8E

Operand length      :  ACS

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

| Original ACC bit | Operand bit | Result bits | | |
| --- | --- | --- | --- | --- |
| | | AND | OR | NOT EQIV. |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 |

OV is cleared.

ACS = 32 or 64 bits is assumed.

CC                  :  Unaltered

Program errors      :  Operand addressing errors
                       ACS = 128 bits (see 7.4.2.13.0)

8.2.5.7. Rotate (ROT)                    Function Code  :  CA

Operand length      :  32 bits

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

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

CC                  :  Unaltered

Program errors      :  Operand addressing errors
                       ACS = 128 bits (see 7.4.2.13.0)

### 2.5.8 Shift 32 Bits (SHS)

Function Code : CV

Operand length : 32 bits

Description : The least significant 32 bits of the contents of ACC are shifted logically in exactly the same way as for Logical Shift (8.3.5.5); in fact if ACS = 32 bits the instructions are identical. If ACS =64 bits, the more significant 32 bits of ACC are unaltered. Zeroes are inserted in the least (leftward shift) or most (rightward shift) significant bit position of the 32 bits shifted as shifting proceeds. OV is cleared.
ACS = 32 or 64 is assumed.

CC : Unaltered

Program errors : Operand addressing errors
ACS = 128 bits (see 7.4.2.13.0)

### 8.2.5.9 Shift While Zero (SHZ)

Function Code : CE

Operand length : 32 bits

Description : The contents of ACC, if non-zero, are shifted logically leftward until bit 0 is a 1. The number of binary places shifted is stored as a 32 bit positive integer in the operand location. OV is cleared.

If ACC is zero, a zero is stored. This condition may be detected by testing ACC for zero after the operation.

ACS = 32 or 64 bits is assumed.

CC : Unaltered

Program errors : Operand addressing errors
Literal operand (see 7.4.2.12.1)
Non-zero bits of stored item truncated
(see 7.4.2.6.0)
ACS = 128 bits (see 7.4.2.13.0)

### 8.2.6 Decimal Instructions

### 8.2.6.1 Decimal Add (DAD)

Function Code : D0

Decimal Subtract (DSB)

Function Code : D2

Operand length : ACS

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

CC                          : Unaltered

Program errors          : Operand addressing errors
                                : Decimal overflow (unless masked) (see 7.4.2.3.)

### 8.2.6.2 Decimal Reverse Subtract (DRSB)             Function Code : D4

Operand length        : ACS

Description           : As for Decimal subtract (8.2.6.1) except that the difference left in ACC is formed by subtracting the original contents of ACC from the operand.

CC                          : Unaltered

Program errors          : Operand addressing errors
                                Decimal overflow (unless masked) (see 7.4.2.3)

### 8.2.6.3 Decimal Compare (DCP)                       Function Code : D6

Operand length        : ACS

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

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

CC                       : 0   Equality
                                   1   ACC less than operand
                                   2   ACC greater than operand
                                   3   Not used

Program errors          : Operand addressing errors

### 8.2.6.4 Decimal Shift (DSH)                        Function Code : D8

Operand length        : 32 bits

Description           : The least significant 7 bits of the operand specify the amount by which the contents of ACC are to be shifted. This amount is

**ICL**

**Product specification**

**Company restricted**

**PSD** 2.5.1

Issue 6/0

Sheet 123

interpreted as a signed integer in the range −64 to +63. Other bits of the operand are ignored. If the integer is positive (=1) all but the least significant 4 bits of ACC are shifted 4i binary places to the left. The sign digit is unaltered. If any of the bits shifted off the leftmost end of ACC are 1's, OV is set and interrupt occurs unless decimal overflow is masked. OV is cleared if all bits shifted off are zeroes.

   Zero bits are inserted in the bit position adjacent to the sign digit as shifting proceeds. If the amount of shift is negative (= −1) all except the least significant 4 bits of ACC are shifted 4i binary places to the right. OV is cleared. The sign digit is unaltered. Bits shifted out of position to the left of the sign are lost. Zeroes are inserted at the most significant end of ACC.

CC                      : Unaltered

Program errors          : Operand addressing errors
                          Decimal overflow (unless masked)(see 7.4.2.3)


**8.2.6.5 Decimal Multiply (DMY)**                    Function Code  :  DA

   Operand length       : ACS

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

   CC                   : Unaltered

   Program errors       : Operand addressing errors
                          Decimal overflow (unless masked)(see 7.4.2.3)


**8.2.6.6. Decimal Divide (DDV)**                     Function Code  :  9A

   Operand length       : ACS

   Description          : The contents of ACC are divided by the operand and the unrounded quotient left in ACC. The rules for determining the quotient are those for Decimal Remainder Divide (8.2.6.8). OV is cleared.

   If the divisor is zero (i.e. all its numeric digits are zeroes) the result is undefined (but OV is cleared), and the zero divide interrupt occurs unless masked.

   CC                   : Unaltered

   Program errors       : Operand addressing errors
                          Zero divide (unless masked)(see 7.4.2.4)

**8.2.6.7 Decimal Reverse Divide (DRDV)**            Function Code  :  9C

   Operand length       : ACS

Description : The operation is exactly as for Divide (8.2.6.6) except that the operand is the dividend and the contents of ACC the divisor.

CC : Unaltered

Program errors : Operand addressing errors
Zero divide (unless masked)(see 7.4.2.4)


### 8.2.6.8 Decimal Remainder Divide (DMDV)       Function Code : 9E

Operand length : ACS

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

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

CC is set to facilitate the evaluation of the PL1 'Mod' function.

If the divisor is zero (all its numeric digits are zero) the quotient, the remainder (which is stacked) and the setting of CC are all undefined (but OV is cleared), and the zero divide interrupt occurs unless masked.

CC : 0 Remainder zero, or remainder greater than 0, divisor greater than 0
1 Remainder greater than 0, divisor less than 0
2 Remainder less than 0, divisor greater than 0
3 Remainder less than 0, divisor less than 0

Program errors : Operand addressing errors Zero divide (unless masked)(see 7.4.2.4)


### 8.2.6.9 Decimal Multiply /Double (DMYD)       Function Code : DC

Operand length : ACS

Description : The double-length product of the contents of ACC and the operand is left in ACC, ACS being doubled in the course of the operation. OV is cleared. ACS = 128 bits is not permitted.

CC : Unaltered

Program errors : Operand addressing errors
ACS = 128 bits (see 7.4.2.13.5)


### 8.2.6.10 Convert to Binary (CBIN)       Function Code : DE

Operand length : Not applicable. Literal must be specified.

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

CC        : Unaltered

Program errors        : Fixed overflow (unless masked)(see 7.4.2.2)

## 8.3 Store-to-store instructions

### 8.3.1. Introduction

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

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

The number of bytes involved in a store-to-store operation (referred to as L) is specified either explicitly in the instruction format, or in the length field of the descriptor in DR. Instructions are 16 or 32 bits long, and use the secondary format described in 6.1. The format of the first 16 bits is as follows:

| Function | h | q | n |
|----------|---|---|---|
| 7 | 1 | 1 | 7 |

$h = 0$ : $L = n + 1$ $(1 \leqslant L \leqslant 128)$
$h = 1$ : $L =$ (Length field of DR) $(0 \leqslant L \leqslant 224)$ (n field reserved)
$q = 0$ : 16-bit instruction
$q = 1$ : 32-bit instruction

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

| Mask | Literal/Filler |
|------|----------------|
| 8 | 8 |

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

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

**ICL**  Product
specification

Company
restricted

Issue      6/0

Sheet      127

ignored.  If L=0 the instruction will be interpreted as a null operation, the contents of the (ACC) and (DR) strings being ignored and unaltered (virtual store interrupts cannot occur).

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

1) DR must contain a type 0 or type 1 descriptor with size Code 3, or an escape descriptor, else see 7.4.2.10.6.

2) When L = n + 1 (literal), L not less than (length field of DR else see 7.4.2.11.0.

3) When ACC contains a descriptor, it must be of the correct type (type 0 or 1 with size code 3, except for Table check and Table translate), and ACS = 64 bits (for some instructions (ACC) is only used with the 16 bit instruction format), else see 7.4.2.10.7 or 8.

4) When the (ACC) string is processed left to right, and the instruction is 16 bits long, (length field of descriptor in ACC) must be not less than L,else see 7.4.2.11.2.

Most store-to-store operations can be interrupted in mid-flight, and resumed after the interruption.  As well as DR and, where relevant, ACC being updated in the course of the operation, additional facilities are provided (see section 5.3) enabling the instructions to be thus resumed.

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

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

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

The following table summarises whether mask and filler/literal bytes in B or the second half of along instruction are used by each store-to-store instruction. For precise details see section 8.3.3.

| Instruction Mnemonic | 16-bit Instruction Format | | 32-bit Instruction Format | |
|---|---|---|---|---|
| | B(16-23) | B(24-31) | Instr.(16-23) | Instr.(24-31) |
| SWEQ SWNE | used | used | used | used |
| CPS | not used | not used | used | used if Li(ACC)lgth |
| MV | not used | not used | used | used if Li(ACC)lgth |
| CHOV | not used | not used | ignored (reserved) | ignored (reserved) |
| MVL | used | used | used | used |
| TCH TTR PK | not used | not used | ignored (reserved) | ignored (reserved) |
| SUPK | not used | used if CC = 0 | ignored (reserved) | used if CC = 0 |
| INS | used if CC ≠ 0 | used if CC = 0 | used if CC ≠ 0 | used if CC = 0 |
| ANDS ORS NEQS | not used | not used | ignored (reserved) | used |

## 8.3.2 List of Instructions

Scan while equal                    Table translate
Scan while unequal                  Pack
Compare strings                     Suppress and unpack
Move                                Conditional insert
Check overlap                       And strings
Move literal                        Or strings
Table check                         Not equivalent strings

## 8.3.3 Instruction descriptions

### 8.3.3.1 Scan While Equal (SWEQ)                    Function Code : A0

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

Each byte in the (DR) string, working from left to right, is compared with the reference byte. Only these bits in each byte, including the reference byte, which correspond to 0's in the mask byte, are compared. The operation stops when the (DR) string is exhausted, or when inequality is

**ICL**

**Product
specification**

**Company
restricted**

**PSD** ___2.5.1___

Issue ___6/0___

Sheet ___129___

detected, whichever occurs first; in the latter case (DR) will finish
pointing to the first byte in the (DR) string which is not equal to the
reference byte.

CC is set to indicate whether or not inequality was found, and if so
whether (the unmasked portion of) the (DR) byte was less than or greater
than (the unmasked portion of) the reference byte.

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

| | | | |
|---|---|---|---|
| CC | : | 0 | Inequality not found |
| | | 1 | Not used |
| | | 2 | (DR) string byte greater than reference byte(unmasked portions) |
| | | 3 | (DR) string byte less than reference byte(unmasked portions) |

Program errors    : Any failures of standard checks 1 and 2.


### 8.3.3.2 Scan While Unequal (SWE)                    Function Code  :  A2

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

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

Program errors    : Any failures of standard checks 1 and 2.


### 8.3.3.3 Compare Strings (CPS)                    Function Code  :  A4

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

Where inequality is found, DR will finish pointing to the first (DR)
string byte which compared unequal, and ACC likewise unless the (ACC)
string had already expired. If L=0, and none of the other checks fail, a
null operation is performed, leaving ACC and DR unaltered, but leaving
CC=0.

| CC | : | 0 | Inequality not found |
|---|---|---|---|
| | | 1 | Not used |
| | | 2 | (DR) string byte greater than (ACC) string byte (unmasked portions) |
| | | 3 | (DR) string byte less than (ACC) string byte (unmasked portions) |

Program errors    :  Any failures of standard checks 1 - 4

### 8.3.3.4 Move (MV)                                Function Code  :  B2

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

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

CC                :  Unaltered

Program errors    :  Any failures of standard checks 1 - 4.

### 8.3.3.5 Check Overlap (CHOV)                     Function Code  :  B4

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

| CC | : | 0 | No overlap |
|---|---|---|---|
| | | 1 | Overlap - (ACC) address greater than (DR) address |
| | | 2 | Overlap -(ACC) address less than (DR) address |
| | | 3 | Not used |

Program errors    :  Any failures of standard checks 1 - 4.

**ICL**

**Product
specification**

**Company
restricted**

**PSD**   2.5.1

Issue        6/0

Sheet        131

### 8.3.3.6 Move Literal (MVL)                              Function Code  :  80

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

CC                 :   Unaltered

Program errors     :   Any failures of standard checks 1 and 2.


### 8.3.3.7 Table Check (TCH)                              Function Code  :  80

Description        :   The descriptor in ACC points to a table check of
bits.  This descriptor must be of type 0, with size code 1 bit, and a
valid bound; USC and BCI must be zero.  Successive bytes in the (DR)
string are checked against this table in the following way:  the more
significant 5 bits of the byte are used as a modifier of the address in
the ACC descriptor to reference a byte in the table, and the least
significant 3 bits to refer to one of the bits (numbered 0-7) in that
byte.  Thus, if the value of the byte is 10010101 (=8 x 18 + 5) and the
byte address in ACC is A, the check bit is bit 5 of byte (A + 18).  If the
value of the byte is not less than the bound field of the descriptor in
ACC, an interrupt occurs.  Processing of the (DR) string, from left to
right, continues until the string is exhausted, or a byte whose check bit
is 1 is found; in the latter case DR is left pointing to that byte.  CC is
used to indicate the reason for termination .  ACC is unaltered.  The (DR)
string is unaltered.  16 or 32 bit instruction forms are permitted. In the
32-bit form, the mask and literal bytes are ignored (reserved).  If L-0,
and none of the other checks fail, a null operation is performed, leaving
DR unaltered, but leaving CC=0.

Certain hardware implementations may access some or all of the check bit
table before any DR string reference is made against it.  Such
implementations may generate virtual store interrupt conditions for
referenced or unreferenced parts of the table.

If Program Mask bit 5 (bound check) is set, the table is assumed to be 32
bytes long.

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

Program errors     :   Any failures of standard checks 1 to 3. (DR) string
                       byte not less than bound field of descriptor in
                       ACC. (see 7.4.2.5.8.)

### 8.3.3.8 Table Translate (TTR)                                    Function Code :  A6

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

Certain hardware implementations may access some or all of the translation
table before any DR string reference is made against it.   Such
implementations may generate virtual store interrupt conditions for
referenced or unreferenced parts of the table.

If program mask Bit 5 (bound check) is set, the table is assumed to be 256
bytes long.

CC                          : Unaltered

Program errors              : Any failures of standard checks 1 to 3. (DR) string
                              byte not less than bound field of descriptor in
                              ACC. (see 7.4.2.5.8.)

### 8.3.3.9 Pack (PK)                                            Function Code : 90

Description        : For each (DR) string byte, working from left to
right, the contents of ACC are shifted decimally left by 1 place and the
least significant 4 bits of the byte inserted in the space thus created
next to the sign digit of ACC.  ACS may be 32, 64 or 128 bits.  OV is set
if any non-zero bits are shifted off the top of ACC, and in this case
decimal overflow interrupt will occur unless masked.  If no non-zero bits
are shifted off OV is cleared.  The sign digit is unaltered.  If L=0, and
none of the other checks fail, a null operation is performed, leaving ACC
and DR unaltered; OV is cleared.

Either 16 or 32 bit forms may be specified but for 32 bits the mask and
literal bytes are ignored (reserved).

The results of the instruction is undefined if L is not less than 128

CC                          : Unaltered

Program  errors             : Any failures of standard checks 1 and 2. Decimal
                              overflow (unless masked)(see7.4.2.3)

8.3.3.10 Suppress & Unpack (SUPK)          Function Code : 94

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

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

|                        | CC = 0                         | CC ≠ 0                     |
|------------------------|--------------------------------|---------------------------|
| Unpacked digit = 0     | Insert literal<br>CC unaltered | Insert digit<br>CC unaltered |
| Unpacked digit ≠ 0     | Insert digit<br>Set CC = 2<br><br>*Stack descriptor | Insert digit<br>Set CC = 2 |

*If CC = 0 and the digit being unpacked in non-zero, a type 1 descriptor is generated and stacked, which has 1 in its length field and whose address field contains 1 less than the current address in DR - i.e. it points to the byte immediately to the left of the position in the (DR) string where the digit is inserted. This causes SF to be incremented by two words. (A similar effect is achieved by the Start significance instruction - see section 8.1.5.11)

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

OV is cleared. ACS may be 32, 64 or 128 bits. With the 32-bit instruction format the mask byte is ignored (reserved). The result of the instruction is undefined if L greater than or equal to 128

Notes a) After a number has been completely unpacked ACC will contain no non-zero digits, and only by testing CC can it be ascertained whether the number was positive, negative or zero.
      b) If the descriptor which is stacked when the first non-zero digit is unpacked is not used for sign insertion (.e.g.) it must be removed from the stack anyway.

**ICL**

**Product
specification**

**Company
restricted**

**PSD**    2.5.1

Issue    6/0

Sheet    134

CC      :   0   CC was 0, and all digits unpacked were 0's
                   1   CC was 1, and all digits unpacked were 0's
                   2   CC was 2 or 3, or some non-zero digits
                        unpacked. Sign positive.
                   3   CC was 2 or 3, or some non-zero digits
                        unpacked. Sign negative.

Program errors    :   Any failures of standard checks 1 and 2.

### 8.3.3.11 Conditional Insert (INS)            Function Code : 92

Description      :   This instruction is similar to Move literal
(8.3.3.6) in that the source byte overwrites successive bytes of the (DR)
string. However the source byte is defined differently, as follows:
                       CC = 0    Literal (bits 24-31 of B, if 16-bit format)
                       CC ≠ 0    Mask (bits 16-23 of B, if 16-bit format)
Each (DR) string byte is completely overwritten.

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

This instruction is intended for use with Suppress and unpack (8.3.3.10).

CC                 :   Unaltered

Program errors    :   Any failures of standard checks 1 and 2.

### 8.3.3.12 And Strings  (ANDS)               Function Code : 82
###         Or Strings   (ORS)                  Function Code : 84
###         Not Equivalent Strings (NEQS)      Function Code : 86

Description       :   Each byte of the (DR) string is replaced by the
result of performing the appropriate logical operation betwen itself and
the corresponding byte of the source string. In the 16-bit format the
source string is defined as the (ACC) string. In the 32 bit format ACC is
not used and the source string is L copies of the literal byte. The mask
byte is ignored (reserved).

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

CC                 :   Unaltered

Program errors    :   Any failures of standard checks 1 - 4.

## 8.4 Bridgeware Instruction

Two instructions are provided for converting 6 bit data to or from 8 bit data.

### 8.4.1 Compress ACC (COMA)      Function Code : 98
    Expand ACC   (EXPA)      Function Code : 88

These instructions use the primary format described in Section 6.1.

Operand length      : Not applicable, literal must be specified.

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

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

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

CC      : Unaltered

Program errors      : ACS = 128 bits (see 7.4.2.13.8.)

**ICL**

**Product
specification**

**Company
restricted**

**PSD**   2.5.1

Issue   6/0

Sheet   136

## 9   Privileged Operations

### 9.1   General

The only privileged instruction is Activate (Function Code 3E) and PRIV must be set for its execution.

Other privileged operations will be performed by using the instructions Load (8.1.3.6) or Store (8.1.3.7.) with appropriate image store operand addresses and, in the latter case operands in ACC (see section 3.3 et seq.). These operations are privileged in the sense that they require appropriate settings of PRIV, ISR and DGW (see section 3.3.2) to access the image store locations concerned.

Included in the functions thus performed with image store operands are the following:-

> Fire I/O                         )
> Record I/O interrupt state      )    see [2]
>
> Initial load
> Set or read SMAC registers
> Set or read SAC registers
> Communicate with linked processors
> Clear slave stores
> Load PSTB
> Set real-time clock
> Clear real-time clock overflow
>
> - and other implementation-defined actions.

### 9.2   Instruction Descriptions

> Activate (ACT)                            Function Code : 3E
>
> Operand length          :   128 bits
>
> Description                :   The first two words of the operand are loaded to LSTB (see section 3.2.3). Bits 14-30 of the first word, and 0-3 and 29-31 of the second word are ignored (spare). The third word is ignored (spare). Bits 0-13 of the fourth word contain the new value of SSN; bits 14-31 are ignored (spare). After loading LSTB the action of the instruction is to generate the base address of segment (SSN+1) (effectively by concatenating the bit pattern 10 .....0 with bits 0-12 of the fourth word of the operand; if the segment number is in the range 0-8191, the new LSTB is used to translate it) and unload the contents of words 0-15 of that segment (the 'process state') to the appropriate CPU registers, the reverse of the stack-switching interrupt process portrayed in Figure 4 of section 5.3. In emulating machines, E and EM bits of the new PSR and SSR will be examined after unloading words 0-7, and subsequent action will depend on their values. Virtual addressing mode is assumed throughout. Methods of changing addressing mode are implementation-defined.

If there is disagreement between the values of SSN specified by the operand and in words 0 and 4 of the process state the result is undefined. A system error interrupt may occur if an attempt is made to load an odd segment number to SSN.

The new process defined by the undumped process state is entered at the instruction specified by PC, qualified if necessary by the setting of II (see section 5.3.9.). In emulating machines, if E=1 in new PSR, and EM in new SSR has a locally valid value, alien code is executed.

If bit 31 of the first word of the operand is a 1, and if the EP interrupt mask bit is not set in the new SSR, an Event Pending interrupt occurs on resumption of the process. This may occur before the registers have been completely undumped and, if II is set, it will occur before attempting completion of the uncompleted instruction. The EP bit in SSR is ignored and is not cleared.

PSTB is not altered. Execution of Activate may cause the new value of IC (section 3.2.6) to be decremented.

This instruction requires PRIV = 1 to be executed.

CC                   :   As specified in new PSR

Program errors       :   Operand addressing errors
                             Privilege (see 7.4.2.9.6.)
                             (Emulating machines; PEI 14) New E=1 and
                             EM=0 or invalid value

(Note: Masking of program error interrupts is controlled by the program and interrupt mask bits in PSR and SSR at the beginning of the instruction. The occurrence of an unmasked program error during the intitial state of Activiate will prevent the loading of LSTB, SSN and the other processor registers (i.e. effectively it is the old process, not the new one, being interrupted.) However in the case of an error in switching to emulation, implementation defined action will occur.)

9.3   **Bootstrap**

When the initial load button is pressed on the operator's console a transfer into store is initiated from the device nominated on the load device address switched on that console. Subsequent actions are as follows:

1 The transfer is into real addresses starting at 0 (all store addresses quoted below are real word addresses).

2 At the end of the transfer, the following sequence occurs, instead of the interrupt procedure described in section 5 (no IST is required). The CPU may load its microprogram store, in an implementation-defined manner, from real words 64 onwards. The image store locations in Block 0, lines 0-15 (section 3.5.1.), are loaded with the contents of real words 32-47, and LSTB and PSTB are loaded from implementation-defined locations in words 48-63 the initial CPU status is thus completely defined). Instruction execution commences at the instruction indicated by PC.

Whether the address of PC and addresses subsequently generated are treated as virtual or real depends on the setting of the RAM bit in SSR (see section 3.2.1). If virtual, all segment and page tables must have been set up as part of the initial transfer; and provision must be made for addressing the controller communication area in real words 0-7. (see below).

3 For the purpose of the initial transfer the communication area for the peripheral controller involved is in real words 0-7; the bootstrap response area (see [2]) is in words 6 and 7. There are no stream SAWs or response areas involved.

The CPU can establish the SAC/trunk/stream number of the input device as follows: SAC number from CPU interrupt flags (in image store Block 0, implementation defined); trunk number from SAC interrupt flags (see section 3.5.2); and stream number from bits 24-31 of real word 6 (bootstrap response area).

Differences in operation due to configuration details are dealt with in [6].

# APPENDIX 1
## Alphabetical list of instructions

### Miscellaneous

| Name | Section | Mnemonic | F Code |
|------|---------|----------|--------|
| Add to B | 8.1.4.4 | ADB | 20 |
| Adjust SF | 8.1.2.4 | ASF | 6E |
| Call | 8.1.2.8 | CALL | 1E |
| Compare & Increment B | 8.1.4.6 | CPIB | 2E |
| Compare B | 8.1.4.5 | CPB | 26 |
| Copy DR | 8.1.3.10 | CYD | 12 |
| Copy PSR | 8.1.3.2 | CPSR | 34 |
| Decrement B & jump if non-zero | 8.1.2.12 | DEBJ | 24 |
| Dope vector multiply | 8.1.4.7 | VMY | 2C |
| Escape exit | 8.1.2.15 | ESEX | 3A |
| Exit | 8.1.2.9 | EXIT | 38 |
| Idle | 8.1.2.17 | IDLE | 4E |
| Increment & Test | 8.1.2.7 | INCT | 56 |
| Increment address | 8.1.5.10 | INCA | 14 |
| Jump | 8.1.2.11 | J | 1A |
| Jump & link | 8.1.2.10 | JLK | 1C |
| *Jump on arithmetic - condition false | 8.1.2.14 | JAF | 06 |
| *Jump on arithmetic - condition true | 8.1.2.14 | JAT | 04 |
| *Jump on CC | 8.1.2.13 | JCC | 02 |
| Load | 8.1.3.6 | L | 60 |
| Load address | 8.1.5.5 | LDA | 72 |
| Load B | 8.1.4.1 | LB | 7A |
| Load bound | 8.1.5.7 | LDB | 76 |
| Load DR | 8.1.5.1 | LD | 78 |
| Load LNB | 8.1.2.1 | LLN | 7C |
| Load relative | 8.1.5.4 | LDRL | 70 |
| Load type & bound | 8.1.5.6 | LDTB | 74 |
| Load upper half | 8.1.3.8 | LUH | 6A |
| Load XNB | 8.1.2.2 | LXN | 7E |
| Load CTB | 8.1.2.18 | LCT | 30 |
| Modify DR | 8.1.5.8 | MODD | 16 |
| Modify PSR | 8.1.3.1 | MPSR | 32 |
| Multiply B | 8.1.4.4 | MYB | 2A |
| Out | 8.1.2.16 | OUT | 3C |
| Pre-call | 8.1.2.21 | PRCL | 18 |
| Raise LNB | 8.1.2.3 | RALN | 6C |
| Read real-time clock | 8.1.3.11 | RRTC | 68 |
| Set ACS 128 & load | 8.1.3.3 | LSQ | 66 |
| Set ACS 32 & load | 8.1.3.3 | LSS | 62 |
| Set ACS 64 & load | 8.1.3.3 | LSD | 64 |
| Stack & load | 8.1.3.5 | SL | 40 |
| Stack & load B | 8.1.4.2 | SLB | 52 |
| Stack & load DR | 8.1.5.2 | SLD | 50 |
| Stack, set ACS 128 & Load | 8.1.3.4 | SLSQ | 46 |
| Stack, set ACS 32 & load | 8.1.3.4 | SLSS | 42 |
| Stack, set ACS 64 & load | 8.1.3.4 | SLSD | 44 |
| Start significance | 8.1.5.11 | SIG | 28 |
| Store | 8.1.3.7 | ST | 48 |

**ICL**

**Product specification**

**Company restricted**

**PSD**   2.5.1

Issue   6/0

Sheet   140

| Name | Section | Mnemonic | F Code |
|------|---------|----------|--------|
| Store B | 8.1.4.3 | STB | 5A |
| Store DR | 8.1.5.3 | STD | 58 |
| Store LNB | 8.1.2.5 | STLN | 5C |
| Store SF | 8.1.2.6 | STSF | 5E |
| Store upper half | 8.1.3.9 | STUH | 4A |
| Subtract from B | 8.1.4.4 | SBB | 22 |
| Test & decrement | 8.1.2.7 | TDEC | 54 |
| Validate address | 8.1.5.9 | VAL | 10 |
| Store CTB | 8.1.2.20 | STCT | 36 |
| Store XNB | 8.1.2.19 | STXN | 4C |

*Tertiary format

Store-to-store-

| Name | Section | Mnem | F Code |
|------|---------|------|--------|
| And strings | 8.3.3.12 | ANDS | 82 |
| Check overlap | 8.3.3.5 | CHOV | B4 |
| Compare strings | 8.3.3.3 | CPS | A4 |
| Conditional insert | 8.3.3.11 | INS | 92 |
| Move | 8.3.3.4 | MV | B2 |
| Move literal | 8.3.3.6 | MVL | B0 |
| Not equivalent strings | 8.3.3.12 | NEQS | 86 |
| Or strings | 8.3.3.12 | ORS | 84 |
| Pack | 8.3.3.9 | PK | 90 |
| Scan while equal | 8.3.3.1 | SWEQ | A0 |
| Scan while unequal | 8.3.3.2 | SWNE | A2 |
| Suppress & Unpack | 8.3.3.10 | SUPK | 94 |
| Table check | 8.3.3.7 | TCH | 80 |
| Table translate | 8.3.3.8 | TTR | A6 |

**Bridging**

| Name | Section | Mnem | F Code |
|------|---------|------|--------|
| Compress ACC | 8.4.1 | COMA | 98 |
| Expand ACC | 8.4.1 | EXPA | 88 |

**Privileged**

| Name | Section | Mnem | F Code |
|------|---------|------|--------|
| Activate | 9.2 | ACT | 3E |

## List of instructions in mnemonic alphabetical order

| Mnemonic | Name | Section |
|----------|------|---------|
| ACT | Activate | |
| ADB | Add to B | 8.1.4.4 |
| AND | And | 8.2.5.6 |
| ANDS | And strings | 8.3.3.12 |
| ASF | Adjust SF | 8.1.2.4 |
| CALL | Call | 8.1.2.8 |
| CBIN | Convert to binary | 8.2.6.10 |
| CDEC | Convert to decimal | 8.2.4.10 |
| CHOV | Check overlap | 8.3.3.5 |
| COMA | Compress ACC | 8.4.1 |
| CPB | Compare B | 8.1.4.5 |
| CPIB | Compare & increment B | 8.1.4.6 |
| CPS | Compare strings | 8.3.3.3 |
| CPSR | Copy PSR | 8.1.3.2 |
| CYD | Copy DR | 8.1.3.0 |
| DAD | Decimal add | 8.2.6.1 |
| DCP | Decimal compare | 8.2.6.3 |
| DDV | Decimal divide | 8.2.6.6 |
| DEBJ | Decrement B & jump if non-zero | 8.1.2.12 |
| DMDV | Decimal remainder divide | 8.2.6.8 |
| DMY | Decimal multiply | 8.2.6.5 |
| DMYD | Decimal multiply double | 8.2.6.9 |
| DSB | Decimal subtract | 8.2.6.1 |
| DSH | Decimal shift | 8.2.6.4 |
| DRDV | Decimal reverse divide | 8.2.6.7 |
| DRSB | Decimal reverse subtract | 8.2.6.2 |
| ESEX | Escape exit | 8.1.2.15 |
| EXIT | Exit | 8.1.2.9 |
| EXPA | Expand ACC | 8.4.1 |
| FIX | Fix | 8.2.3.10 |
| FLT | Float | 8.2.4.4 |
| IAD | Integer add | 8.2.4.1 |
| ICP | Integer compare | 8.2.4.3 |
| IDLE | Idle | 8.1.2.17 |
| IDV | Integer divide | 8.2.4.6 |
| IMDV | Integer remainder divide | 8.2.4.8 |
| IMY | Integer multiply | 8.2.4.5 |
| IMYD | Integer multiply double | 8.2.4.9 |
| INCA | Increment address | 8.1.5.10 |
| INCT | Increment & test | 8.1.2.7 |
| INS | Conditional insert | 8.3.3.11 |
| IRDV | Integer reverse divide | 8.2.4.7 |
| IRSB | Integer reverse subtract | 8.2.4.2 |
| ISB | Integer subtract | 8.2.4.1 |
| ISH | Arithmetic shift | 8.2.4.4 |
| J | Jump | 8.1.2.11 |
| JAF | Jump on arith. condition false | 8.1.3.14 |
| JAT | Jump on arith. condition true | 8.1.3.14 |
| JCC | Jump on CC | 8.1.2.13 |
| JLK | Jump & link | 8.1.2.10 |

| | | |
|---|---|---|
| L | Load | 8.1.3.6 |
| LB | Load B | 8.1.4.1 |
| LCT | Load CTB | 8.1.2.18 |
| LD | Load DR | 8.1.5.1 |
| LDA | Load address | 8.1.5.5 |
| LDB | Load bound | 8.1.5.7 |
| LDRL | Load relative | 8.1.5.4 |
| LDTB | Load type & bound | 8.1.5.6 |
| LLN | Load LNB | 8.1.2.1 |
| LSD | Set ACS 64 & load | 8.1.3.3 |
| LSQ | Set ACS 128 & load | 8.1.3.3 |
| LSS | Set ACS 32 & load | 8.1.3.3 |
| LUH | Load upper half | 8.1.3.8 |
| LXN | Load XNB | 8.1.2.2 |
| MODD | Modify DR | 8.1.5.8 |
| MPSR | Modify PSR | 8.1.3.1 |
| MV | Move | 8.3.3.4 |
| MVL | Move literal | 8.3.3.6 |
| MYB | Multiply B | 8.1.4.4 |
| NEQ | Not equivalent | 8.2.5.6 |
| NEQS | Not equivalent strings | 8.3.3.12 |
| OR | Or | 8.2.5.6 |
| ORS | Or strings | 8.3.3.12 |
| OUT | Out | 8.1.2.16 |
| PK | Pack | 8.3.3.9 |
| PRCL | Pre-call | 8.1.2.21 |
| RAD | Floating add | 8.2.3.1 |
| RALN | Raise LNB | 8.1.2.3 |
| RCP | Floating compare | 8.2.3.3 |
| RDV | Floating divide | 8.2.3.6 |
| RDVD | Floating divide double | 8.2.3.8 |
| RMY | Floating mulitply | 8.2.3.5 |
| RMYD | Floating multiply double | 8.2.3.9 |
| ROT | Rotate | 8.2.5.7 |
| RRDV | Floating reverse divide | 8.2.3.7 |
| RRSB | Floating reverse subtract | 8.2.3.2 |
| RRTC | Read real-time clock | 8.1.3.11 |
| RSB | Floating subtract | 8.2.3.1 |
| RSC | Scale | 8.2.3.4 |
| SBB | Subtract B | 8.1.4.4 |
| SHS | Shift 32 bits | 8.2.5.8 |
| SHZ | Shift while zero | 8.2.5.9 |
| SIG | Start significance | 8.1.5.11 |
| SL | Stack & Load | 8.1.3.5 |
| SLB | Stack & load B | 8.1.4.2 |
| SLD | Stack & load DR | 8.1.5.1 |
| SLSD | Stack,set ACS 64 & load | 8.1.3.4 |
| SLSQ | Stack,set ACS 128 & load | 8.1.3.4 |
| SLSS | Stack,set ACS 32 & load | 8.1.3.4 |
| ST | Store | 8.1.3.7 |
| STB | Store B | 8.1.4.3 |
| STCT | Store CTB | 8.1.2.20 |
| STD | Store DR | 8.1.5.3 |
| STLN | Store LNB | 8.1.2.5 |
| STSF | Store SF | 8.1.2.6 |

**ICL**

**Product specification**

**Company restricted**

PSD   2.5.1

Issue   6/0

Sheet   145

| | | |
|---|---|---|
| STUH | Store upper half | 8.1.3.9 |
| STXN | Store XNB | 8.1.2.19 |
| SUPK | Supress and unpack | 8.3.3.10 |
| SWEQ | Scan while equal | 8.3.3.1 |
| SWNE | Scan while unequal | 8.3.3.2 |
| TCH | Table check | 8.3.3.7 |
| TDEC | Test & decrement | 8.1.2.7 |
| TTR | Table translate | 8.3.3.8 |
| UAD | Logical add | 8.2.5.1 |
| UCP | Logical compare | 8.2.5.4 |
| URSB | Logical reverse subtract | 8.2.5.3 |
| USB | Logical subtract | 8.2.5.2 |
| USH | Logical shift | 8.2.5.5 |
| VAL | Validate address | 8.1.5.9 |
| VMY | Dope vector multiply | 8.1.4.7 |

## Table showing allotted function codes
### First hexadecimal digit

Second hexadecimal digit (even)

|   | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |   |
|---|---|---|---|---|---|---|---|---|---|
| 0 | ** | VAL | ADB | LCT | SL | SLD | L | LDRL | 0 |
| 2 | JCC | CYD | SBB | MPSR | SLSS | SLB | LSS | LDA | 2 |
| 4 | JAT | INCA | DEBJ | CPSR | SLSD | TDEC | LSD | LDTB | 4 |
| 6 | JAF | MODD | CPB | STCT | SLSQ | INCT | LSQ | LDB | 6 |
| 8 | * | PRCL | SIG | EXIT | ST | STD | RRTC | LD | 8 |
| A | * | J | MYB | ESEX | STUH | STB | LUH | LB | A |
| C | * | JLK | VMY | OUT | STXN | STLN | RALN | LLN | C |
| E | * | CALL | CPIB | ACT | IDLE | STSF | ASF | LXN | E |

First digit

Second digit

|   | 8 | 9 | A | B | C | D | E | F |   |
|---|---|---|---|---|---|---|---|---|---|
| 0 | TCH | PK | SWEQ | MVL | UAD | DAD | IAD | RAD | 0 |
| 2 | ANDS | INS | SWNE | MV | USB | DSB | ISB | RSB | 2 |
| 4 | ORS | SUPK | CPS | CHOV | URSB | DRSB | IRSB | RRSB | 4 |
| 6 | NEQS | * | TTR | * | UCP | DCP | ICP | RCP | 6 |
| 8 | EXPA | COMA | FLT | FIX | USH | DSH | ISH | RSC | 8 |
| A | AND | DDV | IDV | RDV | ROT | DMY | IMY | RMY | A |
| C | OR | DRDV | IRDV | RRDV | SHS | DMYD | IMYD | RMYD | C |
| E | NEQ | DMDV | IMDV | RDVD | SHZ | CBIN | CDEC | ** | E |

\*     -   Unassigned function code

\*\*   -   illegal function code
        (00 Tertiary, FE Primary format)

02-0E   Tertiary format
10-7E   Primary format (miscellaneous)

80-86)
90-96)   Secondary format
A0-A6)
B0-B6)

88-8E,C0-CE   Primary format (logical)
98-9E,D0-DE       "         "     (decimal)
A8-AE,E0-EE       "         "     (fixed-point)
B8-BE,F0-FC       "         "     (floating point)

## Appendix 2

Summary of additional facilities at AML 1.

| Item | Description | Sheet References |
|------|-------------|------------------|
| 1. | Alternative format for RRTC | 8.1.3.11 |
| 2. | Alternative form of VALIDATE instruction | 8.1.5.9 |
| 3. | System Call Count | 5.3.15 |
| 4. | Parameter space check for System Calls | 5.3.15 |
| 5. | Addition of image store bit to control clearing of slaves in INCT & TDEC | 4.3.5 & 8.1.2.7 |
| 6. | (B+N) operand form | 6.1.1 |
| 7. | Vector Descriptor, Type 0, size code 4 (=16 bits) | 6.2 |
| 8. | Vector descriptor, for signed items | 6.2 |
| 9. | Null descriptors | 6.2 |

**ICL**

**Product specification**

**Company restricted**

**PSD** 2.5.1

Issue 6/0

Sheet 148

A3          Appendix 3 Microcode Routines

This appendix lists the numbers that have been assigned for use with microcode descriptors. This appendix does not define the implementation of the available functions, but a brief description is given where applicable.

A3.1          DR8-23=0000 Maths Library Functions.

| DR24-31 | Function Mnemonic | Notes |
|---------|-------------------|-------|
| 00 | NULL | |
| 01 | SQRT | $\sqrt{x}$ |
| 02 | LOG | $\log_e x$ |
| 03 | EXP | $e^x$ |
| 04 | SIN | ) |
| 05 | COS | ) radian only |
| 06 | TAN | ) |
| 07 | ATAN | ) |
| 08-FF | Reserved | |

The above microcode descriptors must be used with a JLK instruction. The input parameter is taken from ACC and the result is returned to ACC.

A3.2          DR-23=0001 to 000F.

Not allocated.

A3.3          DR-23 = 0010 Error Status Monitor.

Values of DR24-31 of 00 to 0F are used on S1/S2 to initiate internal OCP tests. On completion of a test, CC will be set to zero. If any test fails, the Error Status Monitor will be turned off.