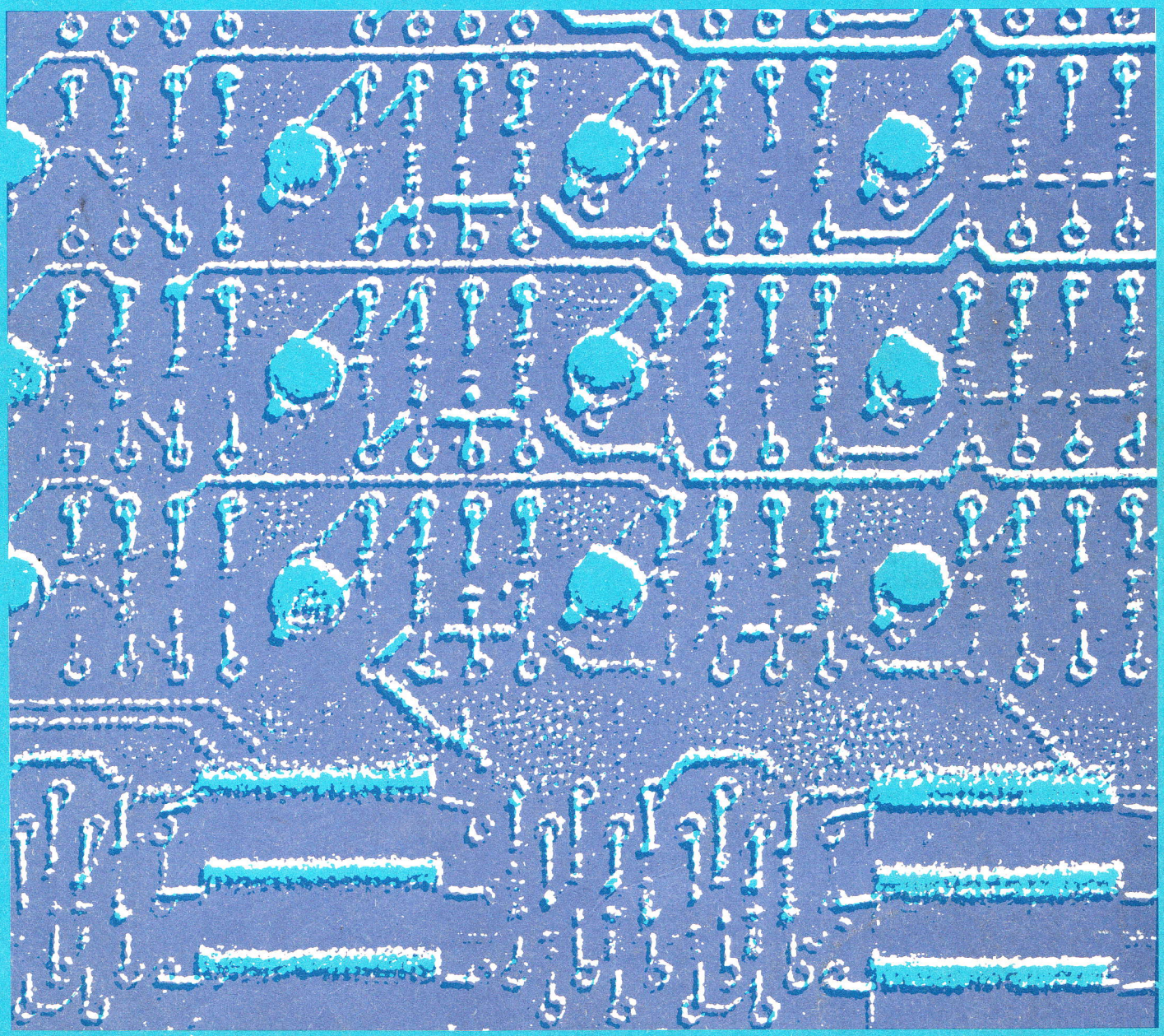
P800M Programmer's Guide 3

Volume III: Software Processors



PHILIPS Data Systems

PHILIPS

P800M Programmer's Guide 3
Volume III: Software Processors

A Publication of:
Philips Data Systems,
Department SSS, T & D,
P.O. Box 245, 7300 AE Apeldoorn, The Netherlands.

Publication Number 5122 991 28384

May 1983

Printed in the Netherlands.

This volume describes the processors running under the control of the Multi Application Monitor. These processors are:

- Assembler
- Overlay Linkage Editor
  Update Processor
- Librarian
- Debugging Package
- Transaction-oriented Disc File Management
- Sort Processor.

Part 1 of the manual describes the Assembly Language. A description of FORTRAN may be found in the FORTRAN Reference Manual (5122 991 1140x), and of RTL/2 in a number of manuals (e.g. RTL/2 Language Specification, 5122 011 2895x).

This manual should be used with the other MAS Manuals:

- Vol. I: Multi Application Monitor
- Vol. II: Instruction Set
- Vol. IV: Trouble Shooting Guide.

While every care has been taken in the preparation of this book, some errors may remain. Should the reader find an error or omission, or have any other comment to make, he is invited to contact:

SSS, Training and Documentation,

at the address on the opposite page. A form is provided at the end of this book, for the user's convenience.

PART 1 ASSEMBLY LANGUAGE

A module or program written in the Assembly Language consists of a series of statements. There are two types of statements:

- Instructions

  The program instructions normally form the bulk of the program, and are the actual work tools by which data may be input, processed and output. Each instruction in the P800M instruction set is fully described in Volume II.

- Directives

  The directives are used to guide the actual assembling process and to structure the program according to the programmer's needs. The directives are described in Chapter 5 of this part.

After the program is written, it is input to the Assembler and then to the Overlay Linkage Editor to convert to load module format and to fill in external references.

The following symbols are used to define the syntax of the P800M Assembly Language and Assembly Directives.

::=	means "is defined as".
_	space (i.e. one or more significant spaces)
[ ]	the syntactic items between these square brackets may be omitted.
{a | b}	means a or b
...	ellipsis indicates repetition of the last syntactic item.

The following list contains the definition of all items used:

```
<absolute symbol>                ::= characters representing a value

<address expression>             ::= <expression>

<character>                      ::= {<letter> | <digit> | <delimiter>}

<character constant>             ::= `<character>[<character>]'

<character string>               ::= `<character>[<character>] ...'

<comments>                       ::= {characters (in comments field) |
                                     * characters (on new line)}

<common field definition>        ::= <common field name>[<common field length>]

<common field definition list>   ::= <common field definition>,
                                     <common field definition>, ...

<common field length>            ::= <predefined expression>

<common field name>              ::= <label>

<constant>                       ::= {<decimal constant> |
                                     <hexadecimal constant> |
                                     <character constant>}

(cnd)                            ::= condition value or condition mnemonic
                                     (see condition table with the commands
                                     AB, ABR, ABI, RB, RF)

<data expression>                ::= {<expression> | <character string>}
                                      <data expression>

<decimal constant>               ::= <digit string>
                                       range -32768 to +32767 in 16 bits
                                       range 0 to 255         in  8 bits
```

```
<delimiter>                          ::= {+ plus |
                                         - minus |
                                         * asterisk |
                                         = equals |
                                         ' apostrophe |
                                         , comma |
                                           blank |
                                         / slash |
                                         ( left parenthesis |
                                         ) right parenthesis |
                                         . period |
                                         : colon}

<digit>                              ::= 0 to 9 inclusive

<digit string>                       ::= <digit>[<digit>] ...

<entry name>                         ::= <label> within reference module

<expression>                         ::=  {<predefined expression> |

            <external name>{+ | -}<absolute symbol value>{+ | -}<value>}

<external name>                      ::= <label> defined in another module

<field definition>                   ::=
            <field length definition>{= | :}<field value definition>

<field length definition>            ::= number of bits (1-16)

<field number>                       ::= <decimal constant>

<field number list>                  ::= <field number>[,<field number>] ...

=<field value definition>            ::= <value> to be placed in field

:<field value definition>            ::= address of word, relative to FORM

<hexadecimal constant>               ::= {X'<hexa digit>[<hexa digit>]' |
                                         /<hexa digit>[<hexa digit>]}
                                          range 0 - /FFFF in 16 bits
                                          range 0 - /FF   in 8 bits.

<hexa digit>                         ::= {<digit> | A to F inclusive}

<integer>                            ::= 0 - /FFFF or -32767 < i < 32767

<internal symbol>                    ::= <label>

<k>                                  ::= short constant

L                                    ::= long constant indicator

<label>                  ::= <letter>{<letter> | <digit> | <delimiter>} ...
                                                Maximum length 6 characters.

<letter>                             ::= A to Z inclusive
```

```
<lk>                                ::= <long constant>

<long constant>                     ::= <expression>

<m>                                 ::= <address expression>

<mnemonic>                          ::= characters representing instruction or
                                        directive

<module name>                       ::= <label>

<operand>                           ::= <expression>

<operation code>                    ::= <mnemonic>

<predefined absolute expression>  ::=
{{+ | -}<absolute symbol value>[{+ | -}<absolute symbol value>[{+ | -}<value>]]
    | <relocatable symbol>-<relocatable symbol>[{+ | -}<value>]}

<predefined expression>             ::=
    {<predefined absolute expression> |
    <relocatable symbol>[{+ | -}<absolute symbol value>[{+ | -}<value>]]}

<register expression>               ::= <register name>

<register name>                     ::= P, A1 - A15 inclusive

<relocatable symbol>                ::= characters representing an address

<r1, r2, r3>                        ::= <register expression>

S                                   ::= store indicator

<short constant>                    ::= <predefined absolute expression>
                                        (absolute value < 256)

<statement>                         ::= {[<label>]_<operation code>_[<operand1>]
                                        [,<operand2>][,<operand3>]_[<comments>]
                                        | [* <comments>]}

<value>                             ::=
                                  {<decimal constant> | <hexadecimal constant>}

*                                   ::= {indirection indicator (in operation code
                                        field) |
                                        current value of location counter
                                        (in operand field)}
```

A source module consists of a sequence of statements. The Assembler interprets each line as it is presented.

A statement can be divided into the following fields:

- label field
- operation field
- operand field
- comments field.

<statement> ::= {[<label>]_<operation-code>_[<operand>]_[<comments>] |
* [<comments>]}

Each field has to be separated from the following by one (or more) space character(s), shown here as underlines. Spaces may not appear in the fields themselves, except when specified in a character constant or a comments field.

Instead of spaces, a backslash may be used for separation (see "Input of Source Statements and Corrections", below). One or more spaces at the beginning of a statement indicate that there is no label field. If there are eleven or more spaces at the beginning of a statement, all following characters are considered as belonging to the comments field.

An * (asterisk) at the beginning of a statement identifies that line as a comments line.

LABEL FIELD

<label> ::= <letter>{<letter> | <digit> | <delimiter>} ...
Maximum length 6 characters.

Labels (or identifiers) in a module are used to refer to other statements in a module.

In most cases, the assembler assigns to each label a word address value which is the numerical equivalent (absolute or relocatable) of the label.

The maximum number of characters in a label recognised by the Assembler is six. The first of those must always be a letter. A label may contain more than six characters, but the additional characters will not be taken into account. If the label has already been allocated to another statement, an error message is output.

Period signs in a label are not significant, e.g.

```
M.AS256  LDK  A1, 6
MAS256   ABR  A4
```

will give an error message.

The value of a label is normally regarded as relocatable, except when:
- an absolute address is equated by an EQU directive
- the label appears in an absolute program section, defined by the AORG directive, and is not equated by an EQU directive to a label previously defined as relocatable.

OPERATION FIELD

<operation code> ::= {<mnemonic>[{S | (cnd) | L}][*] | <directive>}

- <mnemonic>

The operation field normally contains the mnemonic of a standard instruction. It is possible, however, to generate one's own instruction mnemonics by means of the FORM and XFORM directives; see Chapter 5.

- S

Allowed after the mnemonic of certain register to register and memory reference instructions. It indicates that the result of the operation must be stored in a memory location and not in a register (bit 15 of the instruction is set to 1). In fact, S has to be considered as a part of the instruction mnemonic, e.g. ClR and ClRS instructions are to be considered as two different instructions.

The S may be preceded by a period sign, although the Assembler does not take this sign into account.

E.g. AD.S = ADS

- (cnd) ::= {<condition value> | <condition mnemonic>}

<condition value> ::= {0 | 1 | 2 | 3 | 4 | 5 | 6 | 7}

<condition mnemonic> ::= {Z | P | N | O | E | G | L | A | | U |
NZ | NP | NN | NE | NG | NL | NA | NR}
(See condition table below.)

This specifies the condition under which a conditional branch instruction is to be performed. The table below shows how the conditional mnemonics and condition values may be used in the Assembler.

CONDITIONAL NOTATION

COND.REG CONTENTS	(cnd)			
	GENERAL	ARITHM.	COMPARE	I/O
0	(0)	(Z) Zero	(E) Equal	(A) Accepted
1	(1)	(P) Positive	(G) Greater	(R) Refused
2	(2)	(N) Negative	(L) Less	--
3	(3)	(O) Overflow	--	(U) Unknown

NOT - CONDITION NOTATION

COND.REG CONTENTS				
	GENERAL	ARITHM.	COMPARE	I/O
0	(4)	(NZ) Not Zero	(NE) Not Equal	(NA) Not Accepted
1	(5)	(NP) Not Pos.	(NG) Not Greater	(NR) Not Refused
2	(6)	(NN) Not Neg.	(NL) Not Less	--
3	(7)	Unconditional		

- L

Allowed after the mnemonic of a constant instruction and in the ABL instruction. It specifies that the operand is contained in 16 bits, i.e. that the instruction must be assembled as a "long" instruction.

The L may be preceded by a period sign; the Assembler does not take the period sign into account.

E.g. LDKL and LDK.L give the same result.

Note: Constant instructions with registers A8 to A15 (inclusive), or with a value greater than 255, must always be long.

```
LDKL    A8,/4E20
LDKL    A1,/FFFF
LDK     A2,/FF
LDKL    A12,/BUF2 load address of BUF2 in A12
```

- *

Indicates the indirect addressing mode in a register to register or a memory reference instruction.

- <directive>

See Chapter 5 of this Part.

Location Counter

The Assembler maintains a location counter, which is a software counter used to assign relative or absolute memory addresses to program elements. The location counter starts with a relative value of zero, or at the absolute address defined by an AORG directive. The value of the counter is incremented in steps of 2 or a multiple of 2, depending on the length of the current instruction.

The current value of the counter may be referred to by an * in the operand field. In absolute program sections * has an absolute value. The value may be changed by a RES or RORG directive.

The location counter may take neither a negative relative value nor an odd value.

See also the note under the DATA directive description, in Chapter 5.

OPERAND FIELD

The operand field contains a maximum of three operands. An operand may be an address expression, a register expression or constant expression, a predefined expression, or a predefined absolute expression associated with the current machine instruction or assembly directive. The structure and meaning of the operand depends on the type of instruction and directive, and is explained below.

All expressions must be separated by a comma.

```
<expression>                        ::=  {<predefined expression> |

              <external name>{+ | -}<absolute symbol value>{+ | -}<value>}

<register expression>               ::= <register name>

<register name>                     ::= P, A1 - A15 inclusive

<predefined expression>             ::=
    {<predefined absolute expression> |
    <relocatable symbol>[{+ | -}<absolute symbol value>[{+ | -}<value>]]}

<predefined absolute expression>  ::=
{{+ | -}<absolute symbol value>[{+ | -}<absolute symbol value>[{+ | -}<value>]]
    | <relocatable symbol>-<relocatable symbol>[{+ | -}<value>]}
```

Note: * is considered to be a relocatable symbol.

In the instruction syntax the following mnemonics are used; they mean:

```
<m>  ::= <address expression>  ::= <expression>

<r1> | <r2> | <r3> ::= <register expression>  ::= <register name>

<k>  ::= <short constant> ::= <predefined absolute expression>
              (absolute value < 256)

<lk> ::= <long constant> ::= <expression>
```

The table below shows the results of a combination of positive and negative absolute or relocatable symbols:

1st term / 2nd term	+R	-R	+A	-A
+R	E	A	R	R
-R	A	E	E	E
+A	R	E	A	A
-A	R	E	A	A

Where: R = relocatable A = absolute E = erroneous.

Address Expression

The address specified in a memory reference instruction can be either absolute or relocatable.

An absolute address is the actual address in memory where the information the user needs can be found.

A relocatable address is relative to the beginning of the program in which it appears.

The address expression may contain one of the following terms or a combination of them:

* Asterisk, which is a predefined expression representing the current value of the location counter. This counter is incremented by two or a multiple of two, depending on the length of the instruction.

<symbol> Used to refer to an instruction or data word with the same identifier in its label field. The Assembler will convert the symbol to a relative address.

<displacement value>
Which can be attached to '*' or 'symbol' to indicate a word not labelled by an identifier.

Register Expression

Register expressions consist of one, two, or three characters. The register expressions recognised by the Assembler are:

P	P-register
A1 ... A14	Registers 1 to 14 (general purpose registers)
A15	Register 15 (stack pointer)

Constants

A variety of constant types may be specified in the operand of an instruction or directive.

```
<constant>                    ::= {<decimal constant> |
                                  <hexadecimal constant> |
                                  <character constant>}
```

Decimal Constants

```
<decimal constant>            ::= <digit string>
                                    range -32768 to +32767 in 16 bits
                                    range 0 to 255         in  8 bits.
```

The decimal constant is a digit or integer, contained in an 8-bit character or 16-bit word, whose value may range from 0 to 32767.

Hexadecimal Constants

```
<hexadecimal constant>        ::= {X'<hexa digit>[<hexa digit>]' |
                                  /<hexa digit>[<hexa digit>]}
                                   range 0 - /FFFF in 16 bits
                                   range 0 - /FF   in 8 bits.

<hexa digit>                  ::= {<digit> | A to F inclusive}
```

The hexadecimal constant is considered to be a hexadecimal value or bit string in the range from 0 to /FFFF.

Character Constants

```
<character constant> ::=   `<character>[<character>] ...'
```

A character constant is composed of a character string enclosed in single quotation marks. The string is composed of the characters described in the character set.

A character constant can be used with a machine instruction only if the constant consists of either one character (short constant) or two characters (long constant). Longer strings can be specified in a DATA directive. A single quote mark (') used as a character is specified by two consecutive single quote marks.

COMMENT FIELD

Comments may be included after each instruction or group of instructions, to explain the reason and meaning of the instruction(s). Comments can be written from the 40th column, or data is considered to be comments when it is separated from the operand by a space character.

Comments are printed on the assembly listing up to column 72. They are not included in the generated object program.

A line is a comment line when the line starts with an * (asterisk), or when the first eleven characters of that line are blank.

INPUT OF SOURCE STATEMENTS AND CORRECTIONS

The user may type in the statements and corrections from the operator's typewriter. He may do so by counting the number of characters to obtain a neat output on the listing device. This procedure is rather cumbersome when many statements have to be typed in. An easier way of input from the typewriter is by typing a backslash between the various parts of the statement.

Example:

```
1st column     10th column     19th column     40th column
<label>   _    <opcode>    _  <operand>    _  <comments>
```

may be typed as follows:

```
<label>\<opcode>\<operand>\<comments>
```

without having to count for the first column of each field.

Example:

```
DATAF\LDK\A4,4
\ABL(7)\HALT
DEVUN\LDK\A4,5
\ABL(7)\HALT
ADDIT\LDK\A1,0\SET INDEX REGISTER FOR BUFFER.
\LDK A3/00FF\SET LOGICAL CONSTANT INTO A3.
```

Sixteen registers are available for use by the programmer. These 16 registers, which have the predefined symbols P and A1 to A15, are called the scratchpad. They may be addressed from either the instruction being carried out or from the toggle switches on the control panel.

The specific designation of registers within the scratchpad is:

P-Register

This register is used to hold the address of the next instruction to be executed. It is incremented in steps of two if the program is executed sequentially, or it may be altered to hold the required new address if a branch is to be carried out.

Working Registers (A1-A14)

The working registers may be used with an instruction in any of the following ways:

- As accumulators, where the data to be processed can be found in a register.
- As pointers, where the specified register contains the operand address rather than the operand itself.
- As index registers, where the contents of the specified registers and the contents of the word following the instruction are added together to produce the operand address.

It is a recommended standard that register A14 be used as the application stack pointer.

Register A15

This register is used by the Monitor as its stack pointer and, as such, it is updated whenever it is used for memory addressing. It may also be addressed by an instruction in the same way as the registers A1 to A14.

Note: P, A1, A2, A3, etc., can only be used to refer to the registers. If they are used for other purposes, an error message will be output for the Assembler processor.

Name	Meaning	Internal Value	
		decimal	binary
P	Instruction Counter	0	0 0 0 0
A1	Register 1	2	0 0 1 0
A2	Register 2	4	0 1 0 0
A3	Register 3	6	0 1 1 0
A4	Register 4	8	1 0 0 0
A5	Register 5	10	1 0 1 0
A6	Register 6	12	1 1 0 0
A7	Register 7	14	1 1 1 0
A8	Register 8	1	0 0 0 1
A9	Register 9	3	0 0 1 1
A10	Register 10	5	0 1 0 1
A11	Register 11	7	0 1 1 1
A12	Register 12	9	1 0 0 1
A13	Register 13	11	1 0 1 1
A14	Register 14	13	1 1 0 1
A15	Monitor Stack Pointer	15	1 1 1 1

bit 5 6 7 8 in the instruction format.

Condition Register

The Condition Register is a 2-bit hardware register, the contents of which are determined by the result of the most recently executed "effective" instruction. By "effective", it is implied that not all instructions affect the contents of the CR. Thus the contents of the CR reflect the result of, for example, a foregoing compare instruction.

By matching these contents with a condition value (range 0 - 7), or a condition mnemonic (see syntax description), a conditional program branch may be set up.

FUNCTIONAL OPERATION OF INSTRUCTIONS

Load and Store Instructions

Load Instructions

Before the programmer can perform an operation on the contents of a memory location or a register, its contents must be placed in one of the registers A1 through A15, an operation which is performed by the load instructions. The contents of any memory location or any register are loaded into any register or memory location where the operation will take place.

The contents of a number of memory locations may be loaded in the same number of consecutive registers by means of a multiple load instruction. The first register to be loaded is always register A1.

When working in system mode on a POOM with MMU board (see also "Memory Management Unit", in Chapter 6), locations beyond 32K can also be loaded, as their addressing is taken care of by the MMU.

Store Instructions

Companion to the load instructions mentioned above are the store instructions which store the contents of a register, or a number of consecutive registers, containing the result of an operation, into a register or a memory location or a number of memory locations.

Arithmetic Instructions

Arithmetic instructions perform the normal arithmetic functions such as add, subtract, multiply and divide. The instruction operand operates upon the contents of the specified instruction.

This type of instruction includes also the double add and double subtract instructions, where operations take place on the contents of two consecutive memory addresses and registers A1 and A2.

Logical Instructions

Instructions described under this heading are called logical instructions because they operate on binary information according to the rules of logic. The first operand, which may be a memory location, a register (R1 or R3), or a constant, is compared with the second operand, register R2. The result is placed in a register or possibly in memory. In the instruction set, for each logical instruction is described in which way the contents of a memory location is ANDed or ORed.

Character Handling Instructions

Character handling instructions operate on a character level. Characters may be exchanged or compared, or 8 bits of a constant may be placed in 8 bits of a register.

Branch Instructions

These instructions cause a branch to an address in memory, either when a certain condition is fulfilled or unconditionally. In branch instructions on condition the instruction mnemonic is followed by a number ranging from 0 to 6,enclosed in brackets. When the number is (7) or omitted, the branch is unconditional.

These numbers are compared with the contents of the condition register set by the previous instruction.

The condition number has the following meanings:

(0) branch if CR = 0	(4) branch if CR ≠ 0
(1) branch if CR = 1	(5) branch if CR ≠ 1
(2) branch if CR = 2	(6) branch if CR ≠ 2
(3) branch if CR = 3	(7) unconditional branch

Example:

```
        :
        LDK     A2,4
LABEL   SUK     A2,1
        RB(4)   LABEL     Branch if CR ≠ 0.
        :
```

The Assembler allows the programmer to write, instead of a number, a condition mnemonic, e.g. Z, E, A (see the Condition Table in Chapter 3).

Unconditional branches are made by the following instructions:

- Absolute or relative branch instructions, without a condition indicator or when (7) is specified.
- CF, RTN, EX instructions.

Long format absolute instructions permit branching, forwards as well as backwards, to any address in the program. Short format absolute branch instructions may only branch to locations /0000 to /00FE. Relative forward and backward instructions may not skip more than 127 locations backwards or 128 locations forwards.

The Assembler gives an error indication if the permissible branch range is exceeded.

The address to which control is to pass may be indicated in various ways:

1. By means of a symbolic address expression
   ABL(3) LABEL

2. By an absolute address held in a register
   ABR(7) A5

3. By using a constant to indicate an absolute memory address (short constant)
   AB /84

4. By means of a displacement value added to or subtracted from the instruction counter value (RB and RF instructions only). This displacement value is computed by the Assembler from an address expression used in the operand, and may not exceed more than 128 words forwards or 127 backwards.

```
TWENTY  EQU     20
        RB(0)   TWENTY
```

Another group of branch instructions are the Call Function and Return from Function instructions. The Call Function instruction provides a link to a subroutine by branching to the first instruction of the subroutine. To be able to resume the execution of the main program after the subroutine has been executed, the contents of the P-register and the Program Status Word are stored in the stack. When the last instruction of the subroutine (RTN) is executed, the contents of P and PSW are restored.

A special group within the branch instructions is formed by the instructions EX, EXK and EXR. These instructions allow the programmer to address a memory location whose contents are the binary representation of another instruction. The latter instruction is executed before the program continues with the next instruction in sequence.

Example:

```
          :
          LDKL      A3,CIO
          LDKL      A4,SST
          :
CIO       CIO       A1,1,TY
          EXR*      A4          EXECUTE SST
          RB(4)     *-2
          :
          EXR*      A3          EXECUTE CIO
          :
SST       SST       A7,TY
          RB(4)     *-2
```

The Execute instruction may not refer to another EX, EXK or EXR instruction, nor to Call Function, RTN or double format instructions.

Shift Instructions

Shift instructions operate on a bit level. These instructions allow to rotate the contents of one of the registers A1 to A7 by n positions in the direction and manner specified in the instructions. Double shift instructions permit operation on two registers.

Control Instructions

These instructions perform the control of the program by allowing the program to be interrupted or not, or to reset an internal interrupt. Except for the LKM instruction, control instructions should only be used in Stand Alone programming.

INH and ENB are two companion instructions. The program part between these two instructions is not interruptible, as INH inhibits all interrupts. ENB sets the machine status to allow interrupts.

Examples:

```
          IDENT     TEST
OUT       EQU       *
          RORG      OUT+/600
START     HLT                          )
          INH                          )
          LDK       A5,0               )
          LDKL      A11,BUF            )
          LDK       A2,0               )
AGAIN     CIO       A2,1,/30           )    program inhibited
          RB(NA)    AGAIN              )
          LC        A3,BUFPT,A5        )
          :                            )
          ENB                          )
```

The RIT instruction is used to reset an internal interrupt which was previously set by an interrupt from the control panel, power failure/automatic restart, real-time clock, or by a program error.

The programmer may specify a 5-bit hexadecimal value in the operand of this instruction to clear specific interrupts.

```
INTRTC   RIT  /1B  Reset the real-time clock interrupt.
```

I/O Instructions

I/O instructions handle the data transfer between the CPU and peripherals, the operation of contrl units for these peripherals and status control.

In Monitor-controlled programs the I/O functions, initiated by these instructions, are taken over by a general I/O routine which is called each time an LKM instruction, followed by a DATA directive, is used. The user need not, therefore, write his own I/O routines.

When the programmer has to write a Stand Alone program, he must write his own I/O routines. Since there is no memory protection option, except when working with Memory Management Unit MMU, the programmer must be careful not to overwrite parts of a program already in memory.

External Transfer Instructions

These instructions may only be used in system mode. The instructions RER and WER may be used to address an external register. The function of these instructions is described in Chapter 6.

The remainder of the instructions in this group are instructions involving the operation of the MMU in the P857M. They permit to load the 16 registers on the MMU board with information pertaining to the up to 16 pages into which a program can be divided.

Example:

```
SEGTAB   DATA      /0000
         DATA      /0400
         :
         DATA      /3000
         :
         TL        SEGTAB
```

A Table Store (TS) instruction writes the contents of these registers in the MMU, which are updated during the program execution, back to the specified reserved locations.

Move Table Instructions

The instructions under this heading are only accepted on the P800 models fitted with an MMU. They allow copying of a string of consecutive memory locations into another area, or when working in system mode with MMU, a string of consecutive memory locations from a user area to a system area and vice versa.

String Instructions

The instructions under this chapter are only accepted on the P854, P858 and P859. They allow moving, filling and comparing strings of consecutive byte locations.

ADDRESSING MODES

In Volume II we see how addressing takes place from a hardware point of view. The conditions an instruction must fulfil to meet the requirements of the Assembler are explained on the preceding pages. Specific examples, with source statements and explanation concerning the arithmetic instructions AD and ADR, are given to show the operation within the CPU.

See Volume II for the hardware operation of these instructions. The order in which these examples are given is in accordance with the description on those pages.

Direct Addressing

AD A1,LABEL — The contents of the memory location with symbolic address LABEL are added to the contents of register A1. The result is placed in A1.

ADS A1,LABEL — The contents of the memory location with address LABEL are added to the contents of register A1. The result is stored in LABEL.

Indexed Addressing

AD A2,LABEL,A10 — The contents of register A10 are added to the address LABEL. The result gives an address whose contents are added to the contents of A2. The result of the latter operation is placed in A2.

ADS A2,LABEL,A10 — The contents of register A10 are added to the address LABEL. The result gives an address whose contents are added to the contents of A2. The result of the latter operation is stored in the address: LABEL + contents of A10.

Indirect Addressing

AD* A2,LABEL — The contents of LABEL point to an address whose contents are added to the contents of register A2. The result is placed in A2.

ADS* A2,LABEL — The contents of LABEL point to an address whose contents are added to the contents of register A2. The result is placed in the location whose address is in LABEL.

Indexed Indirect Addressing

AD* A2,LABEL,A10 — LABEL is added to the contents of register A10. The result points to a location whose contents are added to the contents of register A2. The result is placed in register A2.

ADS* A2,LABEL,A10 — LABEL is added to the contents of register A10. The result points to a location whose contents are added to the contents of register A2. The result is placed in the address obtained by adding LABEL to the contents of A10.

Register to Register Operation

ADR A1,A2 — The contents of A2 are added to the contents of A1. The result is placed in A1.

Register Addressing

ADR* A1,A2 — The contents of the address pointed to by A2 are added to the contents of register A1. The result is placed in A1.

ADRS A1,A2 — The contents of the address pointed to by A2 are added to the contents of A1. The result is stored in the address pointed to by A2.

Directives are used to provide a framework for a program and to guide the assembly process. The directives are written in the program, and are printed on the assembly listing if the listing option is specified in the ASM command.

PROGRAM FRAMEWORK

The directives IDENT and END form respectively the first and last statements in the module. They are mandatory.

The IDENT directive is used for identification purposes and the END directive generates the END cluster, after which the assembly process is stopped and a symbol table is printed.

LINKAGE CONTROL

Some modules which must grouped into one larger program contain references to identifiers defined in other modules.

By means of the directives ENTRY and EXTRN, the user is able to refer to certain parts in other modules, whereas the directive COMN allows to transfer data among several modules either written in Assembly Language or in FORTRAN.

By using a COMN, the programmer can define one or more common blocks. Each common block may be divided into a number of subfields of varying length, each having a symbolic name which can be referred to directly, but only in the module in which it is declared.

COMN blocks may be labelled or blank; a COMN block is labelled if a name is attached to it.

The Linkage Editor allocates a space to the blank common block at the end of the link-edit run (see Linkage Editor). This block is placed at the end of the entire program. A labelled common is placed at the end of the first module that refers to it.

The ENTRY, EXTRN and COMN directives must always follow immediately after the IDENT directive and in this order, though it is not necessary for all of them to be specified.

```
So: IDENT, ENTRY, EXTRN, COMN     or
    IDENT, EXTRN, COMN            or
    IDENT, ENTRY, COMN            etc.
```

ASSEMBLY CONTROL

When it is necessary to check whether a certain condition is satisfied before assembling a number of source lines, the user may include the directives IFT, IFF and XIF. The assembly of the IDENT, END and XIF directives is never bypassed by IFT or IFF.

By means of the STAB directive the user may specify one or more internal symbols which are to be used for Debugging purposes. All these symbols must have been defined previously in the current module.

Common block names are handled as externals.

The RORG and AORG directives are used to reset the location counter to a relocatable or absolute value indicated in the operands of those two directives.

The AORG and RORG directives are respectively used to define an absolute module section and a relative module section. The AORG and RORG directives are only to be used for self-contained executable programs. If the Linkage Editor encounters an absolute address specified by an AORG directive, it issues a warning message and treats the address as relative.

The RORG directive is used to reset the location counter to a relocatable value, indicated in the operand of this directive, after the AORG directive has set the location counter to give absolute addresses.

VALUE DEFINITION

The directives DATA and EQU are used to define certain values in a module.

AREA RESERVATION

The directive RES can be used to skip over an area in memory. The RES directive saves a memory area of a given length, specified in the operand, advancing the location counter by twice the number of words specified.

LISTING CONTROL

The Assembler normally produces an output listing for each assembly. By means of the directives EJECT, NLIST and LIST, the programmer may determine which parts of the modules do not need to be listed.

SYMBOL GENERATION

Two directives, FORM and XFORM, allow the user to make a number of special instructions for a specific purpose or program. In the FORM directive the user may define the bit configuration and the mnemonic of the special instruction. If two FORM-defined instructions are to be specified, which differ only in the contents of certain fields, the programmer may use the XFORM directive to specify the second instruction.

Any useful pseudo-instruction or system macro can be defined once and thereafter used without having to be generated by a FORM directive in every program where it is used.

IDENT — Program Identification

Syntax: _IDENT_<module name>

where:

<module name> A symbol which is specified according to the rules for a label.

The IDENT directive specifies the name to be given to the object module output by the Assembler. It is used for identification purposes in selective loading or updating (see Part 3, Linkage Editor, and Part 4, Update). This directive must always be present, and must be the first statement in a module.

END — End of Assembly

Syntax: [<label>]_END_[<predefined expression>][,<symbol>]

where:

<label> The label is given a relative value equal to the length of the relative section of the generated object program. This length includes the length of the optional symbol table (see STAB directive). The length is 0 if this module is absolute.

<predefined expression>
This expression, if present, gives the address of the first instruction to be executed in the program after loading.

<symbol> This parameter gives an entry point name to the internal symbol table of the generated object program when the STAB directive has been assembled. The internal symbol table consists of a list of all relocatable symbols defined, with their numerical equivalents. The STAB directive must immediately precede the END directive.

This directive must be the last statement in a module, and terminates the assembly by writing an :EOS mark.

ENTRY — Define Entry Point Name

Syntax: _ENTRY_<entry name>[,<entry name>] ...

where:

<entry name> is a label in this module, which can be referred to by an operand of an instruction in another module. The maximum number of entry names which can be specified in one ENTRY directive is determined by the length of one line (up to column 72, inclusive).

The ENTRY directive is used to declare entry names, i.e. labels which are defined in the current module and used as operands in another module.

Examples:

```
            IDENT    PROG                    IDENT    PROGA
            ENTRY    NUMB1,NUMB2,NUMB3       ENTRY    LABEL,REFER
            EXTRN    LABEL,REFER             EXTRN    NUMB1,NUMB2,NUMB3

START       :                                :
NUMB1       LDKL     A3,LABEL     GO         LDK1     A2,0
            :                                :
NUMB2       ST       A6,REFER     LABEL      ST*      A4,MEMO
            :                                :
NUMB3       CF       A14,EOS      REFER      AD       A3,TOTAL
            :                                :
            :                                :
            END      START        TOTAL      SU       A5,TARRA
                                             :
                                             END
```

EXTRN — Define External Reference

Syntax: _EXTRN_<external>[,<external>] ...

where:

<external> is the name of an external reference (a label in another module). The maximum number of external names which can be specified in one EXTRN directive is determined by the length of one line (up to column 72, inclusive).

The EXTRN directive is used to declare external names, i.e. labels which are defined in another module and used as operands in the current module.

Example: see ENTRY.

Syntax: [<label>]_COMN_<common field definition list>

where:

<common field definition list> ::= <common field definition>[,<common field definition>] ...

<common field definition> ::= <common field name>[<common field length>]
<common field name> ::= <identifier>
<common field length> ::= (<predefined absolute expression>)

If the parameter <common field length> is omitted, the default value assumed by the Assembler is 1. The field length must be given in words.

The COMN directive facilitates communication between modules written in Assembly Language and FORTRAN.

Example:

```
A_COMN_FVAL1(3),FVAL2(3),INTGV(10)
```

which defines a labelled common, named A, with length 3 + 3 + 10 = 16 words.

A is defined as an external reference and common block name. Either the common block name itself or the subfield names may be referred to in the same module. The subfield names are then considered to be equivalent to:

<common block name> + <absolute displacement>

For examples:

```
LD_A1,FVAL2 is equivalent to LD_A1,A+6

ST_A2,INTGV+18 is equivalent to ST_A2,A+30
```

The displacements in this example are counted in characters. Blank commons can only be referred to by the subfield names defined in the operand field.

```
_COMN_VAL1(3),VAL2(4)
_COMN_VAL3(9),VAL4(10)
```

These directives define a blank common of 3 + 4 + 9 + 10 = 26 words.

VAL2, for instance, may be used in symbolic expressions and is equivalent to:

blank common "name" + 6

More than one blank common may be specified in one module.

IFT (IF True)

Syntax: _IFT_<predefined absolute expression>=<predefined absolute expression>

If the first parameter ≠ second parameter, the source line(s) following IFT up to the next XIF directive are not assembled.

IFF (IF False)

Syntax: _IFF_<predefined absolute expression>=<predefined absolute expression>

If the first parameter = the second parameter, the source line(s) following IFF up to the next XIF directive are not assembled.

XIF (End of Conditional Assembly)

Syntax: _XIF_

This directive allows all subsequent statements to be assembled until a new IFT or IFF statement is encountered.

These directives are used to indicate that a block of statements is to be assembled only if a certain condition is fulfilled. The assembly of the IDENT, END and XIF directives is never bypassed.

Syntax: _STAB_[<internal symbol list>]

where:

<internal symbol list> ::= <internal symbol>[,<internal symbol>] ...

The STAB directive outputs, at the end of the relocatable program section of the generated module, one or more internal symbols to be used for Debugging purposes (the internal symbol is the address given to a symbol in the program after assembly). All symbols must have been declared previously in the current module.

STAB must immediately precede the END directive.

If the STAB directive does not contain a parameter in the operand field, all internal symbols of the module will be included.

The programmer may not specify entry points, external reference names or commons. This directive is only taken into account when in the END directive the parameter <symbol> is specified, giving the name of the internal symbol table.

AORG — Assign Absolute Origin — AORG

Syntax: _AORG_<predefined absolute expression>

This directive assigns an even absolute value to the location counter. The location counter receives the value specified by <predefined absolute expression>.

From the time AORG is given and until a RORG directive is given, the location counter is incremented in the same way as if it were relative, i.e. by increments of 2 or 4 depending on the length of the instruction. All labels are given an absolute value, unless they are equated to a predefined relative value by an EQU directive.

RB and RF instructions in an absolute program section cannot refer to an address in a relocatable program section, as the displacement cannot be calculated.

* The user should be aware of the fact that the Disc Assembler accepts absolute addresses, but that the Disc Overlay Linkage Editor does not, and will output a warning message and treat the address as relative.

RORG — Assign Relative Origin — RORG

Syntax: _RORG_[<predefined relocatable expression>]

The RORG directive allows the user to specify the beinning of a relocatable module by assigning a relative value, which must always be even, to the location counter. Its value may never become negative. If RORG has no operand, the location counter is given the last relocatable value it has previously received. This value is equal to the length of the relocatable module at the time this directive is assembled.

Syntax: [<label>]_DATA_<data expression>[,<data expression>]

where:

<data expression> ::= {<character string> | <expression>}

<label> is given the address of the first or only word of data.

<data expression> either an expression, or a character string consisting of from one to thirty-two ASCII characters enclosed by single quote marks. A series of words is generated, of two characters each, left justified. When the number of characters is odd, the rightmost character of the last word is a space.

The DATA directive is used to assign a value to one or more words in the module, for inclusion in the object module. The maximum number of words assigned by one DATA directive is 16.

Note: Though * points to the current value of the location counter, * refers always to the first word of a generated sequence (multiple word instruction, FORM defined instruction, DATA directive).

Examples:

```
_DATA_`ABC',/0A0D,1,/A,2,'DEF'
```

will generate the following words (in hexadecimal code):

4142	4320	0A0D	0001	000A	0002	4445	4620
'A B	C _'	1f cr	1	/A	2	'D E	F _'

An ECB may be built as follows:

```
ECB_DATA_1,BUF2,6,0,0,0
```

ECB

0001	xxxx	0006	0000	0000	0000

(in which /xxxx is the address of BUF2).

```
_DATA_-(128,+12,/3AB,-/A,LABEL,'TEXT:'
```

generates the following:

FF80	000C	03AB	FFF6	xxxx	5445	5954	3A20
-128	+12	/3AB	-/A	LABEL	'T E	X T	:_'

Syntax: <label>_EQU_{<register expression> | <predefined expression>}

Labels are normally defined by being assigned memory values as they appear in the label field of an instruction. The EQU directive may be used to define a label in a direct manner by assigning to it the value of an expression in the operand field. The symbol in the label field is made equivalent to the value in that operand field. This value may be absolute or relocatable.

A symbol, provided it differs from standard mnemonics and FORM-defined mnemonics, may be used as an operation mnemonic, but may not be followed by an operand. The Assembler generates one code word each time this mnemonic appears in the operand field.

Examples:

```
CT      EQU  /41C4      CT may now be used anywhere in the program to represent
                        the value /41C4 or the instruction CIO A1,1,/04.
        :
        CT              -    /41C4 represents an instruction.
        LDKL A1,CT      -    CT is loaded as a constant.

VAL     EQU  10         VAL receives the value 10. (Short constant.)
        :
        LDK  A1,VAL

LAB     EQU  *          LAB receives the value of the location counter.

C:1     EQU  25         Each time the Assembler encounters C:1 or REG:3, they
REG:3   EQU  A3         are replaced by 25 and A3, respectively:

        LDK  A1,C:1       --->  LDK    A1,25
        LDK  REG:3,1      --->  LDK    A3,1
        LDK  REG:3,C:1    --->  LDK    A3,25
```

Syntax: <label> RES <predefined absolute expression>

where:

<label> receives the address of the first word of the reserved area.

<predefined absolute expression>
specifies the length of the area to be reserved, in words.

If <predefined absolute expression> is 0, the location counter is not updated; if <label> is specified, the statement is equivalent to:

```
<label>   EQU  *
```

The RES directive is used to reserve a number of memory words. The programmer may specify this number in the parameter. The location counter is incremented or decremented, depending on the positive or negative value of the parameter. If positive, a memory area of the specified value is reserved. If negative, a memory area of the specified size is reserved before the place identified by <label>. The value of the latter is not changed, but the location counter is reset to a lower value by subtracting twice the value specified.

Examples:

```
location
counter
0000                RES      4     Reserve 4 words.
0008      LAB1      RES      -2    Reserve two words before LAB1.
0004      INS       RES      0     INS receives the value of the location
                                   counter.
```

Symbol Table:

```
LAB1      0008      INS 0004
```

Example of Stack Reservation

```
STACK     RES  4          STACK      -->|          |
                                       ------------
BASE      EQU  *-2                     |          |
                                       ------------
     :                                 |          |
                                       ------------
     :                                 |          |
                                       ------------
     :                  BASE  *-2 -->|          |
                                       ------------
     :                        *   -->|          |
                                       ------------
          LDKL      A14,BASE
```

Load stack base address into A14.

EJECT — Continue Listing on New Page

Syntax: _EJECT_

This directive causes the remainder of the current page of the line printer paper to be left blank; the listing is continued at the top of the next page.

NLIST — Suspend Listing

Syntax: _NLIST_

The NLIST directive causes the Assembler listing to be suspended from the point where this directive is given until either the END directive or a LIST directive.

Lines which contain errors will continue to be printed during this phase.

LIST — Resume Listing

Syntax: _LIST_

The LIST directive causes the Assembler to resume the listing after it has been suspended by an NLIST directive.

Syntax: <label> FORM <field definition>[,<field definition>] ...
<field definition>[/<field number list>]

where <field definition> ::= <field length definition>[{= | :}<field value definition>]
<field number list> ::= <field number>[,<field number>] ...
<field number> ::= <decimal constant>

<label> Defines the new instruction mnemonic. The operand field of the directive must then contain values to be placed in any non-predefined fields. The last non-predefined value is the default.

<field length definition>
Specifies the number of bits to be allocated to a field of the word, in the range 1 to 16. If several fields are defined inside a word, the sum of the field lengths must be 16. The maximum number of consecutive words defined by a single FORM directive is eight.

<field value definition>
If the value is preceded by an equals sign (=), may be used to place a value into the field to which it refers.

If the value is preceded by a colon (:), the value indicates the address of a word relative to the first word of the expansion defined by FORM. The value definition itself may be a predefined expression, an external reference without any displacement, or a predefined absolute or relocatable expression. If a particular field has not received a value definition, it will be filled with zeroes.

<field number list>
If the programer wishes to put the values of the operand field of the FORM-defined mnemonic in an order different from that of the non-predefined field they are to occupy, or if the user wishes to alter the values held by any of the predefined fields, he must use the <field number list> parameter in the FORM directive.

Each field generated is given a number, beginning with 0 for the first field, 1 for the second field, up to n-1 for the nth field. n may not exceed 15.

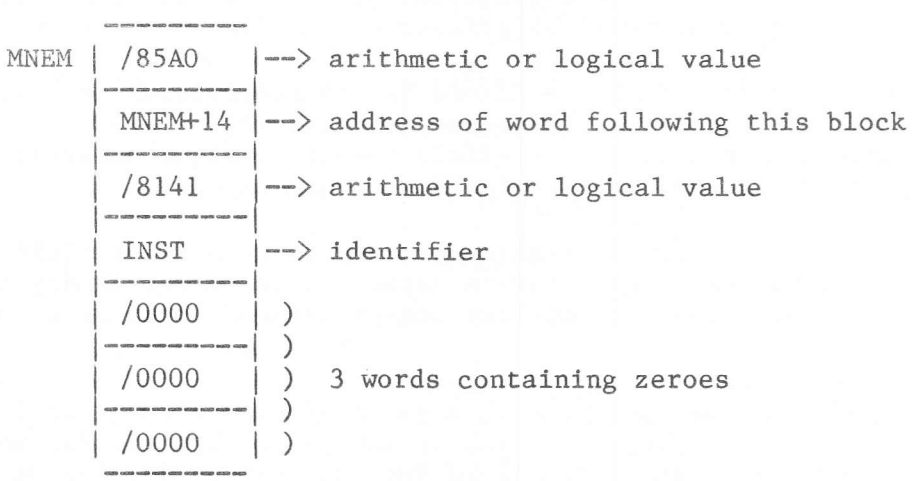
The <field number list> must be preceded by a / (slash), and be placed after the last field definition on the FORM directive. All fields not predefined in the field definition must be specified in the <field number list>, if it is used.

A field number is represented as a decimal integer.

If a <field number list> is specified after a FORM directive, the operand expressions following the pseudo-mnemonic will occupy the fields specified in the field number list in the given order. In this way, the contents of predefined fields may be altered while blank fields may be left blank.

This directive is used to define the format of a group of from one to eight words named by an identifier, which can be used as an instruction mnemonic later in this module. The directive is written as follows:

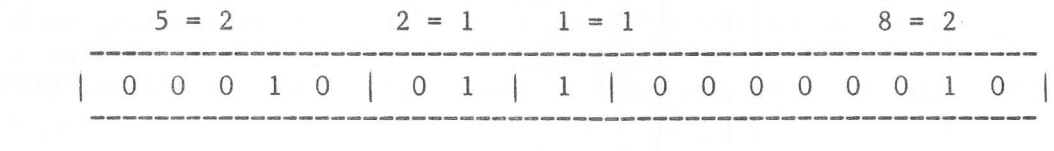
Example: MNEM FORM 16=/85A0,16:14,16=/8141,16=INST,16,16,16



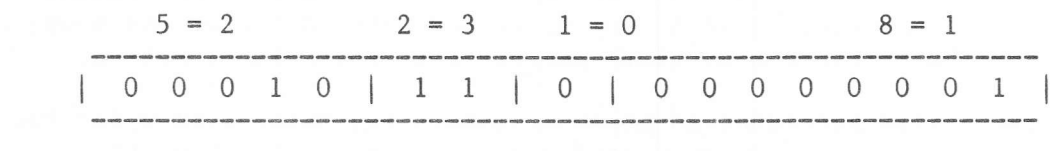
The parameter 16:14 indicates a word address seven words from the beginning of the expansion defined by FORM. The programmer has to specify this address, as the last three words are left zero.

Example

Suppose the user has specified in his program, by means of a FORM directive, a 16-bit word of the following format:



He wishes to have this word changed to:



He may do so by using the following instruction sequence in his module, using the <field number list> in the FORM directive.

```
        IDENT     EXAM
        :
WORD    FORM      5=2,2=1,1=1,8=2/2,1,3
                                 |_____|
        :                          |
        WORD      0,3,1 <----------
        :
        END
```

The Assembler will now change the fields as follows:

- field no 2 (1=1) will be changed to contain the value 0
- field no 1 (2=1) will be changed to contain the value 3
- field no 3 (8=2) will be changed to contain the value 1
- field no 0 (5=2) will keep the value 2.

The operand expressions following a pseudo-mnemonic are positional parameters. If one parameter is omitted (other than the rightmost one), its position must be indicated by a comma.

If a FORM defined mnemonic is identical with a standard instruction mnemonic, the pseudo-mnemonic is given priority.

Example:

This example shows how the programmer may make an I/O request if not all parameters are known. Without the FORM directive he would have to write the instruction sequence:

```
            LDK       A7,order
            LDKL      A8,DECB
            LKM
            DATA      1

00000                                 IDENT   FORM
00001                         INOUT   FORM    8=/07,8,16=/80A0,16,16=/2804,16=1
00002       0000              BUFFER  RES     10
00003       0014  0008        DECB    DATA    8,BUFFER,20,0,0,0
            0016  0000 R
            0018  0014
            001A  0000
            001C  0000
            001E  0000
00004       0020  0782        START   INOUT   /82,DECB
            0022  80A0
            0024  0014 R
            0026  2804
            0028  0001
00005       002A  2804                LKM
00006       002C  0003                DATA    3
00007                                 END     START

            SYMBOL TABLE

BUFFER      0000 R    DECB    0014 R    START     0020 R
ASS.ERR     00000

:EOF
```

From now on the programer may use INOUT /82,DECB instead of LDK A7, ...

In this example, the newly FORMed INOUT instruction creates the same coding as the four instructions at the top of the page. Note that fields 2 and 4 are left blank in the initial definition of the function, as these will be specified when used. (They indicate, respectively, the type of I/O required and the address of the Event Control Block (ECB) of the device concerned.)

Syntax: <label> XFORM <label of FORM directive>,<field list>

The XFORM may be used each time two FORM-defined pseudo-mnemonics have to be defined which do not differ in their format, but only in the values of the predefined fields.

<field list> is a series of field definitions, giving the format of the new pseudo-mnemonic and the contents of its fields.

The field length definitions must be the same as those of the FORM-directive referred to, and appear in the same order.

Example:

```
INST1     FORM      8=/FF,4,4,16/1,3,2
INST2     FORM      8=/33,4,4,16/1,3,2
```

The XFORM directive may be used to generate an INST2 instruction as follows:

```
INST2     XFORM     INST1,8=/33,4,4,16
```

Data transfers between input/output devices and the central processor are controlled by device control units, each of which may have one or more devices attached to it, depending on the type of device. Control units are attached to the central processor by an interrupt or break line, by address lines and by other signal lines, which are used by the computer to determine whether a data transfer can be performed.

Data transfers take place through a channel, the General Purpose Bus. The actual programming of the data transfers may be on a character or word basis, where either each word (or character) is programmed and transferred individually via the Programmed Channel, or the user may program blocks of words or characters via the I/O Processor. In the latter case, external registers on the I/O Processor must be addressed.

STAND-ALONE OR MONITOR PROGRAMMING

The basic difference between Stand Alone programming and Monitor controlled programming is caused by the fact that in Stand Alone programming the user has to write his own input/output routines, whereas in Monitor controlled programming the user may call certain Monitor functions by means of Links to Monitor which execute the input/output. For information on programming in either mode, refer to the P800M Software Training Manual (Publication No. 5122 991 1243X); see also later in this Chapter.

INTERRUPT SYSTEM

When working in interrupt mode, each interrupt program may be connected to an interrupt level. As the actioning of an interrupt involves the direct accessing of the interrupt level's start address from its hardware interrupt location, the contents of this location must have been previously loaded with the correct address.

The start addresses loaded in these locations are not fixed and must be defined by the programmer.

Interrupt Level	Interrupt Location
0 to 62	/0000 to /007C

where level 0 has the highest priority and 62 the lowest. The first 16 levels are hardware interrupt levels, of which level 0 has the highest priority.

For example, if the control panel interrupt is wired to interrupt level 7, the start address of the corresponding routine should be placed in location /E.

SYSTEM STACK

To save the contents of registers when the main program is interrupted, the hardware interrupt routine automatically uses register A15. This register addresses the stack which is to hold the contents of the P-register and the Program Status Word at the time the program was interrupted. It is, therefore, necessary to reserve sufficient space for the stack and to load register A15 with its start address. This may be done by using the appropriate assembly directives and by defining the start address by means of an identifier. The start address is the highest address reserved, as the stack is filled from the higher towards the lower addresses.

Apart from the contents of the P-register and PSW, the stack may be used to save the contents of other registers as required by the program. These registers are saved by means of Store instructions (one for each register). Before returning to the main program, Load instructions are required to restore the contents of the stack, prior to RTN. During the hardware action further interrupts are inhibited. If the user wishes to allow the specific routine to be interrupted, he must give an ENB instruction.

USER STACK

We have seen that with the A15 stack the P-register, the PSW and any other registers are saved with Store instructions in this stack towards the lower addresses. Now, if a user calls a subroutine with a CF instruction, the contents of the P-register and the PSW are automatically stored in a stack he has set up previously, for example as follows:

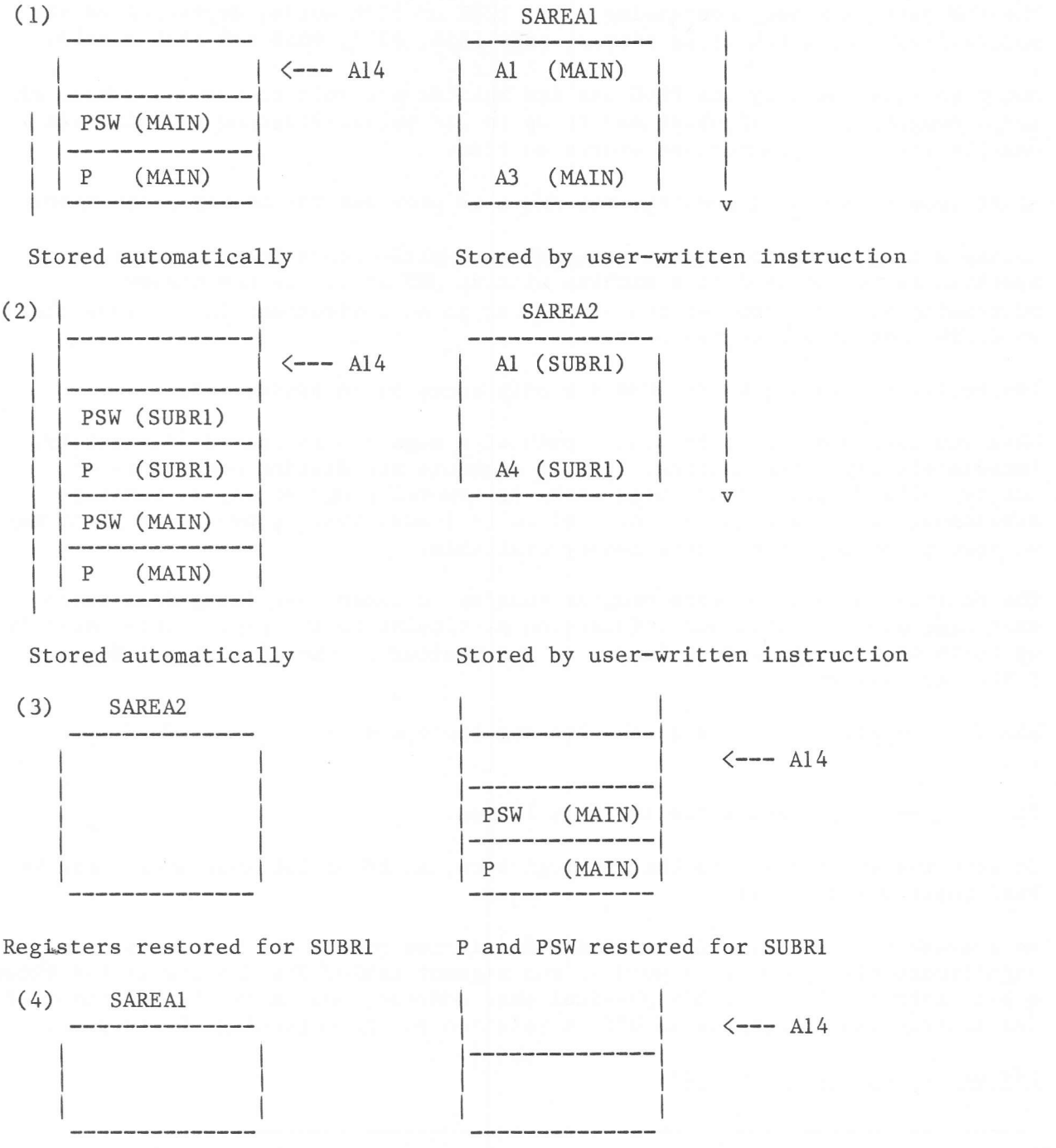
```
    RES   20
STB EQU   *-2
    LDKL  A14,STB       then the subroutine is called:

    CF    A14,SUBR      and the P and PSW are stored in the A14 stack (other
                        registers may also be used as stack pointers).
```

For example, for a program with two subroutines, one subroutine calling another one, the saving may be done as follows:

```
        IDENT MAIN      IDENT  SUBR1          IDENT SUBR2
SAREA1  RES  3          ST     A1,SAREA1           ST   A1,SAREA2
SAREA2  RES  4          ST     A2,SAREA1+2         ST   A2,SAREA2+2
        :               ST     A3,SAREA1+4         ST   A3,SAREA2+4
   (1)  CF   A14,SUBR1  :
        :          (2)  CF     A14,SUBR2           ST   A4,SAREA2+6
        END             :                          :
                        LD     A1,SAREA1           LD   A1,SAREA2
                        LD     A2,SAREA1+2         LD   A2,SAREA2+2
                        LD     A3,SAREA1+4         LD   A3,SAREA2+4
                        :                          LD   A4,SAREA2+6
                   (4)  RTN    A14           (3)  RTN  A14
                        END                        END
```

In this example the following save operations take place:



Note:

It is possible to return from SUBR2 directly to the main program, but in such a case the user must update the A14 register contents, i.e. the stack pointer, himself (with 4, in this case).

MEMORY MANAGEMENT UNIT (MMU)

The MMU extends memory addressing up to 128K or 512K words, depending on the model of P800 to which it is fitted; only P854, P857, P858 and P859 models.

Owing to this facility the P800 and its Monitor are able to serve a number of large programs, each of which may be up to 32K words. Programs of this size usually will be segmented and stored on disc.

Apart from extended addressing, the MMU also provides for memory protection.

Coding a program for operation with MMU and Multi-Access Monitor requires no specific rules compared to a machine without MMU as far as the memory addressing is concerned, as the addressing in an environment larger than 32K words is transparent to the user.

Instructions relating to the MMU are only accepted in system mode.

When the user program is called, a path of n segments is loaded into memory, immediately after the Monitor. These n segments are divided over parts of memory called pages, of 2K words each. As several programs may be running simultaneously, the pages do not need to be loaded next to each other, but may be spread out over the entire memory available.

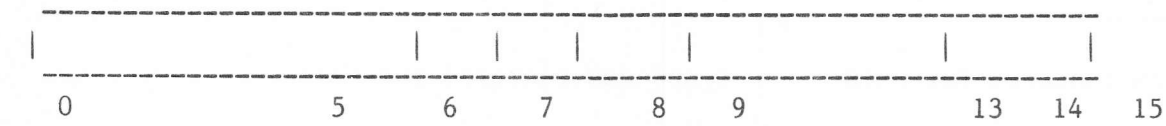
The Monitor builds, for each program running, a table containing data where each page may be loaded and information particular to the page. This table is up to 16 words long, and is loaded by the Monitor in the 16-register Segment Table, as follows:

```
LDR     A4,A11    where A11 contains the table address
:
:
TL      A4        where the table is loaded.
```

To save the information in the MMU registers, an ES or ESR instruction may be used (system mode only).

An address in the user program is divided in two parts. The four most significant bits point to a word in the segment table. The MMU translates these 4 bits into a 6-bit or 8-bit physical page address, and takes the remainder of the instruction address as an offset relative to the beginning of the page.

Layout of Segment Table Word



bits 0 to 5 — Physical page address, as derived from the four most significant bits in the instruction address.

bit 6 — Page error indicator. This bit is set by the Monitor when a program attempts to access a missing or wrong page. The MMU will give a "Page Fault" interrupt. This bit is not used for system programs.

bit 7 Read-only page. When this bit is set, the page is protected against overwriting. A "Page Fault" interrupt is given when a program tries to write into it.

bit 8 Modified page. This bit is set by the MMU when a write operation took place in this page. Instead of being overlaid, the page is first written back onto disc before the area is used again.

bits 9 to 13 Not used.

bits 14, 15 Not used with MMU's which can access only 128K words; used as the most significant two bits of the page address, in MMU's which can handle up to 512K words.

Memory Protect

The memory protect facility of the MMU is obtained by setting bit 7 in the table containing the words to be loaded in the MMU segment table registers. Remember, however, that instructions concerning the MMU are only accepted in system mode. If an attempt is made to access a protected page, a "Page Fault" interrupt is given. This interrupt has the highest priority, and causes storing in the system stack of:

- the address of the instruction which caused the interrupt
- the PSW
- a word containing the page address of the page in which the fault was detected, and the program level.

This interrupt is reset automatically after a branch has been made to the interrupt routine address.

FLOATING-POINT PROCESSOR

The Floating Point Processor is an optional, high speed arithmetic processor which may be included in the P857M system. It performs by hardware, single precision, all floating point arithmetic operations.

Operation

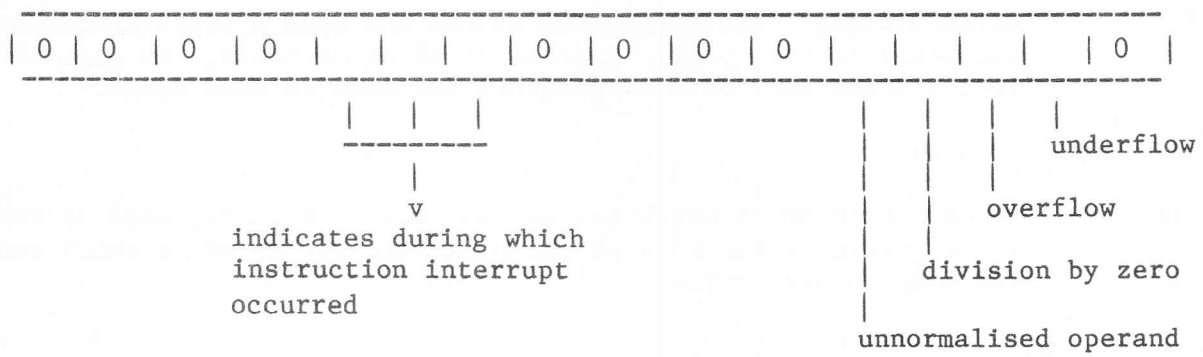
The board contains three 16-bit floating-point accumulators (FPA's) holding the result of a floating point operation, or the floating point operand, or the first floating point operand where the second floating point operand is temporarily placed in three other 16-bit registers.

Program instructions are fetched and decoded by the CPU. The significant bits of each instruction, i.e. op-code, mode, etc., are also copied to an instruction register on the FPP board. When a floating point instruction is encountered in the program, the Floating Point Processor is activated by the CPU and the latter stops.

Some decoding of the instruction register contents takes place on the FPP board, and an arithmetic unit on this board is signalled the type of operation it has to perform. The arithmetic unit takes the information to be operated upon from the contents of the FPA, registers A1 and A2, or the contents of consecutive memory locations.

The result is stored in FPA, or A1 and A2, or a number of consecutive memory locations.

During or immediately after the execution, either a status register is reset to zero (no errors) or bits in this register are set to 1. The contents of the status register may be:

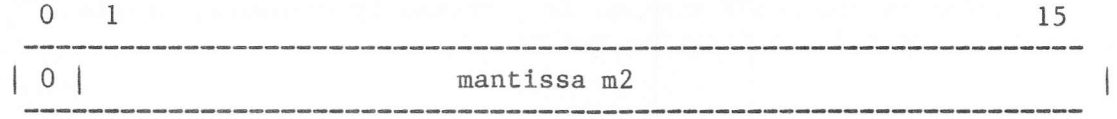


Any abnormal condition gives an FPP interrupt and sets the CPU condition register to 3. The FPP interrupt must be connected to one of the eight internal interrupt levels. If no error was caused, the CPU fetches the next instruction.

Floating Point Format

Floating Point Data are real numbers contained in three consecutive 16-bit words. The first two words contain the mantissa, which is a left normalised, double precision number. The exponent is held in the third word as a single precision integer.

sign bit of the mantissa.



sign bit of the exponent.

The sign bit of the second mantissa word is always zero. The mantissa scale is between:

positive: $+\frac{1}{2} \leq m \leq 1^{-e}$)
)
negative: $-1 \leq m \leq -\frac{1}{2}^{-e}$) where $e = 2^{-30}$

The exponent scale is between: $-2^{15} \leq E \leq +2^{15}-1$

A floating point number is: (m1,m2) * 2^{E}, where E = the exponent.

The absolute value is: | DATA | < 10^{9868}. The accuracy is 9 decimal digits.

The Floating Point Processor also allows the conversion of floating point data to integer format and vice versa. In this case the Processor permits operations with single precision integers (in 16 bits), and double precision integers (in 32 bits, the most significant bit of the second word being 0).

TRAP ACTION

Instructions input to the P800M computer are checked and decoded by the CPU's hardware. If an unexecutable instruction is encountered, a trap action is started, which consists of a hardware and software operation.

The hardware operation of the trap consists of the following actions:

- The CPU does not attempt to carry out the instruction.
- Interrupts are inhibited.
- Information which refers to the instruction's address and processor status (P and PSW) is saved.
- An indirect branch is made to location /7E (start of trap routine).

The software operation of the trap consists of:

- Save the address in P.
- Save the instruction's bit pattern and its second word, if any.

STAND-ALONE INPUT AND OUTPUT PROGRAMMING

Programmed Channel

To control the data transfer between the device and the CPU, the following instructions are, in general, available:

CIO Start	Start input or output
CIO Stop	Stop input or output
INR	Input one character
OTR	Output one character
SST	Send status of the control unit
TST	Test if the control unit is busy.

The register <r3> used in the CIO instruction must always contain additional information for the control unit, e.g. input, output, parity, echo. Which information must be loaded can be found in the relevant hardware manuals delivered with the system.

When the CIO Start instruction is accepted (test the condition register), it is followed by an INR or OTR instruction. When the last character is transferred, a CIO Stop instruction must be given. This instruction should be followed by an SST instruction, which gives the status of the relevant control unit and may reset an interrupt and switch a control unit to the Inactive State.

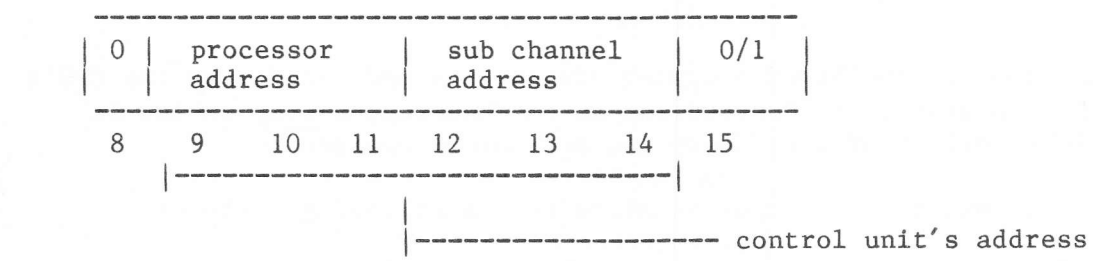
I/O Processor

The I/O processor allows the high speed transfer of variable length or fixed length data blocks between a suitable control unit and the processor.

Up to eight I/O processors may be connected to the General Purpose Bus, each of which may control up to eight control units via eight subchannels.

Each I/O processor contains two working registers, which are used to effect register to register exchanges with the CPU internal registers.

Before a data transfer can be realised, the user has to specify two control words for two external registers. These external registers are addressed by two WER instructions, in which the address part must be composed as follows:



where processor and sub channel address are determined at system installation time. Both addresses, which may range from 0 to 7, together form the attached control unit address. Bit 15 determines which control word is sent:

bit 15 = 0: first control word
1: second control word.

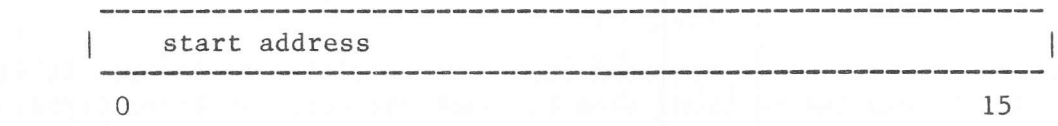
- Format of Control Words

The format of the first control word is:

where:

bit 0 = 1: exchange is in word mode
0: exchange is in character mode.
bit 1 = 1: exchange is from memory to control unit (output)
0: exchange is from control unit to memory (input).
bits 2 and 3 are 0.
bits 4 to 15: specify the number of characters or words to be transferred.

The format of the second word is:



When operating in word mode the first word of the block is always even (bit 15 = 0). In character mode, when bit 15 = 1, the right hand character is addressed (odd address); when bit 15 = 0, the left hand character is addressed (even address).

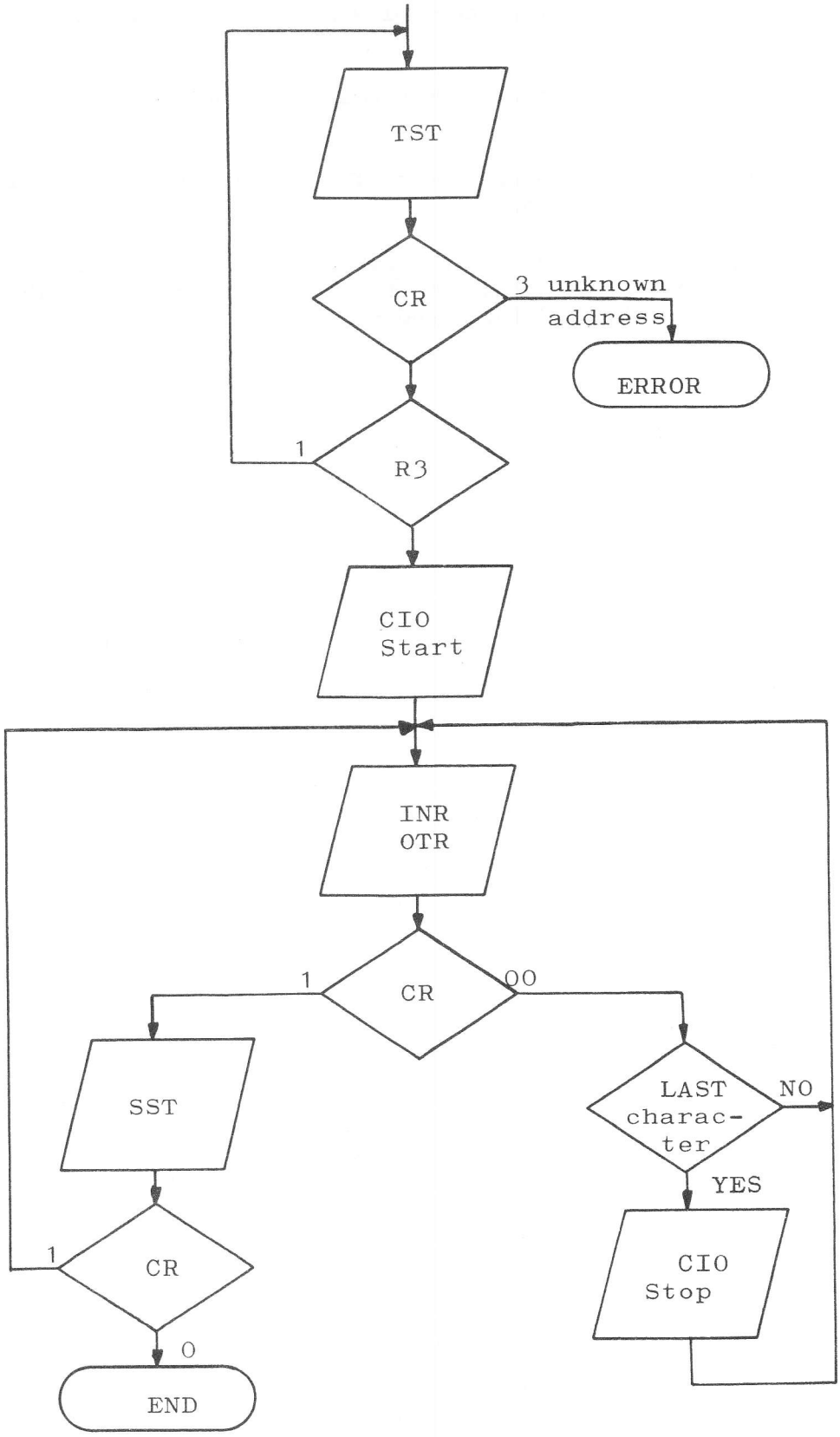
Example:

```
    :
    LDKL      A1,/8032        word mode, input, 50 words
    LDKL      A2,BUF          starting address of block
    WER       A1,/A           send control words (000 1010 and 000 1011)
    WER       A2,/B
    :
    CIO       A4,1,/01        start input (address: 000001)
    :
```

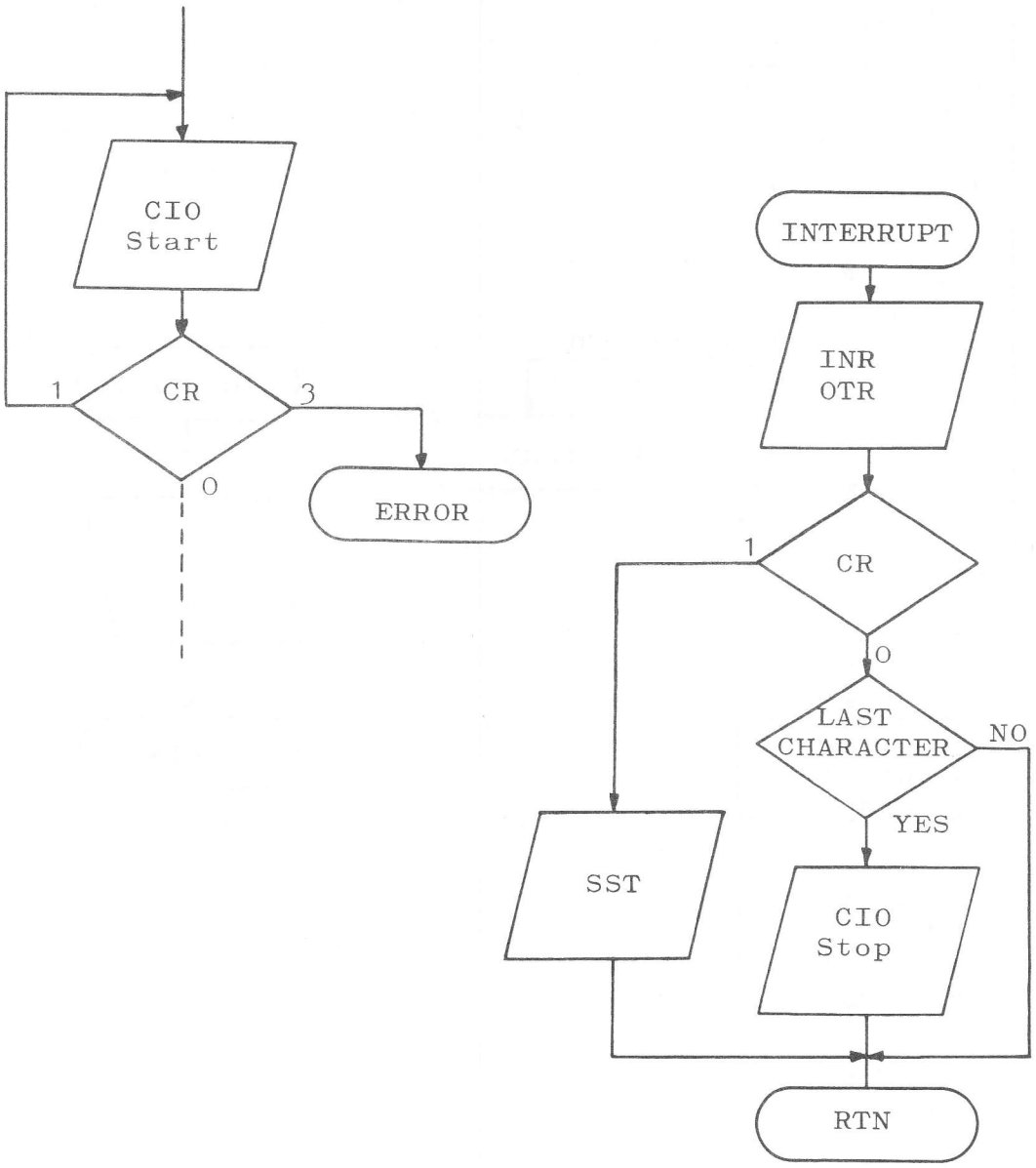
The RER instruction may now be used to read a transfer's effective length after termination of the I/O operation.

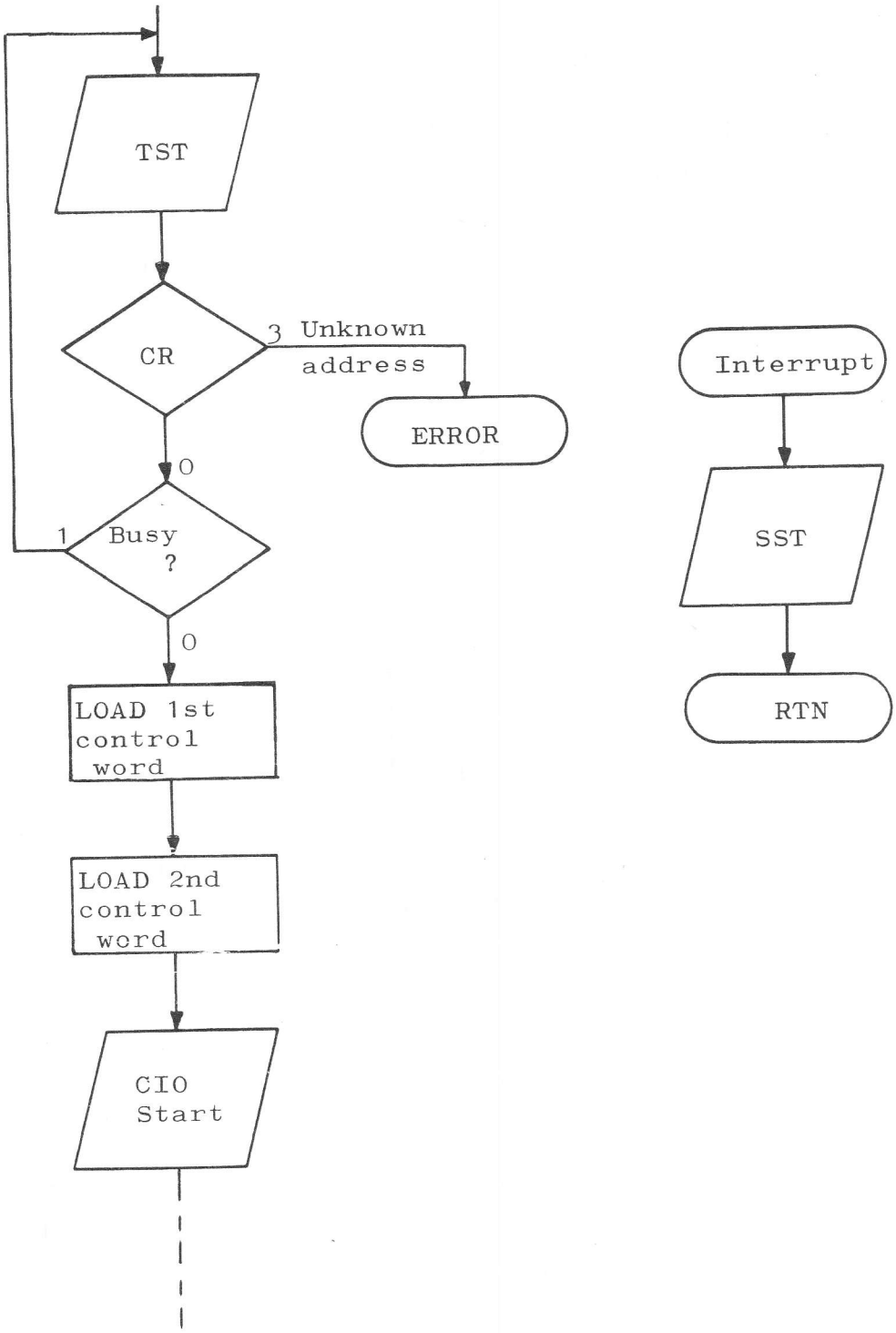
When the exchange is completed, an SST instruction should be issued, to check the status of the control unit and set it to the Inactive state. The control unit may now be re-initialised for a new transfer.

a) Without Interrupts



b) With Interrupt Handling




TST
CR
3 Unknown address
ERROR
0
1 Busy ?
0
LOAD 1st control word
LOAD 2nd control word
CIO Start
Interrupt
SST
RTN

SOURCE PROGRAM CALLING A FORTRAN LIBRARY SUBROUTINE

When writing a program in Assembly Language, it may be useful to have some operation performed by a subroutine, which has been specifically included in the FORTRAN library to execute such a function.

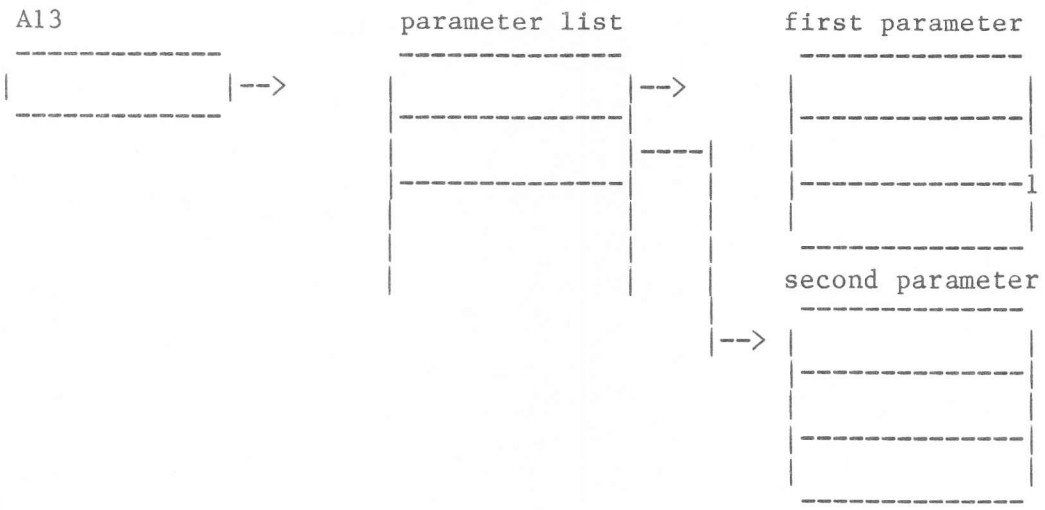
The user may call this subroutine from his Assembly program in the following way:

Suppose the user wishes to multiply two floating point numbers. The FORTRAN library subroutine, which executes this multiplication, has F:RM as entry point.

The framework of the Assembly program, with only the relevant details, is written as follows:

```
          IDENT     ASMPRO
          EXTRN     F:RM
          :
FLNUM1    DATA      -
          DATA      -
          DATA      -
          :
FLNUM2    DATA      -
          DATA      -
          DATA      -
          :
          LDKL      A13,PARLIS
          CF        A14,F:RM
          :
PARLIS    DATA      FLNUM1
          DATA      FLNUM2
```

Before the CF instruction is executed, register A13 must contain the address of a parameter list. This list must contain the addresses of the two floating point numbers to be used as operands.



The subroutine in the library contains the following relevant items:

```
IDENT     FRTLIB
ENTRY     F:RM
:
:
RTN       A14
```

This subroutine does not use the stack of the calling program, except for the return. When values are to be returned to the main program, an integer value will be returned to A1 or a real value to the registers A1 to A3 inclusive (the mantissa in A1 and A2, and the exponent in A3).

The main program must now be Link-Edited with the called subroutine from the FORTRAN library. The Linkage Editor selects those modules required for program execution.