mitra 15

compagnie internationale
pour l'informatique

Reference manual

Contents

General	1-1
Introduction	1-1
MITRA 15 main features	1-1
Models MITRA 15/20, MITRA 15/30	1-3
MITRA 15 operating system	1-8
Applications	1-11
GENERAL LAYOUT	2-1
Core memory	2-1
Processing units	2-1
Micro-programmed ROM (or micro-processor)	2-2
Registers	2-4
Logical and arithmetical operator	2-4
Indicators	2-5
Communication with the environment	2-6
Interrupts, suspensions, traps	2-6
Mode and protection	2-9
STRUCTURE OF A PROGRAM	3-1
Definition of modularity	3-1
Definition of a section	3-1
Section and segment bases	3-1
Consequences of modularity on MITRA 15 programs	3-2
Constituents of a program	3-5
Section calls	3-7
System's management concepts	3-10
ASSEMBLY LANGUAGE	4-1
Source line format	4-2
Basic character set	4-2
Symbols	4-2
Constants	4-3
Expressions	4-4
ADDRESSING MODES	5-1
Symbolic representation of the instruction	5-1
Addressing mode representation	5-2
PSEUDO-INSTRUCTIONS	6-1
Source text segmentation	6-1
Assembly pseudo-instructions	6-6
Page pseudo-instruction	6-14

Contents

INSTRUCTIONS 7-1
General 7-1
Symbolic notations 7-2
Load and store instructions 7-6
LBL 7-8
SBL 7-9
LBR 7-10
SBR 7-10
LBX 7-12
LDA 7-13
STA 7-14
LDE 7-15
STE 7-16
LDX 7-17
STX 7-18
LDR 7-19
STR 7-20
LEA 7-21
SPA 7-22
STS 7-23
DLD 7-24
DST 7-25

Fixed-point arithmetic 7-25
ADD 7-26
ADM 7-27
SUB 7-28
MUL 7-29
DIV 7-30

Logical operations 7-31
IOR 7-31
EOR 7-32
AND 7-33
CMP 7-34

Register incrementation and decrementation 7-35
ICX 7-35
DCX 7-36
ICL 7-37
DCL 7-37

Shift operations 7-38
SHR 7-40
SLLS 7-41
SRCS 7-42
SAD 7-43
SLCD 7-44
SLCS 7-45
SAS 7-46
SRLS 7-47
SRCD 7-48
SHC 7-49
SLLD 7-51
SRLD 7-52
PTY 7-53
NLZ 7-54

Contents

Inter-register operations 7-55
SRG 7-56
XAE 7-57
XAX 7-57
XEX 7-58
XAA 7-58
CCE 7-59
ACE 7-59
CCA 7-60
AEE 7-60
CNX 7-61
AIE 7-61
AAE 7-62
LNE 7-62
CNA 7-63
CMX 7-63

Floating-point arithmetic 7-64
FAD 7-66
FSU 7-67
FMU 7-68
FDV 7-69

Byte string processing 7-70
MVS 7-70
CPS 7-71
TRS 7-72

Branch instructions 7-73
BRU 7-74
BRX 7-75
BCT 7-76
BOT 7-77
BCF 7-78
BOF 7-79
BAZ 7-80
BAN 7-81
BE 7-82
BZ 7-83
BL 7-84
BLZ 7-85
BNE 7-86
BNZ 7-87
BGE 7-88
BPZ 7-89

System communication instructions 7-90
CLS 7-90
RTS 7-93
CSV 7-94
RSV 7-96
DIT 7-97
DITR 7-100

Control instructions 7-101
TES 7-102
STM 7-103
CLM 7-103

Contents

RD 7-104
WD 7-104
LDP 7-105

INPUT/OUTPUT CONTROL SYSTEM 8-1
Input/output system organization 8-1
Input/output interface 8-1
Transfers 8-6
Operational labels 8-11
Handler utilization 8-14

LIST OF PSEUDO-INSTRUCTIONS A-1

LIST OF INSTRUCTIONS B-1

ADDRESSING MODES C-1

ASSEMBLER OPERATION D-1

The informations contained in this manual can be modified without warning.

1. General

I-1. INTRODUCTION

MITRA 15 is a real time computer relying on a modular design and advanced micro-programmed structure for an efficient approach to numerous application fields, such as process control, scientific computation, remote-processing or transaction management.

The system includes up to four processing units specialized through read-only micro-programmed memories (ROM) and arranged around a planar structure core memory. According to their micro-programs, these units become CPU's, IOP's or special-purpose units. Each processing unit is provided with a "MINIBUS" for connecting a comprehensive range of peripheral units.

MITRA 15 is available in two fully compatible models which only differ by their processing capacity and the range of connectable peripheral units. Thus, each user can select the model and configuration best suited to its specific requirements.

I-2. MITRA 15 MAIN FEATURES

I-2.1. Core memory

MITRA 15's memory is a lithium-ferrite core random access memory organized in 16-bit words with 2 additional bits, 1 for parity and 1 for protection. The very short 800 nanoseconds word read/write cycle provides an outstanding transfer rate of 2.5 Mbytes per second (millions of bytes/second).

Memory contents are adressable on a byte basis and alterable on a byte, half-word or word basis.

The memory is made up of 4 096-word blocks (i.e. 8 192 bytes) up to a maximum of eight. Its capacity can thus be extended from 4 096 words to 32 768 words per 4 096-word increments.

■ Dynamic memory protection

This feature provides full protection to any memory area against unwanted attemps to alter its contents. The protection its assigned on a dynamic basis (LDP instruction).

A 1-bit protection "lock" is associated with every memory word. Besides, an indicator of the program status acts as a "key" : when this indicator is set to 1, the program is able to gain access to all memory locations; otherwise the program can only gain access to unprotected areas.

■ Parity

Full parity ckeck is provided both in the memory an during I/O operations.

I-2.2. Processing units

The functions available in a conventional computer are shared between a wired module, indentical for all processing units, the so-called "micro-processor", and the contents of a ready-only control memory which "specializes" the micro-processor to provide the functions of a CPU, IOP or special-purpose unit.

A processing unit comprises a fast register block, five program indicators, a micro-programmed read-only memory, an operator and an interrupt/suspension system.

■ Fast register memory

This memory is implemented in MSI bipolar integrated circuit registers which are organized in eight 16-bit program-adressable register blocks. The capacity of each processing unit can be extended from two to eight 8-register blocks per 2-block modules. Access time : 60 nanoseconds per word.

In the CPU, the first block (block 0) is assigned to the program context, the remaining blocks being available for peripheral transfers.

In an IOP, all block are available for peripheral transfers.

Block 0 of CPU

P	Program counter
L	Local base
G	General base
A	Accumulator
E	Extended A reg.
X	Index

} Program context

■ Program indicators

C	Carry or operation test
O	Overflow or operation test
MS	Mode : master/slave
MA	Interrupt Mask
PR	Memory protection

For detailed description of indicators see chapter II.

■ Instructions

MITRA 15 has a set of 86 instructions including :

- 40 memory reference instructions,
- 29 register instructions,
- 12 shift instructions,
- 5 special instructions.

All instructions have a fixed format :

3	5	8
Mode	Function	Displacement

or

4	4	8
Mode	Function	Displacement

They operate on bytes (half-words), words, double-words or unlimited length byte strings.

Addressing modes :

- immediate addressing for operands which can be coded in one byte;
- direct, indirect and indexed addressing with respect to the local base;
- direct, indirect and indexed addressing with respect to the general base.

■ Micro-programmed read-only memory

This non-destructive read out permanent memory is pre-recorded. Each 16-bit word contains one micro-instruction. The memory is implemented in MSI bipolar IC's and its access time is 60 nanoseconds. Its capacity is 512 or 1024 words per processing unit.

Three different versions are available for the following functions :

MC1 : executes the basic instruction code and the coupling functions for peripherals which are connectable to CPU's Minibus only.

MC2 : executes the complementary code (optional instructions) and the coupling functions for peripherals which are connectable to the Minibus of either the CPU or an IOP.

MC3 : executes the coupling for peripherals which are connectable to an IOP's Minibus.

■ Interrupts (IT)

32 priority interrupt levels are available which can be armed, masked or triggered by program. They provide up to 112 discrete external interrupts by grouping.

When an interrupt is triggered, the program context is automatically swapped in 30 µs.

For a special high speed interrupt level, this swapping is effected in 5 µs only, by register block switching.

■ Suspensions

MITRA 15 also offers 32 suspension levels organized on a priority basis for micro-program coupling of peripherals requiring urgent or frequent transfers.

The maximum response time is 300 µs.

■ Minibus

Each processing unit is provided with a Minibus for device controllers connection. This Minibus is implemented as a printed circuit located in the chassis wiring and provides non-specialized plug-in locations for all controller cards.

I-3. MODELS MITRA 15/20, MITRA 15/30

MITRA 15/20

■ MITRA 15/20 CPU comprising, as standard :

- 512-word micro-programmed ROM (MC1),
- 64 fast access registers,
- Basic code providing for 77 instructions,
- Priority interrupt system.

■ Core memory

4K to 32K 16-bit words per 4K increments.

■ Performance

- Addressing mode : direct, indirect, indexed, relative, immediate, local and general.
- 1 index, 2 bases.
- 77 instructions, including 33 memory reference instructions.
- Load, store or add word in 2.1 µs.

■ Main options

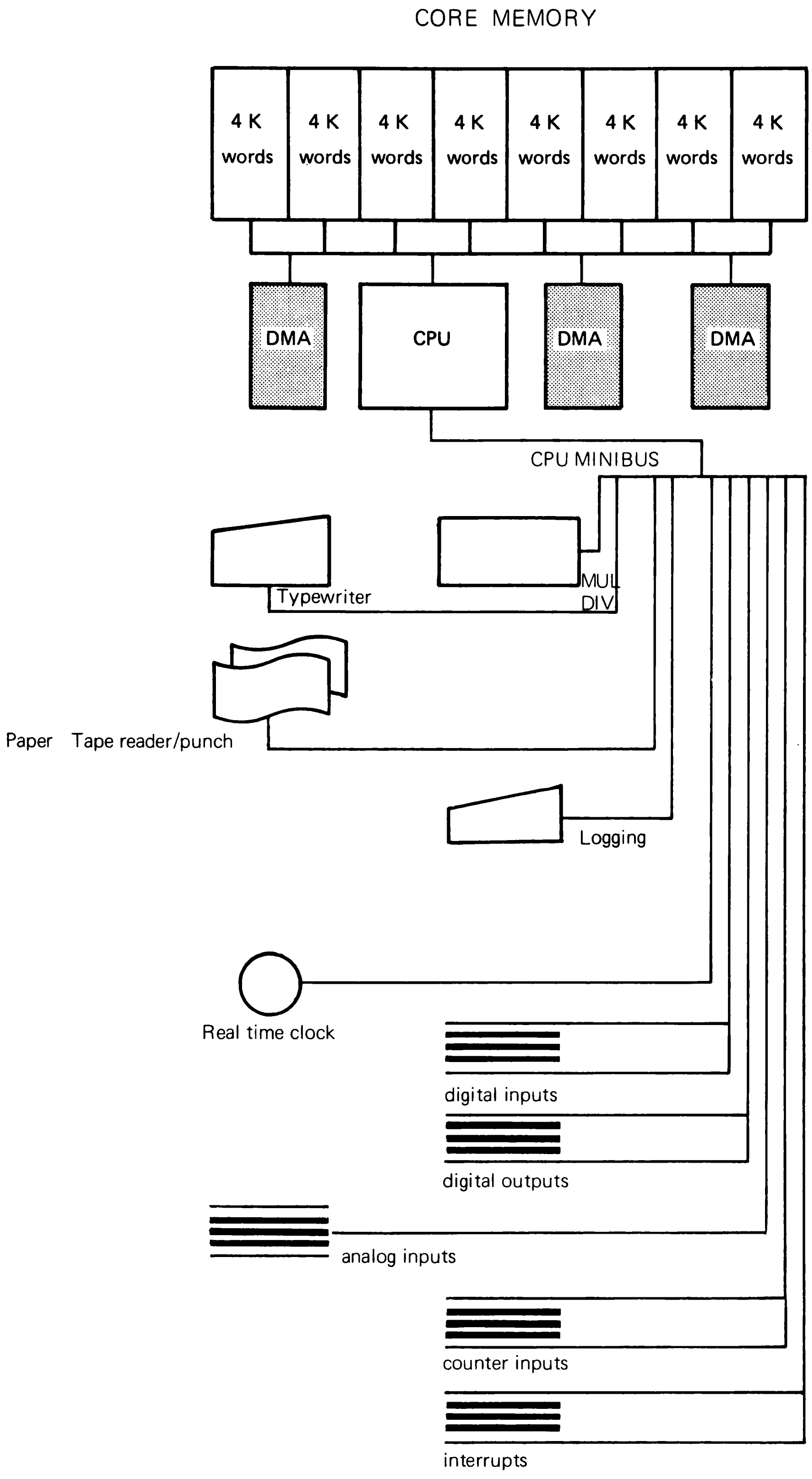
- Up to 3 direct memory accesses (DMA)
- Additional micro-programmed ROM (MC2)
- Up to 32 priority interrupt levels per 1 or 4 levels modules.
- Wired MUL/DIV (7 and 8 µs).
- Floating point operator (OVF).
- Power failure protection.

■ Software

- MITRAS 1 Assembler;MITRAS 2 Extended Assembler; LP 15; BASIC; FORTRAN IV.
- Linkage editor.
- 2 Monitors : Basic Monitor MOB and Real-Time Monitor MTR.
- Libraries.

■ Basic peripherals (Range I)

- Console typewriter (with paper tape reader/punch),
- 300 char./sec. paper tape reader,
- 60 char./sec. paper tape punch,
- Logging slow printer (15 char./sec.),
- 2 to 128 16-bit digital input lines for logical levels, or filtered, or relayed,
- 2 to 64 16-bit digital output lines for logical levels or relayed,
- Counter inputs; real-time clock,
- Analog inputs.



MITRA 15/20 general layout

MITRA 15/30

■ MITRA 15/30 CPU comprising, as standard :

- 1024-word micro-programmed ROM,
- 32 fast access registers,
- Extended operation code for 86 instructions,
- Priority interrupt system,
- Wired MUL/DIV,
- Power failure protection.

■ Core memory

4K to 32K 16-bit words per 4K increment

■ Performance

- Addressing modes : direct, indirect, indexed, relative, immediate, local and general.
- 1 index, 2 bases
- 86 instructions, including 40 memory reference instructions
- Load, read, write or add word in 2.1 µs
- MUL/DIV in 7 and 8 µs, respectively.

■ Main options

- Up to 3 direct memory accesses (DMA)
- Up to 3 input/output processors (IOP)
- Extension to 128 fast access registers per processing unit (64-level modules)
- 32 priority interrupt levels per 1 or 4 level modules
- Floating-point operator (OVF)

■ Software

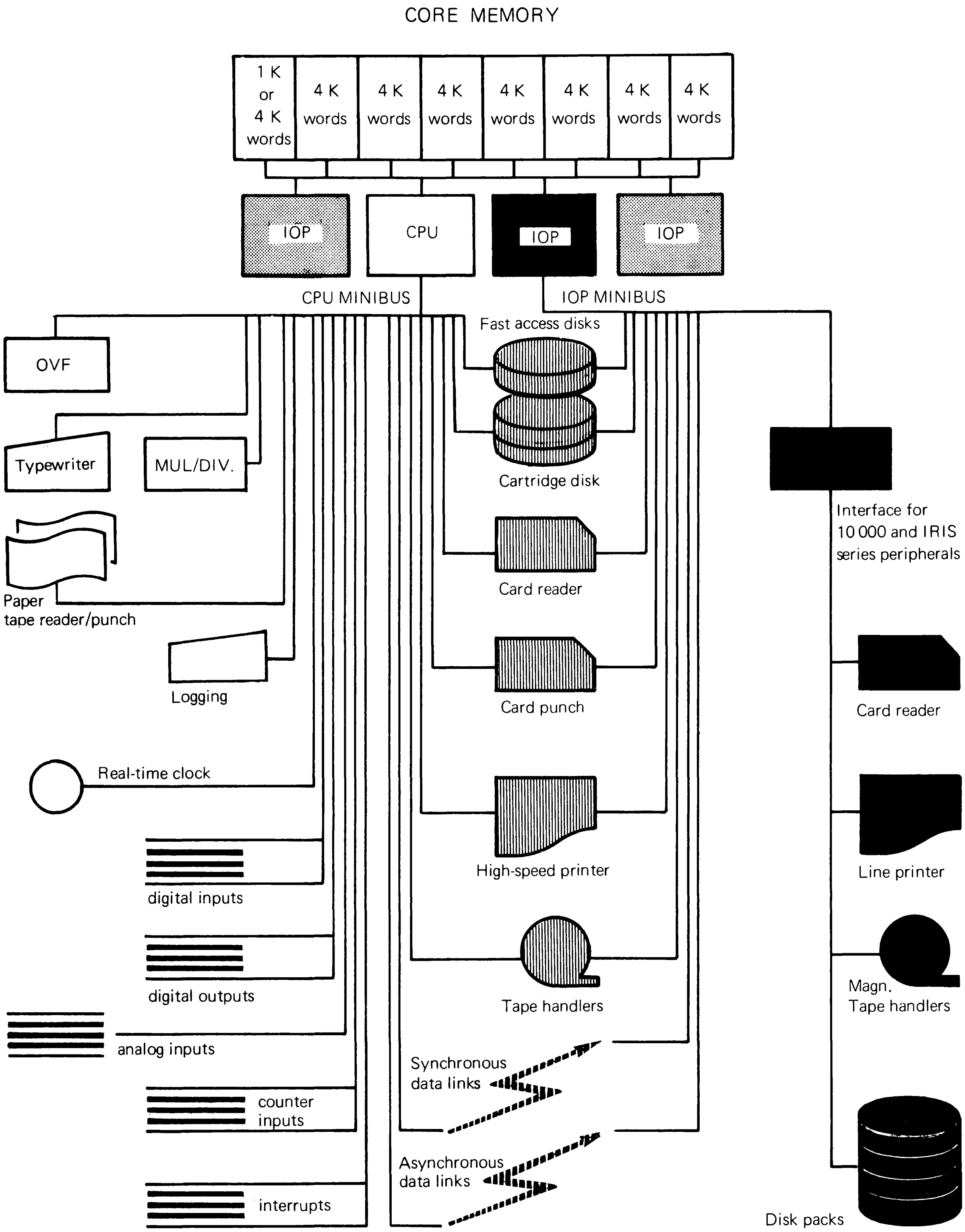
- MITRAS 1 Assembler; MITRAS 2 Extended Assembler; Macro-generator; LP 15; BASIC; FORTRAN IV; Librarian; File management system.
- 3 Monitors : Basic Monitor MOB; Real-Time Monitor MTR; Disk Real-Time Monitor MTRD.

■ Peripherals

• Range I (Model 20)

• Range II :

- One head per track fixed-disks; average access time 10 ms, transfer rate 150 kbyte/sec, capacity 100 to 1600 kbytes.
- Movable head disk-pack units; average access time 60 or 90 ms; transfer rate 100 or 150 kbyte/sec; capacity 2.5 to 5 Mbytes or 6.2 to 24.8 Mbytes.
- Card reader : 200 or 600 cpm.
- Card punch : 20 or 40 cpm.



MITRA 15/30 general layout

- Line printers : 132 columns; 200, 400 or 600 lpm.
- Communication controller for 1 synchronous data link, full duplex, 1200/4800 bauds
- Communication controller for 2 asynchronous data links, full duplex, 50 to 1200 bauds.

• Range III :

- OCTET interface for connecting all CII 10 000 or IRIS Series peripherals : (card readers, printers, tape handlers, etc...).

I-4. MITRA 15 OPERATING SYSTEM

Depending on the availability of a fast access disk unit, the software is offered in two different versions : the resident system and the disk system.

■ The resident system provides :

- 2 Monitors : the Basic Monitor MOB and the Real-time Monitor MTR
- Assemblers : MITRAS and LP 15; Compilers : BASIC and FORTRAN and a Macro-generator.

■ The disk system provides :

- The real-time Monitor MTRD
- The resident system processors, a librarian and a linkage module.

In addition, the software includes :

- Debugging commands available as extension of each monitor.
- A comprehensive library of "real-time" mathematical programs, and a file management system.
- MITRA 15 Simulators available for use on CII 10 070, IRIS 50, IRIS 80, IBM 360, etc.

MITRAS I Assembler

Translates the symbolic MITRAS language generates in one single pass a relocatable binary object-listing and a list of error diagnostics.

Both source and relocatable binary programs are normally on paper tape; memory requirement : 4K words.

MITRAS 2 Extended Assembler

Translates the symbolic MITRAS language; has a larger set of pseudo-instructions than MITRAS I. Memory requirement : 8K words.

LINKAGE EDITOR

Operates in two passes for converting binary relocatable programs generated during various assembly or compilation runs, into a relocatable memory image format which can be loaded for execution by the Basic Monitor.

The linkage editor also provides a memory map of the relative location of the various modules and a listing of the common sub-routines which are called. Memory requirement : 4K words.

BASIC MONITOR MOB

Perform computer control and handles user's communications with the system and the basic processors. Its main functions are :

- trap processing,
- internal interrupt control,
- program loading,
- input/output control,
- program execution control.

Memory requirement : 4K words.

REAL-TIME MONITOR MTR

Handles simultaneously interrupt-dependent batched jobs in core. Controls and supervises all privileged operations, such as I/O handling or memory protection, and provides operator communication. Memory requirement : 8K words.

	4 Kwords	8 Kwords	12 Kwords	16 Kwords	Simulation
OPERATING SYSTEM resident system	Basic Monitor MOB	Real-time Monitor MTR			control command analyzer interpreter
disk system		disk real-time Monitor MTRD	linkage module		
PROGRAM GENERATION resident system	MITRAS I linkage editor loader-editor	MITRAS 2 BASIC	LP 15	FORTRAN IV	Assembler linkage editor
disk system		MITRAS 2 linkage editor BASIC LP 15 Librarian	FORTRAN IV	Macro-generator	LP 15
LIBRARY	Mathematical programs real-time programs communication programs file management system packages				

Structure of MITRA 15 standard software

DISK REAL-TIME MONITOR MTRD

This disk-oriented version of the MTR Monitor has additional capabilities for overlay control and user's libraries management, as well as for automatic linking of batched programs (compile-link, load-and-go); requires 8K words memory and a fast access disk unit.

LOADER - LINKAGE EDITOR

Operates in one pass for loading binary relocatable programs for immediate execution.

This processor can only accept binary programs generated by MITRAS I. Memory requirement : 4K words.

LP 15

This assembler type language has a syntax which is closely related to that of sophisticated languages such as ALGOL, but with the feature of direct access to MITRA 15's registers.

The binary object programs thus generated have an efficiency which is practically equivalent to that of assembled programs. Memory requirement : 12K words without a disk unit.

BASIC

This conversational compiler provides for time-shared operation and alphanumerical data processing. Memory requirement : 8K words.

FORTRAN IV

This compiler generates in one single pass a relocatable binary object-program in the format required by the linkage editor. May call sub-routines written in another language and translated in relocatable binary format; compatible with CII 10 020, IRIS 45 and IRIS 50. Memory requirement : 16K words or 12K words with a disk unit.

AMAP EXTENSION (DEBUGGING AIDS)

An AMAP extension available with every monitor as a debugging aid and provides instruction execution records, halt on address, memory dumps and contents alteration, through special monitor commands.

LINKAGE MODULE

Provides for automatic linking of batched programs in the deffered processing area with concurrent real-time programs.

This processor is controlled by the disk real-time monitor MTRD; requires 12K memory words and a fast access disk unit.

MACRO-GENERATOR

Translation program using user-defined procedures. It provides in one pass a program in assembly or compilation language. Memory requirement : 16K words.

LIBRARIAN

Provides for handling the system library constitutive files through commands such as : insert, replace, copy, load, dump on external medium... Memory requirement : 8K words and a fast access disk unit.

UTILITY PROGRAMS

These programs are available for :

- Updating and correcting source programs on sequential access media (paper tape, magnetic tape, etc.).
- Handling and updating library programs on sequential access media. Memory requirement : 4K words.

MITRA 15 SIMULATORS

These simulation programs are available for CII 10 070, IRIS 50, IRIS 80, IBM 360, etc. computers and include :

- a MITRA 15 interpreter,
- MITRAS Assembler and Linkage editor,
- a system generator,
- LP 15 Compiler.

They perform assembly, linkage edition and debugging functions on programs intended for later exploitation on any MITRA 15 configuration.

I-5. APPLICATIONS

LABORATORIES	Spectrometry
	Gazeous chromatography
	Cristallography
MEDECINE	Chemical analysis
	Electrocardiography
ENGINEERING	Components testing
	Seismography
	Ranging
INDUSTRY	Chemicals
Monitoring	Oil and derivates
Automation	Steel industry
Process control	Mechanical engineering, aerospace, etc.
REMOTE PROCESSING	Deconcentrated companies
Front-end computers	Public Administrations
Satellite stations	Universities
Front ends	
SCIENTIFIC COMPUTATION	Education
Time-sharing	Design office
Data centers	Private companies
TRANSACTION PROCESSING	Insurance companies
Data collection	Banks
File management	Public services...

2. General layout

MITRA 15 is built around a planar structure core memory the capacity of which can be extended modularly by 4K 16-bit words blocks. This core memory has four access ports for connecting up to four processing units or direct memory access controllers.

Each processing unit includes a micro-programmed read-only memory (ROM); a specific micro-program pre-recorded in this memory specializes the associated processing unit for performing the functions of :

- a central processing unit (CPU),
- an input/output processor (IOP), or
- a special-purpose unit for a particular process.

Each processing unit controls a so-called MINIBUS which is a peripheral bus designed for direct connection of peripheral controllers.

II-1. CORE MEMORY

The core memory is basically organized in 18-bit words each comprising 16 data bits, 1 parity bit and 1 memory protection bit.

Read/write operations are executed in two separate half-cycles. A read cycle includes a destructive read-out half-cycle followed by a rewrite half-cycle. A write cycle includes a clear half-cycle followed by a write half-cycle.

Memory access time is 400 ns (1/2 cycle) and a full read/write cycle lasts 800 ns.

Though memory transfers are performed on a word basis, micro-commands allow the programmer to operate on bytes, i.e. on half-words. Thus, all MITRA 15 addresses point to byte locations, even-numbered addresses corresponding to word locations.

The memory is built up with 4096-word modules, i.e. 8192 bytes. MITRA 15 is designed for a maximum of eigh modules corresponding to a maximum capacity of 32 768 words (or 65 536 bytes).

The control logic supplies the timing signals required for operating the memory proper (half-cycles timing control), and the transfer signals for data exchanges with the processing units; in addition it deals with the four accesses relative priorities.

II-2. PROCESSING UNITS

The operation of a MITRA 15 processing unit, and more specifically of the CPU, may be described at two fully distinct levels :

■ A first level corresponding to what may be termed "user-level" and the knowledge of which is sufficient for programming an application on MITRA 15.

It includes the following features :

- the standard instruction set detailed in chapter VII;
- six general registers of block 0;
- the five program indicators;
- the interrupt system.

■ A second level corresponding to what may be termed "micro-processor" level

This micro-processor includes the following features :

- a set of about forty hardware-implemented basic micro-instructions;
- a read-only memory implemented on module boards and which contains the sub-routine set, (also called "micro-programs") defining MITRA 15's standard instructions set and peripheral coupling functions;
- operational registers;
- micro-processor status indicators;
- a so-called "suspension system" corresponding, for the second level, to the interrupt system of the first level.

The following sections describe the various components of a processing unit, viz :

- micro-programmed ROM
- S and M memory transfer registers
- fast-access register blocks
- status indicators
- interrupt and suspension systems.

II-3. MICRO-PROGRAMMED ROM (OR MICRO-PROCESSOR)

This non-destructive ROM is pre-recorded in factory and implemented in integrated circuits (access time : 60 ns per word).

Each memory word is 16 bits long and contains one micro-instruction.

The control ROM of a processing unit contains either 512 words (MC1), or 1024 words (MC1 + MC2).

Any micro-instruction is executed in 300 ns.

The address of the currently executed micro-instruction is contained in a 10-bit register called T-register.

A micro-instruction has the following format :

0 1	2 6	7 9	10 15
M	OP	CC	AD

Each micro-instruction has a dual purpose :

1) It controls a number of functions, viz. :

- memory control (2 bits : M-field)
- basic operation code (generally 5 bits : OP-field)
- complementary operation code (3 bits : CC-field) defining for instance a general register address

2) It defines the address of the next micro-instruction (through a 6-bit modifier : AD-field), by updating T-register contents.

In fact, micro-instructions are not stored sequentially.

No indexing adder is associated with T-register, since its contents is not incremented by one unit from a micro-instruction to the next, as in a sequential addressing scheme.

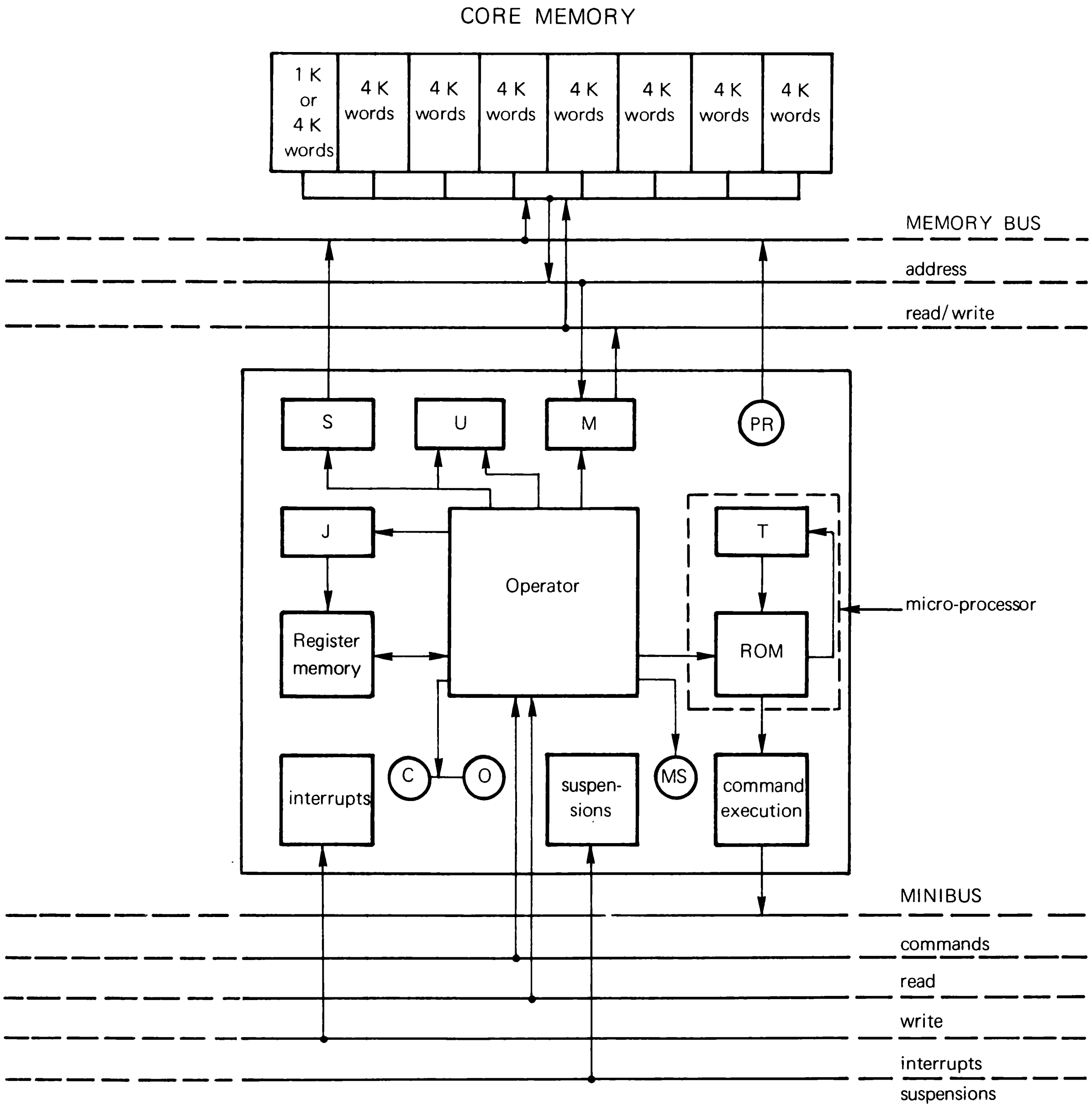
CPU's and IOP's are differenciated by the kind and contents of their respective control memories.

In the CPU, MC1 control memory (512 words) executes the basic operation code and the coupling functions for peripherals which are connectable to its Minibus only (Range I).

MC2 control memory provides for executing the complementary operation code (optional instructions) and the coupling functions for peripherals which are connectable either to the Minibus of the CPU, or to the Minibus of an IOP (Range II).

The CPU's control memory then includes 1024 words (MC1 + MC2).

MC3 control memory executes the coupling functions for peripherals which are connectable to the Minibus of an IOP only (Range III).



Processing unit layout

II-4. REGISTERS

II-4.1. Memory transfer registers

S-register is a 15-bit address register, though actual addresses are 16 bits long. The rightmost bit of an address, which specifies the desired byte within the addressed word, is in fact ignored by the memory logic.

M-register is an 18-bit data register receiving the transferred memory words. Two of these bits are reserved for parity and protection tests; the 16 other bits are used for date exchanges with U-register.

II-4.2. Fast-access registers

A standard MITRA 15 processing unit includes eight register blocks each comprising eight 16-bit integrated circuit registers numbered 0 through 7. Eight optional blocks are available on 15/30 model.

These registers have different assignments in the CPU and in an IOP.

In the CPU, the first block (block 0) is reserved for program execution; its first six registers have the following functions :

A	-	Accumulator,
E	-	Accumulator extension,
P	-	Program counter,
X	-	Index register,
L	-	Local base register,
G	-	General base register,

the last two registers, V and W ared used by micro-programs.

The other blocks are normally assigned to peripheral transfers through the suspension system (channel memories).

In an IOP, all register blocks are available for peripheral transfers.

Each register has a unique address form 0 to 63 (or 127). In the micro-programs, a general register address is generated from :

- the contents of the corresponding field of the micro-instruction format (3 bits).

- the contents of J-register.

It will be seen in section II-8. that a high-speed interrupt causes an automatic switching of the register block. In the new block, the registers have then the same assignment as in block 0, but for other programs.

II-5. LOGICAL AND ARITHMETICAL OPERATOR

The logical and arithmetical operator includes a universal register, or U-register, and a dual-input operator. The 16-bit U-register cannot be directly accessed by the instructions, but constitutes an accumulation register for the micro-processor. In this respect it can contain one operand of a micro-instruction and/or store the result. Both operands of a micro-instruction may also be provided by :

- a general register (operand 2)

- M-register in connection with the core memory (operand 1)

- the I/O interface (operand 2)

- the control memory (operand 2)

- the stack (operand 2)
- the indicators (operand 2)

The results of the operation are stored in the following devices :

- U-register (universal)
- M-register (data) } for core memory transfers
- S-register (address) }
- a general register
- the indicators

II-6. INDICATORS

MITRA 15's central processor includes nine indicators :

■ Four indicators reserved for micro-processor use :

B-indicator	: assigned to U-register overflows
Tz-indicator	: for a zero micro-instruction result
To-indicator	: for the sign of a micro-instruction result
Ao-indicator	: address of the processed byte.

■ Five program-accessible indicators

C = Carry

This indicator has two different meanings according to the last instruction by which it is set.

• Carry/borrow (arithmetic type instruction)

- When a positive number is added (negative number subtracted), if the result is obtained without becoming zero, C is reset (C = 0).

- When a positive number is added (negative number subtracted), if the result is obtained after becoming zero, C is set (C = 1).

- When a negative number is added (positive number subtracted), if the result is obtained without becoming zero, C is set (C = 1).

- When a negative number is added (positive number subtracted), if the result is obtained after becoming zero, C is reset (C = 0).

• For other instructions using C-indicator, the status C-indicator, the status C = 1 after execution denotes a zero value in a register or, in the case of a comparison, equality of two values.

O = Overflow

This indicator also has two different meanings according to the last instruction executed.

• For arithmetic type instructions, O-indicator is used for overflow. More precisely, when both operands have the same sign, if the result is of the opposite sign, O is set (O = 1). Otherwise, O is reset (O = 0).

• For other instructions using O-indicator, the status O = 1 after execution denotes a negative value in a register (leftmost bit set) or, in the case of a comparison, that A-register value is less than the addressed word value.

MS = Master/Slave mode

For programs executed in Master mode, this indicator is set to 1, otherwise the program is executed in normal or "slave" mode.

See chapter VII "Instructions" for detailed description of the above three indicators for every instruction.

MA = Interrupt mask indicator

This indicator is set to 1 for masked interrupts, otherwise MA = 0.

PR = Memory protection "key"

When PR = 1, the program is able to gain access to any memory location.

When PR = 0, the program is only allowed to gain access to unprotected memory areas ("protection lock" cleared).

These five indicators are included in the context of a specific program.

II-7. COMMUNICATION WITH THE ENVIRONMENT

The processing unit is coupled to the peripheral controllers via a so-called MINIBUS which is accessible through micro-instructions. The interface includes :

- for data : 16 output bit lines and 16 input bit lines;
- three function bit lines;
- for addresses : 6 bit lines or 10 bit lines in particular cases;
- a sync line;
- a reset line.

The peripheral Minibus, on which the peripheral controllers are connected, includes 16 unidirectional data lines, both for input and output, an address and peripheral control bus, as well as interrupt and suspension lines.

II-8. INTERRUPTS - SUSPENSIONS - TRAPS

II-8.1. Interrupts

The interrupt system operates when :

- an interrupt signal occurs;
- a special micro-instruction, located by definition at specific "interrupt point", occurs;
- interrupts are unmasked;
- the priority level of the current program is lower than that of the incoming interrupt.

There are 32 interrupt levels (IT levels). Each of these levels has an associated memory address containing the context pointer of a program specifically assigned to this level. These 32 context pointers are stored in a table pointed to by the contents of memory address 10.

When an interrupt condition occurs :

- the condition is stored in a flip-flop (one per signal),
- its IT-level is hardware-coded and compared with that of the task currently processed (register 8),

- if the interrupt is accepted, its specific IT-level (0 through 32) is stored in the hardware of the micro-processor when the interrupt test micro-instruction is executed,

- then, the micro-program performs the following operations :

. Storage of the interrupted task context at an address depending on its rank (the latter being stored in register 8).

. Loading of the interrupting task context from an address depending on its rank.

. Call of the first instruction of the interrupting task. (See "Communication with the micro-processor" page II-8).

When the task is over or must wait for the occurence of a specific event, it releases the processing unit through an interrupt de-activation and context swapping instruction DIT which :

- acknowledges the interrupt calling for the task,

- stores the task's context, and

- calls for the next task waiting at the same IT-level, or, if there is no such task, for a task waiting at the next lower IT-level.

If no task is waiting, the computer executes a wait loop until an external event occurs at the lowest level.

The total number of interrupt levels is 32, 4 internal and 28 external. Besides, up to 4 interrupts may be on a same level, providing a total number of multiplexed external interrupts equel to 112 (28 x 4).

As a rule, standard peripheral controllers use one interrupt level each.

Internal interrupt levels are assigned to the following tasks :

- operator's console interrupt request,

- power turn on,

- power shut down,

- program (level 0).

High-speed interrupt

Optionally, one external IT-level may be of the "high-speed" type, i.e. may call for a task the context of which is stored in a register block other than block 0, which contains the interrupted task context. Consequently, the task switching only requires that the indicators be transferred in block 0; it lasts about 2 µs.

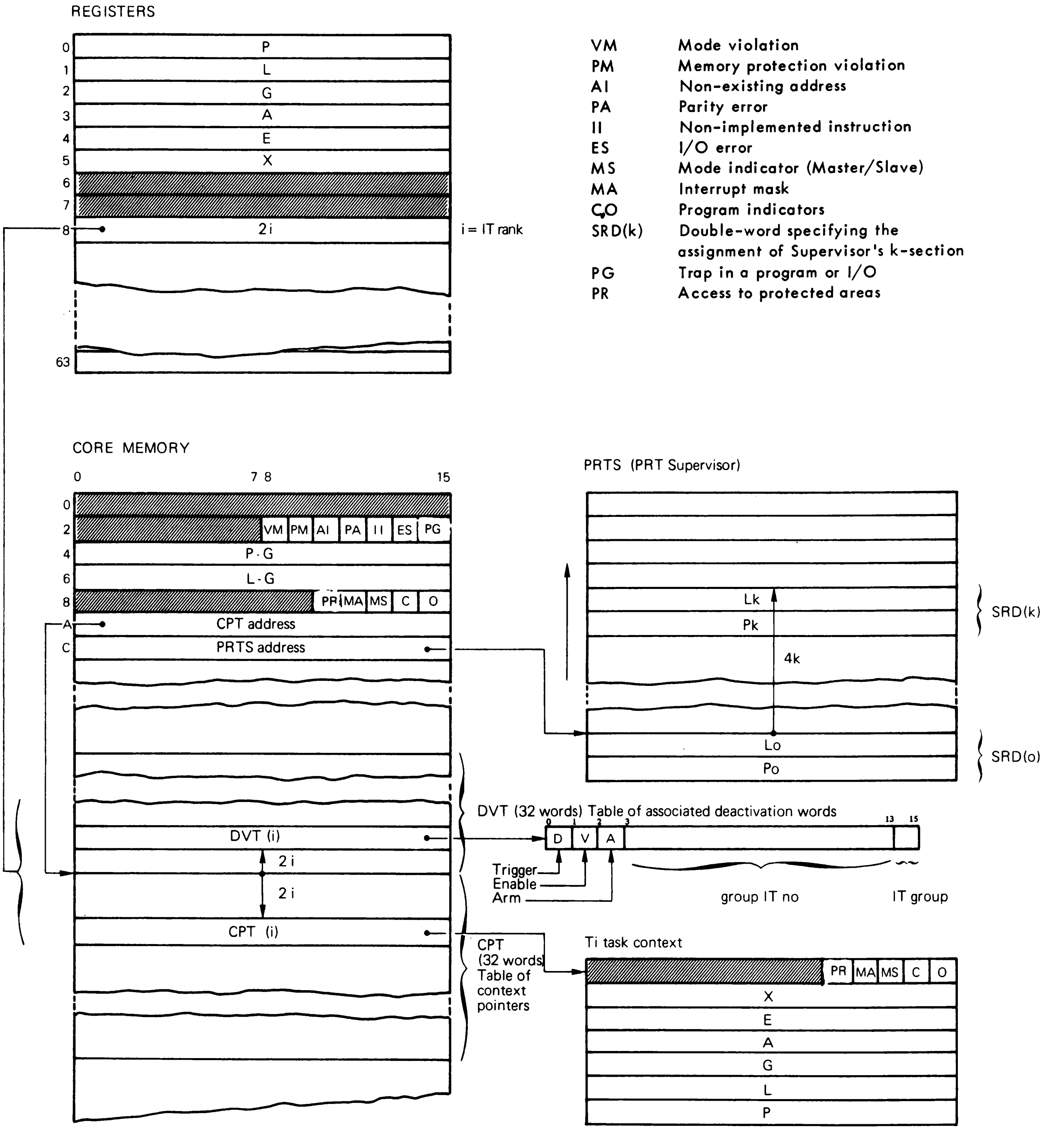
When this "high-speed task" is acknowledged, the control is returned to the interrupted task (the context of which is still in block 0) through a special DITR instruction by-passing the usual context swapping in block 0.

II-8.2. Suspensions

The suspension system is able to interrupt the current micro-program at the end of every micro-instruction, and to launch a special micro-program. The suspension request is either issued by a peripheral or internal to the micro-processor (processing unit).

On occurence of a suspension, the micro-processor's status, i.e. the contents of U-, J-, T-registers and of B, Tz, To, Ao indicators are transferred in a stack. The suspension micro-program is then executed.

At the end of the suspension program, the initial contents of U-, J-, T-... registers are restored from the values previously saved in the stack.



VM	Mode violation
PM	Memory protection violation
AI	Non-existing address
PA	Parity error
II	Non-implemented instruction
ES	I/O error
MS	Mode indicator (Master/Slave)
MA	Interrupt mask
C,O	Program indicators
SRD(k)	Double-word specifying the assignment of Supervisor's k-section
PG	Trap in a program or I/O
PR	Access to protected areas

Communication with the micro-processor

The stack has a capacity of four suspensions, i.e. the number of suspension levels is four. The number of suspension signals is 32, or 8 per level, assigned as follows :

- 5 internal suspensions :

. traps (1)
. interrupts (2)
. control panel (1)
. power failure (1)

- 27 external suspensions associated with peripherals.

II-8.3. Traps

The origin of a trap is an abnormal condition detected at the end of a micro-instruction.

The trap processing micro-program :

- protects bytes 4 to 9 of the memory which contain L- and P-register values and the indicators status of the context of the instruction which initiated the trap;
- signals the cause of the trap by setting a bit in memory word 2;
- performs a call to supervisor section 0.

The following abnormal conditions initiate a standard trap :

- non-existing memory address : the user has specified an address exceeding the available memory.
- memory protection violation : the user attempts to write in a protected memory area with a zero PR-key.
- parity fault in core memory read-out signals.

Other traps may be initiated by the following causes :

- operating mode violation : attempt to use priviledged instructions in a slave mode program.
- invalid instruction : incorrect OP-code specified.
- "watch-dog" timer runout.

In all these situations :

- the current instruction is aborted,
- the micro-processor's stack is not triggered,
- a special micro-program generates a supervisor call.

The operations performed by the standard monitors in response to a trap condition are described in the corresponding utilization manuals. The trap status word is described in "Communication with the micro-processor" diagram page II-8.

II-9. MODE AND PROTECTION

II-9.1. Operating modes

- Normal or "slave" mode.

In this mode, priviledged instructions cannot be executed and any attempt to execute such an instruction causes a "mode violation" trap. MS indicator is reset (MS = 0).

- Priviledged or "master" mode.

In this mode all instructions, whether priviledged or not, are executable. MS indicator is set (MS = 1).

The various supervisor modules are examples of programs which must be executed in master mode. (See CSV and RSV instructions).

It should be noted that addressing modes are different in master and slave modes (see Chapter V "Addressing modes") to provide absolute addressing capability in master mode.

II-9.2. Memory protection system

The protection system becomes operative whenever PM key-switch is turned on the control panel.

The operation is as follows :

- a 1-bit protection "lock" is associated with each memory word and may be set by a LDP instruction (LoaD Protection).
- the program status includes a PR-indicator which acts as a "key".

If key value is 1 (override key), the program may gain access to all memory locations.

If key value is 0, the program may only gain access to memory locations whose lock value is 0.

■ PR-key loading

The PR indicator is loaded with the program context.

It is preserved before being forced to 1 during any supervisor call SVC and restored to its previous value when the supervisor returns the control to the calling program.

■ Protection violation

If a "zero key" program attemps an access to any location having a 1 lock value, the protection system operates and initiates a "protection violation" trap.

Memory protection and operating mode are independent.

3. Structure of a program

III-1. DEFINITION OF MODULARITY

In programming art, as in other techniques, the modularity consists in breaking down a system in to smaller elements with standard interfaces.

Since the introduction of the "sub-program" concept, modularity is an acomplished fact in programmation. As a main program may also be considered as a module, we rather call them "sections". The following advantages are due to modularity :

- easier system specification,
- easier software writing, by sharing the work between a number of programmers,
- identical sections may be used in different system without rewriting,
- easier debugging and assistance on software products.

III-2. DEFINITION OF A SECTION

A section mainly comprises an instruction sequence called a program segment. The purpose of these instructions is to process data which are either assigned to the section, or shared between a number of sections.

Data which pertain to a section in proper make up a "local data segment" (LDS).

Data which are common to several sections make up a "common data section" (CDS).

Accordingly a section is either the common data section CDS, or a local data segment (LDS) plus an executable program segment (Local Program Segment = LPS).

The CDS is accessible from any point in the program.

More particularly, the CDS may be accessed from a LDS in general addressing mode (direct, indirect or indexed indirect).

Symbols and labels defined in the CDS are applicable to the whole program.

A LDS is accessed from the associated LPS in local addressing mode (direct, indirect or indexed indirect).

Symbols and labels defined in a LDS are applicable to the section only. Nevertheless, they may be referenced in the CDS. A program segment is exclusively made up of unalterable items (instructions), and this improves relocatability and simplifies writing of re-entrant sub-routines.

III-3. SECTION AND SEGMENT BASES

• General base G

General base G is uniquely assigned to the program; it constitutes an implicit base to which every address referenced by this program is related. Accordingly, the micro-processor automatically adds this base value to all addresses specified in the instructions.

• Local base L

Local base L is the implicit base value for all local data contained in a local data segment (LDS).

- Program base P

Program base P is assigned to a local program segment (LPS).

Initially, base P is the starting address of the section and from there on acts as a program counter for the currently executed section (see Chapter III-2.).

The actual values of L- and P-bases may be unknown at the time a program is written. At linkage edition time, they are automatically generated in relative value with respect to the general base of the program and stored in the associated PRT.

III-4. CONSEQUENCES OF MODULARITY ON MITRA 15 PROGRAMS

From the hardware viewpoint, modularity implies the existence of special instructions for section calls and returns.

From the software viewpoint, program modularity is a fundamental concept of the assembly language which includes so-called "segmentation" pseudo-instructions :

CDS : Common Data Section

LDS : Local Data Segment

LPS : Local Program Segment

FIN : End of segment or section (LDS, LPS or CDS)

IDS : Indirect Data Segment.

We shall call "program module" the result of an assembly or compilation processing. When a module is written in assembly language it is rather called "assembly module". Every assembly module must conclude with an END pseudo-instruction. A program may be built up from modules of various origins (differing by their source language, author, creation date, etc.).

The linkage editor interconnects the various modules into a complete executable program.

Remark :

To facilatate the programming, particularly in the case of re-entrant sub-routines, the assembler recognizes so-called "dummy data segments" which are images of later-defined data or of data belonging to another LDS (or CDS) than the LDS in which the dummy area is defined.

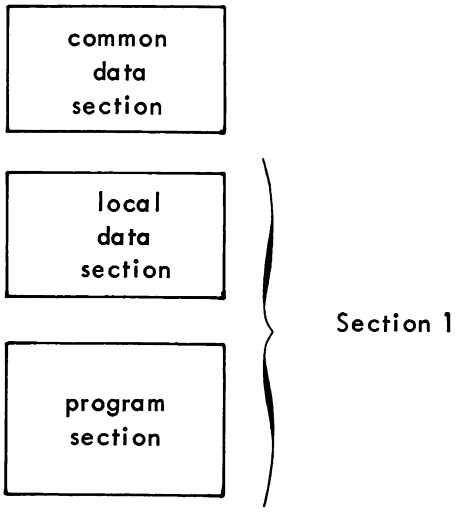
These dummy segments are treated as formal parameters, in particular for defining relative displacements with respect to the beginning of the segment (description of dynamic data blocks, index values, etc.) but generate no object code.

Example 1 : Typical organization of a program

```
COMMON      CDS
TWB         RES         16
C1          DATA        1
            FIN

LOCAL       LDS
            RES         2
C2          DATA        &F0
            FIN

SPROG       LPS         LOCAL
DEB         LDA         = 3
            AND         C2
            RTS
            FIN         DEB
```



Example 1 : Typical organization of a program (continued)

```
LOCP        LDS
            RES         2            local
U           DATA,1      28           data
V           DATA,1      31           section
TAB         DATA        ATAB
ATAB        RES         1024
            FIN
                                                    Section 2
PRINC       LPS         LOCP
INIT        LDA         U
            ADD         = C1
            STA         TAB          program
            CLS         SPROG        section
            CSV         M:EXIT
            FIN         INIT
            END         PRINC
```

End of file code (%EOD on card and paper tape).

Example 2 : Other possible special organizations

```
PROG        CDS
              :
            FIN

LPS1        LPS         PROG
              :
            FIN
            END         LPS1
```

This LPS having no associated LDS cannot use the local addressing mode; it must use the general addressing mode

---

```
            CDS
              :
            FIN

LDS1        LDS
              :
            FIN

LPS1        LPS         LDS1
              :
            FIN
LPS2        LPS         LDS1
              :
            FIN
            END         LPS1
```

These two LPS are both associated with the same LDS. The local symbols are deleted at the beginning of the next LDS. Nevertheless, there are two distinct sections (two items in the PRT).
The local base L being initially the same for both sections, no mutual calls are allowed (through CLS pseudo-instructions).

Example 2 : Other possible special organizations (continued)

```
                    CDS
                     .
                     .
                    FIN
LDS1                LDS
                     .            This LPS having no associated LPS, it is only
                     .            accessible through indirect addressing via an
                    FIN           item of the CDS.
LDS2                LDS
                     .
                     .
                    FIN
LPS2                LPS
                     .            This LPS cannot refer to the LDS called LDS2,
                     .            since local symbols are deleted after every
                    FIN           occurence of an LDS pseudo-instruction.
                    END        LPS2
```

Example 3 :

First module	Second module	Third module	Remarks
PROG CDS	PROG CDS DUM	PROG CDS DUM	These CDS reflet each others. The dummy CDS DUM do not generate any object code. They use to satisfy the general addressing modes and the references. Also they enable each program to have in clear the elements it uses.
RES 16	RES 16	RES 16	
C1 DATA 2	C1 DATA 2	C1 DATA 2	
C2 RES 4	C2 RES 4	C2 RES 4	
C3 DATA D1	C3 DATA D1	C3 DATA D1	
C4 RES 2	C4 RES 2	C4 RES 2	
C5 DATA C2	C5 DATA C2	C5 DATA C2	
FIN	FIN	FIN	
LDS1 LDS			
RES 2			
D1 DATA C1			
FIN			
LPS1 LDS LDS1			
DEB1 LDA D1			
CLS LPS2			
CLS LPS3			
CSV M:EXIT			
FIN DEB1			
END LPS1			

Example 3 :

First module	Second module	Third module	Remarks
	LDS2 LDS	LDS2 LDS DUM	The two program segments LPS2 and LPS3 are linked to the same data segment LDS2. They are separely assembly, but one of the two references a dummy segment DUM which also uses to satisfy the local addressing modes and allows the programmer to have in clear the elements he uses. The DUM segment do not generate any object code.
	RES 2	RES 2	
	D2 DATA 4	D2 DATA 4	
	RES 5	RES 5	
	D3 DATA C2	D3 DATA C2	
	D4 DATA C4	D4 DATA C4	
	FIN	FIN	
	LPS2 LPS LDS2		
	DEB2 LDA D2		
	LDX =2		
	STA Ch,x		
	RTS		
	FIN DEB2		
	END		
		LPS3 LPS LDS2	
		DEB3 LDA D4	
		LDX = 0	
		STA C4	
		RTS	
		FIN DEB3	
		END	

After these three modules be linked, a executable IMT of the following forme will be obtain.

CDS PROG	LDS LDS1	LPS LPS1	LDS LDS2	LPS LPS2	LPS LPS3

Running section

III-5. CONSTITUENTS OF A PROGRAM

■ Task Working Block (TWB)

The first sixteen words of the CDS are called the "Task Working Block" or TWB.

This 16-word area is reserved to the Monitor which may store therein the return address to the calling task, as well as the caller's local data base (L) and the program indicators.

The Monitor may maintain in the TWB a pointer to the system's common data area (ZC).

All programs which require Monitor Calls must reserve 16 words at the beginning of their respective CDS.

This feature allows for monitor sections re-entry, the latters operating in the calling program.

Program Relocation Table (PRT)

The sections are assigned through a section relocation double-word (SRD), which contains the initial values of L and P with respect to G :

Section relocation double-word (SRD)

0 — 15
l = L - G
p = P - G

The PRT is made up of all the SRD of the program sections.

This PRT is stored in the locations immediately preceding G-address, thus the SRD of section no. n has an address given by :

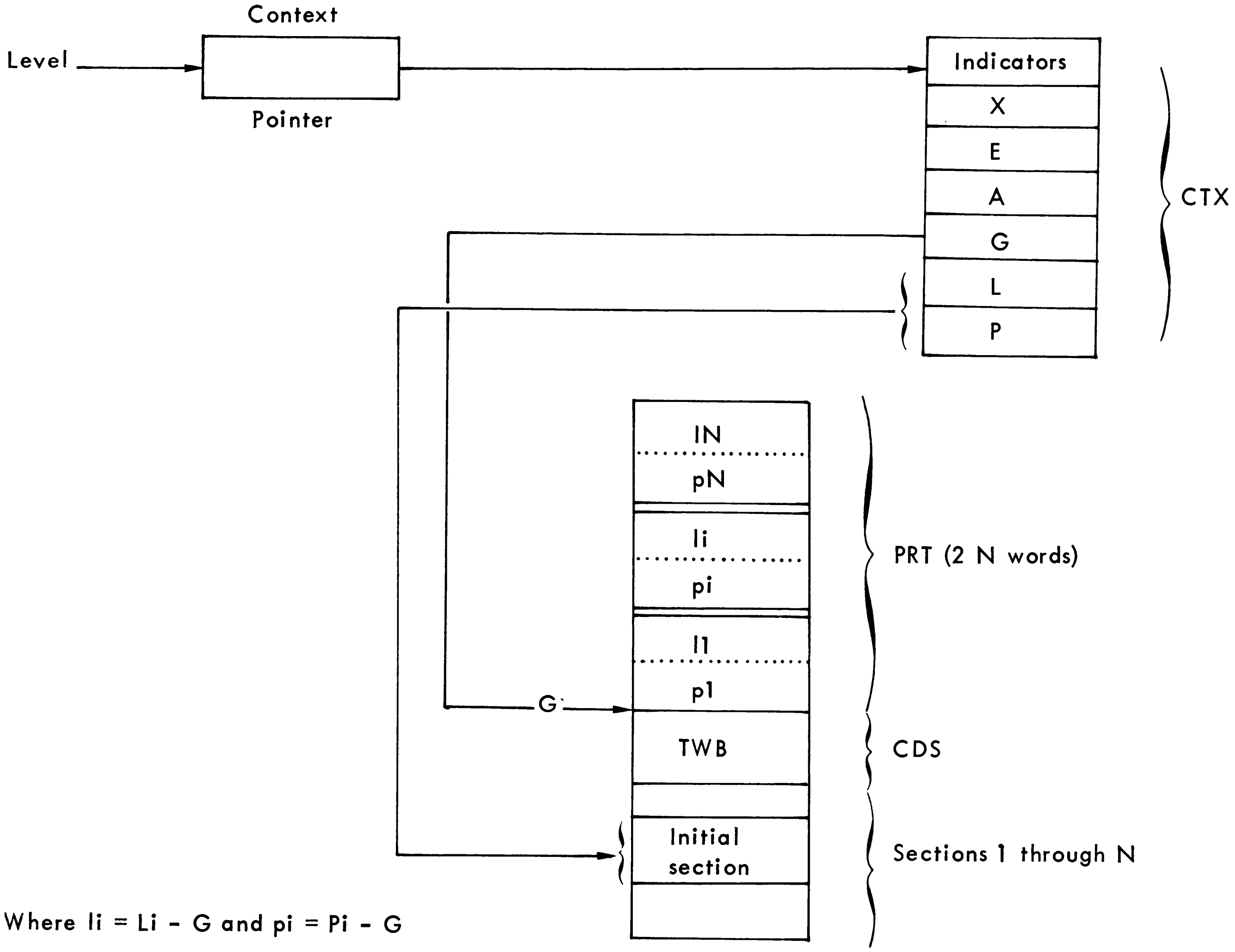
G - 4n

This table is built at linkage edition time.

Note :

The Monitor's PRT is pointed to by the contents of a fixed address as that of the micro-processor (address 12).

The PRT is the communication area between the different sections of a same program or between a program and the Monitor (for the Monitor's PRT).



Where li = Li - G and pi = Pi - G

Structure of a program

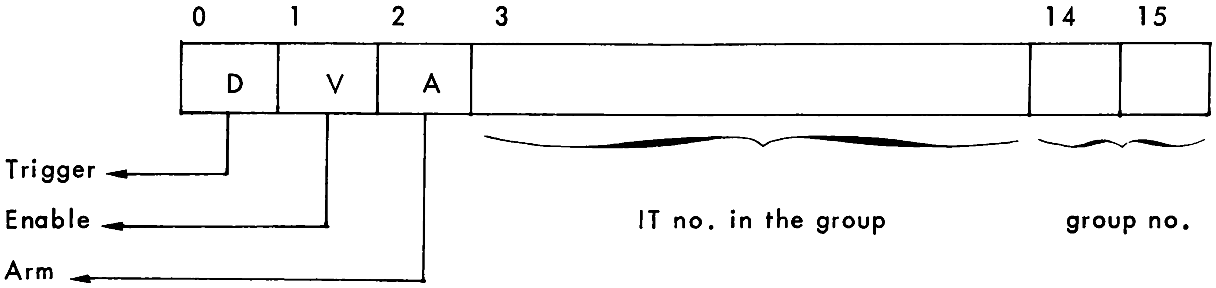
Remark :

The CDS, which is accessible from any section of a program, constitutes an implicit communication area between the sections.

■ De-activation word table or DeVice Table (DVT)

This 32-word table precedes in core memory the Context Pointer Table (CPT) which is also 32-word long.

A DVT word has the following format :



Bits 3 to 15 are also called "interrupt configuration".

The interrupt system and the DVT are described in Chapter II.

■ Context (CTX)

The context is the communication area between a priority level and an associated program. It groups seven words :

Word 1 : Status indicators

Word 2 : Initial X value

Word 3 : Initial E value

Word 4 : Initial A value

Word 5 : Initial G value

Word 6 : Initial L value

Word 7 : Initial P value

The context is used for initializing and restarting a task, and for protecting its status when the corresponding level is activated or de-activated.

When activated, a task level defines in the context table (CTX) the specific pointer fo the associated context. P-, L-, G-, A-, E- and X-registers, as well as the status indicators are loaded from the context area and program execution begins at address P.

Conversely, when a level is interrupted by a higher priority level, or when it is acknowledged, the current contents of P-, L-, G-, A-, E- and X-registers and of the status indicators are stored in the context area.

For further details, see DIT instruction description (Chapter VII).

III-6. SECTION CALLS

There are two kinds of sections :

- sections pertaining to a given program, accessible through a CALL SECTION (CLS).
- sections available to all programs : supervisor section or common library section, accessible through a CALL SUPERVISOR (CSV).

■ Program section call (CLS instruction)

During the execution of a CLS instruction, the processor :

- stores the contents of P (program address) and L (local data base) in the first two words of the called section's local area (after subtracting G-base). These elements are required for "returning" to the task and therefore must be saved.

- Loads P- and L-registers with the starting address and the local data segment address, respectively, of the called section which may then be executed.

During the execution of a RTS (ReTurn Section), the processor :

- Restores in P- and L-registers the values which had been saved at the beginning of the called section's local segment.

Note :

When several sections of a program are separately assembled, if one contains a call to another, it is not necessary to declare that the calling section is external to the module. This declaration is implicit and the linkage editor performs the necessary checks.

The transfer diagram is given in the description of CLS instruction (Chapter VII "Instructions").

■ Supervisor call (CSV instruction)

The supervisor sections and the sections constitued by common sub-programs make up the "resident operating system".

Hereafter, we shall call "system section" a section of the operating system.

A "system section" :

a) remains at the calling program's priority level;

b) processes both the calling task's data and its own local data;

c) is automatically executed in master mode.

Moreover, since a task is identified by its G-base value it is logical to associate the call with this base rather than L-base.

In the CALL SUPERVISOR, i.e. in a system section, G has the same function as L in the CALL SECTION.

Paragraph (c) above, which is associated with class 0 addressing modes (see chapter V), implies paragraph (b) since a system section may :

- access its own data in LD, LI and LIX addressing modes, it being understood that these data have absolute addresses and, therefore, system sections are resident with an implicit zero local base. (In this respect, the operating system is a single program).

- access the calling task data in GD and GIX addressing modes, since the general base G remains that of the calling task.

When executing a RETURN SUPERVISOR instruction (RSV), the processor restores in L- and P-registers the values which had been previously saved in the calling program's TWB.

The mode of the calling program is automatically re-established by RSV instruction.

The communication diagram of a supervisor call is given in the description of CSV instruction (Chapter VII).

Example of re-entrant section programmation

M:MOVE module of MOB Monitor for moving a byte string.

- Main program

```
PRINC       CDS                              TWB
            RES        16
            FIN

LDS1        LDS
CH1         TEXT       "ABCDEFGHIJKL"
CH2         RES,1      12
            FIN

LPS1        LPS        LDS1
DEB         LEA        CH2
            XAX                              Parameters loading
            LEA        CH1
            LDE        =12
            CSV        M:MOVE
            FIN        DEB
            END        LPS1
```

- M:MOVE re-entrant module

```
FICTIV      CDS        DUM
            RES        4
T0          RES        1
T1          RES        1
T2          RES        1                     Dummy TWB. Generates no object code.
T3          RES        1                     Used for proper generation of instruction displacements
T4          RES        1                     in the LPS.
T5          RES        1                     Since a CSV does not alter G-base value, M:MOVE
T6          RES        1                     will operate in the main program's TWB.
T7          RES        1
N3          RES        1
N2          RES        1
N1          RES        1
N0          RES        1
            FIN

SUPER       LDS
            FIN

M:MOVE      LPS        SUPER
            SPA        #N0                   Entry point of the module for the CSV.
            BRU        $+2
            RSV

R:MOVE      XEX                              Entry point of the module for a branch instruction
            DST        #T0

C           DCX        =1
            BCF        O
            LBR        @ #T1,X
            SBR        @ #T0,X
            BRU        C
```

```
O               LDA         # T1
                BRU         @ # N0
                FIN
                END
```

III-7. SYSTEM'S MANAGEMENT CONCEPTS

■ Functions of the Supervisor

- Task management : connection at interrupt levels, queuing and "distribution" functions.

- Input/Output management : initialization on user's call, checking of interrupt-initiated transfers termination, etc...

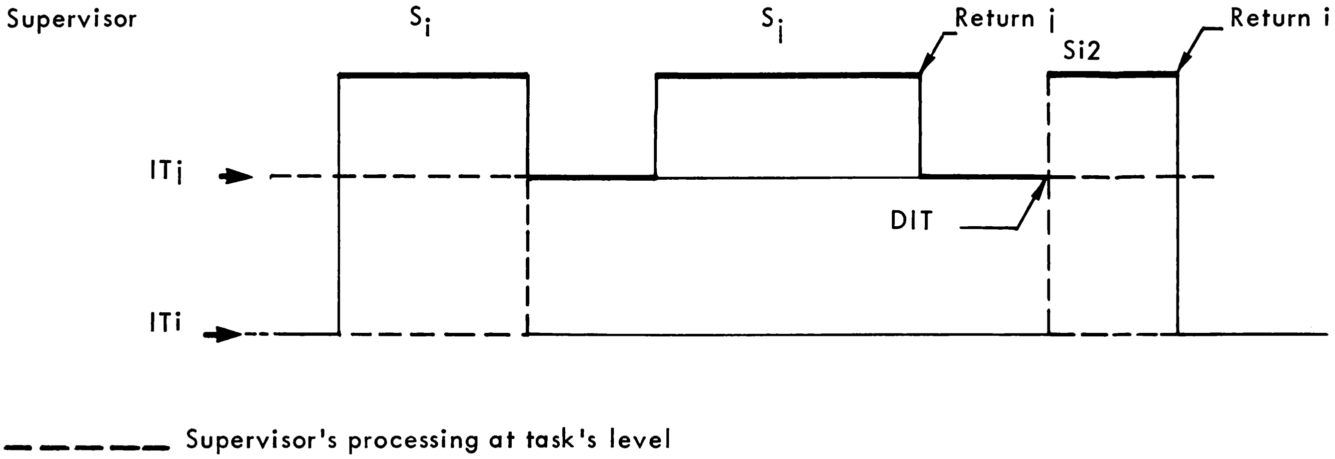
- Resource management : reservation and release on user's call.

- Event management.

- Delay management.

- Etc...

Operation and re-entrance of Supervisor calls



The Supervisor operates at the level of the calling task and this provides supervisor context protection at this level.

Besides, for full re-entrance capability, variable data operated upon by the Supervisor must be stored in an area specified by the calling task : this is the purpose of the TWB described in paragraph III-5.

■ Data area management

- System common data area

In order that a relative address may always be positive with respect to any G-base value, this area is located in the upper portion of the memory.

It comprises a group of fixed-length blocks which are dynamically assigned on user's request.

To have the program relocatable with respect to this area, it is the address ZC of the whole common area which the Supervisor stores in G + 6, in relative value with respect to G, the address of the actually assigned block being provided in a register (preferably X) as a relative value with respect to ZC. This address is assigned by the Loader.

The task will address this block in GIX mode with :

- D = 6,
- (G + 6) = ZC,
- (X) = Block address with respect to ZC.

To progress in the block, the task increments or decrements X-register.

This procedure has an obvious advantage :

When a program is dynamically relocated, e.g. after a swapping, the only action of the system is to update (G + 6) contents with the new ZC relative address to provide the connection with its data, without any attention from the user.

Thus, the main purpose of this area is :

- to provide communication between separate programs,
- to allow for dynamic relocatability of the programs.

- Program common data area

Every program includes a common data section (CDS) accessible in general addressing mode. This addressing mode. This addressing being always relative to G-base, the actual location of the program may be unknown to the user without any influence on the programmation.

- Local data areas

Since every program section may have a local data area, the corresponding base must be updated at the beginning of the section (when a called section is entered) or upon return to the calling section.

This updating is automatic and requires no attention from the programmer who needs only state the name of the section to be executed.

The linkage editor builds a relative location table in which every section is defined by the relative addresses of its entry point and local data segment.

Some programs may require a direct access to more than 256 bytes in a data segment; this is provided for by instructions for incrementing and decrementing L-base, where by the direct access area is shifted. The Assembler is made aware of such shifting by a BASE pseudo-instruction.

4. Assembly language

The Assembler is a language translation processor which converts a source program, written in "symbolic assembly language", into an object program.

The programmer is assisted in its task by the following convenient features :

- assembly "pseudo-instructions" for generating data of various kinds,
- possibility of sharing the job between several programmers (program divided into segments and sections),
- possibility of easily writing re-entrant sub-routines owing to full separation of data and work areas.

The source program is processed in a single assembly pass during which :

- every source line is read,
- symbols are entered into tables,
- relative addresses are assigned at the beginning of declared segments,
- pseudo-instructions are executed,
- the "relocatable binary" (RB) object text is edited along with a directory of satisfied references, the object listing and a list of errors which have been detected at this level,
- "forward" or downstream references, which cannot be solved by the Assembler, are actually processed by the Linkage Editor.

MITRAS 1 Assembler requires the following minimum hardware resources :

- 4 K-words of core memory (including I/O processing), and
- a console typewriter (Teletype ASR33).

MITRAS 2 Extended Assembler requires an additional 4 K-word memory module.

Remarks :

1. Hereafter the features which are available with MITRAS 2 only are distinctly pointed out by a vertical dotted line in the margin.

Pseudo-instructions which are not accepted by MITRAS I are underlined.

2. MITRAS 1 Assembler requires about 4800 bytes (without label table) thus leaving, under MOB basic monitor, about 1000 bytes of table space, i.e. 100 common labels.

Source language instructions are of two kinds :

- Machine code instructions, which are each converted into a single machine word specifying an instruction executable by MITRA 15's internal logic. In the following they will be called "instructions".
- Assembly instructions which are command statements controlling the assembler either for assembly procedure, or for data or text generation. In the following they will be called "pseudo-instructions".

IV-1. SOURCE LINE FORMAT

IV-1.1. Instruction or pseudo-instruction line

An instruction or pseudo-instruction line has a maximum of four fields :

- a label field :

Always beginning at column 1 and containing a 1 to 6-character symbol beginning with an alpha character and ending with a blank column.

- a command field :

Beginning at the first non blank column after the label field (or at the first non blank column after column 1 when the label field is unused) and ending with a blank column. This field must contain a command statement both for an instruction and a pseudo-instruction.

- an argument field :

Beginning at the first non blank column after the command field and ending with a blank column, except if the first non blank column contains a special character "*" in which case this field is ignored as such.

The argument field cannot extend beyond column 57 with MITRAS I and column 72 with MITRAS 2.

- a comment field :

Beginning at the first column after the special character "*".

IV-1.2. Comment lines

A comment line is a line the first non blank character of which is a special character "*". These lines are ignored by the Assembler but appear in the object listing.

IV-1.3. Blank lines

Blank lines are accepted and treated as empty comment lines.

IV-2. BASIC CHARACTER SET

The Assembler accepts all the following characters :

- alphabetic characters : letters A through Z and ":".
- numeric characters : digits 0 through 9.
- special characters : blank + - * / . , ( ) " = # $ % & @ etc.

Furthermore it accepts all characters recognized by the peripherals. These characters make up a subset of EBCDIC.

No check is performed on the characters which are included in a comment field or a byte string.

IV-3. SYMBOLS

A symbol is an identifiable group of up to 6 alphanumerical characters, the first of which is alphabetical. No blank or special characters are allowed.

A symbol is defined when it appears in the label field of a source line.

In all cases, a symbol identifies the source line to which it belongs.

It may also identify the memory address of the code generated by the source line. In such a situation, a numerical value is assigned to the symbol and is equal to the most significant byte memory address.

IV-3.1. Pseudo-instructions prohibiting assignment of a value to the label

Those are :

GOTO, BASE, BND, DEF, REF, FIN, END, PAGE

A symbol may appear in the label field, but no value is assigned to it.

Its only purpose is to mark the corresponding line in the argument field of a GOTO pseudo-instruction.

IV-3.2. Commands for assigning an address value to the label

- Assignment commands :

the EQU pseudo-instruction provides for assigning a numerical value to the symbol in a label field.

- Generation commands :

. Machine instructions
. Generation of pseudo-instructions :

RES, DATA, GEN, TEXT, DO.

. Segmentation pseudo-instructions :

CDS, LDS, IDS, LPS, BASE.

Any symbol appearing in the label field of such a command is entered into the assembly symbol table and an address value is assigned to it.

The address value is always relative to the beginning of the segment which contains the symbol in a label field.

An address value specified in operand field of DATA and GEN pseudo-instructions will be relocated, at linkage edition time, by the value of L or P base of the segment in which it has been defined, so as to become relative to the general base G of the program.

In resident programs declared in Master Mode, the loader will generally relocate the address values by the general base G, since, in that case, local mode indirect addresses must be absolute.

However, in a LDS, it is possible to force a label expression to remain relative to the base, even for a program executable in Master Mode.

For this, the label expression must be preceded by the special character "#" .

This procedure is allowed in a CDS, though it is basically ineffective.

IV-4. CONSTANTS

Data may be directly entered in assembly language as alphanumerical constants. Three types of constants are permitted in statements :

IV-4.1. Decimal integer constants

A decimal integer constant is represented by a decimal integral number of 5 digits or less, with or without a sign :

Example : +75 75 -75

The maximum absolute value for an unsigned number is $2^{16} - 1 = 65,535$.

The constant generated by the Assembler is in pure binary form (in two's complement for negative values) and occupies the area specified in the generation pseudo-instruction.

IV-4.2. Hexadecimal constants

A hexadecimal constant is represented by an integral hexadecimal number of 4 digits or less, preceded by the special character "&".

Example : &1A &E3FF

IV-4.3. Character string constants

A character string constant is a sequence of alphabetical, numerical or special characters in quotation marks. The internal representation of normalized characters is "EBCDIC".

A translation module included in all standard monitors provides for automatic translation ASCII-EBCDIC and EBCDIC-ASCII, should they be required; this translation is performed by the input-output system.

Example : "CHARACTER STRING"

A quotation mark is represented in the string by two consecutive quotation marks.

Example :

"NEXT""CHARACTER" represents : NEXT"CHARACTER

IV-5. EXPRESSIONS

An expression is made up of one or several symbols or constants combined through arithmetic operators.

An expression is represented by a single value which is computed by the Assembler or by the Linkage Editor according to the rules specified in section IV-3.2.

An expression is said to be computable when its value may be determined at the first encounter; therefore, it must contain no forward or external references.

IV-5.1. Operators

The Assembler accepts the following operators :

"Minus" unary operator (example : -3)

Subtraction operator (example : A-3)

+ Addition operator

When the unary minus operator is followed by a constant, the Assembler generates the latter in pure binary two's complement form.

IV-5.2 Expression evaluation

Two kinds of expressions are to be considered :

- label expression :

A symbol identifying a specific memory location whose address is the value of the label.

Such a symbol may be reduced to the special character $ in which case it specifies the current location counter value.

- Predefined symbol :

A predefined symbol specifies no memory location; its value is absolute and defined by EQU pseudo-instructions preceding its utilization.

- (Forward) reference

A forward reference is a symbol which has not yet been defined. It may be defined later on either by a label, or through an EQU or REF pseudo-instruction.

- Conventional representations :

Elements of assembly language syntax are represented by their denominator contained between square brackets (e.g. : <expression>).

The definition is given as an identity relation the lefthand portion of which is the representation of the elements to be defined. The identity symbol is ": : =" and the righthand portion specifies the various compositions of the elements to be defined. When this portion contains several elements in succession, the latters must appear in the same order. However, when such elements are separated by "slash marks" (/), one must select one or the other.

Example 1 :

<ab> : : = - <value>/<value>

In this case, "ab" may be indentical with "-value" or "value".

Example 2 :

<value> : : = <term>/<value><sign><term>

In this case, "value" may be identical with "term" or "value" followed by "sign" followed by "term". This is equivalent to the statement that "value" is a sequence of "term" separated by a "sign".

MITRAS I Version

<Term> : : = <constant>/<predefined symbol>

<Constant> : : = <integral decimal constant>/<integral hexadecimal constant>

<Label expression> : : = <label>/<label><sign><term>

<Reference expression> : : = <reference>/<reference><sign><term>

<Sign> : : = + / -

<Predefined expression> : : = <term>/-<term>/<label expression>

Predefined expressions are always computable by the Assembler. Some expressions may be computable at linkage edition time only.

MITRAS 2 Version

<Constant> : : = <integral decimal constant>/<integral hexadecimal constant>

<Term> : : = <constant>/<predefined symbol>

<Displacement> : : = <label>-<label>

<Value> : : = <term>/<displacement>/<value>+<displacement>/<value><sign><term>

<Label expression> : : = <label>/<label><sign><value>

<Reference expression> : : = <reference>/<reference><sign><value>

<Predefined expression> : : = <value>/-<value>/<label expression>

<Expression> : same structure as for "predefined expression", but any label may be replaced by an address reference.

Predefined expressions are always computable by the Assembler. Some expressions may be computable at linkage edition time only.

Remark :

A label may be reduced to the special character $ (current value of the location counter) but only as the first term of an expression.

Example of such label expressions :

$ + 2	valid
2 + $	invalid

5. Addressing modes

V-1. SYMBOLIC REPRESENTATION OF THE INSTRUCTIONS

V-1.1. Representation conventions

Hereafter the following representation conventions will be used :

- $\left\{\begin{matrix} = \\ \vdots \end{matrix}\right\}$ One of the terms between braces may be specified and excludes all others (possible terms are stacked vertically).

- $[\quad]$ The term between square brackets may be omitted being either optional or implicit.

- ... The expression bounded by the end separator immediately preceding the ellipsis mark and the associated begin separator may be repeated.

Examples :

$$\left\{\begin{matrix} A \\ B \\ C \end{matrix}\right\} \quad [,D]$$

One term out of A, B and C must be specified, D is optional.

$$\left\{\begin{matrix} [A] \\ B \\ C \end{matrix}\right\}$$

One term out of A, B and C must be specified, but A may be omitted when selected.

$$\left\{\begin{array}{l} A \quad [,B] \ldots, C \\ D \end{array}\right\} \ldots$$

The expression between braces may be repeated; in the first possible term, B element is optional but may be repeated.

V-1.2. Instruction representation

All instructions are represented in accordance with the following format :

$$[\text{label}] \quad \text{OP} \quad \left[\left\{\begin{array}{lll} \left\{\begin{matrix} = \\ @ \end{matrix}\right\} & D & [,X] \\ [\#] & D & \\ @\# & D & ,X \end{array}\right\}\right]$$

Wherein :

OP : Operation code

D : Displacement

= : Immediate addressing (parameter) operand value = displacement

@ : Indirect addressing

# : Relative addressing with respect to general base (CDS)

,X : Indexing

Remark :

For instructions or pseǜdo-instructions whose name has four characters, the first three only are used for operation code recognition purpose.

V-2. ADDRESSING MODE REPRESENTATION

V-2.1. Addressing class

MITRA 15 addressing capabilities are adapted according to the various instruction operation codes.

Addressing functions may be classified into three main groups corresponding to three instruction classes :

- Class 0 instructions

These instructions control the following operations

- register load and store operations
- fixed-or floating-point arithmetic operations
- logical operations
- byte string operations
- comparaison

- Class 2 instructions

These are conditional or unconditional branch instructions.

- Class 1 instructions

- shift operations
- index operations
- base operations
- section or supervisor calls
- input/output operations
- register operations
- interrupt and interrupt masking operations

These three groups make up a very comprehensive instruction set which will be discussed later on after a brief description of addressing forms pertaining to each type.

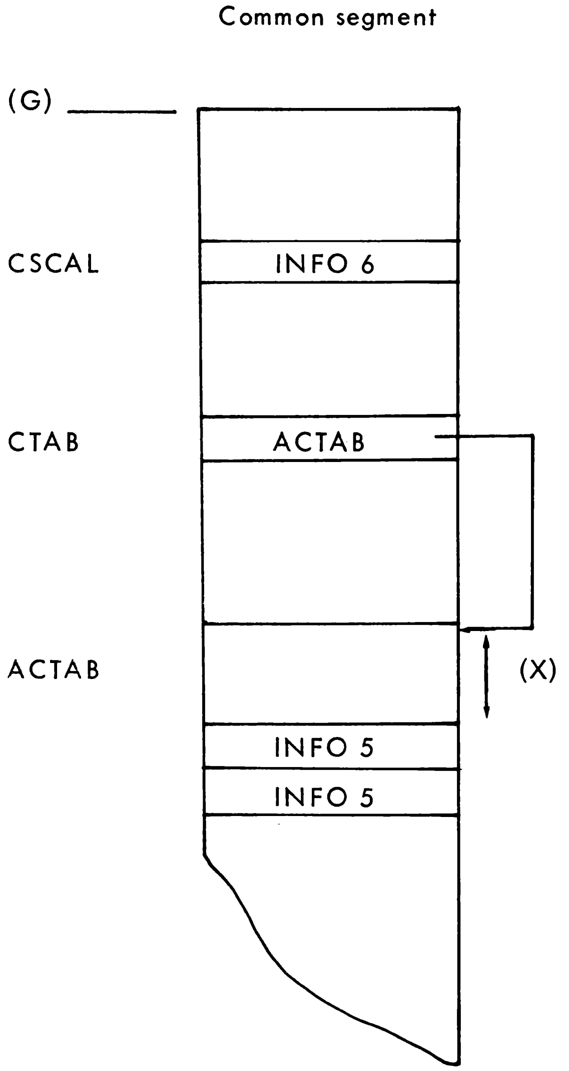
The following conventions are used in the dicussion :

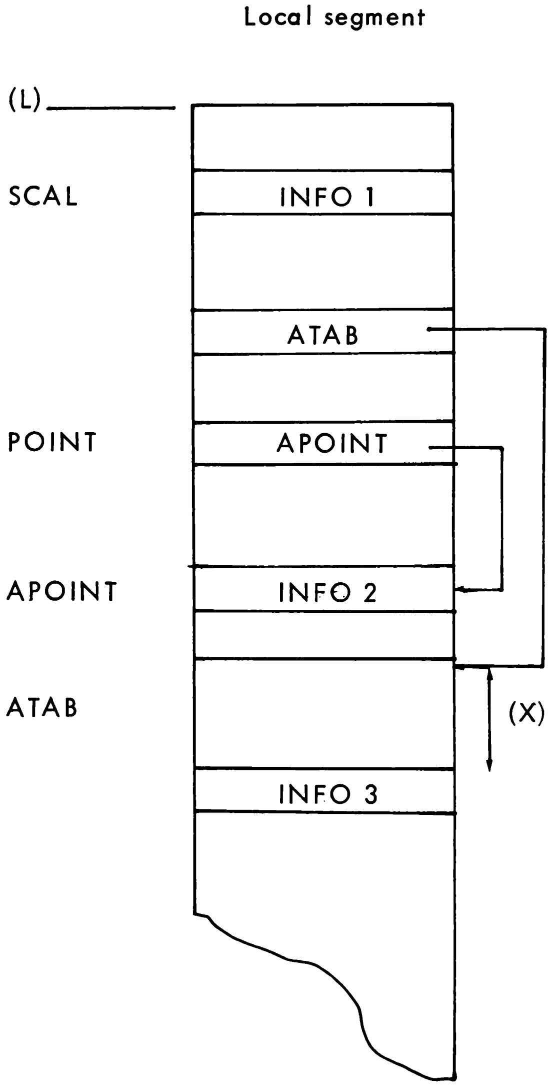
- L Local base
- G General base
- G' General base in slave mode or zero in master mode
- X Index register
- P Program base
- D Displacement
- ( ) Contents of

■ Class 0 addressing

Mode	Assembly language	Addressed data	Addressing function
Direct, Local DL	IDENT	Byte, word or double-word located in the first 256 bytes of the local segment.	Y=(L)+D
Indirect, Local IL	@ IDENT	Byte, word or double-word located anywhere and pointed at through the local segment.	Y=G'+((L)+D)
Indirect, Local, Indexed ILX	@ IDENT,X	Element of a byte, word or double-word array located anywhere and pointed at through the local segment.	Y=G'+((L)+D)+(X)
Direct, General DG	# IDENT	Byte, word or double-word located in the first 256 bytes of the common segment.	Y=(G)+D
Indirect, General, Indexed IGX	@#IDENT,X	Element of an array pointed at through the common segment.	Y=(G)+((G)+D)+(X)
Parameter or immediate P	=OPERAND	A 1-byte operand is specified in the instruction. This byte may be extended on the left by 8 leading zeroes, if required.	(Y) = D Y = (P)

Example of class 0 addressing





Instruction	Operand
LDA #CSCAL	INFO 6
DLD @#CTAB, X	INFO 5

Instruction	Operand
LDA SCAL	INFO 1
LDA @POINT	INFO 2
LBR @TAB, X	INFO 3

Instruction	Operand
LDA = INFO 4	INFO 4

■ Class 1 addressing

This class includes :

- either instructions without actual operand, i.e. register contents swapping, section end, etc...

- or instructions whose operand is generally known (possibly through an unknown modifier) at programmation time : shift, increment, index, etc.

The following modes are permitted :

Mode	Assembly Language	Operand	Addressing function
Parameter or immediate P	=PARAM	Operand defined by displacement value	(Y)=D Y=(P)
Parameter, Indexed PX	=PARAM,X	Operand defined by value plus X-register contents.	(Y)=D+(X) Y=(P)
Direct, Local DL	IDENT	Operand located in the first 256 bytes of the local segment.	Y=(L)+D

Remark 1 :

To simplify program writing, a number of symbolic instruction codes recognized by the Assembler specify both the operation code and the displacement.

For example SRG, which is a register instruction, is specified through its displacement :

SRG	= 02	exchange A and E
SRG	= 04	A and X
SRG	= 06	E and X
⋮		
SRG	= 1C	$-A \longrightarrow A$
⋮		
etc.		

In actual practice, for the Assembler,

XAE	is equivalent to SRG = 02
XAX	SRG = 04
XEX	SRG = 06
⋮	⋮
CNA	SRG = 1C
⋮	
etc.	

In cddition, MITRAS 2 Assembler recognizes 14 shift instruction mnemonics which specialize the two operation codes SHR and SHC.

Examples :

SHR = &23	equivalent to	SRCS = 3	(shift, right, circular, single)
SHR = &E8	equivalent to	SRCD = 8	(shift, right, circular, double)
SHC = &0B	equivalent to	SLLD = 11	(shift, left, logical, double)
SHC = &4E	equivalent to	SRLD = 14	(shift, right, logical, double)

Remark 2 :

Instructions CLS and CSV may be used in two different ways :

a) The operand is a LPS name; the Assembler generates a blank word and the Linkage Editor determines, one the one hand, if the instruction to be generated is a CLS or a CSV according to the section type (monitor or user section) and, on the other hand, the corresponding section number.

This is the normal case wherein the user is not concerned with the section number.

b) The operand is not a LPS name. These instructions are treated as any class 1 instruction, the three addressing modes being available.

Example for a program module

```
PROG      CDS
          RES     16
          FIN

L1        LDS
          FIN
P1        LPS     L1
          RTS
          FIN     P1

L2        LDS
          FIN
P2        LPS     L2
          RTS
          FIN     P2

L3        LDS
          FIN
P3        LPS     L3
          RTS
          FIN     P3

L4        LDS
          RES     3
NUMSEC    DATA    3
          FIN
P4        LPS     L4
DEB       CLS     S1       ───►Call S1       ⎫
          CLS     =2       ───►Call S2       ⎪
          LDX     =1                         ⎬ These utilizations require the knowledge
          CLS     =2, X    ───►Call S3       ⎪ of the section number in the program's PRT
          CLS     NUMSEC   ───►Call S3       ⎪
          CSV     M:EXIT   ───►Call monitor  ⎭
          FIN     DEB
          END     P4
```

Note : CLS NUMSEC is not available with MITRAS 1.

■ Class 2 addressing

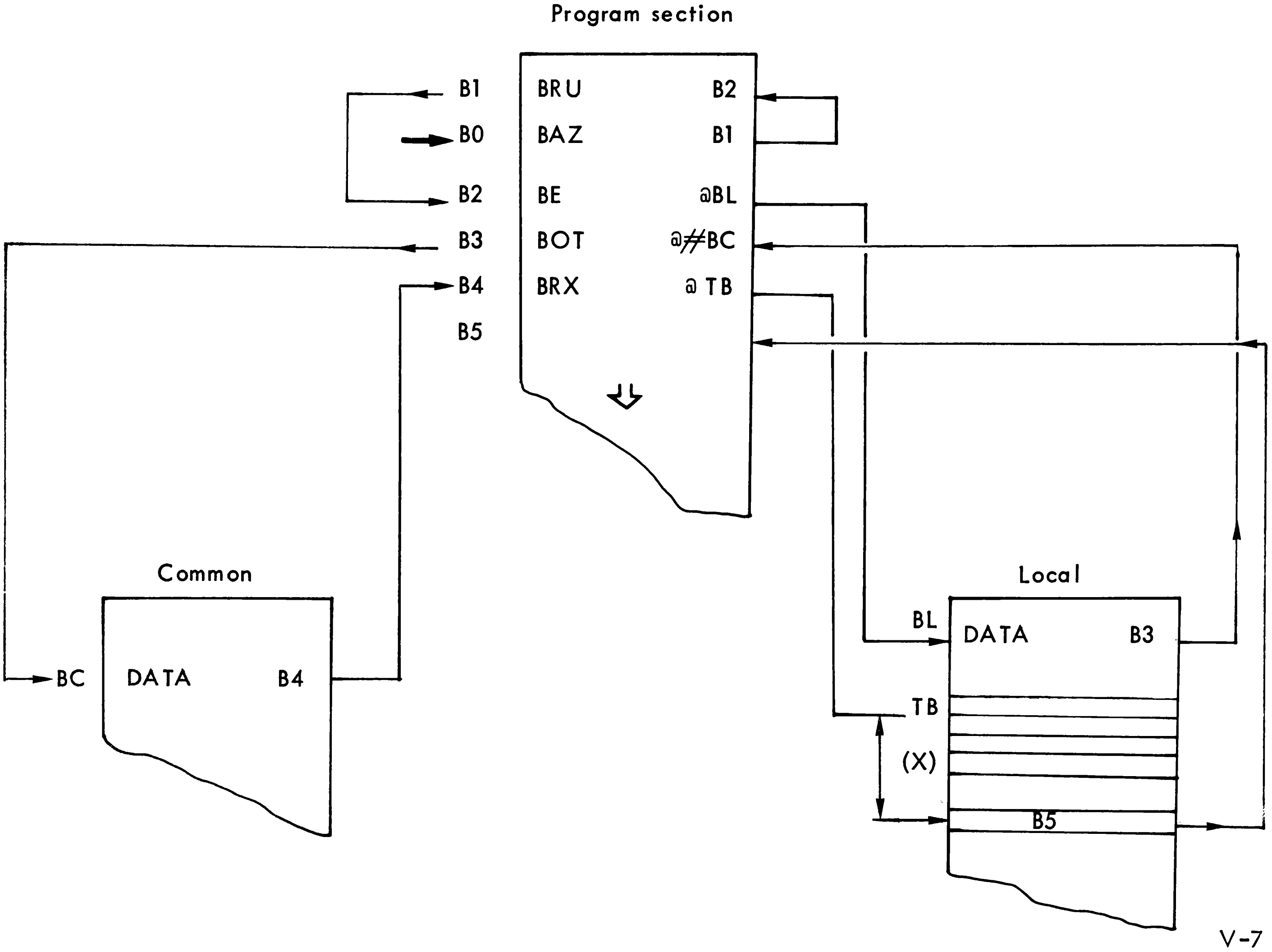
Normally, instructions pointed at by a branch instruction belong to the same section as the branch instruction. However program section length is unlimited.

Four addressing modes are permitted :

Mode	Assembly language	Branch instruction	Addressing function
Relative downstream (plus) RP	LABEL	Any instruction within 512 bytes downstream	Y=(P)+2D
Relative upstream (minus) RM	LABEL	Any instruction within 512 bytes upstream	Y=(P)-2D
Indirect, Local IL	@LABEL	Any instruction pointed at through the local segment.	Y=G'+((L)+D)
Indirect, General IG	@#LABEL	Any instruction pointed at through the common segment.	Y=G'+((G)+D)

In addition, an indexed unconditional branch instruction is also available. For indirect branch instructions, the index is used for pre-indexation (more convenient for "Branch table" processing), contrary to data indexation which is a post-indexation (more convenient for accessing element of an array).

Examples :



V-2.2. Permitted expressions

- Class 0

Predefined or reference expression

- Class 1

P	:	value
PX	:	value
DL	:	predefined or reference expression

- Class 2

RP	:	reference expression
RM	:	label expression
DL	:	predefined or reference expression
DG	:	predefined or reference expression

6. Pseudo-instructions

VI-1. SOURCE TEXT SEGMENTATION

VI-1.1. General

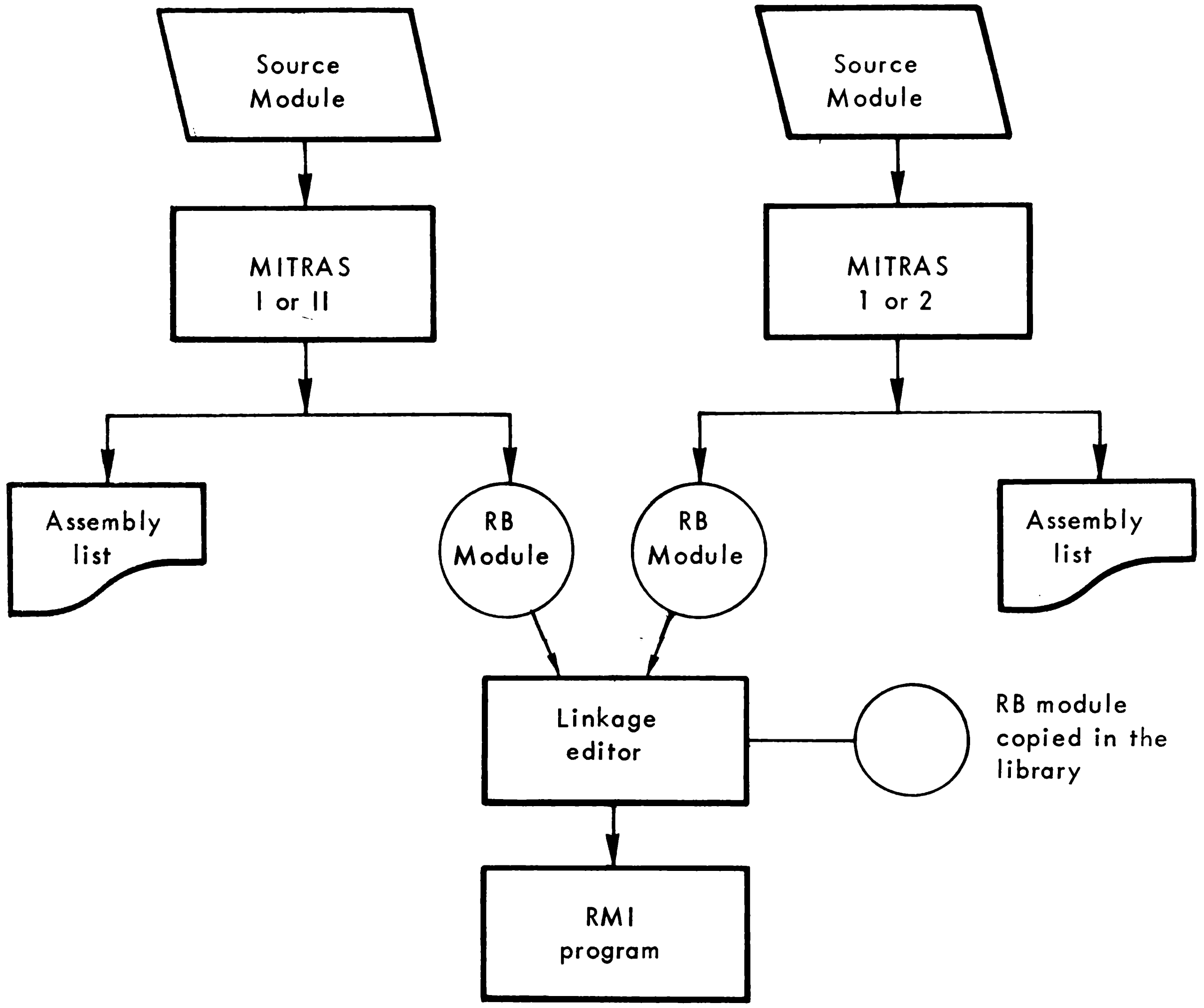
The source text is translated by the Assembler into an object module in "relocatable binary" format (RB). The Assembler can only satisfy the reterences to symbols of the assembled source text.

The Assembled modules are converted by the Linkage Editor into a complete program represented by a "relocatable memory image" (RMI). All external references (section names) between program modules are then satisfied.

The RMI is loaded into core by the Loader starting from a general base address G which is only defined at loading time.

VI-1.2. Source text

The source text is the assembly unit. It comprises one or several segments and must be terminated by an END pseudo-instruction.



VI-1.3. Common data section

If the assembly module is to include an actual or dummy common data section (CDS), the latter must always be declared before any other segment of the assembly module.

VI-1.4. Sections

Every section must include an executable local program segment (LPS) which defines the section. A local data segment (LDS) is normally associated with a LPS.

When such a LDS is actually defined in the same module as the LPS, it must precede the latter in the source text. Several LPS may be associated with a single LDS and the number of sections is equal to the number of LPS.

VI-1.5. Identifier scopes

Identifiers may be classified into internal labels defined within the assembly module, and external labels declared through DEF and REF pseudo-instructions.

■ Internal labels

• Labels defined in the CDS

These labels are defined for the whole assembly module and may be referenced from any segment. However, they connot be redefined as local labels without causing a "double definition" error.

• Labels defined in a LDS

These labels are defined until the appearance of another LDS pseudo-instruction.

They may be referenced from the CDS, from the LDS itself and from any LPS following the LDS in which the label is defined, up to a new LDS.

• Labels defined in a LPS

They may be referenced from LPS itself, from the CDS or from the associated LDS (which normally precedes the LPS).

■ External labels

A label is said to be "external" when it has a meaning outside the assembly module in which it has been defined (where it appears in label field).

Thus, being known at linkage edition time, it provides a convenient link to other modules without resorting to a CDS (actual or dummy) or to a Call Section.

A label is external when declared through a DEF pseudo-instruction which must appear in the segment in which the label has been defined.

The external label may be referenced in another module provided that it is declared in the latter module through a REF pseudo-instruction. The REF pseudo-instruction must appear in the segment where the external label is used.

The "external" status does not modify the notions of "common" or "local" labels, for the Assembler.

When a label belonging to the CDS is to be declared in a REF pseudo-instruction, it must be preceded by a "#" special character.

Example :

REF #LAB1, #LAB2, #LAB3

Do not confuse external labels and segment names. The latters, though known outside the assembly module at linkage edition time, are only accessible through Call Section or Call Supervisor.

Example :

```
PROG        CDS
            DEF         ETIQ0
            RES         16
ETIQ0       DATA        1
            FIN

LDS1        LDS
            RES         8
            DEF         ETIQ1
            REF         ETIQ3
ETIQ1       DATA        2
            DATA        ETIQ3
            FIN

LPS1        LPS         LDS1
            REF         ETIQ2
Z           LDA         ETIQ2
            FIN         Z
            END

LDS2        LDS'
            REF         ETIQ1
            REF         #ETIQ0
X           DATA        ETIQ1
ETIQ2       DATA        0
ETIQ3       DATA        4
            FIN

LPS2        LPS         LDS2
Y           LDA         #ETIQ0
            STA         X
            LDA         ETIQ1
            FIN         X
```

Remark :

It is important to remember that, for local external labels, the displacement is generated relative to the LDS of the section containing the corresponding DEF but used relative to the LDS of the section containing the corresponding REF.

VI-1.6. Location counter

The location counter contents is a byte address with a maximum value of $2^{16} - 1 = 65\ 535$.

The location counter is symbolically represented by the special character "$".

This counter is reset to zero at every segment declaration, so that all references calculated at assembly time are always relative to the starting address of the declared segment.

VI-1.7. Segmentation pseudo-instructions

These pseudo-instructions define the assembly module structure in terms of sections and segments.

They are :

- Common data section : CDS
- Local data segment : LDS
- Indirect data segment : IDS
- Executable local program segment : LPS
- End of segment : FIN
- End of module : END

Every segment opened by a segmentation pseudo-instruction must conclude with a FIN pseudo-instruction.

■ CDS/FIN pseudo-instruction

This pseudo-instruction identifies the common data section CDS.

Format :

Label	Command	Argument
<Name>	CDS . . .	[DUM]
[<label>]	FIN	

Result :

- The location counter is reset.
- All labels may be referenced form the module declared sections.
- If DUM option is specified, no code is generated and the section is dummy.

■ LDS/FIN pseudo-instruction

This pseudo-instruction identifies a local data segment LDS.

Format :

Label	Command	Argument
<Name>	LDS . . .	[DUM]
[<label>]	FIN	

Result :

- The location counter is reset.
- "Name" defined in label field is an implicit external definition.
- If DUM option is specified, no code is generated and the segment is dummy.

■ IDS/FIN pseudo-instruction

This pseudo-instruction identifies an indirect access data segment within a LDS or CDS. This segment is such that any label located between the IDS pseudo-instruction and the associated FIN pseudo-instruction is defined in relative value within the declared indirect segment.

Format :

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

Result :

- The location counter is reset, but its current value is saved.

When the indirect segment IDS has been terminated by a FIN pseudo-instruction, the location counter is restored to its previous value incremented by its current relative value (zero if the IDS is dummy).

- If the DUM option is used, no code is generated and the segment is dummy.

- A label defined in an IDS pseudo-instruction is treated as a normal LDS label for an actual IDS, or as a zero value for a dummy IDS.

■ LPS/FIN pseudo-instruction

This pseudo-instruction identifies an executable program segment LPS and thus a program section.

Format :

Label	Command	Argument
<Name-1>	LPS	<Name-2>
[<label>]	FIN	<label-2>

Result :

- The location counter is reset.

- < Name-1 > identifies the LPS and the section; this is an external implicit definition.

The name referenced in a CALL SECTION is the corresponding external implicit reference.

- <label-2> is the effective starting address for section execution.

- <Name-2> references the associated LDS.

■ END pseudo-instruction

This pseudo-instruction marks the end of an assembly module.

Format :

Label	Command	Argument
[<Label>]	END	[<Section name>]

Result :

- The assembly of the module is terminated.

- <section name> specified in argument field defines the first section executed after program loading. This label constitues an implicit external reference.

- At linkage edition time, only one END pseudo-instruction may be encountered and its argument field must declare a section name.

■ Processing of %EOD

It should be noted that the Assembler's symbolic input file is a source file terminated by a standard end-of-file record (%EOD).

This file end marker is not strictly necessary for program end recognition, however the absence of %EOD will cause an undetermined operation sequence, particularly under linking module control.

An %EOD record detected before an END pseudo-instruction (i.e. in the course of a program) is indicated by a third level error causing immediate assembly abortion.

VI-2. ASSEMBLY PSEUDO-INSTRUCTIONS

These statements are used either to direct common or local data assembly or executable program segment assembly.

Assembly pseudo-instructions are classified as follows :

Pseudo-instruction	Data Segment	Program Segment
1. Addressing		
RES	X	X
BND	X	
BASE		X
2. Symbol definition		
EQU	X	X
3. Assembly control		
GOTO	X	X
DO	X	
PAGE		X

Pseudo-instruction	Data Segment	Program Segment
4. Data generation		
DATA	X	
TEXT	X	
GEN	X	X
5. External definition identification		
DEF	X	X
REF	X	X

VI-2.1. Assembly of a data segment

■ Addressing statements

• RES pseudo-instruction

Reservation of a memory area.

Format :

Label	Command	Argument
[<Label>]	RES [,1]	<Value>
TOTO	RES RES,1	3 5

Result :

- The [,1] option specified in the command field, indicates that the reservation unit is the byte, otherwise it is the word.

- If the selected unit is the word, the location counter is first advanced to the word boundary.

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

• BND pseudo-instruction

The location counter is advanced to a word boundary.

Format :

Label	Command	Argument
[<Label>]	BND	
TOTO	BND	

Result :

The location counter is advanced to an even value, i.e. a word boundary.

■ Symbol definition statement : EQU pseudo-instruction

Format :

Label	Command	Argument
Name	EQU	<predefined expression> /"<character>" /"<character> <character>"
ZON TOTO TATA TUTU TITI	RES EQU EQU EQU EQU	4 ZON ZON+2 5 "AB"

Result :

- An expression specified an argument field defines the symbol which is declared in label field. No forward reference is allowed in the expression.
- One or two alphanumeric characters between quotes (") may appear in argument field.
- The $ symbol representing the current value of the location counter is allowed in argument field.
- Any symbol specified in label field of an EQU pseudo-instruction cannot be redefined.
- No string is allowed with MITRAS I.

■ Assembly control statements

• GOTO pseudo-instruction

Format :

Label	Command	Argument
<Label>	GOTO ,k	<label 1>, <label 2>, <label n>
TOTO NN TATA	GOTO ,2 EQU2 GOTO,NN	BR1,BR2,BR3,...,BRn BR1,BR2,BR3

Result :

- The assembler jumps to the source line whose label field contains the K label declared in argument field of the GOTO pseudo-instruction.

- K is a "value" type expression (see page IV-6 and IV-7) which may be calculated when the GOTO pseudo-instruction is processed.

- All labels declared in argument field must refer to source lines following the GOTO pseudo-instruction.

- When K is not comprised between 1 and n, an error message is edited and assembly is resumed at the line following the GOTO pseudo-instruction.

- If an END pseudo-instruction appears before the selected label is reached, the assembly process is interrupted and an error message is issued.

• DO pseudo-instruction

Iterative assembly of an instruction.

Format :

Label	Command	Argument
[<Label>]	DO	<value>
TOTO	DO	7
	DATA	&78A2

Result :

- The absolute expression declared in argument field must be computable and provides an integral value less than 128 representing the number of iterative assemblies of the next line in sequence.

- The iteration index is symbolically represented by the special character "%" and may be included in the line to be assembled. At each iteration step it is replaced by the iteration counter contents.

- The iteration counter is initially set to 1 during the first loop.

- When the iteration is over, the assembly process is normally resumed at the second line after the DO pseudo-instruction.

- If the absolute expression is negative or zero, or if it is not computable (in the second case, an error message is issued), the Assembler directly steps to the second line after the DO pseudo-instruction.

- When a label is specified in label field, it is associated with the first generated byte.

■ Data and text generation statements

• DATA pseudo-instruction

Data generation.

Format :

Label	Command	Argument
[<Label>]	DATA [,1]	[#] <exp 1> [,[#]<exp 2>] ...
TOTO	DATA	ETIQ1,#ETIQ2,ETIQ1-ETIQ2,3,&8AF2
	DATA,1	7,&8E, 5+4

Result :

- A DATA pseudo-instruction generates data items having the values of "i-expressions".

- An "i-expression" is an expression as defined in IV-5.2. or a string of 1 or 2 characters. (The string is not allowed with MITRAS I).

- Every generated data item is right justified on one or two bytes, according to the selection or omission of [,1] option in command field.

- When a label is specified in label field, it is associated with the first generated byte.

- If <exp 1> is preceded by a # symbol, the corresponding expression must be left relative to G during a master mode loading.

Remark :

Under MITRAS 2, operational labels (standard and user's) are common explicit symbols. Thus, an operational label may be specified in an I/O control block just by writing :

```
DATA,1    M:BI
```

in the case of M:BI operational label.

Under MITRAS I the operational number must be either specified directly or defined through an EQU pseudo-instruction.

For further information on operational labels, see chapter 8 "Input/Output control system".

Example :

```
MITRAS 2
             CDS
              .
              .
              .
             FIN

LDS1         LDS
CB           DATA          0
             DATA,1        &80
             DATA,1        M:EO
             DATA          # STRING
             DATA          8
STRING       TEXT          "WRITING"
             FIN

LPS1         LPS           LDS1
DEB          LEA           CB
             CSV           M:IO
             LEA           CB
             CSV           M:WAIT
             CSV           M:EXIT
             FIN           DEB

             END           LPS1

END OF FILE
```

```
           CDS                              CDS
            :                      M:EO     EQU         6
           FIN                              FIN

LDS1       LDS                     LDS1     LDS
CB         DATA       0            CB       DATA        0
           DATA,1     &80                   DATA,1      &80
           DATA,1     6                     DATA,1      M:EO
           DATA       # STRING              DATA        # STRING
           DATA       8                     DATA        8
STRING     TEXT       "WRITING"    STRING   TEXT        "WRITING"
           FIN                              FIN

LPS1       LPS        LDS1         LPS1     LPS         LDS1
DEB        LEA        CB           DEB      LEA         CB
           CSV        M:IO                  CSV         M:IO
           LEA        CB                    LEA         CB
           CSV        M:WAIT                CSV         M:WAIT
           CSV        M:EXIT                CSV         M:EXIT
           FIN        DEB                   FIN         DEB

           END        LPS1                  END         LPS1

END OF FILE                        END OF FILE
```

- GEN pseudo-instruction

Value generation

Format :

Label	Command	Argument
[<Label>]	GEN, area list	Expression list
	GEN,4,2,2	7,1,1
TOTO	GEN,1,1,3,8,2,1	0,1,2,&F2,3,1

Result :

- The GEN pseudo-instruction generates a byte or word having a specific binary configuration.

- "Area list" is a sequence of term-type expressions each specifying the length (in bits) of an area to be generated. The generated areas must have a total length of 8 or 16 bits, and no zero-length area is allowed.

- "Expression list" a sequence of expressions of the same type as those which are defined by a DATA pseudo-instruction, defining the contents of every generated area. At assembly time, the listed values are inserted in the corresponding areas on a rank basis from left to right (first value in first area, and so on).

All values are right justified in their respective areas. The first area contains the most signifiant value.

The items of area list and expression list are separated by commas.

- TEXT pseudo-instruction

Generation of a character string.

Format :

Label	Command	Argument
[<Label>]	TEXT	" <character string>"
TOTO	TEXT	"CHARACTER STRING"

Result :

- The character string is assemblied in EBCDIC format in an area beginning at the current address of location counter and ending at an address corresponding to the last generated character.

- The first character of the string is stored in the first byte of the area, and so on.

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

- DEF pseudo-instruction

Format :

Label	Command	Argument
[<Label-1>]	DEF	<Label-2>..
	DEF	ETIQ

Result :

- Labels are declared as external definitions.

- Labels are defined in the current section.

- A label must be declared as external definition prior to being used in the section.

- REF pseudo-instruction

Format :

Label	Command	Argument
[<Label>]	REF	[#] <Label>..
ETIQ1	REF REF	ETIQ1 # ETIQ2

Result :

- Labels are declared as external definitions.

- The labels must be defined prior to being used in the current section and will be defined later on at assembly time.

- A label belonging to the CDS must be preceded by the symbol # .

VI-2.2. Program segment assembly

■ Addressing statements

- RES pseudo-instruction

Reservation of a memory area (see section VI-2.1. page VI-7).

- BASE pseudo-instruction

Directs relative addressing.

Format :

Label	Command	Argument
[<Label>]	BASE	[<Label>]
ETIQ ETIQ1	BASE BASE	 ETIQ2

Result :

- The label declared in argument field is a LDS label.

- All LDS labels referenced in the program segment between two BASE pseudo-instructions are generated in relative value with respect to the address specified in argument field.

- A BASE pseudo-instruction without label declaration in argument field simply terminates the relative addressing specified by the preceding BASE pseudo-instruction.

- A new relative addressing requires another BASE pseudo-instruction with a label declaration in argument field.

- A BASE pseudo-instruction with label declaration in argument field terminates the relative addressing of the preceding BASE pseudo-instruction and opens a relative addressing on the new label declared in argument field.

- A BASE pseudo-instruction with label declaration is closed either by a new BASE pseudo-instruction or by a FIN pseudo-instruction which terminates the assembly of the program segment.

■ Symbol definition statement : EQU pseudo-instruction

See section VI-2.1. page VI-7.

■ Assembly control statements

- GOTO pseudo-instruction

Conditional assembly branch (see section VI-2.1. page VI-8).

■ Instruction generation statements

- GEN pseudo-instruction

Generates a value. (See section VI-2.1. page VI-11).

This pseudo-instruction used in an executable program area allows to generates non-standard instructions specific to a special configuration.

If, after GEN pseudo-instruction, the incremented location counter is not on a word boundary (generation of an odd number of bytes), the Assembler signals an error, generates a zero value byte and steps the location counter by one unit.

■ External definition identification statements

- DEF pseudo-instruction

See section VI-2.1. page VI-12.

- REF pseudo-instruction

See section VI-2.1. page VI-12.

VI-3. PAGE PSEUDO-INSTRUCTION

This pseudo-instruction asks for printing the assembly listing on the next page, when the output device is a printer.

Format :

Label	Command	Argument
[<Label>] ETIQ	PAGE PAGE	

7. Instructions

VII-1. GENERAL

This chapter describes MITRA 15's instruction set which is divided into eleven functional categories :

- Load and store instructions
- Fixed-point arithmetic operations
- Logical operations
- Register incrementation
- Shift operations
- Inter-register operations (SRG or set register)
- Floating-point arithmetic operations
- String processing instructions
- Branch instructions
- System communication instructions
- Control instructions

In addition, every instruction has three fundamental attributes :

• Class : indicating the permitted addressing modes and thus the exact meaning of the calculated address in each case.

Class	Permitted addressing modes	
0	DL	Direct, Local
	P	Parameter
	DG	Direct, General
	IL	Indirect, Local
	IGX	Indirect, General, Indexed
	ILX	Indirect, Local, Indexed
0'	DL	Direct, Local
	DG	Direct, General
	IL	Indirect, Local
	IGX	Indirect, General, Indexed
	ILX	Indirect, Local, Indexed
1	DL	Direct, Local
	PX	Parameter, Indexed
	P	Parameter

Class	Permitted addressing modes	
2	RP	Relative, Plus
	RM	Relative, Minus
	IL	Indirect, Local
	IG	Indirect, General

The functions of these addressing modes are described in chapter V.

However, as regards addressing a few instructions have a particular behavior which will be described individually. These instructions are BRX and TES.

- Mode : indicates the operational mode(s) of the CPU in which the instruction may be executed. A so-called "priviledged" instruction is executable in master mode only. These instructions give rise to an additional trap condition : "mode violation".

- Option. Some "optional" instructions are not standard on all CPUs. They give rise to an additional trap condition : "non-implemented instruction".

However SHC, DIV, FAD, FSU, FMU and FDV may be simulated by a special monitor module (TRAP module).

VII-2. SYMBOLIC NOTATIONS

■ Used in instruction function description

A Accumulator register

$(\bar{A})$ Logical complement of A

A_{0-7} Most signifiant byte of A

A_{8-15} Least signifiant byte of A

C Carry indicator

D Displacement value (rightmost byte in the instruction format extended to the left by a zero value byte). Thus :

```
xxxxxxxxyyyyyyyy   ←——   Instruction
00000000yyyyyyyy   ←——   Displacement
```

E Accumulator extension register

E, A Extended accumulator : juxtaposition of E and A in this order. Most signifiant word in E.

G General base value

G' { In Master mode G' = G
In Slave mode G = 0

L Local base value

MA Interrupt mask indicator

MS Mode indicator (Master/Slave)

N Calculated operand (Contents of Y-address word)

N_{11-15} Contents of bits 11-15 of calculated operand

O	Overflow indicator
P	Program base value
PR	Memory protection indicator
Rn	n-number general register
X	Index register
Y	Calculated memory address (Displacement value processed according to addressing mode function)
Y_2	Word address corresponding to y If Y is even : $Y_2 = Y$ If Y is odd : $Y_2 = Y - 1$
α	Current rank in a byte or word string processed by an instruction; the first byte or word in the string is rank zero.
bp	Protection bit
r	Contents of R-register
rn	Contents of Rn-register
y	Y-address byte
y_2	Y_2 -address word
( )	Contents of ... E.g. $y_2 = (Y_2)$ $N = (Y_2)$
→	Replaces
↔	Exchanged
$\not\#$	Altered but non-signifiant
$\oplus$	Exclusive OR
$\wedge$	Logical AND
$\vee$	Logical OR

■ Used in examples

The purpose of this chapter is to familiarize the user with assembly language writing, thus with every instruction examples are given to show various possibilities which will enable him to program without a complete knowledge of Assembler capabilities.

The following conventions will be used in order to simplify example representation :

Identifiers include four letters followed by a serial number. These letters have the following meaning :

1st letter	E	Label
	S	Symbol (defined through an EQU pseudo-instruction)
2nd letter	P	Predefined
	A	Anticipated
3rd letter	D	Defined in a data segment
	P	Defined in a program segment
4th letter	C	Common
	L	Local

Example :

EPDC27 denotes a predefined label of the common data.

Thus all examples are written relatively to a CDS and a LDS having the following form :

```
EXEMPL          CDS

EPDC1           DATA            7
EPDC2           DATA            EPDC1
EPDC3           DATA            EPDC4
EPDC4           RES             20
EPDC5           DATA,1          &FF,00
EPDC6           DATA            EPDC5
EPDC7           DATA            EPDC8
EPDC8           RES,1           10
EPDC9           DATA            "AB"
                DATA            "CD"
EPDC10          DATA            EPDC9
EPDC11          DATA            6
EPDC12          DATA            &82
EPDC13          DATA            EPPL2
EPDC14          DATA            EPPL3
EPDC15          DATA            EPPL1
                DATA            EPPL2
                DATA            EPPL3
                DATA            EPPL4
EPDC16          TEXT            "ABCDEFGHIJ"
EPDC17          DATA            EPDC16
EPDC18          DATA,1          "A"
                DATA,1          "E"
                DATA,1          "X"
EPDC19          DATA            EPDC18
EPDC20          DATA            EPDC21
                DATA            EPDC21 + 2
                DATA            EPDC21 + 4
EPDC21          DATA            3
                DATA            4
                DATA            5
EPDC22          DATA            EPDC23
                DATA            EPDC23 + 2
                DATA            EPDC23 + 4
                DATA            EPDC23 + 6
                DATA            EPDC23 + 8
EPDC23          RES             10
EPDC24          DATA            EPDC25
                DATA            EPDC25 + 1
                DATA            EPDC25 + 2
                DATA            EPDC25 + 3
EPDC25          RES,1           5
EPDC26          DATA            &30A2
                DATA            &400B
                DATA            &40FA
                DATA            &BF01
EPDC27          DATA            EPD26
                DATA            EPDC26 + 4
EPDC28          RES             5
EPDC29          DATA            EPDC28
SPDC1           EQU             EPDL1
SPDC2           EQU             3
```

```
SPDC3           EQU             SPL2
SPDC4           EQU             EPDL14
                FIN


LDS1            LDS

EPDL1           DATA            7
EPDL2           DATA            EPPL1
EPDL3           DATA            EPDL4
EPDL4           RES             20
EPDL5           DATA,1          &FF,00
EPDL6           DATA            EPDL5
EPDL7           DATA            EPDL6
EPDL8           RES,1           10
EPDL9           DATA            "AB"
                DATA            "CD"
EPDL10          DATA            EPDL9
EPDL11          DATA            6
EPDL12          DATA            &82
EPDL13          DATA            EPPL2
EPDL14          DATA            EPPL3
EPDL15          DATA            EPPL1
                DATA            EPPL2
                DATA            EPPL3
                DATA            EPPL4
EPDL16          TEXT            "ABCDEFGHIJ"
EPDL17          DATA            EPDL16
EPDL18          DATA,1          "A"
                DATA,1          "E"
                DATA,1          "X"
EPDL19          DATA            EPDL18
EPDL20          DATA            EPDL21
                DATA            EPDL21 + 2
                DATA            EPDL21 + 4
                DATA            3
                DATA            4
                DATA            5
EPDL22          DATA            EPDL23
                DATA            EPDL23 + 2
                DATA            EPDL23 + 4
                DATA            EPDL23 + 6
                DATA            EPDL23 + 8
EPDL23          RES             10
EPDL24          DATA            EPDL25
                DATA            EPDL25 + 1
                DATA            EPDL25 + 2
                DATA            EPDL25 + 3
EPDL25          RES,1           5
EPDL26          DATA            &30A2
                DATA            &400B
                DATA            &40FA
                DATA            &BF01
EPPL27          DATA            EPPL26
                DATA            EPPL26 + 4
```

```
EPDL28      RES        5
EPDL29      DATA       EPDL28
SPDL1       EQU        EPDL1
SPDL2       EQU        3
SPDL3       EQU        SPDL2
SPDL4       EQU        EPDL14
            FIN
LPS1        LPS        LDS1
            .
            .
            .    Instructions
            .
            .
            FIN        EPPL1
            END        LPS1
```

VII-3. LOAD AND STORE INSTRUCTIONS

VII-3.1. Introduction

The following table contains the load and store instruction which are individually described hereafter.

	Byte		Word		Double word	
	Load	Store	Load	Store	Load	Store
Data	LBL LBR LBX	SBL SBR	LDA LDE LDX LDR	STA STE STX STR	DLD	DST
Address			LEA	SPA		
Selective				STS		

VII-3.2. Description

These instructions are described in the following order :

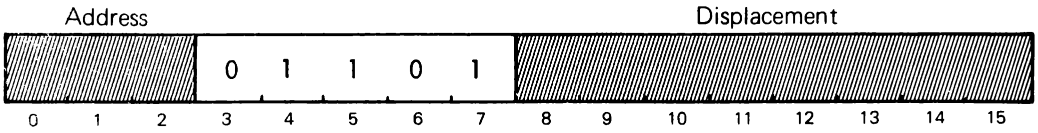
Byte	LBL	Load Byte Left in A
	SBL	Store Byte Left in A
	LBR	Load Byte Right in A
	SBR	Store Byte Right in A
	LBX	Load Byte right in X
Word	LDA	LoaD A
	STA	STore A
	LDE	LoaD E
	STE	STore E
	LDX	LoaD X
	STX	STore X
	LDR	LoaD Register
	STR	STore Register
	LEA	Load Effective Address

Word	SPA	Store Program Address
	STS	STore Selective in A
Double word	DLD	Double LoaD of E, A extended accumulator
	DST	Double STore of E, A extended accumulator

NAME : Load Byte Left in A **LBL**

Class : 0 Non-priviledged Standard

Instruction format :



Addressing mode	Hexadecimal code	Execution time (µs)
DL	0D	2,6
P	2D	3
DG	4D	2,6
IL	6D	3,6
IGX	8D	3,6
ILX	AD	3,6

Function : $y \longrightarrow (A_{0-7})$
(A_{8-15}) unaffected

Y-address operand loaded in leftmost byte of A-register.
Rightmost byte of A-register unaffected.

Modified elements :

- Registers : A_{0-7}
- Memory locations
- Indicators : C-O

Indicators :

C	O	Upon execution
0	0	(A) > 0
0	1	(A) < 0
1	0	(A) = 0

Trap conditions : standard

Examples :

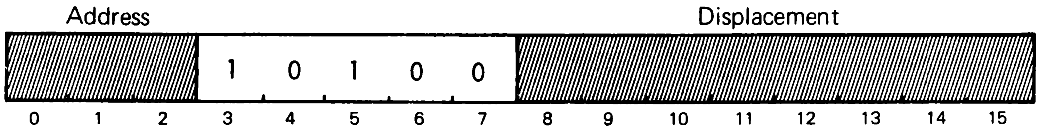
```
LBL          EPDL1 + 1
LBL          = 0
LBL          # EPDC1
LBL          @ EPDL6
LBL          @ # EPDC7,x
LBL          @ EPDL7,x
```

SBL

NAME : Store Byte Left in A

Class : 0' Non-priviledged Standard

Instruction format :



Addressing mode	Hexadecimal code	Execution time (μs)
DL	14	2,5
DG	54	2,5
IL	74	3,5
IGX	94	3,5
ILX	B4	3,5

Function : $(A_{0-7}) \rightarrow y$
(A) unaffected

Leftmost byte of A-register stored at Y-address in core memory.

Modified elements :

- Memory locations : y

Trap conditions : standard

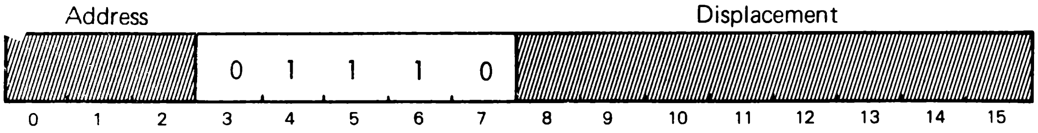
Examples :

```
SBL       EPDL8
SBL       ≠ EPDC8 + 3
SBL       @EPDL24
SBL       @≠ EPDC24,x
SBL       @EPDL24,x
```

NAME : Load Byte Right in A **LBR**

Class : 0 Non-priviledged Standard

Instruction format :



Addressing mode	Hexadecimal code	Execution time (µs)
DL	0E	2,6
P	2E	3
DG	4E	2,6
IL	6E	3,6
IGX	8E	3,6
ILX	AE	3,6

Function : $y \longrightarrow (A_{8-15})$
$0 \longrightarrow (A_{0-7})$

Y-address operand loaded in rightmost byte of A-register.
Leftmost byte of A-register cleared.

Modified elements :

- Register : A
- Memory locations
- Indicators : C-O

Indicators :

C	O	Upon execution
0	0	(A) > 0
1	0	(A) = 0

Trap conditions : standard

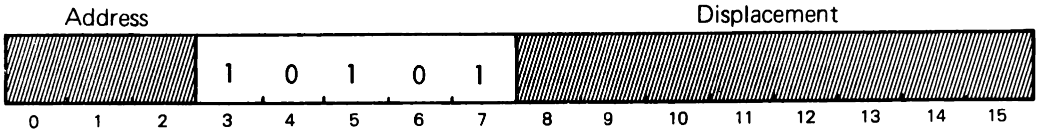
Examples :

LBR	SPDL4
LBR	=SPDL2
LBR	# EPDC1
LBR	@EPDL6
LBR	@# EPDC7,x
LBR	@EPDL7,x

NAME : Store Byte Right in A **SBR**

Class : 0' Non-priviledged Standard

Instruction format :



Addressing mode	Hexadecimal code	Execution time (μs)
DL	15	2,5
DG	55	2,5
IL	75	3,5
IGX	95	3,5
ILX	B5	3,5

Function : $(A_{8-15}) \longrightarrow y$
(A) unaffected

Rightmost byte of A-register stored at Y-address in core memory.

Modified elements :

- Memory locations : y

Trap conditions : standard

Examples :

```
SBR          EPDL8 + 3
SBR          # EPDC8
SBR          @ EPDL7
SBR          @# EPDC24,x
SBR          @# EPDL24,x
```

NAME : Load Byte Right in X **LBX**

Class : 0 Non-priviledged Standard

Instruction format :

Address								Displacement							
▨	▨	▨	0	1	1	1	1	▨	▨	▨	▨	▨	▨	▨	▨
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Addressing mode	Hexadecimal code	Execution time (μs)
DL	0F	2,6
P	2F	3
DG	4F	2,6
IL	6F	3,6
IGX	8F	3,6
ILX	AF	3,6

Function : $y \longrightarrow (X_{8-15})$
$0 \longrightarrow (X_{0-7})$

Y-address operand loaded in rightmost byte of X-register.
Leftmost byte of X-register cleared.

Modified elements :

- Register : X
- Memory locations
- Indicators : C-O

Indicators :

C	O	Upon execution
0	0	(X) > 0
1	0	(X) = 0

Trap conditions : standard

Examples :

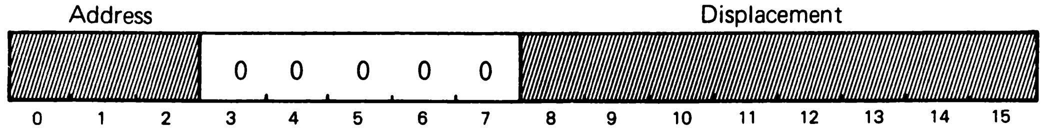
```
LBX          EPDL5
LBX          =1
LBX          # EPDC5
LBX          @EPDL6
LBX          @= EPDC7,x
LBX          @EPDL7,x
```

NAME : LoaD A

LDA

Class : 0 Non-priviledged Standard

Instruction format :



Addressing mode	Hexadecimal code	Execution time (μs)
DL	00	2,3
P	20	2,5
DG	40	2,2
IL	60	3,4
IGX	80	3,4
ILX	A0	3,4

Function : $y_2 \longrightarrow (A)$

A-register loaded with Y_2 -address operand.

Modified elements :

- Register : A
- Memory locations
- Indicators : C-O

Indicators :

C	O	Upon execution
0	0	(A) > 0
0	1	(A) < 0
1	0	(A) = 0

Trap conditions : standard

Examples :

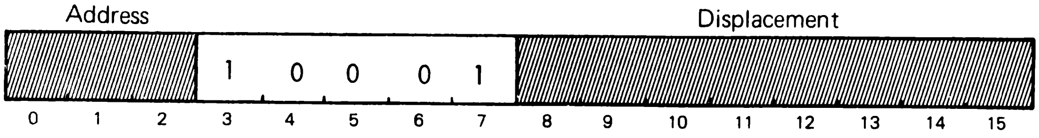
```
LDA            EPDL1
LDA            =7
LDA            # EPDC1
LDA            @EPDL2
LDA            @# EPDC3,x
LDA            @EPDL3,x
```

STA

NAME : STore A

Class : 0' Non-priviledged Standard

Instruction format :



Addressing mode	Hexadecimal code	Execution time (μs)
DL	11	2,2
DG	51	2,2
IL	71	3,2
IGX	91	3,2
ILX	B1	3,2

Function : (A) → y_2
(A) unaffected

A-register contents stored at Y_2 -address in core memory.

Modified elements :

- Memory locations : y_2

Trap conditions : standard

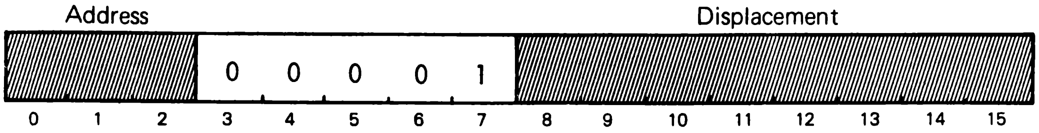
Examples :

```
STA          EPDL23 + 4
STA          ≠ EPDC23
STA          @EPDL3
STA          @≠ EPDC22,x
STA          @EPDL22,x
```

NAME : LoaD E **LDE**

Class : 0 Non-priviledged Standard

Instruction format :



Addressing mode	Hexadecimal code	Execution time (µs)
DL	01	2,3
P	21	2,5
DG	41	2,3
IL	61	3,4
IGX	81	3,4
ILX	A1	3,4

Function : $y_2 \longrightarrow (E)$

Y_2 -address operand loaded in E-register.

Modified elements :

- Registers : E
- Memory locations
- Indicators : C-O

Indicators :

C	O	Upon execution
0	0	$(E) > 0$
0	1	$(E) < 0$
1	0	$(E) = 0$

Trap conditions : standard

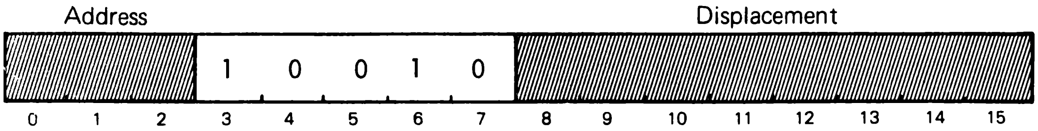
Examples :

```
LDE          SPDL1
LDE          =&FF
LDE          # SPDC1
LDE          @EPDL3
LDE          @# EPDC3,x
LDE          @EPDL3,x
```

NAME : STore E **STE**

Class : 0' Non-priviledged Standard

Instruction format :



Addressing mode	Hexadecimal code	Execution time (μs)
DL	12	2,2
DG	52	2,2
IL	72	3,2
IGX	92	3,2
ILX	B2	3,2

Function : (E) → y_2
(E) unaffected

E-register contents stored at Y_2 -address memory location.

Modified elements :

- Memory locations : y_2

Trap conditions : standard

Examples :

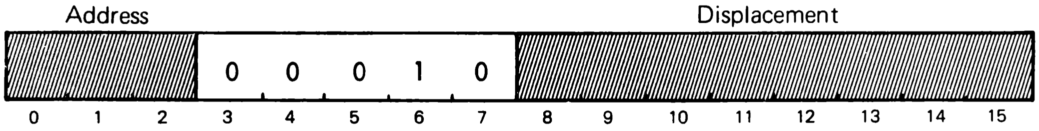
```
STE          EPDL4
STE          #EPDC4 + 2
STE          @EPDL3
STE          @# EPDC22,x
STE          @EPDL22,x
```

NAME : LoaD X

LDX

Class : 0 Non-priviledged Standard

Instruction format :



Addressing mode	Hexadecimal code	Execution time (µs)
DL	02	2,3
P	22	2,5
DG	42	2,3
IL	62	3,4
IGX	82	3,4
ILX	A2	3,4

Function : $y_2 \longrightarrow (X)$

Y_2 -address operand loaded in X-register.

Modified elements :

- Registers : X
- Memory locations :
- Indicators : C-O

Indicators :

C	O	Upon execution
0	0	(X) > 0
0	1	(X) < 0
1	0	(X) = 0

Trap conditions : standard

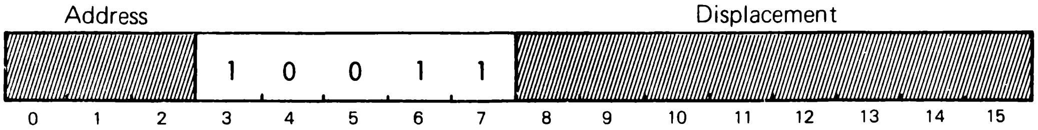
Examples :

```
LDX        EPDL2
LDX        =4
LDX        #EPDC1
LDX        @EPDL3
LDX        @# EPDC3,x
LDX        @EPDL3,x
```

NAME : STore X **STX**

Class : 0' Non-priviledged Standard

Instruction format :



Addressing mode	Hexadecimal code	Execution time (µs)
DL	13	2,2
DG	53	2,2
IL	73	3,2
IGX	93	3,2
ILX	A3	3,2

Function : $(X) \rightarrow y_2$
(X) unaffected

X-register contents stored at Y_2 -address memory location.

Modified elements :

- Memory location : y_2

Trap conditions : standard

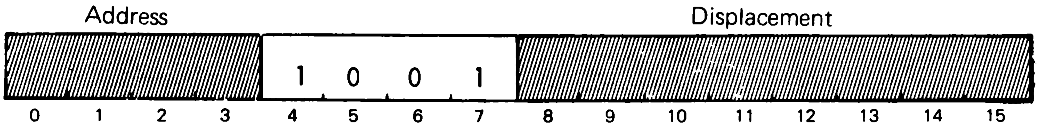
Examples :

STX	EPDL23 + 6
STX	# EPDC23 + 2
STX	@EPDL22
STX	@# EPDC22,x
STX	@EPDL22,x

NAME : LoaD Register **LDR**

Class : 1 Non-priviledged Standard

Instruction format :



Addressing mode	Hexadecimal code	Execution time (µs)
DL	39	3,7
PX	E9	3,7
P	F9	3,7

Function : $(R_n) \longrightarrow A$
(R_n) unaffected
with n = (Y)

A-register loaded with the contents of n-register (n = register no.).

Modified elements :

- Registers : A
- Indicators : C-O

Indicators :

C	O	Upon execution
0	0	$(A) > 0$
0	1	$(A) < 0$
1	0	$(A) = 0$

Trap conditions : standard

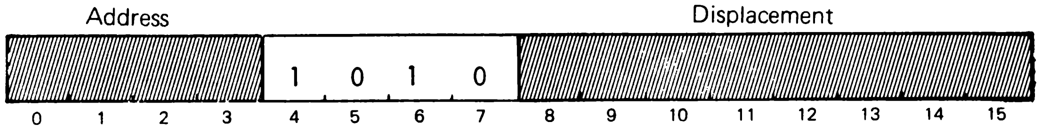
Examples :

```
LDR          EPDL1
LDR          =0,x
LDR          =22
```

NAME : STore Register **STR**

Class : 1 Priviledged Standard

Instruction format :



Addressing mode	Hexadecimal code	Execution time (μs)
DL	3A	4
PX	EA	4
P	FA	4

Function : (A) → (R_n)
(A) unaffected
with n = (Y)

A-register contents stored in n-register (n = register no.).

Modified elements :

- Registers : R

Trap conditions :

- Standard and mode violation

Miscellaneous :

This instruction is executable in master mode only since it alters the contents of a given register (specifically I/O registers). Normally reserved for monitor or operating system use.

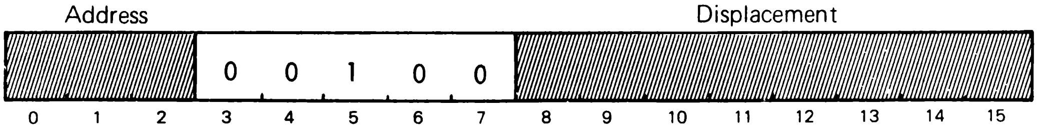
Examples :

```
STR             EPDL1
STR             =2,x
STR             =4
```

NAME : Load Effective Address **LEA**

Class : 0 Non-priviledged Standard

Instruction format :



Addressing mode	Hexadecimal code	Execution time (µs)	
DL	04	2,7	3,3
P	24	3,1	3,7
DG	44	2,7	3,3
IL	64	3,7	4,3
IGX	84	3,7	4,3
ILX	A4	3,7	4,3
		Master mode	Slave mode

Function : Y - (G) → A

G-register contents subtracted from computed address. Result, representing the actual address, stored in A-register.

Modified elements :

- Registers : A

Trap conditions : standard

Examples :

```
LEA          EPDC7
LEA          =&2F
LEA          ≠ EPDL11
LEA          @EPDC2
LEA          @≠ EPDC22,x
LEA          @EPDL22
```

NAME : Store Program Address **SPA**

Class : 0' Non-priviledged Standard

Instruction format :

Address								Displacement							
▨	▨	▨	1	1	0	0	0	▨	▨	▨	▨	▨	▨	▨	▨
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Addressing mode	Hexadecimal code	Execution time (μs)	
DL	18	3,1	3,4
DG	58	3,1	3,4
IL	78	4,2	4,5
IGX	98	4,2	4,5
ILX	B8	4,2	4,5
		Master mode	Slave mode

Function : (P) + G' → y_2

This instruction is normally followed by an indirect branch to a sub-routine. Therefore, the current address incremented by four (P-register contents + 4) is stored in A-register. This new address is generally the sub-routine return address.

Modified elements :

- Registers
- Memory locations : y_2
- Indicators

Trap conditions : standard

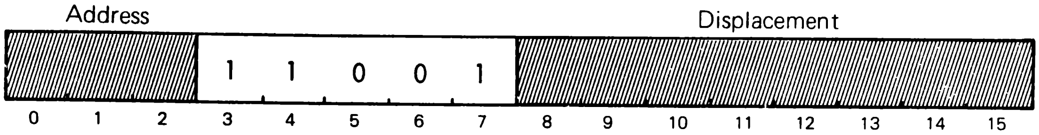
Examples :

```
                                              |          Return
SPA          EPDL28                           |
BRU          @  ------------------------------|-->  BRU      @EPDL28
 .            .                               |      .          .
-------------------------------------------------------------------
 .            .                               |      .          .
SPA          # EPDC28                         |
BRU          @  ------------------------------|-->  BRU      @# EPDC28
 .            .                               |      .          .
-------------------------------------------------------------------
 .            .                               |      .          .
SPA          @EPDL29                          |
BRU          @  ------------------------------|-->  BRU      @ EPDL29
 .            .                               |      .          .
-------------------------------------------------------------------
 .            .                               |      .          .
SPA          @# EPDC29,x                      |
BRU          @  ------------------------------|-->  BRU      @ # EPDC29
 .            .                               |      .          .
-------------------------------------------------------------------
 .            .                               |      .          .
SPA          @EPDL29,x                        |
BRU          @  ------------------------------|-->  BRU      @EPDL29
```

NAME : STore Selective in A **STS**

Class : 0' Non-priviledged Standard

Instruction format :



Addressing mode	Hexadecimal code	Execution time (μs)
DL	19	3,4
DG	59	3,4
IL	79	4,4
IGX	99	4,4
ILX	B9	4,4

Functions : $y_2 \wedge (\bar{E}) \vee (A) \wedge (E) \longrightarrow y_2$
(A) and (E) unaffected

The bits of A-register which correspond to set bits in E-register are stored into corresponding position of y_2 memory address. Remaining bits of the memory word are not modified. Thus, for instance :

0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	y_2	Before execution
1	1	0	0	0	1	0	0	1	1	1	1	0	0	1	1	E	
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	Upon execution

Modified elements :

- Registers
- Memory locations : y_2
- Indicators : C-O

Indicators :

C	O	Upon execution
0	0	Result $>$ 0
0	1	Result $<$ 0
1	0	Result = 0

Trap conditions : standard

Examples :

STS	EPDL4
STS	# EPDC23
STS	@EPDL3
STS	@# EPDC22,x
STS	@EPDL22,x

NAME : Double LoaD of A and E **DLD**

Class : 0' Non-priviledged Standard

Instruction format :

Address								Displacement							
▨	▨	▨	1	0	0	0	0	▨	▨	▨	▨	▨	▨	▨	▨
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Addressing mode	Hexadecimal code	Execution time (μs)
DL	10	3,6
DG	50	3,6
IL	70	4,8
IGX	90	4,8
ILX	B0	4,8

Function : $(Y_2) \longrightarrow (E)$
$(Y_2 + 2) \longrightarrow (A)$

E-register loaded with the contents of Y_2 memory address and A-register loaded with the contents of $Y_2 + 2$ memory address.

Modified elements :

- Registers : E-A
- Memory locations
- Indicators : C-O

Indicators :

C	O	Upon execution
0	0	(E) > 0
0	1	(E) < 0
1	0	(E) = 0

Trap conditions : standard

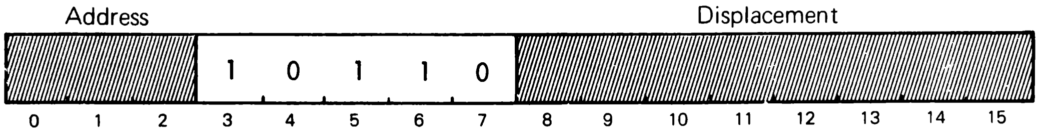
Examples :

```
DLD          EPDL9
DLD          # EPDC9
DLD          @EPDL10
DLD          @# EPDC3,x
DLD          @EPDL3,x
```

NAME : Double STore of A and E **DST**

Class : 0' Non-priviledged Standard

Instruction format :



Addressing mode	Hexadecimal code	Execution time (μs)
DL	16	3,6
DG	56	3,6
IL	76	4,7
IGX	96	4,7
ILX	B6	4,7

Function : (E) ⟶ (Y_2)
(A) ⟶ $(Y_2 + 2)$
(A) and (E) unaffected

E-register contents stored at Y_2 -memory address and A-register contents stored at Y_2 + 2 memory address.

Modified elements :

- Registers
- Memory locations : (Y_2) and $(Y_2 + 2)$
- Indicators

Trap conditions : standard

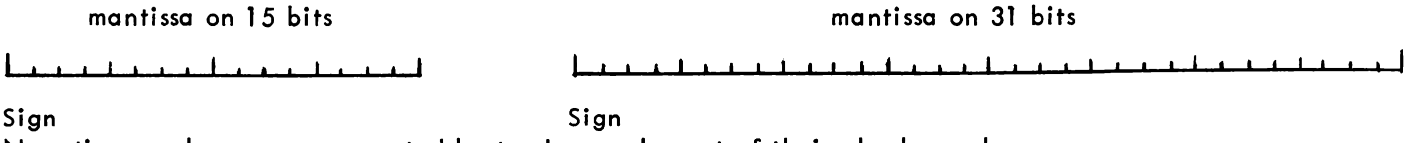
Examples :

DST	EPDL4
DST	# EPDC4
DST	@ EPDL3
DST	@ # EPDC22,x
DST	@ EPDL22,x

VII-4. FIXED-POINT ARITHMETIC

VII-4.1. Introduction

Operands are signed numbers on one or two words.



Negative numbers are represented by two's complement of their absolue value.

VII-4.2. Description

Fixed-point arithmetic instructions perform the four operations and are described in the following order :

ADD	ADDition in A	MUL	MULtiply integer
ADM	ADd to Memory	DIV	DIVide integer
SUB	SUBtract		

NAME : ADDition in A **ADD**

Class : 0 Non-priviledged **Standard**

Instruction format :

Address								Displacement							
▨	▨	▨	0	0	1	0	1	▨	▨	▨	▨	▨	▨	▨	▨
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Addressing mode	Hexadecimal code	Execution time (µs)
DL	05	2,3
P	25	2,5
DG	45	2,3
IL	65	3,4
IGX	85	3,4
ILX	A5	3,4

Function : $(A) + y_2 \longrightarrow (A)$

Y_2 -address operand added with A-register contents; result in A.

Modified elements :

- Registers : A
- Memory locations
- Indicators : C-O

Indicators :

C	O	Upon execution
1	-	Carry
-	1	Overflow

Trap conditions : standard

Examples :

```
ADD          EPDL12
ADD          =122
ADD          # EPDC12
ADD          @EPDL2
ADD          @# EPDC20,x
ADD          @EPDL20,x
```

NAME : ADd to Memory **ADM**

Class : 0' Non-priviledged Standard

Instruction format :

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Address								Displacement							
▨	▨	▨	1	0	1	1	1	▨	▨	▨	▨	▨	▨	▨	▨

Addressing mode	Hexadecimal code	Execution time (μs)
DL	17	2,6
DG	57	2,6
IL	77	3,7
IGX	97	3,7
ILX	B7	3,7

Function : y_2 + (A) → Y_2 and A

A-register contents added to Y_2 -address contents; result in A and at Y_2 -address.

Modified elements :

- Registers : A
- Memory locations : y_2
- Indicators : C-O

Indicators :

C	O	Upon execution
-	1	Carry
1	-	Overflow

Trap conditions : standard

Examples :

ADM	EPDL4
ADM	# EPDC4
ADM	@EPDL3
ADM	@# EPDC20,x
ADM	@EPDL20,x

NAME : SUBtract in A **SUB**

Class : 0 Non-priviledged Standard

Instruction format :

Address								Displacement							
▨	▨	▨	0	0	1	1	0	▨	▨	▨	▨	▨	▨	▨	▨
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Addressing mode	Hexadecimal code	Execution time (μs)
DL	06	2,3
P	26	2,5
DG	46	2,3
IL	66	3,4
IGX	86	3,4
ILX	A6	3,4

Function : (A) - y_2 ⟶ (A)

Y_2 -address operand subtracted from A-register contents; result in A.

Modified elements :

- Registers : A
- Memory locations
- Indicators : C-O

Indicators :

C	O	Upon execution
1	-	Carry
-	1	Overflow

Trap conditions : standard

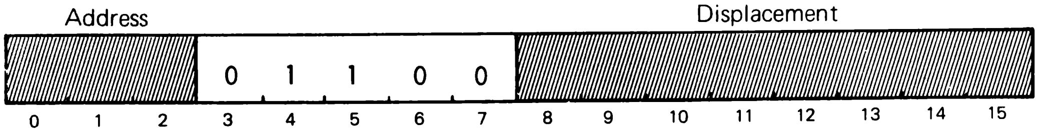
Examples :

```
SUB          EPDL11
SUB          =2
SUB          # EPDC11
SUB          @EPDL20
SUB          @# EPDC20,x
SUB          @EPDL20,x
```

NAME : MULtiply integer **MUL**

Class : 0 Non-priviledged Standard

Instruction format :



Addressing mode	Hexadecimal code	Execution time (µs)
DL	0C	8,1 38,4+0,9 n
P	2C	8,5 38,8+0,9 n
DG	4C	8,1 38,8+0,9 n
IL	6C	9,1 39,4+0,9 n
IGX	8C	9,1 39,4+0,9 n
ILX	AC	9,1 39,4+0,9 n

Wired Micro-programmed

n = no. of set bits in multiplier

Function : $(A) \times y_2 \longrightarrow (E,A)$

Y_2 -address operand multiplied algebraically by A-register contents; result in E and A.

The most significant portion of the result is stored in E, the least significant in A.

Modified elements :

- Registers : E-A
- Memory locations
- Indicators : C-O

Indicators :

C	O	Upon execution
0	0	E > 0
0	1	E < 0
1	0	E = 0

Trap conditions : standard

Examples :

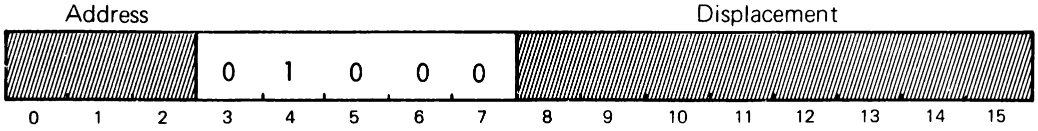
MUL	EPDL11
MUL	=&2E
MUL	# EPDC1
MUL	@EPDL20
MUL	@# EPDC20,x
MUL	@EPDL20,x

DIV

NAME : DIVide integer

Class : 0 Non-priviledged Optional

Instruction format :



Addressing mode	Hexadecimal code	Execution time (μs)
DL	08	9,1 42 +0,3 n
P	28	9,5 42,4+0,3 n
DG	48	9,1 42 +0,3 n
IL	68	10,1 43 +0,3 n
IGX	88	10,1 43 +0,3 n
ILX	A8	10,1 43 +0,3 n

Wired Micro-programmed

n = no. of set bits in the quotient.

Function : $(E,A) : y_2 \rightarrow (A)$
remainder → E
Remainder sign same as dividend sign (except when zero, in which case the sign is +).

Extended accumulator contents (least significant portion of the dividend in A, most significant portion in E) divided by Y_2 -address operand (divisor). Quotient in A, remainder in E.

The remainder is of the same sign as E-register initial contents, except when there is no remainder, in which case the sign is +.

Modified elements :

- Registers : E-A (unaffected if divisor = 0 or if an overflow occurs).
- Memory locations
- Indicators : C-O

Indicators :

C	O	Upon execution
#	1	Overflow or zero divisor

Trap conditions : standard and non-implemented instruction

Examples :

```
DIV            EPDL1
DIV            =5
DIV            # EPDC11
DIV            @EPDL2
DIV            @# EPDC20,x
DIV            @EPDL20,x
```

VII-5. LOGICAL OPERATIONS

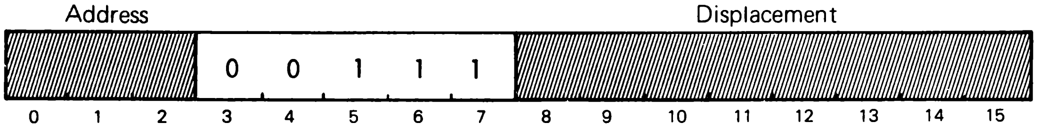
IOR	Inclusive OR
EOR	Exclusive OR
AND	AND
CMP	CoMPare

NAME : Inclusive OR

IOR

Class : 0 Non-priviledged Standard

Instruction format :



Addressing mode	Hexadecimal code	Execution time (µs)
DL	07	2,3
P	27	2,5
DG	47	2,3
IL	67	3,4
IGX	87	3,4
ILX	A7	3,4

Function : $(A) \vee Y_2 \longrightarrow (A)$

Inclusive ORing between Y_2 -address operand and A-register contents; result in A.

Modified elements :

- Registers : A
- Memory locations
- Indicators : C-O

Indicators :

C	O	Upon execution
0	0	(A) > 0
0	1	(A) < 0
1	0	(A) = 0

Trap conditions : standard

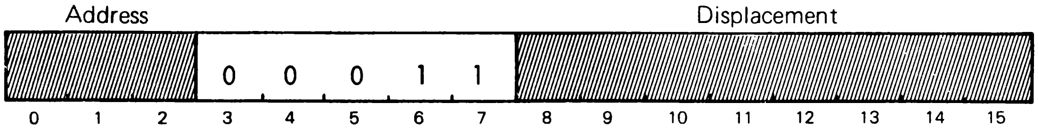
Examples :

```
IOR           EPDL21
IRO           =&01
IOR           #EPDL9
IOR            @EPDL20
IOR            @# EPDC20,x
IOR            @EPDL20,x
```

NAME : Exclusive OR **EOR**

Class : 0 Non-priviledged Standard

Instruction format :



Addressing mode	Hexadecimal code	Execution time (µs)
DL	03	2,3
P	23	2,5
DG	43	2,3
IL	63	3,4
IGX	83	3,4
ILX	A3	3,4

Function : $(A) \oplus y_2 \longrightarrow (A)$

Exclusive ORing between Y_2 -address operand and A-register contents; result in A.

Modified elements :

- Registers : A
- Memory locations
- Indicators : C-O

Indicators :

C	O	Upon execution
0	0	(A) > 0
0	1	(A) < 0
1	0	(A) = 0

Trap conditions : standard

Examples :

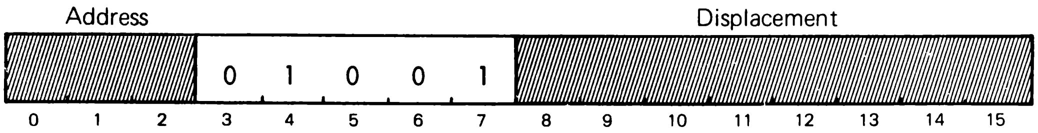
```
EOR          EPDL1
EOR          =&80
EOR          # EPDC1
EOR          @EPDL2
EOR          @# EPDC20,x
EOR          @EPDL20
```

AND

NAME : Logical AND

Class : 0 Non-priviledged Standard

Instruction format :



Addressing mode	Hexadecimal code	Execution time (µs)
DL	09	2,3
P	29	2,5
DG	49	2,3
IL	69	3,4
IGX	89	3,4
ILX	A9	3,4

Function : (A) $\wedge$ y_2 → (A)

Logical ANDing (intersection) between Y_2 -address operand and A-register contents; result in A.

Modified elements :

- Registers : A
- Memory locations
- Indicators : C-O

Indicators :

C	O	Upon execution
0	0	(A) > 0
0	1	(A) < 0
1	0	(A) = 0

Trap conditions : standard

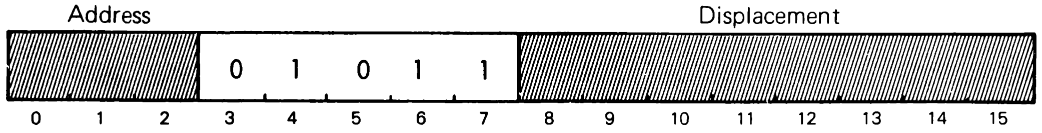
Examples :

AND	EPDL1
AND	=&0F
AND	# EPDC1
AND	@EPDL20
AND	@# EPDC20,x
AND	@EPDL20,x

NAME : CoMpare **CMP**

Class : 0 Non-priviledged Standard

Instruction format :



Addressing mode	Hexadecimal code	Execution time (us)		
		>	<	=
DL	0B	4,1	4,5	3,2
P	2B	4,5	5	3,6
DG	4B	4,1	4,5	3,2
IL	6B	5,1	5,6	4,2
IGX	8B	5,1	5,6	4,2
ILX	AB	5,1	5,6	4,2

Function : Algebraic comparaison between (A) and y_2
Result in indicators

A-register contents compared to Y_2 -address operand; result of comparaison in Carry and Overflow indicators.

A-register contents is treated as the first term of the comparaison, Y_2 -address operand as the second.

Modified elements :

- Indicators : C-O

Indicators :

C	O	Upon execution
0	0	(A) > y_2
0	1	(A) < y_2
1	0	(A) = y_2

Trap conditions : standard

Examples :

```
CMP          EPDL1
CMP          =3
CMP          # EPDC1
CMP          @ EPDL2
CMP          @ # EPDC20,x
CMP          @ EPDL20,x
```

VII-6. REGISTER INCREMENTATION AND DECREMENTATION

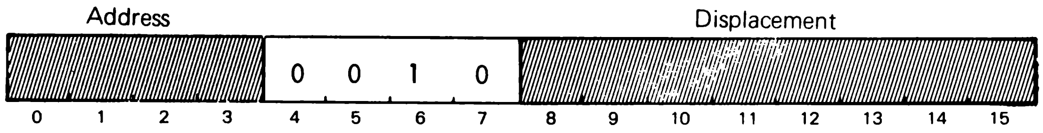
ICX	InCrement X
DCX	DeCrement X
ICL	InCrement L
DCL	DeCrement L

ICX

NAME : InCrement X

Class : 1 Non-priviledged Standard

Instruction format :



Addressing mode	Hexadecimal code	Execution time (μs)
DL	32	2,2
PX	E2	2,2
P	F2	2,2

Function : $(X) + y_2 \longrightarrow (X)$

X-register contents incremented by Y_2 -address operand; result in X-register.

Modified elements :

- Registers : X
- Indicators : C-O

Indicators :

C	O	Upon execution
1	-	Carry
-	1	Overflow

Trap conditions : standard

Examples :

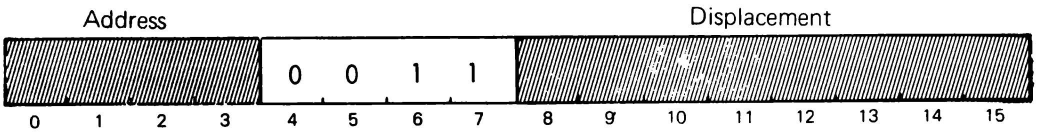
```
ICX            EPDL11
ICX            =&1A,x
ICX            =22
ICX            =0,X          X multiplied by 2.
```

NAME : DeCrement X

DCX

Class : 1 Non-priviledged Standard

Instruction format :



Addressing mode	Hexadecimal code	Execution time (μs)
DL	33	2,2
PX	E3	2,2
P	F3	2,2

Function : (X) - y_2 → (X)

X-register contents decremented by Y_2 -address contents; result in X-register.

Modified elements :

- Registers : X
- Indicators : C-O

Indicators :

C	O	Upon execution
1	-	Carry
-	1	Overflow

Trap conditions : standard

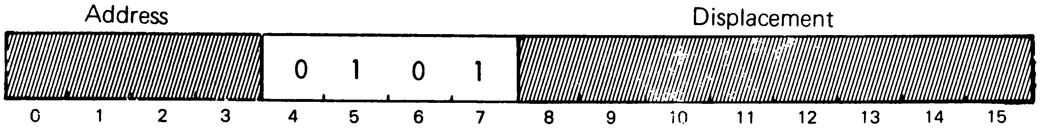
Examples :

```
DCX          EPDL1
DCX          =3,x
DCX          =9
```

NAME : InCrement L **ICL**

Class : 1 Non-priviledged Standard

Instruction format :



Addressing mode	Hexadecimal code	Execution time (µs)
DL	35	2,2
PX	E5	2,2
P	F5	2,2

Function : $(L) + y_2 \longrightarrow (L)$

L-register contents incremented by Y_2 -address operand; result in L-register.

Modified elements :

- Registers : L

Trap conditions : standard

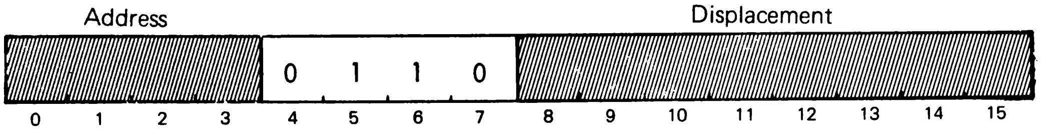
Examples :

```
ICL            EPDL5
ICL            =50,x
ICL            =255
```

NAME : DeCrement L **DCL**

Class : 1 Non-priviledged Standard

Instruction format :



Addressing mode	Hexadecimal code	Execution time (µs)
DL	36	2,2
PX	E6	2,2
P	F6	2,2

Function : $(L) - y_2 \longrightarrow (L)$

L-register contents decremented by Y_2 -address contents; result in L-register.

Modified elements :

- Registers : L

Trap conditions : standard

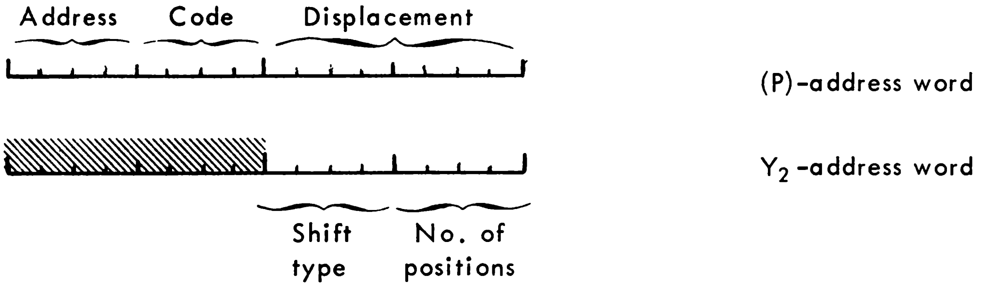
Examples :

```
DCL          EPDL5
DCL          =5,x
DCL          =&1F
```

VII-7. SHIFT OPERATIONS

VII-7.1. Introduction

There are two basic shift instructions : SHR (SHift Register) and SHC (SHift speCial). Both use a word located at Y_2 memory address to specify the shift type and the number of shift steps.



In most cases, the addressing mode is parameter (immediate) and these two words are obviously identical.

For the sake of clarity, each instruction derived from SHR and SHC will be described separately.

Their mnemonics are recognized by MITRAS II.

Remark 1 :

A particular case of the SHC family, DITR, is not a shift instruction. Though mentionad with SHC derivates, it will be fully described with system branch instructions.

Remark 2 :

"Direct, Local" addressing is only applicable to SHR and SHC themselves, since it is meaningless for their derivates (SLLS..., SLLD...).

VII-7.2. Description

	SHR	SHift Register
SHR derivates	SLLS	Shift Left Logical Single (A)
	SRCS	Shift Right Logical Single (A)
	SAD	Shift Arithmetic Double (E, A)
	SLCD	Shift Left Circular Double (E, A)
	SLCS	Shift Left Circular Single (A)
	SAS	Shift Arithmetic Single (A)
	SRLS	Shift Right Logical Single (A)
	SRCD	Shift Right Circular Double (E, A)
SHC derivates	SHC	SHift speCial
	SLLD	Shift Left Logical Double (E, A)
	SRLD	Shift Right Logical Double (E, A)
	PTY	compute PariTY (A)
	NLZ	NormaLiZe length (E, A)

The following notation will be used in the description of the hexadecimal code of SHR and SHC families.

xx

hexadecimal code of SHR
or SHC instruction

y - n

no. of shift steps

additional code in hexadecimal

Example :

SRLS = 5

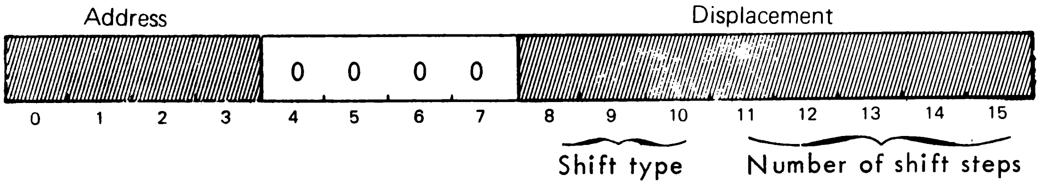
Address	0	0	0	0	1	1	0	0	0	1	0	1
0 3	4			7	8		10	11				15

E0 C-n n=5 ⟹ E0 C5

NAME : SHift Register SHR

Class : 1 Non-priviledged Standard

Instruction format :



Addressing mode	Hexadecimal code
DL	30
PX	E0
P	F0

Execution time depending on operation type.

Function : r shifted ⟶ r (r denoting A or E, A)

- arithmetic shift : r_0 repeated at each shift step
- circular shift : r_0 considered as following r

N_{11-15} number of shift steps ($0 \leq N_{11-15} \leq 31$)

N_{8-10} shift type

- 0	Left, Logical, single (A)	SLLS
- 1	Right, circular, single (A)	SRCS
- 2	Arithmetic, double (E, A)	SAD
- 3	Left, circular, double (E, A)	SLCD
- 4	Left, circular, single (A)	SLCS
- 5	Arithmetic, single (A)	SAS
- 6	Right, logical, single (A)	SRLS
- 7	Right, circular, double (E, A)	SRCD

Modified elements :

- Registers : A (and E for double length shifts)
- Memory locations
- Indicators : C-O

Indicators :

C	O	Upon execution
0/1	≠	Last bit shifted out of r
#	#	If step no. = zero

Trap conditions : standard

Examples :

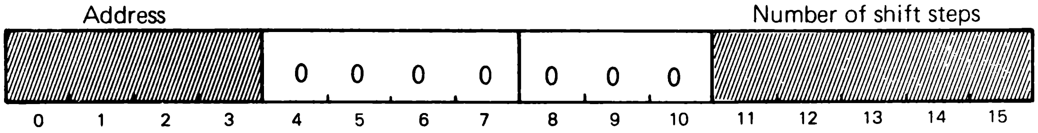
SHR	EPDL5 (equivalent to SRCD = 31)
SHR	=11,x (if (x) = 0, equivalent to SLLS = 11)
SHR	=&41 (equivalent to SAD = 1)

NAME : Shift Left, Logical, Single

SLLS

Class : 1 Non-priviledged Standard

Instruction format :



Addressing mode	Hexadecimal code		Execution time (μs)
PX	E0	0-n	4,3 + 1,2 n
P	F0	0-n	4,3 + 1,2 n

n = number of shift steps

Function : (A) shifted → (A)
$n = ((P))_{11-15} = N_{11-15}$

Contents of A-register shifted n positions to the left. Lower order bits replaced by 0's.

Modified elements :

- Registers : A
- Indicators : C-O

Indicators :

C	O	Upon execution
0/1	≠	Last bit shifted out of A
#	#	If initial shift = 0

Trap conditions : standard

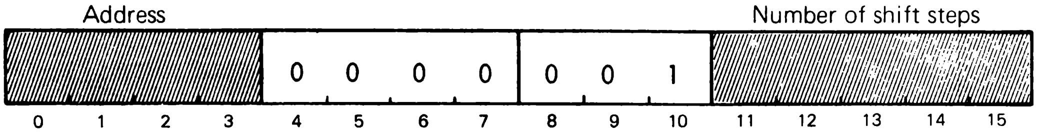
Examples :

```
SLLS        =SPDL3,x
SLLS        =2
SLLS        =2       } equivalent to SLLS = 6
SLLS        =4,x
```

NAME : Shift Right, Circular, Single **SRCS**

Class : 1 Non-priviledged Standard

Instruction format :



Addressing mode	Hexadecimal code	Execution time (μs)
PX	E0 2-n	4,3 + 1,5 n
P	E0 2-n	4,3 + 1,5 n

n = number of shift steps

Function : (A) shifted arith. → (A)
$n = ((P))_{11-15} = N_{11-15}$

Contents of A-register shifted circularly n positions to the right, bit 0 following bit 15.

Modified elements :

- Registers : A
- Indicators : C-O

Indicartors :

C	O	Upon execution
0/1	≠	Last bit shifted out of A
#	#	If initial shift = 0

Trap conditions : standard

Examples :

SRCS	=5,x
SRCS	=SPDL3

NAME : Shift Arithmetic, Double **SAD**

Class : 1 Non-priviledged Standard

Instruction format :

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Address				0	0	0	0	0	1	0	Number of shift steps				

Addressing mode	Hexadecimal code	Execution time (μs)
PX	E0 4-n	4,3 + 2,1 n
P	F0 4-n	4,3 + 2,1 n

n = number of shift steps

Function : (E, A) shifted ⟶ (E,A)
$n = ((P))_{11-15} = N_{11-15}$

Contents of E,A extended register shifted arithmetically n positions to the right. Bit 0 (sign bit) restored after each shift step.

Modified elements :

- Registers : E-A
- Indicators : C-O

Indicators :

C	O	Upon execution
0/1	≠	Last bit shifted out of A
#	#	If initial shift = 0

Trap conditions : standard

Examples :

```
SAD            =SPDL2,x
SAD            =&03
```

NAME : Shift Left, Circular, Double **SLCD**

Class : 1 Non-priviledged Standard

Instruction format :

Address Number of shift steps

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
▨	▨	▨	▨	0	0	0	0	0	1	1	▨	▨	▨	▨	▨

Addressing mode	Hexadecimal code		Execution time (μs)
PX	E0	6-n	4,3 + 2,1 n
P	F0	6-n	4,3 + 2,1 n

n = number of shift steps

Function : (E, A) shifted → (E, A)
$n = ((P))_{11-15} = N_{11-15}$

Contents of E, A extended register shifted circularly n positions to the left, bit 31 following bit 0.

Modified elements :

- Registers : E-A
- Indicators : C-O

Indicators :

C	O	Upon execution
0/1	≠	Last bit shifted out of E
#	#	If initial shift = 0

Trap conditions : standard

Examples :

```
SLCD            =15,x
SLCD            =1
```

NAME : Shift Left, Circular, Single **SLCS**

Class : 1 Non-priviledged Standard

Instruction format :

Address											Number of shift steps				
▨	▨	▨	▨	0	0	0	0	1	0	0	▨	▨	▨	▨	▨
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Addressing mode	Hexadecimal code	Execution time (μs)
PX	E0 8-n	4,3 + 1,5 n
P	F0 8-n	4,3 + 1,5 n

n = number of shift steps

Function : (A) shifted → (A)
$n = ((P))_{11-15} = N_{11-15}$

Contents of A-register shifted circularly n positions to the left, bit 15 following bit 0.

Modified elements :

- Registers : A
- Indicators : C-O

Indicators :

C	O	Upon execution
0/1	≠	Last bit shifted out of A
#	#	If initial shift = 0

Trap conditions : standard

Examples :

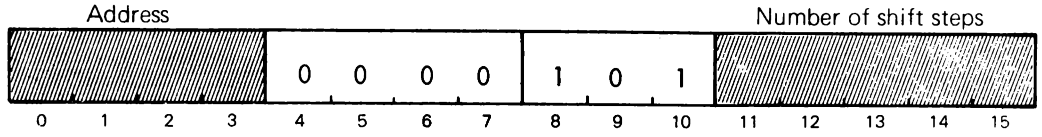
SLCS	=2,x
SLCS	=7

NAME : Shift right, Arithmetic, Single

SAS

Class : 1 Non-priviledged Standard

Instruction format :



Addressing mode	Hexadecimal code	Execution time (µs)
PX	E0 A-n	4,3 + 1,5 n
P	F0 A-n	4,3 + 1,5 n

n = number of shift steps

Function : (A) shifted → (A)
$n = ((P))_{11-15} = N_{11-15}$

Contents of A-register shifted arithmetically n positions to the right. Bit 0 (sign bit) is restored after each shift step.

Modified elements :

- Registers : A
- Indicators : C-O

Indicators :

C	O	Upon execution
0/1	≠	Last bit shifted out of A
#	#	If initial shift = 0

Trap conditions : standard

Examples :

SAS	=&2,x
SAS	=SPDL2

NAME : Shift Right, Logical, Single **SRLS**

Class : 1 Non-priviledged Standard

Instruction format :

Address											Number of shift steps				
▨	▨	▨	▨	0	0	0	0	1	1	0	▨	▨	▨	▨	▨
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Addressing mode	Hexadecimal code		Execution time (µs)
PX	E0	C-n	4,3 + 1,5 n
P	F0	C-n	4,3 + 1,5 n

n = number fo shift steps

Function : (A) shifted ⟶ (A)
$n = ((P))_{11-15} = N_{11-15}$

Contents of A-register shifted n positions to the right. Upper order bits replaced by 0's.

Modified elements :

- Registers : A
- Indicators : C-O

Indicators :

C	O	Upon execution
0/1	≠	Last bit shifted out of A
#	#	If initial shift = 0

Trap conditions : standard

Examples :

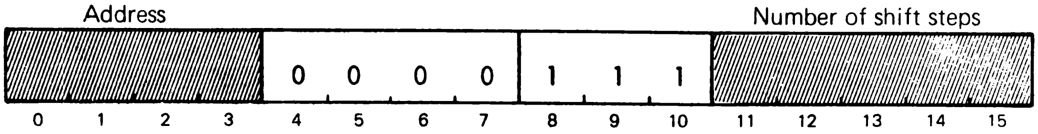
```
SRLS          =0,x
SRLS          =8
```

NAME : Shift Right, Circular, Double

SRCD

Class : 1 Non-priviledged Standard

Instruction format :



Addressing mode	Hexadecimal code	Execution time (µs)
PX	E0 E-n	4,3 + 2,7 n
P	F0 E-n	4,3 + 2,7 n

n = number of shift steps

Function : (E, A) shifted → (E, A)
$n = ((P))_{11-15} = N_{11-15}$

Contents of E, A extended register shifted circularly n positions to the right, bit 0 following bit 31.

Modified elements :

- Registers : E-A
- Indicators : C-O

Indicators :

C	O	Upon execution
0/1	≠	Last bit shifted out of A
#	#	If initial shift = 0

Trap conditions : standard

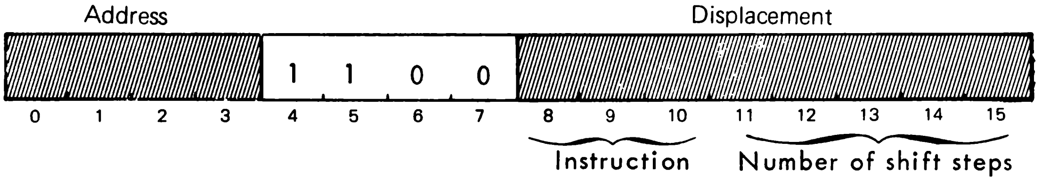
Examples :

```
SRCD          =18,x
SRCD          =&12
```

NAME : SHift speCial **SHC**

Class : 1 Non-priviledged (Except DITR when N_{10} =1) Optional

Instruction format :



If N_{10} =1, N_{11-15} is meaningless

Addressing mode	Hexadecimal code
DL	3C
PX	EC
P	FC

Execution times vary according to shift types.

Function : r shifted → r (r is A or E, A)
N_{11-15} number of shift step
$0 \leqslant N_{11-15} \leqslant 32$
N_{8-10} instruction type

- 0 Shift left logical in E, A (SLLD)
N_{11-15} = number of shift steps

- 4 Shift right logical in E, A (SRLD)
N_{11-15} = number of shift steps

- 2 Compute parity in A (PTY)
N_{11-15} = number of shift steps
Upon execution, E contains the number of set bits shifted out of (A)

- 6 Double lengh normalize in E, A
N_{11-15} = maximum number of shift steps

- 1, 3, 5, 7 } High-speed interrupt de-activation (DITR). Only bit N_{10} is taken into consideration when set.

Modified elements :

- Registers : A (and E for double shifts)
- Indicators : C - O

Indicators :

C	O	Upon execution
0/1	≠	Last bit shifted out
≠	≠	If step no. = zero

Remark :

This page only describes the general usage of SHC. For further details see the individual description of each particular case.

Trap conditions : Standard and non-implemented instruction
(Mode violation for DITR)

Miscellaneous : DITR will be described separately with the control instruction.

Examples :

SHC	EPDL12 (equivalent to SRLD = 2)
SHC	=5,x (if x = 5, equivalent to SLLS = 10)
SHC	&48 (equivalent to PTY = 8)

SLLD

NAME : Shift Left, Logical, Double

Class : 1 Non-priviledged Optional

Instruction format :

Address											Number of shift steps				
▨	▨	▨	▨	1	1	0	0	0	0	0	▨	▨	▨	▨	▨
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Addressing mode	Hexadecimal code	Execution time (μs)
PX	EC 0-n	4,9 + 1,2 n
P	FC 0-n	4,9 + 1,2 n

n = number of shift steps

Function : (E, A) shifted → (E, A)
$n = ((P))_{11-15} = N_{11-15}$

Contents of E, A extended register shifted n positions to the left. Lower order bits replaced by 0's.

Modified elements :

- Registers : E-A
- Indicators : C-O

Indicators :

C	O	Upon execution
0/1	≠	Last bit shifted out of E
#	#	If step no. = zero

Trap conditions : Standard and non-implemented instruction

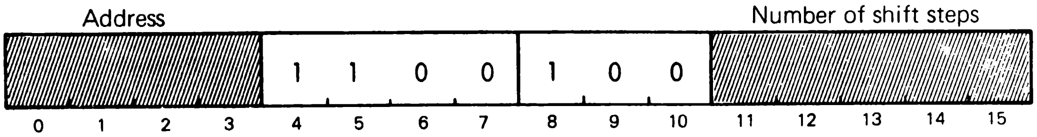
Examples :

```
SLLD        =5,x
SLLD        =&0E
```

NAME : Shift Right, Logical, Double **SRLD**

Class : 1 Non-priviledged Optional

Instruction format :



Addressing mode	Hexadecimal code	Execution time (μs)
PX	EC 8-n	4,9 + 1,8 n
P	FC 8-n	4,9 + 1,8 n

n = number of shift steps

Function : (E, A) shifted ⟶ (E, A)
$n = ((P))_{11-15}$

Contents of E, A extended register shifted n positions to the right. Upper order bits replaced by 0's.

Modified elements :

- Registers : E-A
- Indicators : C-O

Indicators :

C	O	Upon execution
0/1	#	Last bit shifted out of A
#	#	If step no. = zero

Trap conditions : Standard and non-implemented instruction

Examples :

```
SRLD          =3,x
SRLD          =31
```

NAME : PariTY check in A **PTY**

Class : 1 Non-priviledged Optional

Instruction format :

Address											Number of shift steps				
▨	▨	▨	▨	1	1	0	0	0	1	0	▨	▨	▨	▨	▨
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Addressing mode	Hexadecimal code	Execution time (µs)
PX	EC 4-n	5,8 + 1,2 n + 0,3 m
P	FC 4-n	5,8 + 1,2 n + 0,3 m

n = number of shift steps
m = number of set bits detected

Function : (A) shifted ⟶ (A)
(E) = number of set bits shifted out of (A)
$n = ((P))_{11-15} = N_{11-15}$

A-register contents shifted circularly in positions to the left. When the instruction is over, E-register contains the number of set bits which have been shifted out of A.

Modified elements :

- Registers : A
- Indicators : C-O

Indicators :

C	O	Upon execution
0/1	-	Last bit shifted out of A
-	0/1	$\overline{(E_{15})}$ = parity bit
≠	≠	If step no. = zero

Trap conditions : Standard and non-implemented instruction

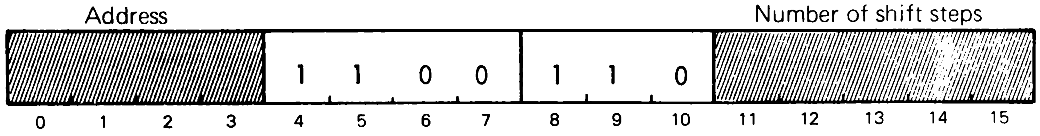
Examples :

```
PTY        =2,x
PTY        =&F
```

NAME : double length NormaLiZe (option) **NLZ**

Class : 1 Non-priviledged Optional

Instruction format :



Addressing mode	Hexadecimal code	Execution time (μs)
PX	EC C-n	4,9 + 1,8 n
P	FC C-n	4,9 + 1,8 n

n = number of shift steps

Function : (E, A) shifted ⟶ (E, A)
X is decremented by the actual number of shift steps
$n = ((P))_{11-15} = N_{11-15}$

The contents of E, A extended register is shifted to the left until bit 0 is different from bi 1 or up to a maximum of n positions. X-register is decremented by the actual number of shift steps.

Modified elements :

- Registers : E - A - X
- Indicators : C-O

Indicators :

C	O	Upon execution
0	0	Normalization 01 ....
0	1	Stop on zero count
1	0	Normalization 10 ....

Trap conditions : Standard and non-implemented instruction

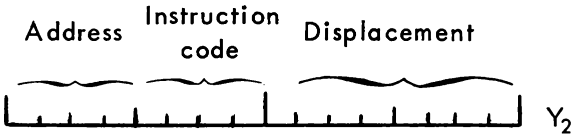
Examples :

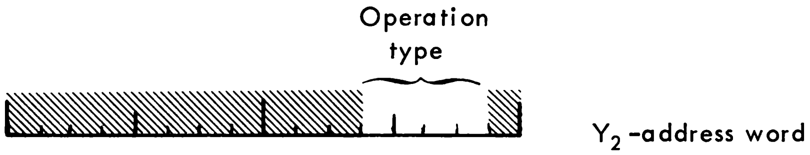
```
NLZ          =10,x
NLZ          =20
```

VII-8. INTER-REGISTER OPERATIONS

VII-8.1. Introduction

SRG (SeT Register) is the basic instruction for inter-register operations. A word located at Y memory address is used to specify the operation type.





In parameter addressing mode which is normally used, these two words are obviously identical.

For the sake of clarity, each instruction derived from SRG will be described separately.

Their mnemonics are recognized by MITRAS I and II.

Remark 1 :

Two particular cases of SRG, RTS and RSV, do not belong to the inter-register operation class. Though mentioned with SRG, they will be fully described with system branch instructions.

Remark 2 :

Any addressing mode other than "parameter" is meaningless for SRG-family instructions.

VII-8.2. Description

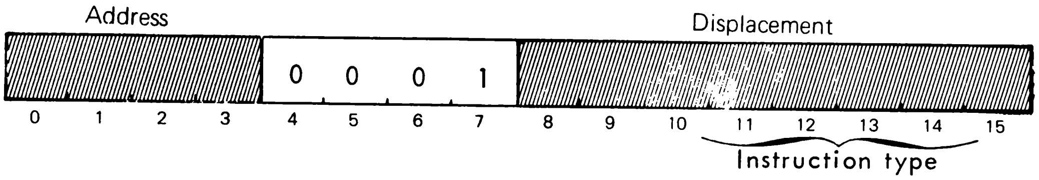
Inter-register operations will be described in the following order :

	SRG	Set ReGister
SRG derivates	XAE	eXchange A and E
	XAX	eXchange A and X
	XAA	eXchange left byte of A and right byte of A
	CCE	Copy Complement E
	ACE	Add Carry and E
	CCA	Copy Complement A
	AEE	A Exclusive OR with E
	CNX	Copy Negative X
	AIE	A Inclusive OR with E
	AAE	A And E
	LNE	Load Negative E
	CNA	Copy Negative A
	CHX	Copy Half X

NAME : Set ReGister **SRG**

Class : 1 Non-priviledged Standard

Instruction format :



Addressing mode	Hexadecimal code
DL	31
PX	E1
P	F1

Execution time depending on operation type.

Function : N_{11-14}

- 0	ReTurn Section → (RTS)
- 1	Exchange contents of A and E → (XAE)
- 2	Exchange contents of A and X → (XAX)
- 3	Exchange contents of E and X → (XEX)
- 4	Exchange (A_{0-7}) and (A_{8-15}) → (XAA)
- 5	Complement E → (CCE)
- 6	Return SuperVisor → (RSV)
- 7	Add Carry in E → (ACE)
- 8	Complement A →(CCA)
- 9	A Exclusive OR E in A → (AEE)
- A	Copy Negative X → (CNX)
- B	A OR E in A → (AIE)
- C	A and E in A → (AAE)
- D	Load Negative in E → (LNE)
- E	Copy Negative A → (CNA)
- F	Compute Half X → (CHX)

Modified elements :

- Registers : see individual instructions
- Indicators : see individual instruction

Trap conditions : standard

Examples :

SRG	EPDL11 (equivalent to XEX)
SRG	=&10,x ((x) = &E, equivalent to CHX)
SRG	=14 (equivalent to ACE)

NAME : eXchange contents of A and E **XAE**

Class : 1 Non-priviledged Standard

Instruction format :

Address											Instruction type				
1	1	1	1	0	0	0	1	▨	▨	▨	0	0	0	1	▨
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Addressing mode	Hexadecimal code	Execution time (µs)
P	F1 02	4,3

Function : (A) ⟷ (E)

Contents of A-register and E-register exchanged

Modified elements :

- Registers : E-A

Trap conditions : standard

Examples : XAE

NAME : eXchange contents of A and X **XAX**

Class : 1 Non-priviledged Standard

Instruction format :

Address											Instruction type				
1	1	1	1	0	0	0	1	▨	▨	▨	0	0	1	0	▨
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Addressing mode	Hexadecimal code	Execution time (µs)
P	F1 04	4,3

Function : (A) ⟷ (X)

Contents of A-register and X-register exchanged.

Modified elements :

- Registers : A - X

Trap conditions : standard

Examples : XAX

NAME : eXchange contents of E and X **XEX**

Class : 1 Non-priviledged Standard

Instruction format :

Address											Instruction type				
1	1	1	1	0	0	0	1	▨	▨	▨	0	0	1	1	▨
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Addressing mode	Hexadecimal code	Execution time (µs)
P	F1 06	4,3

Function : (E) ⟷ (X)

Contents of E-register and X-register exchanged.

Modified elements :

- Registers : E - X

Trap conditions : standard

Examples : XEX

NAME : eXchange left byte of A and right byte of A **XAA**

Class : 1 Non-priviledged Standard

Instruction format :

Address											Instruction type				
1	1	1	1	0	0	0	1	▨	▨	▨	0	1	0	0	▨
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Addressing mode	Hexadecimal code	Execution time (µs)
P	F1 08	2,8

Function : $(A_{0-7}) \longleftrightarrow (A_{8-15})$

Rightmost and leftmost bytes of A-register exchanged.

Modified elements :

- Registers : A

Trap conditions : standard

Examples : XAA

NAME : Copy Complement, logical E — CCE

Class : 1 Non-priviledged Standard

Instruction format :

Address											Instruction type				
1	1	1	1	0	0	0	1	▨	▨	▨	0	1	0	1	▨
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Addressing mode	Hexadecimal code		Execution time (μs)
P	F1	0A	2,8

Function : $(\bar{E}) \longleftrightarrow (E)$

One's complement of E-register contents (bit 1 → 0, bit 0 → 1).

Modified elements :

- Registers : E

Trap conditions : standard

Examples : CCE

NAME : Add Carry in E — ACE

Class : 1 Non-priviledged Standard

Instruction format :

Address											Instruction type				
1	1	1	1	0	0	0	1	▨	▨	▨	0	1	1	1	▨
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Addressing mode	Hexadecimal code		Execution time (μs)
P	F1	0E	3,4

Function : $(E) + C \longrightarrow (E)$

Carry indicator value added to E-register contents; result in E.

Modified elements :

- Registers : E
- Indicators : C-O

Indicators :

C	O	Upon execution
1	-	Carry
-	1	Overflow

Trap conditions : standard

Examples : ACE

NAME : Copy Complement, logical A — CCA

Class : 1 Non-priviledged Standard

Instruction format :

Address											Instruction type				
1	1	1	1	0	0	0	1	▨	▨	▨	1	0	0	0	▨
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Addressing mode	Hexadecimal code	Execution time (μs)
P	F1 10	2,8

Function : $(\overline{A}) \rightarrow (A)$

One's complement of E-register contents (bit 1 → 0, bit 0 → 1).

Modified elements :

- Registers : A

Trap conditions : standard

Examples : CCA

NAME : A Exclusive OR E in A — AEE

Class : 1 Non-priviledged Standard

Instruction format :

Address											Instruction type				
1	1	1	1	0	0	0	1	▨	▨	▨	1	0	0	1	▨
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Addressing mode	Hexadecimal code	Execution time (μs)
P	F1 12	3,1

Function : $(A) \oplus (E) \rightarrow (A)$

Exclusive OR between A-register contents and E-register contents; result in A.

Modified elements :

- Registers : A
- Indicators : C-O

Indicators :

C	O	Upon execution
0	0	$A > 0$
0	1	$A < 0$
1	0	$A = 0$

Trap conditions : standard

Examples : AAE

CNX

NAME : Copy Negative X

Class : 1 Non-priviledged Standard

Instruction format :

Address											Instruction type				
1	1	1	1	0	0	0	1	▨	▨	▨	1	0	1	0	▨
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Addressing mode	Hexadecimal code	Execution time (μs)
P	F1 14	3,1

Function : - (X) ⟶ (X)

Two's complement (complement to 2^{16}) of X-register contents; result in X.

Modified elements :

- Registers : X

Trap conditions : standard

Examples : CNX

AIE

NAME : A Inclusive or E in A

Class : 1 Non-priviledged Standard

Instruction format :

Address											Instruction type				
1	1	1	1	0	0	0	1	▨	▨	▨	1	0	1	1	▨
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Addressing mode	Hexadecimal code	Execution time (μs)
P	F1 16	3,1

Function : (A) $\vee$ (E) ⟶ (A)

Inclusive OR between A-register contents and E-register contents; result in A.

Modified elements :

- Registers : A
- Indicators : C-O

Indicators :

C	O	Upon execution
0	0	$A > 0$
0	1	$A < 0$
1	0	$A = 0$

Trap conditions : standard

Examples : AIE

NAME : A And E in A

AAE

Class : 1 Non-priviledged Standard

Instruction format :

Address Instruction type

1	1	1	1	0	0	0	1	▨	▨	▨	1	1	0	0	▨
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Addressing mode	Hexadecimal code	Execution time (μs)
P	F1 18	3,1

Function : (A) ∧ (E) → (A)

Logical AND (intersection) between A-register contents, and E-register contents; result in A.

Modified elements :

- Registers : A
- Indicators : C-O

Indicators :

C	O	Upon execution
0	0	A > 0
0	1	A < 0
1	0	A = 0

Trap conditions : standard

Examples : AAE

NAME : Load, Negative in E

LNE

Class : 1 Non-priviledged Standard

Instruction format :

Address Instruction type

1	1	1	1	0	0	0	1	▨	▨	▨	1	1	0	1	▨
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Addressing mode	Hexadecimal code	Execution time (μs)
P	F1 1A	3,1

Function : - 1 → (E)

-1 value loaded into E-register.

Modified elements :

- Registers : E

Trap conditions : standard

Examples : LNE

NAME : Copy Negative A **CNA**

Class : 1 Non-priviledged Standard

Instruction format :

Address											Instruction type				
1	1	1	1	0	0	0	1	▨	▨	▨	1	1	1	0	▨
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Addressing mode	Hexadecimal code		Execution time (µs)
P	F1	1C	3,4

Function : - (A) → (A)

Two's complement (complement to 2^{16}) of A-register contents; result in A.

Modified elements :

- Registers : A
- Indicators : C-O

Indicators :

C	O	Upon execution
1	-	Carry
-	1	Overflow

Trap conditions : standard

Examples : CNA

NAME : Compute Half X **CHX**

Class : 1 Non-priviledged Standard

Instruction format :

Address											Instruction type				
1	1	1	1	0	0	0	1	▨	▨	▨	1	1	1	1	▨
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Addressing mode	Hexadecimal code		Execution time (µs)
P	F1	1E	3,1

Function : (X) / 2 → (X)

X-register contents shifted one position to the right. Sign bit (bit 0) restored. As a result, X-register contents is divided by two.

Modified elements :

- Registers : X

Trap conditions : standard

Examples : CHX

VII-9. FLOATING-POINT ARITHMETIC

VII-9.1. Introduction

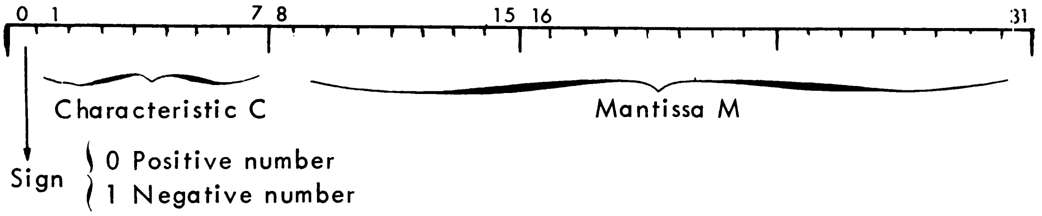
These instructions operate on single precision floating-point format operands and are implemented in an optional device called "floating-point operator" or OVF.

In single precision floating-point format, the numbers are represented on a double-word. This double-word has the following structure :

- A sign bit (position 0)

- A base-16 exponent whose value is increased by 64 to provide the "characteristic" (positions 1 through 7).

- A 6 hexadecimal digit mantissa (positions 8 through 31).



A floating-point number N has the following formel definition :

- If $N \geqslant 0$ $\quad N = M \times 16^{C-64}$

with $M = 0$ or $16^{-6} \leqslant M < 1$ and $0 \leqslant C \leqslant 127$

- A positive floating-point number having a zero mantissa must also have a zero characteristic; it represents the zero value.

- A positive floating-point number is "normalized" if its mantissa satisfies the relation : $1/16 \leqslant M < 1$.

- A negative number is represented by the two's complement of its absolute value. More specifically the representation of a negative number N includes a sign bit (0), a characteristic and a mantissa on a double-word length. The number N is represented by the two's complement of this double-word.

Normally floating-point arithmetic instructions operate on normalized operands (no pre-normalization effected) and provide normalized results.

For non-normalized operands, the results are normalized or not, as the case may be.

Limits of normalized floating-point format :

$0.86362 \times 10^{-76} < N < 0.72370 \times 10^{+76}$

VII-9.2. Description

The floating-point instructions perform the four operations and are described in the following order :

FAD	Floating ADd
FSU	Floating SUbtract
FMU	Floating MUltiply
FDV	Floating DiVide

Examples of floating-point representations :

Nombre Décimal	SINGLE PRECISION FLOATING FORMAT			Hexadecimal value
	±	C	M	
$+(16^{+63})(1-2^{-24})$	0	111 1111	1111 1111 1111 1111 1111 1111	7F FFFFFF
$+(16^{+3})(5/16)$	0	100 0011	0101 0000 0000 0000 0000 0000	43 500000
$+(16^{-3})(209/256)$	0	011 1101	1101 0001 0000 0000 0000 0000	3D D10000
$+(16^{-63})(2047/4096)$	0	000 0001	0111 1111 1111 0000 0000 0000	01 7FF000
$+(16^{-64})(1/16)$	0	000 0000	0001 0000 0000 0000 0000 0000	00 100000
0	0	000 0000	0000 0000 0000 0000 0000 0000	00 000000
$-(16^{-64})(1/16)$	1	111 1111	1111 0000 0000 0000 0000 0000	FF F00000
$-(16^{-63})(2047/4096)$	1	111 1110	1000 0000 0001 0000 0000 0000	FF 801000
$-(16^{-3})(209/256)$	1	100 0010	0010 1111 0000 0000 0000 0000	C2 2F0000
$-(16^{+3})(5/16)$	1	011 1100	1011 0000 0000 0000 0000 0000	BC B00000
$-(16^{+63})(1-2^{-24})$	1	000 0000	0000 0000 0000 0000 0000 0001	80 000001

NAME : Floating ADd (option) **FAD**

Class : 0' Non-priviledged Optional

Instruction format :

Address								Displacement							
▨	▨	▨	1	1	0	1	0	▨	▨	▨	▨	▨	▨	▨	▨
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Addressing mode	Hexadecimal code
DL	1A
DG	5A
IL	7A
IGX	9A
ILX	BA

Function : $(E, A) + (Y_2 , Y_2 + 2) \longrightarrow (E,A)$

Contents in floating format of E, A extended accumulator added with floating operand contained in Y_2 -address double-word; result in E, A.

Modified elements :

- Registers : E - A
- Indicators : C-O

Indicators :

C	O	Upon execution
0	0	Result ≠ 0, no overflow
0	1	Overflow
1	0	Result = 0
1	1	Underflow

Trap conditions : standard

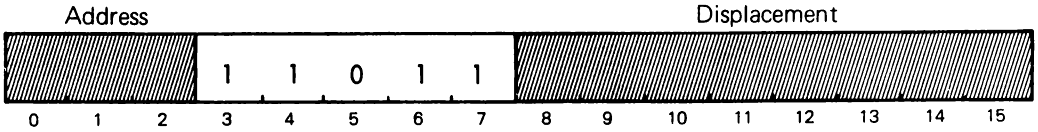
Examples :

```
FAD          EPDL26
FAD          # EPDC26
FAD          @EPDL27
FAD          @# EPDC27,x
FAD          @EPDL27, x
```

NAME : Floating SUbtract **FSU**

Class : 0' Non-priviledged Optional

Instruction format :



Addressing mode	Hexadecimal code
DL	1B
DG	5B
IL	7B
IGX	9B
ILX	BB

Function : $(E, A) - (Y_2 , Y_2 + 2) \longrightarrow (E, A)$

Floating operand contained in Y_2-address double-word subtracted from contents in floating format of E, A extended accumulator; result in E, A.

Modified elements :

- Registers : E - A
- Indicators : C-O

Indicators :

C	O	Upon execution
0	0	Result ≠ 0, no overflow
0	1	Overflow
1	0	Result = 0
1	1	Underflow

Trap conditions : standard

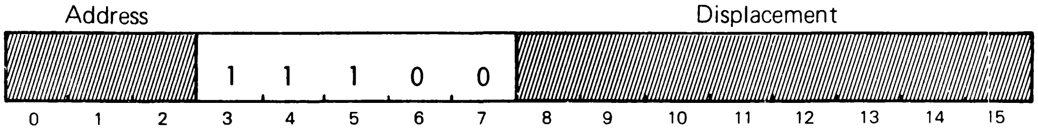
Examples :

```
FSU            EPDL26
FSU            # EPDC26
FSU            @EPDL27
FSU            @# EPDC27,x
FSU            @EPDL27,x
```

NAME : Floating MUltiply **FMU**

Class : 0' Non-priviledged Optional

Instruction format :



Addressing mode	Hexadecimal code
DL	1C
DG	5C
IL	7C
IGX	9C
ILX	BC

Function : $(E, A) \times (Y_2 , Y_2 + 2) \longrightarrow E, A$

Contents in floating format of E, A extended accumulator (multiplicand) multiplied by floating operand contained in Y_2-address double-word (multiplier); result in E, A.

Modified elements :

- Registers : E - A
- Indicators : C-O

Indicators :

C	O	Upon execution
0	0	Result ≠ 0, no overflow
0	1	Overflow
1	0	Result = 0
1	1	Underflow

Trap conditions : standard

Examples :

```
FMU          EPDL26
FMU          ≠ EPDC26
FMU          @ EPDL27
FMU          @≠ EPDC27,x
FMU          @ EPDL27,x
```

NAME : Floating DiVide **FDV**

Class : 0' Non-priviledged Optional

Instruction format :

Address								Displacement							
▨	▨	▨	1	1	1	0	1	▨	▨	▨	▨	▨	▨	▨	▨
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Addressing mode	Hexadecimal code
DL	1D
DG	5D
IL	7D
IGX	9D
ILX	BD

Function : (E, A) : (Y_2 , Y_2 + 2) ⟶ (E, A)

Contents in floating format of E, A extended accumulator (dividend) divided by floating operand contained in Y_2 -address double-word (divisor); result in E, A.

Modified elements :

- Registers : E - A
- Indicators : C-O

Indicators :

C	O	Upon execution
0	0	Result ≠ 0, non overflow
0	1	Overflow
1	0	Result = 0
1	1	Underflow

Trap conditions : standard

Examples :

FDV	EPDL26
FDV	# EPDC26
FDV	@ EPDL27
FDV	@# EPDC27,x
FDV	@ EPDL27,x

VII-10. BYTE STRING PROCESSING

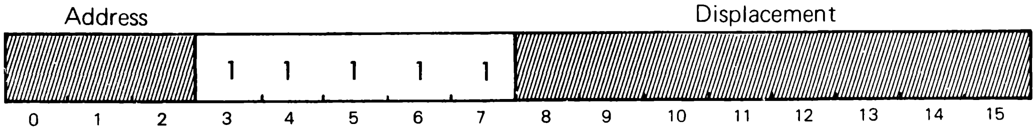
Byte or character string operations are performed by three instructions :

MVS	MoVe byte String
CPS	ComPare byte String
TRS	TRanslate byte String

NAME : MoVe byte String **MVS**

Class : 0' Non-priviledged Optional

Instruction format :



Addressing mode	Hexadecimal code	Execution time (μs)
DL	1F	3 + 3,3 n
DG	5F	3 + 3,3 n
IL	7F	4 + 3,3 n
IGX	9F	4 + 3,3 n
ILX	AF	4 + 3,3 n

n = number fo shift steps

Function : For α varying on a byte basjs from (E) - 1 to 0,
$((G) + (A) + \alpha) \longrightarrow (Y + \alpha)$
When transfer is over -1 ⟶ E

A byte string beginning at an address defined with respect to G-base by the contents of A-register and whose length (in bytes) is specified in E-register, is stored in core memory starting from Y-address.

When the transfer is over, E-register contents is -1 and A-register contents is unmodified.

Modified elements :

- Registers : E
- Memory locations : (Y) to (Y + (E) - 1)

Trap conditions : standard and non-implemented instruction

Miscellaneous : This instruction is interruptible between each byte transfer

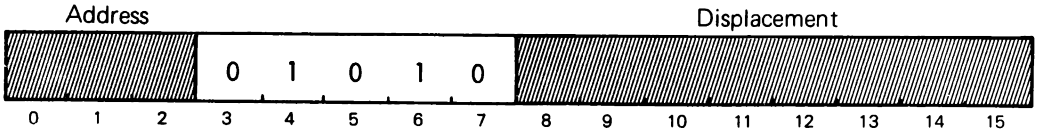
Examples :

```
MVS          EPDL4
MVS          # EPDC4
MVS          @EPDL3
MVS          @# EPDC3,x
MVS          @EPDL3,x
```

NAME : ComPare byte String **CPS**

Class : 0 Non-priviledged Optional

Instruction format :



Addressing mode	Hexadecimal code	Execution time (µs)
DL	0A	4,2 + 3,4 n
P	2A	4,6 + 3,4 n
DG	4A	4,2 + 3,4 n
IL	6A	5,2 + 3,4 n
IGX	8A	5,2 + 3,4 n
ILX	AA	5,2 + 3,4 n

n = number of comparaison made

Function : For α varying on a byte basis from 0 to (E) - 1,
$y = ((G) + (A) + \alpha)$

Upon execution :

- if found : A = address of reference byte within the string
- if not found : A = address of first non-processed byte

A byte y read at Y-address in core memory is sequentially compared to every byte of a string beginning at an address defined with respect to G-base by the contents of A-register and whose length is specified in E-register.

Upon execution :

If the reference byte is found in the string, its address with respect to G-base is in A-register and E-register contains the unprocessed string length.

If the reference byte y is not found in the string, A-register contains the starting address with respect to G-base and E-register final contents is zero.

Modified elements :

- Registers : E - A
- Indicators : C-O

Indicators :

C	O	Upon execution
0	0	Character found
0	1	Character no found

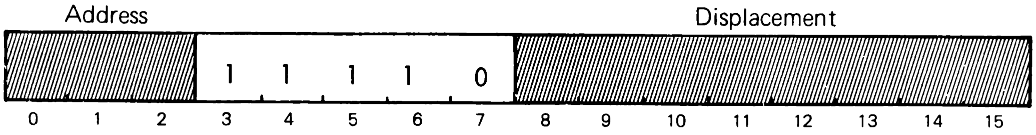
Trap conditions : standard and non-implemented instruction
Miscellaneous : This instruction is interruptible between byte comparisons.
Examples :

CPS	EPDL18	CPS	@ EPDL19
CPS	=&6C	CPS	@ # EPDC19,x
CPS	# EPDC18	CPS	@ EPDL19,x

NAME : TRanslatable byte String **TRS**

Class : 0' Non-priviledged Optional

Instruction format :



Addressing mode	Hexadecimal code	Execution time (μs)
DL	1E	4,5 + 2,75 n
DG	5E	4,5 + 2,75 n
IL	7E	5,5 + 2,75 n
IGX	9E	5,5 + 2,75 n
ILX	BE	5,5 + 2,75 n

n : number of bytes to be translated

Function : For α varying on a byte basis from 0 to (E) - 1
$(Y + ((A) + (G) + \alpha)_e) \longrightarrow ((A) + (G) + \alpha)$
When transfer is over, $0 \longrightarrow (E)$

Given : an origin code O and a result code R, as well as a 256 consecutive byte translation table starting at Y-calculated address and organized as follow :

Table relative beginning address hexadecimal)	Contents
00	Value of R-code corresponding to 00 value in O-code.
01	Value of R-code corresponding to 01 value in O-code.
02	Value of R-code corresponding to 02 value in O-code.
⋮	⋮
FF	Value of R-code corresponding to FF value in O-code.

A string beginning at an address defined with respect to G-base by the contents of A-register and whose length is specified in E-register is translated byte per byte through the translation table by TRS instruction.

Starting from Y calculated address, the origin string is overwritten byte per byte by the result string.

Translation table creation is obviously the user's responsibility.

Modified elements :

- Registers : A (if initial E ≠ 0) - E
- Memory locations : ((A)) to ((A) + (E) - 1)

Trap conditions : standard and non-implemented instruction

Examples :

TRS	EPDL18	TRS	@ ≠ EPDC19,x
TRS	≠ EPDC18	TRS	@ EPDL19,x
TRS	@EPDL19		

VII-11. BRANCH INSTRUCTIONS

These instructions normally provide for interrupting the sequential instruction execution in a program segment.

They include :

BRU	BRanch Unconditional
BRX	BRanch with indeX
BCT	Branch on Carry True
BOT	Branch on Overflow True
BCF	Branch on Carry False
BOF	Branch on Overflow False
BAZ	Branch on A Zero
BAN	Branch on A Negative
BE	Branch on Equel to
BZ	Branch on Zero
BL	Branch on Less than
BLZ	Branch on Less than Zero
BNE	Branch on Not Equal to
BNZ	Branch on Not Zero
BGE	Branch on Greater than or Equal to
BPZ	Branch on Positive or Zero

Remark 1 :

An indicator is "true" when set and "false" when reset.

Remark 2 :

If P-register contents is even, the executed instruction is located at (P) absolute address.

If P-register contents is odd, the executed instruction is located at (P) - 1 absolute address.

Remark 3 :

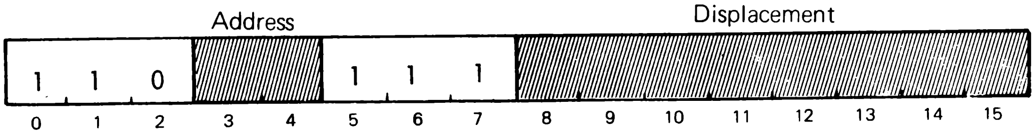
Several conditional branches correspond to the same basic instruction. These additional mnemonics (not recognized by MITRAS I) are provided for easier program writing when used after a load or compare instruction.

Basic instruction	Mnemonics for use after loading	Mnemonics for use after comparison
BCT	BZ	BE
BOT	BLZ	BL
BCF	BNZ	BNE
BOF	BPZ	BGE

NAME : BRanch Unconditional **BRU**

Class : 2 Non-priviledged Standard

Instruction format :



Addressing mode	Hexadecimal code	Execution time (μs)
RP	C7	1,8
RM	CF	1,8
IL	D7	3
IG	DF	3

Function : Y ⟶ (P)

Y-address is loaded into P-register and execution proceeds at Y-address.

Modified elements :

- Registers : P

Trap conditions : standard

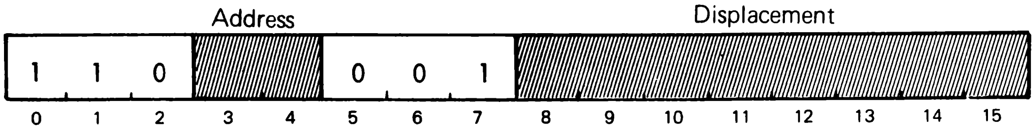
Examples :

```
                BRU     @EPDL14
EPPL2           BRU     EPPL3+1
EPPL3           BRU     @# EPDC13
                BRU     EPPL3-1 ⟶ Equivalent to BRU   EPPL2
                BRU     $
```

NAME : BRanch indeXed **BRX**

Class : 2 Non-priviledged Standard

Instruction format :



Addressing mode	Hexadecimal code	Execution time (µs)
RP	C1	2,1
RM	C9	2,1
IL	D1	3,2
IG	D9	3,2

Function : RP : (P) + 2 De + 2 (X) → (P)
RM : (P) - 2 De - 2 (X) → (P)
IL : (De + (L) + (X)) + G' → (P)
IG : (De + (G) + (X)) + G' → (P)

Y calculated address is loaded into P-register and execution proceeds at Y-address. In indirect addressing, the index is used for a pre-indexing executed prior to indirect addressing.

Y-address is calculated according to the above formulae.

Remark :

BRX may be used with relative plus or minus addressing modes, but its most frequent application is based on indirect addressing to provide multiple branch capability through an address table.

Modified elements :

- Registers : P

Trap conditions : standard

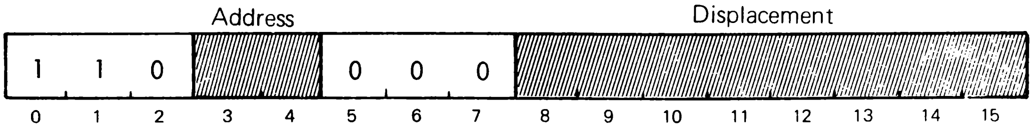
Examples :

EPPL0	BRX	EPPL1	(for X = 2, branch at EPPL3)
EPPL1	BRX	EPPL3	(for X = 1, branch at EPPL4)
EPPL2	BRX	EPPL3	(for X = 0, branch at EPPL3)
EPPL3	BRX	EPPL2	(for X = 1, branch at EPPL1)
EPPL4	BRX	EPPL2	(for X = 2, branch at EPPL0)
	BRX	@ EPDL15	(for X = 0, branch at EPPL1)
	BRX	@ # EPDC15	(for X = 2, branch at EPPL2)
	BRX	@ EPDL15	(for X = 4, branch at EPPL3)
	BRX	@ # EPDC15	(for X = 6, branch at EPPL4)

NAME : Branch on C True **BCT**

Class : 2 Non-priviledged Standard

Instruction format :



Addressing mode	Hexadecimal code	Execution time (μs)	
RP	C0	2,1	1,9
RM	C8	2,1	1,9
IL	D0	3,3	1,9
IG	D8	3,3	1,9

If continue in sequence
If branch

Function : If C = 1 $\Longrightarrow$ Y → (P)
If C = 0 $\Longrightarrow$ (P) + 2 → (P)

If Carry indicator is set (1), Y calculated address in loaded into P-register and execution continues at Y-address.

If Carry indicator is reset (0), execution proceeds in sequence.

Modified elements :

- Registers : P

Indicators :

C	O	Function according to initial value
0		Continue in sequence
1		Branch

Trap conditions : standard

Examples :

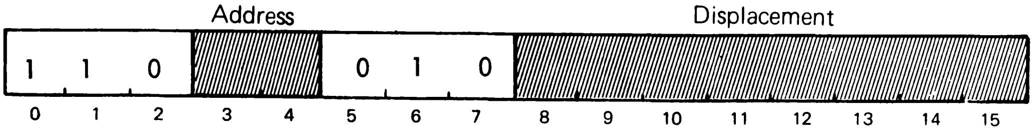
```
                BCT     @# EPDC14
EPPL2           BCT     $+1
EPPL3           BCT     EPPL3+1 →   Equivalent to BCT $+1
                BCT     @EPDL13
```

BOT

NAME : Branch on O True

Class : 2 Non-priviledged Standard

Instruction format :



Addressing mode	Hexadecimal code	Execution time (µs)	
RP	C2	2,1	1,9
RM	CA	2,1	1,9
IL	D2	3,3	1,9
IG	DA	3,3	1,9

If continue in sequence
If branch

Function : If O = 1 $\Longrightarrow$ Y $\longrightarrow$ (P)
If O = 0 $\Longrightarrow$ (P) + 2 $\longrightarrow$ (P)

If Overflow indicator is set (1), Y calculated address is loaded into P-register and execution continues at Y-address.

If Overflow indicator is reset (0), execution proceeds in sequence.

Indicators :

C	O	Function according to initial value
	0	Continue in sequence
	1	Branch

Trap conditions : standard

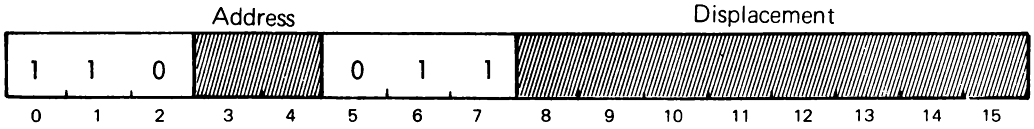
Examples :

EPPL4	BOT	EPPL1
EPPL1	BOT	@ # EPDC14
EPPL3	BOT	@EPDL13
EPPL2	BOT	EPPL4

NAME : Branch on C False **BCF**

Class : 2 Non-priviledged Standard

Instruction format :



Addressing mode	Hexadecimal code	Execution time (µs)	
RP	C3	2,1	1,9
RM	CB	2,1	1,9
IL	D3	3,3	1,9
IG	DB	3,3	1,9

If continue in sequence
If branch

Function : If C = 0 $\Longrightarrow Y \longrightarrow (P)$
If C = 1 $\Longrightarrow (P) + 2 \longrightarrow (P)$

If Carry indicator is reset (0), Y calculated address is loaded into P-register and execution continues at Y-address.

If Carry indicator is set (1), execution proceeds in sequence.

Modified elements :

- Registers : P

Indicators :

C	O	Function according to initial value
0		Branch
1		Continue in sequence

Trap conditions : standard

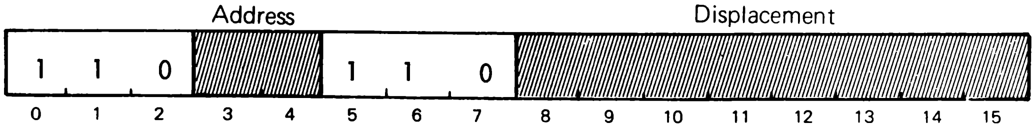
Examples :

EPPL4	BCF	EPPL1
EPPL3	BCF	EPPL4
EPPL2	BCF	@ EPDL14
EPPL1	BCF	@ # EPDC13

NAME : Branch on O False **BOF**

Class : 2 Non-priviledged Standard

Instruction format :



Addressing mode	Hexadecimal code	Execution time (μs)	
RP	C6	2,1	1,9
RM	CE	2,1	1,9
IL	D6	3,3	1,9
IG	DE	3,3	1,9

If continue in sequence
If branch

Function : If O = 0 $\Longrightarrow$ Y → (P)
If O = 1 $\Longrightarrow$ (P) + 2 → (P)

If Overflow indicator is reset (0), Y calculated address is loaded into P-register and execution continues at Y-address.

If Overflow indicator is set (1), execution proceeds in sequence.

Modified elements :

- Registers : P

Indicators :

C	O	Function according to initial value
	0	Branch
	1	Continue in sequence

Trap conditions : standard

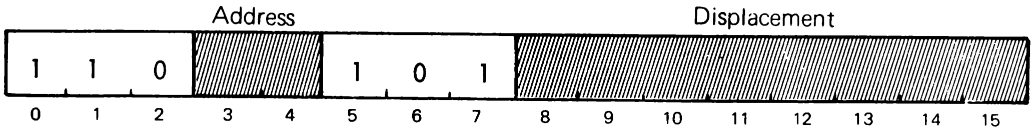
Examples :

```
EPPL1        BOF      EPPL2+2
EPPL2        BOF      @EPDL14
EPPL3        BOF      @# EPDC13
             BOF      EPPL2-1 → Equivalent to BOF   EPPL1
```

NAME : Branch on A Zero **BAZ**

Class : 2 Non-priviledged Standard

Instruction format :



Addressing mode	Hexadecimal code	Execution time (µs)	
RP	C5	2,4	2,2
RM	CD	2,4	2,2
IL	D5	3,5	2,2
IG	DD	3,5	2,2

If continue in sequence
If branch

Function : If $(A) = 0 \Longrightarrow Y \rightarrow (P)$
If $(A) \neq 0 \Longrightarrow (P) + 2 \rightarrow (P)$

If A-register contents is zero, Y calculated address is loaded into P-register and execution continues at Y-address.

If A-register contents is not zero, execution proceeds in sequence.

Modified elements :

- Registers : P

Trap conditions : standard

Examples :

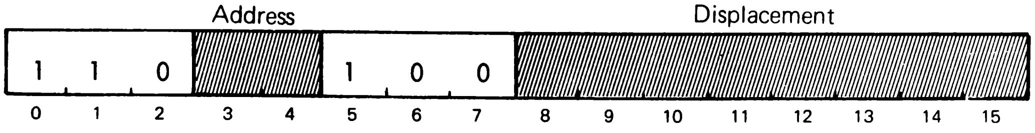
```
EPPL2     BAZ     @# EPDC14
          BAZ     $+2
EPPL2     BAZ     $-1
          BAZ     @EPDL13
```

NAME : Branch on A Negative

BAN

Class : 2 Non-priviledged Standard

Instruction format :



Addressing mode	Hexadecimal code	Execution time (µs)	
RP	C4	2,4	2,2
RM	CC	2,4	2,2
IL	D4	3,5	2,2
IG	DC	3,5	2,2

If continue in sequence
If branch

Function : If (A) < 0 ⟶ Y ⟶ (P)
If (A) ⩾ 0 ⟶ (P) + 2 ⟶ (P)

If A-register contents is negative, Y calculated address is loaded into P-register and execution continues at Y-address.

If A-register contents is zero or positive, execution proceeds in sequence.

Modified elements :

- Registers : P

Trap conditions : standard

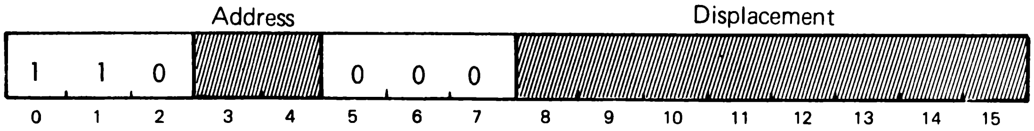
Examples :

EPPL1	BAN	$+2
EPPL3	BAN	EPPL1
	BAN	@EPDL13
EPPL2	BAN	@# EPDC14

NAME : Branch if Equal **BE**

Class : 2 Non-priviledged Standard

Instruction format :



Addressing mode	Hexadecimal code	Execution time (μs)	
RP	C0	2,1	1,9
RM	C8	2,1	1,9
IL	D0	3,3	1,9
IG	D8	3,3	1,9

If continue in sequence
If branch

Function : If Carry = 1 $\Longrightarrow$ Y $\longrightarrow$ (P)
If Carry = 0 $\Longrightarrow$ (P) + 2 $\longrightarrow$ (P)

Normally, this instruction is preceded by a comparison.

If the first term of the comparison was equal to the second, Y calculated address is loaded into P-register and execution continue at Y-address.

If the first term of the comparison was different from the second, execution proceeds in sequence.

Modified elements :

- Registers : P

Indicators :

C	O	Function according to initial value
0		Continue in sequence
1		Branch

Trap conditions : standard

Miscellaneous : This instruction is equivalent to a BCT

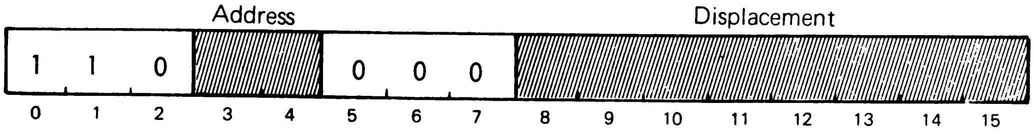
Examples :

```
EPPL1       BE      EPPL4
EPPL4       BE      @# EPDC13
EPPL2       BE      @EPDL14
EPPL3       BE      EPPL1
```

NAME : Branch if Zero **BZ**

Class : 2 Non-priviledged Standard

Instruction format :



Addressing mode	Hexadecimal code	Execution time (µs)	
RP	C0	2,1	1,9
RM	C8	2,1	1,9
IL	D0	3,3	1,9
IG	D8	3,3	1,9

If continue in sequence

If branch

Function : If Carry = 1 $\Longrightarrow$ Y → (P)
If Carry = 0 $\Longrightarrow$ (P) + 2 → (P)

This instruction is normally preceded by a load instruction.

If the previously stored value is zero, Y calculated address is loaded into P-register and execution continues at Y-address.

If the previously stored value is different from zero, execution proceeds in sequence.

Modified elements :

- Registers : P

Indicators :

C	O	Function according to initial value
0		Continue in sequence
1		Branch

Trap conditions : standard

Miscellaneous : This instruction is equivalent to a BCT

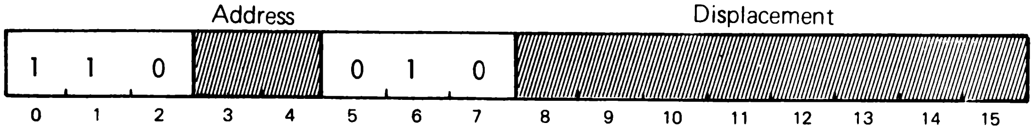
Examples :

```
EPPL4        BZ        @EPDL14
EPPL2        BZ        EPPL1
EPPL3        BZ        @# EPDC13
EPPL1        BZ        EPPL4
```

NAME : Branch if Less **BL**

Class : 2 Non-priviledged Standard

Instruction format :



Addressing mode	Hexadecimal code	Execution time (μs)	
RP	C2	2,1	1,9
RM	CA	2,1	1,9
IL	D2	3,3	1,9
IG	DA	3,3	1,9

If continue in sequence
If branch

Function : If Overflow = 1 $\Longrightarrow$ Y → P
If Overflow = 0 $\Longrightarrow$ (P) + 2 → (P)

Normally, this instruction is preceded by a comparison.

If the first term of the comparison was less than the second, Y calculated address is loaded into P-register and execution continues at Y-address.

If the first term of the comparison was equal to or greater than the second, execution proceeds in sequence.

Modified elements :

- Registers : P

Indicators :

C	O	Function according to initial value
	0	Continue in sequence
	1	Branch

Trap conditions : standard

Miscellaneous : This instruction is equivalent to a BOT

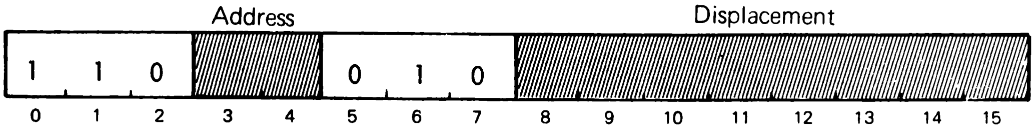
Examples :

```
EPPL4         BL         @EPDL14
EPPL2         BL         EPPL1
EPPL3         BL         @# EPDC13
EPPL1         BL         EPPL4
```

NAME : Branch if Less Zero **BLZ**

Class : 2 Non-priviledged Standard

Instruction format :



Addressing mode	Hexadecimal code	Execution time (µs)	
RP	C2	2,1	1,9
RM	CA	2,1	1,9
IL	D2	3,3	1,9
IG	DA	3,3	1,9

If continue in sequence
If branch

Function : If Overflow = 1 $\Longrightarrow$ Y → (P)
If Overflow = 0 $\Longrightarrow$ (P) + 2 → (P)

This instruction is normally preceded by a load instruction.

If the previously stored value was less than zero, Y calculated address is loaded into P-register and execution at Y-address.

If the previously stored value was equal to or greater than zero, execution proceeds in sequence.

Modified elements :

- Registers : P

Indicators :

C	O	Function according to initial value
	0	Continue in sequence
	1	Branch

Trap conditions : standard

Miscellaneous : This instruction is equivalent to a BOT

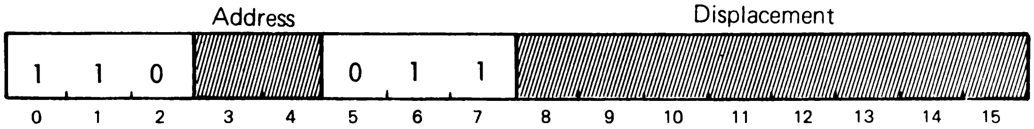
Examples :

```
EPPL0          BLZ       @# EPDC14
EPPL2          BLZ       EPPL1
EPPL3          BLZ       @EPDL13
EPPL1          BLZ       $-3   → Equivalent to BLZ     EPPL0
```

NAME : Branch if Not Equal **BNE**

Class : 2 Non-priviledged Standard

Instruction format :



Addressing mode	Hexadecimal code	Execution time (μs)	
RP	C3	2,1	1,9
RM	CB	2,1	1,9
IL	D3	3,3	1,9
IG	DB	3,3	1,9

If continue in sequence
If branch

Function : If Carry = 0 $\Longrightarrow$ Y → (P)
If Carry = 1 $\Longrightarrow$ (P) + 2 → (P)

Normally, this instruction is preceded by a comparison.

If the first term of the comparison was different from the second, Y calculated address is loaded into P-register and execution continues at Y-address.

If the first term of the comparison was equal to the second, execution proceeds in sequence.

Modified elements :

- Registers : P

Indicators :

C	O	Function according to initial value
0		Branch
1		Continue in sequence

Trap conditions : standard

Miscellaneous : This instruction is equivalent to a BCF

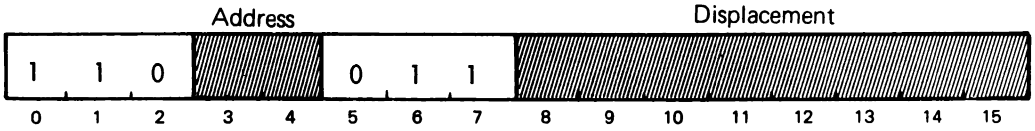
Examples :

```
              BNE      $+1
              BNE      @EPDL13
EPPL2         BNE      @# EPDC14
EPPL3         BNE      $-3
```

NAME : Branch if Not Zero **BNZ**

Class : 2 Non-priviledged Standard

Instruction format :



Addressing mode	Hexadecimal code	Execution time (µs)	
RP	C3	2,1	1,9
RM	CB	2,1	1,9
IL	D3	3,3	1,9
IG	DB	3,3	1,9

If continue in sequence
If branch

Function : If Carry = 0 $\Longrightarrow$ Y $\rightarrow$ (P)
If Carry = 1 $\Longrightarrow$ (P) + 2 $\rightarrow$ (P)

This instruction is normally preceded by a load instruction.

If the previously stored value was different from zero, Y calculated address is loaded into P-register and execution continues at Y-address.

If the previously stored value was equal to zero, execution proceeds in sequence.

Modified elements :

- Registers : P

Indicators :

C	O	Function according to initial value
0		Branch
1		Continue in sequence

Trap conditions : standard

Miscellaneous : This instruction is equivalent to a BCF

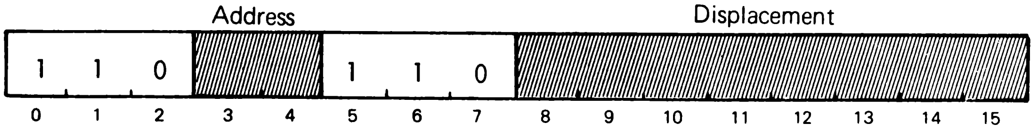
Examples :

EPPL3	BNZ	EPPL1
EPPL4	BNZ	@EPDL13
EPPL2	BNZ	@# SPDC14
EPPL1	BNZ	EPPL4

NAME : Branch if Greater or Equal **BGE**

Class : 2 Non-priviledged Standard

Instruction format :



Addressing mode	Hexadecimal code	Execution time (µs)	
RP	C6	2,1	1,9
RM	CE	2,1	1,9
IL	D6	3,3	1,9
IG	DE	3,3	1,9

If continue in sequence
If branch

Function : If Overflow = 0 $\Longrightarrow$ Y → (P)
If Overflow = 1 $\Longrightarrow$ (P) + 2 → (P)

Normally, this instruction is preceded by a comparison.

If the first term of the comparison was greater than or equal to the second, Y calculated address is loaded into P-register and execution continue at Y-address.

If the first term of the comparison was less than the second, execution proceeds in sequence.

Modified elements :

- Registers : P

Indicators :

O	Function according to initial value
0	Branch
1	Continue in sequence

Trap conditions : standard

Miscellaneous : This instruction is equivalent to a BOF

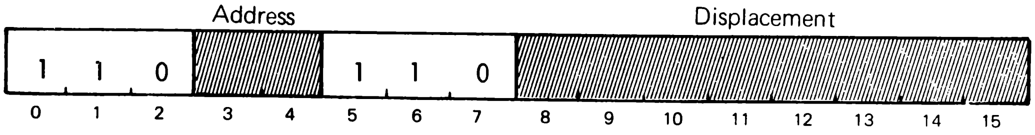
Examples :

```
EPPL4          BGE     EPPL1
EPPL1          BGE     @EPDL13
EPPL2          BGE     @# EPDC14
EPPL3          BGE     EPPL4
```

NAME : Branch if Positive or Zero **BPZ**

Class : 2 Non-priviledged Standard

Instruction format :



Addressing mode	Hexadecimal code	Execution time (μs)	
RP	C6	2,1	1,9
RM	CE	2,1	1,9
IL	D6	3,3	1,9
IG	DE	3,3	1,9

If continue in sequence
If branch

Function : If Overflow = 0 $\Longrightarrow$ Y → (P)
If Overflow = 1 $\Longrightarrow$ (P) + 2 → (P)

This instruction is normally preceded by a load instruction.

If the previously stored value was greater than or equal to zero, Y calculated address is loaded into P-register and execution continues at Y-address.

If the previously stored value was less than zero, execution proceeds in sequence.

Modified elements :

- Registers : P

Indicators :

O	Function according to initial value
0	Branch
1	Continue in sequence

Trap conditions : standard

Miscellaneous : This instruction is equivalent to a BOF

Examples :

```
EPPL2      BPZ    @EPDL14
EPPL3      BPZ    $+2
           BPZ    @# EPDC13
           BPZ    $-2  → Equivalent to BRU    EPPL3
```

VII-12. SYSTEM COMMUNICATION INSTRUCTIONS

These instructions perform sophisticated call and communication operations between system constituents : program modules, monitor modules, interrupt subroutines.

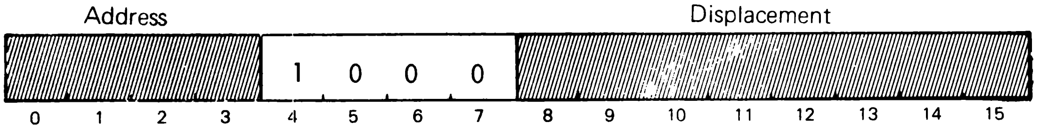
They include :

CLS	CalL Section
RTS	ReTurn Section
CSV	Call SuperVisor
DIT	De-activate InTerrupt
DITR	De-activate high-speed InteRrupt

NAME : CalL Section **CLS**

Class : 1 Non-priviledged Standard

Instruction format :



Addressing mode	Hexadecimal code	Execution time (µs)	
DL	38	8,3	7,9
PX	E8	8,3	7,9
P	F8	8,3	7,9

In master mode
In slave mode

Function : $(P) - G' \longrightarrow (((G) - 4\,N) + (G))$
$(L) - G' \longrightarrow (((G) - 4\,N) + (G) + 2)$

$(G) + ((G) - 4\,N) \longrightarrow (L)$
$(G) + ((G) - 4\,N + 2) \longrightarrow (P)$

where N is the calculated operand, i.e. (Y_2)

$N \neq 0$

Note

As discussed before, a program is made up of a number of sections individually characterized by a local base L and a program base P. These characteristic values are entered in the program's PRT (one SRD per section).

Purpose of the CLS

A CLS is basically a branch instruction providing connection between a given section ("calling") and another section ("called") of the same program, while ensuring permanent communications between the two sections as well as an easy return means.

Involved elements

- Program's PRT

- First two words of the calling section's LDS

- Contents of CLS calculated address which contains the called section number. (Note that Assemblers and Compilers which operate with a Linkage Editor offer the convenient possibility of calling a section by its name; thus, the actual number of the section may be unknown to the programmer).

Operation of the CLS

- L- and P-base values, relative to G, of the calling section are stored in the first two words of the called section's LDS.

- With the section number, L- and P-base values of the called section are fetched and stored in the corresponding registers.

- A branch is made at the called section.

Communication with the calling section

Three methods are available for transferring the parameters between calling and called sections :

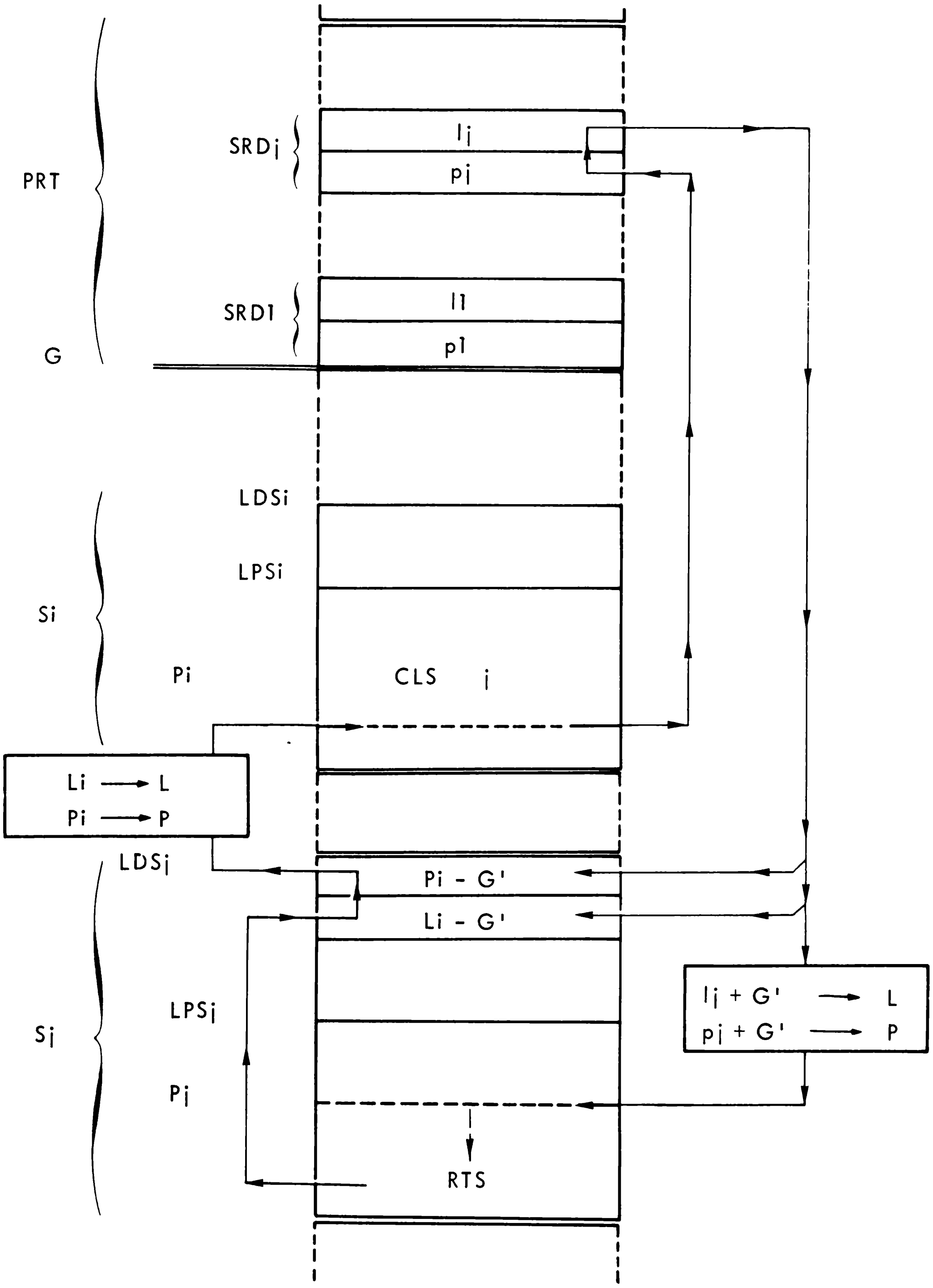
1) Through the Common Data Section (CDS)

2) In indirect local (IL) or indirect local indexed (ILX) addressing modes, via the second word of the LDS.

3) Via A, E, X-registers and/or C-O indicators.

Operation flow chart

j = called section no.
i = calling section no.



Modified elements :

- Registers : L - P

- Memory locations : first two words of called section's LDS.

Trap conditions : standard

Miscellaneous : The CLS operating with elements of the called section is not re-entrant. It may be used both in Master mode and Slave mode.

Examples :

```
CLS            LPS1
CLS            =2
CLS            =2,x
```

NAME : ReTurn Section **RTS**

Class : 1 Non-priviledged Standard

Instruction format :

Address											Instruction type				
1	1	1	1	0	0	0	1	▨	▨	▨	0	0	0	0	▨
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Addressing mode	Hexadecimal code	Execution time (μs)
P	F1 00	4,3 4,7

In master mode: 4,7
In slave mode: 4,3

Function : ((L)) + G' + 2 → (P)
((L) + 2) + G' → (L)

The RTS placed in a section called by a CLS provides the return to the calling section by restoring in L and P-registers the corresponding values contained in the first two words of the called section's LDS.

The operation of the RTS is illustrated on the CLS diagram.

Modified elements :

- Registers : L - P

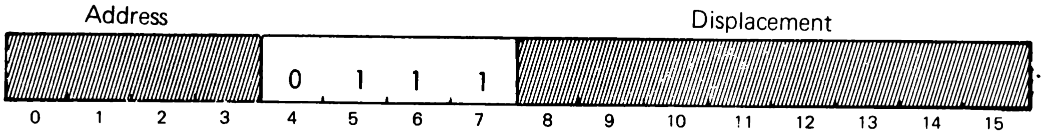
Trap conditions : standard

Examples : RTS

NAME : Call SuperVisor **CSV**

Class : 1 Non-priviledged Standard

Instruction format :



Addressing mode	Hexadecimal code	Execution time (µs)
DL	37	9,9
PX	E7	9,9
P	F7	9,9

Function : (P) - (G) → ((G))
(L) - (G) → ((G) + 2)
Indicators → ((G) + 4)

1 → PR
1 → MS
((12) - 4 N) → (L)
((12) - 4 N + 2) → (P)

Where N is the calculated operand, i.e. (Y_2)

Note :

As discussed before, a monitor is made up of a number of sections individually characterized by a local base L and a program base P. These characteristic values are entered in the PRTS (Supervisor's PRT) located anywhere in the memory and pointed at through absolute address 12 (decimal). A monitor operates in Master mode and overrides memory protection.

Purpose of the CSV

A CSV is basically a branch instruction providing connection between a user program section and a supervisor section, while ensuring the re-entrance of the Supervisor section and an easy return to the user program.

Involved elements

- PRTS (Supervisor's PRT)

- PRTS pointer

- First three words of the calling program's CDS

- Contents of CSV calculated address which contains the called section number. (Note that Assemblers and Compilers which operate with a Linkage Editor offer the convenient possibility of calling a supervisor section by a name M:xxxx; thus, the actual number of the section may be unknown to the programmer).

Operation of the CSV

- L- and P-base values, relative to G, of the current section, and its indicators are stored in the first three words of the calling program's CDS.

- With the section number, L- and P-base values are fetched.

- MA and PR indicators are forced to 1 .

- A branch is made at the called section.

Communication with the calling section

These communications are made in indirect general indexed (IGX) addressing mode via the second word of the calling program's CDS.

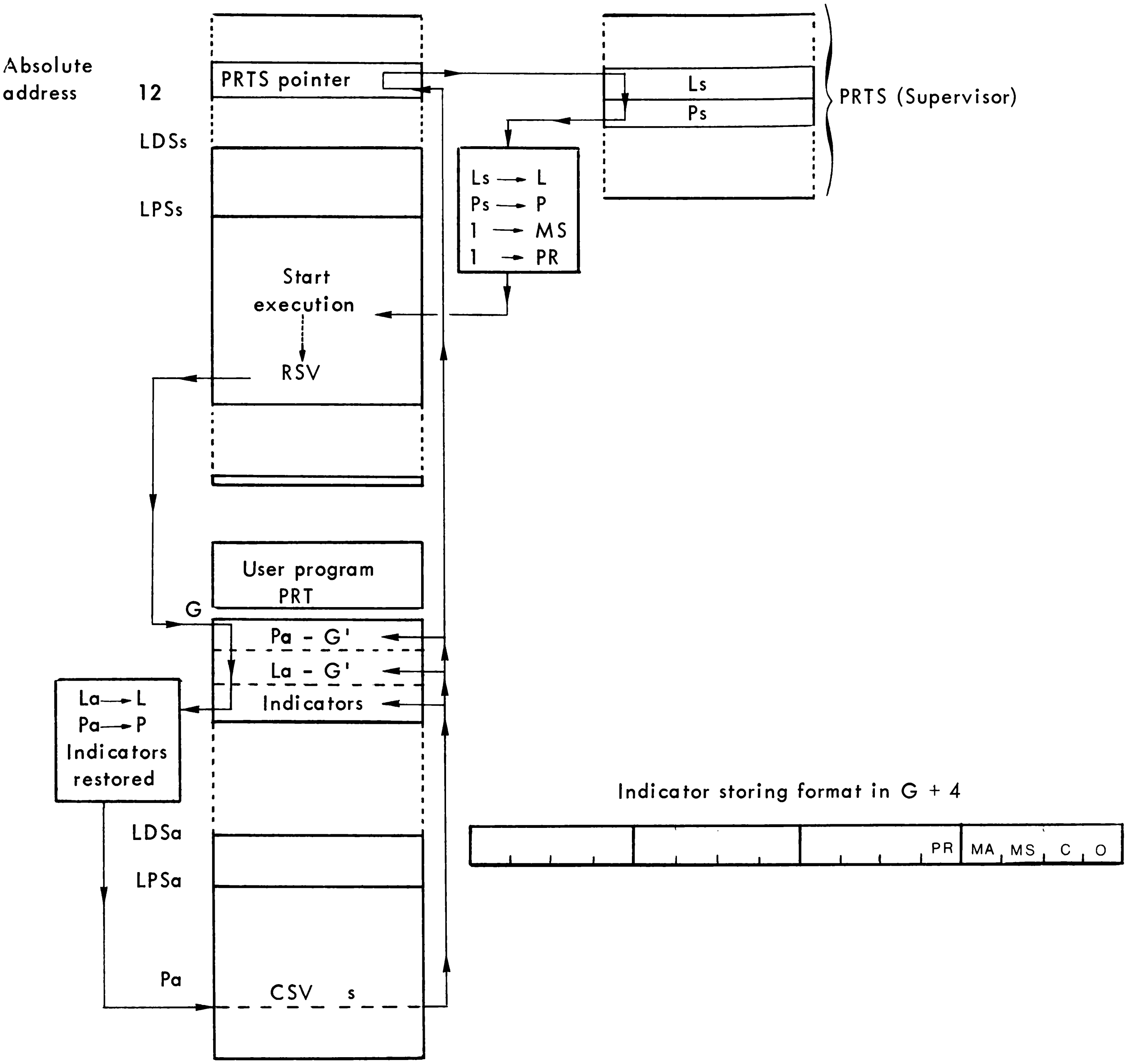
However, it is still possible to transfer the parameters via A, E and X-registers.

Re-entrance

For full re-entrance, the supervisor section stores all variable elements it handles in the calling program's CDS (addressing mode : direct general or direct general indexed).

Standard modules use a 16-word TWB for this purpose.

Operation flow chart



Modified elements :

- Registers : L-P
- Memory locations : The first three words of the calling's program's CDS, i.e. (G) through (G+5)
- Indicators : MS set
  PR set

Trap conditions : standard

Miscellaneous : The program is interruptible between a CSV and the immediately following instruction (for interrupt masking, should it be required).

Examples :

```
CSV             M:IO
CSV             =2
CSV             =2,x
```

NAME : Return SuperVisor

RSV

Class : 1 Priviledged Standard

Instruction format :

Address								Instruction type							
1	1	1	1	0	0	0	1	▨	▨	▨	0	1	1	0	▨
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Addressing mode	Hexadecimal code	Execution time (µs)
P	F1C0	6,7

Function : ((G) + 4) → Indicators
(G) + ((G) + 2) → (L)
(G) + 2 + ((G)) → (P)

The RSV placed in a monitor section (Master mode) called by a CSV provides the return to the calling section by restoring in L- and P-registers the corresponding values contained in the first two words of the CDS of the program to which the calling section belongs. It also restores the initial status of the indicators.

Modified elements :

- Registers : L-P
- Indicators : All indicators are restored (to the value they had before the CSV if the G-base has not been changed in the meantime).

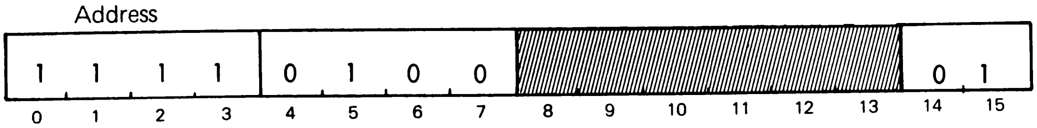
Trap conditions : standard and mode violation

Examples : RSV

NAME : De-active InTerrupt **DIT**

Class : 1 Priviledged Standard

Instruction format :



Addressing mode	Hexadecimal code	Execution time (µs)
P	F4	32,5

Function :

Note

As discussed before, the level of a program is the interrupt level at which it may be activated. A program which cannot be activated at any interrupt level is said to be "at zero level".

MITRA 15's interrupt system is based on structure of hierarchized levels.

When an interrupt is received at a higher level than that of the current program, the latter is interrupted otherwise. If the interrupt level is lower than that of the current program the interrupt is placed in waiting state unitl the upper level is de-activated.

Purpose of the DIT

Since the acceptance of an interrupt causes a branch to the corresponding subroutine, the function of DIT is to terminate this interrupt subroutine and to return control to the interrupted program.

In this respect, the DIT instruction is an actual system branch .

Elements involved

The program context comprises the current contents of X, E, A, G, L, P registers and indicators.

Every interrupt has an associated pointer indicating a memory area in which the context may be saved on occurence of an interrupt at this level. The memory area for saving the context at a given level is actually reserved only if a program is connected to this level. Generally, this area is located immediately after the program storage area.

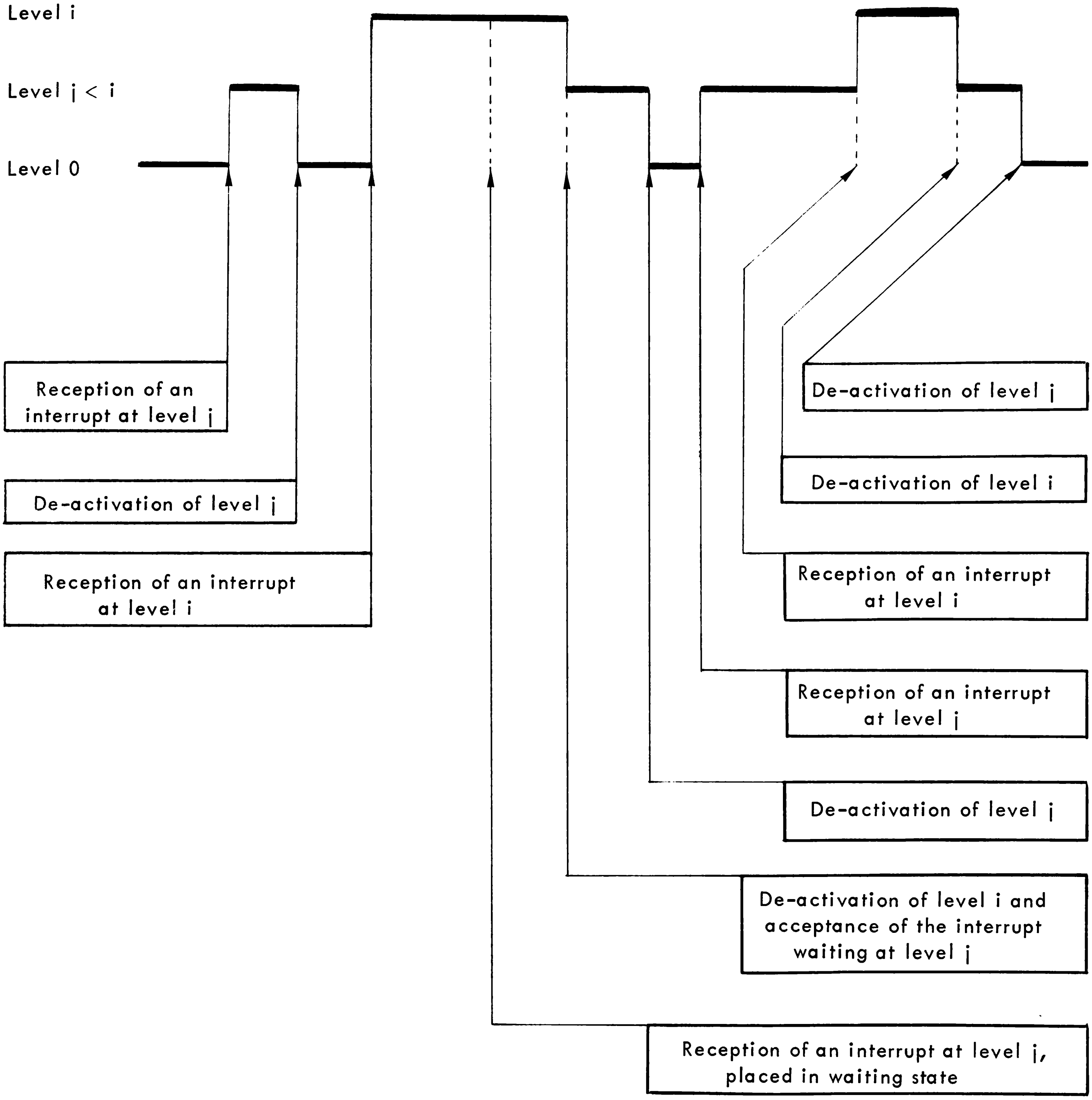
This area contains the register and indicator values at the last interrupt time (either if a higher level interrupt has been accepted, or if the level has been de-activated). If the program is never interrupted, the saving area contains the initial program contents (at the time its execution is started).

The 32 context pointers (indicating the saving area of each level) are stored in increasing level order in the context pointer table (at increasing addresses starting from CPT address).

Saved elements are stored on a one-element-per-word basis at increasing addresses starting from the pointed address in the following order :

- Indicators, X, E, A, G, L, P.

Example of interrupt process


Level i
Level j < i
Level 0
Reception of an interrupt at level j
De-activation of level j
Reception of an interrupt at level i
De-activation of level j
De-activation of level i
Reception of an interrupt at level i
Reception of an interrupt at level j
De-activation of level j
De-activation of level i and acceptance of the interrupt waiting at level j
Reception of an interrupt at level j, placed in waiting state

Operation of the DIT

- P-register incremented by 2
- Current level de-activated
- Context stored from the address which is pointed by the corresponding context pointer
- Acceptance of the highest priority waiting level (R8 updated)
- Context loaded with the contents of the saving area which is pointed at by the new level's pointer (execution is started).

Acceptance of an interrupt

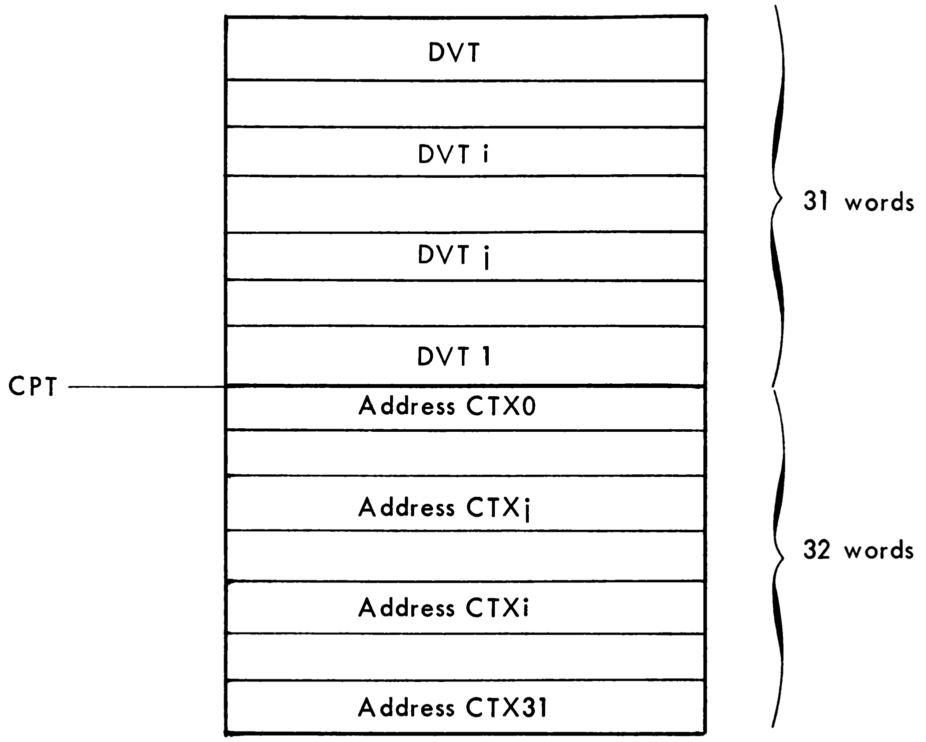
DIT performs the return branch of the interrupt subroutine which had been call at the time the corresponding interrupt was accepted. This instruction :

- Stores the interrupted level context from the memory address indicated by the associated context pointer.
- Accepts the level of the highest priority interrupt (R8 up-dated)
- Loads the context from the saving area which is pointed at by the new level (execution is started).

Interrupt configurations

There are 31 interrupt configurations (obviously zero level needs none). These configurations are used by DIT to know located exactly the interrupt level to be de-activated.

They are stored in ascending level order in DVT table (in descending address order starting from CPT address).



Where i > j

This table may be located anywhere in the memory, the CPT address being in a word whose absolute address is 10 (decimal).

Current program level

The current program level is indicated by R8-register which contains the double of this level number.

Modified elements :

- Registers : A, E, X, P, L, G, R8
- Memory locations : The eight words of the de-activated interrupt context
- Indicators : All

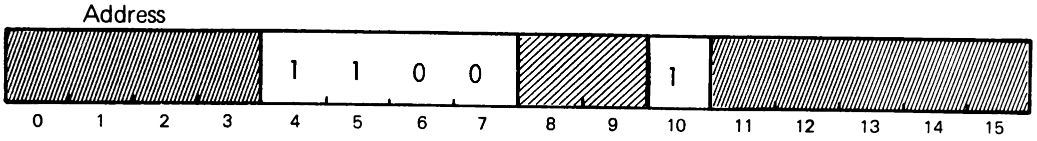
Trap conditions : standard and mode violation

Miscellaneous : Since DIT performs a context swapping, no protection or masking is required.
All interrupt subroutine must conclude with a DIT.
A DIT is meaningless at level zero (which cannot be de-activated).
Examples : DIT

NAME : De-activate high-speed InTerrupt **DITR**

Class : 1 Priviledged Optional

Instruction format :



Addressing mode	Hexadecimal code		Execution time (μs)
PX	EC	20	6,1
P	FC	20	6,1

Function :

Comparative analysis of normal and high-speed interrupt

- Acceptance of a normal interrupt includes the following operations :

- Context of currently executed level (specified by R8) is saved in core memory.

- Calling level (Na) is accepted and R8 is updated.

- Context elements corresponding to Na are loaded in the registers.

- Acknowledgment of a normal interrupt is performed by the final DIT instruction of the interrupt subroutine and includes the following operations :

- Context of currently processed interrupt is saved in core memory.

- Corresponding level is de-activated.

- Waiting level is accepted and R8 is updated.

- Context elements corresponding to the new level are loaded in the registers.

- Acceptance of the high-speed interrupt includes the following operations :

- Normal interrupts are placed in waiting status until acknowledgment of the high-speed interrupt.

- Current indicators are saved in register 6 of block 0.

- R12 is loaded with the number of the block which is reserved for high-speed interrupt processing.

- Indicators are loaded with the contents of register 6 in the reserved block.

- Acknowledgment of the high-speed interrupt is performed by the final DITR instruction of the interrupt subroutine and includes the following operations :

- Indicators are saved in register 6 of the reserved block.

- R12 is cleared (return to block 0).

- High-speed level is de-activated (normal interrupts are enabled).

- Previous indicators saved in register 6 of block 0 are restored.

Modified elements :

- Registers : return to block 0
- Indicators : restored

Trap conditions : standard and mode violation

Examples : DITR

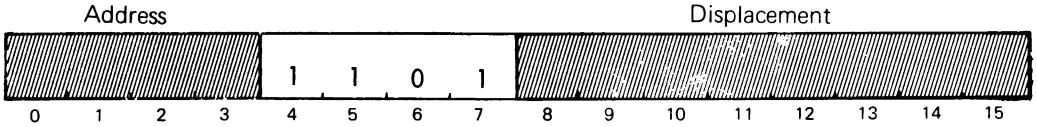
VII-13. CONTROL INSTRUCTIONS

TES	TEst and Set
STM	SeT interrupt Mask
CLM	CLear interrupt Mask
RD	Read Direct
WD	Write Direct
LDP	LoaD memory Protection

NAME : TEst and Set **TES**

Class : 1 Priviledged Optional

Instruction format :



Addressing mode	Hexadecimal code	Execution time (µs)
DL	3D	3,6
PX	ED	3,4
P	FD	3,4

Function :

$$P \left\{ \begin{array}{l} (De)_2 \longrightarrow A \\ 0 \longrightarrow (De)_2 \end{array} \right.$$

$$PX \left\{ \begin{array}{l} (De + (X))_2 \longrightarrow A \\ 0 \longrightarrow (De + (X))_2 \end{array} \right.$$

$$DL \left\{ \begin{array}{l} ((De + (L)))_2 \longrightarrow A \\ 0 \longrightarrow ((De + (L)))_2 \end{array} \right.$$

This instruction tests and clears a memory location without being interrupted. Initial value loaded in A. Test result loaded in the indicators. This instruction is used when several processors work in a common memory area, for example to enable a processor to perform an "occupation test" on a table.

The protection bit is also reset.

Modified elements :

- Registers : A
- Memory locations : y_2
- Indicators : C-O

Indicators :

C	O	Upon execution
0	0	$y_2 < 0$
0	1	$y_2 > 0$
1	0	$y_2 = 0$

Trap conditions : Mode violation

Examples :

```
TES             EPDL28
TES             =&3E,x
TES             =9
```

STM

NAME : SeT interrupt Mask

Class : 1 Priviledged Standard

Instruction format :

Address

1	1	1	1	0	1	0	0	▨	▨	▨	▨	1	▨	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Addressing mode	Hexadecimal code	Execution time (μs)
P	F4	3,4

Functions : (N_{12}) ⟶ Interrupt mask (MA)

MA-indicator is set.
As a consequence all interrupt levels are masked.

Modified elements :

- Indicators : MA

Trap conditions : standard and mode violation

Examples : STM

CLM

NAME : CLear interrupt Mask

Class : 1 Priviledged Standard

Instruction format :

Address

1	1	1	1	0	1	0	0	▨	▨	▨	▨	▨	▨	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Addressing mode	Hexadecimal code	Execution time (μs)
P	F4	3,4

Function : (N_{12}) ⟶ Interrupt mask (MA)

MA-indicator is reset. As a consequence, all interrupt levels are masked.

Modified elements :

- Indicators : MA

Trap conditions : standard and mode violation

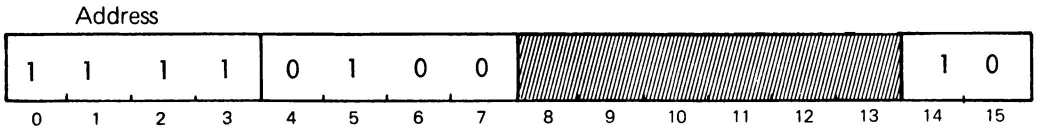
Examples : CLM

RD

NAME : Read Direct

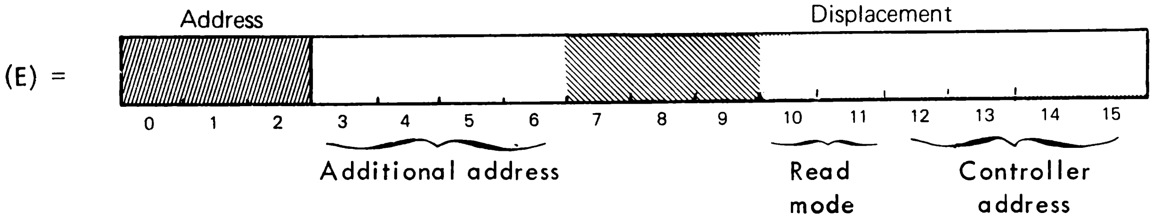
Class : 1 Priviledged Standard

Instruction format :



Addressing mode	Hexadecimal code	Execution time (µs)
P	F4	3,5

Function : The read mode is determined by the contents of E-register



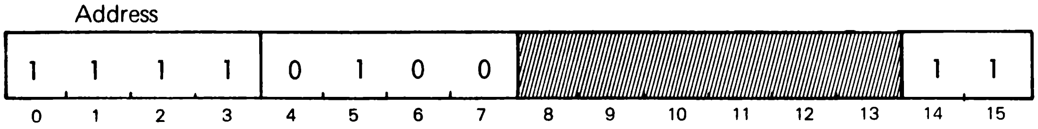
RD is the input instruction and its meaning depends on the addressed controller.

WD

NAME : Write Direct

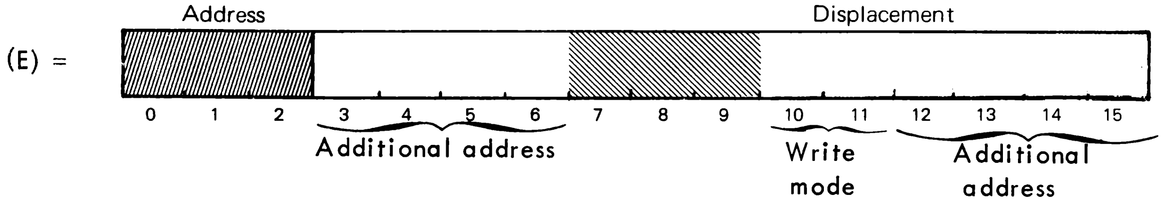
Class : 1 Priviledged Standard

Instruction format :



Addressing mode	Hexadecimal code	Execution time (µs)
P	F4	3,5

Function : The write mode is determined by the contents of E-register.



WD is the output instruction and its meaning depends on the addressed controller.

NAME : LoaD memory Protection LDP

Class : 1 Priviledged Optional

Instruction format :

Address								Displacement							
▨	▨	▨	▨	1	0	1	1	▨	▨	▨	▨	▨	▨	▨	▨
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Addressing mode	Hexadecimal code	Execution time (μs)
DL	3B	3,1 + 1,8 n
PX	EB	3,1 + 1,4 n
P	FB	3,1 + 1,4 n

n = number of words in the string.

Function : For X varying 0 and (E)-1
$N_{15} \rightarrow bp((A) + 2X)$

Upon execution E = -1
A = Address of the first non-processed word

LDP loads the protection bit in a word string.

The starting address is in A and the string length is specified in E.

Each protection bit is loaded with the value of bit 15 in the calculated operand (contents of calculated address). The string is protected when this bit is set (1), otherwise, there is no protection.

Modified elements :

- Registers : E-A
- Memory locations : (y_2) to $(y_2 + (E) - 1)$

Trap conditions : standard and mode violation

Miscellaneous : This instruction is interruptible between two words

Examples :

```
LDP         =1
LDP         =0
LDP         =0,x
LDP         EPDL21
```

8. Input/Output control system

VIII-1. INPUT/OUTPUT SYSTEM ORGANIZATION

The following sections only describe Input/Output operations under Monitor control.

Though the procedure has a standard aspect and is mainly oriented towards standard peripheral transfers, it will be seen later that it also provides for processing non-standard peripheral transfers and offers the convenient feature of a direct user check on the validity of such transfers.

An Input/Output operation may be divided into five steps :

a) Reservation of a buffer, if required.

b) Transfer request (resident system call).

c) Transfer initialization.

d) Transfer completation.

e) Transfer end and validity check.

Step a) is and independent supervisor call : reservation of a dynamic data block.

Steps a) and b) are performed by the requesting task (CSV M:IO or M:ZIO).

Step c) is performed by the Monitor in response to the M:IO call. Control is returned to the calling program upon transfer initialization.

Steps d) and e) are controlled by the system in full independence from the calling task, and in apparent simultaneity with the currently executed tasks; the calling program may follow the transfer progress, if required (CSV M:WAIT or M:ZWAT).

Remark :

If the device is busy at transfer request time, two different processes may be initiated (according to the Monitor).

1) The request is queued and control is returned to the calling task.

From the user's viewpoint, everything goes on as if transfer initialization step c) were actually completed, i.e. he may consider that step d) has begun and may wait for transfer end, if required.

In such a case, step d) combines the delay for waiting for the logical release of the device and the duration of the actual transfer initialization and performance.

Queuing is always possible at or above MTR-level.

2) No queuing of the request; control is returned to the calling task only when the transfer has beep actually initialized.

VIII-2. INPUT/OUTPUT INTERFACE

VIII-2.1. Definition

Most of supervisor processing is common to all types of transfer during steps b), c) and e) described above :

1) Transfer request is analyzed for a possible preliminary processing,

2) Logical occupation of the handler is tested,

3) If required, the transfer request is queued and the Supervisor manages the queue,

4) All parameters required for physical initialization and transfer control are set up,

5) If possible, the device initial status is partially or completely read and analyzed if the whole procedure or part of it is common to all devices,

6) A branch is made to the specific control module of the device (handler),

7) A number of error conditions are acknowledged and processed when the transfer is over (sole processing specific to step e)).

The Input/Output interface of the Operating System includes the whole of these processings common to all transfers which are performed between the time the user issues a request and the time a handler takes over, and between the transfer end and the release of its results to the user.

- Input/Output processing levels

Three level classes may be considered :

a) Physical level (level 0) at which the handler performs operations which are specific to a given type of device.

b) Logical level (level 1) which includes steps 2 through 7 above. This level is that of the resident core of the basic monitor. It is usually called "I/O supervisor" (or more restrictively, "I/O interface").

c) Upper levels (2 and above) which perform step 1 of the above processing according to a hierarchy which defines the degree of sophistication of the operating system :

- records blocking/unblocking,
- file management system,
- I/O macros of sophisticated languages (FORTRAN, ...),
- specialized packages.

Level 1 controls all actual user transfers and constitutes a resident program module independent from user programs, with a number of specialized tools or "handlers" corresponding to the various device types.

In contrast, levels 2 and above are usually subroutine librairies supplied with the system for static or dynamic integration to a task, according to their origin :

- either resident as common subroutines,
- or selected in a system library and generated with a program by the linkage editor.

These subroutines are mutually related according to the general hierarchy which has been defined elsewhere for subroutines in general.

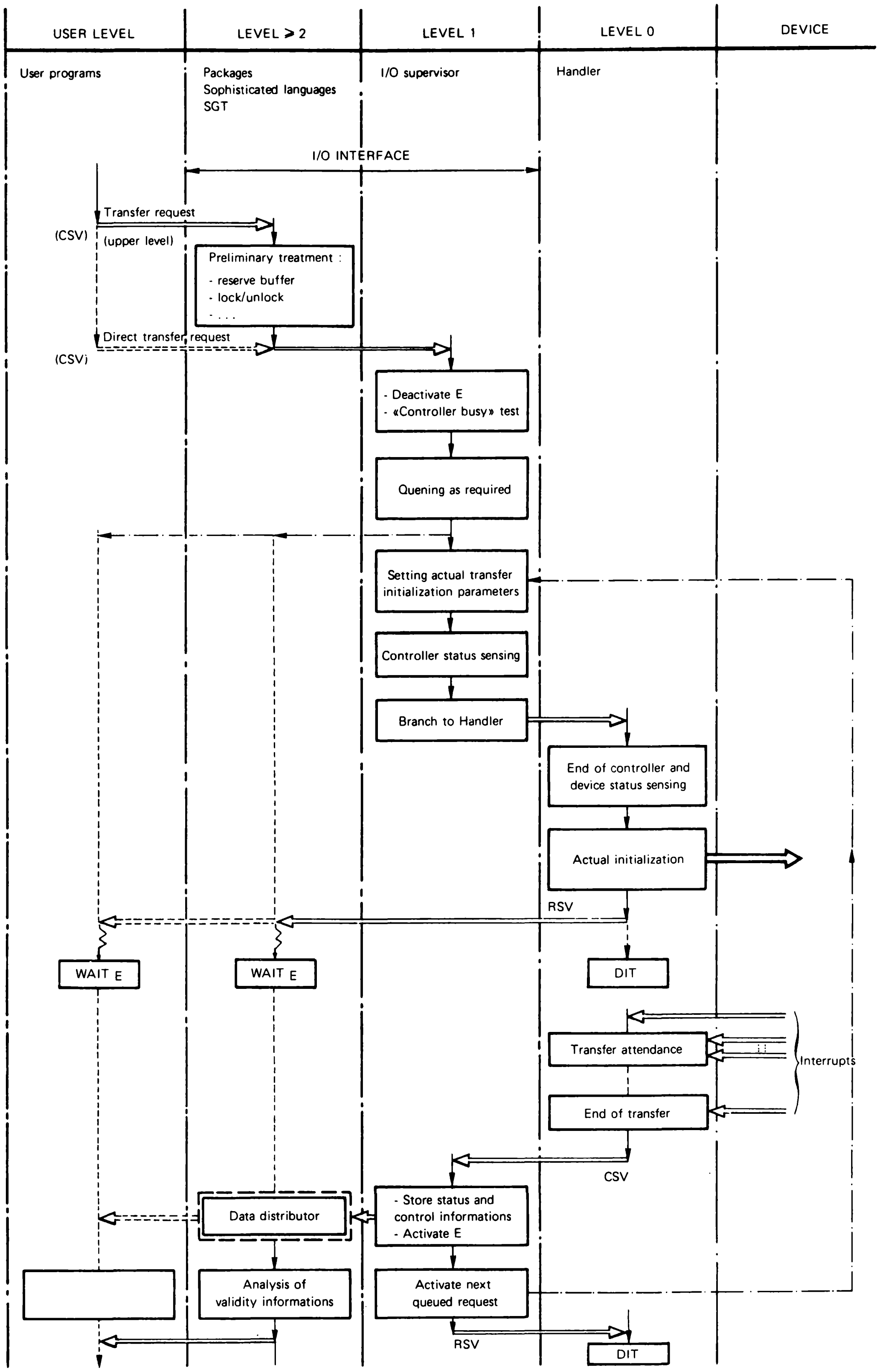
The block diagram page VIII-3 illustrates the general I/O organization.

VIII-2.2. Input/Output control module or "handler"

Hereafter, a device controller should be regarded as including both the physical controller unit and the coupling micro-program.

The actual initialization of the transfer to or from the device, the initialization control, the possible transfer attendance and the end of transfer acknowledgment depend on the type of device, i.e. of the controller. These operations must be performed by a module which is specific of the controller.

This specialized control module is called a "handler".



General Input/output organization

It includes two sections :

- Handler 1 or "initialization handler",
- Handler 2 or "transfer control handler".

Depending on the type of device, the associated controller will operate in one of the following two modes :

- blocked data transfers,
- individual data item transfers.

If a transfer requested at level 1 applies to blocked data, the handler operating mode will be one of the following :

- The handler controls the transfer of one data block in one single operating cycle;
- The handler controls the elementary data item transfers to or from the data block. This transfer is either fully controlled by handler 2, or every elementary data transfer is controlled by handler 1 cyclically re-activated by handler 2 (for exemple to avoid rewriting in handler 2 all initialization and checking functions of handler 1).

- Handler 1 : initialization of a transfer

Handler 1 has the following functions :

a) If required, it analyses the initial status of the controller and/or device, before initiating the transfer. In case of abnormal conditions, the transfer is aborted and the error conditions are signalled to the I/O supervisor.

b) It issues a block transfer or read/write command, according to the operating modes of the controller and handler 2.

c) It checks that the controller actually takes over the initialization for abnormal conditions which do not require an interrupt. If an abnormal condition is detected, the procedure is as defined in a) above.

d) It returns the control to the calling program.

Handler 1 always operates at the priority level of the calling program.

The calling program is usually the I/O supervisor (level 1), but it may also be handler 2 when the data transfers are automatically re-activated.

- Handler 2 : transfer control

Handler 2 is a Master mode immediate program which is triggered by the controller interrupts to perform the following functions :

- it takes over the transfer steps upon interrupt, in turn with the micro-program;
- if required, it re-activates handler 1;
- it controls the end of transfer operations.

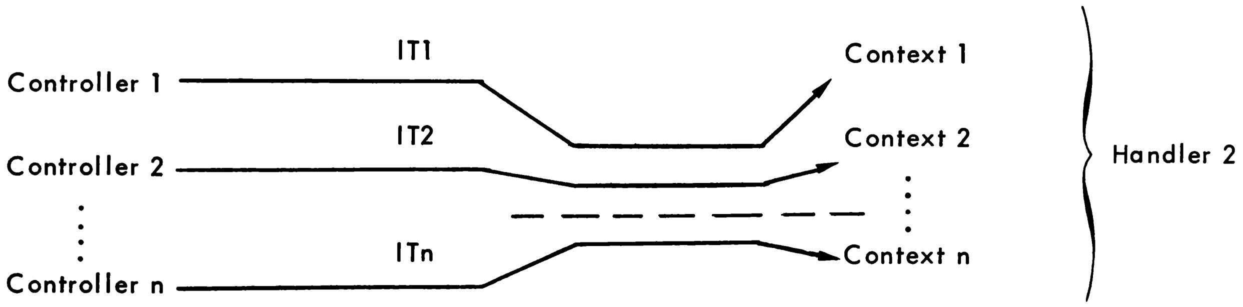
During each step, handler 2 may detect abnormal conditions which may required the transfer to be interrupted or resumed. In the latter case, the transfer is resumed by a return to handler 1.

Handler 2 always operates at the priority level of the controller interrupt.

At the end of the transfer (normal or abnormal), handler 2 performs a call to the I/O supervisor to signal the end and the validity of the transfer, before de-activating the associated interrupt level.

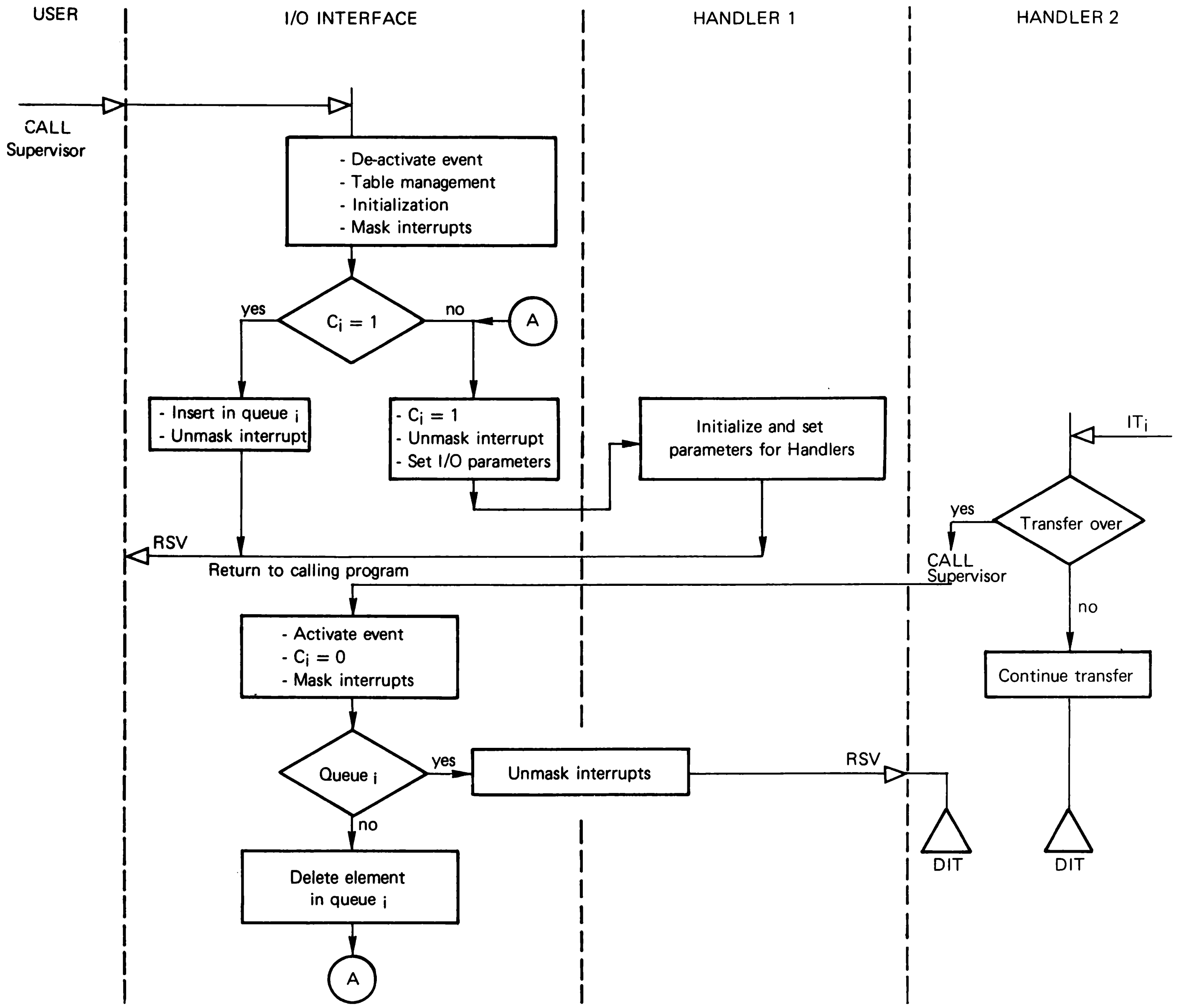
Remark :

Several controllers of the same type may be serviced in apparent simulaneity by handler 2. In effect, the corresponding interrupts are then grouped at the same level before reaching the processing unit, so that when handler 2 is occupied by a given interrupt, the other interrupts are waiting.



When queue management is available, each device controller has its own queue. Each queue contains chained elements and is managed through a pair of pointers. The queue elements are taken among free elements which are also chaine and managed through a pair of "hole pointers".

A "busy indicator" Ci is associated with every serviced controller i.



Input/Output requests management

VIII-3. TRANSFERS

VIII-3.1. Transfer requests

- I/O calls : CSV M:IO or CSV M:ZIO

A transfer control block CB is associated with the CALL SUPERVISOR of the transfer request. This block contains user-defined parameters. At the end of the transfer, it also contains status information supplied by the system.

When the CALL is executed, A-register should contain the CB address relative to G-base. When this address is defined during execution, one may conveniently use a LEA instruction which provides resolution relative to G whatever the mode, master or slave.

```
The calling sequence is then :  LEA CB
                                CSV M:IO
```

For CSV M:ZIO (available at and above MTR-level), which provides for common area I/O's, the address contained in A must be relative to the common area address (available in location 6 of the program).

The resident associates to the transfer request a dynamic event defined by the CB address and returns the control to the calling program once the transfer is initialized or queued.

The user may then perform a call to the M:WAIT module (wait event) for an event which is the beginning or the end of the transfer.

- Description of the control block parameters

	0	Event byte
	1	Indicators
	2	Command (function)
	3	Operational label
	4	Buffer address relative to G or ZC according to M:IO or M:ZIO
	5	
	6	Number of bytes to be transferred
	7	
Optional	8	Branch address on error or abnormal
	9	
Optional	10	Additional information (sector address on disk storage)
	11	
Optional	12	Time-out
	13	
Optional	14	Interrupt level no
	15	Reserved byte

For M:IO call, the I/O buffer address and the error branch address must always be relative to G.

In Slave mode this is always the case.

In Master mode, where references are normally translated into absolute addresses at loading time, these references must be preceded by a "#" character at assembly time (or a "." character in case of a LP compilation), to preserve relocatability.

For M:ZIO these addresses will be relative to ZC.

- Byte 0 : event byte

When the transfer is over, the resident sets the bits of the "event byte" according to the following code (bit 0 is set when the resident takes over and reset at the end of the transfer).

- Bit 0
  - = 0 : I/O operation over
  - = 1 : I/O operation in progress
- Bit 1
  - = 1 : error or abnormal end
- Bit 2
  - = 0 : logical error (e.g. incorrect call format)
  - = 1 : physical error (e.g. signalled by the controller)
- Bit 3
  - = 0
    - U = 0 : error detected after transfer end
    - U = 1 : status information supplied when the transfer is over
  - = 1
    - U = 0 : error during transfer initialization
    - U = 1 : status information supplied when the transfer is initialized
- Bits 4, 5, 6 and 7 : error or abnormal end code.

Bits 2 through 7 are significant only if bit 1 is set (=1).

Bits 4, 5, 6, 7 provide 16 different codes for every combinaison of bits 2 and 3.

- Byte 1 :

0	1	2	3	4	5	6	7
U	E	S	T	I			

U = 1 : the user checks the transfer on resultant status information.

E = 1 : branch address in case of error or abnormal end (standard error processing).

S = 1 : additional information present (e.g. sector address on disk storage)

T = 1 : time out requested

I = 1 : activate interrupt after transfer

U = 1 and E = 1 conditions are mutually exclusive.

- Byte 2 : specifies the requested I/O function according to the following coding table :

	0	1	2	3	4	5	6	7
Read, forward	0	0	0	0	-	-	-	-
Read, backward	0	0	1	0	-	-	-	-
Write	1	0	0	0	-	-	-	-
Write EOD/Tape mark	1	0	1	0	-	-	-	-
Format/rewind	1	0	-	1	-	-	-	-
Skip block, forward	-	1	0	0	-	-	-	-
Skip file, forward	-	1	0	1	-	-	-	-
Skip block, backward	-	1	1	0	-	-	-	-
Skip file, backward	-	1	1	1	-	-	-	-

- Byte 3 : contains the numerical value of the operational label specified at assembly time (index in I/O tables). (See section VIII-4. hereafter).

- Bytes 4 and 5 : contain the address of the first byte to be transferred to/from the core memory.

- Bytes 6 and 7 : contain the number of bytes to be transferred.

- Bytes 8 and 9 : (optional)

If U = 0 and E = 1, they contain a user-defined branch address used in case of error or abnormal end.

If U = 1 and E = 0, they contain the transfer status bits supplied by the controller and set by the resident.

- Bytes 10 and 11 : (optional)

They contain additional information whose basic function is to indicate a sector address on disk storage. For other devices, they may contain an other parameter or even the address of an additional parameter table.

- Bytes 12 and 13 : (optional)

They contain the requested time-out value.

- Bytes 14 and 15 : (optional)

They contain the interrupt level number which must be triggered by the resident when the transfer is over.

Remark :

Optional bytes are evaluated in the order of appearance of U, E, S, T, and I indicators.

When a given option is specified, the bytes corresponding to the preceding options must be reserved.

VIII-3.2. Transfer validity checking (CSV M:WAIT or CSV M:ZWAT)

The transfer validity may be ascertained either in standard mode, or in user mode :

- Standard mode (U = 0)

All controller status bits are sensed by the resident (interface and handler which returns an easily interpretable abnormal condition code to the user. If requested in the initial CB (E = 1), the resident may return the control to a user specified address in case of abnormal condition.

If the error is impossible to correct, the control is not returned to the user, but to the typewriter after printing the corresponding program identifier and error message.

- User mode (U = 1, E = 0)

The resident loads the status bits in bytes 8 and 9 without performing any sensing; the control is always returned to the user program.

In both cases, the transfer is considered to be over, even if a blocking condition occurs during initialization.

- CSV M:WAIT (corresponding to CSV M:IO)
- Calling sequence :

```
LEA           CB
CSV           M:WAIT
```

At the time the CSV is met, the address relative to G of the I/O transfer control block must be stored in A-register.

- Function :

Wait for completion of an I/O transfer.

If the transfer is already over when the M:WAIT call is received, control is returned to the user (possibly at a restart address for abnormal conditions, if requested in byte 1 of the CB).

If the transfer is still in progress when the M:WAIT call is received, the latter :

- stores the calling task level in the I/O CB;

- de-activates this level if not zero, or waits for the completion of the transfer if the level is zero.

M:WAIT call temporarily de-activates the level of the program which is waiting for an I/O transfer end, in order that lower priority programs be undertaken without being blocked. The pending program is restarted as soon as the transfer is over, its level being re-activated by the handler of the controller involved in the transfer.

- Outputted elements :

$A < 0$ in case of I/O error.

In this case, the user should examine CB byte 0 for ascertaining the cause of the error.

$A \geqslant 0$ no error

- CSV M:ZWAT

Its function is the same as that of M:WAIT but relates to a M:ZIO call. The transmitted CB address must be relative to ZC.

VIII-3.3. Communication between I/O interface and handler

The I/O interface (level 1) is basically made up of two modules :

      M:IO
and M:IO2

- Initializer M:IO

- De-activation of the event which is associated with the CB by resetting bit 0 of CB byte 0.

- Bits 1 through 7 reset in CB byte 0.

- Controller occupation tested and a waiting loop established, if required

- Initialization of logical parameters required for transfer control.

- Branch to the transfer control module (H1) of the device.

M:IO supplies the following parameters to H1

(A) = user buffer absolute address

(E) = element of OLTB table associated with the transfer operational label (see table description)

(X) = user CB absolute address

$(T3)_{TWB}$ = byte count for transfer

- Upon return from H1, M:IO examines A-register contents.

$A = 0 \longrightarrow$ return to calling program

$A \neq 0 \longrightarrow A_{9-15}$ loaded in $CB\ (0)_{1-7}$; associated event activated : $CB(0)_0 = 0$;

and :

If U-indicator $[CB(1)_0]$ = 1 or

If U = 0 and A $>$ 0, control is returned to the user

If U = 0 and A $<$ 0, M:IO edits an error message and the user task is aborted.

Under MOB Basic Monitor, only one user I/O request is allowed at a time for a given device : no queue is provided.

Every I/O request of the user program will be associated with an event and the user may request by program to wait until it is activated.

Any other I/O request for the same controller (e.g. on console interrupt) occuring before the event is activated will cause a waiting loop at calling program level, until the first transfer is over.

Interrupts should be masked during the test, since the higher priority console interrupt may request an I/O transfer and change the logical status of the devices.

Under other Monitors, the user may issue several I/O requests for the same device; they will be queued and satisfied in turn as soon as the device becomes available.

- Handler 1 : H1

- Senses the controller and/or device initial status as regards specific conditions of this device type, if required;

- Sends the transfer request according to the controller and handler 2 operation mode;

- Checks the proper acceptance of this initialization by the controller for abnormal conditions which do not cause an interrupt;

- Returns the control to the calling program.

Note :

Handler 1 always operates at the calling program level, i.e. at the physical interrupt level to which it is connected.

Upon occurence of abnormal conditions, the transfer is considered over and the error conditions are sent to interface 1 through A-register.

- Transfer end

When the transfer initialized by handler 1 is over, the interrupt associated with the controller activates handler 2.

We saw before that a block transfer may be carried out in two different ways :

(i) Transfer of the whole block in one time.

(2) Transfer of the block per data items.

- First case : block fully transferred

Handler 2 :

- Reads device status

- Checks proper device operation during the transfer

- Calls M:IO2

H2 supplies M:IO2 with

(T1) = Element of OLTB table
(T0) = User CB absolute address
(A) = Report on the operation which has just been terminated.

Module IO2 :

- Processes transfer errors, if any.
- Re-initializes the logical parameters which were used during the transfer.
- Activates the associated event (CB).
- Returns the control to the calling program.

- Second case : transfer interrupted

Handler 2 :

- Reads the device status,
- Checks for proper transfer : on error, module M:IO2 is called,
- Tests if all data have been transferred : if transfer is over, see first case above,
- Restarts the next data transfer by a branch to handler 1,
- De-activates the interrupt level.

VIII-4. OPERATIONAL LABELS

VIII-4.1. General

The operational labels system has been introduced to enable the programs to process a logical environment.

An I/O transfer may be requested through an operational label which represents the actual I/O function. The correspondance between operational labels and physical devices is handled by the Monitor through assignment statements.

With such a system, the I/O programming is practically independent from the physical devices which are actually used and this ensures a total compatibility of the program with any configuration.

VIII-4.2. Definition

An operational label is a number which is generally a 4-character mnemonic code.

Operational labels are distributed into two classes : standard operational labels written M:XX with a predetermined function and user operational labels written U:FX whose functions are user-specified.

Decimal number	Mnemonic	Function	Mode
1	M:BI	Binary Input	Binary
2	M:BO	Binary Output	Binary
3	M:CI	Command Input	Alphanumeric
4	M:OC	Operator Console	Alphanumeric
5	M:EI	Element Input	Binary or Alphanumeric
6	M:EO	Element Output	Binary or Alphanumeric
7	M:LO	Listing Output	Alphanumeric
8	M:LL	Listing Log	Alphanumeric

Decimal number	Mnemonic	Function	Mode
9	M:DO	Diagnostic Output	Alphanumeric
10	M:SI	Source Input	Alphanumeric
11	M:SL	System's Library	Binary
12	M:UL	User's Library	Binary
13	M:SY	SYstem	Binary
14	M:EP	Executable Programs	Binary
15	M:GI	Go Input	Binary
16	M:GO	Go Output	Binary
17	U:F1		
18	U:F2		
19	U:F3		
20	U:F4		
21	U:F5		
22	U:F6		
23	U:F7		
24	U:F8		
25	U:F9		
26	U:FA		
27	U:FB		
28	U:FC		
29	U:FD		
30	U:FE		
31	U:FF		

VIII-4.3. Operational labels assignment

Every Monitor has for standard operational labels standard assignments defined at generation time. These assignments may be modified by monitor commands, except the following :

M:OC Typewriter

M:SY
M:EP
M:UL
M:SL
M:GI
M:GO

} System disk unit

User operational labels have no standard assignments and it is the user's responsibility to specify its own assignments.

- ASSIGN

This command is available under MOB-E, MTR, MTRD.

$$\text{\%AS[SIGN]/[F]},\ \left\{\begin{matrix}\text{M}\\\text{U}\end{matrix}\right\} : \text{O}_1\,\text{O}_2,\ \text{T}{:}t_1 t_2,\ \left[\text{C} : [\&]\, c_1 c_2\right],\left[\text{D} : [\&]\, d\right],\ \left[\left\{\begin{matrix}\text{BN}\\\text{AN}\end{matrix}\right\}\right]$$

F	Denotes an operational label reserved for Foreground use (MTR or MTRD)
M:O_1 O_2	Specifies the assigned standard operational label
U:O_1 O_2	Specifies the assigned user operational label (MTR or MTRD)
T:t_1 t_2	Specifies the type of device to which the operational label is assigned
C:c_1 c_2	Specifies the controller number within the specified type
D:d	Specifies the device number on the controller
BN	The corresponding file is binary
AN	The corresponding file is alphanumeric

Possible values for device type are :

T:NO	Cancel label. An input or output request with such a label is not performed.
T:TY	Typewriter (key-in and type-out)
T:PT	Console paper tape reader/punch
T:PR	High-speed paper tape reader
T:PP	High-speed paper tape punch
T:MC	Minicassette
T:LP	Line printer
T:CR	Card reader
T:CP	Card punch
T:DC	DIAD disk unit
T:9T	9-track magnetic tape unit
T:DM	DIAM disk-pack unit
T:VU	CRT display console
T:PL	Plotter

- STOL

This command is available under MTRD.

$$\text{\%STOL/}\left\{\begin{matrix}\text{M}\\\text{U}\end{matrix}\right\} :\text{O}_1\,\text{O}_2,\ \text{T}{:}t_1 t_2,\left[\text{C} : [\&]\, c_1 c_2\right],\left[\text{D} : [\&]\, d\right],\ \left[\left\{\begin{matrix}\text{BN}\\\text{AN}\end{matrix}\right\}\right]$$

This command provides for changing operational labels standard assignments.

Options have the same meaning as for %ASSIGN.

A %STOL command without any option restores the normal assignment of standard operational labels.

VIII-5. HANDLER UTILIZATION

VIII-5.1 . Typewriter handler (# 15 001)

■ General

- The typewriter handler controls type-ins and print-outs (T:TY) as well as paper tape read and punch (T:PT).

The various functions are determined by the selected operational label, by its assignment (typewriter or paper tape, input or output, binary or alphanumeric) and by the function byte (two elements of the I/O CB).

- A typewriter is 72 characters long.

- For alphanumeric input or output, the ISO 7-bit code is used (ASCII). ASCII ⟷ EBCDIC code conversions are performed by the handler, since the internal code is always EBCDIC.

- The paper tape punch drive motor in energized when the "tape on" character is detected (&12 code in ASCII). This code is not punched unless the punch is already energized at that time. The motor is shut down after punching the "tape off" character (&14 code in ASCII).

- The typewriter controller is busy either in typewriter mode or in paper tape mode, and for each of these modes, in input or output mode. The typewriter functions connot be performed simultaneously.

■ Functions

• Available commands

Hexadecimal code	Functions
00	Read
80	Write with format
90	Write without format

• Alphanumeric input : typewriter or paper tape code : &00

- The input is made one character at a time and each character is converted in EBCDIC before being transferred into user's buffer. The transfer is terminated when the specified number of characters has been reached.

However, the "carriage return" character (&0D code in EBCDIC or ASCII) always terminates the transfer after being actually transmitted to user.

- The "Paper feed" characters (&0A code in EBCDIC and &15 code in ASCII) are neither transmitted to the user nor counted in the total number of characters of the transfer.

- The "TAPE OFF" or "NULL" characters included in a record heading are not transmitted to the user and are not counted in the total number of characters of the transfer.

- When the %EOD group is detected in a record heading, the CB event byte (byte 0) is forced to 41 hexadecimal and represents an "end-of-file" mark.

• Alphanumeric output : typewriter or paper tape. Codes 80 and 90.

- The output is made one character at a time, each character being converted in ASCII at transfer time without any alteration of the user's buffer.

The transfer is terminated when the specified number of characters has been reached. No filtering is made.

- When the %EOD group is written in a record heading, the CB event byte (byte 0) is forced to 41 hexadecimal and represents an "end-of-file" mark.

- No character filtering is made.

- In paper tape mode, the transferred data are preceded by a "TAPE ON" character (not punched) and followed by a "TAPE OFF" character (punched).

If the user's buffer contains a "TAPE OFF", the handler restarts the punch (sends a "TAPE ON" which is not punched).

- Write with format. Code 80.

The user's buffer data transfer is preceded by two "paper feed" and "carriage return" characters for starting the line at the left margin.

- Write without format. Code 90.

No line adjustement character is provided. The format is entirely the user's responsibility.

- Binary input from paper tape. Code 00.

- The data are read one character at a time. The transfer is terminated when the specified number of characters has been reached.

- The "TAPE OFF" or "NULL" characters included in a record heading are not transmitted to the user and are not counted in the total number of characters specified for the transfer.

- When the %EOD group is detected in a record heading, the CB event byte (byte 0) is forced to 41 hexadecimal and represents an "end-of-file" mark.

- Binary output on paper tape. Code 80

- The data are punched one character at a time. The transfer is terminated when the specified number of characters has been reached.

- The user's buffer data transfer is preceded by a "TAPE ON" (not punched) and followed by a "TAPE OFF" (punched).

All bytes having a value of 14 or 94 hexadecimal (corresponding to a "TAPE OFF") are followed by a "TAPE ON" (not punched) for restarting the punch motor.

- When the %EOD group is detected in a record heading, the CB event byte (byte 0) in forced to 41 hexadecimal.

- Example :

```
ES        CDS
          RES       16                  TWB
BBINS     DATA      &0001     )
          DATA      &0002     }         BINARY OUTPUT BUFFER
          DATA      &0003     )
BALPS     TEXT      "MESSAGE"           Alphanumeric output buffer
          BND
BBINE     RES       8                   Binary input buffer
BALPE     RES       7                   Alphanumeric input buffer
          FIN

L1        LDS
          RES       2
CB1       DATA      0
          DATA,1    0                   Read command
          DATA,1    1 or DATA,1 M:BI                Binary input
          DATA      # BBINE
          DATA      16
CB2       DATA      0
          DATA,1    0                   Read command
          DATA,1    4 or DATA,1 M:OC                Alphanumeric input
          DATA      # BALPE
          DATA      14
```

```
CB3      DATA      0
         DATA,1    80                 Write command
         DATA,1    2 or DATA,1 M:BO                   Binary output
         DATA      # BBINS
         DATA      6
CB4      DATA      0
         DATA,1    80                 Write command
         DATA,1    7 or DATA,1 M:LO                   Alphanumeric output
         DATA      # BALPS
         DATA      7
         FIN

P1       LPS       L1
DEB      LEA       CB1
         CSV       M:IO               Input of a 16-byte binary record from M:BI
         CSV       M:WAIT
         LEA       CB2
         CSV       M:IO               Input of an alphanumeric record 14-characters max. from
         CSV       M:WAIT             typewriter (M:OC)
         LEA       CB3
         CSV       M:IO               Output of a binary record 0001, 0002, 0003 on M:BO
         CSV       M:WAIT
         LEA       CB4
         CSV       M:IO               Output of "MESSAGE" text on M:LO
         CSV       M:WAIT
         FIN       DEB

         END       P1
```

VIII-5.2. 300 char./sec. paper tape reader handler (# 15 062/60)

■ General

- This handler controls the binary or alphanumeric input from the paper tape (T:PR), each of these two functions being determined by the selected operational label, by its assignment and by the function byte (two elements of the I/O CB).

- For alphanumeric input, the ISO 7-bit code is used (ASCII). The ASCII ——► EBCDIC code conversion is performed by the handler, since the internal code is always EBCDIC.

■ Functions

The available command byte is &00 (read).

• Alphanumeric input. Code &00

- The input is made one character at a time and each character is converted in EBCDIC before being transferred into user's buffer. The transfer is terminated when the specified number of characters has been reached.

However, the "carriage return" character (&0D code in EBCDIC or ASCII) always terminates the transfer after being actyally transmitted to the user.

- The "paper feed" characters (&0A code in EBCDIC and &15 code in ASCII) are neither transmitted to the user nor counted in the total number of characters of the transfer.

The "TAPE OFF" or "FULL" characters included in a record heading are neither transmitted to the user nor counted in the total number of characters of the transfer.

- When the %EOD group is detected in a record heading, the CB event byte (byte 0) is forced to 41 hexadecimal and represents an "end-of-file" mark.

- Binary input. Code &00

- The data are read one character at a time. The transfer is terminated when the specified number of characters has been reached.

- The "TAPE OFF" or "NULL" characters included in a record heading are neither transmitted to the user nor counted in the total number of characters specified for the transfer.

- When the %EOD group is detected in a record heading, the event byte (CB byte 0) is forced to 41 hexadecimal representing an "end-of-file" mark.

VIII-5.3. 60 char./sec. paper tape punch handler (# 15 060)

■ General

- This handler controls the binary or alphanumeric output on paper tape (T:PP), each of these functions being determined by the selected operational label, by its assignment and by the function byte (two elements of the I/O CB).

- For alphanumeric output, the ISO 7-bit code is used (ASCII). The ASCII ⟶ EBCDIC code conversion is performed by the handler, since the internal code is always EBCDIC.

■ Functions

The available command byte is &80 (write)

- Alphanumeric output. Code &80.

- The output is made one character at a time, each character bieng converted in ASCII at transfer time without any modification of the user's buffer. The transfer is terminated when the specified number of characters has been reached. No filtering is made.

- When the %EOD group is written in a record heading, the CB event byte (byte 0) is forced to 41 hexadecimal representing an "end-of-file" mark.

- Binary output. Command &80.

- The output is made one character at a time and the transfer is terminated when the specified number of characters has been reached.

VIII-5.4. 300 char./sec. card reader handler (# 15 120)

■ General

- This handler controls the binary or alphanumeric input from the cards (T:CR), each of these functions being determined by the selected operational label, by its assignment and by the function byte (two elements of the I/O CB).

- For alphanumeric input, the EBCDIC card code is used on a one character per column basis (80 characters).

- For binary input, each card contains up to 120 bytes.

■ Functions

The available command byte is &00 (read).

- Alphanumeric input. Code 00.

- 80 characters are read.

- When the %EOD group is detected in a record heading, the CB event byte (byte 0) is forced to 41 hexadecimal representing an "end-of-file" mark.

- Binary input. Code 00.

- 120 bytes are read.

- When the %EOD group is detected in a record heading, the CB event byte (byte 0) is forced to 41 hexadecimal representing an "end-of-file" mark.

Note that, in this case, %EOD is a 4-character group, punched in the first four columns of the card (i.e. the first six bytes in binary reading mode).

Thus, the EOD mark is the same in binary and alphanumerical reading.

VIII-5.5. 200 l.p.m. printer handler (≠ 15 412)

■ General

This handler controls the alphanumerical outputs on the line printer (T:LP).

The print line is 132 characters long.

The alphanumeric output code is the ISO 7-bit standard code (ASCII). The user's buffer may be coded in ASCII or EBCDIC.

- EBCDIC-coded buffer. This is the most frequent case

The user must use an alphanumeric operational label (M:LO, etc.). The EBCDIC → ASCII conversion is performed by the handler in the buffer before the output transfer. The reciprocal ASCII → EBCDIC conversion will be performed by the handler at the end of the transfer. This double conversion has no effect on the final contents of the buffer.

- ASCII-coded buffer

The user must use a binary operational label (M:BI, etc.); no code conversion is performed.

■ Functions

- Available commands :

Hexadecimal code	Function
80	Write without format
90	Write with format

- Write without format mode

A paper feed is executed after the requested printing.

- Write with format

The first byte in the user's buffer will specify a paper positioning operation executed before the actual printing.

This skip code is conted in the number of transferred bytes.

The printing operation is normally followed by a paper feed, except when the format byte contains E0 or 60 (EBCDIC code).

Table of format byte coding (skip codes)

Code		
ASCII	EBCDIC	Function
5C	E0	Omit paper feed after printing
2D	60	Skip 0 line
7B	C0	Skip 1 line
41	C1	Skip 2 lines
42	C2	Skip 3 lines
C3	C3	Skip 3 lines
44	C4	Skip 4 lines
C5	C5	Skip 5 lines
C6	C6	Skip 6 lines
47	C7	Skip 7 lines
48	C8	Skip 8 lines
C9	C9	Skip 9 lines
60	CA	Skip 10 lines
E1	CB	Skip 11 lines
E2	CC	Skip 12 lines
63	CD	Skip 13 lines
E4	CE	Skip 14 lines
65	CF	Skip 15 lines
30	F0	Skip on channel 0 (botton of form)
B1	F1	Skip on channel 1
B2	F2	Skip on channel 2
33	F3	Skip on channel 3 (top of form)

VIII-5.6. Fast-access disk handler (# 15 200/1/2/3/4)

■ General

This handler controls the transfers between the core memory and the fast-access disk.

The associated CB must include an additional information : the disk address. This address specifies a number of disk-sectors (the first sector being sector 0).

0		Event byte
1		Indicators
2		Command or function
3		Operational label
4		Buffer address
5		
6		Number of bytes to be transferred
7		
8		Branch address on abnormal end
9		
10		Disk-address (sector number)
11		

The transfer is always initiated at the beginning of a sector. If the requested byte number is not a multiple of 256, the last sector which is only partly occupied, is filled with zeroes. Thus, the modified memory area is always a multiple of 256.

The transfer being executed one word at a time, the byte count in the CB must be even.

This handler considers the disk as an ordinary device; the MTRD handler, which is more powerful, performs the disk area management tasks.

■ Functions

- Available commands :

Hexadecimal code	Function
00	Read
10	Read and update disk-address
80	Write
90	Write and update disk-address

- Disk-address update :

For 10 and 90 command codes, the disk-address of the CB is incremented by the number of transferred sectors. This updating function may simplify the following transfers.

Appendix A — List of pseudo-instructions

The following representation conventions are used in the table below :

- Pseudo : Name of the pseudo-instruction
- Format : Operand format
- Type : Type of operand
  . LE : Label expression
  . RE : Reference expression
  . V : Value
  . C : Character string
  . S : Segment name
- Function : Definition of the pseudo-instruction function
- DATA : Authorized in a CDS or LDS
- PROG : Authorized in a LPS
- LAB : Assigns an address to a label

PSEUDO	FORMAT	Type	FUNCTION	DATA	PROG	LAB
CDS	[DUM]		Definition of the common data section or CDS. No code generated if DUM is present.			x
LDS	[DUM]		Definition of a local data segment or LDS. No code generated if DUM is present.			x
IDS (MITRAS II only)	[DUM]		Identification of an indirect access data segment. Any label specified in this segment is defined with respect to its starting address. No code generated if DUM is present.			x
LPS	NAME	S	Definition of an executable local program segment LPS and thus of a program section. NAME is the name of the associated LDS			x
END	SECTION NAME	S	Indication of the assembly module end SECTION NAME is the name of the first section to be executed.			

PSEUDO	FORMAT	Type	FUNCTION	DATA	PROG	LAB
FIN	[LABEL]	LE	Indication of a segment end. - LABEL may only be used in a LPS. In this case, it defines the starting address in the segment.	x	x	x
RES,1	VALUE	V	Reservation of a memory area. - ,1 indicates, when specified, that the reservation unit is the byte, otherwise it is the word. - VALUE defines ghe length of the memory area.	x	x	x
BND			This pseudo-instructions advances the location counter to a word boundary.	x	x	
EQU	Predefined expression $	LE	Definition of an equivalence between the symbol in label field and the quantity defined in operand field.	x	x	x
GOTO,n (MITRAS II only)	Lab1, Lab2, . . . , Labn	LE	Specification of a conditional branch at assembly time. - ,n is a value pointing at a label in operand field at which the branch is made.	x	x	
DO (MITRAS II only)	VALUE %		Specification of an iterative assembly of an instruction. - VALUE = number of iterations - % : at every iteration cycle, this character takes the value of the iteration counter.	x		
DATA,1	[#] Expression 1 [,[#] Expression 2]...	LE V	Data generation statement - ,1 if present in command field the locations are expressed in bytes, otherwise in words. - # This symbol denotes that the following expression must be left relative to G upon loading in Master mode.	x		
GEN, Area list (MITRAS II only)	Expression list	V	Value generation statement. - Area list is a sequence of values each defining the length of an area to be generated. - Expression list is a sequence of expressions defining the contents of the declared areas.	x	x	x

PSEUDO	FORMAT	Type	FUNCTION	DATA	PROG	LAB
TEXT	"Character string"	C	Generation of a character string. - Character string is made up of alphanumeric characters.	x		x
DEF	Label [, Label]	LE	Declaration of labels as external definitions.	x	x	
REF	[#] Label [, [#] Label]	LE	Declaration of labels as external references. - # indicates that the label belongs to the CDS.	x	x	
BASE	[Label]		This statement requests that all address generated at assembly time be relative to an address specified in operand field.		x	
PAGE (MITRAS II only)			The assembly listing is printed on the next page if the output device is the printer.	x		

Appendix B – List of instructions

Instructions are arranged in alphabetic order

Instr.	Class	Code according to addressing mode							
		P	DL	IL	ILX	DG	IGX	RP	RM
ADD	0	25	05	65	A5	45	85	-	-
ADM	0'	-	17	77	B7	57	97	-	-
AND	0	29	09	69	A9	49	89	-	-
BAN	2	-	–	D4	-	–	DC	C4	CC
BAZ	2	-	–	D5	-	–	DD	C5	CD
BCF	2	-	–	D3	-	–	DB	C3	CB
BCT	2	-	-	D0	-	–	D8	C0	C8
BOF	2	-	–	D6	-	–	DE	C6	CE
BOT	2	-	–	D2	-	-	DA	C2	CA
BRU	2	-	-	D7	-	–	DF	C7	CF
BRX	2	-	–	D1	-	-	D9	C1	C9
• CLM	1	F400	-	-	-	-	-	-	-
CLS	1	F8	38	-	-	-	-	E8 o	-
CMP	0	2B	0B	6B	AB	4B	8B	-	-
* CPS	0	2A	0A	6A	AA	4A	8A	-	-
CSV	1	F7	37	-	-	-	-	E7 o	-
DCL	1	F6	36	-	-	-	-	E6 o	-
DCX	1	F3	33	-	-	-	-	E3 o	-
• DIT	1	F401	-	-	-	-	-	-	-
* DIV	0	28	08	68	A8	48	88	-	-
DLD	0'	-	10	70	B0	50	90	–	-
DST	0'	-	16	76	B6	56	96	—	-
EOR	0	23	03	63	A3	43	83	–	-
* FAD	0'	-	1A	7A	BA	5A	9A	–	-
* FDV	0'	-	10	7D	BD	5D	9D	-	-
* FMU	0'	-	1C	7C	BC	5C	9C	-	-
* FSU	0'	-	1B	7B	BB	5B	9B	–	-
ICL	1	F5	35	-	-	–	—	E5 o	-
ICX	1	F2	32	-	-	–	–	E2 o	-
IOR	0	27	07	67	A7	47	87	–	-

Instr.	Class	Code according to addressing mode							
		P	DL	IL	ILX	DG	IGX	RP	RM
LBL	0	2D	0D	6D	AD	4D	8D	-	-
LBR	0	2E	0E	6E	AE	4E	8E	-	-
LBX	0	2F	0F	6F	AF	4F	8F	-	-
LDA	0	20	00	60	A0	40	80	-	-
LDE	0	21	01	61	A1	41	81	-	-
*•LDP	1	FB	3B	-	-	-	-	-	-
• LDR	1	F9	39	-	-	-	-	-	-
LDX	0	22	02	62	A2	42	82	-	-
LEA	0	-	04	64	A4	44	84	-	-
MUL	0	2C	0C	6C	AC	4C	8C	-	-
* MVS	0'	-	1F	7F	BF	5F	9F	-	-
• RD	1	F402	-	-	-	-	-	-	-
RSV	1	F1	-	-	-	-	-	-	-
RTS	1	F100	-	-	-	-	-	-	-
SBL	0'	-	14	74	B4	54	94	-	-
SBR	0'	-	15	75	B5	55	95	-	-
* SHC	1	FC	3C	-	-	-	-	o EC	-
SHR	1	F0	30	-	-	-	-	o E0	-
SPA	0'	-	18	78	B8	58	98	-	-
SRG	1	F1	31	-	-	-	-	o E1	-
STA	0'	-	11	71	B1	51	91	-	-
STE	0'	-	12	72	B2	52	92	-	-
• STM	1	F408	-	-	-	-	-	-	-
• STR	1	FA	3A	-	-	-	-	o EA	-
STS	0'	-	19	79	B9	59	99	-	-
STX	0'	-	13	73	B3	53	93	-	-
SUB	0	26	06	66	A6	46	86	-	-
* TES	1	FD	3D	-	-	-	-	o ED	-
* TRS	0'	-	1E	7E	BE	5E	9E	-	-
• WD	1	F403	-	-	-	-	-	-	-

Note :

•	:	Priviledged instruction
*	:	Optional instruction
o	:	PX addressing mode

Instructions are arranged in ascending hexadecimal order

Code	Instruc.	Class	Adr.	Code	Instruc.	Class	Adr.	Code	Instruc.	Class	Adr.	Code	Instruc.	Class	Adr.
00	LDA	0	DL	20	LDA	0	P	40	LDA	0	DG	60	LDA	0	IL
01	LDE	0	-	21	LDE	0	-	41	LDE	0	-	61	LDE	0	-
02	LDX	0	-	22	LDX	0	-	42	LDX	0	-	62	LDX	0	-
03	EOR	0	-	23	EOR	0	-	43	EOR	0	-	63	EOR	0	-
04	LEA	0	-	24	LEA	0	-	44	LEA	0	-	64	LEA	0	-
05	ADD	0	-	25	ADD	0	-	45	ADD	0	-	65	ADD	0	-
06	SUB	0	-	26	SUB	0	-	46	SUB	0	-	66	SUB	0	-
07	IOR	0	-	27	IOR	0	-	47	IOR	0	-	67	IOR	0	-
08	* DIV	0	-	28	* DIV	0	-	48	* DIV	0	-	68	* DIV	0	-
09	AND	0	-	29	AND	0	-	49	AND	0	-	69	AND	0	-
0A	* CPS	0	-	2A	* CPS	0	-	4A	* CPS	0	-	6A	* CPS	0	-
0B	CMP	0	-	2B	CMP	0	-	4B	CMP	0	-	6B	CMP	0	-
0C	MUL	0	-	2C	MUL	0	-	4C	MUL	0	-	6C	MUL	0	-
0D	LBL	0	-	2D	LBL	0	-	4D	LBL	0	-	6D	LBL	0	-
0E	LBR	0	-	2E	LBR	0	-	4E	LBR	0	-	6E	LBR	0	-
0F	LBX	0	-	2F	LBX	0	-	4F	LBX	0	-	6F	LBX	0	-
10	DLD	0'	-	30	SHR	1	DL	50	DLD	0'	-	70	DLD	0'	-
11	STA	0'	-	31	SRG	1	-	51	STA	0'	-	71	STA	0'	-
12	STE	0'	-	32	ICX	1	-	52	STE	0'	-	72	STE	0'	-
13	STX	0'	-	33	DCX	1	-	53	STX	0'	-	73	STX	0'	-
14	SBL	0'	-	34				54	SBL	0'	-	74	SBL	0'	-
15	SBR	0'	-	35	ICL	1	-	55	SBR	0'	-	75	SBR	0'	-
16	DST	0'	-	36	DCL	1	-	56	DST	0'	-	76	DST	0'	-
17	ADM	0'	-	37	CSV	1	-	57	ADM	0'	-	77	ADM	0'	-
18	SPA	0'	-	38	CLS	1	-	58	SPA	0'	-	78	SPA	0'	-
19	STS	0'	-	39	• LDR	1	-	59	STS	0'	-	79	STS	0'	-
1A	* FAD	0'	-	3A	• STR	1	-	5A	* FAD	0'	-	7A	* FAD	0'	-
1B	* FSU	0'	-	3B	*•LDP	1	-	5B	* FSU	0'	-	7B	* FSU	0'	-
1C	* FMU	0'	-	3C	* SHC	1	-	5C	* FMU	0'	-	7C	* FMU	0'	-
1D	* FDV	0'	-	3D	* TES	1	-	5D	* FDV	0'	-	7D	* FDV	0'	-
1E	* TRS	0'	-	3E				5E	* TRS	0'	-	7E	* TRS	0'	-
1F	* MVS	0'	-	3F				5F	* MVS	0'	-	7F	* MVS	0'	-

Note :

* : Optional instruction

• : Priviledged instruction

Code	Instruc.	Class	Adr.	Code	Instruc.	Class	Adr.	Code	Instruc.	Class	Adr.	Code	Instruc.	Class	Adr.
80	LDA	0	IGX	A0	LDA	0	ILX	C0	BCT	2	RP	E0	SHR	1	PX
81	LDE	0	-	A1	LDE	0	-	C1	BRX	2	-	E1	SRG	1	-
82	LDX	0	-	A2	LDX	0	-	C2	BOT	2	-	E2	ICX	1	-
83	EOR	0	-	A3	EOR	0	-	C3	BCF	2	-	E3	DCX	1	-
84	LEA	0	-	A4	LEA	0	-	C4	BAN	2	-	E4			
85	ADD	0	-	A5	ADD	0	-	C5	BAZ	2	-	E5	ICL	1	-
86	SUB	0	-	A6	SUB	0	-	C6	BOF	2	-	E6	DCL	1	-
87	IOR	0	-	A7	IOR	0	-	C7	BRU	2	-	E7	CSV	1	-
88	*DIV	0	-	A8	* DIV	0	-	C8	BCT	2	RM	E8	CLS	1	-
89	AND	0	-	A9	AND	0	-	C9	BRX	2	-	E9	• LDR	1	-
8A	*CPS	0	-	AA	* CPS	0	-	CA	BOT	2	-	EA	• STR	1	-
8B	CMP	0	-	AB	CMP	0	-	CB	BCF	2	-	EB	*•LDP	1	-
8C	MUL	0	-	AC	MUL	0	-	CC	BAN	2	-	EC	* SHC	1	-
8D	LBL	0	-	AB	LBL	0	-	CD	BAZ	2	-	ED	* TES	1	-
8E	LBR	0	-	AE	LBR	0	-	CE	BOF	2	-	EE			
8F	LBX	0	-	AF	LBX	0	-	CF	BRU	2	-	EF			
90	DLD	0'	-	B0	DLD	0'	-	D0	BCT	2	IL	F0	SHR	1	P
91	STA	0'	-	B1	STA	0'	-	D1	BRX	2	-	F1	SRG	1	-
92	STE	0'	-	B2	STE	0'	-	D2	BOT	2	-	F2	ICX	1	-
93	STX	0'	-	B3	STX	0'	-	D3	BCF	2	-	F3	DCX	1	-
94	SBL	0'	-	B4	SBL	0'	-	D4	BAN	2	-	F4	• SYS[1]	1	-
95	SBR	0'	-	B5	SBR	0'	-	D5	BAZ	2	-	F5	ICL	1	-
96	DST	0'	-	B6	DST	0'	-	D6	BOF	2	-	F6	DCL	1	-
97	ADM	0'	-	B7	ADM	0'	-	D7	BRU	2	-	F7	CSV	1	-
98	SPA	0'	-	B8	SPA	0'	-	D8	BCT	2	IG	F8	CLS	1	-
99	STS	0'	-	B9	STS	0'	-	D9	BRX	2	-	F9	• LDR	1	-
9A	*FAD	0'	-	BA	* FAD	0'	-	DA	BOT	2	-	FA	• STR	1	-
9B	*FSU	0'	-	BB	* FSU	0'	-	DB	BCF	2	-	FB	*•LDP	1	-
9C	*FMU	0'	-	BC	* FMU	0'	-	DC	BAN	2	-	FC	* SHC	1	-
9D	*FDV	0'	-	BD	* FDV	0'	-	DD	BAZ	2	-	FD	* TES	1	-
9E	*TRS	0'	-	BE	* TRS	0'	-	DE	BOF	2	-	FE			
9F	*MVS	0'	-	BF	* MSV	0'	-	DF	BRU	2	-	FF			

Note :

* : Optional instruction

• : Priviledged instruction

[1] SYS : this mnemonic is not recognized by the Assembler

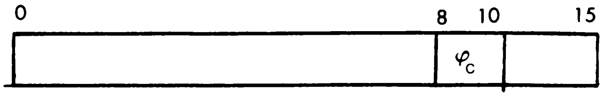
SRG instruction

Instruction	Function	Code
AAE	$(A) \cap (E) \longrightarrow (A)$	F118
ACE	$(E) + C \longrightarrow (E)$	F10E
AEE	$(A) \oplus (E) \longrightarrow (A)$	F112
AIE	$(A) \cup (E) \longrightarrow (A)$	F116
CCA	$(\overline{A}) \longrightarrow (A)$	F110
CCE	$(\overline{E}) \longrightarrow (E)$	F10A
CHX	Arithmetic shift 1 step left	F11E
CNA	$-A \longrightarrow (A)$	F11C
CNX	$-X \longrightarrow (X)$	F114
LNE	$-1 \longrightarrow (E)$	F11A
RTS	Return section	F100
RSV	Return supervisor	F10C
XAA	$A_{0-7} \longleftrightarrow A_{8-15}$	F108
XAE	$(A) \longleftrightarrow (E)$	F102
XAX	$(A) \longleftrightarrow (X)$	F104
XEX	$(E) \longleftrightarrow (X)$	F106

SYS instruction(1)

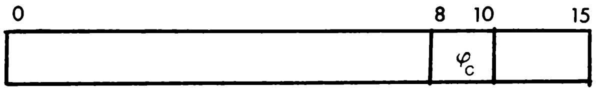
Function	Class	Instruction	Code
Clear IT mask	1	CLM	F400
De-activate IT	1	DIT	F401
Read direct	1	RD	F402
Write direct	1	WD	F403
Set IT mask	1	STM	F408

SHR instruction



Function	φ_c	Code
Shift Logical Left Single	0	SLLS
Shift Circular Right Single	1	SRCS
Shift Arithmetic Right Double	2	SAD
Shift Circular Left Double	3	SLCD
Shift Circular Left Single	4	SLCS
Shift Arithmetic Right Single	5	SAS
Shift Logical Right Single	6	SRLS
Shift Circular Right Double	7	SRCD

SHC instruction



Function	φ_c	Code
Shift Logical Left Double	0	SLLD
Compute parity	2	PTY
Shift Logical Right Double	4	SRLD
Normalize E,A	6	NLZ
De-activate fast IT	1 3 5 7	DITR

(1) SYS : this mnemonic is not recognized by the Assembler

Addressing :

Class 0		Class 0'		Class 1		Class 2	
P	Y=D	DL	Y=D+(L)	P	N=D	RP	Y=(P)+2D
DL	Y=D+(L)	IL	Y=(D+(L))+G'	PX	N=D+(X)	RM	Y=(P)-2D
IL	Y=(D+(LL)+G'	ILX	Y=(D+(L))+G'+(X)	DL	N=(D+(L))	DL	Y=(D+(L))+G'
ILX	Y=(D+(L))+G'+(X)	DG	Y=D+(G)			DG	Y=(D+(G))+G'
DG	Y=D+(G)	IGX	Y=(D+(G))+(G)+(X)				
IGX	Y=(D+(G))+(G)+(X)						

Appendix D - Assembler operation

I - ASSEMBLY LISTING

I-1. FORMAT OF AN ASSEMBLY LISTING LINE

The general line format of an assembly listing is illustrated below.

A line contains the following informations :

- Up to two error flags.
- A decimal line number.
- The current hexadecimal contents of the location counter.
- The object code (hexadecimal) generated by the Assembler.
- A indication of a forward reference if the argument address is anticipated.
- The source line image.

The lines which are skipped under control of a GOTO are not annotated except when they contain an invalid operation code.

```
1      4          9         14          20                          72
       |          |          |
E E D D D D * L L L L * X X X XA * * S S S .......... S S S
```

*	:	Space
EE	:	Error flags
DDDD	:	Decimal number of source line
LLLL	:	Hexadecimal value of location counter
XXXX or **XX	:	Hexadecimal value of a word or byte generated at LLLL location
A	:	When present, the reference is forward
SS..SS	:	Source text line

Format of an assembly listing line

I-2. OBJECT LISTING

In addition to the hexadecimal representation of the object code and the edition of the corresponding source text, the Assembler provides :

- A list of defined and/or referenced segments names, except under MITRAS I.
- A table of satisfied or unsatisfied labels per section (optional).

I-2. OBJECT LISTING

In addition to the hexadecimal representation of the object code and the edition of the corresponding source text, the Assembler provides :

- A list of defined and/or referenced segments names, except under MITRAS 1.

- A table of satisfied or unsatisfied labels per section (optional).

II - OPERATING OPTIONS

II-1. MITRAS I

■ Command format

%ASS 1/options indicated on console switches (M:OC).

■ Options

Assembly listing :

- requested : console switch 15 reset
- omitted : console switch 15 set

Relocatable binary :

- requested : console switch 14 reset
- omitted : console switch 14 set

Output of an additional %EOD after the RB module :

- requested : console switch 13 reset
- omitted : console switch 13 set

II-2. MITRAS II

■ Command format

$$\left\{\begin{array}{l}\text{\%ASS2}\\ \text{\%CALL/ASS2}\end{array}\right\} \text{/[SI][,BO] [,LO][,LL]}$$

%ASS2/ Output on M:OC by the processor when loaded and started by %LOAD and %RUN.

%CALL/ASS2/ Output on M:OC by the processor when started without console interrupt. In this case, options are given on M:OC.

Under control of the linking module, the command and its options are entered via M:CI.

■ Options

- SI : Source file read on M:SI. When this option is omitted, no other option should be present, and all options are implicit.

- BO : Relocatable binary output requested.

- LO : Assembly listing output requested.

- LL : Label table printing requested, together with severity level and number of incorrect lines.

- Option for assembly end

When MITRAS II is loaded and started through a %LOAD and a %RUN, before returning to the beginning when the assembly is over, the following message is printed on M:OC :

%%EOD?

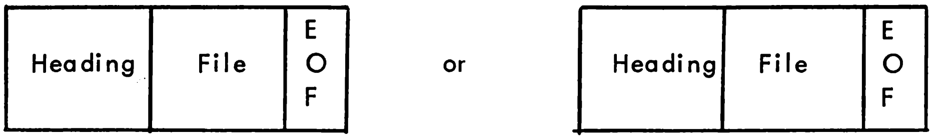
Meaning that the optional output of an additional "end-of-file" mark after the RB has been allocated.

The operator's answer on M:OC may be :

- OUI if the additional "end-of-file" mark is requested.
- NON if the additional "end-of-file" mark is not requested.

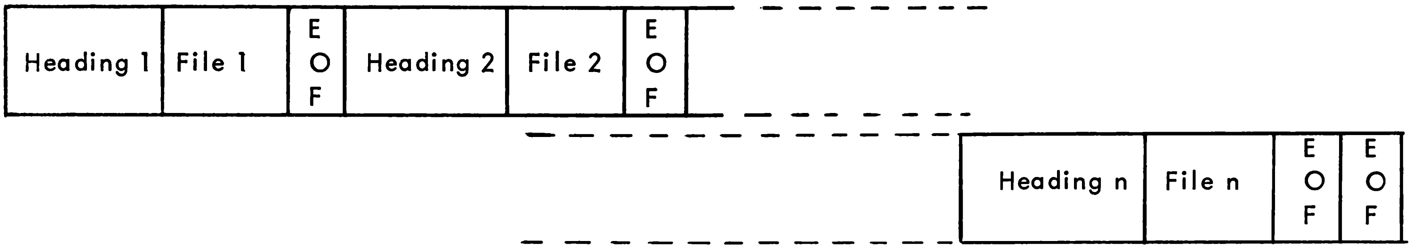
- Additional "end-of-file"

When a program module has been assembled, the relocatable binary output may have one of the following format :

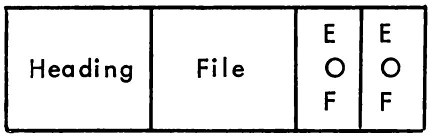


- where E.O.F. stands for "end-of-file".

This is necessary for establishing the RB input file of the Linkage Editor. This file is organized as follows :



The Linkage Editor may then output a RMI file with the following format :



III-2. TABLE FORMAT

III-2.1. Local label table

A table containing the local labels is printed after every LPS, before FIN pseudo-instruction. It has the following format :

LABE LLLL Z

- LABE : Label name.
- LLLL : Corresponding value of location counter or label reference number if it has not yet defined.
- Z : D = LDS label
  P = LPS label
  A = Label value is absolute
  X = Label not yet defined
  R = Label declared in a REF pseudo-instruction

III-2.2. Common label table

When the module assembly is completed, after END pseudo-instruction, a table of common labels is printed.

It has the following format :

LABE LLLL Z

- LABE : Label name
- LLLL : Corresponding value of location counter or label reference number if it has not yet been defined.
- Z : C = CDS label
  D = LDS label (declared in a DEF of the CDS and defined in a LDS).
  P = LPS label (declared in a DEF of the CDS and defined in a LPS).
  A = Label value is absolute
  X = Label not yet defined
  R = Label declared in a REF

III-2.3. Severity level

After the common label table, MITRAS I outputs the following message :

NSV $\left\{\begin{matrix}0\\1\\2\\3\end{matrix}\right\}$ NB ERR XXX

Indicating the highest error level encountered during assembly and the number of incorrect lines.

III-3. ERROR PROCESSING

III-3.1. Definition of assy error level

Four error levels will be considered :

- LEVEL 0 : No error or presumed error during assembly.
- LEVEL 1 : At least one presumed error. Linkage edition and program execution are possible.
- LEVEL 2 : Confirmed error for which the assembler has selected an option. Link edition is possible, but the edited program cannot be correctly executed in most cases without %MODIFY cards.
- LEVEL 3 : Major error. The assembly continues but linkage edition will be impossible. The source program must be corrected and re-assembled.

As a general rule, any level-3 error impedes link edition.

III-3.2. List of errors detected during assembly

The incorrect source line is printed with 1 or 2 leading characters which identify the type of error.

Two error flags are set, at most.

D : Double definition of a label. The first definition is assumed right.

E : Syntactic or semantic error in operand field of an instruction or pseudo-instruction.

- For an instruction, the operand is ignored, the corresponding zone being cleared.
- For a segmentation pseudo, the operand field of the source card is ignored.

- For an EQU, the label becomes equivalent to an absolute zero value.

- For GOTO and DO pseudos, the operand field is ignored.

- For a TEXT pseudo, the operand field is ignored after the error (downstream).

- For a DATA pseudo, if the error affects the definition of the selected resolution, a DATA 0 will be generated by the Assembler.

If the error affects the operand field, the latter is ignored after the error (downstream).

- For a GEN pseudo, an error in the partitioning of the selected areas will cause a zero-word to be generated.

An error in the operand field stops the scanning of the card. However, everything upstream the error is normally taken into consideration.

F : Incorrect expression in operand part of RES pseudo : a zero-word is generated.

G : - Negative variable address.
- Address > 255
- Calculated displacement exceeding the available number of bits.
In all cases, incorrect value replaced by zero.

I : - Command unknown to the Assembler : replaced by RES 1.
- Instruction or pseudo forbidden in this segment : the command is ignored.
- No asterisk between operand and comment field.

J : - Forbidden character in this label : label ignored.
- Mandatory label absent for a segmentation pseudo in the RB : label replaced by zero.
- Label forbidden in this pseudo (BASE, BND, FIN, ...) : label ignored.
- No label for an EQU : card ignored.

N : - Label table overflow.

O : - For a DATA, the result of an expression overflows the assigned partitions : result truncated.
- Discrepancy between the number of expressions and the number of partitions in a GEN.

P : - Location counter overflow (>65 535).

R : - A %EOD group met before the END card.

T : - Multiple definition of a segment.

X : - At the time a word is generated, the location counter value is odd. A zero-byte is first generated to increment the counter to a word boundary. If the source line includes a label, the corresponding value taken is the initial value of the location counter and not the corrected value.

V : The currently assembled source line contains a forward branch whose reach cannot be checked because the forward branch control table is full.

W : The currently assembled source line has a label to which one or several forward branches have been made.
Among these branches, at least one has a reach exceeding 255.

III-3.3. Distribution of error types on the levels

- Level 0
- Level 1 : V-C
- Level 2 : D-E-F-G-I-J-O-T-W-X
- Level 3 : I-N-P-R

Appendix C — Addressing modes

1) Class 0 addressing

Instructions of this type may only address any data in the local or common segments. Class 0' Prohibits parameter or immediate addressing.

This class includes :

- Load and store instructions
- Arithmetic instructions (fixed- or floating-point)
- Logical operations
- Byte string instructions
- Comparison instructions

Six addressing modes are allowed :

Mode	Assembly language	Addressed data	Addressing function
Direct, Local DL	IDENT	Byte, word or double-word located in the first 256 bytes of the local segment.	Y=(L)+D
Indirect, Local IL	@IDENT	Byte, word or double-word located anywhere and pointed at through the local segment.	Y=G'+((L)+D)
Indirect, Local, Indexed ILX	@IDENT,X	Element of a byte, word or double-word array located anywhere and pointed at through the local segment.	Y=G'+((L)+D)+(X)
Direct, General DG	# IDENT	Byte, word or double-word located in the first 256 bytes of the common segment.	Y=(G)+D
Indirect, General, Indexed IGX	@# IDENT,X	Element of an array pointed at through the common segment.	Y=(G)+((G)+D)+(X)
Parameter or immediate	=OPERAND	A 1-byte operand is specified in the instruction. This byte may be extended on the left by 8 leading zeroes, if required.	(Y) = D Y = (P)

2) Class 1 addressing

These instructions are :

- Either without operand : register swapping, end of section, etc.,

- Or instructions whose operand is generally known (possibly through an unknown modifier) at program writing time : shift, index, increment, etc.

This class includes :

- Shift instructions

- Base instructions

- Section or supervisor calls

- I/O instructions

- Inter-register operations

- Interrupt-initiated and interrupt mask instructions.

Three addressing modes are allowed :

Mode	Assembly language	Operand	Addressing function
Parameter or immediate P	=PARAM	Operand defined by displacement value.	(Y) = D Y = (P)
Parameter, Indexed PX	=PARAM,X	Operand defined by value plus X-register contents.	(Y)=D+(X) Y = (P)
Direct, Local DL	IDENT	Operand located in the first 256 bytes of the local segment.	Y=(L)+D

3) Class 2 addressing

This class includes conditional and unconditional branch instructions.

Normally the instructions which are pointed at by a branch instruction belong to the same section.

Four addressing modes are allowed :

Mode	Assembly language	Branch instruction	Addressing function
Relative downstream (plus) RP	LABEL	Any instruction within 512 bytes downstream	Y=(P)+2D
Relative upstream (minus) RM	LABEL	Any instruction within 512 bytes upstream	Y=(P)-2D
Indirect, Local IL	@ LABEL	Any instruction pointed at through the local segment.	Y=G'+((L)+D)
Indirect, General iG	@ #LABEL	Any instruction pointed at through the common segment.	Y=G'+((G)+D)

cii COMPAGNIE INTERNATIONALE POUR L'INFORMATIQUE

RC : 669805764 B
R.C. Versailles – SIRENE : 669805764

Siège Social
Direction Commerciale
Division des Petits Ordinateurs
et des Applications Spécialisées
Direction Après-Vente
68, Route de Versailles
78430 Louveciennes
Tél. 954 9080

Direction Générale
Institut de Formation
Parc de Rocquencourt
78150 Le Chesnay
Tél. 954 4400

Centre de Vélizy
Division Militaire Spatiale
et Aéronautique
Direction Après-Vente
10 - 12 avenue de l'Europe
78140 Vélizy
Tél. 946 9670

Centre des Clayes-sous-bois
Avenue Jean Jaurès
78340 Les Clayes-sous-bois
Tél. 055 8000

Centre de Toulouse
Avenue du Général Eisenhower
31023 Toulouse
Tél. (61) 40 1140.

DÉLÉGATIONS RÉGIONALES

RHONE-ALPES
177, rue Garibaldi Immeuble M + M
69003 Lyon
Tél. (78) 62 9065

Tour Mont Blanc
15, bd. du Maréchal Leclerc
38000 Grenoble
Tél. (76) 44 9922

18-20 av. du Maréchal Foch
21000 Dijon
Tél. (80) 32 2047

OUEST
3 Place du Colombier
35000 Rennes
Tél. (99) 30 8454

CENTRE-OUEST
9, place Rouget de Lisle
37000 Tours
Tél. (47) 20 2209

MIDI-PYRÉNÉES
Av. du Général Eisenhower
31023 Toulouse
Tél. (61) 40 3563

SUD-EST
433, rue Paradis
13008 Marseille
Tél. (91) 77 0994

AQUITAINE
353, bd du Président Wilson
33200 Bordeaux
Tél. (56) 08 6363

EST
25, avenue Robert Schuman
57000 Metz
Tél. (87) 68 4921

15, rue des Francs Bourgeois
67000 Strasbourg
Tél (88) 32 1103

NORD
13, boulevard de la Liberté
59000 Lille
Tél. (20) 57 7353