cii COMPAGNIE INTERNATIONALE POUR L'INFORMATIQUE

MITRA 125

REFERENCE MANUAL

VOLUME 1

DECEMBER 1975

N° 5419/U1/EN

SUMMARY

TABLE OF FIGURES

I - INTRODUCTION

The MITRA 125 computer supports two main program classes which are differentiated by their execution modes, i.e. :

- privileged mode
- user mode

Privileged mode is subdivided into three submodes, i.e. :

- the "privileged program" mode
- the "privileged subroutine" mode
- the "supervisor" mode

In privileged mode, the entire set of computer instructions can be executed ; programs and data are not protected in this mode ; its utilization must be restricted.

This operating mode is described in Volume II of the Reference Manual.

User mode is subdivided into two submodes, i.e. :

- "program" mode
- "subroutine" mode

In user mode, only the instructions which cannot cross the various protection barriers are executed ; this is the normal operating mode for applications programs.

II - GENERAL DESCRIPTION OF MITRA 125 IN USER MODE

II-1. GENERAL DEFINITIONS

II-1.1. Virtual Machine

The MITRA 125 Central Processing Unit (CPU) executes only a single instruction at a time. However, because of the automatic register save and restore mechanisms ("context switching"), the CPU can be considered as a set of n "virtual machines" of which only one is active at any given time.

II-1.2. Task

A task is the association of an executable program, an execution priority, and a virtual machine that executes it.

"Task creation" is defined as "the action which consists of performing this association".

A monitor module allows creating a task upon request from a user program which sends to it the name of the program in library and the execution priority.

II-1.3. Task Context

Task context is a main memory table in which various software-visible physical registers are saved at the time of a task switching. Contents of this table are used to reinitialize the registers when the interrupted task resumes execution.

II-1.4. Context Switching (see figure 2.1)

"Context switching" is the operation which consists of saving the software-visible physical registers in the current task context and to reinitialize these registers from the contents of the interrupting task context.

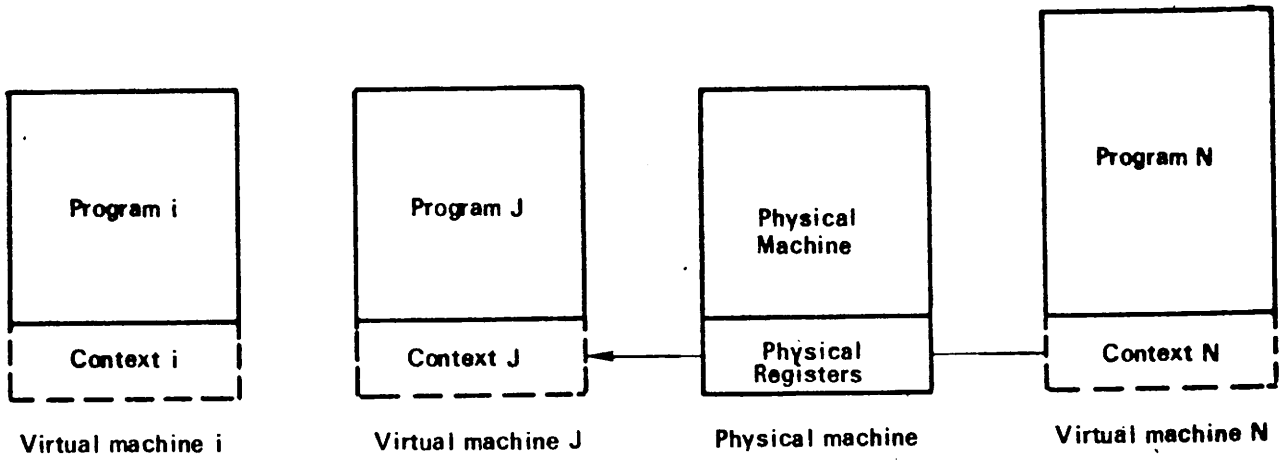


Fig. 2.1 - CONTEXT SWITCHING

The physical machine which was associated with virtual machine J is placed at the disposal of virtual machine N.

This "context switching" operation is automatically performed on the MITRA 125 :

- via a microprogrammed dispatcher when a program interrupt request with a level higher than the current level occurs : case for "immediate" tasks.
- via the MMT 2 Monitor Scheduler : case for "deferred" tasks.

II-1.5. Immediate Task

An immediate task is a task which is directly activated when a signal occurs (programmed or external), arming the program interrupt flip-flop to which the task was connected by the monitor.

MITRA 125 can receive 128 interrupts divided into 32 priority levels. Four interrupts with the same priority can be connected to each interrupt level.

These interrupts can be armed, disarmed, enabled, disabled, triggered, and globally masked by a program executed in privileged mode. They are not accessible in User mode.

II-1.6. Deferred Task

A deferred task is a task which is not directly associated with a program interrupt. Deferred tasks are automatically managed by the MMT 2.

The concepts of immediate and deferred tasks cannot be distinguished at the assembly language level.

II-1.7. Segmentation

A segment is a continuous program and/or data area.

In User mode, only a limited number of segments are directly accessible. Other segments are accessible only through the monitor modules.

Segment Descriptor

When a segment is in main memory, it is defined by a pair of words called the "segment descriptor".

The first word called the Segment Base gives the memory start address (expressed in octo-words) of the beginning of the segment's addressable area.

The second word called the Segment Length gives the length (expressed in bytes) of the segment's addressable area.

Any program attempt to exceed the boundaries of the addressed segment produces a program trap (call of a standard monitor module).

Segment Accessibility (see figure 2-2.)

The descriptor of a segment directly accessible in User mode must be loaded in one of the base and length registers referenceable in User mode by a program executed in Privileged mode (generally, the monitor).

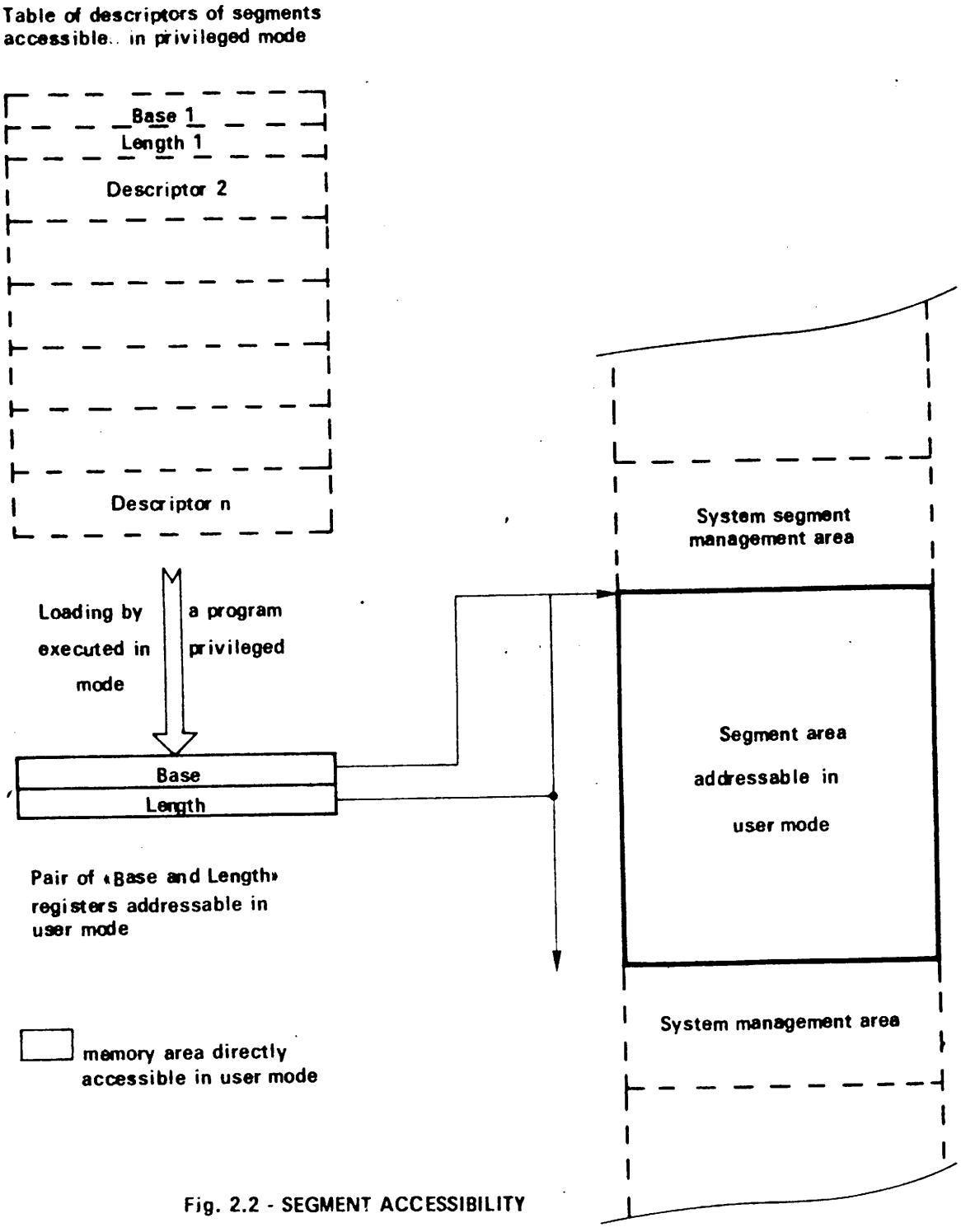


Fig. 2.2 - SEGMENT ACCESSIBILITY

II-1.8. Task Group (see figure 2-3.)

A "Task Group" is a set of tasks sharing a number of common segments which constitute the Group's operating domain.

A Group Segment Descriptor Table (GST) is associated with each task group and pointed by a word of the context of each task.

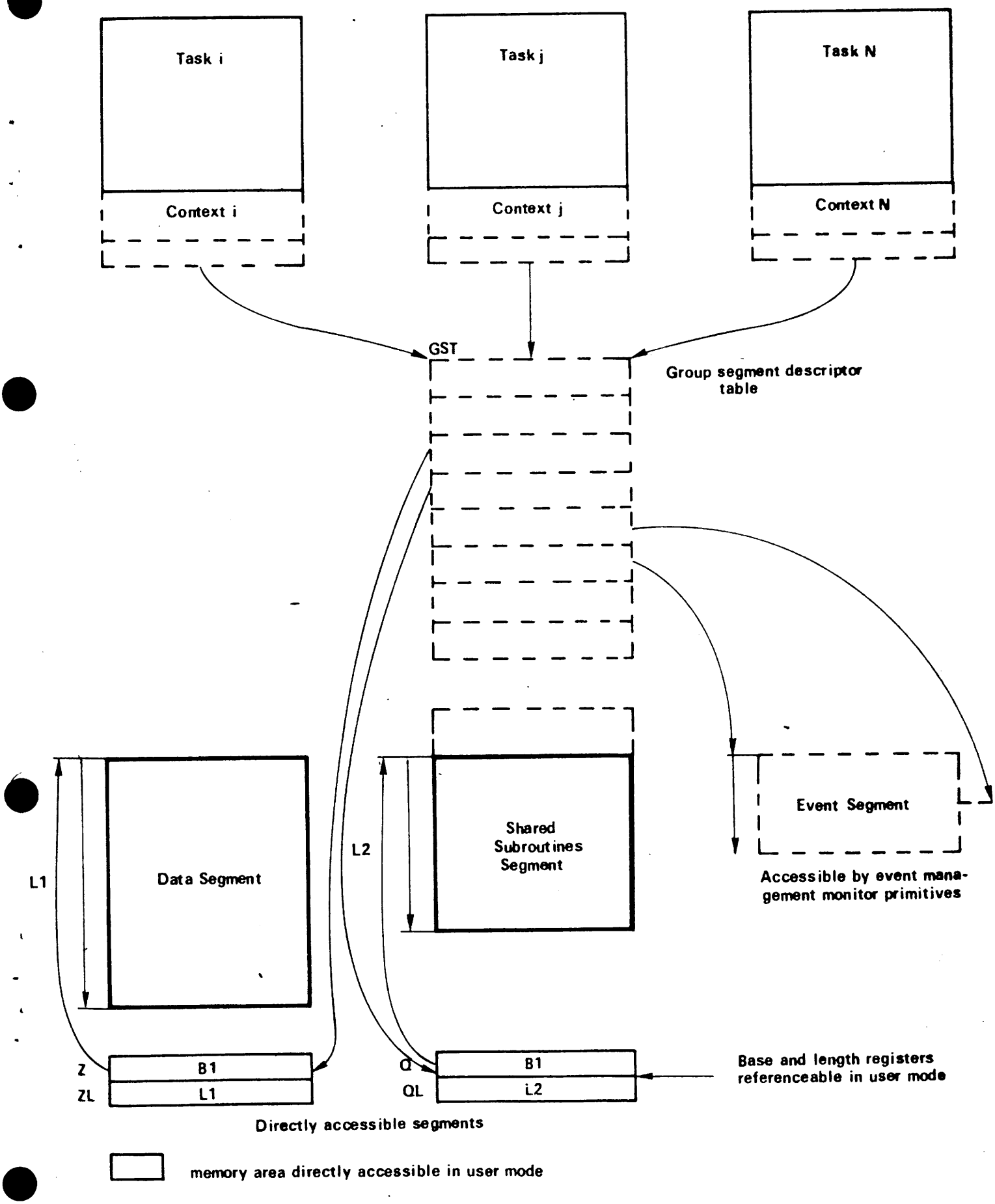


Fig. 2.3 - TASK GROUP TOPOLOGY

Contents definition of these segments depends on the monitor : the contents are defined in the Monitor's Utilization Manual.

Examples of Segments :

- Data segment shareable by the group's tasks
- Subroutine segment shareable by the group's tasks
- Event segment
- Semaphore segment

...

II-1.9. User Visibility

The program has available in User mode :

- the set of instructions allowed in User mode to reference directly accessible segments.
- the set of primitives allowed in User mode, supplied by the monitor. These primitives can be called in User mode by the instruction "Call Supervisor" (CSV).
- the set of subroutines executed in Privileged mode and belonging to the external subroutine segment of the task group. These subroutines can be called in User mode by the instruction "Call External Subroutine" (CLQ) under certain conditions (see CLQ description, Chapter IX).
- declaration macroinstructions for segments accessed only by the supervisor. These macroinstructions are supplied with the monitor.

Taking into account these facilities, most programmers working on a given application must be familiar only with the contents of Volume I of this Manual.

II-2. USER MODE

User mode is subdivided into two submodes :

- "program" mode
- "external subroutines" mode, called simply the "subroutine mode".

"Program" mode is for the execution of the task's own program.

"Subroutine" mode is for the execution of an external subroutine which can be shared between several tasks.

II-2.1. Status Indicators

Program operating mode is reflected by three status indicators which are saved in the task context :

PV	= 0	NON PRIVILEGED MODE
PV	= 1	PRIVILEGED MODE
SV	= 0	NON SUPERVISOR MODE
SV	= 1	SUPERVISOR MODE
SP	= 0	NON SUBROUTINE MODE
SP	= 1	SUBROUTINE MODE

User mode is characterized by : PV=0 and SV=0.

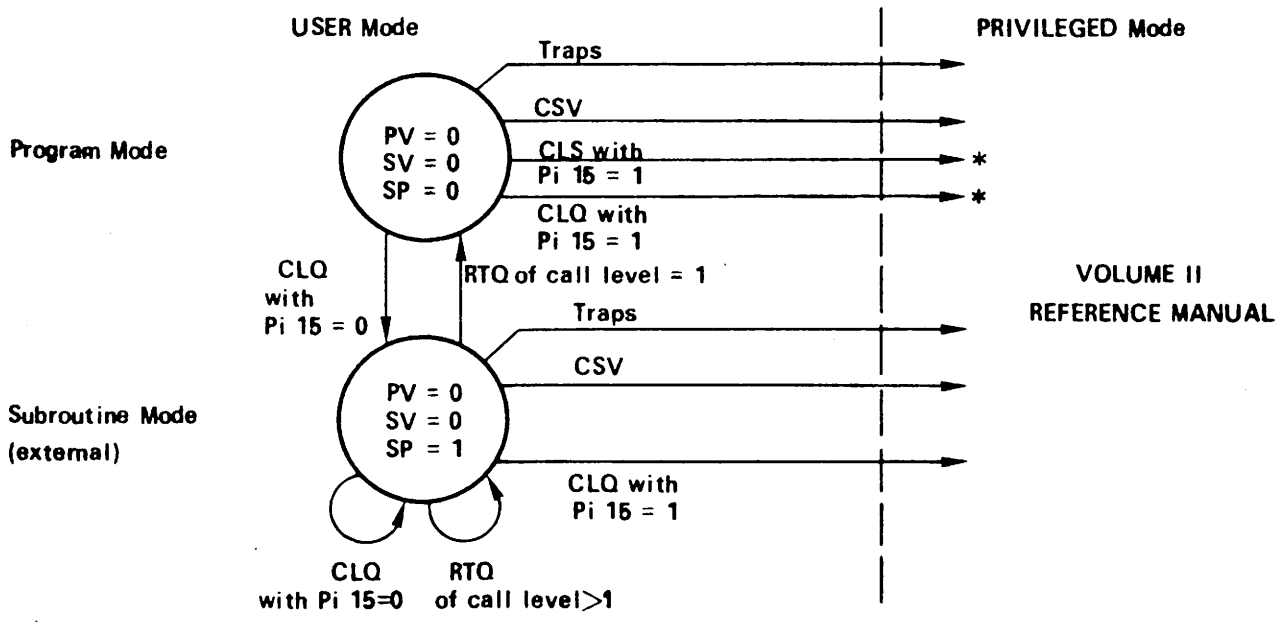
II-2.2. Intermode Transitions (see figure 2-4.)

Passing from one mode to another is performed :

. explicitly by the execution of call and return instructions to supervisor modules and external subroutines :

CLS	Call missing section
CSV	Call monitor module
CLQ	Call external subroutine
RTQ	Return from external subroutine

. implicitly via a trap.



* Refer to I-7.2

Fig. 2.4 - INTERMODE TRANSITIONS

II-2.3. Segments Directly Accessible to a Task Being Executed in User Mode
(see figure 2-5.)

A task being executed in User mode has direct access to 4 segments :

II-2.3.1. Internal Task Segments

- The "program segment" internal to the task, defined by base G and length GL registers ;
- the "stack segment" internal to the task, defined by base SB and length SL words present in the task context.

II-2.3.2. External Task Segments

- The external "subroutine segment" defined by base Q and length QL registers, and generally common to all group tasks ;
- the external "data segment" defined by base Z and length ZL registers, which may be common to all group tasks.

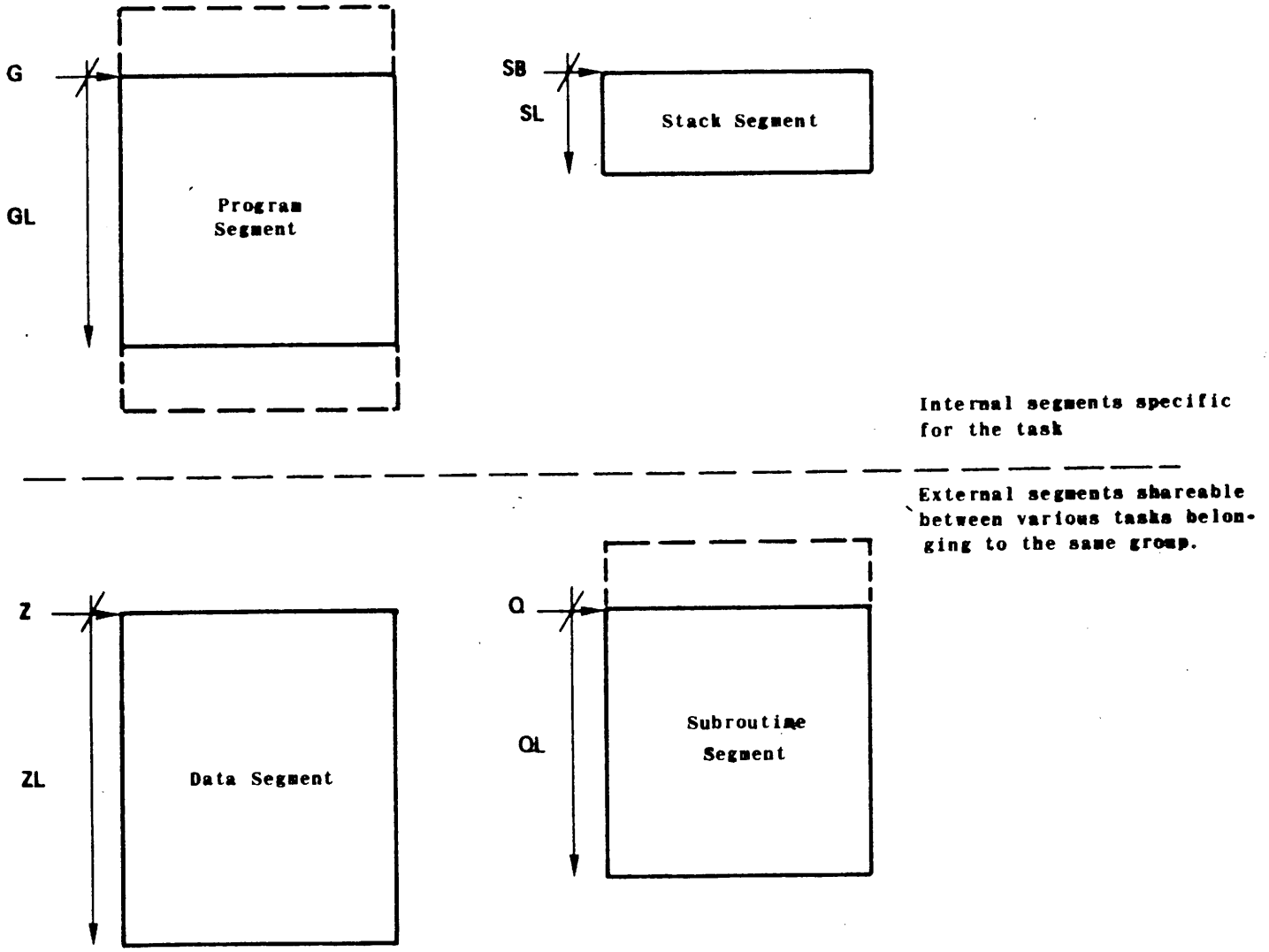


Fig. 2-5 - SEGMENTS DIRECTLY ACCESSIBLE TO A TASK EXECUTED IN USER MODE

II-2.4. Program Segment (see figure 2-6.)

A program segment is associated with each task when the task is created by the monitor.

A program segment comprises a number of sections :

- a Common Data Section (CDS)
- a Main Program Section
- generally, Subroutine Sections

A program or subroutine section contains two subsections :

- a Local Data Subsection (LDS)
- a Local Program Subsection (LPS)

An LDS can be common to several LPSs.

A Subroutine Section of this segment is called via instruction CLS and returned to via instruction RTS.

When the called section is not in memory at the time of the CLS, it is automatically transferred into memory by the monitor.

Sections and subsections are defined by pointers relative to base G and located in a table under G, called PRT, and generated by the Linkage Editor.

The program segment is defined by base G and length GL registers.

G and GL are automatically saved and restored whenever the context is switched.

The user can access from the program segment by using suitable instructions and addressing types :

- Data :
  - . of the program segment itself,
  - . of the data segment based by Z,
  - . of the stack segment of the task based by SB.
- The Subroutines :
  - . of the program segment itself via instruction CLS ,
  - . of the subroutine segment based by Q via instruction CLQ,
  - . of the monitor via instruction CSV

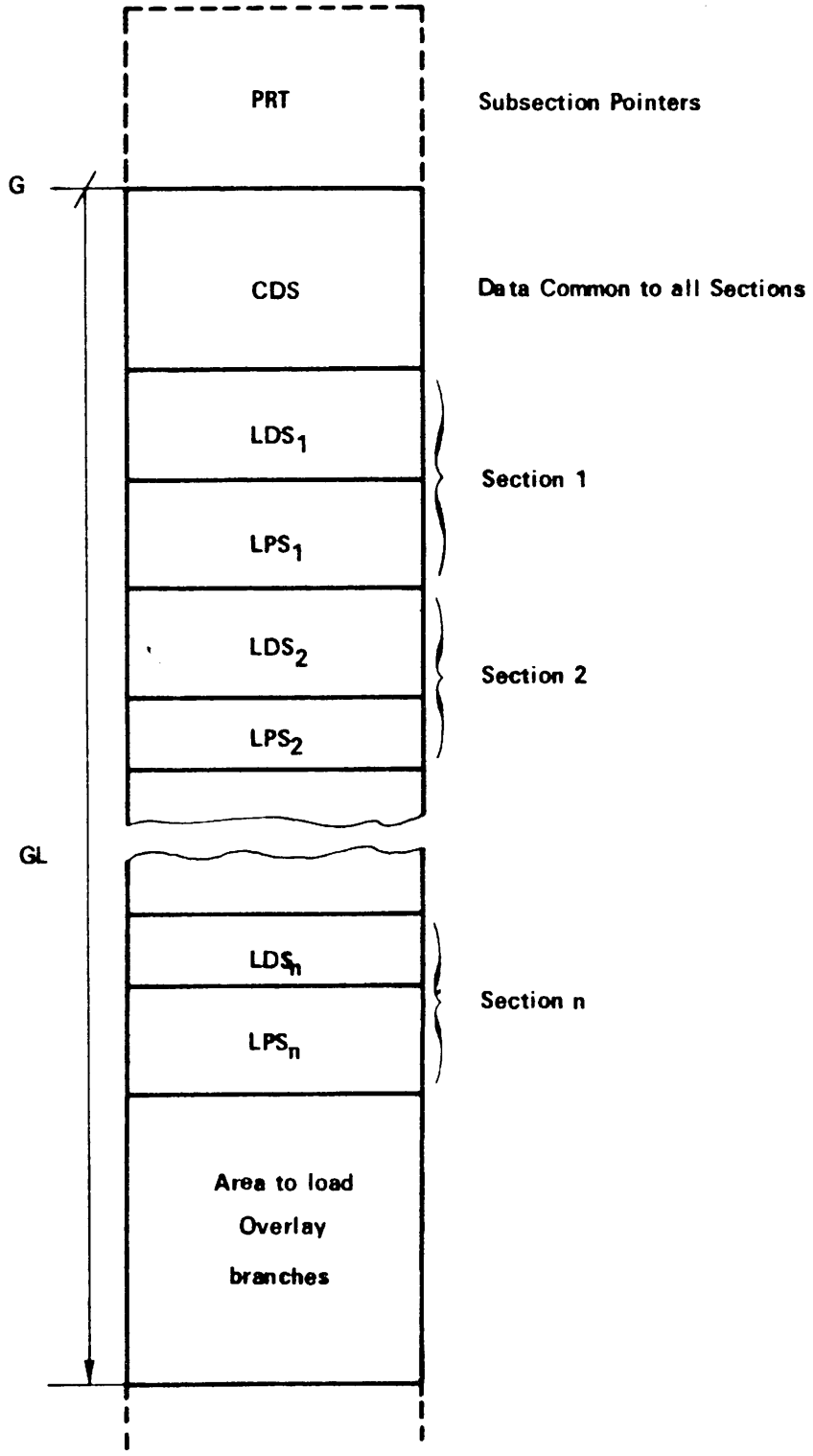


Figure 2.6 - PROGRAM SEGMENT TOPOLOGY

II-2.5. Stack Segment (see figure 2-7.)

A stack segment is associated with each task.

Stack elements have variable lengths, the length being defined when the elements are stacked in using instruction PUSH n. Stacked words are the first n words of the CDS of the task.

Instruction PULL restores the last stack element in the CDS.

The stack segment of a task is defined by three words from its context :

SB : Stack Base : Base of stack segment
SL : Stack Length : Length of stack segment
ST : Stack Top : Top of stack segment

Instructions PUSH and PULL directly operate on these three context words.

Instructions PUSH and PULL can be executed in the program segment (internal) or in the subroutine segment (external).

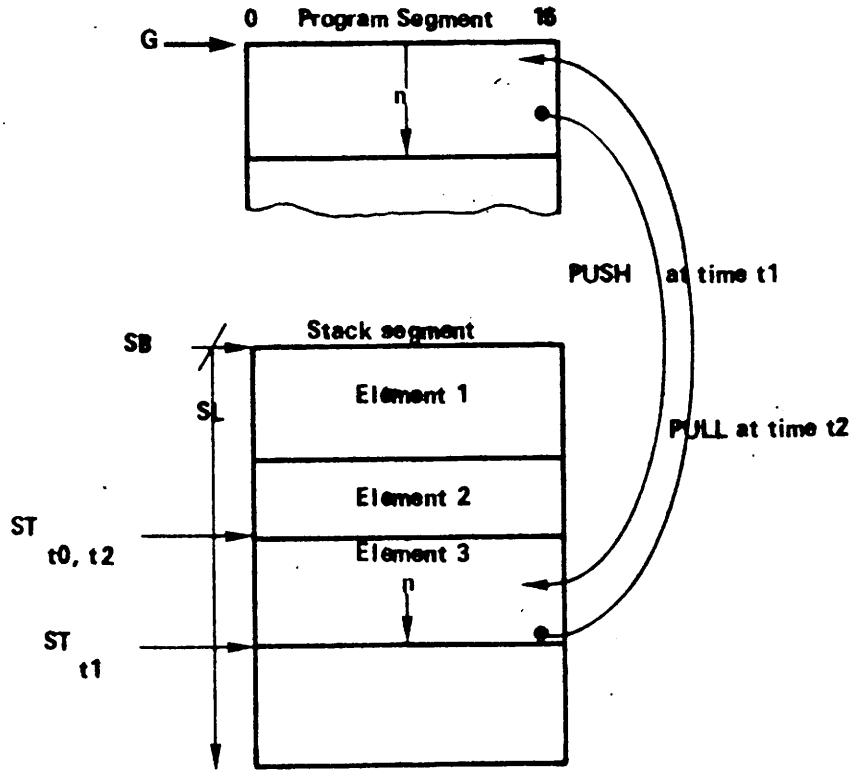


Fig. 2.7 - STACK SEGMENT TOPOLOGY

II-2.6. Data Segment (see figure 2-8.)

Each task has access to a data segment at a given time. This data segment may be :

- either the data segment shared between the group tasks (SDS = Shared Data Segment) ;
- or a system data block momentarily allocated to the task by the monitor.

The data segment is defined by base Z and length ZL registers.

The nature of the data segment referenceable at a given time depends on the monitor which supplies primitives for loading Z and ZL with the suitable descriptor (see Monitor Utilization Manual) to the user.

Z and ZL are automatically saved and restored whenever the context is switched.

The data segment can be accessed from the program segment or from the subroutine segment.

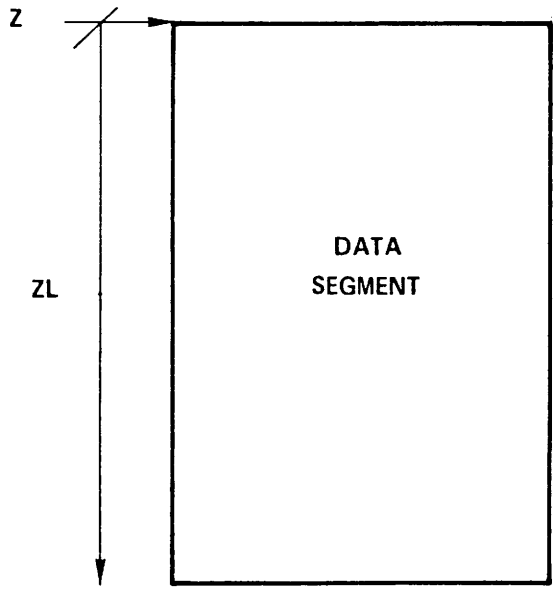


Fig. 2.8 - DATA SEGMENT TOPOLOGY

II-2.7. Subroutine Segment (see figure 2-9.)

Each task has access to a subroutine segment. This subroutine segment is defined when the task is created and is :

- either the subroutine (reentrant or not) segment shared between the group tasks (SPS = Shared Program Segment)
- or a reentrant subroutine segment shared between several tasks whose program segments amount to a CDS and to instruction CLQ (Call Main Subroutine of the Subroutine Segment).

A subroutine segment has the same structure as a program segment but without a CDS and without overlays. It comprises several subroutine sections.

A subroutine section contains :

- a data subsection (LDS = Local Data Subsection)
- a program subsection (LPS = Local Program Subsection)

An LDS can be common to several LPSs.

A subroutine section of this segment is called via instruction CLQ and returned to via instruction RTQ.

Sections and subsections are defined by pointers relative to Q and located in a table under Q, called PRTQ, generated by the Linkage Editor.

The subroutine segment is defined by base Q and length QL registers.

Q and QL are automatically restored from the contents of the context at each switching.

Note : Q and QL registers contents are not saved at the time of a context switching. The subroutine segment is always the same during the task's entire lifetime.

The user can access from the subroutine segment by using suitable instructions and addressing types :

- Data :
  - . of the subroutine segment itself,
  - . of the program segment based by G,
  - . of the stack segment based by SB.

- Subroutines :

. of the subroutine segment itself via instruction CLQ,

. of the monitor via instruction CSV.

Explanations :

PRTQ		: Subsection Pointers
LDS1	LPS1	: Section 1
LDS2	LPS2	: Section 2
LDSn	LPSn	: Section n

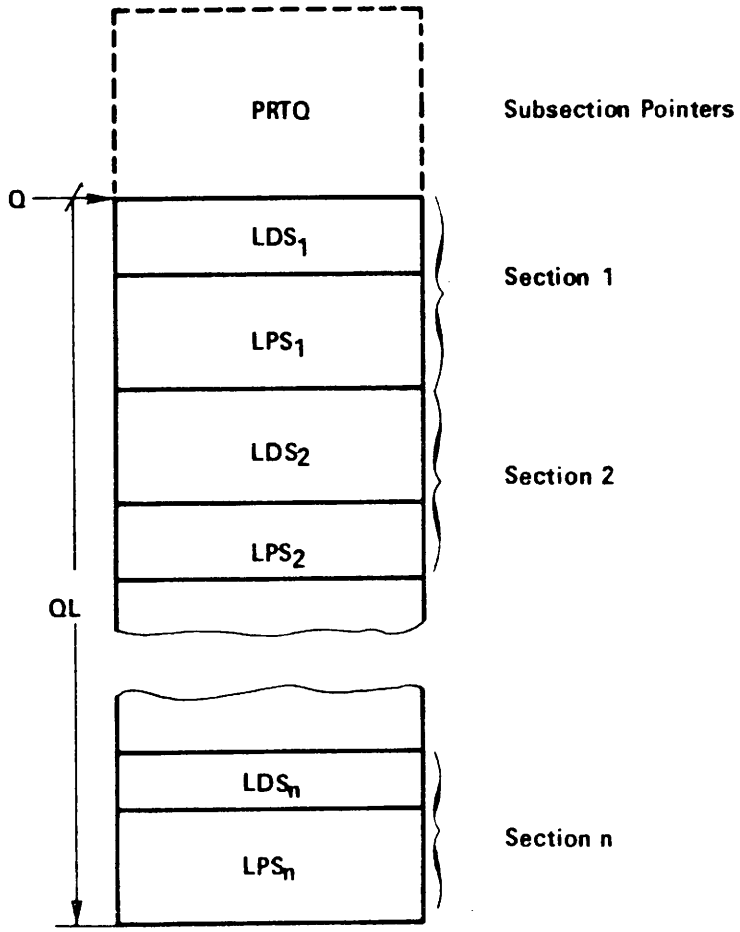


Figure 2.9 - SUBROUTINE SEGMENT TOPOLOGY

II-3. GENERAL REGISTERS

II-3.1. Register Access

The general registers are addressable in User mode via general instruction LDR and via specialized instructions (LDA, LDX, ...). All general registers are 16-bit registers.

II-3.2. Register Addresses (see figure 2-10.)

General registers have one address for access via instruction LDR.

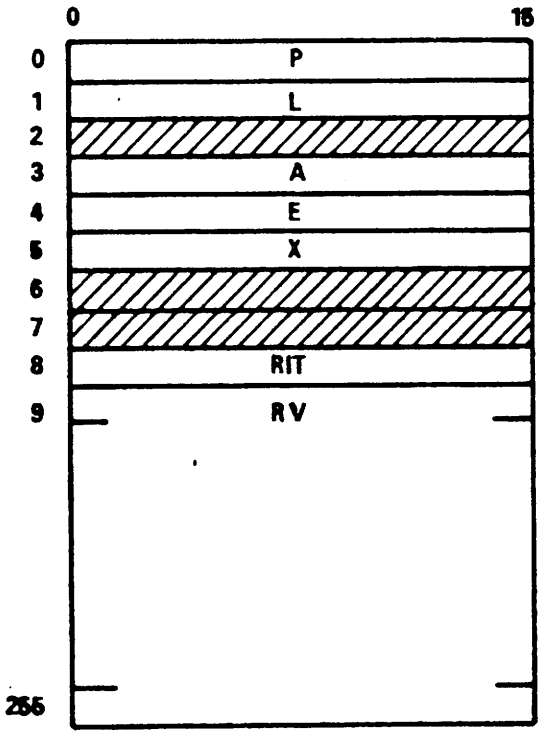


Fig. 2.10 - REGISTER ADDRESSES

II-3.3. Register Description

P = Program Counter

Contents of register P give on 16 bits the byte address (always even) relative to the following :

- . G in program mode
- . Q in external subroutine mode

of the instruction being executed. When this instruction has been executed, P contains the relative address of the next instruction to be executed.

L = Current Address of Local Data Area

Contents of register L give on 16 bits the byte address relative to :

- . G in program mode
- . Q in external subroutine mode

of the beginning of the addressable data area in Local mode (refer to chapter IV).

Instructions ICL (InCrement L), DCL (DeCrement L) permit modifying this address.

Base L is initialized with the start address of the Local Data Subsection via instructions CLS and CLQ.

A	= Accumulator
E	= Accumulator Extension
X	= Index Register
RIT	= Interrupt levels and ranks
RV	= Channel Registers of Integrated Couplers

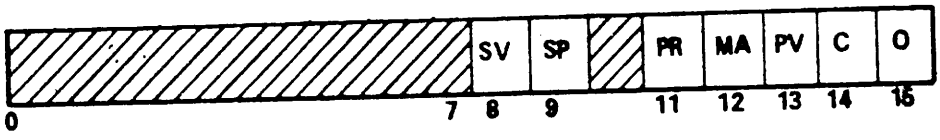
Note : Although all these registers can be read by instruction LDR executable in User mode, registers RIT and RV are normally operated on only by the Monitor ; for their description, refer to Volume II of this Manual.

II-4. STATUS INDICATORS

II-4.1. Indicator Access

Indicators are accessible in User mode via instruction LDI (Load Indicator Into a Register).

II-4.2. Indicators Setting In A After Execution of LDI :



Bit 15 O = Overflow

This indicator at 1 means overflow for an arithmetic instruction : its meaning depends on the instruction for the other instructions.

Bit 14 C = Carry

This indicator at 1 means carry for an arithmetic instruction ; its meaning depends on the instruction for the other instructions.

Bit 13 PV = Privileged Mode

This indicator at 0 means that the user has no access to instructions executable in Privileged mode.

Bit 12 MA = Mask

This indicator at 1 means that interrupt acceptance is forbidden (except during execution of instruction DIT).

Bit 11 PR = Protection

This indicator at 1 means that the user can access the memory cells protected by a protection bit.

Bit 9 SP = Subroutine Mode

This indicator at 0 means that addresses are relative to base G for local addressings (DL, IL, ILX, ILEX) ; this indicator at 1 means that addresses

are relative to base Q for local addressings (DL, IL, ILX, ILEX) (Refer to Chapter IV).

Bit 8 SV = Supervisor Mode

This indicator at 0 means that addresses are relative to base G (General Base) or base Q (Shared Program Base).

In User mode, SV = 0 , PV = 0
Bits 0 through 7 and bit 10 are not used.

II-5. BASE AND LENGTH REGISTERS

Base and Length Registers are not accessible in User mode so that segments are protected from each other (intersegment protection). Descriptors of segments accessible in User mode must be loaded beforehand in these base registers by a privileged program (usually, the monitor) called by the task itself or loaded in its context by another privileged task (generally, belonging to the monitor).

Base registers contain an octo-word address on 16 bits. Hardware automatically adds with the desired justification the address relative to the beginning of a segment and the base value (see figure 2-11).

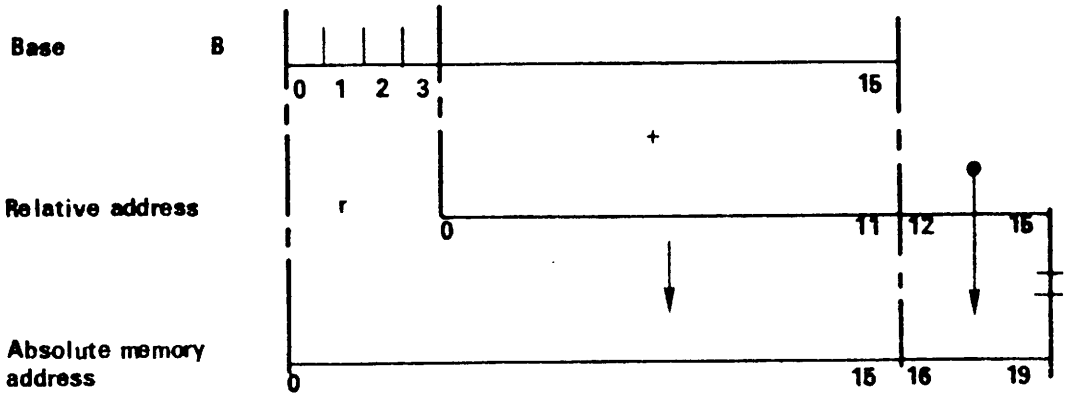


Fig. 2.11 - ABSOLUTE ADDRESS COMPUTATION

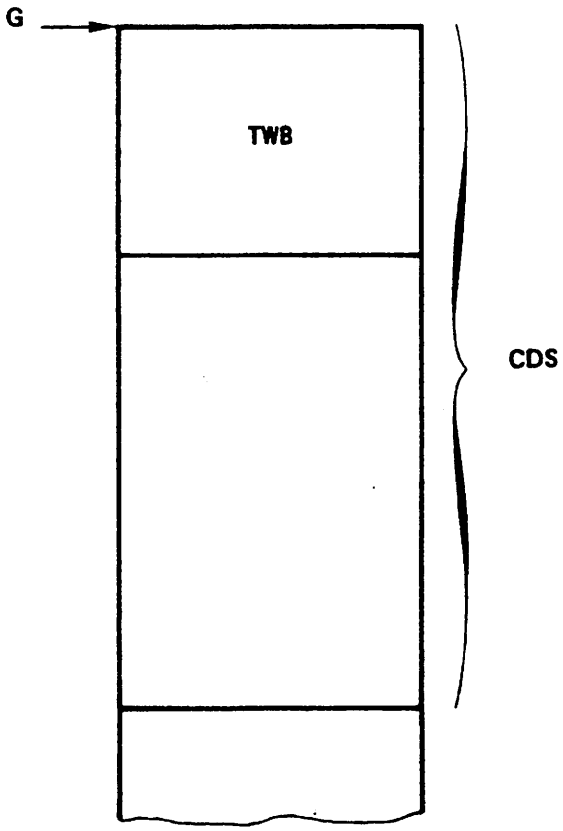


Fig. 2.12 - CDS ORGANIZATION

II-6. CONTEXT

The task's context is not accessible in User mode so that segments are protected from each other and illegal mode changes are impossible.

General registers, indicators, base and length registers (G, GL) (Z, ZL) are automatically saved and restored whenever the context is switched.

Base and length registers (Q, QL) are not saved but are automatically restored with the contents of the new context at context switching time.

The stack descriptor (SB, SL, ST) is permanently resident in the context ; instructions PUSH and PULL have direct access to the context.

II-7. USER PROGRAM COMMUNICATION ELEMENTS (NORMAL OR SHAREABLE)

II-7.1. Task Working Block (see figure 2-12.)

The first words of the CDS of a program constitute the Task Working Block (TWB). This area is used :

- by instruction CSV to store P and L values and calling program indicators ;
- by the monitor modules as a work area and/or parameter transmission area ;
- by external reentrant subroutines as a work area and/or parameter transmission area.

Programs invoking monitor sections or reentrant subroutine segment sections must therefore begin their CDS with a TWB reservation.

Instructions PUSH and PULL permit saving all or part of the TWB and thus offer recurrence capabilities of shareable subroutines and monitor modules.

II-7.2. Section Call And Section Assignment Tables (PRT and PRTQ) (see figures 2-13 and 2-14.)

Section assignment tables are automatically created by the Linkage Editor and are not user accessible. These tables are used by instructions CLS and CLQ.

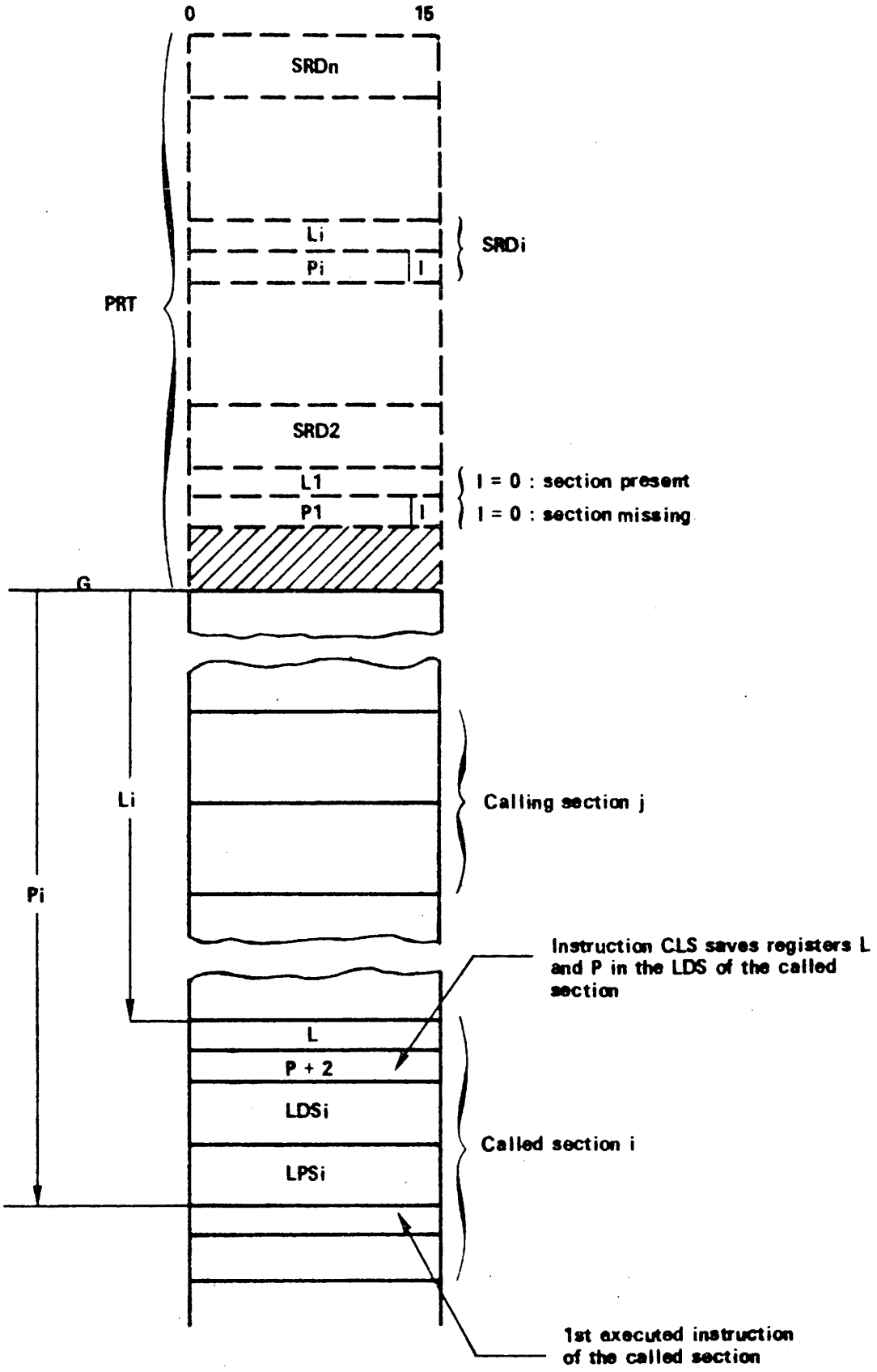


Fig. 2.13 - PRT TOPOLOGY

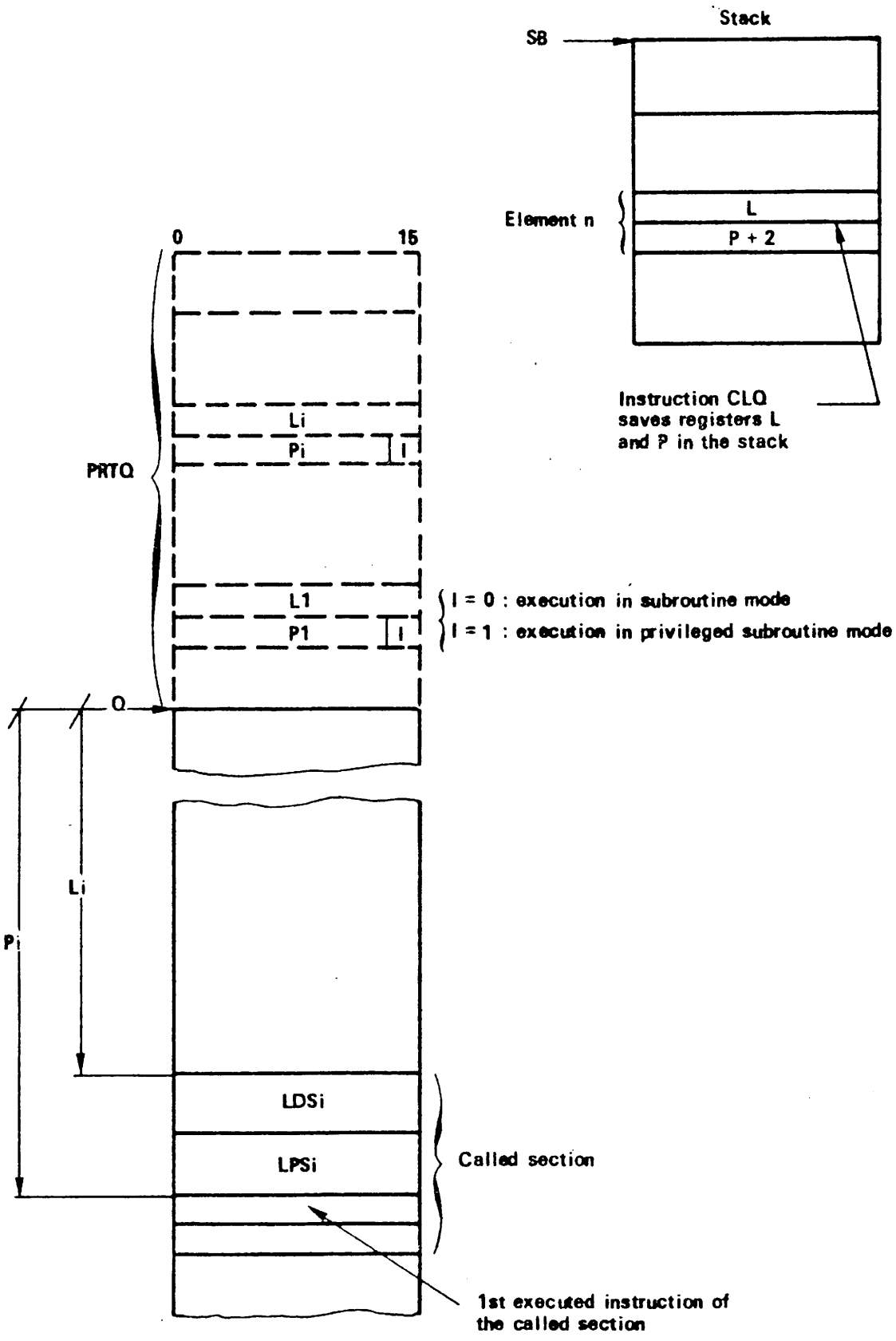


Fig. 2.14 - PRTQ TOPOLOGY

A resident section or a section belonging to an overlay branch is represented by a double-word (SRD : Section Relocation Double-Word) which contains the initial values of L and P (called respectively Li and Pi) relative to G (Program Segment) or to Q (Subroutine Segment), and an indicator (Pi15).

Initial value of L is the start address of subsection LDS.

Initial value of P is the entry point address in the section ; this is not necessarily the start address of subsection LPS.

PRT and PRTQ are made up respectively of the set of SRDs of the program segment and subroutine segment sections.

SRD_0 does not exist.

II-7.2.1. Call Program Segment Section (CLS) and Return (RTS)

A program segment section can be called only from a section of the same program segment.

Indicator Pi15 of the SRDi indicates if the called section is or not in memory.

- If the section is missing, a supervisor module is automatically called via instruction CLS. This supervisor module loads in memory the overlay branch to which the missing section belongs.
- If the section is present, registers L and P+2 are saved by instruction CLS in the first 2 words of the LDS of the called section and registers L and P initialized with the contents of the SRDi.

Values L and P+2 saved in the LDS of the called section are used by instruction RTS to return to the calling section.

II-7.2.2. Call Subroutine Segment Section (CLQ) and Return (RTQ)

A subroutine segment section can be called from :

- a section of the same subroutine segment
- a program segment section

Indicator Pi15 of the SRDi indicates if the called section must be executed :

- in subroutine mode (user)
- or in privileged subroutine mode

Register L, P+2 and indicators are saved by instruction CLQ in the task's stack.

Values L, P+2 and indicators saved in the stack are used by instruction RTQ to return into the calling section.

Note : Only indicator SP (Shared Mode) is restored by RTQ if the subroutine has been executed in normal mode to prevent an illegal mode change executed in a non-privileged, shared mode by a subroutine modifying indicators saved in the stack.

II-7.3. Call Monitor Section

A monitor section (refer to Volume II) can be called from :

- a program segment section
- a subroutine segment section

A monitor section i can be called explicitly or implicitly :

- explicitly via instruction CSVi
- implicitly via :
  - . a standard trap or a trap specific for the execution of an instruction. This is the case for monitor section 0 which performs trap processing.
  - . execution of a CLS instruction, if the called section is not in memory. This is the case for monitor section 1 which loads the overlay branch. (Refer to § II-7.2.1.).

II-7.3.1. Call Monitor Section Explicitly Via Instruction CSVi (Call Supervisor) (see figure 2-15.)

The supervisor comprises two types of sections :

. sections that may be called in User mode from the program segment or the subroutine segment ;

. sections that may be called only in Privileged mode. When one of these sections is called in User mode, a trap occurs (Refer to Volume II).

If the monitor section can be called in User mode, L, P+2, and indicators are stored in the first 3 words of the CDS of the calling task.

Indicators SV, PR, and PV are forced to 1, SP to 0.

Values L, P+2, and indicators saved in the CDS of the calling task are restored when control is returned to the calling task.

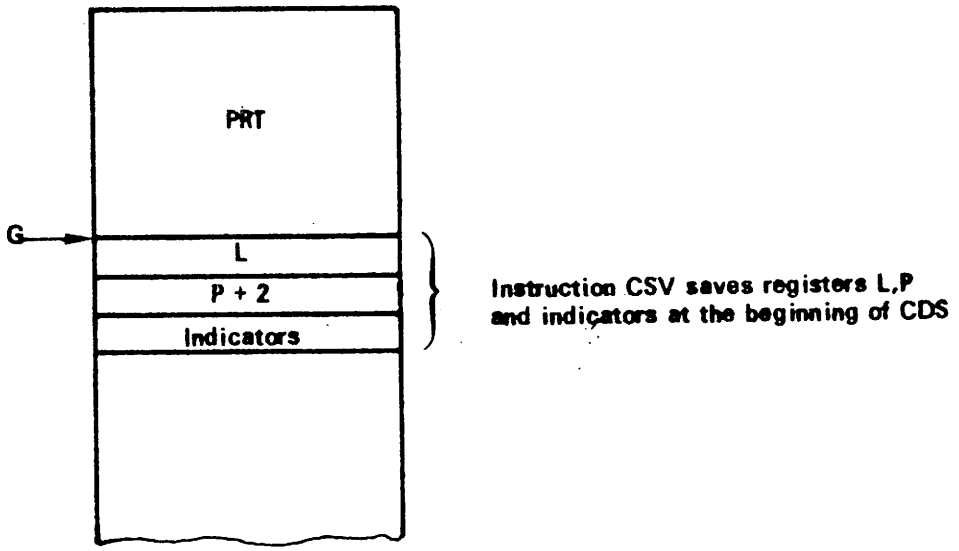


Fig. 2.15 - EXPLICIT CALL OF A MONITOR SECTION

II-7.3.2. Call Monitor Section 0 Implicitly Via Traps

Standard Causes of Traps Caused by a User Program :

Standard trap causes are as follows :

Trap Type	Standard Cause
DT	Segment Length Overflow : attempt to access a segment with a relative address greater than its length.
VM	Mode Violation ; attempt to execute an instruction in an illegal mode for this instruction.
PM	Memory Protection ; attempt to write in a memory cell protected by the protection bit while the protection key indicator is zero.
AI	Non-existent Address ; attempt to access a non-existent memory cell.
II	Illegal instruction.

Special Trap Causes

Special causes of traps due to a user program are described for each instruction concerned.

Trap Processing

Traps are processed by Monitor section 0. Refer to Volume II of this Manual and the Monitor Utilization Manual.

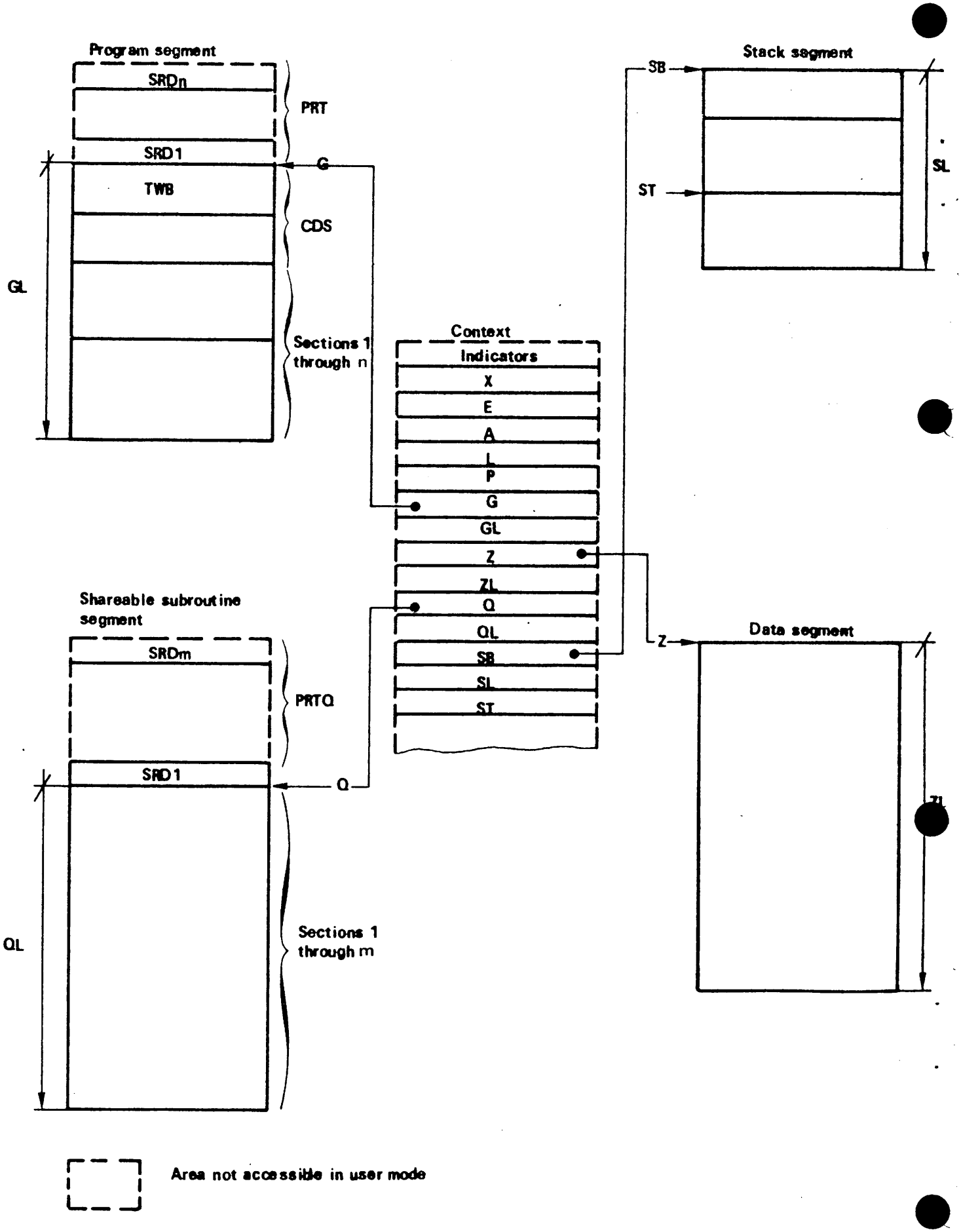


Fig. 2.16 - USER PROGRAMS STRUCTURE

III - ASSEMBLY LANGUAGE

III-1. INTRODUCTION

The MITRA 125 Assembler is a macro-assembler with which the user can write data and machine instructions in a symbolic format (Assembly Language) and define and call macro-instructions. The Assembler generates a Relocatable Binary (RB) object text.

The MITRA 125 Assembler can satisfy only references to the assembled source text symbols. The MITRA 125 Linkage Editor will convert the Relocatable Binary object texts separately assembled into an object text representing a Relocatable Memory Image (RMI).

The MITRA 125 Assembler operates in two passes :

- during the first pass, the source text is read, macro-instructions are expanded, and definitions of symbols are recorded in a table ;
- during the second pass, the Relocatable Binary object text is generated, the source text is edited as well as the symbols dictionary and the list of possible errors.

Source text format is an 80-character card image (characters 73 through 80 being ignored by the Macro-Assembler).

The MITRA 125 Assembly Language includes :

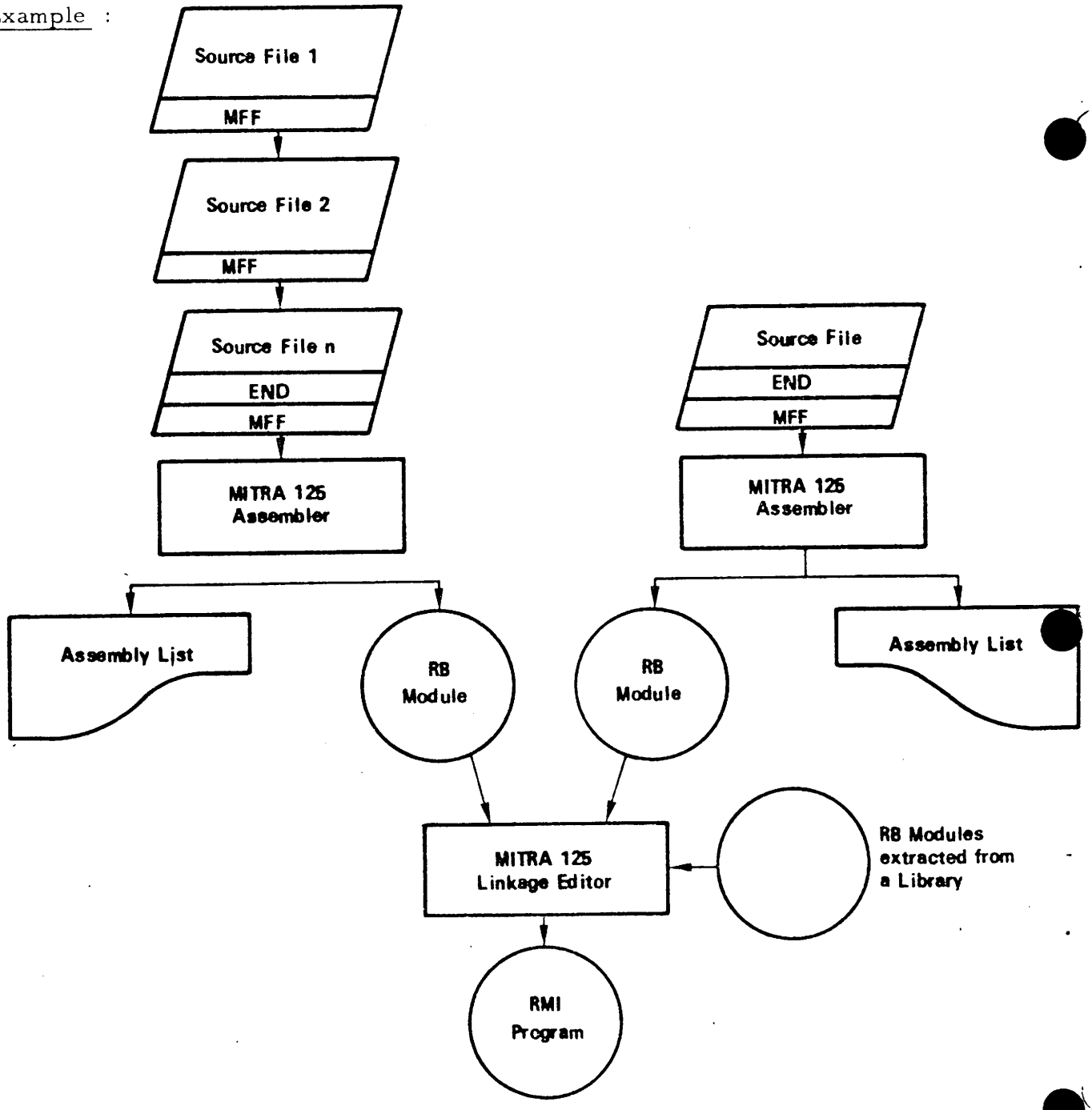
- the possibility to define and use macro-operations,
- a set of sectioning pseudo-instructions defining the user program's software structure,
- a set of assembly pseudo-instructions defining the processing to be performed by the Assembler,
- a machine instruction set.

Remark : The MITRA 125 Assembler can translate the MITRA 15 Assembly Language.

III-2. SOURCE TEXT

The source text written in MITRA 125 Assembly Language constitutes the Assembly Unit or Assembly Module. It contains one or more concatenated source file(s) ; each file is terminated by an End Of File Mark (MFF), the last file being terminated by a source line containing pseudo-instruction END (which indicates end of Assembly Module) and followed by an End Of File Mark (MFF).

Example :



N.B. : MFF = End of File Mark

III-3. MITRA 125 ASSEMBLY LANGUAGE LINE FORMAT

III-3.1. Instruction Or Pseudo-Instruction Line

An instruction or pseudo-instruction line comprises four fields maximum separated by one or more blanks.

Label Field (optional)

This field begins necessarily in column 1 ; it contains a 1 to 6 alphanumeric character symbol whose first character must be alphabetical, and ends with an empty column.

Command Field

It begins in the 1st non-empty column which follows the label field (or the 1st non-empty column which follows column 1 if the label field is not used), and ends with an empty column.

Argument Field (optional)

It begins in the 1st non-empty column which follows the command field and ends with an empty column.

Comments Field (optional)

It begins in the first non-empty column which follows the argument field.
To insure compatibility with the MITRA 15 Assembly Language, the comments field can begin in the 1st column which follows special character " * ".

III-3.2. Comments Line

A comments line is a line whose 1st character is special character " * ".

III-3.3. Meta-Instruction Line

A meta-instruction line is a line whose 1st character is special character " < ".
It is considered as a comments line by the Assembler.

III-3.4. Label Line

A label line is a line which contains a 1 to 6 alphanumeric character label in column 1, optionally followed by a comments line. The label line is processed by the assembler when it searches for a label located in the operand field of a GOTO pseudo-instruction.

III-3.5. Blank Line

A blank line is considered as an empty comments line.

III-3.6. Continuation Line

A continuation line is a line whose 1st character is special character ";". A continuation line can be followed only by a MACRO pseudo-instruction line (terminated by special character ";"), or an instruction line containing a macro-instruction (terminated by special character ";").

III-4. BASIC CHARACTERS

The Assembler accepts all characters recognized by input peripherals. These characters constitute an EBCDIC code subset. No check is performed on the characters appearing in the comments field or in a character string.

Basic characters are classified as follows :

- Alphanumeric Characters :

  alphabetical : A through Z as well as ":"
  decimal : 0 through 9
  hexadecimal : 0 through 9 and A through F

- Special Characters :

  Blank + - * / . , ( ) " = # $ % & @ ' ? ... etc.

III-5. BASIC ELEMENTS

The Assembly Language comprises constants, symboles, operators, and separators.

III-5.1. Constants

Data can be entered directly in assembly language as integer or floating constants.

III-5.1.1. Integer Constants

- Decimal Integer Constants :
  A decimal integer constant is a sequence of at most 5 decimal characters. These characters represent a digital $base_{10}$ value whose maximum depends on the number of bits available for its $base_2$ representation, but which can never exceed $2^{15} - 1$, i.e. 32767.

  Example : 75 002 31720

- Hexadecimal Integer Constant :
  A hexadecimal integer constant is a sequence of at most 4 hexadecimal characters preceded by special character "&". These characters represent a digital $base_{16}$ value whose maximum depends on the number of bits available for its $base_2$ representation, but which can never exceed $2^{16} - 1$, i.e. &FFFF.

  Example : &0 &000F &8000

III-5.1.2. Decimal Floating Constants

A decimal floating constant is a sequence of decimal characters with a decimal point and optionally followed by an exponent of form E $\pm$ exponent value expressed on 1 or 2 decimal characters.

Example : 13.7 0.24E - 02

III-5.1.3. Character String Constants

A character string constant is a sequence of characters other than the character "quotation marks", placed between quotation marks.

If the user wishes to insert the "quotation mark" character in the string, it will be represented by two consecutive quotation marks.

Example :

"CHAIN"	CHAIN
"A" "B"	A"B
"&F1"	&F1

III-5.2. Symbol

A symbol is a sequence of 1 to 6 alphanumeric characters ; the first character must be alphabetical. It cannot contain special characters.

Symbols can be classified into two categories : command mnemonics and user symbols.

III-5.3. Command Mnemonics

They appear in the command field of an instruction or pseudo-instruction and designate the computer instruction or pseudo-instruction to be executed by the Assembler.

III-5.4. User Symbols

They appear either in the label field, in the command field, or in the argument field of an instruction or pseudo-instruction.

A symbol is called defined when it appears in the label field of an instruction or pseudo-instruction.

A symbol is called forward or forward referenced when it appears in the argument field of an instruction or pseudo-instruction prior to being defined.

A symbol is called predefined when it appears in the argument field of an instruction or pseudo-instruction and has already been defined.

A symbol is called external or externally referenced when it appears in the argument field of an REF pseudo-instruction.

III-5.4.1. Value Assigned to a Symbol

For pseudo-instructions "GOTO, BASE, BND, DEF, REF, XREF, FIN, END, PAGE, TITLE, SPACE, LIST, NOLIST, EXT, FINR, IF FINM, FIND, GLOBAL", no value is assigned to the symbol in the label field.

For pseudo-instructions "EQU", SET", the value assigned to the symbol in the label field is that of the expression in the argument field.

For all cases, the value assigned to the symbol in the label field is that of the location counter.

III-5.4.2. Symbol $

A symbol appearing in the argument field of an instruction can be reduced to special character $. It then designates the current value of the location counter.

III-5.4.3. Symbol %

A symbol in the argument field of an instruction can be reduced to special character %. It designates then the current value of the index of loop DO or DUP/FIND.

III-5.5. Operators

III-5.5.1. Arithmetic Operators

- unary minus	(example : -3)
- subtraction	(example : A - 3)
+ unary plus	(example : +3)
+ addition	(example : A + 3)

* multiplication	(example : A * 3)
/ division	(example : A / 3)

III-5.5.2. Addressing Operators

= parameter mode	(example : = 3)
@ indirection	(example : @ A)
$ extended addressing	(example : $ A)
,X indexing	(example : A,X)

III-5.5.3. Generation Operators

Address to be generated relatively to base G in Supervisor Mode.
(Example : A DATA #B)

III-5.5.4. Membership Operator

The element belongs to the CDS (Example : REF #A)

III-5.5.5. Logical Operator

.AND.	Logical AND	(Example : A.AND.3)
.EOR.	Exclusive OR	(Example : A.EOR.3)
.IOR.	Logical OR	(Example : A.IOR.3)
.SHR.	Shift Register Right Single	(Example : A.SHR.2)
.SHL.	Shift Register Left Single	(Example : A.SHL.2)

III-6. EXPRESSIONS

III-6.1. Expression Definition

An expression comprises one or more symbols or integer constants interconnected by arithmetic or logical operators.

An expression is represented by a unique value computed by the Assembler or by the Linkage Editor.

III-6.2. Expression Symbols Definition

During the first pass of the Assembler, the following are declared defined :

- symbols that appeared in the label field of an addressing pseudo-instruction (RES, DATA, ...) ;
- symbols that appeared in the label field of an EQU or SET pseudo-instruction.

During the second pass of the Assembler, the following are declared defined :

- symbols that appeared in the label field of an EQU or SET pseudo-instruction ;
- all symbols of the current subsection located in the label field of an addressing pseudo-instruction (RES, DATA, ...) ;
- symbols that appeared in the label field of an EQU or SET pseudo-instruction ;
- all current subsection symbols located in the label field of an EQU pseudo-instruction.

III-6.3. Expression Classes

There are three classes of expressions :

- Class P1 Expression :

  This is an expression computable at the first Assembler pass, i.e., where all symbols of the expression are defined at the first pass.

- Class P2 Expression :

  This is an expression computable at the second Assembler pass, i.e., where all symbols of the expression are defined at the second pass.

- Class P3 Expression :

  This is an expression computable with the Linkage Editor, i.e., where all symbols of the expression are defined at Linkage Editing.

III-6.4. Type Of Expression

There are two types of expressions :

- An expression may be absolute, i.e., independent of any subsection base.
- An expression may be relative, i.e., dependent on a subsection base.

III-6.5. Expression Evaluation

An expression is evaluated according to the following decreasing priority rules :

- unary operator (+ and -)
- Shift Register Right/Left Single (.SHR. and .SHL.)
- multiplication and division (* and /)
- addition and subtraction (+ and -)
- logical AND (.AND.)
- logical inclusive OR and logical exclusive OR (.IOR. and .EOR.)

An expression containing several operators is evaluated left to right, each operation being performed according to the priority rules between the partial results surrounding the operator.

Only additions and subtractions can be performed on symbols relative to a subsection base or on external references.

III-6.6. Determination Of The Type Of Expression

A function F1 which associates :

- value 0 to any absolute symbol (or constant)
- value G to any symbol relative to G
- value L to any symbol relative to L
- value P to any symbol relative to P
- value G to any external reference relative to G
- value E to any external reference not relative to G
- operator + to operator +
- operator - to operator -

- operator + to other operators

is defined.

Let R1 be the result of the expression's evaluation after each element is transformed by function F1.

Another function F2 which associates :

- value 0 to any absolute symbol (or constant)
- value T to any symbol relative to a base
- value T to any external reference
- operator + to operator +
- operator - to operator -
- operator + to other operators

is defined.

Let R2 be the result of the expression's evaluation after each element is transformed by function F2.

III-6.7. Class And Type Of An Expression Located In The Argument Field Of An Instruction

Addressing Mode	R1					R2		Class		
	0	G	L	P	E	0	T	P1	P2	P3
DL,IL,ILX,ILE,ILEX	X		X		X	X	X	X	X	X
DG,IG,IGX,IGXE	X	X				X	X	X	X	X
P,PX	X	X	X	X	X	X	X	X	X	X
RM,RP				X			X	X	X	

If the expression is class P3, i.e., computable at Linkage Editing time, the Assembler selects between local addressing and general addressing depending on the type of the first symbol of the expression.

III-6.8. Class And Type Of An Expression Located In The Argument Field Of A Pseudo-Instruction

Pseudo-Instruction	R1 0 G L P E	R2 0 T	Class P1 P2 P3
RES,GOTO,DO,DIM,DUP,...	X	X	X
EQU,SET	X X X X X	X X	X
DATA,GEN,DATAB,PTW, PTB,...		X X	X X X

IV - GENERAL USER PROGRAM STRUCTURE

IV-1. GENERAL

IV-1.1. Definitions

There are two types of user programs :

- programs executed in "program" mode and based by G
- external subroutines executed in "subroutine" mode and based by Q.

The program segment and external subroutine segment are globally relocatable with all addresses being relative to their location bases G and Q, respectively.

Write rules are the same for the program segment and the external subroutine segment except for the following :

- a shareable reentrant subroutine must have only constants in its subsection LDS.
- instruction CLS is not executable in "subroutine" mode.
- instruction RTQ is not executable in "program" mode.

IV-1.2. User Program Source Structure

The program segment and the external subroutine segment are declared as follows :

1) a common declarations section (SDEC)

The common declarations section must always, if it exists, be declared before the set of Macros and before the common data section of the assembly unit.

2) a set of Macros (MACRO)

The set of macros must always, if it exists, be declared before any assembly unit section.

3) a common data section (CDS)

The common data section - effective or dummy - must always, if it exists, be

declared before any assembly unit subsection. The subroutine segment has never an effective CDS.

4) a set of user sections

Each section necessarily has an executable program subsection (LPS) which defines the section. A local data subsection (LDS) is normally associated with an executable program subsection.

If it is effectively defined in the same assembly unit, the LDS precedes necessarily the LPS in the source text.

Several LPSs can be associated with the same LDS ; there are as many sections as LPSs.

Example :

Section SDEC
Definition of absolute symbols A1 and A2.
Definition of system macro-operations S1 and S2.

```
NSDEC   SDEC
A1      EQU     1
S1      MACRO
        GLOBAL  P1
        FINM
A2      EQU     2
S2      MACRO
        GLOBAL  P1, P2
        FINM
        FIN
```

User Macro-Operations
User macro S1 redefining the system macro S1.

```
S1      MACRO
        GLOBAL  P1
        FINM
```

User macro U2

```
U2      MACRO
        GLOBAL  P3
        FINM
```

Section CDS

with call to user macro S1

```
NCDS    CDS
A3      SET     3
        DATA    A3
        S1
        FIN
```

Program Section

Subsection LDS with call to system macro S2

```
NLDS    LDS
        RES     2
        S2
        FIN
```

Subsection LPS with call to user macro U2

```
NLPS    LPS     NLDS
        LDA     = 0
        U2
        FIN
        END
```

End Of Assembly Module

End Of File Mark (MFF)

IV-1.3. Scope Of Identifiers Defined In Sections And Subsections

Labels defined in the assembly module are distinguished from external assembly module labels declared with pseudo-instructions DEF, DEFX, REF, REFX.

- Internal Assembly Module Labels :

  . Labels defined in the SDEC.

  They are defined on the entire assembly module and can be referenced in any subsection ; as a result, they can be redefined without causing a "double definition" error.

  . Labels defined in the CDS

  They are defined on the entire assembly module and can be referenced in any subsection ; as a result, they cannot be redefined as local labels without causing a "double definition" error.

  . Labels defined in an LDS

  They are defined until a new LDS is encountered.

  They can be referenced from the CDS, in the LDS itself, and from any LPS following the LDS where they are defined.

  . Labels defined in an LPS

  They are defined until a new LDS or LPS is encountered.

  They can be referenced from the CDS, in the LPS itself, and from the associated LDS.

- External Assembly Module Labels :

  A label is said to be external to an assembly module when it is known outside the assembly module where it is defined.

  When the label is known only at Linkage Editing time, it is said to be external to the RB (Relocatable Binary) module.

  When the label is known only at group Generation time, it is said to be external to the RMI (Relocatable Memory Image) module.

  A label external to the RB module is declared by a DEF pseudo-instruction appearing in the section or subsection in which it is defined. It can be used in

another assembly module provided it is declared there by a REF pseudo-instruction.

A label external to the RMI module is declared by a DEFX pseudo-instruction appearing in the section or subsection in which it is defined. It can be used in another assembly module provided it is declared there by an REFX pseudo-instruction.

Pseudo-instructions DEF, DEFX, REF, REFX shall define the label before it is used.

Remark : Subsection names become implicit external references when they are used as such. When they are used as labels, their status depends on the segment to which they are attached (common in CDS, local in LDS or LPS).

If a symbol is referenced in a CDS, it cannot be defined in two different LDSs.

```
TOTO    CDS
        DATA    TATA
        FIN
LDS1    LDS
TATA    DATA    1
        FIN
LDS2    LDS
TATA    DATA    1
        FIN
```

The second definition of symbol TATA in LDS2 is a double definition.

If a symbol is defined in two different LPSs, these LPSs cannot be associated with the same LDS or CDS.

```
LDS1    LDS
        DATA    TATA
        FIN
LPS1    LPS     LDS1
TATA    LDA     = 1
        FIN
LPS2    LPS     LDS1
TATA    LDA     = 2
        FIN
```

The second definition of symbol TATA in LPS2 is a double definition.

IV-2. SECTION AND SUBSECTION ACCESS

SDEC is accessible to the entire program ; absolute symbols defined in the SDEC are common to the entire program.

Macros are accessible to the entire program located outside the SDEC.

Symbols and labels defined in the CDS are common to the entire program.

An LDS is accessible to the associated LPS. Symbols and labels in an LDS are local to the section. But they can be referenced in CDS.

V - INSTRUCTIONS CLASSES AND ADDRESSING TYPES

V-1. SYMBOLIC INSTRUCTION REPRESENTATION

V-1.1. Representation Conventions

The following representation conventions will be used hereafter :

A | B | C	one of the terms must appear and excludes the others.
[A]	the term between square brackets may not appear either because it is not mandatory, or because it is implicit.
...	possible repetition of the expression between the closure separator which immediately precedes the suspension points and the opening separator associated with it.

Examples :

A | B | C [,D]	one of the terms, A, B, or C, must be present. Term D is optional.
[A] | B | C	one of the terms, A, B, or C, must be present ; if A is the term, it may not be indicated.
A | D,B...,C	the term B can be repeated.

V-1.2. Source Instruction Representation

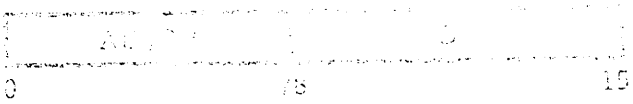
The following format conventions are used to write instructions :

LABEL	COMMAND	ARGUMENT
[Label]	Operation Code	[= | @ | $] D [,X] #

= : value of operand equals displacement
@ : indirection
$: extended addressing
,X : indexing
D : displacement
: expression relative to G

V-1.3. General Format Of Logical Instructions

IN



AD : Addressing mode
OP : Operation code
D : Displacement

Length of fields AD and OP depends on the instruction class.

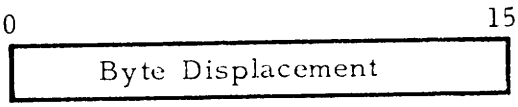
V-1.4. Indirect Addressing And Extended Addressing

MITRA 125 instructions use 2 types of indirect addressing :

- single indirect addressing called simply "indirect addressing" ;
- extended indirect addressing called simply "extended addressing".

V-1.4.1. Single Pointer

Indirect addressing uses a "single pointer" (or "indirect word") one word long and word-justified, which contains a byte address relative to G or Q depending on the mode (program or subroutine) and indirect addressing type (local or general).



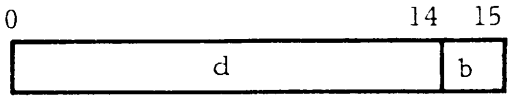
V-1.4.2. Logical Pointer

Extended addressing uses two types of "logical pointers" :

- logical word pointer used by the instructions on a word ;
- logical byte pointer used by the instructions on a byte.

Logical Word Pointer (see figure 5-1.)

A logical word pointer consists of a word-justified word containing a "generalized address" :



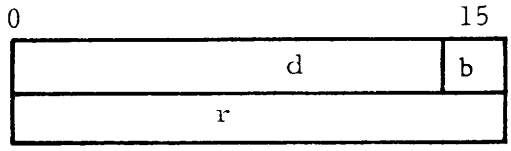
d_r = d, 0 is a word displacement relative to a segment base B.

b designates the base to which d_r is relative :

if b = 0, B = G
if b = 1, B = Z

Logical Byte Pointer (see figure 5-2.)

A logical byte pointer consists of two consecutive word-justified words containing a "generalized byte address" :



d_r = d, 0+r is a byte displacement relative to a segment base B.

b designates the base to which d_r is relative :

if b = 0, B = G
if b = 1, B = Z

For the general case, d and r have any values.

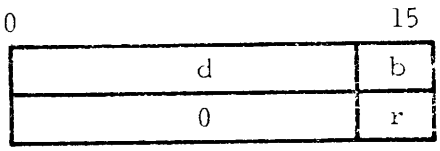
Two special cases frequently used are :

1. "normalized" logical byte pointer :

d = any value

$r_{0-14} = 0$

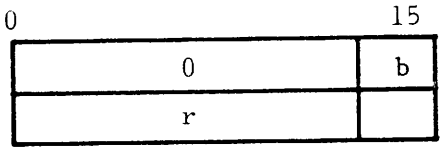
r_{15} = 0 or 1



2. "extended" logical byte pointer :

d = 0

r = any value



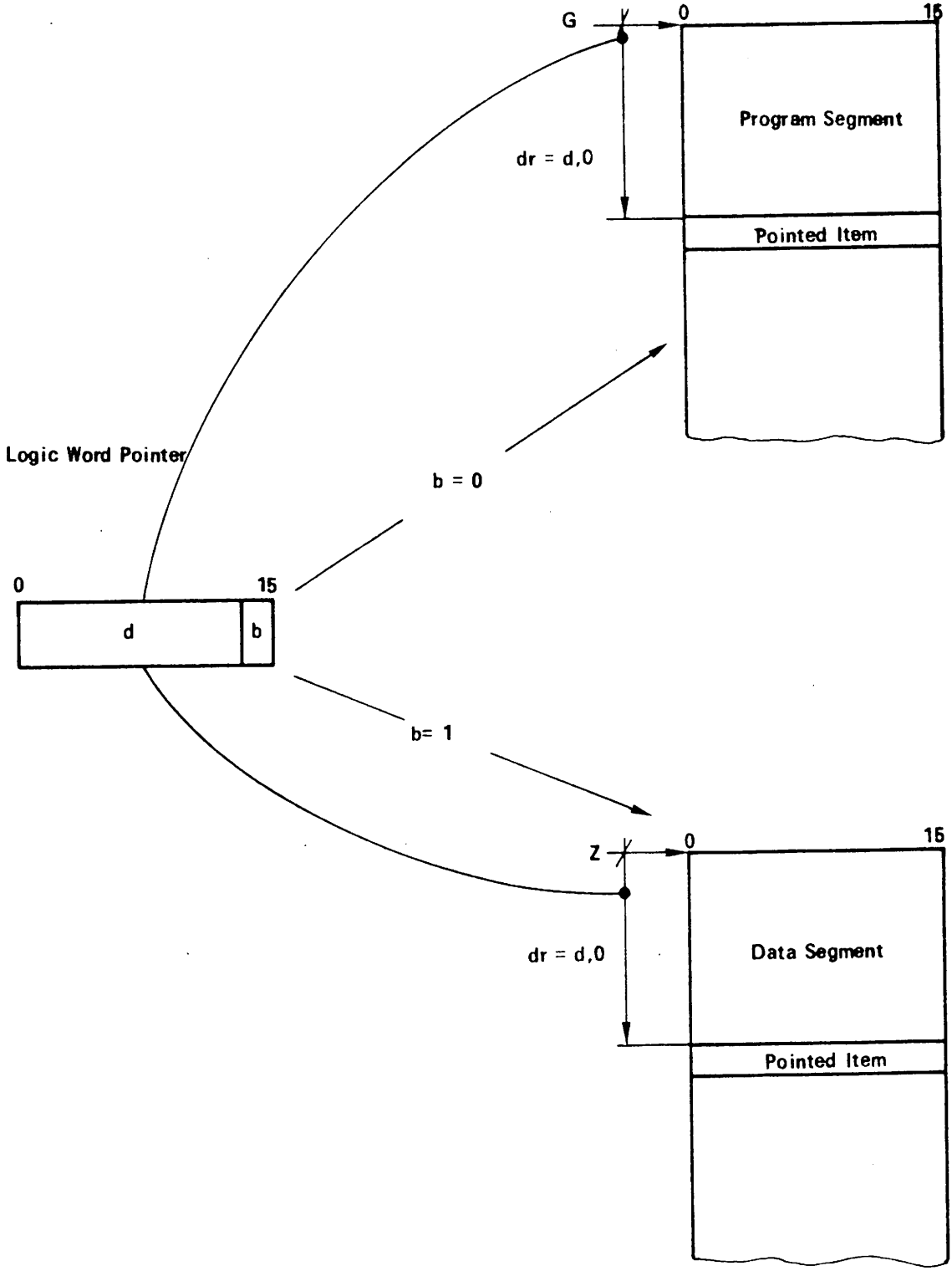


Fig. 5.1 - LOGICAL WORD POINTER TOPOLOGY

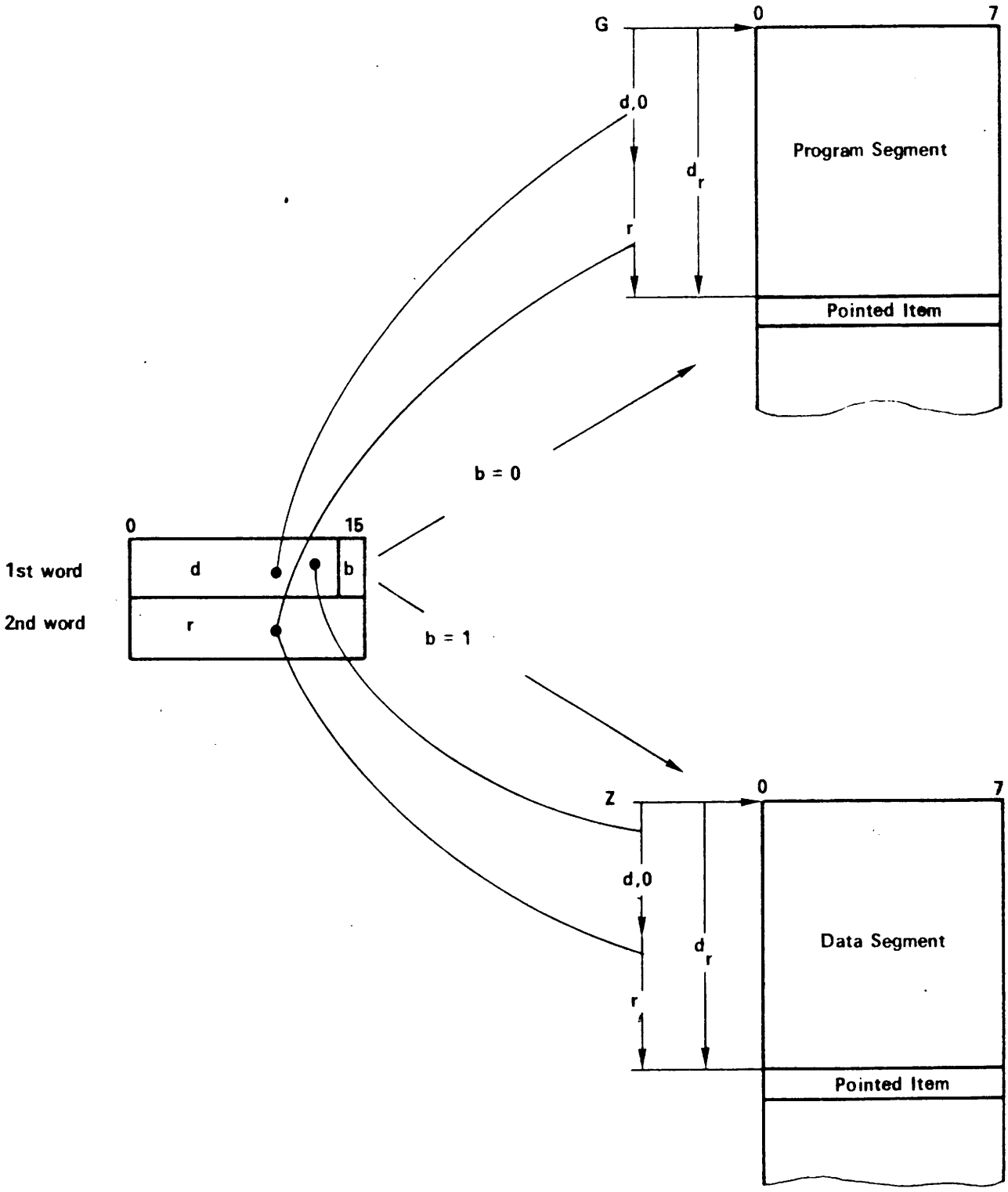


Figure 5.2 - LOGICAL BYTE POINTER TOPOLOGY

V-2. ADDRESSING REPRESENTATION

The addressing type indicates in what way the displacement will be processed at the time of instruction execution.

V-2.1. Instruction Classes

MITRA 125 instruction addressing possibilities depend on the instruction class to which the instruction belongs.

There are five instruction classes : classes 0, 0', 1, 1', and 2. They are distinguished by the 4 most significant bits of the instruction.

IN 0-1 \ IN 2-3	00	01	10	11
0 0	0	0'	0	1 and 1'
0 1	0	0'	0	0'
1 0	0	0'	0	0'
1 1	2	2	1 and 1'	1 and 1'

V-2.2. Addressing Different Types Of Instructions

Addressings are represented by the following conventions :

G : Program Segment Base
Q : Subroutine Segment Base
G' : Base G if "program mode" (SP = 0)
Base Q if "subroutine mode (SP = 1)
Z : Data Segment Base

B : Base G if the logical pointer referenced is relative to G
Base Z if the logical pointer referenced is relative to Z

P : Contents of register P
L : Contents of register L
X : Contents of register X
D : Value of displacement contained in the instruction
Y : Computed address of the operand relative to the addressed segment base (G, Q, Z)
N : Value of computed operand
IN : Instruction

Syntax of addressing functions :

(I) : Contents of I
I/J : Address I is relative to base J
(I)/j : Contents of memory cell of address I relative to base J
$(I)_{0-14/J,0}$: 16-bit word resulting from the concatenation of bits 0 through 14 of (I)/J and a least significant bit equal to 0.

V-3. CLASS 0 INSTRUCTIONS

V-3.1. Definition Of Class 0 Instructions

There are sixteen class 0 instructions. They are distinguished by bits 4, 5, 6, and 7 of the instruction.

LDA	LoaD register A
LDE	LoaD register E
LDX	LoaD register X
EOR	Exclusive OR
LEA	LoaD Effective Address
ADD	ADDition
SUB	SUBtraction
IOR	Inclusive OR
DIV	DIVision algebraic
AND	AND operation
CPS	ComPare byte to String
CMP	CoMPare algebraic

MUL	MULtiplication algebraic
LBL	Load Byte Left of register A
LBR	Load Byte Right of register A
LBX	Load Byte right of register X

IN 4-5 \ IN 6-7	00	01	10	11
0 0	LDA	LDE	LDX	EOR
0 1	LEA	ADD	SUB	IOR
1 0	DIV	AND	CPS	CMP
1 1	MUL	LBL	LBR	LBX

V-3.2. Addressing Types Of Class 0 Instructions

Class 0 instructions have all 9 types of addressing :

Coding IN 0 1 2 15	Type	Symbolic Representation	Addressing Function
0 0 0 X	Direct Local DL	IDENT	Y/G' = L+D
0 1 1 0	Indirect local IL	@ IDENT	Y/G' = (L+D)/G'
0 1 1 1	Extended local EL	$ IDENT	Y/B = $(L+D)_{0-14}$/G',0+r
1 0 1 0	Indirect local Indexed ILX	@ IDENT,X	Y/G' = (L+D)/G'+X
1 0 1 1	Extended local Indexed ELX	$ IDENT,X	Y/B = $(L+D)_{0-14}$/G',0+X+r
0 1 0 X	Direct general DG	[#] IDENT	Y/G = D
1 0 0 0	Indirect General Indexed IGX	@ [#] IDENT,X	Y/G = (D)/G+X
1 0 0 1	Extended General Indexed EGX	$ [#] IDENT.X	Y/B = $(D)_{0-14}$/G; 0+X+r
0 0 1 X	Parameter P	= PARAM	N = D

For all byte instructions (LEA, LBL, LBR, LBX, SBL, SBR, CDS, MVS, TRS), r is the contents of the second word of the referenced logical byte pointer.

$r = (L + D + 2)/G'$ in EL and ELX mode

$r = (D + 2)/G$

For other instructions, $r = 0$

Note 1 : For addressings of type IL, EL, ILX, and ELX, L must be a word address ($L_{15} = 0$) ; otherwise, the addressing type specified will be inverted :

@ and $L_{15} = 1 \equiv \$$

$ and $L_{15} = 1 \equiv @$

Note 2 : Previous addressing functions show that :

A subroutine segment instruction can have access to data belonging to :

- the program segment / G
- the subroutine segment / Q
- the data segment / Z

A program segment instruction can have access to data belonging to :

- the program segment / G
- the data segment / Z

It cannot have access to data of the subroutine segment.

Class 0 Addressing Examples : (see figure 5-3)

1) Addressings DG, DL, IL, IGX, ILX, P

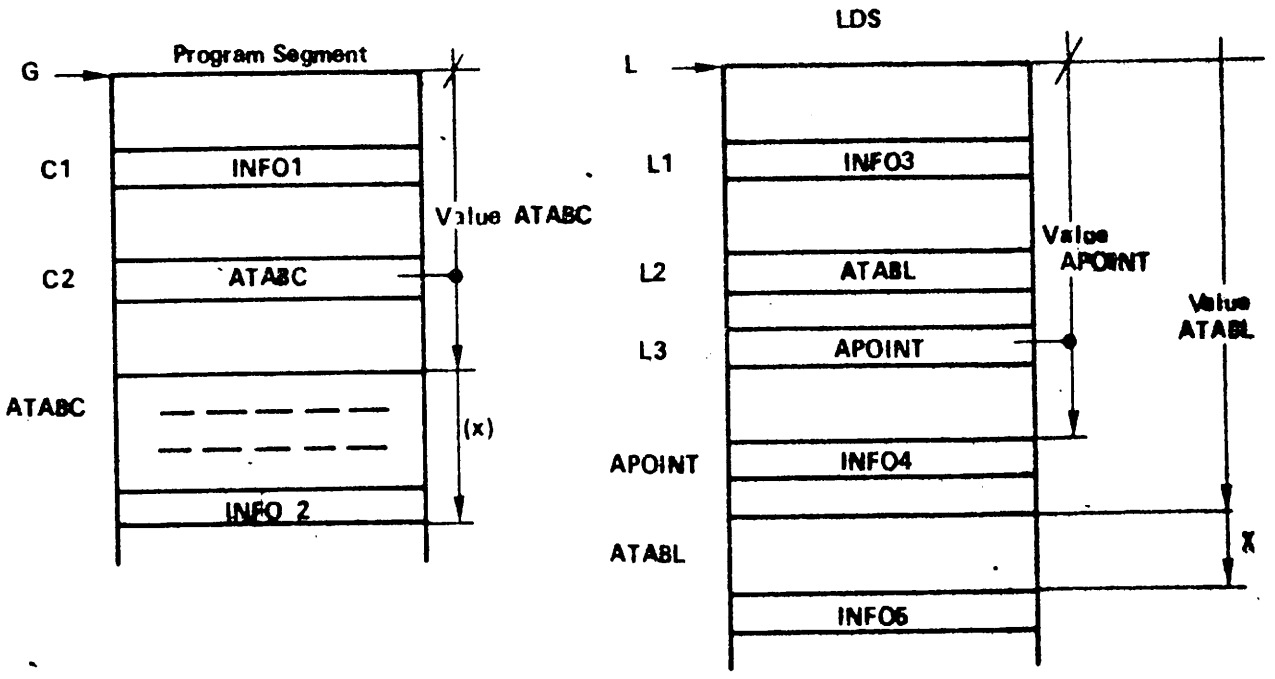


Figure 5.3

Type	Instruction	Referenced Data
DG	LDA C1	INFO1
DL	LDA L1	INFO3
IL	LDA @L3	INFO4
IGX	LDA @C2,X	INFO2
ILX	LDA @L2,X	INFO5
P	LDA = INFO6	INFO6

The local subsection may belong to the program segment, or to the subroutine segment as well as the LPS to which the executed instruction belongs.

2) Addressings EL, ELX, EGX (see figure 5-4.)

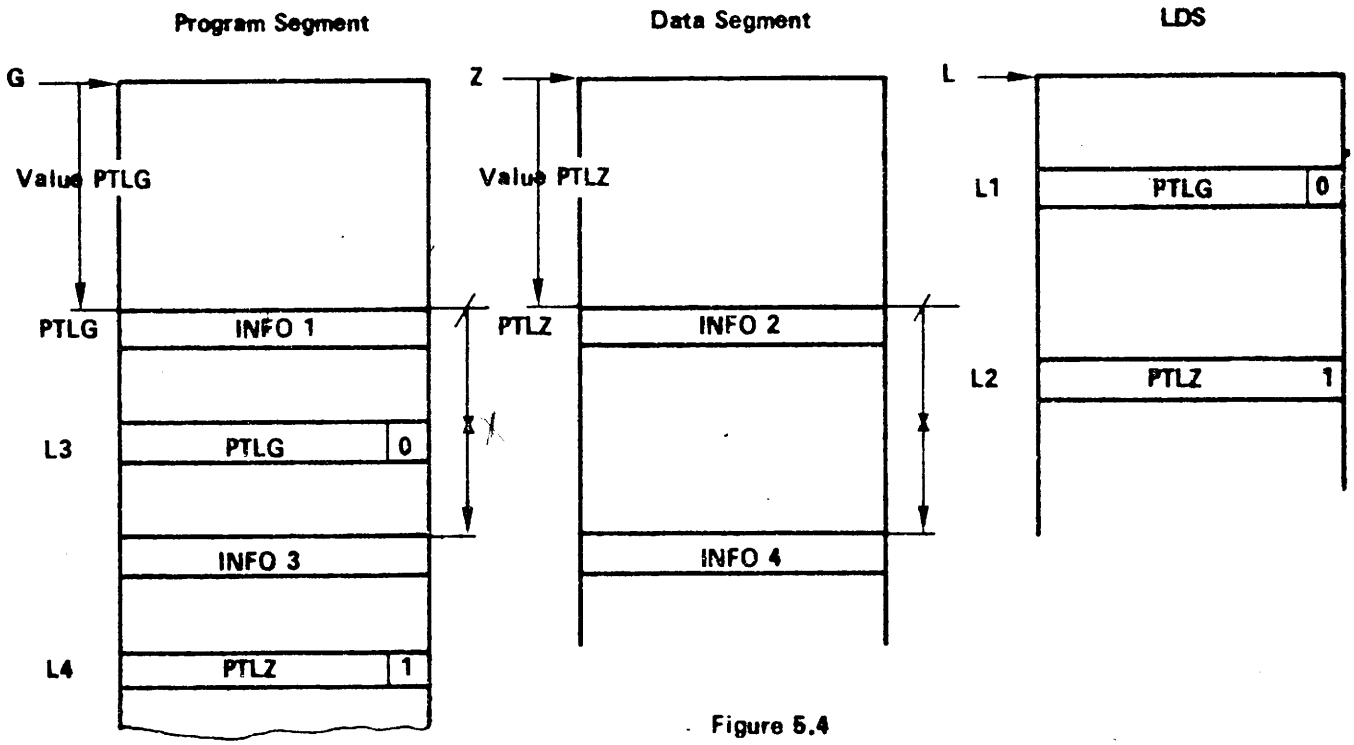


Figure 5.4

Type	Instruction	Referenced Data
EL	LDA $L1	INFO1
EL	LDA $L2	INFO2
ELX	LDA $L1,X	INFO3
ELX	LDA $L2,X	INFO4
EGX	LDA $L3,X	INFO3
EGX	LDA $L4,X	INFO4

The local subsection may belong to the program segment, or to the subroutine segment as well as the LPS to which the executed instruction belongs.

V-4. CLASS 0' INSTRUCTIONS

V-4.1. Definition Of Class 0' Instructions

There are 16 class 0' instructions. They are distinguished by bits 4, 5, 6, and 7 of the instruction.

DLD	LoaD A, E extended register
STA	STore register A
STE	STore register E
STX	STore register X
SBL	Store Byte Left of register A
SBR	Store Byte Right of register A
DST	STore E, A extended register
ADM	ADdition Memory
SPA	Store Program Address
STS	STore Selective register A
FAD	Floating ADdition, single length
FSU	Floating SUbtraction, single length
FMU	Floating MUltiplication, single length
FDV	Floating DiVision, single length
TRS	TRanscode byte String
MVS	MoVe byte String

IN 4-5 \ IN 6-7	00	01	10	11
0 0	DLD	STA	STE	STX
0 1	SBL	SBR	DST	ADM
1 0	SPA	STS	FAD	FSU
1 1	FMU	FDV	TRS	MVS

IV-4.2. Addressing Types Of Class 0' Instructions

Class 0' instructions have the same addressing types as class 0 instructions except for the parameter mode which does not exist.

V-5. CLASS 1 INSTRUCTIONS

V-5.1. Definition Of Class 1 Instructions

There are 9 class 1 instructions. They are distinguished by bits 4, 5, 6, and 7 of the instruction.

ICX	InCrement register X
DCX	DeCrement register X
ICL	InCrement base L
DCL	DeCrement base L
CSV	Call SuperVisor
CLS	CalL Section of program segment
LDR	LoaD Register
TES	TEst and Set
CLQ	CalL section of subroutine segment

IN 4-5 \ IN 6-7	00	01	10	11
0 0			ICX	DCX
0 1		ICL	DCL	CSV
1 0	CLS	LDR		
1 1		TES	CLQ	

V-5.2. Addressing Mode Of Class 1 Instructions

Coding IN 0/1/2/3	Type	Symbolic Representation	Addressing Function
0 1 1 1	Direct Local DL	IDENT	Y/G' = L + D N = (Y)/G'
1 1 1 1	Parameter P	= PARAM	N = D
1 1 1 0	Indexed Parameter	= PARAM,X	N = D + X

Class 1 instructions have 3 addressing types.

Examples :

DCX = 3	Decrement X by 3
ICX = 0,X	X + X
DCL IDENT	with IDENT DATA 3
	→ DCL = 3
DCX = 4,X	load X with value - 4

V-6. CLASS 1' INSTRUCTIONS

V-6.1. Definition Of Class 1' Instructions

Class 1' has 5 instruction families. They are distinguished by bits 4, 5, 6, and 7 of the instruction and have special addressing types which depend on the type 1 family from which they come, viz. :

Family	SHR
Family	SRG
Family	MCB
Family	SHC
Family	COV

IN 6-7 IN 4-5	00	01	10	11
0 0	SHR	SRG		
0 1				
1 0				MCB
1 1	SHC			COV

V-6.2. Family SHR

Instructions of family SHR are all shift instructions. They are distinguished by bits 8-10 of the operand computed according to class 1 rules. Bits 11-15 of the computed operand are used to designate the number of shifts :

SLLS	Shift Left, Logical, of register A
SRCS	Shift Right, Circular, of register A
SAD	Shift right, Arithmetic, of registers E,A
SLCD	Shift Left, Circular, of registers E,A
SLCS	Shift Left, Circular, of register A
SAS	Shift right, Arithmetic, of register A
SRLS	Shift Right, Logical, of register A
SRCD	Shift Right, Circular, of registers E,A

N 9-10 / N8	00	01	10	11
0	SLLS	SRCS	SAD	SLCD
1	SLCS	SAS	SRLS	SRCD

V-6.3. Family SRG

Instructions of family SRG are distinguished by bits 10-14 of the operand computed according to class 1 rules. Instructions of this family are instructions without operand.

RTS	Return To user Section
RTQ	Return To shared program section
XAE	eXchange contents of registers A and E
XAX	eXchange contents of registers A and X
XEX	eXchange contents of registers E and X
XAA	eXchange left byte of register A and right byte of register A
CCE	Copy Complement of register E
ACE	Add Carry in register E
CCA	Copy Complement of register A

AEE	register A Exclusive-OR with register E in register A
CNX	Copy Negative of register X
AIE	register A Inclusive-OR with register E in register A
AAE	register A AND with register E in register A
LNE	Load Negative in register E
CNA	Copy Negative in register A
CHX	Compute Half of register X
NBP	Normalize logical Byte Pointer
AEA	add registers A and E in register A
ICE	InCrement register E by 1
DCE	DeCrement register E by 1
AAX	Add registers A and X in register X
AXA	add registers A and X in register A
SAX	Subtract register A from register X
TSX	TeSt register X
STI	STore register A in Indicators
LDI	LoaD register A with Indicators
CAE	Compare arithmetic registers A and E
CNE	Copy Negative of register E
CAA	Compare left byte of register A with right byte of register A
SRP	Save and Reset Pointer (A_{15})
HLT	HaLT

N 10-11-12 \ N 13-14	00	01	10	11
0 0 0	RTS RTQ	XAE	XAX	XEX
0 0 1	XAA	CCE		ACE
0 1 0	CCA	AEE	CNX	AIE
0 1 1	AAE	LNE	CNA	CHX
1 0 0	NBP	AEA	ICE	DCE
1 0 1		AAX	AXA	SAX
1 1 0	TSX	STI	LDI	CAE
1 1 1	CNE	CAA	SRP	HLT

V-6.4. Family MCB

Instructions of family MCB are distinguished by bits 0-3 and 8 of the instruction. Bit 9 defines the addressing mode (refer to Instruction Description).

IN 8 \ IN 0-3	0011
0	PUSH
1	PULL

PUSH = Stack In ; PULL = Stack Out

V-6.5. Family SHC

Instructions of family SHC are distinguished by bits 5-11 of the operand computed according to class 1 rules. Bits 11-15 of the computed operand are used as an operand by the instruction.

N 0-9 \ N 10-11	00	01	10	11
0 0	SLLD		CBE	CBA
0 1	PTY	CMZ	RBE	RBA
1 0	SRLD		TBE	TBA
1 1	NLZ		SBE	SBA

SLLD	Shift Left, Logical, of registers E and A
CBE	Change status of Bit k of register E
CBA	Change status of Bit k of register A
PTY	Compute PariTY
CMZ	Count Most significant Zeros of register A
RBE	test and Reset Bit k of register E
RBA	test and Reset Bit k of register A

SRLD	Shift Right, Logical, of registers E,A
TBE	Test Bit k of register E
TBA	Test Bit k of register A
NLZ	NormaLiZe registers E,A
SBE	test and Set Bit k of register E
SBA	test and Set Bit k of register A

V-6.6. Family COV

Instructions of family COV are distinguished by bits 6-15 of the operand computed according to class 1 rules. This instruction family permits creating, in particular, new instructions.

Four instructions executable in User mode of this family currently exist ; these are double-precision floating point instructions.

N_{8-11} = 0011

N_{12-15} =		
0000	FMUD	Floating Multiplication Double precision
0001	FDVD	Floating Division Double precision
0010	FADD	Floating AdD Double precision
0011	FSUD	Floating SUbtraction Double precision

Double-precision floating point instructions are quadruple-word instructions.

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15		
Operation Code		
IN 1	00000	D1
IN2	00000	D2
IN3	00000	D3

The first word defines the operation code.
The next three words define the address of operand 1 (IN1, D1), operand 2 (IN2, D2), and the result (IN3, D3).

Each pair (IN, D) defines 8 addressing modes : DL, DG, IL, EL, ILX, ELX, IGX, EGX.
Operands are computed according to class 0' rules.

V-7. CLASS 2 INSTRUCTIONS

V-7.1. Definition Of Class 2 Instructions

There are eight class-2 instructions. They are distinguished by bits 5, 6, and 7 of the instruction.

BCT	Branch if Carry True (BE, BZ)
BRX	BRanch indeXed
BOT	Branch if Overflow True (BL, BLZ)
BCF	Branch if Carry False (BNE, BNZ)
BAN	Branch if register A Negative
BAZ	Branch if register A at Zero
BOF	Branch if Overflow False (BGE, BPZ)
BRU	Branch Unconditional

IN 5 \ IN-6-7	00	01	10	11
0	BCT	BRX	BOT	BCF
1	BAN	BAZ	BOF	BRU

Remark : An indicator is said to be "True" if it is set to 1.
An indicator is said to be "False" if it is set to 0.

V-7.2. Addressing Types Of Class 2 Instructions (see figure 5-5.)

IN0-4	Type	Symbolic Representation	Addressing Function
11000	Relative Plus RP	LABEL	Y/G' = P + 2D if BRX : Y/G' = P + 2D + 2X
11001	Relative Minus RM	LABEL	Y/G' = P - 2D if BRX : Y/G' = P - 2D - 2X
11011	Indirect General IG	@[#]LABEL	Y/G' = (D)/G if BRX : Y/G' = (D + X)/G
11010	Indirect Local IL	@ LABEL	Y/G' = (L + D)/G' if BRX : Y/G' = (L+D+X)/G'

Instructions designated by a sequence break instruction normally belong to the same program section as this break instruction. Program sections can, however, have any length.

In addition, there is an indexed, unconditional branch instruction. If the branch is indirect, indexing is a pre-indexing (better for processing branch tables) as compared to data indexing which is a post-indexing (better for accessing table elements).

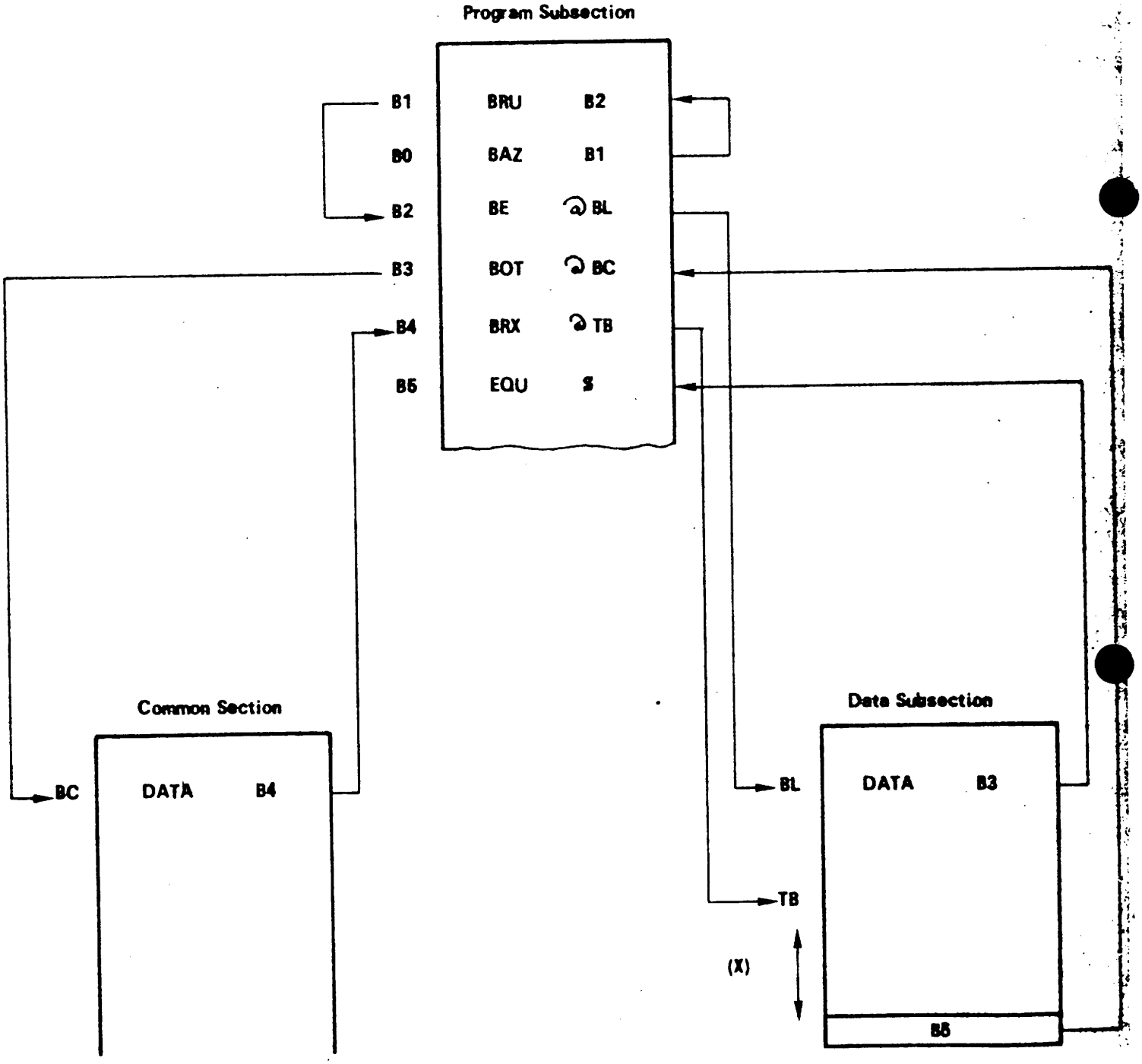


Fig. 5.5 - CLASS 2 ADDRESSING MODE

VI - PSEUDO-INSTRUCTIONS

VI-1. GENERAL

Pseudo-instructions are divided into sectioning pseudo-instructions and assembly pseudo-instructions.

VI-2. SECTIONING PSEUDO-INSTRUCTIONS

Sectioning pseudo-instructions are used to define the assembly module structure into sections and subsections. They are :

- common declaration pseudo-instructions : SDEC/FIN.
- common data section declaration pseudo-instructions : CDS/FIN.
- declaration pseudo-instructions of data subsections describing COMMON FORTRAN : COMS/FINC.
- local data subsection declaration pseudo-instructions : LDS/FIN.
- indirect data subsection declaration pseudo-instructions : IDS/FIN.
- executable program subsection declaration pseudo-instructions : LPS/FIN, and LPSP/FIN.
- entry point declaration pseudo-instruction for entry into an executable program subsection : ENTRY.
- "end of assembly module" pseudo-instruction : END.

VI-2.1. Pseudo-Instructions SDEC/FIN

Function :

Pseudo-instruction SDEC identifies the common declaration section.

Format :

Label	Command	Argument
< Name >	SDEC	
[< Label >]	FIN	

Result

Absolute symbols declared in section SDEC can be referenced in all the module sections.

No code is generated.

System macro declarations must be in section SDEC.

Section SDEC must be terminated by a FIN pseudo-instruction.

VI-2.2. Pseudo-Instruction CDS/FIN

Function :

Pseudo-instruction CDS identifies the common data section.

Format

Label	Command	Argument
< Name > [< Label >]	CDS : FIN	[DUM]

Result :

Location counter is reset to zero.

All labels defined in this section can be referenced in all the module sections.

If option DUM is specified in the argument field of pseudo-instruction CDS, no code is generated ; the section is dummy.

Section CDS must be terminated by a FIN pseudo-instruction.

VI-2.3. Pseudo-Instructions COMS/FINC

Function

Pseudo-instruction COMS identifies a data subsection describing a COMMON FORTRAN.

Format :

Label	Command	Argument
<COMMON name> [<label>]	COMS FINC	[DUM]

Result :

The location counter is reset to zero.

If option DUM is specified in the argument field of pseudo-instruction COMS, no code is generated.

The user can describe a COMMON FORTRAN in a COMS subsection using assembly pseudo-instructions DATA, DATAF, RES, ...

Subsection COMS must be terminated by a FINC pseudo-instruction.

Remark : If the main program is written in FORTRAN and the subroutines are written in assembly language, the user describes the COMMON FORTRAN in a dummy data subsection (COMS DUM/FINC) and he creates in the LDS of his subroutine a pointer to the COMMON FORTRAN using pseudo-instruction COMMON.

If the main program is written in assembly language and the subroutines in FORTRAN, the user, at the end of his direct CDS, creates a pointer to the COMMON FORTRAN using pseudo-instruction COMMON and he describes the COMMON FORTRAN in a real data subsection (COMS/FINC).

VI-2.4. Pseudo-Instructions LDS/FIN

Function

Pseudo-instruction LDS identifies a local data subsection.

Format

Label	Command	Argument
< Name >	LDS [,16]	[DUM]
[< Label >]	FIN	

Result

The location counter is reset to zero.

< name > defined in the label field of the pseudo-instruction LDS is an implicit external definition.

If option DUM is specified in the argument field of pseudo-instruction LDS, no code is generated : the subsection is dummy.

If option [,16] is specified in the command field, the linkage editor will place the subsection at an octo-word address.

Subsection LDS must be terminated by a FIN pseudo-instruction.

VI-2.5. Pseudo-Instructions IDS/FIN

Function

Pseudo-instruction IDS identifies an indirect access data subsection within a CDS or an LDS such that any label appearing between pseudo-instruction IDS and the associated pseudo-instruction FIN is defined relatively to the beginning of the declared indirect subsection.

This is a number type and no longer an address type relative value.

Format :

Label	Command	Argument
< Name >	IDS	[DUM]
[< Label >]	FIN	

Result

The value of the location counter is saved.

The location counter is reset to zero.

After the IDS subsection is closed by pseudo-instruction FIN, the initial value of the location counter is restored with its current relative value added or not depending on whether the IDS subsection is effective or not.

If option DUM is specified in the argument field of the pseudo-instruction, no code is generated ; the subsection is dummy.

The label defined in the label field of pseudo-instruction IDS is considered as a normal CDS or LDS label if subsection IDS is real ; if IDS is dummy, it assumes value zero.

An IDS subsection declared within a dummy CDS section or a dummy LDS section becomes dummy (no code is generated), but the initial value of the location counter is restored with its current relative value added.

Subsection IDS must be terminated by a FIN pseudo-instruction.

VI-2.6. Pseudo-Instructions LPS/FIN

Function

Pseudo-instruction LPS identifies an executable program subsection and, therefore, a program section.

Format

Label	Command	Argument
< Name 1 >	LPS [,16]	< Name 2 >
[< Label 1 >]	FIN	< Label 2 >

Result :

The location counter is reset to zero.

< Name 1 > is the LPS name (and the program section name) ; this is an implicit external definition. The name referenced in a CALL SECTION will be the corresponding implicit external reference.

< Name 2 > indicates the associated LDS name.

< Label 2 > defines the effective execution start address of the section.

If option [,16] is specified, the linkage editor will place the subsection at an octoword address.

Subsection LPS must be terminated by a FIN pseudo-instruction.

VI-2.7. Pseudo-Instruction LPSP

Function

Pseudo-instruction LPSP identifies an executable program subsection whose associated LDS is located in the resident program or in the calling branch.

Format

Label	Command	Argument
[< Name 1 >]	LPSP	[< Name 2 >]

Result

< Name 1 > is the LPSP name.

< Name 2 > indicates the name of the associated LDS.

If the name of the associated LDS is the same as that of the last declared LDS, processing is identical to that of LPS.

If the name of the associated LDS is different from that of the last declared LDS, no error is reported, and the LPSP is processed like an LPS without local data section.

VI-2.8. Pseudo-Instruction ENTRY

Function

Pseudo-instruction ENTRY is used to create an entry point into an executable program subsection.

Format

Label	Command	Argument
< Name 1 >	ENTRY	

Result

The location counter is reset to zero on the listing.

< Name 1 > is the ENTRY name which is stored in the subsection table and the symbol table.

< Name 1 > can be called by CLS as an executable program subsection name.

Symbols defined after an ENTRY pseudo-instruction are relative to the beginning of the executable program subsection.

Remark : Pseudo-instruction FIN of the executable program subsection should not have a section start address.

VI-2.9. Pseudo-Instruction END

Function

Pseudo-instruction END indicates the end of the assembly module.

Format

Label	Command	Argument
[< Label >]	END	[< Name >]

Result

Module assembly is terminated.

< Name > indicated in the argument field of pseudo-instruction END defines the name of the section to be initiated after the program is loaded.

Source lines, possibly located between pseudo-instruction END and the End Of File Mark (MFF) are not processed but cause an error.

VI-3. ASSEMBLY PSEUDO-INSTRUCTIONS

Assembly pseudo-instructions define the processing to be performed by the assembler. They are :

- Addressing pseudo-instructions : RES, BND, BASE.
- Symbol definition pseudo-instructions : EQU, SET, STATUS
- Assembly-control pseudo-instructions : GOTO, JMP, DO, DUP, FIND, RMT, RESET, FINR, HERE, ERROR.
- Data generation pseudo-instructions : DATA, DATAF, DATAB, TEXT, TEXTC, TEXTA, GEN, GENOV, PTW, GENCLS, PTB.
- External definition pseudo-instructions : DEF, DEFX, REF, REFX.
- Listing formatting pseudo-instructions : NOLIST, LIST, TITLE, PAGE, SPACE.
- Library definition pseudo-instructions : EXT, XREF.
- COMMON FORTRAN pseudo-instructions : COMMON

Assembly pseudo-instructions can be used for assembling an SDEC, CDS, LDS, IDS, LPS ; they are distributed between these sections or subsections as indicated by the table on the next page.

PSEUDO-INSTRUCTION	SDEC	CDS-LDS-IDS	LPS	COMS
RES		X	X	X
BND		X	X	X
BASE			X	
EQU	X	X	X	X
SET		X	X	X
STATUS		X	X	X
GOTO		X	X	X
JMP		X	X	X
DO		X	X	X
DUP/FIND		X	X	X
RMT/FINR /HERE		X	X	
RESET		X	X	
ERROR		X	X	
DATA		X		X
DATAB		X		
DATAF		X		X
TEXT		X		X
TEXTC		X		X
TEXTA		X		X
GEN		X	X	
GENOV		X	X	
GENCLS		X	X	
PTW		X		
PTB		X		
DEF/DEFX		X	X	
REF/REFX		X	X	
NOLIST	X	X	X	X
LIST	X	X	X	X
TITLE	X	X	X	X
PAGE	X	X	X	X
SPACE	X	X	X	X
EXT		X	X	
XREF		X	X	
COMMON		X		

VI-3.1. Addressing Pseudo-Instructions

VI-3.1.1. Pseudo-Instruction RES

Function

Reserve a memory area

Format

Label	Command	Argument
[< Label >]	RES [,1]	< Expression >

Result

If option [,1] is specified in the command field, the location unit is the byte ; otherwise, it is the word.

The location counter is first advanced to a word address if the location unit is the word.

The value assigned to the symbol defined in the label field is the first address of the reserved area.

VI-3.1.2. Pseudo-Instruction BND

Function

Advance location counter to a word address .

Format

Label	Command	Argument
[< Label >]	BND	[16]

Result

If option [16] is missing in the argument field, the location counter is advanced up to a word

If option [16] is specified in the argument field, the location counter is advanced up to a multiple-of-16 word address.

No value is assigned to the symbol defined in the label field.

VI-3.1.3. Pseudo-Instruction BASE

Function

Relative addressing check.

Format

Label	Command	Argument
[< Label 1 >]	BASE	[< Label 2 >]

Result

< Label 2 > declared in the argument field is a label defined in the local data subsection associated with the program subsection.

All LDS addresses referenced in the LPS between two BASE pseudo-instructions are generated with a value relative to the address defined by < Label 2 >.

A BASE pseudo-instruction without declaration of < Label 2 > in the argument field simply closes the relative addressing opened by the previous BASE pseudo-instruction. To open another relative addressing, it will be necessary to declare another BASE pseudo-instruction with a < label 2 > declaration in the argument field.

A BASE pseudo-instruction with a < Label 2 > declaration in the argument field closes the relative addressing of the previous BASE pseudo-instruction if it exists and opens a relative addressing on the new < Label 2 > declared in the argument field.

Relative addressing is closed either by a new BASE pseudo-instruction or by pseudo-instruction FIN which terminates program subsection assembly.

VI-3.2. Symbol Definition Pseudo-Instructions

VI-3.2.1 Pseudo-Instruction EQU

Function

Symbol definition without redefinition possibility.

Format

Label	Command	Argument
< Name >	EQU	< Expression >

Result

The expression which appears in the argument field defines the value and type of the symbol declared in the label field.

Symbol $ which represents the value of the location counter can appear in the argument field.

The symbol which appears in the label field cannot be redefined.

Remark

Symbols appearing in <expression > must be either absolute, i.e., independent from any base, or defined in the same subsection :

Example 1

```
LDS1    LDS
A       DATA    1
B       DATA    2
C       EQU     B - A
        FIN
```

Expression B - A is correct because symbols B and A belong to the same subsection LDS1.

Example 2

```
LDS1        LDS
A           DATA    1
            FIN
LPS1        LPS     LDS1
B           LDA     = 1
C           EQU     B - A
            FIN
```

Expression B - A is not correct because symbols B and A do not belong to the same subsection.

VI-3.2.2. Pseudo-Instruction SET

Function

Definition of a symbol with a redefinition possibility.

Format

Label	Command	Argument
< Name >	SET	< Expression >

Result

The expression which appears in the argument field defines the value and type of the symbol declared in the label field.

Symbol $ which represents the location counter value can appear in the argument field.

A symbol which appeared in the label field of a SET pseudo-instruction can be redefined by another SET pseudo-instruction.

VI-3.2.3. Pseudo-Instruction STATUS

Function

Define current status of a symbol.

Format

Label	Command	Argument
< Label >	STATUS	< Symbol >

Result

< Label > is stored in the symbol table (EQU type symbol) with as a value :

&FFFF	if < Symbol > is not defined.
0	if < Symbol > is defined as an absolute value
1	if < Symbol > is defined in CDS
2	if < Symbol > is defined in LDS
3	if < Symbol > is defined in LPS

Before a symbol is used, a test can be made to find out if it is defined. If the symbol is not defined, it can then be defined before it is used.

Example

```
                    ¦
                    ¦
L       STATUS      A
        JMP,LAB     L.NE.&FFF
A       EQU
LAB
                    ¦
                    ¦
```

Symbol A will therefore always be defined when line LAB is reached.

VI-3.3. Assembly Check Pseudo-Instructions

VI-3.3.1. Pseudo-Instruction GOTO

Function

Conditional assembly branch.

Format

Label	Command	Argument
[< Label >]	GOTO,k	< Label 1 >[, < Label 2 >] ...

Result

The assembler resumes assembly at the source line which contains in the label field the kth label declared in the argument field of the pseudo-instruction.

k is an absolute expression.

Labels declared in the argument field must be referenced to sources lines which follow the GOTO pseudo-instruction.

If k = 0, assembly resumes at the source line which follows the pseudo-instruction.

If k is not between 0 and n (n = number of labels in the argument field), or if the expression defining the value of k is not correct, an error message is printed out and assembly resumes at the source line which follows the pseudo-instruction.

If one of the following pseudo-instructions : FIN, FINM, FIND, or END appear before the searched for label, assembly is interrupted and an error message is printed out.

To facilitate special GOTO applications, source lines that contain only one label can be referenced. This label is never considered as defined ; it will therefore not have lateron a double definition.

VI-3.3.2. Pseudo-Instruction JMP

Function

Conditional assembly branch.

Format

Label	Command	Argument
[< Label 1 >]	JMP,< Label 2 >	<Expl > < code >< Exp2 >

Result

The assembler performs between expressions < Expl> and < Exp2> defined in the argument field a comparison of the type specified by < code> .

If the condition is met, assembly resumes at the source line which contains in its label field < label 2 > declared in the command field of pseudo-instruction JMP.

<Code> is one of the following condition operator :

EQ	(equal)
NE	(non equal)
GT	(Greater)
GE	(greater or equal)
LT	(less than)
LE	(less than or equal)

VI-3.3.3. Pseudo-Instruction DO

Function

Assembly iteration of an instruction .

Format

Label	Command	Argument
[< Label >]	DO	< Expression >

Result

The expression declared in the argument field supplies an integer number k less than or equal to 2^{16} - 1 which represents the number of times that the next source line must be assembled.

The iteration index is symbolically represented by special character % and can be used in the source line to be assembled. Whenever an iteration is executed, it assumes the iteration counter value.

The iteration counter begins at value 1 for the first loop.

When the iteration is terminated, assembly normally resumes at the second source line which follows the pseudo-instruction.

If the expression declared in the argument field is zero or if the expression defining the value of k is incorrect, assembly resumes directly at the second source line which follows the pseudo-instruction.

If a label appears in the label field, it is associated with the first byte generated.

VI-3.3.4. Pseudo-Instruction DUP/FIND

Function

Assembly iteration of an instruction block.

Format

Label	Command	Argument
[<Label >]	DUP	< Expression >
[< Label >]	FIND	

Result

The expression declared in the argument field supplies an integer number k less than or equal to $2^{16} - 1$ which represents the number of times that the following source lines between pseudo-instructions DUP and FIND must be assembled.

The iteration index is symbolically represented by the special character % and can be used in the source lines to be assembled. At each iteration, it assumes the value of the iteration counter.

The iteration counter begins with value 1 for the first loop.

When the iteration is terminated, assembly resumes normally at the source line which follows pseudo-instruction FIND.

If the expression declared in the argument field is negative or zero or if the expression representing the value of k is incorrect, assembly resumes directly at the line which follows pseudo-instruction FIND.

If a label appears in the label field, it is associated with the 1st byte generated.

If pseudo-instruction END or subsection pseudo-instruction FIN appear before pseudo-instruction FIND, assembly is interrupted and an error message is printed out.

A DUP/FIND pseudo-instruction can appear within the iteration loop of a DUP/FIND pseudo-instruction. The interleaving level is limited to 6.

A GOTO pseudo-instruction can appear within the iteration loop of a DUP/FIND pseudo-instruction but cannot concern a label outside the iteration loop (no check performed).

Examples

```
DUP                          DUP
 |                            |
 |                            |
DUP                          GOTO, 1  LABEL
 |                            |
 |                            |
FIND          LABEL          |
 |                            |
 |                            |
FIND                         FIND
```

VI-3.3.5. Pseudo-Instruction RMT/FINR

Function

Define a REMOTE sequence.

Format

Label	Command	Argument
< Name >	RMT	
[< Label >]	FINR	

Result

Sequence REMOTE with name < name > located between pseudo-instructions RMT and FINR is saved for a later assembly.

The saved code will be assembled when a pseudo-instruction HERE is detected.

A REMOTE sequence can contain instructions and pseudo-instructions except for pseudo-instructions MACRO, FINM, RMT, FINR, HERE.

A REMOTE sequence can appear in a section.

A REMOTE sequence can appear in a macro-operation ; in this case, pseudo-instruction FINR must appear before pseudo-instruction FINM and the REMOTE sequence will be saved only when the macro-operation is called.

REMOTE sequences with the same name are concatenated in the order in which they are saved.

Example 1

```
M1      RMT
        LDA     P1
        ADD     P2
        STA     P1
        TOTO    A1 ; A2 ; A3
        FINR
```

A TOTO macro-instruction can appear in a REMOTE sequence if macro-operation TOTO does not itself contain a REMOTE sequence or pseudo-instruction HERE.

Example 2

```
        M1          MACRO           P1 ; P2 ; P3

        M2          RMT
                    DATA            P1
                    GE, P2          P3
                    FINR
                    FINM
```

Macro-operation M1 can be called several times ; at each call, sequence REMOTE M2 is concatenated to the REMOTE sequences with the same name.

Calling macro-instruction M1 cannot appear in a REMOTE sequence.

Example 3

```
        M1          MACRO           P1 ; P2 ; P3 ; P4
                    LOCAL           L1 ; L2
        L1          SET             3
        L2          SET             8 - L1
        P1          P2
                    GOTO, P4        L3 ; L4
                    GEN, L1, L2     P4 ; P4
        L3          P3
        L4          FINM
```

Call of macro-operation M1 by the following macro-instruction :

```
        M1      N1 ; RMT ; FINR ; 1
```

generates a REMOTE sequence with correct name N1.

Call of macro-operation M1 by the following macro-instruction :

```
        M1      N1 ; RMT ; FINR ; 2
```

is incorrect, because pseudo-instruction FINR will not be found ; with pseudo-instruction FINM, sequence REMOTE will be ignored.

VI-3.3.6. Pseudo-Instruction RESET

Function

Cancel all REMOTE sequences stored under the same name.

Format

Label	Command	Argument
[< Label >]	RESET	< REMOTE name >

Result

< Label > , if it exists, is not stored in the symbol table.

Generation sequences REMOTE stored under the name < REMOTE name > are reset to zero. This eliminates the cumulative effect of the REMOTE sequences.

Example 1

without RESET :

```
A       RMT
        Sequence 1
        FINR
        HERE            A
                  ------------------> Generation sequence 1

A       RMT
        Sequence 2
        FINR
        HERE            A
                  ------------------> Generation sequence 1
                                              and
                                      Generation sequence 2
```

REMOTE sequences cannot be dissociated and will be generated each time HERE appears.

Example 2

With RESET

```
        A       RMT
                Sequence 1
                FINR
                HERE            A       Generation sequence 1
                RESET                   Cancellation sequence 1

        A       RMT
                Sequence 2
                FINR
                HERE            A       Generation sequence 2
```

VI-3.3.7. Pseudo-Instruction HERE

Function

Call a REMOTE sequence

Format

Label	Command	Argument
[< Label >]	HERE	< Name >

Result

REMOTE sequences with name < name > previously saved by pseudo-instructions RMT/FINR are assembled.

A HERE pseudo-instruction cannot appear in a REMOTE sequence.

VI-3.3.8. Pseudo-Instruction ERROR

Function

Force an error at assembly time.

Format

Label	Command	Argument
[< Label >]	ERROR, < level >	

Result

< level >	= 1	Warning N°24 edited without any generation.
< level >	= 2	Error N°75 edited with a word generated at 0.
< level >	= 3	Serious error N°96 edited causing the end of assembly.

< label > , if it exists, is not stored in the table. The user can indicate, on the listing, errors that he might have detected during the processing of a macro (invalid parameters, etc.)

VI-3.4. Data And Text Generation Pseudo-Instructions

VI-3.4.1 Pseudo-Instruction DATA

Function

Generate data.

Format

Label	Command	Argument
[< Label >]	DATA [,1]	[# ⌜$]< Exp1 > ⌜,[# ⌜$]< Exp2 >...

Result

Pseudo-instruction DATA generates data which have the values of the expressions declared in the argument field.

Data is generated right-justified on a byte if option [,1] is used in the command field.

Data is generated right-justified on a word if option [,1] is not used in the command field.

If a label is present in the label field, it is associated with the first byte generated.

If < Exp1 > is not preceded by symbol # or symbol $, this expression must remain relative to G'.

If < Exp1 > is preceded by symbol #, this expression must remain relative to G.

If < Exp1 > is preceded by symbol $, this expression must remain relative to Z.

VI-3.4.2. Pseudo-Instruction DATAB

Function

Generate values divided by 16.

Format

Label	Command	Argument
[< Label >]	DATAB	< Exp1 >[, < Exp2 >] .....

Result

Pseudo-instruction DATAB generates data on a word which has the values of the expressions declared in the argument field and indicates to the Linkage Editor that these data shall be divided by 16.

If a label is present in the label field, it is associated with the first byte generated.

If the data is not a multiple of 16, the Linkage Editor will immediately assume the next higher multiple of 16.

VI-3.4.3. Pseudo-Instruction DATAF

Function

Generate floating point values on 2 words.

Format

Label	Command	Argument
[< Label >]	DATAF	[- | +]< floating constant >[, [- | +]< floating constant>]...

Result

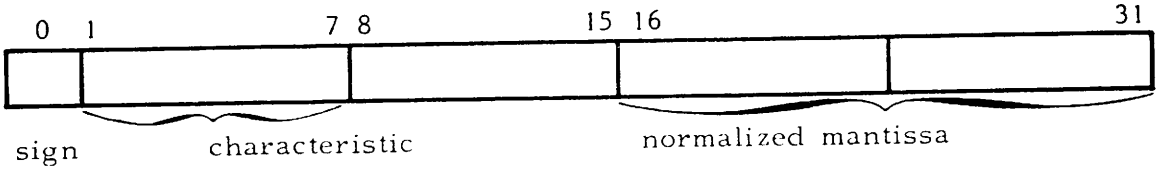
Pseudo-instruction DATAF generates data on two consecutive words.

The single-precision floating constant located in the argument field is converted into an internal floating point format.

Input format :

< integer part > .[<decimal part >] [E <sign> < exponent >]

Output format : (on 2 words)



If a label exists in the label field, it is associated with the first byte generated.

VI-3.4.4. Pseudo-Instruction TEXT

Function

Generate a character string.

Format

Label	Command	Argument
[< Label >]	TEXT	" < character string > "

Result

The character string is generated in EBCDIC format in an area which begins at the address given by the current value of the location counter and ends with the last character generated at any address.

The first byte contains the first character of the string and so on.

If a label exists in the label field, it identifies the first byte generated.

VI-3.4.5. Pseudo-Instruction TEXTC

Function

Generate a character string with its length as the first byte.

Format

Label	Command	Argument
[< Label >]	TEXTC	" < character string > "

Result

The character string is generated in EBCDIC format in an area which begins at the address given by the current value of the location counter and which contains the length of the string in the 1st byte generated and the characters of the string in the following bytes.

If a label exists in the label field, it identifies the first byte generated (string length).

VI-3.4.6. Pseudo-Instruction TEXTA

Function

Generate an ASCII character string.

Format

Label	Command	Argument
[< Label >]	TEXTA	"< Character string > "

Operation

The character string is generated in ASCII format in an area which begins at the address given by the current value of the location counter and ends with the last character generated at any address.

The first byte contains the first character of the string and so on.

If a label exists in the label field, it identifies the first byte generated.

VI-3.4.7. Pseudo-Instruction GEN

Function

Generate values.

Format

Label	Command	Argument
[< Label >]	GEN, < area list >	[$ | #] < Exp1 > [,[$ | #]<Exp2>]...

Result

Pseudo-instruction GEN generates one or two bytes depending on the configuration specified.

< area list > is a list of expressions separated by commas, each expression defining (in binary elements) the length of the generated area (sum of areas must be 8 or 16 binary elements). Zero length areas are forbidden.

Expressions separated by commas declared in the argument field define the contents of each generated area.

There is a one to one correspondence left to right between elements of the area list and elements of the expression list. The code is generated so that the first area contains the first value, etc.

Values are right-justified in their area, the first area corresponds to the most significant bits of the entity.

If a label exists in the label field, it identifies the first generated byte.

If an area is justified on a word boundary and has a 16-bit length, the corresponding expression can be preceded by symbol # (expression relative to G) or symbol $ (expression relative to Z).

VI-3.4.8. Pseudo-Instruction GENOV

Function

Generate the number of the monitor overlay sections. This number will be satisfied by processor OVSGEN.

Format

Label	command	Argument
[< Label >]	GENOVM	< section name >

Result

< Label > , if it exists, is not stored in the symbol table.

This pseudo-instruction is processed like a reference external to the RMI.

VI-3.4.9. Pseudo-Instruction GENCLS

Function

Generate the CLS number on a word.

Format

Label	Command	Argument
[< Label >]	GENCLS	< Name 1 >

Result

< Name 1 > is the CLS name.

< Label > , if it exists, is stored in the symbol table.

VI-3.4.10. Pseudo-Instruction PTW

Function

Generate a logical word pointer (refer to chapter V, § 1-4).

Format

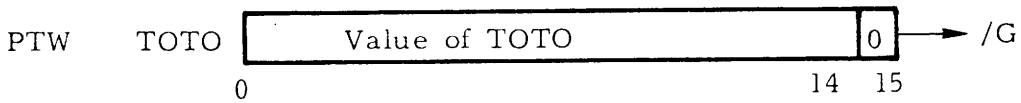
Label	Command	Argument
[< Label >]	PTW	[$ | #] < Exp1 > [,[$ | #]< Exp2 >]...

Result

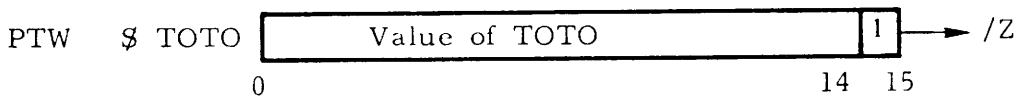
Pseudo-instruction PTW generates logical word pointers on a word which have the values of the expressions < exp i > located in the argument field.

If a label exists in the label field, it is associated with the first generated byte.

If < exp i > is not preceded by character $, the pointer generated is relative to G :



If < exp i > is preceded by character $, the pointer generated is relative to base Z :



VI-3.4.11. Pseudo-Instruction PTB

Function

Generate a logical byte pointer (refer to chapter V, § 1-4).

Format

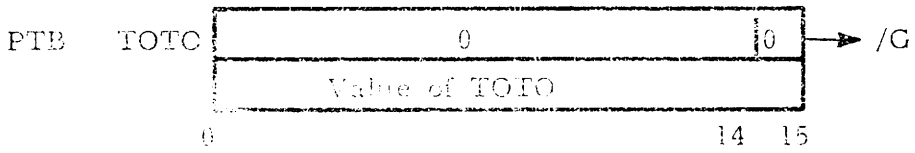
Label	Command	Argument
[< Label >]	PTB	[$ |#] < Exp1 > [,[$ |#] < Exp2 >]...

Result

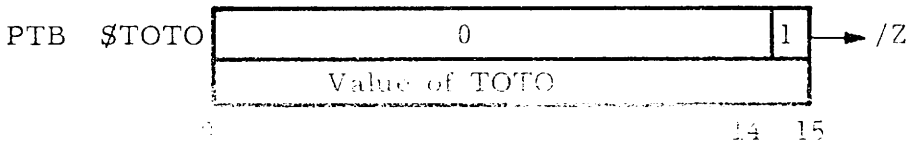
Pseudo-instruction PTB generates the logical "extended" byte pointers which have the values of the expressions < exp i > located in the argument field.

If a label exists in the label field, it is associated with the first generated byte.

If < exp i > is not preceded by character $, the generated pointer is relative to G :



If < exp i > is preceded by character $, the generated pointer is relative to base Z :



VI-3.5. External Definition Pseudo-Instructions

VI-3.5.1. Pseudo-Instruction DEF

Function

Define a label as an external definition for the RB module.

Format

Label	Command	Argument
[< Label >]	DEF	< Label 1 > [, < Label 2 >]...

Result

Pseudo-instruction DEF declares a label located in the argument field of the pseudo-instruction as an external definition for the RB module.

If a label is declared as an external definition, its definition must appear after

pseudo-instruction DEF.

If a label is declared as an external definition, pseudo-instruction DEF must appear before it is used.

Example 1

```
        LDS1        LDS
                    DEF     A
        A           RES     1
                    DATA    A
                    FIN
```

Example 2

```
        LDS1        LDS
        A           RES     1
                    DEF     A
                    FIN
```

The declaration of label A as an external definition appearing after the definition of label A leads to an error.

Example 3

```
        LDS1        LDS
                    DATA    A
                    DEF     A
                    FIN
```

The declaration of label A as an external definition appearing after label A is used causes an error.

VI-3.5.2. Pseudo-Instruction REF

Define a label as an external reference for the RB module.

Format

Label	Command	Argument
[< Label >]	REF	[#]< Label 1 > [,[#] < Label 2 >]...

Result

Pseudo-instruction REF declares a label located in the argument field of the pseudo-instruction as an external reference for the RB module.

The label must be declared as an external reference before it is used.

If the label is preceded by character # , it is a CDS label.

VI-3.5.3. Pseudo-Instruction DEFX

Function

Define a label as an external definition for the RMI module.

Format

Label	Command	Argument
[<Label >]	DEFX	< Label 1 > [,< Label 2 >]...

Result

Pseudo-instruction DEFX declares a label located in the argument field of the pseudo-instruction as an external definition for the RMI module.

If a label is declared as an external definition, its definition must appear after pseudo-instruction DEFX.

If a label is declared as an external definition, pseudo-instruction DEFX must appear before it is used.

VI-3.5.4. Pseudo-Instruction REFX

Function

Define a label as an external reference for the RMI module.

Format

Label	Command	Argument
[< Label >]	REFX	< Label 1> [, < Label 2 >]...

Result

Pseudo-instruction REFX declares a label located in the argument field of the pseudo-instruction as an external reference for the RMI module.

The label must be declared as an external reference before it is used.

VI-3.6. Listing Formatting Pseudo-Instructions

VI-3.6.1. Pseudo-Instructions NOLIST/LIST

Function

Inhibit source text print-out.

Format

Label	Command	Argument
[< Label >]	NOLIST	
[< Label >]	LIST	

Result

Source lines located between pseudo-instructions NOLIST and LIST are not printed out except if assembly option EDIT exists in the assembler call card.

VI-3.6.2. Pseudo-Instruction TITLE

Function

Print-out title and subtitle on the assembly list.

Format

Label	Command	Argument
[< Label >]	TITLE	" < character string> "

Result

The 1st TITLE pseudo-instruction encountered indicates a title ; the character string located in the argument field is printed out at the top of all the pages of the assembly list.

The 2nd TITLE pseudo-instruction encountered indicates a subtitle ; the character string located in the argument field is printed out under the previously defined title.

If a new TITLE pseudo-instruction is encountered, the [illegible] string located in the argument field replaces the previous subtitle character string and is printed out under the previously defined title.

VI-3.6.3. Pseudo-Instruction PAGE

Function

New page.

Format

Label	Command	Argument
[< Label >]	PAGE	

Result

Pseudo-instruction PAGE steps to the next page on the assembly list if the output device is a printer.

VI-3.6.4. Pseudo-Instruction SPACE

Function

New line(s).

Format

Label	Command	Argument
[< Label >]	SPACE	< Expression >

Result

Pseudo-instruction SPACE skips k lines on the assembly list if the output device is a printer ; the value of k is given by the value of the expression located in the argument field.

VI-3.7. Library Definition Pseudo-Instructions

VI-3.7.1. Pseudo-Instruction EXT

Function

Pseudo-instruction EXT declares the subsections required for program module linkage editing.

Format

Label	Command	Argument
[< Label >]	EXT	< subsection 1 name > [, < subsection 2 name >]...

Result

Pseudo-instruction EXT declares external subsections to be integrated with the user program module for linkage editing. These subsections do not have to be necessarily called by a CALL SECTION or designated at linkage editing time.

VI-3.7.2. Pseudo-Instruction XREF

Function

Declare the subsections required for the resolution of external references.

Format

Label	Command	Argument
[< Label >]	XREF	< subsection 1 name >[, < subsection 2 name >]...

Result

Pseudo-instruction XREF declares the external subsections that cannot be integrated with the program module but are designed to satisfy the references to the data that they contain.

The Linkage Editor, therefore, does not have to integrate these subsections with the program module to satisfy the external references.

VI-3.8. Pseudo-Instruction COMMON

Function

Declare access pointer to a COMMON FORTRAN.

Format

Label	Command	Argument
<Pointer Name >	COMMON	[< COMMON Name >]

Result

Pseudo-instruction COMMON declares the pointer name through which the COMMON FORTRAN can be accessed.

Pseudo-instruction COMMON must be used at the end of the user's direct CDS.

The assembler generates a word containing the address of the COMMON whose name is transmitted into the argument field of the pseudo-instruction ; if the name is non-existent, it is the "blank COMMON FORTRAN".

The assembler positions the "FORTRAN Pool" on the first COMMON pseudo-instruction that appears.

The user can describe the COMMON FORTRAN in a COMS section using pseudo-instructions DATA. DATAF, RES.

VII - STRUCTURES

VII-1. INTRODUCTION

To simplify data organization, and, as a result, facilitate program writing and maintenance, the MAS 125 Macro-Assembler uses original program packages that allow processing data blocks or "structures". Processing of a data block includes its definition (or organization) and access possibilities.

MAS 125 uses the following pseudo-instructions to define a data set :

- Structure Definition Pseudo-Instructions (Beginning, End, Name, Length, Type) ;
- Definition Pseudo-Instructions Of Constituent Elements Of A Structure
  An element may be a double-word, a word, a byte, a bit field, or a bit.

The user may access a structure element :

- with the MAS 125 Macro-Assembler macro-instructions ; these are denoted Internal Macro-Instructions ;
- or by creating his own accesses himself.

For each internal macro-instruction, MAS 125 performs checks concerning data block onwnership and verifications on access validity.

VII-2. GENERAL

VII-2.1. Structure Definition

A structure is a data block organized in elements of various lengths called "items" and "subitems". The beginning and end of a structure are declared by pseudo-instructions STRUC and FINST.

VII-2.2. Item Definition

An item is a structure element that has a byte, word, or double-word length. Items are declared by pseudo-instructions BYTE, WORD, and LONG.

VII-2.3. Subitem Definition

A subitem is a structure element that has a bit length or a bit group length. Subitems are declared by pseudo-instructions BIT and BITS.

VI-2.4. Structure Topology

Structure topology will be defined according to the order in which item and sub-item declaration pseudo-instructions appear.

Structure length will be defined by pseudo-instruction SIZE.

Structure name will be defined by pseudo-instruction NAME.

VII-2.5. Structure Redefinition

If the user describes the topology of the same structure several times, only the first definition will be accepted.

VII-2.6. Structure Organization

Structures can be organized :

- in simple structures
- in structures belonging to a structure table.

A structure table contains only simple structures of the same length.

VII-2.7. Structure Justification

The first byte of a simple structure or a structure belonging to a structure table must always be at a word address.

VII-3. STRUCTURE DEFINITION PSEUDO-INSTRUCTION

VII-3.1. Pseudo-Instruction STRUC

Function

Structure declaration.

Format

Label	Command	Argument
[< Label >]	STRUC	[DUM]

Result

Pseudo-instruction STRUC defines the beginning of a structure.

If option DUM exists in the argument field of the pseudo-instruction, no code is generated, the structure is dummy.

Pseudo-instruction STRUC can appear in an SDEC section, in a CDS section, or in an LDS subsection.

Option DUM is mandatory if pseudo-instruction STRUC appears in an SDEC section.

If < Label > is present, < Label > can be used in any expression instead of the name of the structure defined by pseudo-instruction NAME.

If < Label > is present and if option DUM is missing, < label > is considered as a normal CDS or LDS label.

< Label > is a symbol with four characters maximum other than symbols IOR, EOR, AND, SHR, and SHL.

Remark : If the structure is real, its address must be even.

VII-3.2. Pseudo-Instruction NAME

Function

Structure name declaration.

Format

Label	Command	Argument
	NAME	< Structure Name >

Result

Pseudo-instruction NAME defines the structure name.

Pseudo-instruction NAME must immediately follow pseudo-instruction STRUC of the structure declaration.

The structure name is a symbol with four characters maximum other than symbols IOR, EOR, AND, SHR, and SHL.

VII-3.3. Pseudo-Instruction SIZE

Function

Structure size declaration.

Format

Label	Command	Argument
	SIZE	< Expression P1 >

Result

Pseudo-instruction SIZE defines the structure length, in bytes.

Pseudo-instruction SIZE must immediately follow pseudo-instruction NAME of the structure name declaration.

< Expression P1 > indicates the structure length, in bytes.

Example : Simple Structure

```
L1          STRUC
            NAME        N1
            SIZE        10
              |
              |
              |
            FINST
```

Remark : The length of a structure defined in pseudo-instruction SIZE is necessarily even.

The assembler generates implicitly an EQU type symbol, named " structure name ", SZ, which has the value of the structure length.

Example

```
            STRUC       DUM
            NAME        TOTO
            SIZE        24
              |
              |
              |
              |
```

The assembler generates implicitly :

```
            TOTO  SZ  EQU  24
```

VII-3.4. Pseudo-Instruction LONG

Function

Double-word item declaration.

Format 1

Label	Command	Argument
[< Label >]	LONG	[< Character String >]

Format 2

Label	Command	Argument
[<Label>]	LONG	= < Label 2 >

Result

Pseudo-instruction LONG defines the item as a double-word.

< Label > is the name of the item ; it is a symbol of 1 to 6 alphanumeric characters other than symbols IOR, EOR, AND, SHR, and SHL.

Format 1 of the pseudo-instruction declares a double-word item ; < Label > gives a name to this item. If the structure is real and if the argument is missing, the item is initialized at 0 (zero).

If the structure is real and if the argument exists, the item is initialized with < Character String > ; this character string is a string of four alphanumeric characters maximum ; if the string's length is less than four characters, the string is filled in with spaces.

The value associated with the item name, if the item name is specified, is equal to the displacement of the item relatively to the beginning of the structure.

Format 2 of the pseudo-instruction makes an equivalence between the item named < Label 1 > and the item with an already declared name < Label 2 >.

The value associated with the item name < Label 1 > is equal to the displacement of the item with an already declared name < Label 2 > relatively to the beginning of the structure.

Remark : By definition, an item can be identified only by its name in a structure (and not by its displacement) ; equivalence between two items must therefore be used only with great care.

VII-3.5. Pseudo-Instruction WORD

Function

Word item declaration.

Format 1

Label	Command	Argument
[< Label >]	WORD	< Expression >

Format 2

Label	Command	Argument
[< Label 1 >]	WORD	= < Label 2 >

Result

Pseudo-instruction WORD defines the item as a word.

The item must be located in the structure at a word address.

< Label > is the name of the item ; it is a symbol of 1 to 6 alphanumeric characters other than symbols IOR, EOR, AND, SHR, and SHL.

Format 1 of the pseudo-instruction declares a word item ; < Label> gives a name to the item.

If the structure is real and if the argument is missing, the item is initialized at 0 (zero).

If the structure is real and if the argument exists, the item is initialized with < expression >.

The value associated with the item name, if the item name is specified, is equal to the displacement of the item relatively to the beginning of the structure.

Format 2 of the pseudo-instruction makes an equivalence between the item named < Label 1 > and the item with an already declared name < Label 2 >.

The value associated with the item name < Label 1 > is equal to the displacement of the item with the already declared name < Label 2 > relatively to the beginning of the structure.

Remark : By definition, an item can be identified only by its name in a structure (and not by its displacement) ; equivalence between two items must, therefore, be used only with great care.

VII-3.6. Pseudo-Instruction BYTE

Function

Byte item declaration.

Format 1

Label	Command	Argument
[< Label >]	BYTE	< Expression >

Format 2

Label	Command	Argument
[< Label 1 >]	BYTE	= < Label 2 >

Result

Pseudo-instruction BYTE defines the item as a byte.

< Label 1 > is the name of the item ; it is a symbol of 1 to 6 alphanumeric characters other than symbols IOR, EOR, AND, SHR, and SHL.

Format 1 of the pseudo-instruction declares a byte item ; < label > gives a name to this item.

If the structure is real and if the argument is missing, the item is initialized at 0 (zero).

If the structure is real and if the argument exists, the item is initialized with < expression >.

The value associated with the item name, if the item name is specified, is equal to the displacement of the item relatively to the beginning of the structure.

Format 2 of the pseudo-instruction makes an equivalence between the item named < Label 1>and the item with an already declared name < Label 2>.

The value associated with the item name < Label 1> is equal to the displacement of the item with the already declared name < Label 2 > relatively to the beginning of the structure.

Remark : By definition, an item can be identified only by its name in a structure (and not by its displacement) ; equivalence between two items must, therefore, be used only with great care.

VII-3.7. Pseudo-Instruction BITS

Function

Bit group subitem declaration.

Format

Label	Command	Argument
[< Label 1>]	BITS	= < Label 2 > , < beginning> , < length >

Result

Pseudo-instruction BITS defines a subitem as a bit group.

< Label 1> is the name of the subitem ; it is a symbol of 1 to 6 alphanumeric characters other than symbols IOR, EOR, AND, SHR, and SHL.

Pseudo-instruction BITS makes an equivalence between subitem named < Label 1> and the item with an already declared name < Label 2 > which must be type WORD or BYTE.

< Beginning > indicates the rank of the first bit of the bit group of the subitem within the item.

< Length > indicates the number of bits of the bit group of the subitem within the item.

The value associated with the subitem name has in the first byte the rank of the first bit of the group and in the second byte the number of bits of the bit group within the item.

VII-3.8. Pseudo-Instruction BIT

Function

Bit subitem declaration.

Format

Label	Command	Argument
[<Label 1 >]	BIT	= < Label 2 > , < Beginning >

Result

Pseudo-instruction BIT defines a subitem as a bit.

<Label 1>is the name of the subitem ; it is a symbol of 1 to 6 alphanumeric characters other than symbols IOR, EOR, AND, SHR, and SHL.

Pseudo-instruction BIT makes an equivalence between the subitem named <Label 1 > and the item with an already declared name < Label 2 > which must be type WORD

or BYTE.

<Beginning> indicates the rank of the bit of the subitem within the item.

The value associated with the subitem name is equal to the rank of the bit within the item.

VII-3.9. Pseudo-Instruction FINST

Function

"End of structure" declaration.

Format

Label	Command	Argument
	FINST	

Result

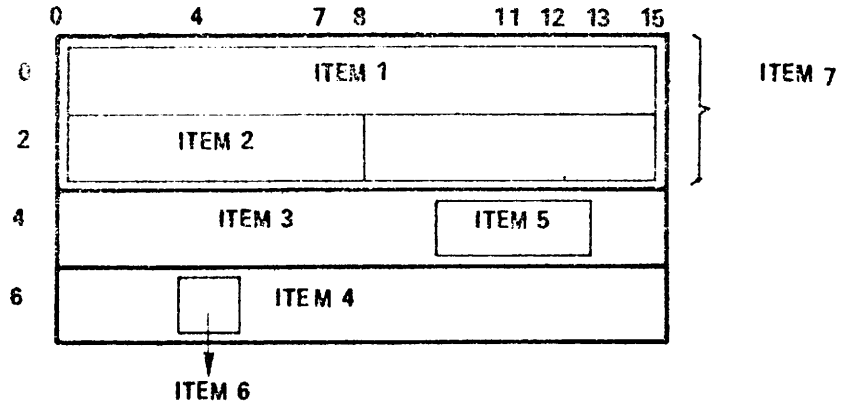
Pseudo-instruction FINST must always immediately follow the last pseudo-instruction of a structure and it indicates the end of the structure.

VII-3.10. Structure Declaration Example

Declaration :

```
        FDT1        STRUC       DUM
                    NAME        FD
                    SIZE        8
        ITEM1       WORD
        ITEM2       BYTE
                    BYTE
        ITEM3       WORD
        ITEM4       WORD
        ITEM5       BITS        =ITEM3, 11,13
        ITEM6       BIT         =ITEM4, 4
        ITEM7       LONG        =ITEM1
                    FINST
```

Topology



Value associated with items and subitems :

ITEM1	&0000
ITEM2	&0002
ITEM3	&0004
ITEM4	&0006
ITEM5	&0B03
ITEM6	&0004
ITEM7	&0000

VII-3.11. Utilization Of A Structure Element Symbol In An Expression

Item or subitem symbols can be used in any expression as operands.

In this case, the symbol has the value associated with the item or subitem.

Example

```
DATA    <structure name> . <item name>
LDX     <structure name> . <item name> . <subitem name>
LDA     <structure name> . <item name> + TOTO
```

This offers the user the possibility to define his own accesses to the structure elements and, if necessary, external access macros.

The assembler can check here only the validity of the item or subitem symbols. The element defined by the address expression can obviously not be checked by a consistency check.

This is the only method that can be used to access BITS subitems. The assembler supplies no internal macro to access this type of subitem.

VII-4. STRUCTURE-ELEMENT ACCESS PRINCIPLE

VII-4.1. Structure Access

Items of a simple structure are directly accessible via internal item access macros.

Items of a structure belonging to a structure table are accessible by internal item access macros only after the structure itself has been accessed. The structure can be accessed by an internal macro with the following general format :

Label	Command	Argument
	ACCESS [,X]	< Structure Name> [,X | A]

VII-4.2. Structure-Item Access

An item of a structure can be accessed by internal macros with the following format :

Label	Command	Argument
[< Label >]	< Operation Code >	[< Addressing >] < Element > [,X]

< Label > is the internal macro label ; it will be associated with the first generated byte.

< Operation Code > is either the name of a MITRA 125 instruction of class 0 or 0', or the name of one of the internal macros LOAD or STORE.

< Addressing > is one of the following addressing symbols :

@ , $, # , @ # , ?

< Element > has the following format :

< Structure Name > . < Item Name >

For each internal macro, the assembler performs the following checks :

- compatibility between the type of item addressed and the type of internal macro used :

. a double-word item (LONG) can be used only with instructions addressing the double word (DLD, DST).

. a word item (WORD) can be used only with instructions addressing the word (LDA, STA, ...)

. a byte item (BYTE) can be used only with instructions addressing the byte (LBR, SBR, ...)

- legality of addressing used to access the structure.
- verification that the addressed item belongs to the structure.

Macro LOAD :

If the user wants to load an item whose type is not known (LONG, WORD, BYTE) in register A or in extended register E,A, he can use a LOAD internal macro :

If the item type is LONG, macro LOAD generates DLD.

If the item type is WORD, macro LOAD generates LDA.

If the item type is BYTE, macro LOAD generates LBR.

Macro STORE :

If the user wants to store register A or extended register E,A in an item whose type is not known (LONG, WORD, BYTE), he can use a STORE internal macro :

If the item type is LONG, macro STORE generates DST.

If the item type is WORD, macro STORE generates STA.

If the item type is BYTE, macro STORE generates SBR.

The assembler generates one or more instructions for each internal macro used.

VII-4.3. Access To A Bit Subitem Of A Structure

Item access :

To access a bit subitem, the user must load the item beforehand in one of the registers (A or E).

If the item is a byte, two cases are possible :

- internal bit-access macros are used ; the item must be placed in the left byte of A or E.

- bit access instructions are directly used ; the item can be placed to the right or left ; the user has to define a correct expression in the argument field of the instruction.

VII-4.3.1. Access To A Bit Subitem Via An Internal Macro

A subitem can be accessed after its item has been correctly loaded in A or E using internal macros with the following format :

Label	Command	Argument
[< Label >]	< Operation Code >	< Structure Element >

< Label > is the label of the internal macro ; it will be associated with the first generated byte.

< Operation Code > is the name of a MITRA 125 instruction addressing the bit (CBA, CBE, RBA, RBE, TBA, TBE, SBA, SBE)

< Structure Element > has the following format :

< Structure Name > . Item Name > . < Subitem Name >

For each internal macro, the assembler performs the following checks :

. check that the addressed subitem is type BIT
. check that the addressed subitem belongs to the item.
. check that the item belongs to the structure.

The assembler generates an instruction of the following format for each internal macro used :

[< Label >] < Operation Code > = < Value Associated With the Subitem >

VII-4.3.2. Access To A Bit Subitem Via A Bit Access Instruction

The item is a BYTE type item and has been placed in the left byte of A or E, or it is a WORD type item :

[< Label >] < Operation Code > = < Structure Element >

The item is a BYTE type item and has been placed in the right byte of A or E :

[< Label >] < Operation Code > = < Structure Element > + 8

< Operation Code > and < Structure Element > have the same definition as in the internal bit access macros.

The assembler performs only the validity check of < Structure Element > . The bit actually addressed by the address expression can obviously not be checked by a consistency check.

VII-4.4. Pointers And Literals To Be Declared By The User

Some sequences generated by the internal macros with access to structures or structure items use :

- literals with format K : xxxx
- literals with format Z : xxxx
- pointers with format T : xxxx

The assembler references them ; the user has to reserve them.

SYMBOLS GENERATED	THE USER MUST CODE
K : < hexadecimal value >	K : < hex. value> DATA &< hex. value >
Z : < hexadecimal value >	Z : < hex. value> PTB &< hex. value >
	or Z : < hex. value> PTB $&< hex. value >
T : < structure name >	

Simple structure containing the byte items and accessed by a logical pointer

T : < structure name > RES2 and store, before access, the generalized address of the structure in the 1st word. The 2nd word must contain a value zero.

Simple structure which does not contain byte items :

T : < structure name > RES1 and store, before access, the generalized address of the structure in this word.

Structure belonging to a structure table :

T : < structure name > RES1 and store, before access, the table displacement relative to the beginning of the segment in this word.

Remark : The simple structures whose last item has an associated value < 256 do not require literals (parameter mode addressing).

The bit subitems do not require literals (displacement < 16, parameter mode

addressing).

Literals K : < hexadecimal value > generated by certain structure access macros are used in two different ways :

- as a byte displacement and take thus the values associated with the different items.
- as a logical word pointer relative to Z and take thus the values associated with the various word items + 1 .

Literals Z : < hexadecimal value > generated by certain structure access macros are used as a logical byte pointer relative to Z :

- the first word contains value 1 (address /Z)
- the 2nd word contains the value associated with the byte items.

VII-5. ACCESS TO AN ITEM OF A SIMPLE STRUCTURE BASED BY A LOGICAL POINTER

Items of simple structures belonging to the program segment or to the data segment and based by a logical pointer can be accessed by access macros with the following format :

Access Macro Format

Label	Command	Argument
[< Label >]	< Operation Code >	< Structure Name > . < Item Name >

Generated Code

If the value associated with the addressed item is less than 256, the generated code is :

```
[< label >] LDX = < associated value >
  < operation code > $T : < structure name >, X
```

If the value associated with the addressed item is greater than 256, the generated code is :

```
[< label >] LDX  K : < associated value >
  < operation code > $T : < structure name > , X
```

User Interface

Access can be made from the program segment or the subroutine segment.

The user must generate constants with values equal to the associated values > 256 of the various items of the structure in the program CDS or in the LDS of the section accessing the structure.

K : < associated value > DATA & < associated value >

The user must reserve in the program CDS or in the LDS of the section accessing the structure :

- a logical word pointer - if the structure does not contain a byte item :

  T : < structure name > RES1

  and store, before access, the generalized address of the structure in this word.

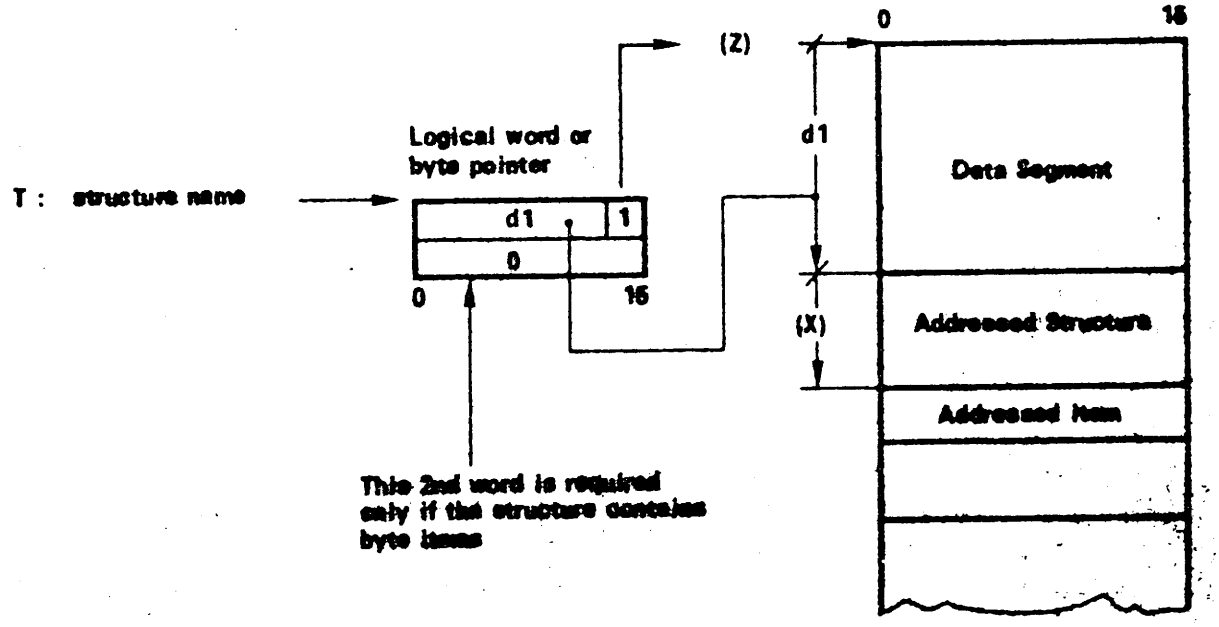
- a logical byte pointer - if the structure contains byte items :

  T : < structure name > RES2

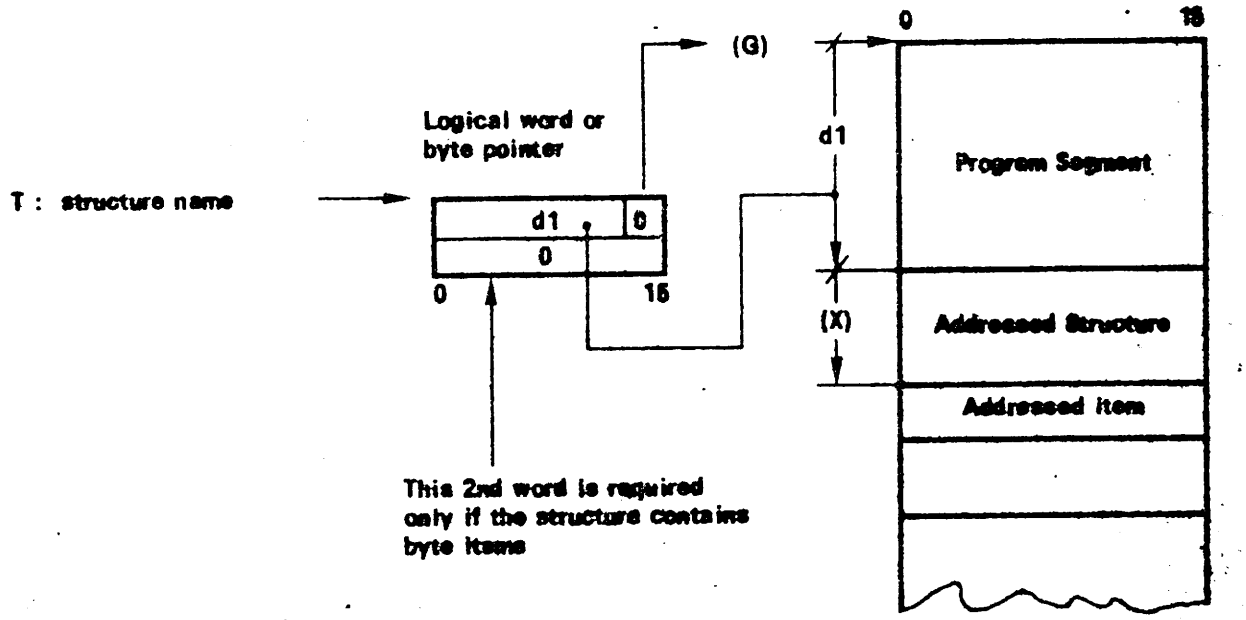
  and store, before access, the generalized address of the structure in the 1st word ; the 2nd word must remain at zero.

Topology

The structure belongs to the program segment (based by G) :



The structure belongs to the data segment (based by Z) :



VII-6. SPECIAL CASE : STRUCTURE BELONGING TO THE PROGRAM SEGMENT AND DIRECTLY BASED BY G

Access Macro Format

Label	Command	Argument
[< Label >]	< Operation Code >	# < Structure Name > . < Item Name >

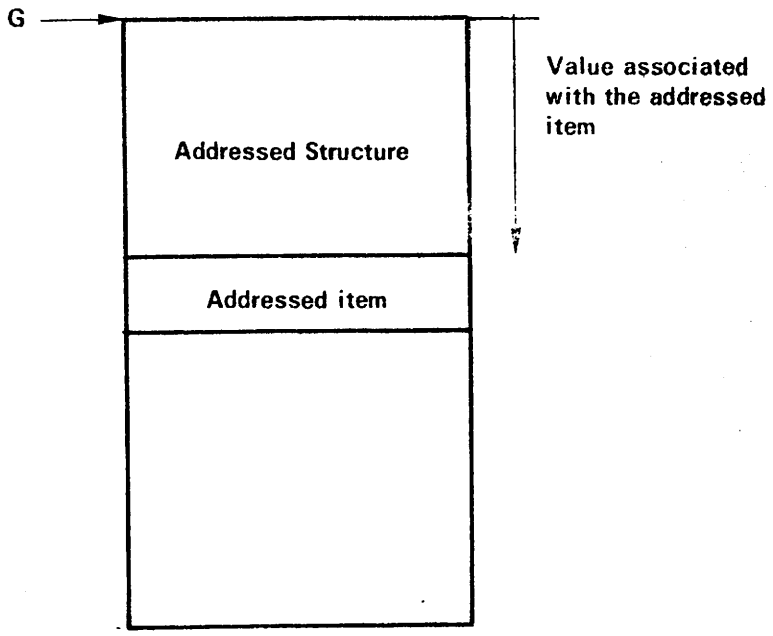
Interface

Base G points to the beginning of the addressed structure.

Generated Code

[< Label >] < Operation Code > # < Value Associated With The Item >

Topology



VII-7. ACCESS TO STRUCTURES BASED BY X

VII-7.1. Access To Simple-Structure Items Or A Table Based By X And Belonging To The Program Segment Or The Subroutine Segment

Items of structures belonging to the program segment or the subroutine segment and based by (X) can be accessed by means of macros with the following format :

Label	Command	Argument
[< Label >]	< Operation Code >	<Structure Name> . < item name>[,X]

Generated Code

[<Label>] < Operation Code > @ K < Associated Value> [, X]

User Interface

If the structure belongs to the program segment, access can be made from the program segment or the subroutine segment.

If the structure belongs to the shareable subroutine segment, access can be made only from this segment.

X must contain - before access - the byte displacement of the structure relatively to the beginning of the segment to which it belongs. If this displacement is zero, (structure at beginning of segment), X can be transmitted in the item-access macro : it will not appear in the generated instruction.

The user must generate, in the CDS or the LDS of the section accessing the structure, constants with same values as those associated with all the structure items :

K : < associated value > DATA & < associated value >

If the structure belongs to the subroutine segment, this constant must be generated in the LDS, since only indirect, local addressing is relative to Q.

VII-7.2. Access To Simple-Structure Items Or A Table Based By X And Belonging To The Data Segment Based By Z

Items of structures belonging to the data segment and based by (X) can be accessed by means of macros with the following format :

Label	Command	Argument
[<Label>]	<Operation Code>	$<Structure Name> . <Item Name> [, X]

Generated Code

If <Operation Code> is an instruction addressing the word :

[<Label>] <Operation Code> $ K : <Associated Value + 1> [, X]

If Operation Code is an instruction addressing the byte :

[<Label>] <Operation Code> $ Z : <Associated Value> [, X]

User Interface

Access can be made from the program segment or the subroutine segment.

X must contain, before access, the byte displacement of the structure relatively to the beginning of the data segment. If this displacement is zero (structure at beginning of segment) [,X] can be omitted in the item-access macros : it will not appear in the generated instruction.

The user must generate, in the CDS or in the LDS of the section accessing the structure :

- logical word pointers relative to Z whose displacements are the values associated with the word items of the structure.

  K : <Associated Value + 1> DATA & <Associated Value + 1>

- logical byte pointers relative to Z whose displacements are the values associated with the byte items of the structure.

  Z : <Associated Value> PTB $& <Associated Value>

VII-7.3. Access To A Structure Belonging To A Structure Table

If the structure belongs to a structure table, it can be based by X using one of the following 3 access macros :

- ACCESS, X < Structure Name >
- ACCEXX, X < Structure Name > , X
- ACCESS, X < Structure Name > , A

Common Interface

The user must reserve a pointer in the LDS of the section accessing the structure :

T : < Structure Name > RES1

and store, before access, the address relative to the beginning of the segment of the structure table in this word.

After execution, X contains the byte displacement of the structure relatively to the beginning of the segment to which it belongs, and previous internal item-access macros can be used.

Access Macro Format 1

Label	Command	Argument
	ACCESS, X	< Structure Name >

Generated Code

ICX T : < Structure Name >

Input Interface

Register X must contain the byte displacement of the addressed structure relatively to the beginning of the table.

Access Macro Format 2

Label	Command	Argument
	ACCESS, X	<Structure Name> [, X]

Generated Code

The code generated by the ACCESS macro depends on the length (in bytes) of the structures organized as a table :

```
SIZE = 2        ICX     = 0 , X
                ICX     T : < Structure Name >
SIZE = 4        ICX     = 0, X
                ICX     = 0, X
                ICX     T : < Structure Name >
SIZE = 2^n      XAX
                SLLS    = n
                XAX
                ICX     T : < Structure Name >
SIZE = n        XAX
                MUL     = n
                XAX
                ICX     T : < Structure Name >
```

Input Interface

Register X must contain the number of the structure addressed in the structure table. The first structure has number zero.

Access Macro Format 3

Label	Command	Argument
	ACCESS, X	< Structure Name > , A

Generated Code

The code generated by the Access Macro depends on the length (in bytes) of the structures organized as a table.

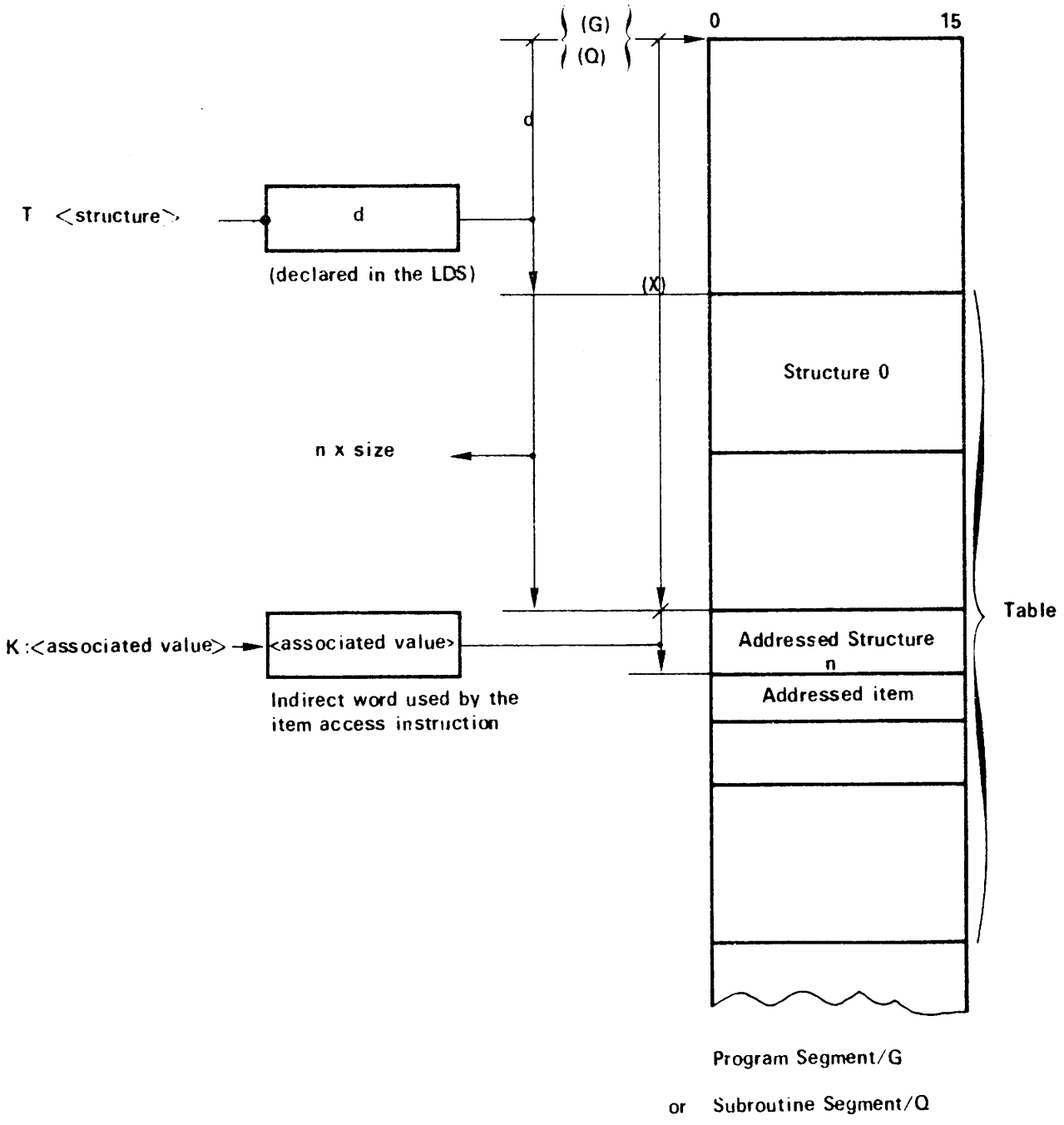
SIZE = 2	SLLS	= 1
	XAX	
	ICX	T : < Structure Name >
SIZE = 4	SLLS	= 2
	XAX	
	ICX	T : < Structure Name >
SIZE = 2^n	SLLS	= n
	XAX	
	ICX	T : < Structure Name >
SIZE = n	MUL	= n
	XAX	
	ICX	T : < Structure Name >

Input Interface

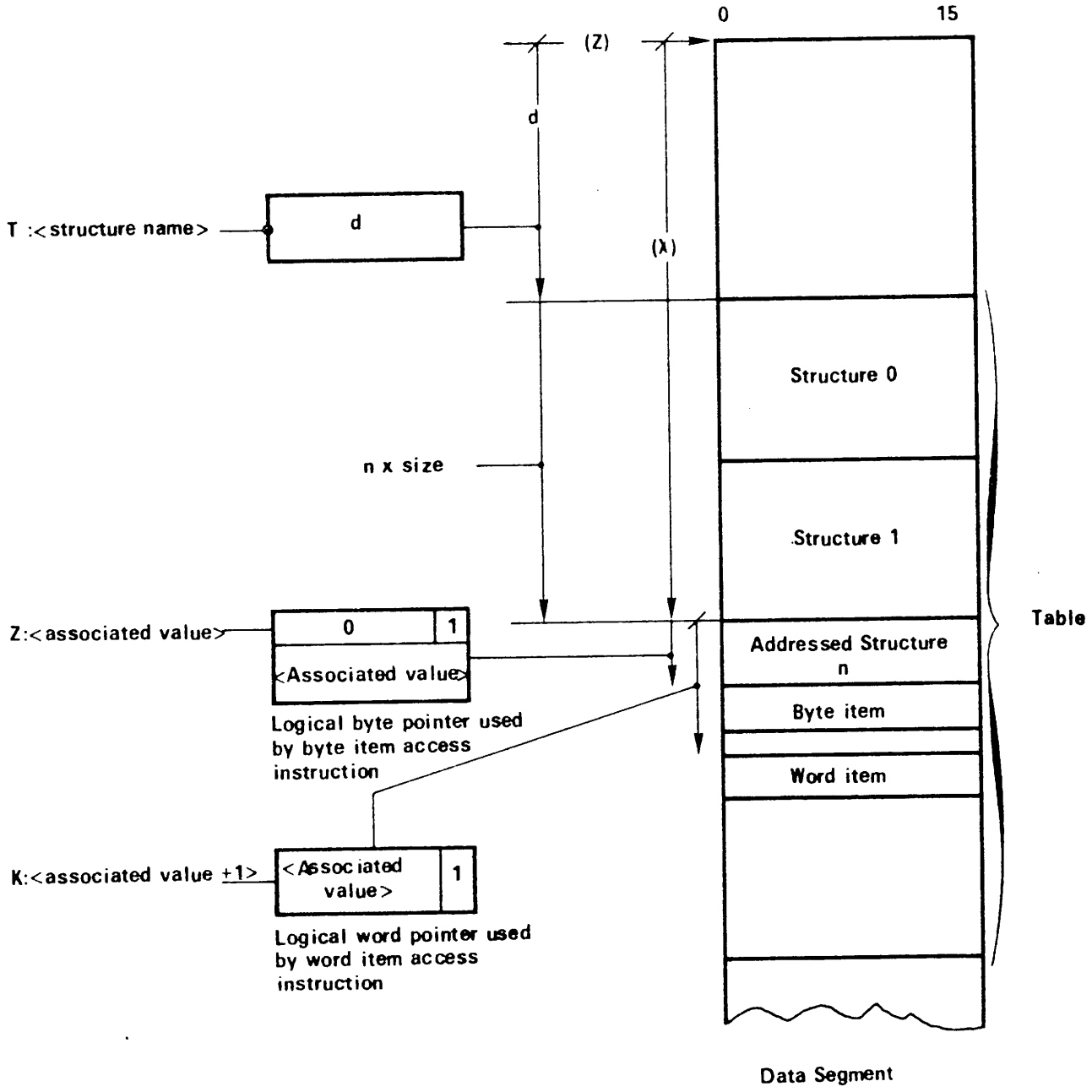
Register A must contain the number of the structure addressed in the structure table. The first structure has number zero.

Topology

The structure belongs to the program segment based by G or to the subroutine segment based by Q.



The structure belongs to the data segment based by Z



VII-8. SPECIAL ACCESS CASE TO A ZERO DISPLACEMENT ITEM IN A STRUCTURE BELONGING TO A STRUCTURE TABLE OF THE PROGRAM SEGMENT OR SUB-ROUTINE SEGMENT

VII-8.1. Structure Access

Since the value associated with the item is zero, an ICX instruction can be economized in the generated code by using one of the following 2 access macros :

- ACCESS < Structure Name > , X
- ACCESS < Structure Name > , A

Common Interface

After execution, X contains the byte displacement of the addressed structure relatively to the beginning of the table .

Access Macro Format 1

Label	Command	Argument
	ACCESS	< Structure Name > , X

Generated Code

The code generated by the ACCESS macro depends on the length (in bytes) of the structures organized as a table.

```
SIZE = 2          ICX      = 0, X
SIZE = 4          ICX      = 0, X
                  ICX      = 0, X
SIZE = 2^n        XAX
                  SLLS     = n
                  XAX
SIZE = n          XAX
                  MUL      = n
                  XAX
```

Input Interface

Register X must contain the number of the structure addressed inside the table. The first structure has number zero.

Access Macro Format 2

Label	Command	Argument
	ACCESS	< Structure Name > , A

Generated Code

The code generated by macro ACCESS depends on the length (in bytes) of the structures organized as a table.

```
SIZE = 2          SLLS    = 1
                  XAX
SIZE = 4          SLLS    = 2
                  XAX
SIZE = 2^n        SLLS    = n
                  XAX
SIZE = n          MUL     = n
                  XAX
```

Input Interface

Register A must contain the number of the structure addressed inside the table. The first structure has number zero.

VII-8.2. Access To An Item With Zero Displacement

The first item of the structure can be accessed by means of an internal macro with the following format :

Label	Command	Argument
[< Label >]	< Operation Code >	@ < Structure Name > . < Item Name > [, X]

Generated Code

The code generated by the macro Operation Code is :

[< Label >] < Operation Code > @ T : < Structure Name > [, X]

User Interface

If the structure belongs to the program segment, access can be made from the program segment or the subroutine segment.

If the structure belongs to the shareable subroutine segment, access can be made solely from this segment.

X must contain, before access, the byte displacement of the structure relatively to the beginning of the table to which it belongs. If this displacement is zero (1st structure of table), X can be omitted ; it will not appear in the generated instruction.

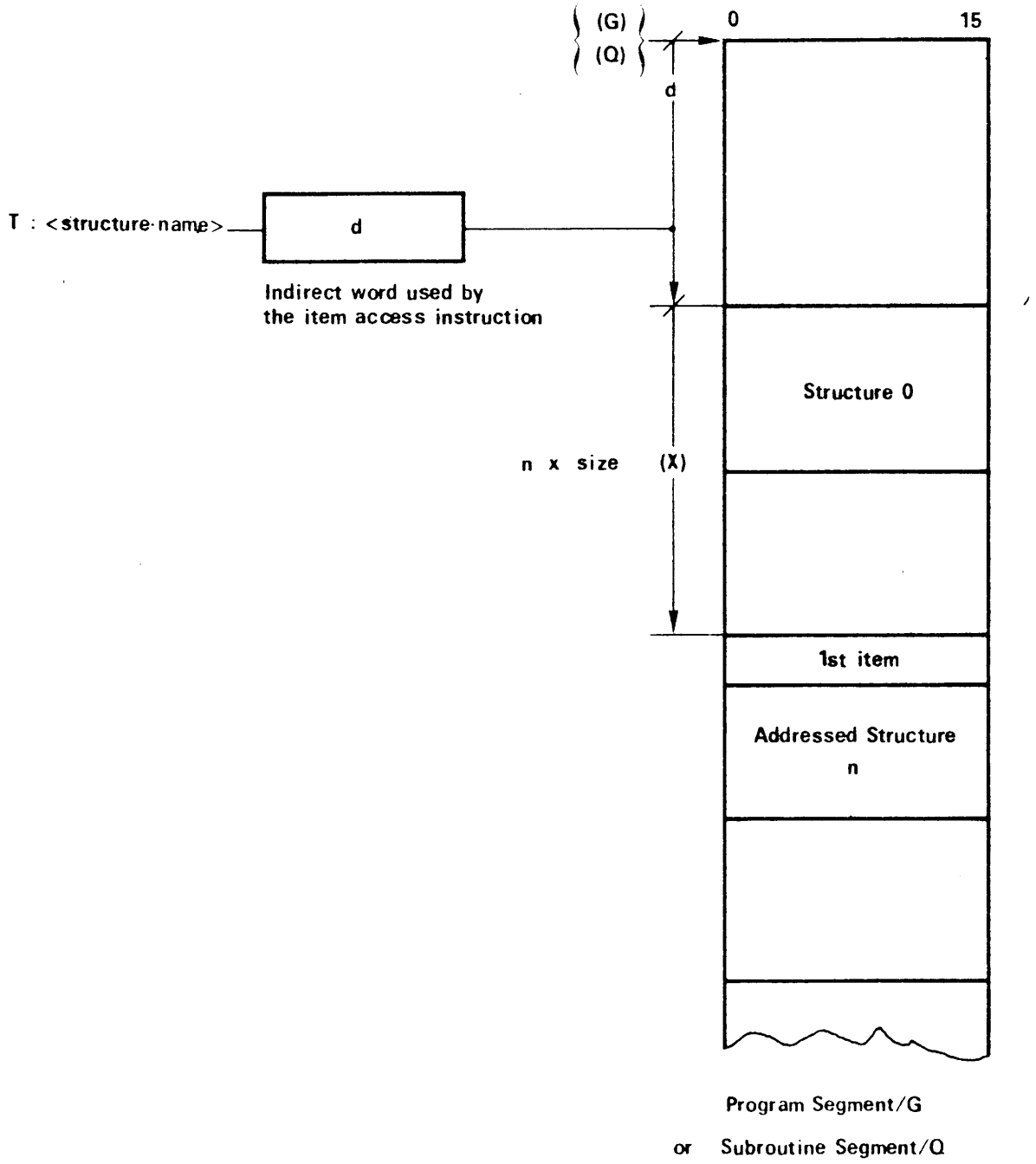
The user must generate a pointer in the CDS or in the LDS of the section accessing the structure :

T : < Structure Name > RES1

and store, before access, the structure table address relative to the beginning of the segment in this word.

If the structure belongs to the subroutine segment, this pointer will be reserved in the LDS since only indirect local addressing is relative to Q.

ACCESS TO AN ITEM WITH ZERO DISPLACEMENT



VIII - MACRO-OPERATIONS

VIII-1. DEFINITION

A macro-operation is a special type of pseudo-instruction whose name and function are defined by the user ; a macro-definition and a macro-instruction correspond to each macro-operation.

A macro-definition defines which source line images a macro-instruction will create ; it comprises a header line, "prototype" line images, and an "end" line.

A macro-instruction is the instruction that calls a macro-definition. The sequence of lines generated by a macro-instruction is an open subroutine since the lines generated are automatically inserted in the source text processed by the assembler instead of the macro-instruction.

The expansion of a macro-operation consists of duplicating each one of the "prototype" lines belonging to the macro-definition, substitutable arguments being replaced by the arguments of the macro-instruction which has caused the expansion.

Macro-definitions must be at the head of the user program.

Example

Macro-Definition :

```
TOTO    MACRO    A; B; C      Header
        LDA      A            Prototype
        ADD      B            Line
        STA      C            Images
        FIN                   End
```

Macro-Instruction :

```
        TOTO     X; Y; Z
```

Macro-Operation Expansion :

```
        LDA      X
        ADD      Y
        STA      Z
```

VIII-2. MACRO-DEFINITION

A macro-definition comprises several parts :

- a macro-operation identifier ;
- a list of substitutable arguments of the macro-operation ;
- a list of declarations of the macro-operation symbols ;
- the body of the macro-operation or "prototype" line images ;
- an "end of macro-operation" declaration.

VIII-2.1. Macro-Operation Identifier

The name of a macro-operation is identified by the symbol located in the label field of a MACRO pseudo-instruction.

This name becomes the operation code of the macro-instruction for this macro-operation.

VIII-2.2. Substitutable Macro-Operation Arguments

Substitutable arguments of a macro-operation are identified by the symbols located in the argument field of a MACRO pseudo-instruction.

These are formal parameters corresponding to the fields of a macro-operation which can be modified each time the macro-operation is used (label field, command field, and argument field).

They are present in two different places of the macro-definition : in the header line and in the prototype line image.

VIII-2.3. Macro-Operation Symbols Definition

A symbol used in the body of a macro-operation can be :

- declared as a parameter : it will be replaced when the macro-operation is expanded by its value.
- declared as a global variable : it will be known and can be used by all the MACROs which have declared it GLOBAL.
- declared as a local variable : it will be known and can be used only within the MACRO which has declared it LOCAL.
- not declared : it is any symbol of the user program.

VIII-2.4. Macro-Operation Body

The body of the macro-operation contains prototype card images defining the actual operation, i.e., the sequence of instructions that must be inserted as well as the position of the substitutable fields in each instruction (label field, command field, and argument field).

The macro-operation body can be made up of :

- MITRA 125 instructions (provided they are enabled in the segment in which the call macro-instruction is located) ;
- MITRA 125 pseudo-instructions (except for pseudo-instructions MACRO, FINM, LIST, and NOLIST and provided they are enabled in the segment in which the call macro-instruction is located) ;
- macro-instructions ;
- pseudo-instructions specific for the Macros (IF...GOTO, IF...INIT).

VIII-2.5. End Of Macro-Operation

The end of a macro-operation is identified by the occurrence of pseudo-instruction FINM.

Any macro-operation must be terminated by an end of macro-operation.

Remark :

Calls can be recursive.
The comments field is not accepted within a macro-operation.

VIII-3. MACRO-OPERATION PSEUDO-INSTRUCTIONS

VIII-3.1. Definition

Macro-operation pseudo-instructions permit defining a macro-operation ; They are :

- macro declaration : MACRO
- global variable declaration : GLOBAL
- local variable declaration : LOCAL
- conditional branch : IF ... GOTO
- conditional initialization : IF ... INIT
- "end-of-Macro" declaration : FINM

VIII-3.2. Macro Declaration Pseudo-Instruction : MACRO

Function

Pseudo-instruction MACRO defines a macro-operation ; this is the first macro-definition card.

Format

Label	Command	Argument
< Name >	MACRO	[< Parameter 1> [; < Parameter 2 >]...]

Result

The name defined in the label field is the name of the macro-operation which will be defined in the associated prototype.

The list of parameters defined in the argument field contains the substitutable arguments that will be found in the associated prototype. They are substituted in the order in which they occur.

Parameters can be non-initialized formal parameters with the format param or initialized formal parameters with the format :

(< param > = < character string >)

The macro declaration pseudo-instruction can have several continuation lines : character ";" placed at the end of the argument field indicates that the parameter list follows in a continuation line ; this continuation line has character ";" in column 1.

The result of this pseudo-instruction is to define the macro-operation by connecting the name and the substitutable arguments to the associated prototype, and to store the prototype in the macro-operation skeleton table.

Remark :

The name of a MACRO cannot be a sectioning pseudo-instruction's name (SDEC, MACRO, CDS, COMS, LDS, IDS, LPS, FINM, FIN, FINC, END), an assembly control pseudo-instruction's name (DO, DUP, FIND, RMT, FINR, HERE), or a structure pseudo-instruction's name (STRUC, FINST).

If pseudo-instruction MACRO is located in section SDEC, the macro-operation is considered a system macro-operation.

If pseudo-instruction MACRO is located after section SDEC, the macro-operation is considered a user macro-operation.

A macro-operation can be redefined ; the redefinition of this macro-operation replaces the old prototype by the new one.

VIII-3.3. Global Variable Declaration Pseudo-Instruction : GLOBAL

Function

Pseudo-instruction GLOBAL defines the global variables kown by several macro-operations.

Format

Label	Command	Argument
[< Label >]	GLOBAL	< Symbol 1 > [, < Symbol 2 >]...

Result

Pseudo-instruction GLOBAL must, if it exists, always immediately follow pseudo-instruction MACRO.

Symbols appearing in the argument field are defined as global variables, i.e., known by several macro-operations.

VIII-3.4. Local Variable Declaration Pseudo-Instruction : LOCAL

Function

Pseudo-instruction LOCAL defines variables local to a macro-operation.

Format

Label	Command	Argument
[< Label >]	LOCAL	< Symbol 1 > [, < Symbol 2 >]...

Result

Pseudo-instruction LOCAL must always - if it exists - immediately follow pseudo-instruction MACRO and pseudo-instruction(s) GLOBAL, if they exist.

Symbols in an argument field are defined as local variables, i.e., known only by the macro-operation being executed.

VIII-3.5. Conditional Branch Pseudo-Instruction : IF ... GOTO ...

Function

Pseudo-instruction IF ... GOTO ... makes a conditional branch within a macro-operation.

Format

Label	Command	Argument
[< Label >]	IF, " < Parameter > "	GOTO, < Label 2 >

Result

< Parameter > is the name of a substitutable argument of a macro-operation defined in the argument field of pseudo-instruction MACRO.

< Label 2 > is the name of an internal label defined in the body of the macro-operation.

If < Parameter > is not initialized in the macro-instruction, the assembler resumes the expansion of the macro-operation at the source line labeled by < Label 2 >.

If < Parameter > is initialized in the macro-instruction, the assembler continues the expansion of the macro-operation at the source line which follows the pseudo-instruction.

VIII-3.6. Conditional Initialization Pseudo-Instruction : IF ... INIT ...

Function

Pseudo-instruction IF ... INIT ... makes a conditional initialization within a macro-operation.

Format

Label	Command	Argument
[< Label >]	IF, " < Parameter > "	INIT, < Variable >

Result

< Parameter > is the name of a substitutable argument in a macro-operation defined in the argument field of pseudo-instruction MACRO.

< Variable > is the name of a symbol used in the body of a macro-operation.

If < Parameter > is not initialized at the time of the macro-operation, < Variable > is initialized at value 0.

If < Parameter > is initialized at the time of the macro-operation, < Variable > is initialized at value 1.

The definition of < Variable > is identical to a definition by an EQU pseudo-instruction.

VIII-3.7. "End Of Macro" Declaration Pseudo-instruction : FINM

Function

Pseudo-instruction FINM terminates a macro-operation.

Format

Label	Command	Argument
	FINM	

Result

Pseudo-instruction FINM must always immediately follow the last line of a prototype and indicates the end of the macro-operation.

VIII-4. MACRO-INSTRUCTION

Function

A macro-instruction (call instruction of a macro-definition) can be located in the prototype of a macro-definition, in a common data section, in an LDS or LPS subsection, or outside of a subsection provided the result of the resultant macro-operation expansion is accepted in the section or segment in which the macro-instruction is located.

Format

Label	Command	Argument
[< Label >]	< Name >	[< Parameter 1 > [; < Parameter 2 >]...

Result

The label defined in the label field is associated with the first line of the macro-operation expansion.

The list of parameters separated by ";" characters and defined in the argument field contains the arguments that must replace the substitutable arguments of the prototype of the called macro-definition.

Parameters can be actual parameters or initialized formal parameters with format :

(< Parameter > = < Character String>)

Actual parameters must appear in the same order as formal parameters that they must replace appeared in the called macro-definition.

Initialized formal parameters can appear in any order, but an actual parameter cannot follow an initialized formal parameter.

A macro-instruction can have several continuation lines : character ";" placed at the end of the argument field indicates that the parameter list follows in a continuation line ; this continuation line has character ";" in column 1.

Example

Macro-Definition :

```
TOTO        MACRO     A ; (B = CHAIN2) ; OPE ;
; (C = CHAIN3)
            LDA       A
            ADD       B
            OPE       C
            FINM
```

Macro-Instruction :

```
TOTO        CHAIN1 ; (OPE = STA) ;
; (C = CHAIN4)
```

Macro-Operation Expansion :

```
LDA         CHAIN1
ADD         CHAIN2
STA         CHAIN4
```

Formal parameter A has been replaced by string CHAIN1 when the macro-operation was called.

Formal parameter B has been replaced by string CHAIN2 when the macro-operation was defined.

Formal parameter C, replaced by string CHAIN3 when the macro-operation was defined has been replaced by string CHAIN4 at the macro-operation call.

Examples : Macro-Definition :

```
TOTO        MACRO     (A = CHAIN1) ; B ; OPE ; (C = CHAIN3)
            GLOBAL    A
            LDA       A
            ADD       B
            OPE       C
            FINM
TITI        MACRO     A ; (B = CHAIN4) ; C
            GLOBAL    A
            LDA       A
            SUB       B
            STA       C
            FINM
```

Macro-Instruction :

```
ETIQ1       TOTO    A ; (B = CHAIN2) ; (OPE = STA) ; (C = CHAIN5)
```

Macro-Operation TOTO Expansion :

```
ETIQ1       LDA     CHAIN1
            ADD     CHAIN2
            STA     CHAIN5
```

Macro-Instruction :

```
ETIQ2       TITI    A ; B ; (C = CHAIN6)
```

Macro-Operation TITI Expansion :

```
ETIQ2       LDA     CHAIN1
            SUB     CHAIN4
            STA     CHAIN6
```

Remark

If macro-operation expansion requires location counter justification to a word boundary, the label defined in the label field of the macro-instruction is associated with the expansion line following the generated BND pseudo-instruction.

A parameter can concern one or more fields.

Examples

Macro-Definition :

```
TOTO        MACRO   A ; (B = CHAIN2) ; OPE
            A
            ADD     B
            OPE     C
```

Macro-Instruction :

```
TOTO        LDA     CHAIN1 ; (OPE = STA)
```

Macro-Operation Expansion :

```
            LDA     CHAIN1
            ADD     CHAIN2
            STA     C
```

VIII-5. SUBSTITUTION AND CONCATENATION RULES

VIII-5.1. During Macro Definition

The assembler stores the macro-definition in a skeleton form ; each line is compressed and symbols are renamed (to avoid double names for variables outside the macro and variables in the macro).

Rules governing the renaming of symbols are :

- Any assembly line is a sequence of symbols placed between separators.

  The following are considered as separators :
  . beginning and end of a line
  . characters . $ @ # = , ; + - / * space ? ( ) '

- If the symbol was declared as a parameter (i.e., defined as a keyword), it is renamed by a symbol of type > P xxxx (xxxx = parameter number)

- If the symbol was declared as a local variable (i.e. defined in pseudo-instruction LOCAL), it is renamed by a symbol of type > L yyyy (yyyy = local variable number).

- If the symbol was declared as a global variable (i.e., defined in pseudo-instruction GLOBAL), it is renamed by a symbol of type > G zzzz (zzzz = global variable number).

Example

NAME	MACRO	KEYWORD	
	GLOBAL	GLOB	not stored in the
	LOCAL	LOC	skeleton
GLOB	LDA	= KEYWORD + LOC	
	FINM		

The skeleton will be :

> G 0000 ␣ LDA ␣ = > P 0000 + > L 0000

VIII-5.2. During The Macro Call

Skeleton lines are processed successively.

Any symbol of type > G xxxx remains unchanged ; global variables must have the same name in all the macros that are using them.

Any symbol of type > L yyyy is converted into a symbol of the same type but whose number was updated according to the macro call level.

Any symbol of type > P zzzz is replaced by the string which represents the value of this parameter or by a binary zero if this parameter is not initialized.

If the separators on both sides of the symbol are both not a character ' , a simple substitution takes place, and the separators remain unchanged.

Example

```
    LDA          KEYWORD          Macro definition line
␣ LDA  ␣  > P0000                Skeleton line
```

when the call is made KEYWORD = TOTO

```
␣ LDA  ␣   TOTO                  After substitution
```

If the symbol is between two characters ' , a micro-substitution takes place :

. The symbol is type >P xxxx

The symbol is replaced by the character string which represents it and both separators (') are eliminated.

Example

```
␣ LDA ␣ = 2'  > P 0000'          Skeleton line
  > P 0000  →  345               Macro call
␣ LDA ␣ = 2345                   After substitution
```

. The symbol is type > L yyyy or > G zzzz

The symbol is replaced by the EBCDIC character string which represents the value associated with the symbol (leading zeros are eliminated) ; separators (') are eliminated.

Example

```
  > L 0000␣SET␣4
␣ LDA ␣ =  '  > L 0000' 2     } skeleton
```

```
>L 0000␣SET␣4
␣LDA ␣ = 42
```

Substitution

Remark :

Calls may be recursive.

IX - USER INSTRUCTIONS AND DATA DESCRIPTION

IX-1. GENERAL

This chapter describes MITRA 125 instructions and data usable in User mode (SV=0, PV=0).

These instructions are described in alphabetical order.

The following are indicated for each instruction :

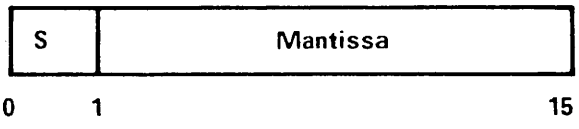
- the instruction name
- the instruction class (the instruction class indicating the permitted addressing type)
- the instruction code versus type of addressing used
- the instruction function
- elements modified by the instruction (memories, registers, indicators)
- possible traps

IX-2. DATA REPRESENTATION

IX-2.1. Fixed Single-Precision Formatted Data

The single-precision fixed format permits representing a decimal number on one MITRA 125 word.

This format comprises 1 sign bit S (bit 0) and 15 mantissa bits M (bits 1 through 15).



A single-precision fixed number N is divided as follows :

Sign S is the sign of the mantissa M
Mantissa M ($0 =< M =< 2^{16} - 1$)

The formal definition of the single-precision fixed number N is as follows :

If N > 0 N = M

A single-precision fixed number with mantissa M zero is a zero.

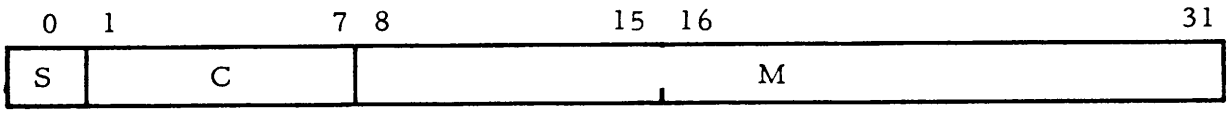
A negative single-precision fixed number is represented by the two's complement of its absolute value.

Representation of single-precision fixed-formatted numbers.

Decimal Number	Hexadecimal Value
+ 32767	7FFF
+ 1	0001
0	0000
- 1	FFFF
- 32767	8001

IX-2.2. Single-Precision Floating Formatted Data

The single-precision floating format permits representing a decimal number on 2 consecutive MITRA 125 words. This format consists of 1 sign bit S (bit 0), 7 characteristic bits C (bits 1 through 7), and 24 mantissa bits M (bits 8 through 31).



Word 1 Word 2

A single-precision floating number N is divided as follows :

- sign S is the sign of mantissa M
- characteristic C equals the base 16 value of exponent E increased by 64 ($0 \leq C \leq 127$).
- mantissa M is made up of 6 hexadecimal digits ($0 \leq M \leq 1 - 2^{-24}$).

Formal definition of the single-precision floating number N is as follows :

- if $N \geq 0$; $N = M \times 16^{C-64}$
- a positive single-precision floating number with zero mantissa M and zero characteristic C is a "true zero".
- a positive single-precision floating number with zero mantissa M and non-zero characteristic C is an "abnormal zero".
- a negative single-precision floating number is represented by the two's

complement of its absolute value ; more precisely, the representation of the absolute value of a negative number N consists of a sign S, a characteristic C, and a mantissa M, which together make up a double word.

The two's complement of this double-word is the representation of N.

Single-precision floating computation instructions work on normalized operands and deliver normalized results.

A positive, single-precision floating number is said to be normalized if its mantissa falls within the inequalities : $1/16 \leq M < 1$.

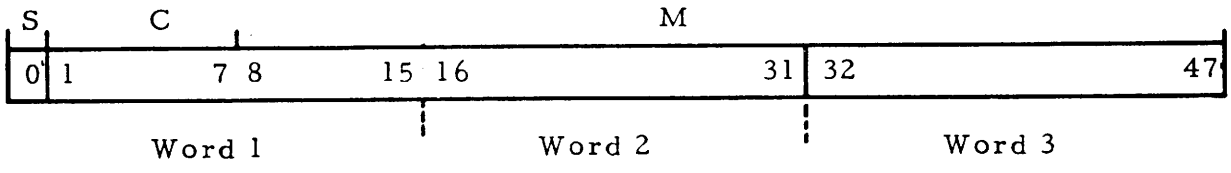
A negative, single-precision floating number is said to be normalized if its two's complement is the representation of a normalized, positive single-precision floating number.

Representation of single-precision, floating formatted numbers :

Decimal number	Hexadecimal value
$+(16^{+63})(1-2^{-24})$	7F FF FF FF
$+(16^{+1})(1/16)$	41 10 00 00
0 (true zero)	00 00 00 00
$-(16^{+1})(1/16)$	BE F0 00 00
$-(16^{+63})(1-2^{-24})$	80 00 00 01

IX-2.3. Double-Precision Floating Formatted Data

The double precision floating format permits representing a decimal number on 3 consecutive MITRA 125 words. This format consists of 1 sign bit S (bit 0), 7 characteristic bits C (bits 1 through 7), and 40 mantissa bits M (bits 8 through 47).



A double-precision floating number N is divided as follows :

- sign S is the sign of mantissa M.
- characteristic C equals the base 16 value of exponent E increased by 64 ($0 \leq C \leq 127$).
- mantissa M is made up of 10 hexadecimal digits ($0 \leq M \leq 1 - 2^{-40}$).

Formal definition of the double-precision floating number N is as follows :

- if $N \geq 0$; $N = M \times 16^{C-64}$
- a positive double-precision floating number with a zero mantissa M and a zero characteristic C is a "true zero".
- a positive double-precision floating number with zero mantissa M and non-zero characteristic C is an "abnormal zero".
- a negative double-precision floating number is represented by the two's complement of its absolute value ; more precisely, the representation of the absolute value of a negative number N consists of a sign S, a characteristic C, and a mantissa M, which together make up a triple-word ; the two's complement of this triple-word is the representation of N.

Double-precision floating point computation instructions work on normalized operands and deliver normalized results.

A positive double-precision floating number is said to be normalized if its mantissa M falls within the inequalities :

$$1/16 \leq M < 1$$

A negative double-precision floating number is said to be normalized if its two's complement is the representation of a normalized positive double-precision floating number.

Representation of double-precision floating formatted numbers :

Decimal number	Hexadecimal value
$+(16^{+63})(1-2^{-24})$	7F FF FF FF
$+(16^{+1})(1/16)$	41 10 00 00
0 (true zero)	00 00 00 00
$-(16^{+1})(1/16)$	BE F0 00 00
$-(16^{+63})(1-2^{-24})$	80 00 00 00

IX-2.4. Character Formatted Data

The character format permits representing a character on a MITRA 125 byte or a character string on n consecutive MITRA 125 bytes.

This character must have an EBCDIC code (Extended Binary Coded Decimal Inter-Change Code) and be recognized by MITRA 125 peripherals.

Printable Character	EBCDIC CODE	Meaning
	40	Space
[	4A	Left Bracket
.	4B	Period
<	4C	Less than
(	4D	Left parenthesis
+	4E	Plus sign
!	4F	Vertical bar
&	50	Ampersand
]	5A	Rigth bracket
$	5B	Dollar sign
*	5C	Asterisk
)	5D	Right parenthesis
;	5E	Semi-colon
^	5F	Logical NOT
-	60	Minus sign, dash
/	61	Slash
,	6B	Comma
%	6C	Percent
_	6D	Underline
>	6E	Greater than
?	6F	Question mark
:	7A	Colon
#	7B	Number sign
@	7C	At sign
'	7D	Apostrophe
=	7E	Equal sign
"	7F	Quotation marks
A	C1	
B	C2	
C	C3	
D	C4	
E	C5	
F	C6	
G	C7	
H	C8	
I	C9	
J	D1	
K	D2	
L	D3	
M	D4	
N	D5	
O	D6	
P	D7	
Q	D8	

R	D9	
/	E0	Inverted diagonal
S	E2	
T	E3	
U	E4	
V	E5	
W	E6	
X	E7	
Y	E8	
Z	E9	
0	F0	
1	F1	
2	F2	
3	F3	
4	F4	
5	F5	
6	F6	
7	F7	
8	F8	
9	F9	

IX-3. Instructions Description

Symbolic Notations Used

A	Accumulator register or its contents
$\overline{A}$	Logical complement of the contents of A
A0-7	Most significant (leftmost) byte of A
A8-15	Least significant (rightmost) byte of A
E	Extension register or its contents
E,A	Extended register made up of accumulator A and Extension register, the most significant bits being in E, or their contents.
X	Index register
Rn	Register n or its contents
L	Local base register or its contents
P	Program counter or its contents
C	Instruction result indicator (Carry)
O	Instruction result indicator (Overflow)
SP	Subroutine mode indicator
PV	Privileged mode indicator
SV	Supervisor mode indicator
PR	Memory protection indicator
MA	Interrupt mask indicator
G	Program segment base register
GL	Program segment length register
Q	Subroutine segment base register
QL	Subroutine segment length register
Z	Data segment base register
ZL	Data segment length register
G'	= G in "Program" mode = Q in "Subroutine" mode

B	= G or Z in extended addressing depending on the base referenced by the logical pointer.
D	Displacement (least significant byte of the instruction extended to one word by a zero most significant byte)
IN	Instruction
(I)	Contents of I
I/J	Address I is relative to base J
(I)/J	Contents of the memory cell with address I/J
N	Operand-computed value
Y	Computed address relative to the beginning of the referenced segment (Y = f (D) where f = addressing function)
Y1	Byte of address Y/G, Y/Q, or Y/Z depending on the operational mode and the type of addressing
Y2	Address word : . if Y is even : Y/G, Y/Q, or Y/Z . if Y is odd : Y-1/G, Y-1/Q, or Y-1/Z
Y1	Absolute address of memory cell containing bytes y1 : (Y1) = y1
Y2	Absolute address of memory cell containing word y2 : (Y2) = y2
I → J	J assumes value I
I ↔ J	J assumes value I and I assumes value J
⊕	Exclusive OR operation
∧	Logical AND operation
∨	Inclusive OR operation
+	Plus
-	Minus
x	Multiply by
/	Divide by

Arithmetic, Algebraic fixed or floating depending on utilization

AAE

Name : A And E in register A

Class : 1' Family SRG

Instruction Code :

Code : Bits 0 - 15

Addressing Mode/ Hexadecimal Code : P / F118

Function

A ∧ E → A

AND the contents of register A and the contents of register E.
The result is stored in register A.

Modified Elements :

- Register A
- Indicators C and O

Indicators C And O After Execution :

A > 0 : C = 0 and O = 0
A < 0 : C = 0 and O = 1
A = 0 : C = 1 and O = 0

Traps : Standard

Example

0 0 1 1 0 0 1 1 | 0 0 1 1 0 0 1 1	Register A before execution
0 1 0 1 0 1 0 1 | 0 1 0 1 0 1 0 1	Register E
0 0 0 1 0 0 0 1 | 0 0 0 1 0 0 0 1	Register A after execution

Name : Add A and X in register X

Class : 1' Family SRG

Instruction Code :

Code : Bits 0 - 15

Addressing Mode / Hexadecimal Code : P / F12A

Function :

A + X → X

The contents of register A are added to the contents of register X. The result is stored in register X.

Modified Elements :

Register X

Traps :

Standard

Name : Add Carry and E in register E

Class : 1' Family SRG

Instruction Code :

Code : Bits 0 - 15

Addressing Mode / Hexadecimal Code : P / F10E

Function :

E + C → E

Value of indicator CARRY is added to the contents of register E.
The result is stored in register E.

Modified Elements :

- Register E
- Indicators C and O

Indicators C And O After Execution :

Carry : C = 1 and O = Don't Care
Overflow : C = Don't Care and O = 1

Traps :

Standard

Name : ADDition

Class : 0

Instruction Code :

Code : Bits 0 - 7
Displacement : Bits 8 - 15

Addressing Mode / Hexadecimal Code :

DL	05
P	25
DG	45
IL, EL	65
IGX, EGX	85
ILX, ELX	A5

Function :

A + y2 → A

Value read in memory at address Y2 is added to the contents of register A. The result is stored in register A.

Modified Elements :

- Register A
- Indicators C and O

Indicators C And O After Execution :

Carry : C = 1 and O = Don't Care
Overflow : C = Don't Care and O = 1

Traps :

Standard

Name : ADdition Memory

Class : 0'

Instruction Code :

Code : Bits 0 - 7
Displacement : Bits 8 - 15

Addressing Mode / Hexadecimal Code :

DL	17
DG	57
IL, EL	77
IGX, EGX	97
ILX, ELX	B7

Function :

y2 + A → y2 and A

The contents of register A are added to the value read in memory at address Y2. The result is stored in the memory word and in register A.

Modified Elements :

- Register A
- Memory locations : y2
- Indicators C and O

Indicators C And O After Execution :

Overflow : C = Don't Care and O = 1
Carry : C = 1 and O = Don't Care

Traps

Standard

Name : Add A with E in register A

Class : 1' Family SRG

Instruction Code :

Code : Bits 0 - 15

Addressing Mode / Hexadecimal Code : P / F122

Function :

A + E → A

The contents of register E are added to the contents of register A. The result is stored in register A.

Modified Elements :

- Register A
- Indicators C and O

Indicators C And O After Execution :

No carry	: C = 0 and O = Don't Care
Carry	: C = 1 and O = Don't Care
No overflow	: C = Don't Care and O = 0
Overflow	: C = Don't Care and O = 1

Traps :

Standard

Name : A Exclusive OR with E in register A

Class : 1' Family SRG

Instruction Code :

Code : Bits 0 - 15

Addressing Mode / Hexadecimal Code : P / F112

Function :

$A \oplus E \rightarrow A$

Exclusive-OR the contents of register A and the contents of register E. The result is stored in register A.

Modified Elements :

- Register A
- Indicators C and O

Indicators C And O After Execution :

A > 0	: C = 0 and O = 0
A < 0	: C = 0 and O = 1
A = 0	: C = 1 and O = 0

Traps :

Standard

Example

0 0 1 1 0 0 1 1 | 0 0 1 1 0 0 1 1	Register A before execution
0 1 0 1 0 1 0 1 | 0 1 0 1 0 1 0 1	Register E
0 1 1 0 0 1 1 0 | 0 1 1 0 0 1 1 0	Register A after execution

Name : A Inclusive-OR with E in register A

Class : 1' Family SRG

Instruction Code :

Code : Bits 0 - 15

Addressing Mode / Hexadecimal Code : P / F116

Function :

$A \vee E \rightarrow A$

Inclusive-OR the contents of register A and the contents of register E. The result is stored in register A.

Modified Elements :

- Register A
- Indicators C and O

Indicators C And O After Execution :

A > 0	: C = 0 and O = 0
A < 0	: C = 0 and O = 1
A = 0	: C = 1 and O = 0

Traps :

Standard

Example

0 0 1 1 0 0 1 1 | 0 0 1 1 0 0 1 1	Register A before execution
0 1 0 1 0 1 0 1 | 0 1 0 1 0 1 0 1	Register E
0 1 1 1 0 1 1 1 | 0 1 1 1 0 1 1 1	Register A after execution

Name : AND

Class : 0

Instruction Code :

Code : Bits 0 - 7

Displacement : Bits 8 - 15

Addressing Mode / Hexadecimal Code :

DL	09
P	29
DG	49
IL, EL	69
IGX, EGX	89
ILX, ELX	A9

Function :

A ∧ y2 → A

AND value read in memory at address Y2 and contents of register A.
The result is stored in register A.

Modified Elements :

- Register A
- Indicators C and O

Indicators C And O After Execution :

A > 0 : C = 0 and O = 0

A < 0 : C = 0 and O = 1

A = 0 : C = 1 and O = 0

Traps : Standard

Example

0 0 1 1 0 0 1 1 | 0 0 1 1 0 0 1 1	Register A before execution
0 1 0 1 0 1 0 1 | 0 1 0 1 0 1 0 1	Y2
0 0 0 1 0 0 0 1 | 0 0 0 1 0 0 0 1	Register A after execution

Name : Add X and A in register A

Class : 1' Family SRG

Instruction Code :

Code : Bits 0 - 15

Addressing Mode / Hexadecimal Code : P / F12C

Function :

A + X → A

The contents of register X are added to the contents of register A. The result is stored in register A.

Modified Elements :

- Register A
- Indicators C and O

Indicators C And O After Execution :

No carry	: C = 0 and O = Don't Care
Carry	: C = 1 and O = Don't Care
No overflow	: C = Don't Care and O = 0
Overflow	: C = Don't Care and O = 1

Traps :

Standard

Name : Branch on A Negative

Class : 2

Instruction Code :

Code : Bits 0 - 7
Displacement : Bits 8 - 15

Addressing Mode / Hexadecimal Code :

RP	C4
RM	CC
IL	D4
IG	DC

Function :

If A < 0 Y → P
If A ≥ 0 P + 2 → P

If the contents of register A are less than 0, computed address Y is loaded in register P which causes execution to continue from address Y.

If the contents of register A are equal to or greater than 0, execution is sequentially continued.

Modified Elements :

Base P register

Traps :

Standard

Name : Branch if A equals Zero

Class : 2

Instruction Code :

Code : Bits 0 - 7
Displacement : Bits 8 - 15

Addressing Mode / Hexadecimal Code :

RP	C5
RM	CD
IL	D5
IG	DD

Function :

If A = 0 Y → P
If A ≠ 0 P + 2 → P

If the contents of register A equal 0, computed address Y is loaded in register P which causes execution to continue from address Y.

If the contents of register A are different from 0, execution is sequentially continued.

Modified Elements :

Base P register

Traps :

Standard

Name : Branch if Carry False

Class : 2

Instruction Code :

Code : Bits 0 - 7

Displacement : Bits 8 - 15

Addressing Mode / Hexadecimal Code :

RP	C3
RM	CB
IL	D3
IG	DB

Function :

If C = 0 Y → P

If C = 1 P + 2 → P

If indicator Carry equals 0, computed address Y is loaded in register P which causes execution to continue from address Y.

If indicator Carry equals 1, execution is sequentially continued.

Modified Elements :

Base P register

Traps :

Standard

Name : Branch if Carry True

Class : 2

Instruction Code :

Code : Bits 0 - 7
Displacement : Bits 8 - 15

Addressing Mode / Hexadecimal Code :

RP	C0
RM	C8
IL	D0
IG	D8

Function :

If C = 1 Y → P
If C = 0 P + 2 → P

If indicator Carry equals 1, computed address Y is loaded in register P which causes execution to continue from address Y.

If indicator Carry equals 0, execution is sequentially continued.

Modified Elements :

Base P register

Traps :

Standard

BE

Name : Branch if Equal

Class : 2

Instruction Code :

Code : Bits 0 - 7

Displacement : Bits 8 - 15

Addressing Mode / Hexadecimal Code :

RP	C0
RM	C8
IL	D0
IG	D8

Function :

If C = 1 Y → P

If C = 0 P + 2 → P

This instruction comes normally after a comparison.

If the first term of the comparison equals the second, computed address Y is loaded in register P which causes execution to continue from address Y.

If the first term of the comparison is different from the second, execution is sequentially continued.

Modified Elements :

Base P register

Traps :

Standard

Miscellaneous :

This instruction is equivalent to a BCT.

Name : Branch if Greater than or Equal to

Class : 2

Instruction Code :

Code : Bits 0 - 7
Displacement : Bits 8 - 15

Addressing Mode / Hexadecimal Code :

RP	C6
RM	CE
IL	D6
IG	DE

Function :

If O = 0 Y → P
If O = 1 P + 2 → P

This instruction comes normally after a comparison.

If the first term of the comparison is greater than or equal to the second, computed address Y is loaded in register P which causes execution to continue from address Y.

If the first term of the comparison is less than the second, execution is sequentially continued.

Modified Elements :

Base P register

Traps :

Standard

Miscellaneous :

This instruction is equivalent to a BOF.

Name : Branch if Less than

Class : 2

Instruction Code :

Code : Bits 0 - 7
Displacement : Bits 8 - 15

Addressing Mode / Hexadecimal Code :

RP	C2
PM	CA
IL	D2
IG	DA

Function :

If O = 1 Y → P
If O = 0 P + 2 → P

This instruction comes normally after a comparison.

If the first term of the comparison is less than the second, computed address Y is loaded in register P which causes execution to continue from address Y.

If the first term of the comparison is equal to or greater than the second, execution is sequentially continued.

Modified Elements :

Base P register

Traps :

Standard

Miscellaneous :

This instruction is equivalent to a BOT.

Name : Branch if Less than Zero

Class : 2

Instruction Code :

Code : Bits 0 - 7

Displacement : Bits 8 - 15

Addressing Mode / Hexadecimal Code :

RP	C2
RM	CA
IL	D2
IG	DA

Function :

If O = 1 Y → P

If O = 0 P + 2 → P

This instruction comes normally after a load instruction.

If the previously loaded value is less than zero, computed address Y is loaded in register P which causes execution to continue from address Y.

If the previously loaded value is equal to or greater than zero, execution is sequentially continued.

Modified Elements :

Base P register

Traps :

Standard

Miscellaneous :

This instruction is equivalent to a BOT.

Name : Branch if Not Equal

Class : 2

Instruction Code :

Code : Bits 0 - 7
Displacement : Bits 8 - 15

Addressing Mode / Hexadecimal Code :

RP	C3
RM	CB
IG	D3
IL	DB

Function :

If C = 0 Y → P
If C = 1 P + 2 → P

This instruction comes normally after a comparison.

If the first term of the comparison is different from the second, computed address Y is loaded in register P which causes execution to continue from address Y.

If the first term of the comparison equals the second, execution is sequentially continued.

Modified Elements :

Base P register

Traps :

Standard

Miscellaneous :

This instruction is equivalent to a BCF.

Name : Branch if Not equal to Zero

Class : 2

Instruction Code :

Code : Bits 0 - 7
Displacement : Bits 8 - 15

Addressing Mode / Hexadecimal Code :

RP	C3
RM	CB
IL	D3
IG	DB

Function :

If C = 0 Y → P
If C = 1 P + 2 → P

This instruction comes normally after a load instruction.

If the previously loaded value is different from zero, computed address Y is loaded in register P which causes execution to continue from address Y.

If the previously loaded value equals zero, execution is sequentially continued.

Modified Elements :

Base P register

Traps :

Standard

Miscellaneous :

This instruction is equivalent to a BCF.

Name : Branch if Overflow False

Class : 2

Instruction Code :

Code : Bits 0 - 7
Displacement : Bits 8 - 15

Addressing Mode / Hexadecimal Code :

RP	C6
RM	CE
IL	D6
IG	DE

Function :

If O = 0 Y → P
If O = 1 P + 2 → P

If indicator Overflow equals 0, computed address Y is loaded in register P which causes execution to continue from address Y.

If indicator Overflow equals 1, execution is sequentially continued.

Modified Elements :

Base P register

Traps :

Standard

Name : Branch if Overflow True

Class : 2

Instruction Code :

Code : Bits 0 - 7
Displacement : Bits 8 - 15

Addressing Mode / Hexadecimal Code :

RP	C2
RM	CA
IL	D2
IG	DA

Function :

If O = 1 Y → P
If O = 0 P + 2 → P

If indicator Overflow equals 1, computed address Y is loaded in register P which causes execution to continue from address Y.

If indicator Overflow equals 0, execution is sequentially continued.

Modified Elements :
Base P register

Traps :

Standard

Name : Branch if Positive or equal to Zero

Class : 2

Instruction Code :

Code : Bits 0 - 7

Displacement : Bits 8 - 15

Addressing Mode / Hexadecimal Code

RP	C6
RM	CE
IL	D6
IG	DE

Function :

If O = 0 Y → P

If O = 1 P + 2 → P

This instruction comes normally after a load instruction.

If the previously loaded value is greater than or equal to zero, computed address Y is loaded in register P which causes execution to continue from address Y.

If the previously loaded value is less than zero, execution is sequentially continued.

Modified Elements :

Base P register

Traps :

Standard

Miscellaneous :

This instruction is equivalent to a BOF.

Name : Branch Unconditional

Class : 2

Instruction Code :

Code : Bits 0 - 7
Displacement : Bits 8 - 15

Addressing Mode / Hexadecimal Code :

RP	C7
RM	CF
IL	D7
IG	DF

Function :

Y → P

Address Y is loaded in register P which causes execution to continue from address Y.

Modified Elements :

Register P

Traps :

Standard

Name : BRanch with indeX

Class : 2

Instruction Code :

Code : Bits 0 - 7
Displacement : Bits 8 - 15

Addressing Mode / Hexadecimal Code :

RP	C1
RM	C9
IL	D1
IG	D9

Function :

RP : P + 2D + 2X → P
RM : P - 2D - 2X → P
IL : (L + D + X) / G' → P
IG : (D + X) / G → P

Computed address Y is loaded in register P which causes execution to continue from address Y.

In indirect mode (IL and IG), indexing is a pre-indexing, i.e. it is executed before indirection.

Remark :

Although BRX can be used in relative mode (RP and RM), normally it is used in indirect mode (IL and IG) to execute multiple branches using an address table.

Modified Elements :

Base P register

Traps : Standard

Name : Branch if equal to Zero

Class : 2

Instruction Code :

Code : Bits 0 - 7
Displacement : Bits 8 - 15

Addressing Mode / Hexadecimal Code :

RP	C0
RM	C8
IL	D0
IG	D8

Function :

If C = 1 Y → P
If C = 0 P + 2 → P

This instruction comes normally after a load instruction.

If the previously loaded value equals zero, computed address Y is loaded in register P which causes execution to continue from address Y.

If the previously loaded value is different from zero, execution is sequentially continued.

Modified Elements :

Base P register

Traps :

Standard

Miscellanesous :

This instruction is equivalent to a BCT.

Name : Compare A left with A right

Class : 1' Family SRG

Instruction Code :

Code : Bits 0 - 15

Addressing Mode / Hexadecimal Code : P / F13A

Function :

This instruction arithmetically compares the contents of the right and left bytes of register A and sets indicators according to result (Bits A_0 and A_8 are not considered as sign bits).

Modified Elements :

Indicators C and O

Indicators C And O After Execution :

$A_{0-7} > A_{8-15}$: C = 0 and O = 0
$A_{0-7} < A_{8-15}$: C = 0 and O = 1
$A_{0-7} = A_{8-15}$: C = 1 and O = 0

Traps :

Standard

Name : Compare A with E

Class : 1' Family SRG

Instruction Code :

Code : Bits 0 - 15

Addressing Mode / Hexadecimal Code : P / F126

Function :

This instruction arithmetically compares the contents of registers A and E and sets indicators according to the result (bits A_0 and E_0 are not considered as sign bits).

Modified Elements :

Indicators C and O

Indicators C And O After Execution :

A > E	: C = 0 and O = 0
A < E	: C = 0 and O = 1
A = E	: C = 1 and O = 0

Traps :

Standard

Name : Change status of Bit K of register A

Class : 1' Family SHC

Instruction Code :

Code : Bits 0 - 11
K : Bits 12 - 15

Addressing Mode / Hexadecimal Code :

PX	EC3
P	FC3

Function :

Indicator C is set according to status of bit K of register A before execution.

Bit K of register A changes status.

Indicator O is set according to the status of register A after execution.

Modified Elements :

- Register A : bit K (where K = N_{12-15})
- indicators C and O

Indicators :

Bit K of A = 0 before execution	: C = 0 and O = Don't Care
Bit K of A = 1 after execution	: C = 1 and O = Don't Care
A ≠ 0 after execution	: C = Don't Care and O = 0
A = 0 after execution	: C = Don't Care and O = 1

Traps :

Standard

Miscellaneous :

User shall make sure that indexing saves the value of IN_{8-11} in field N_{8-11} ; otherwise the instruction executed will not be CBA.

Name : Change status of Bit K of register E

Class : 1' Family SHC

Instruction Code :

Code : Bits 0 - 11
K : Bits 12 - 15

Addressing Mode / Hexadecimal Code :

PX	EC2
P	FC2

Function :

Indicator C is set according to the status of bit K of register E before execution.

Bit K of register E changes status.

Indicator O is set according to the status of register E after execution.

Modified Elements :

- Register E : bit K (where K = N_{12-15})
- Indicators C and O

Indicators :

Bit K of E = 0 before execution	: C = 0 and O = Don't Care
Bit K of E = 1 before execution	: C = 1 and O = Don't Care
E ≠ 0 after execution	: C = Don't Care and O = 0
E = 0 after execution	: C = Don't Care and O = 1

Traps :

Standard

Miscellaneous :

User shall make sure that indexing saves the value of IN_{8-11} in field N_{8-11} ; otherwise the instruction executed will not be CBE (see chapter V).

Name : Copy Complement A

Class : 1' Family SRG

Instruction Code

Code : Bits 0 - 15

Addressing Mode / Hexadecimal Code : P / F110

Function :

$\overline{A} \rightarrow A$

Each bit of register A at 0 or 1 is replaced by its complement, respectively 1 and 0.

Modified Elements :

Register A

Traps :

Standard

Example

| 1 0 1 0 1 0 1 0 | 1 0 1 0 1 0 1 0 | Before execution

| 0 1 0 1 0 1 0 1 | 0 1 0 1 0 1 0 1 | After execution

Name : Copy Complement E

Class : 1' Family SRG

Instruction Code :

Code : Bits 0 - 15

Addressing Mode / Hexadecimal Code : P / F10A

Function

$\overline{E} \rightarrow E$

Each bit of register E at 0 or 1 is replaced by its complement, respectively 1 or 0.

Modified Elements :

Register E

Traps :

Standard

Example :

| 1 0 1 0 1 0 1 0 | 1 0 1 0 1 0 1 0 | Register E before execution

| 0 1 0 1 0 1 0 1 | 0 1 0 1 0 1 0 1 | Register E after execution

Name : Copy Half X

Class : 1' Family SRG

Instruction Code :

Code : Bits 0 - 15

Addressing Mode / Hexadecimal Code : P / F11E

Function :

$X/2 \rightarrow X$

Contents of register X are shifted one bit position to the right ; sign bit (bit 0) is carried. (Divide contents of register X by two).

Modified Elements :

Register X

Traps :

Standard

Name : Call Section Of Subroutine Segment

Class : 1

Instruction Code :

Code : Bits 0 - 7
Displacement : Bits 8 - 15

Addressing Mode / Hexadecimal Code :

DL	3E
PX	EE
P	FE

Function :

Instruction "Call Section" i of subroutine Segment :

- Permits going into subroutine mode (SV = 0, SP = 1) if the calling section is not already executed in this mode.
- Saves elements useful for return : L, P + 2, and indicators in 4 words of a new element of the stack associated with the calling program (P + 2 = address/ G' of instruction following instruction CLQ).
- Branches to address Pi/Q with register L being loaded by Li (Pi and Li are the two words of SRDi).
- Indicators other than SP are not modified except PV which is forced to 1 if bit 15 of Pi = 1, i.e. if the external subroutine section is executed in privileged mode.

Modified Elements :

- Registers L and P
- Indicators SV, SP, PV
- Stack associated with the calling program (ST = ST_0 + 8)

Indicators :

0 → SV
1 → PS
P_{15} → PV

Other indicators are not changed.

Traps :

A length overflow trap occurs if the stack is non-existent (SL = 0).

A Mode Violation trap occurs if the length of the stack associated with the calling program does not permit storing 4 words (ST_0 + 10 > SL) - in this case, the stack is not changed - or if the shared program section is non-existent. (i > maximum number of SRD).

Miscellaneous :

When instructions PUSH, PULL, CLQ, and RTQ operate on the same stack, there must be as many RTQs as CLQs between PUSH and PULL, and as many PULLS as PUSHs between CLQ and RTQ.

If the called section is reentrant, it stores all the variable elements used in the program segment based by G : the program CDS shall begin with a TWB reservation.

A subroutine segment section can be called by a section belonging to the program segment or to the subroutine segment itself.

Name : CalL Section

Class : 1 Instruction not executable in shared program mode.

Instruction Code :

Code : Bits 0 - 7
Displacement : Bits 8 - 15

Addressing Mode / Hexadecimal Code :

DL	38
PX	E8
P	F8

Function :

Instruction "Call section" i of subroutine segment :

- Saves useful elements for return : L, P + 2 in the first two words of the called section LDS (P + 2 = address/G of the instruction following instruction CLS).
- Branches to address Pi/G with register L being loaded by Li (Pi and Li are the two words of the SRDi).

Used Elements :

- User program PRT
- First 2 words of the called section LDS
- Contents of the computed address of instruction CLS ; this address contains the number of the called section.

Communication With Calling Section

There are 3 methods to transfer data between the calling section and the called section :

- through the CDS
- in indirect local or indirect local indexed mode via the second word of LDS
- via register E, A and X.

Modified Elements :

- Registers L and P
- Memory locations : the first 2 words of the LDS of the calling section.

Traps :

Mode violation trap if an attempt to execute instruction CLS in subroutine mode is made.

Miscellaneous :

The CLS operating with called section elements is not reentrant.

Calling an undefined section loads L and P with unpredictable values.

If Pil5 = 1, the user section is not called ; the call is converted into a call to the Monitor's no. 1 section (load overlay branch of missing section).

Name : CoMPare

Class : 0

Instruction Code :

Code : Bits 0 - 7
Displacement : 8 - 15

Addressing Mode / Hexadecimal Code :

DL	0B
P	2B
DG	4B
IL, EL	6B
IGX, EGX	8B
ILX, ELX	AB

Function :

A compared algebraically to y_2

The contents of register A (first term of comparison) are compared algebraically to the value read in memory at address Y2 (2nd term of comparison). The result is stored in the indicators Carry and Overflow.

Modified Elements :

Indicators C and O

Indicators C And O After Execution :

A > y2	: C = 0 and O = 0
A < y2	: C = 0 and O = 1
A = y2	: C = 1 and O = 0

Traps :

Standard

Name : Count Most significant Zero

Class : 1' Family SHC

Instruction Code :

Code : Bits 0 - 15

Addressing Mode / Hexadecimal Code : P / FC40

Function :

Counting in register X the most significant bits at 0 of register A with the first bit at 1 reset to zero.

If all bits of register A are zero, register X is loaded with value &10.

Modified Elements :

- Registers X and A
- Indicators C and O

Indicators :

A = 0 before execution	: C = 0 and O = Don't Care
A ≠ 0 before execution	: C = 1 and O = Don't Care
A ≠ 0 after execution	: C = Don't Care and O = 0
A = 0 after execution	: C = Don't Care and O = 1

Traps :

Standard

Miscellaneous :

If $IN_{11} \neq 0$, instruction CMZ becomes instruction PTY.

Name : Copy Negative A

Class : 1' Family SRG

Instruction Code :

Code : Bits 0 - 15

Addressing Mode / Hexadecimal Code : P / F11C

Function :

$-A \longrightarrow A$

Algebraic complementation (2^{16}'s complement) of the contents of register A. Result is stored in register A.

Modified Elements :

- Register A
- Indicators C and O

Indicators C And O After Execution :

Carry : C = 1 and O = Don't Care
Overflow : C = Don't Care and O = 1

Traps :

Standard

Example :

0 1 0 1 0 1 0 1 | 0 1 0 1 0 1 0 1	Register A before execution
1 0 1 0 1 0 1 0 | 1 0 1 0 1 0 1 1	Register A after execution

Name : Copy negative E

Class : 1' Family SRG

Instruction Code :

Code : Bits 0 - 15

Addressing Mode / Hexadecimal Code : P / F128

Function :

$-E \longrightarrow E$

Algebraic complementation (2^{16}'s complement) of the contents of register E. Result is stored in register E.

Modified Elements :

Register E.

Traps :

Standard

Example :

0 1 0 1 0 1 0 1	0 1 0 1 0 1 0 1	Register E before execution
1 0 1 0 1 0 1 0	1 0 1 0 1 0 1 1	Register E after execution

Name : Copy Negative X

Class : 1' Family SRG

Instruction Code :

Code : Bits 0 - 15

Addressing Mode / Hexadecimal Code : P / F114

Function :

$-X \longrightarrow X$

Algebraic complementation (2^{16}'s complement) of the contents of register X. Result is stored in register X.

Modified Elements :

Register X

Traps :

Standard

Example

0 1 0 1 0 1 0 1	0 1 0 1 0 1 0 1	Register X before execution
1 0 1 0 1 0 1 0	1 0 1 0 1 0 1 1	Register X after execution

Name : ComPare String

Class : 0

Instruction Code :

Code : Bits 0 - 7
Displacement : Bits 8 - 15

Addressing Mode / Hexadecimal Code :

DL	0A
P	2A
DP	4A
IL, EL	6A
IGX, EGX	8A
ILX, ELX	AA

Function :

-Let $|E|$ be the absolute value of the contents of register E for varying from 0 to $|E| - 1$ byte by byte :

if $|E| > 0$: $(A + \alpha)/G$ is compared to Y1
if $|E| < 0$: $(A + \alpha)/Z$ is compared to Y1

-Byte y_1 read in memory at address Y1 is compared successively to bytes of the string whose start address (relatively to G if $E > 0$ and Z if $E < 0$) is contained in register A and whose length equals E .

After execution :

-If character y_1 belongs to the string, register A contains the relative address of the byte found and register E contains the length of the string not processed (as a two's complement when the string is defined relative to Z).

-If character y_1 does not belong to the string, register A contains the relative address of the first character not processed and register E contains value zero.

-This instruction is interruptible between each byte comparison.

Remarks :

If E = 0 initially, only indicators are modified ; they take the status "character

not found".

Byte string is limited to 32 Kbytes.

Modified Elements :

- Registers E, A
- Indicators C and O

Indicators C And O After Execution :

Character found : C = 0 and O = 0
Character not found : C = 0 and O = 1

Traps :

Standard

Name : Call SuperVisor

Class : 1

Instruction Code :

Code : Bits 0 - 7
Displacement : Bits 8 - 15

Addressing Mode / Hexadecimal Code :

DL	37
PX	E7
P	F7

Function :

Instruction "Call Monitor Section" i :

- Saves useful elements for return : L, P + 2, and indicators in the first three words of the calling task program segment CDS (even if the CSV is executed in the external subroutine segment based by Q).
- Forces indicators SV, PR, and PV to 1.
- Branches to monitor section i (see Volume II of this Reference Manual).

Elements Used :

- The first 3 words of the calling user program CDS.
- The contents of computed address of instruction CSV ; this address contains the number of the called monitor section.

Communication With Calling Section

There are two methods to transfer data between the calling user section and the called monitor section :

- in indirect general indexed mode (IGX) by the second word of the CDS ;
- via registers E, A and X.

Modified Elements :

- Registers L and P
- Memory locations : the first 3 words of the calling program CDS

Indicators :

SV is set to 1

PR is set to 1

PV is set to 1

Traps :

A mode violation (VM) trap occurs if the called monitor section i cannot be called in User mode (see Volume II of this Reference Manual).

Miscellaneous :

The program is not interruptible between a CSV and the instruction executed immediately after (to permit, if necessary, immediate masking of interrupts).

To be reentrant, the monitor section stores all the variable elements used in the user program CDS. The user program CDS must begin with a reservation of the TWB used by the called monitor sections.

Name : DeCrement E

Class : 1' Family SRG

Instruction Code :

Code : Bits 0 - 15

Addressing Mode / Hexadecimal Code : P / F126

Function :

E - 1 ⟶ E

The contents of register E are decremented by 1.

Modified Elements :

- Register E
- Indicators C and O

Indicators C And O After Execution :

No carry	: C = 0 and O = Don't Care	
Carry	: C = 1 and O = Don't Care	(Transition from 0 to -1)
No overflow	: C = Don't Care and O = 0	
Overflow	: C = Don't Care and O = 1	(Transition from &8000 to &7FFF)

Traps :

Standard

Name : DeCrement L

Class : 1

Instruction Code :

Code : Bits 0 - 7
Displacement : Bits 8 - 15

Addressing Mode / Hexadecimal Code :

DL	36
PX	E6
P	F6

Function :

$L - y_2 \longrightarrow L$

The value read in memory at address Y2 is subtracted from base L. The result is stored in base L.

Modified Elements :

Register L

Traps :

Standard

Miscellaneous :

An odd computed operand must be used with great care since utilization of indirect or extended addressing with an odd L transforms indirect addressing into extended addressing and vice versa.

Name : DeCrement X

Class : 1

Instruction Code :

Code : Bits 0 - 7
Displacement : Bits 8 - 15

Addressing Mode / Hexadecimal Code :

DL	33
PX	E3
P	F3

Function :

$X - y_2 \longrightarrow X$

The value read in memory at address Y2 is subtracted from register X. The result is stored in register X.

Modified Elements :

- Register X
- Indicators C and O

Indicators C And O After Execution :

Carry : C = 1 and O = Don't Care
Overflow : C = Don't Care and O = 1

Traps :

Standard

Name : DIVision

Class : 0

Instruction Code :

Code : Bits 0 - 7
Displacement : Bits 8 - 15

Addressing Mode / Hexadecimal Code :

DL	08
P	28
DG	48
IL, EL	68
IGX, EGX	88
ILX, ELX	A8

Functions :

E,A / Y2 → A
Remainder → E

Contents of extended accumulator (register E contains the most significant bits, register A contains the least significant bits) is divided by the value read in memory at address Y2. The quotient is stored in register A. The remainder is stored in register E. The remainder sign is the same as the dividend sign except if the remainder is zero in which case the remainder sign is + (positive).

Modified Elements :

- Registers E and A (except if division by zero or result overflow).
- Indicators C and O

Indicators C And O After Execution :

Correct result : C = Don't Care and O = 0
Overflow or Division By
Zero : C = Don't Care and O = 1

Traps :

Standard

Name : Double LoaD

Class : 0'

Instruction Code :

Code : Bits 0 - 7
Displacement : Bits 8 - 15

Addressing Mode / Hexadecimal Code :

DL	10
DG	50
IL, EL	70
IGX, EGX	90
ILX, ELX	B0

Functions :

(Y2) ⟶ E
(Y2 + 2) ⟶ A

Register E is loaded with the value read in memory at address Y2 and register A is loaded with the value read in memory at address Y2 + 2.

Modified Elements :

- Registers E and A
- Indicators C and O

Indicators C And O After Execution :

E > 0 : C = 0 and O = 0
E < 0 : C = 0 and O = 1
E = 0 : C = 1 and O = 0

Traps :

Standard

Name : Double STore

Class : 0'

Instruction Code :

Code : Bits 0 - 7
Displacement : Bits 8 - 15

Addressing Mode / Hexadecimal Code :

DL	16
DG	56
IL, EL	76
IGX, EGX	96
ILX, ELX	B6

Functions :

E $\longrightarrow$ (Y2)
A $\longrightarrow$ (Y2 + 2)
E and A not changed

Contents of register E are stored at address Y2 of main memory, and contents of register A are stored at address Y2 + 2 of main memory.

Modified Elements :

Memory locations : (Y2) and (Y2 + 2)

Traps :

Standard

Name : Exclusive OR

Class : 0

Instruction Code :

Code : Bits 0 - 7
Displacement : Bits 8 - 15

Addressing Mode / Hexadecimal Code :

DL	03
P	23
DG	43
IL, EL	63
IGX, EGX	83
ILX, ELX	A3

Functions :

$A \oplus Y2 \longrightarrow A$

Exclusive OR of value read in main memory at address Y2 and contents of register A. The result is stored in register A.

Modified Elements :

- Register A
- Indicators C and O

Indicators C And O After Execution :

A > 0 : C = 0 and O = 0
A < 0 : C = 0 and O = 1
A = 0 : C = 1 and O = 0

Traps : Standard

Example :

0 0 1 1 0 0 1 1 | 0 0 1 1 0 0 1 1	Register A before execution
0 1 0 1 0 1 0 1 | 0 1 0 1 0 1 0 1	y2
0 1 1 0 0 1 1 0 | 0 1 1 0 0 1 1 0	Register A after execution

Name : Floating ADdition

Class : 0' Requires that a floating point operator be present.

Instruction Code :

Code : Bits 0 - 7
Displacement : Bits 8 - 15

Addressing Mode / Hexadecimal Code :

DL	1A
DG	5A
IL, EL	7A
IGX, EGX	9A
ILX, ELX	BA

Function :

E,A + (Y2) , (Y2 + 2) → E,A

The single-precision, floating point number contained in the double-word at address Y2 in main memory is added to the single-precision floating point number contained in extended register E,A.

The result is normalized and stored in extended register E,A.

Modified Elements :

- Registers E and A
- Indicators C and O

Indicators C And O After Execution :

Result ≠ 0	C = 0 and O = 0
High overflow	C = 0 and O = 1
Result = 0	C = 1 and O = 0
Low overflow	C = 1 and O = 1

Traps :

I/O trap occurs if floating point operator missing.

Miscellaneous :

The result is not rounded-off ; however, if the floating point operator used is a double-precision operator, the operation is performed in double-precision after the mantissa of both operands is filled in with least significant zeros.

High overflow occurs if characteristic C of the result is greater than 127 ; the sign and mantissa of the result are correct, but the characteristic will have the format C - 128.

Low overflow occurs if characteristic C of the result is negative ; the result will have the "true zero" format, i.e., S = C = M = 0.

If the floating point operator is missing, instruction FAD can be simulated using a Monitor module.

Name : Floating ADdition Double precision

Class : 1' Family COV
Requires that a double-precision floating point operator be present.

Instruction Code :

Code : Bits 0 - 15

Addressing Mode / Hexadecimal Code : P / FF32

Function :

OP1 + OP2 → R

Add two double-precision floating point numbers in memory ; the result is stored in memory.
The result is normalized.
There is no rounding off.

Indicators C And O After Execution :

Result ≠	: C = 0 and O = 0
High overflow	: C = 0 and O = 1
Result = 0	: C = 1 and O = 0
Low overflow	: C = 1 and O = 1

Traps :

I/O trap occurs if floating point operator missing.

Miscellaneous :

High overflow occurs if characteristic C of the result is greater than 127. Sign and mantissa of the result are correct, but the characteristic will have format C - 128.

Low overflow occurs if characteristic C of the result is negative : the result will be represented in the "true zero" format, i.e. : S = C = M = 0.

If the double-precision floating point operator is missing, instruction FADD can be simulated using a monitor module.

Name : Floating DiVision

Class : 0' Requires that a floating point operator be present.

Instruction Code :

Code : Bits 0 - 7
Displacement : Bits 8 - 15

Addressing Mode / Hexadecimal Code :

DL	1D
DG	5D
IL, EL	7D
IGX, EGX	9D
ILX, ELX	BD

Function :

E,A / (Y2), (Y2 + 2) → E,A

The single-precision floating point number contained in extended register E,A is divided by the single-precision floating point number contained in the double-word at address Y2 in main memory. The result is stored in extended register E,A.

Modified Elements :

- Registers E and A
- Indicators C and O

Indicators C And O After Execution :

Result ≠ 0	: C = 0 and O = 0
High overflow	: C = 0 and O = 1
Result = 0	: C = 1 and O = 0
Low overflow	: C = 1 and O = 1

Traps :

I/O trap occurs if the floating point operator is missing.

Miscellaneous :

Actual result is between result found and result found plus or minus a least significant bit.

If the divisor is zero, or not normalized, a high overflow occurs and the result is not significant.

If the dividend is zero, or not normalized, the result will be correct but not normalized.

If both operands are normalized, the result is normalized.

If the floating point operator is missing, instruction FDV can be simulated using a monitor module.

Name : Floating DiVision Double precision

Class : 1' Family COV.
Requires that a double-precision floating point operator be present.

Instruction Code :

Code : Bits 0 - 15

Addressing Mode / Hexadecimal Code : P / FF31

Function :

OP1 / OP2 → R

Two double-precision floating point numbers in memory are divided ; the result is stored in memory.

If both operands are normalized, the result is normalized.

Indicators C And O After Execution :

Result ≠ 0	: C = 0 and O = 0
High overflow	: C = 0 and O = 1
Result = 0	: C = 1 and O = 0
Low overflow	: C = 1 and O = 1

Traps :

An I/O trap occurs if the double-precision floating point operator is missing.

Miscellaneous :

If operand 2 (divisor) is zero or is not normalized, a high overflow occurs and the result is not significant.

If operand 1 (dividend) is zero or is not normalized, the result is correct but is not normalized.

Actual result is between the result found and the result found plus or minus a least significant bit.

If the double-precision floating point operator is missing, instruction FDVD can be simulated using a monitor module.

Name : Floating MUltiplication

Class : 0' Requires that a floating point operator be present.

Instruction Code :

Code : Bits 0 - 7
Displacement : Bits 8 - 15

Addressing Mode / Hexadecimal Code :

DL	1C
DG	5C
IL, EL	7C
IGX, EGX	9C
ILX, ELX	BC

Function :

E,A X (Y2) , (Y2 + 2) → E,A

The single-precision floating point number contained in extended register E,A is multiplied by the single-precision floating point number contained in the double-word at address Y2 in main memory. The result is normalized and stored in extended register E,A.

Modified Elements :

- Registers E and A
- Indicators C and O

Indicators C And O After Execution :

Result ≠ 0	: C = 0 and O = 0
High overflow	: C = 0 and O = 1
Result = 0	: C = 1 and O = 0
Low overflow	: C = 1 and O = 1

Traps :

An I/O trap occurs if the floating point operator is missing.

Miscellaneous :

If operands are normalized, the actual result is between the result found and the result found plus or minus a least significant bit.

If the floating point operator is missing, instruction FMU can be simulated using a monitor module.

Name : Floating MUltiplication Double precision

Class : 1' Family COV.
Requires that a double-precision floating point operator be present.

Instruction Code :

Code : Bits 0 - 15

Addressing Mode / Hexadecimal Code : P : FF30

Function :

OP1 x OP2 → R

Two double-precision floating point numbers in memory are multiplied ; the result is stored in memory.

The result is normalized.

Indicators C And O After Execution :

Result ≠ 0	: C = 0 and O = 0
High overflow	: C = 0 and O = 1
Result = 0	: C = 1 and O = 0
Low overflow	: C = 1 and O = 1

Traps :

An I/O trap occurs if the double-precision floating point operator is missing.

Miscellaneous :

If operands are normalized, the actual result is between the result found and the result found plus or minus a least significant bit.

If the double-precision floating point operator is missing, instruction FMUD can be simulated using a monitor module.

Name : Floating SUbtraction

Class : 0' Requires that a floating point operator be present.

Instruction Code :

Code : Bits 0 - 7
Displacement : Bits 8 - 15

Addressing Mode / Hexadecimal Code :

DL	1B
DG	5B
IL, EL	7B
IGX, EGX	9B
ILX, ELX	BB

Function :

E,A - (Y2) , (Y2 + 2) → E,A

The single-precision floating point number contained in the double-word at address Y2 in main memory is subtracted from the single-precision floating point number contained in the extended register E,A.

The result is normalized and stored in extended register E,A.

Modified Elements :

- Registers E and A
- Indicators C and O

Indicators C And O After Execution :

Result ≠ 0	: C = 0 and O = 0
High overflow	: C = 0 and O = 1
Result = 0	: C = 1 and O = 0
Low overflow	: C = 1 and O = 1

Traps :

An I/O trap occurs if the floating point operator is missing.

Miscellaneous :

There is no rounding-off ; however, if the floating point operator used is a double-precision operator, the operation is performed in double-precision after the mantissa of both operands is filled in with least significant zeros.

If the floating operator is missing, instruction FSU can be simulated using a monitor module.

Name : Floating SUbtraction Double precision

Class : 1' Family COV.
Requires that a double-precision floating point operator be present.

Instruction Code :

Code : Bits 0 - 15

Addressing Mode / Hexadecimal Code : P / FF33

Function :

OP1 - OP2 → R

Two double-precision floating point numbers in memory are subtracted. The result is stored in memory.

The result is normalized. There is no rounding-off.

Indicators C And O After Execution :

Result ≠ 0	: C = 0 and O = 0
High overflow	: C = 0 and O = 1
Result = 0	: C = 1 and O = 0
Low overflow	: C = 1 and O = 1

Traps :

An I/O trap occurs if the double-precision floating point operator is missing.

Miscellaneous :

If the double-precision floating point operator is missing, instruction FSUD can be simulated using a monitor module.

Name : HaLT

Class : 1' Family SRG

Instruction Code :

Code : Bits 0 - 15

Addressing Mode / Hexadecimal Code : P / F13E

Function

This instruction halts a program and sets it to wait for an interrupt or a console action.

After the interrupt or console action have occurred (console pushbutton MARCHE (start) or "One Step" pressed), the program is reinitiated and the next instructions are executed.

Modified Elements :

None

Traps :

Standard

Name : InCrement E

Class : 1' Family SRG

Instruction Code :

Code : Bits 0 - 15

Addressing Mode / Hexadecimal Code : P / F124

Function :

E + 1 → E

The contents of register E are incremented by 1.

Modified Elements :

- Register E
- Indicators C and O

Indicators C And O After Execution :

No carry	: C = 0 and O = Don't Care	
Carry	: C = 1 and O = Don't Care	(transition from - 1 to zero)
No overflow	: C = Don't Care and O = 0	
Overflow	: C = Don't Care and O = 1	(transition from &7FFF to &8000)

Traps :

Standard

Name : InCrement L

Class : 1

Instruction Code :

Code : Bits 0 - 7
Displacement : Bits 8 - 15

Addressing Mode / Hexadecimal Code :

DL	35
PX	E5
P	F5

Function :

L + y2 → L

The value read in main memory at address Y2 is added to the contents of base L. The result is stored in base L.

Modified Elements :

Register L

Traps :

Standard

Miscellaneous :

An odd computed operand must be used with great care because the use of an indirect or extended addressing with odd L transforms indirect addressing into extended addressing and vice versa.

Name : InCrement X

Class : 1

Instruction Code :

Code : Bits 0 - 7
Displacement : Bits 8 - 15

Addressing Mode / Hexadecimal Code :

DL	32
PX	E2
P	F2

Function :

X + y2 → X

The value read in main memory at address Y2 is added to the contents of register X. The result is stored in register X.

Modified Elements :

- Register X
- Indicators C and O

Indicators C And O After Execution :

Carry	: C = 1 and O = Don't Care
Overflow	: C = Don't Care and O = 1

Traps :

Standard

Name : Inclusive OR

Class : 0

Instruction Code :

Code : Bits 0 - 7
Displacement : Bits 8 - 15

Addressing Mode / Hexadecimal Code

DL	07
P	27
DG	47
IL, EL	67
IGX, EGX	87
ILX, ELX	A7

Function :

$A \vee y2 \rightarrow A$

Inclusive OR of value read in main memory at address Y2 and the contents of register A. The result is stored in register A.

Modified Elements :

- Register A
- Indicators C and O

Indicators C And O After Execution :

A > 0 : C = 0 and O = 0
A < 0 : C = 0 and O = 1
A = 0 : C = 1 and O = 0

Traps : Standard

Example

0 0 1 1 0 0 1 1 , 0 0 1 1 0 0 1 1	Register A before execution
0 1 0 1 0 1 0 1 , 0 1 0 1 0 1 0 1	y2
0 1 1 1 0 1 1 1 , 0 1 1 1 0 1 1 1	Register A after execution

Name : Load Byte Left

Class : 0

Instruction Code :

Code : Bits 0 - 7
Displacement : Bits 8 - 15

Addressing Mode / Hexadecimal Code :

DL	0D
P	2D
DG	4D
IL, EL	6D
IGX, EGX	8D
ILX, ELX	AD

Function :

$y_1 \rightarrow A_{0-7}$

(A_{8-15}) not changed.

The most significant byte of register A is loaded with the value read in main memory at address Y_1.

The least significant byte of register A remains unchanged.

Modified Elements :

- Register A : A_{0-7}
- Indicators C and O

Indicators C And O After Execution :

A > 0	: C = 0 and O = 0
A < 0	: C = 0 and O = 1
A = 0	: C = 1 and O = 0

Traps :

Standard

Name : Load Byte Right

Class : 0

Instruction Code :

Code : Bits 0 - 7
Displacement : Bits 8 - 15

Addressing Mode / Hexadecimal Code

DL	0E
P	2E
DG	4E
IL, EL	6E
IGX, EGX	8E
ILX, ELX	AE

Functions :

$y_1 \rightarrow A_{8-15}$

$0 \rightarrow A_{0-7}$

The least significant byte of register A is loaded with the value read in main memory at address Y_1.

The most significant byte of register A is reset to zero.

Modified Elements :

- Register A
- Indicators C and O

Indicators C And O After Execution :

$A \neq 0$: C = 0 and O = 0
A = 0 : C = 1 and O = 0

Traps : Standard

Name : Load Byte right X

Class : 0

Instruction Code :

Code : Bits 0 - 7
Displacement : Bits 8 - 15

Addressing Mode / Hexadecimal Code :

DL	0F
P	2F
DG	4F
IL, EL	6F
IGX, EGX	8F
ILX, ELX	AF

Functions :

$y_1 \rightarrow X_{8-15}$

$0 \rightarrow X_{0-7}$

The least significant byte of register X is loaded with the value read in main memory at address Y_1.

The most significant byte of register X is reset to zero.

Modified Elements :

- Register X
- Indicators C and O

Indicators C And O After Execution :

A ≠ 0 : C = 0 and O = 0
A = 0 : C = 1 and O = 0

Traps : Standard

Name : LoaD A

Class : 0

Instruction Code :

Code : Bits 0 - 7
Displacement : Bits 8 - 15

Addressing Mode / Hexadecimal Code :

DL	00
P	20
DG	40
IL, EL	60
IGX, EGX	80
ILX, ELX	A0

Function :

$y_2 \rightarrow A$

Register A is loaded with the value read in main memory at address Y_2.

Modified Elements :

- Register A
- Indicators C and O

Indicators C And O After Execution :

A > 0 : C = 0 and O = 0
A < 0 : C = 0 and O = 1
A = 0 : C = 1 and O = 0

Traps :

Standard

Name : LoaD E

Class : 0

Instruction Code :

Code : Bits 0 - 7
Displacement : Bits 8 - 15

Addressing Mode / Hexadecimal Code :

DL	01
P	21
DG	41
IL, EL	61
IGX, EGX	81
ILX, ELX	A1

Function :

$y_2 \rightarrow E$

Register E is loaded with the value read in main memory at address Y_2.

Modified Elements :

- Register E
- Indicators C and O

Indicators C And O After Execution :

E > 0 : C = 0 and O = 0
E < 0 : C = 0 and O = 1
E = 0 : C = 1 and O = 0

Traps :

Standard

Name : LoaD A with Indicators

Class : 1' Family SRG

Instruction Code :

Code : Bits 0 - 15

Addressing Mode / Hexadecimal Code : P / F134

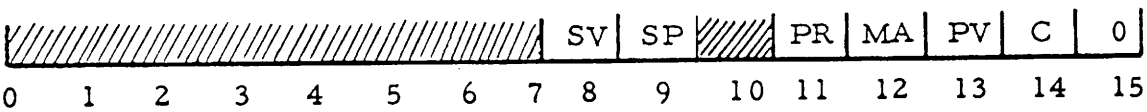
Function :

Load indicators in register A

Modified Elements :

Register A

Register A After Execution :



Traps : standard

Name : LoaD Register

Class : 1

Instruction Code :

Code : Bits 0 - 7

Displacement : Bits 8 - 15

Addressing Mode / Hexadecimal Code :

DL	39
PX	E9
P	F9

Function :

Rn → A where n = (Y)

Rn not changed

Register A is loaded with the value contained in the number n addressed register.

Modified Elements :

- Register A
- Indicators C and O

Indicators C And O After Execution :

A > 0	: C = 0 and O = 0
A < 0	: C = 0 and O = 1
A = 0	: C = 1 and O = 0

Traps :

Standard

Name : LoaD X

Class : 0

Instruction Code :

Code : Bits 0 - 7
Displacement : Bits 8 - 15

Addressing Mode / Hexadecimal Code :

DL	02
P	22
DG	42
IL, EL	62
IGX, EGX	82
ILX, ELX	A2

Function :

$y_2 \rightarrow X$

Register X is loaded with the value read in main memory at address Y_2.

Modified Elements :

- Register X
- Indicators C and O

Indicators C And O After Execution :

X > 0	: C = 0 and O = 0
X < 0	: C = 0 and O = 1
X = 0	: C = 1 and O = 0

Name : Load Effective Address

Class : 0

Instruction Code :

Code : Bits 0 - 7
Displacement : Bits 8 - 15

Addressing Mode / Hexadecimal Code :

DL	04
P	24
DG	44
IL, EL	64
IGX, EGX	84
ILX, ELX	A4

Function :

Load the relative address of operand with respect to the base defined by the addressing mode (and the logical pointer in extended addressing mode) into register A.

If the selected base is Z, register E is loaded with its 2's complement ; otherwise, register E is not changed.

For addressing mode DL : A = L + D

For addressing mode P : A = D

For addressing mode DG : A = D

For addressing mode IL : A = (L + D)/G'

For addressing mode EL : A = $(L + D)_{0-14}$/G', 0 + (L + D + 2)/G'
- . if the referenced logical byte pointer is relative to G, E = E
- . if the referenced logical byte pointer is relative to Z, E = -E

For addressing mode IGX : A = (D)/G + X ; E = E

For addressing mode EGX : A = $(D)_{0-14}$/G, 0 + (D + 2)/G + X
- . if the referenced logical byte pointer is relative to G , E = E
- . if the referenced logical byte pointer is relative to Z , E = -E

For addressing mode ELX : $A = (L + D)_{0-14}/G'$, $0 + (L + D + 2)/G' + X$

. if the referenced logical byte pointer is relative to G, E = E

. if the referenced logical byte pointer is relative to Z, E = -E

Modified Elements :

Register A

Traps :

Standard

Extended Addressing Utilization

LEA used in extended addressing permits loading A and E with values used by byte string instructions.

Name : Load Negative E

Class : 1' Family SRG

Instruction Code :

Code : Bits 0 - 15

Addressing Mode / Hexadecimal Code : P / F11A

Function :

- 1 → E

Value - 1 is loaded in register E

Modified Elements :

Register E

Traps :

Standard

Name : MULtiplication

Class : 0

Instruction Code :

Code : Bits 0 - 7
Displacement : Bits 8 - 15

Addressing Mode / Hexadecimal Code :

DL	0C
P	2C
DG	4C
IL, EL	6C
IGX, EGX	8C
ILX, ELX	AC

Function :

$A \times y_2 \rightarrow E,A$

The contents of register A are multiplied algebraically by the value read in main memory at address Y_2. The result is stored in registers E and A (Register E contains the most significant bits of the result ; register A contains the least significant bits of the result).

Modified Elements :

Registers E and A
Indicators C and O

Indicators C And O After Execution :

E > 0	: C = 0 and O = 0
E < 0	: C = 0 and O = 1
E = 0	: C = 1 and O = 0

Traps : Standard

Name : MoVe byte String

Class : 0'

Instruction Code :

Code : Bits 0 - 7
Displacement : Bits 8 - 15

Addressing Mode / Hexadecimal Code :

DL	1F
DG	5F
IL, EL	7F
IGX, EGX	9F
ILX, ELX	BF

Function :

Let |E| be the absolute value of the contents of register E for α varying from |E| - 1 to 0, byte by byte :

- if E 0 > (A + α)/G $(Y_1 + \alpha)$
- if E 0 < (A + α)/Z $(Y_1 + \alpha)$

The byte string whose start address (relative to base G if E > 0 or relative to base Z if E < 0) is contained in register A and the length in bytes equals |E| and is stored in memory from address Y_1.

After execution, A remains unchanged ; E contains - 1 ; indicators remain unchanged.

This instruction is interruptible between each byte ; register E advances with the number of processed bytes.

Remark : If at the beginning E = 0, after execution only E is modified and equals - 1. String length is limited to 32 Kbytes.

Modified Elements :

Register E
Memory locations Y_1 to $(Y_1 + |E| - 1)$

Traps : Standard

Name : Normalize Byte Pointer

Class : 1' Family SRG

Instruction Code :

Code : Bits 0 - 15

Addressing Mode / Hexadecimal Code : P / F120

Function :

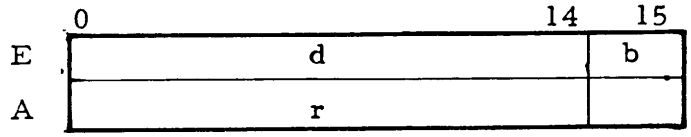
$E + A_{0-14}$, 0 → E

0 → A_{0-14}

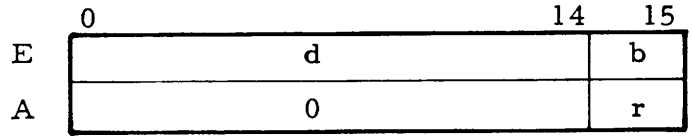
A_{15} → A_{15}

This instruction normalizes a logical byte pointer :

Before Normalization



After Normalization



Modified Elements :

Register E and A
Logical byte pointer

Traps :

Standard

Name : NormaLiZation

Class : 1' Family SHC

Instruction Code :

Code : Bits 0 - 10
Number of shifts : Bits 11 - 15

Addressing Mode / Hexadecimal Code :

PX	ECCn if (n ≤ 15) ECDn if (n > 15)
P	FCCn if (n ≤ 15) FCDn if (n > 15)

Function :

E,A shifted → E,A.
X decremented by the effective number of shifts.

$n = N_{11-15}$ = number of shifts

The contents of extended register E,A is shifted to the left until bit 0 is different from bit 1 or until n bit positions have been shifted.

The contents of register X is decremented by the effective number of shifted bit positions.

Modified Elements :

- Registers E,A and X
- Indicators C and O

Indicators C And O After Execution :

Normalization 01 xx ...	: C = 0 and O = 0
Halt on zero count	: C = 0 and O = 1
Normalization 10 xx ...	: C = 1 and O = 0

Traps :

Standard

Name : PariTY

Class : 1' Family SHC

Instruction Code :

Code : Bits 0 - 10
Number of shifts : Bits 11 - 15

Addressing Mode / Hexadecimal Code :

PX	EC4n if ($n \leq 15$)	EC5n if ($n > 15$)
P	FC4n if ($n \leq 15$)	FC5n if ($n > 15$)

n = number of shifted bit positions ($\neq 0$) = N_{11-15}

Function :

Shifted A → A
E = number of bits at 1 output from A

The contents of register A are circularly shifted n bit positions to the left.

At the end of the instruction, register E contains the number of bits equal to 1 which are output from register A ; C is the last bit output and O is the logical complement of E_{15}, i.e., the parity bit.

Modified Elements :

- Registers A and X
- Indicators C and O

Indicators C And O After Execution :

Last bit output from A is 0	: C = 0 and O = Don't Care
Last bit output from A is 1	: C = 1 and O = Don't Care
No. of bits 1 output from A is odd	: C = Don't Care and O = 0
No. of bits 0 output from A is even	: C = Don't Care and O = 1

Traps : Standard

Miscellaneous :

The user must make sure that indexing saves the value of IN_{8-10} in field N_{8-10} ; otherwise, the instruction executed will not be PTY.

Name : PULL

Class : 1' Family MCB

Instruction Code :

Code : Bits 0 - 15

Addressing Mode / Hexadecimal Code : P / 3B80

Function :

This instruction pulls out of the stack the last element placed in the stack, and stores it in the first n words of the CDS, (n is part of the stack control words), and manages the ST stack top word ($ST = ST_0 - 2 (n + 1)$).

. if the stack becomes empty, indicator C is set to 1
. if the stack was empty, indicators C and O are set to 1.

Modified Elements :

- Stack
- First n words of CDS
- Indicators C and O

Indicators C And O After Execution :

Elements still in stack : C = 0 and O = 0
Stack becomes empty : C = 1 and O = 0
Stack was empty : C = 1 and O = 1

Traps :

A length overflow trap occurs if the stack is non-existent (SL = 0).

Miscellaneous :

This instruction is interruptible between each transferred word.

Since instructions PUSH, PULL, CLQ, and RTQ operate on the same stack, there must be as many RTQs as CLQs between PUSH and PULL and as many PULLs as PUSHs between CLQ and RTQ.

Name : PUSH

Class : 1' Family MCB

Instruction Code :

Code : Bits 0 - 9
Displacement : Bits 10 - 15

Addressing Mode / Hexadecimal Code :

P	3B0
PX	3B4

Function :

This instruction places the first n words of the CDS in the stack and manages the ST stack top word (ST = ST_0 + 2 (n + 1)).

In parameter mode : n = IN_{10-15}

In indexed parameter mode : n = IN_{10-15} + X_{0-15}

If a stack overflow is attempted, stack-in (Push) is not performed and indicator O is set to 1.

Modified Elements :

- Stack
- Indicators C and O

Indicators C And O After Execution :

PUSH (Stack-In) correct	: C = Don't Care and O = 0
Stack overflow attempt	: C = Don't Care and O = 1

Traps :

A length overflow trap occurs if the stack is non-existent (SL = 0).

Miscellaneous :

This instruction is interruptible between each transferred word.

Since instructions PUSH, PULL, CLQ, and RTQ operate on the same stack, there must be as many RTQs as CLQs between PUSH and PULL and as many PULLs as PUSHs between CLQ and RTQ.

Remarks :

The first word of the stack must be initialized to zero by software when the stack is created. It is then used as a working register by hardware during execution of instructions PUSH and PULL.

Name : test and Reset Bit k of register A

Class : 1' Family SHC

Instruction Code :

Code : Bits 0 - 11
K : Bits 12 - 15

Addressing Mode / Hexadecimal Code :

PX	EC7
P	FC7

Function :

Indicator C is set according to the status of bit K of register A before execution (with $K = N_{12-15}$).

Bit K of register A is reset.

Indicator O is set according to the status of register A after execution.

Modified Elements :

- Register A : bit K = 0
- Indicators C and O

Indicators C And O Before And After Execution :

Bit K of A = 0 before execution	: C = 0 and O = Don't Care
Bit K of A = 1 before execution	: C = 1 and O = Don't Care
A ≠ 0 after execution	: C = Don't Care and O = 0
A = 0 after execution	: C = Don't Care and O = 1

Traps : Standard

Miscellaneous :

The user must make sure that indexing saves value IN_{8-11} in field N_{8-11} ; otherwise, the instruction executed will not be KB1.

Name : test and Reset Bit k of register E

Class : 1' Family SHC

Instruction Code :

Code : Bits 0 - 11
K : Bits 12 - 15

Addressing Mode / Hexadecimal Code :

PX	EC6
P	FC6

Function :

Indicator C is set according to the status of bit K of register E before execution (where $K = N_{12-15}$).

Bit K of register E is reset.

Indicator O is set according to the status of register E after execution.

Modified Elements :

- Register E : bit K = 0
- Indicators C and O

Indicators C And O Before And After Execution :

Bit K of E = 0 before execution	: C = 0 and O = Don't Care
Bit K of E = 1 before execution	: C = 1 and O = Don't Care
E ≠ 0 after execution	: C = Don't Care and O = 0
E = 0 after execution	: C = Don't Care and O = 1

Traps :

Standard

Miscellaneous :

The user must make sure that indexing saves the value of IN_{8-11} in field N_{8-11} ; otherwise, the instruction executed will not be RBE.

Name : Return from external subroutine section

Class : 1' Family SRG (instruction executable only in shared program mode)

Instruction Code :

Code : Bits 0 - 15

Addressing Mode / Hexadecimal Code : P / F100

Function :

Instruction "Return From External Section" executed in a shared program section called by CLQ restores indicator SP saved in the stack at the time of the CLQ ; if indicator PV = 0, the other indicators are not changed ; if indicator PV = 1, the indicators saved in the stack are restored except for indicators C and O which are not changed.

Instruction RTQ restores registers P and L with values saved in the stack at the time of the CLQ and updates the stack top ($ST = ST_0 - 8$).

Modified Elements :

- Base L and P registers
- Indicators
- Stack associated with the calling program

Traps :

A length overflow trap occurs if the stack is non-existent (SL = 0).

A mode violation trap occurs if the stack is empty.

Remark :

Instruction RTS has the same code as RTQ in program mode.

Miscellaneous :

Since instructions PUSH, PULL, CLQ, and RTQ operate on the same stack, there must be as many RTQs as CLQs between PUSH and PULL and as many PULLs as PUSHs between CLQ and RTQ.

Name : Return Section

Class : 1' Family SRG

Instruction Code :

Code : Bits 0 - 15

Addressing Mode / Hexadecimal Code : P / F100

Function :

Instruction "Return From Internal Section" executed in a program segment section called by CLS restores registers P and L with the values saved in the first two words of the LDS of the current section.

Modified Elements :

Base L and P registers

Traps :

Standard

Remark :

Instruction RTQ has the same code as RTS in subroutine mode.

Name : Shift Arithmetic Double

Class : 1' Family SHR

Instruction Code :

Code : Bits 0 - 10
Number of shifts : Bits 11 - 15

Addressing Mode / Hexadecimal Code :

PX	E 04n if (n ≤ 15)	E 05n if (n > 15)
P	F 04n if (n ≤ 15)	F 05n if (n > 15)

Function :

E,A shifted → E,A
n = number of shifts = N_{11-15}

The contents of extended register E,A are shifted arithmetically n bit positions to the right ; bit 0 (sign bit) is carried at each shift.

Modified Elements :

- Registers E and A
- Indicators C and O

Indicators C And O After Execution :

Last bit output from A : C = 0/1 and O = Don't Care

Traps :

Standard

Miscellaneous :

The user must make sure that indexing saves the value of IN_{8-10} in field N_{8-10} ; otherwise, the instruction executed will not be SAD.

Name : Shift Arithmetic Simple

Class : 1' Family SHR

Instruction Code :

Code : Bits 0 - 10
Number of shifts : Bits 11 - 15

Addressing Mode / Hexadecimal Code :

PX	E0An if (n ≤ 15) E0Bn if (n > 15)
P	F0An if (n ≤ 15) F0Bn if n > 15)

Function :

A shifted → A
n = number of shifts : N_{11-15}

The contents of register A are shifted arithmetically n bit positions to the right. Bit 0 (sign bit) is carried at each shift.

Modified Elements :

- Register A
- Indicators C and O

Indicators C And O After Execution :

Last bit output from A : C = 0/1 and O = Don't Care

Traps :

Standard

Miscellaneous :

The user must make sure that indexing saves the value of IN_{8-10} in field N_{8-10} ; otherwise, the instruction executed will not be SAS.

Name : Subtract A from X

Class : 1' Family SRG

Instruction Code :

Code : Bits 0 - 15

Addressing Mode / Hexadecimal Code : P / F12E

Function :

X - A → X

The contents of register A are subtracted from the contents of register X.

Modified Elements :

Register X

Traps :

Standard

Name : test and Set Bit K of register A

Class : 1' Family SHC

Instruction Code :

Code : Bits 0 - 11

K : Bits 12 - 15

Addressing Mode / Hexadecimal Code :

PX	ECF
P	FCF

Function :

Indicator C is set according to the status of bit K of register A (with $K = N_{12-15}$).
Bit K of register A is set.
Indicator O is set according to the status of register A.

Modified Elements :

- Register A : bit K of register A
- Indicators C and O

Indicators C And O Before And After Execution :

Bit K of A = 0 before execution	: C = 0 and O = Don't Care
Bit K of A = 1 before execution	: C = 1 and O = Don't Care
(A) ≠ 0 after execution	: C = Don't Care and O = 0
(A) = 0 after execution	: C = Don't Care and O = 1

Traps :

Standard

Miscellaneous :

The user must make sure that indexing saves the value of IN_{8-11} in field N_{8-11} ; otherwise the instruction executed will not be SBA.

Name : test and Set Bit K of register E

Class : 1' Family SHC

Instruction Code :

Code : Bits 0 - 11
K : Bits 12 - 15

Addressing Mode / Hexadecimal Code :

PX	ECE
P	FCE

Function :

Indicator C is set according to the status of bit K of register E (with K = N_{12-15}).
Bit K of register E is set.
Indicator O is set according to the status of register E.

Modified Elements :

- Register E : bit K of register E
- Indicators C and O

Indicators C And O Before And After Execution :

Bit K of E = 0 before execution	: C = 0 and O = Don't Care
Bit K of E = 1 before execution	: C = 1 and O = Don't Care
(E) ≠ 0 after execution	: C = Don't Care and O = 0
(E) = 0 after execution	: C = Don't Care and O = 1

Traps :

Standard

Miscellaneous :

The user must make sure that indexing saves the value of IN_{8-11} in field N_{8-11} ; otherwise the instruction executed will not be SBE.

Name : Store Byte Left

Class : 0'

Instruction Code :

Code : Bits 0 - 7
Displacement : Bits 8 - 15

Addressing Mode / Hexadecimal Code :

DL	14
DG	54
IL, EL	74
IGX, EGX	94
ILX, ELX	B4

Function :

$A_{0-7} \rightarrow Y_1$

(A) unchanged

Left-byte contents (most significant byte) of register A are stored at address Y_1 of main memory.

Modified Elements :

Memory locations Y_1

Traps :

Standard

Name : Store Byte Right

Class : 0'

Instruction Code :

Code : Bits 0 - 7
Displacement : Bits 8 - 15

Addressing Mode / Hexadecimal Code :

DL	15
DG	55
IL, EL	75
IGX, EGX	95
ILX, ELX	B5

Function :

$A_{8-15} \rightarrow Y_1$

(A) unchanged

Right-byte contents (least significant byte) of register A are stored at address Y_1 of main memory.

Modified Elements :

Memory locations Y_1

Traps :

Standard

Name : Shift Left Circular Double

Class : 1' Family SHR

Instruction Code :

Code : Bits 0 - 10
Number of shifts : Bits 11 - 15

Addressing Mode / Hexadecimal Code :

PX	E06n if (n ≤ 15) E07n if (n > 15)
P	F06n if (n ≤ 15) F07n if (n > 15)

Function :

E,A shifted → E,A
n = number of shifts = N_{11-15}

The contents of extended register E,A are shifted circularly n bit positions to the left ; bit 31 follows bit 0.

Modified Elements :

- Registers E and A
- Indicators C and O

Indicators C And O After Execution :

Last bit output from E : C = 0/1 and O = Don't Care

Traps :

Standard

Miscellaneous :

The user must make sure that indexing saves the value of IN_{8-10} in field N_{8-10} ; otherwise, the instruction executed will not be SLCD.

Name : Shift Left Circular Single

Class : 1' Family SHR

Instruction Code :

Code : Bits 0 - 10
Number of shifts : Bits 11 - 15

Addressing Mode / Hexadecimal Code :

PX	E08n if (n ≤ 15)	E09n if (n > 15)
P	F08n if (n ≤ 15)	F09n if (n > 15)

Function :

A shifted → A
n = number of shifts = N_{11-15}

The contents of register A are shifted circularly n bit positions to the left ; bit 15 follows bit 0.

Modified Elements :

- Register A
- Indicators C and O

Indicators C And O After Execution :

Last bit output from A : C = 0/1 and O = Don't Care

Traps :

Standard

Miscellaneous :

The user must make sure that indexing saves the value of IN_{8-10} in field N_{8-10} ; otherwise, the instruction executed will not be SLCS.

Name : Shift Left Logical Double

Class : 1' Family SHC

Instruction Code :

Code : Bits 0 - 10
Number of shifts : Bits 11 - 15

Addressing Mode / Hexadecimal Code :

PX	EC0n if (n ≤ 15) EC1n if (n > 15)
P	FC0n if (n ≤ 15) FC1n if (n > 15)

Function :

E,A shifted → E,A
n = number of shifts = N_{11-15}

The contents of extended register E,A are shifted n bit positions to the left. Least significant bits are filled in with zeros.

Modified Elements :

- Register A
- Indicators C and O

Indicators C And O After Execution :

Last bit output from E : C = 0/1 and O = Don't Care
If shift zero position : C = Don't Care and O = Don't Care

Traps :

Standard

Miscellaneous :

The user must make sure that indexing saves the value of IN_{8-10} in field N_{8-10} ; otherwise, the instruction executed will not be SLLD.

Name : Shift Left Logical Single

Class : 1' Family SHR

Instruction Code :

Code : Bits 0 - 10
Number of shifts : Bits 11 - 15

Addressing Mode / Hexadecimal Code :

PX	E00n if (n ≤ 15)	E01n if (n > 15)
P	F00n if (n ≤ 15)	F01n if (n > 15)

Function :

A shifted → A
n = number of shifts = N_{11-15}

The contents of register A are shifted n bit positions to the left. Least significant bits are filled in with zeros.

Modified Elements :

- Register A
- Indicators C and O

Indicators C And O After Execution :

Last bit output from A : C = 0/1 and O = Don't Care

Traps :

Standard

Miscellaneous :

The user must make sure that indexing saves the value of IN_{8-10} in field N_{8-10} ; otherwise, the instruction executed will not be SLLS.

Name : Store Program Address

Class : 0'

Instruction Code :

Code : Bits 0 - 7
Displacement : Bits 8 - 15

Addressing Mode / Hexadecimal Code :

DL	18
DG	58
IL, EL	78
IGX, EGX	98
ILX, ELX	B8

Function :

$P + 4 \rightarrow y_2$

The current address (contents of register P) plus four is stored at address Y_2 in main memory.

Modified Elements :

Memory locations : Y_2

Traps :

Standard

Miscellaneous :

This instruction is normally followed by a branch to a subroutine sequence ; the stored address being the return address.

Name : Shift Right Circular Double

Class : 1' Family SHR

Instruction Code :

Code : Bits 0 - 10
Number of shifts : Bits 11 - 15

Addressing Mode / Hexadecimal Code :

PX	E0En if ($n \leq 15$)	E0Fn if ($n > 15$)
P	F0En if ($n \leq 15$)	F0Fn if ($n > 15$)

Function :

E,A shifted → E,A
n = number of shifts = N_{11-15}

The contents of extended register E,A are shifted circularly n bit positions to the right ; bit 0 follows bit 31.

Modified Elements :

- Registers E and A
- Indicators C and O

Indicators C And O After Execution :

Last bit output from A : C = 0/1 and O = Don't Care

Traps :

Standard

Miscellaneous :

The user must make sure that indexing saves the value of IN_{8-10} in field N_{8-10} ; otherwise, the instruction executed will not be SRCD.

Name : Shift Right Circular Single

Class : 1' Family SHR

Instruction Code :

Code : Bits 0 - 10
Number of shifts : Bits 11 - 15

Addressing Mode / Hexadecimal Code :

PX	E02n if ($n \leq 15$)	E03n if ($n > 15$)
P	F02n if ($n \leq 15$)	F03n if ($n > 15$)

Function :

A circularly shifted → A
n = number of shifts = N_{11-15}

The contents of register A are shifted circularly n bit positions to the right ; bit 0 follows bit 15.

Modified Elements :

- Register A
- Indicators C and O

Indicators C And O After Execution :

Last bit output from A : C = 0/1 and O = Don't Care

Traps :

Standard

Miscellaneous :

The user must make sure that indexing saves the value of IN_{8-10} in field N_{8-10} ; otherwise, the instruction executed will not be SRCS.

Name : Shift Right Logical Double

Class : 1' Family SHC

Instruction Code :

Code : Bits 0 - 10
Number of shifts : Bits 11 - 15

Addressing Mode / Hexadecimal Code :

PX	EC8n if ($n \leq 15$)	EC9n if ($n > 15$)
P	FC8n if ($n \leq 15$)	FC9n if ($n > 15$)

$n = IN_{11-15}$ = number of bit positions to be shifted.

Function :

E,A shifted → E,A
$n = N_{11-15}$

The contents of extended register E,A are shifted n bit positions to the right ; least significant bits are filled in with zeros.

Modified Elements :

- Registers E and A
- Indicators C and O

Indicators C And O After Execution :

Last bit output from A : C = 0/1 and O = Don't Care
If shift of bit zero : C = Don't Care and O = Don't Care

Traps :

Standard

Miscellaneous :

The user must make sure that indexing saves the value of IN_{8-10} in field N_{8-10} ; otherwise, the instruction executed will not be SRLD.

Name : Shift Right Logical Single

Class : 1' Family SHR

Instruction Code :

Code : Bits 0 - 10
Number of shifts : Bits 11 - 15

Addressing Mode / Hexadecimal Code :

PX	E0Cn if (n ≤ 15)	E0Dn if (n > 15)
P	F0Cn if (n ≤ 15)	F0Dn if (n > 15)

Function :

A shifted → A
n = number of shifts = N_{11-15}

The contents of register A are shifted n bit positions to the right. The most significant bits are filled in with zeros.

Modified Elements :

- Register A
- Indicators C and O

Indicators C And O After Execution :

Last bit output from A : C = 0/1 and O = Don't Care

Traps :

Standard

Miscellaneous :

The user must make sure that indexing saves the value of IN_{8-10} in field N_{8-10} ; otherwise, the instruction executed will not be SRLS.

Name : Save and Reset Parity

Class : 1' Family SRG

Instruction Code :

Code : Bits 0 - 15

Addressing Mode / Hexadecimal Code : P / F13C

Function :

$A_{15} \rightarrow X_{15}$ $\quad X_{0-14} \rightarrow X_{0-14}$

$0 \rightarrow A_{15}$ $\quad A_{0-14} \rightarrow A_{0-14}$

Bit 15 of register A is saved in bit 15 of register X and bit 15 of register A is reset.

Modified Elements :

A_{15} and X_{15}

Other bits of A and X remain unchanged.

Traps :

Standard

Example :

0 0 0 0 0 0 0 0 | 1 1 1 1 1 1 1 1	Register A before execution
1 1 1 1 1 1 1 1 | 0 0 0 0 0 0 0 0	Register X before execution
0 0 0 0 0 0 0 0 | 1 1 1 1 1 1 1 0	Register A after execution
1 1 1 1 1 1 1 1 | 0 0 0 0 0 0 0 1	Register X after execution

Name : STore A

Class : 0'

Instruction Code :

Code : Bits 0 - 7
Displacement : Bits 8 - 15

Addressing Mode / Hexadecimal Code :

DL	11
DG	51
IL, EL	71
IGX, EGX	91
ILX, ELX	B1

Function :

$A \rightarrow y_2$

A unchanged

The contents of register A are stored at address Y_2 in main memory.

Modified Elements :

Memory locations : Y_2

Traps :

Standard

Name : STore E

Class : 0'

Instruction Code :

Code : Bits 0 - 7

Displacement : Bits 8 - 15

Addressing Mode / Hexadecimal Code :

DL	12
DG	52
IL, EL	72
IGX, EGX	92
ILX, ELX	B2

Function :

$E \rightarrow y_2$

E unchanged

The contents of register E are stored at address Y_2 in main memory.

Modified Elements :

Memory locations : Y_2

Traps :

Standard

Name : STore Index with A

Class : 1' Family SRG

Instruction Code :

Code : Bits 0 - 15

Addressing Mode / Hexadecimal Code : P / F132

Function :

$A_{15} \rightarrow O$

$A_{14} \rightarrow C$

This instruction stores bits 15 and 14 of register A in indicators O and C. Other indicators are not changed.

Modified Elements :

Indicators C and O

Traps :

Standard

Name : STore Selective A

Class : 0'

Instruction Code :

Code : Bits 0 - 7
Displacement : Bits 8 - 15

Addressing Mode / Hexadecimal Code :

DL	19
DG	59
IL, EL	79
IGX, EGX	99
ILX, ELX	B9

Function :

$[y_2 \wedge \overline{E}] \vee [A \wedge E] \rightarrow y_2$

Bits of the word located at address Y_2 in main memory corresponding to bits set at one of register E are loaded with the corresponding bits of register A. Other bits of the word are not changed.

Example :

0 0 1 0 0 0 0 0 | 0 1 0 0 0 0 0 0	y_2	
1 1 0 0 0 1 0 0 | 1 1 1 1 0 0 1 1	E	before execution
0 1 1 1 0 1 1 0 | 0 0 1 1 1 1 0 1	A	
0 1 1 0 0 1 0 0 | 0 0 1 1 0 0 0 1	y_2	after execution

Modified Elements :

- Memory locations : Y_2
- Indicators C and O

Indicators C And O After Execution :

$y_2 > 0$: C = 0 and O = 0
$y_2 < 0$: C = 0 and O = 1
$y_2 = 0$: C = 1 and O = 0

Traps : Standard

Name : STore X

Class : 0'

Instruction Code :

Code : Bits 0 - 7

Displacement : Bits 8 - 15

Addressing Mode / Hexadecimal Code :

DL	13
DG	53
IL, EL	73
IGX, EGX	93
ILX, ELX	B3

Function :

$X \rightarrow y_2$

X unchanged

The contents of register X are stored at address Y_2 in main memory.

Modified Elements :

Memory locations : Y_2

Traps :

Standard

Name : SUBtraction

Class : 0

Instruction Code :

Code : Bits 0 - 7
Displacement : Bits 8 - 15

Addressing Mode / Hexadecimal Code :

DL	06
P	26
DG	46
IL, EL	66
IGX, EGX	86
ILX, ELX	A6

Function :

$A - y_2 \rightarrow A$

Value read in memory at address Y_2 is subtracted from the contents of register A. The result is stored in register A.

Modified Elements :

- Register A
- Indicators C and O

Indicators C And O After Execution :

Carry : C = 1 and O = Don't Care
Overflow : C = Don't Care and O = 1

Traps :

Standard

Name : Test Bit K of register A and register A

Class : 1' Family SHC

Instruction Code :

Code : Bits 0 - 11
K : Bits 12 - 15

Addressing Mode / Hexadecimal Code :

PX	ECB
P	FCB

Function :

Indicator C is set according to the status of bit K of register A (with K = N_{12-15}).
Indicator O is set according to the status of register A.

Modified Elements :

Indicators C and O

Indicators C And O After Execution :

Bit K of A = 0	: C = 0 and O = Don't Care
Bit K of A = 1	: C = 1 and O = Don't Care
A ≠ 0	: C = Don't Care and O = 0
A = 0	: C = Don't Care and O = 1

Traps :

Standard

Miscellaneous :

The user must make sure that indexing saves the value of IN_{8-11} in field N_{8-11} ; otherwise, the instruction executed will not be TBA.

Name : Test Bit K of register E and register E

Class : 1' Family SHC

Instruction Code :

Code : Bits 0 - 11

K : Bits 12 - 15

Addressing Mode / Hexadecimal Code :

PX	ECA
P	FCA

Function :

Indicator C is set according to the status of bit K of register E (with $K = N_{12-15}$).
Indicator O is set according to the status of register E.

Modified Elements :

Indicators C and O

Indicators C And O After Execution :

Bit K of E = 0	: C = 0 and O = Don't Care
Bit K of E = 1	: C = 1 and O = Don't Care
$E \neq 0$	: C = Don't Care and O = 0
E = 0	: C = Don't Care and O = 1

Traps :

Standard

Miscellaneous :

The user must make sure that indexing saves the value of IN_{8-11} in field N_{8-11} ; otherwise, the instruction executed will not be TBE.

Name : TEst and Set

Class : 1

Instruction Code :

Code : Bits 0 - 7
Displacement : Bits 8 - 15

Addressing Mode / Hexadecimal Code :

DL	3D
PX	ED
P	FD

Function :

This instruction permits reading a memory location and forcing it to zero without being interrupted.

Initial contents of the memory cell are stored in register A with indicator setting. The protection bit is not affected.

For addressing mode P :

if D is even : (D)/G → A and 0 → (D)/ G
if D is odd : (D)/Z → A and 0 → (D)/Z

For addressing mode PX :

if D + X is even : (D + X)/G → A and 0 → (D + X)/G
if D + X is odd : (D + X)/Z → A and 0 → (D + X)/Z

For addressing mode DL :

if (L + D)15/G' = 0 : (L + D)/G → A and 0 → (L + D)/G
if (L + D)15/G' = 1 : (L + D)/Z → A and 0 → (L + D)/Z

Modified Elements :

- Register A
- Memory location Y
- Indicators C and O

Indicators C And O After Execution :

A > 0	: C = 0 and O = 0
A < 0	: C = 0 and O = 1
A = 0	: C = 1 and O = 0

Traps :

Standard

Name : TRanslate String

Class : 0'

Instruction Code :

Code : Bits 0 - 7
Displacement : Bits 8 - 15

Addressing Mode / Hexadecimal Code :

DL	1E
DG	5E
IL, EL	7E
IGX, EGX	9E
ILX, ELX	BE

Function :

Let |E| be the absolute value of the contents of register E for α varying between 0 and |E| - 1, byte by byte.

if E > 0 : (A + α)/G transcoded → (A + α)/G
if E < 0 : (A + α)/Z transcoded → (A + α)/Z

At the end of transfer, 0 → E

Let code O (Origin) and code R (Result) as well as a 256 consecutive byte transcoding table be located at computed address Y_1 and be constructed as follows :

Address Relative To Beginning Of Table (hexadecimal)	Contents
00	Value in code R corresponding to value 00 in code O
01	Value in code R corresponding to value 01 in code O
FF	Value in code R corresponding to value FF in code O

The string whose address (relative to G if E > 0 or relative to Z if E < 0) is located in A and whose length equals |E| is converted byte by byte using the transcoding table addressed by instruction TRS (computed address Y_1) ; the resulting string covers byte by byte the initial string.

The creation of the transcoding table is the user's responsibility.

After Execution :

A contains the relative address of the first non-transcoded byte ; E contains 0.

This instruction is interruptible between each word ; registers A and E advance according to the number of bytes processed.

Remark :

If at start E = 0, the instruction is ineffective.
Maximum string length is 32 Kbytes.

Modified Elements :

- Registers A and E
- Memory locations from (A)/B to (A + E - 1)/B

Traps :

Standard

Name : TeSt X

Class : 1' Family SRG

Instruction Code :

Code : Bits 0 - 15

Addressing Mode / Hexadecimal Code : P / F130

Function :

Test the contents of register X and set indicators C and O according to the result.

Modified Elements :

Indicators C and O

Indicators C And O After Execution :

X > 0 : C = 0 and O = 0
X < 0 : C = 0 and O = 1
X = 0 : C = 1 and O = 0

Traps :

Standard

Name : eXchange left byte of A with right byte of A

Class : 1' Family SRG

Instruction Code :

Code : Bits 0 - 15

Addressing Mode / Hexadecimal Code : P / F108

Function :

$A_{0-7} \leftrightarrow A_{8-15}$

The most significant byte (left) of register A and the least significant byte (right) of register A are exchanged.

Modified Elements :

Register A

Traps :

Standard

XAE

Name : eXchange A and E

Class : 1' Family SRG

Instruction Code :

Code : Bits 0 - 15

Addressing Mode / Hexadecimal Code : P / F102

Function :

Name : eXchange A and X

Class : 1' Family SRG

Instruction Code :

Code : Bits 0 - 15

Addressing Mode / Hexadecimal Code : P / F104

Function :

$A \leftrightarrow X$

The contents of register A and the contents of register X are exchanged.

Modified Elements :

Registers A and X

Traps :

Standard

Name : eXchange E and X

Class : 1' Family SRG

Instruction Code :

Code : Bits 0 - 15

Addressing Mode / Hexadecimal Code : P / F106

Function :

$E \leftrightarrow X$

The contents of register E and the contents of register X are exchanged.

Modified Elements :

Registers E and X

Traps :

Standard

DAD

Name : Double length fixed point ADdition

Class : 1' Family COV

Instruction Code :

Code : Bits 0 - 7
Displacement : Bits 8 - 15

Addressing Mode / Hexadecimal Code : P / FFE0

Function :

$$E,A + (y_2, y_2 + 2) \rightarrow E,A$$

Modified Elements :

- Registers E and A
- Indicators :
  - C = 1 Carry
  - O = 1 Overflow

Traps :

I/O trap if the Fast Arithmetic Operator is missing.

Name : Double length fixed point DiVision

Class : 1' Family COV

Instruction Code :

Code : Bits 0 - 7
Displacement : Bits 8 - 15

Addressing Mode / Hexadecimal Code : P / FFD0

Function :

$E,A \; / \; (y_2, y_2 + 2) \rightarrow E,A$

Modified Elements :

- Registers E and A
- Indicators :
  - C = 1 Division is 0
  - O = 1 Division is smaller than dividend

Traps :

I/O trap if the Fast Arithmetic Operator is missing.

Name : Double length fixed MUltiplication

Class : 1' Family COV

Instruction Code :

Code : Bits 0 - 7
Displacement : Bits 8 - 15

Addressing Mode / Hexadecimal Code : P / FFC0

Function :

$$E,A \times (y_2, y_2 + 2) \rightarrow E,A$$

Modified Elements :

- Registers E and A
- Indicators :
  - C = 1 Negative Product
  - O = 1 Overflow

Traps :

I/O trap if the Fast Arithmetic Operator is missing.

Name : Double length fixed point SUbtraction

Class : 1' Family COV

Instruction Code :

Code : Bits 0 - 7
Displacement : Bits 8 - 15

Addressing Mode / Hexadecimal Code : P / FFF0

Function :

$E,A - (y_2, y_2 + 2) \rightarrow E,A$

Modified Elements :

- Registers E and A
- Indicators :
  C = 1 Carry
  O = 1 Overflow

Traps :

I/O trap if the Fast Arithmetic Operator is missing.

Name : Floating point NormaliZation

Class : 1' Family COV

Instruction Code :

Code : Bits 0 - 7
Displacement : Bits 8 - 15

Addressing Mode / Hexadecimal Code : P / FF38

Function :

Contents of registers E,A are normalized.
AS long as $|E_{8-11}|$ is different from 0, the mantissa is shifted 4 bits to the left and the characteristic is decremented by 1.

Modified Elements :

- Registers E and A
- Indicators :

Result ≠ 0	: C = 0 and O = 0
Result = 0 (i.e. mantissa = 0)	: C = 1 and O = 0
Overflow (low)	: C = 1 and O = 1

cii COMPAGNIE INTERNATIONALE
POUR L'INFORMATIQUE
DIVISION MILITAIRE SPATIALE ET AERONAUTIQUE
10 - 12 Avenue de l'Europe - 78140 VELIZY (France)
Tél. 946.96.70