Table of Contents

Absolute addressing	addressing specific locations in memory (see also relocatable addressing)
Assembler	a system program which translates programs written in Assembly Language into binary object code
Bootstrap	a program provided for initial loading of the system
Breakpoint	address at which execution of program stops to allow further debugging
Character	eight bits, representing an integer, letter or other data item
Cluster	a set of data in object code
Common	
blank common	an area to which external references can be made from one or more modules
labeled common	a predefined external reference which can be used in several modules
Debugging Package	a processor which allows the programmer to insert breakpoints in a load module and call debugging functions before execution of a program
Directive	an instruction used for providing a framework for a program or for guiding the assembly process
Effective memory address	address in memory where the actual information can be found
Entry point	a label to which an external reference is made
External reference	a reference to an entry point in another program or module
File code	one or two hexadecimal digits associated with an I/O device

Identifier	a character or a combination of characters used to label an instruction or a value which is to be referred to by other instructions
Internal symbol	identifier in a module
IPL	Initial Program Loader. A program to load the monitor
Label	identifier of max. six characters long, the first always being a letter
Linkage Editor	a processor used to link independent object modules before execution
Load Module	program output by the Linkage Editor containing no external references
Location counter	counter used to assign a relative or absolute address to program elements
Mnemonic	abbreviation for an instruction, as used in the operation code field of a source statement, to indicate a machine instruction or directive
Module	a part of a program, enclosed by an IDENT and END directive, which can be treated independently of the rest of the program
Monitor	a system program which supervises the loading, processing and execution of user programs, starts and supervises the operation of processors and initialises I/O operations
Object code	program as translated by a language translator and suitable to be input to the Linkage Editor
Operand	an expression indicating the address, value or register to be operated upon by the machine instruction
Pass	one program run
Real Time Clock	a mechanism by means of which the amount of computer time allocated to a program is measured and a signal is given when that period of time has ended

Relocatable addressing	addressing in relation to the beginning of a program, not to specific locations in memory. The relocation of the addresses is then done by the machine
Source statement	one line in a source program
Stand Alone processor	processor not running under Monitor control. It contains its own I/O routines
Symbol	an identifier, used as an address value in the operand field of other instructions
Update Package	a processor which handles the additions and deletions in source or object programs

PART 1 ASSEMBLY LANGUAGE

This part contains a description of the Assembly Language. In this description it is made clear how the programmer can write his programs using the instructions of the P800M Instruction Set as well as the directives which guide the assembly process when the program is input to the Assembler. The instruction sets of the P800M series computers are upward compatible.

Programs for the P800M computers are written in a symbolic language closely related to the machine code. Each statement (or line) of the program relates to a single machine instruction or to a data item to be taken into account by an instruction.

To write programs in the Assembly Language, the user should be familiar with the syntax of the instructions, which are devided in the following main groups:

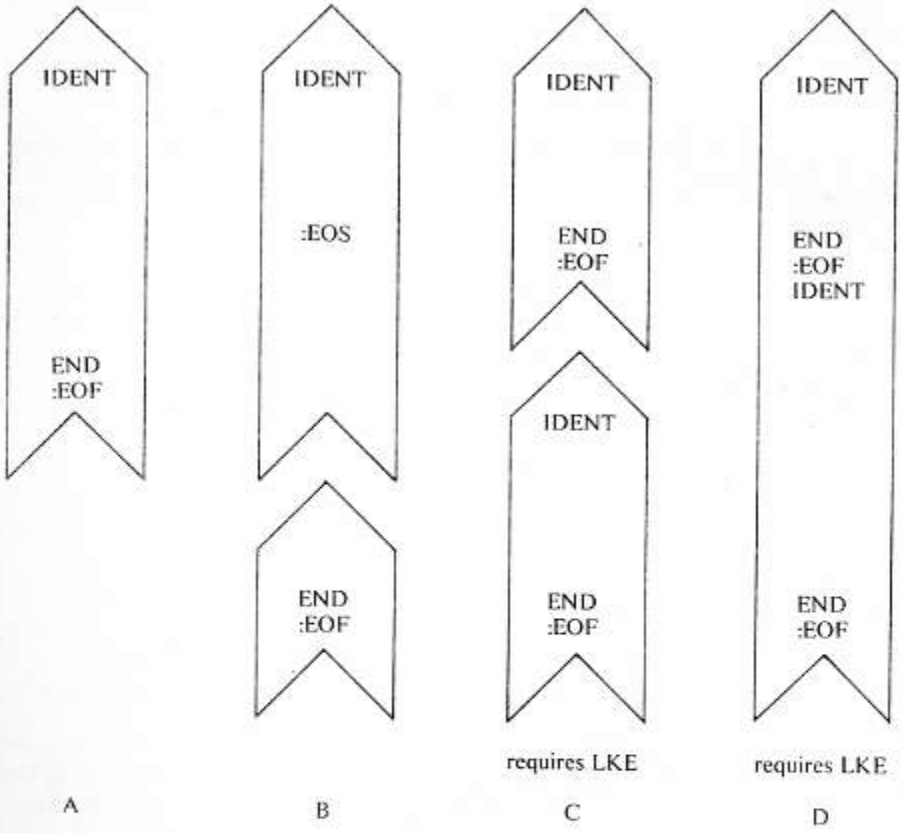
- Load and Store instructions
- Arithmetic instructions
- Logical instructions
- Character handling instructions
- Branch instructions
- Shift instructions
- Control instructions
- Input/Output instructions
- External Transfer instructions
- Move Table instructions.

Programming in Assembly Language requires certain rules to be acceptable to the Assembler.
A source program may consist of one or more modules each of which starts with an identification IDENT and terminates with an END (see directives). The whole source program must be terminated by an "End Of File" mark (:EOF).

NOTE: If a source program consists of several modules the modules need not be separated by :EOF marks but by :EOS marks (End Of Segment). An :EOS mark at the end of a punched tape indicates the physical end of that tape when the program is punched on two tapes. The mark is not part of the Assembly Language.

The following figure shows various possibilities of how programs can be punched on tape.

In example A the program is contained on one punched tape. The program starts with an identification IDENT and is terminated by END which will cause an :EOS to be punched when the program is assembled, and is followed by an :EOF.



Example B is an example of a program punched on two tapes. The first tape starts with an IDENT and is terminated with :EOS which causes the Assembler to wait for operator action. The second tape does not contain an IDENT and is read immediately after the first one. The second tape is terminated by an END and :EOF.

Example C consists of two modules on two tapes both beginning with IDENT and ending with END and :EOF.

Example D consists of several modules punched on one tape. Each module begins with an IDENT and is terminated by END and either an :EOS mark if another module follows this one or by :EOF when it is the last module to be processed. This example requires the Linkage Editor to make of those modules one larger program which can be executed.
Each module of a program consists of a number of characters grouped into lines and each statement in a module is made up of the following characters:

Letters: A to Z inclusive

Digits: 0 to 9 inclusive

Delimiters:

+	plus
—	minus
*	asterisk
=	equal
'	apostrophe
,	comma
	blank
/	slash
(	left parenthesis
)	right parenthesis
.	period
:	colon

Location counter
The Assembler maintains a location counter which is a software counter used to assign a relative or absolute memory address to program elements. The location counter starts with a relative value equal to zero, or it starts at an absolute address defined by the AORG directive, at the beginning of an assembly. The value of the counter is incremented by 2 or a multiple of 2 depending on the kind of instruction given.
The current value of the location counter is referred to by an * in the operand field (see below). In absolute program sections * has an absolute value. In that case the value is incremented in the normal way and 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.

Symbols
A symbol is a character or a string of characters used to represent addresses or values. Symbols may appear in the label field as well as in the operand field of a statement.
Their syntax is the same as for the label (see under label field). Some symbols are predefined and have a special meaning for the Assembler e.g. * indicates the current value of the location counter, P is the instruction counter etc.

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

< >	to enclose syntactic items
|	the vertical stroke has the meaning of or
::=	is composed of
[ ]	the syntactic items between these brackets may be omitted
[ ]	select one of the items between these brackets
␣	space

The following list contains the definition of all items used.

<statement>	::= [<label>]␣<operation code>[<operand>][<comments>] [*<comments>]
<label>	::= <identifier>
<operation code>	::= <mnemonic>[S|(<cnd>)|L] [*]<directive>
<operand>	::= [+|−]<term>[+|−]<-term> [+|−]<term>
<comments>	::= <characters>|*<characters>
<identifier>	::= <letter>|<identifier><letter>| <identifier><digit>|<identifier>
<mnemonic>	::= <letters representing operation code>
<S>	::= <store indicator>
(<cnd>)	::= <numerical condition value>| <condition mnemonic>
<numerical condition value>	::= 0|1|2|3|4|5|6|7
<condition mnemonic>	::= Z|P|N|O|E|G|L|A|U|NA| NR|NZ|NP|NE|NG|NL|NN
<L>	::= < load indicator>
*	::= indirection
<directive>	::= <IDENT, END etc.> see chapter on directives
<DATA defined hexa constant>	::= <see DATA directive>
<module name>	::= <symbol>
<symbol>	::= <characters representing address or value>

<predefined expression>	:: =	<max. of two defined symbols>
<entry point name>	:: =	<identifier within reference module>
<external>	:: =	<identifier defined in other module>
<common-field definition list>	:: =	<commen field definition>,<common field definition> .....
<common field definition>	:: =	<common field name>[<common field length>]
<common field name>	:: =	<identifier>
<common field length>	:: =	predefined (absolute) expression
<internal symbol list>	:: =	<internal symbol>,<internal symbol> .....
<internal symbol>	:: =	<identifier>
<field definition>	:: =	<field length definition>[[=|:]<field value definition>
<field length definition>	:: =	<number of bits>
<=field value definition>	:: =	<value to be placed in field>
<:field value definition>	:: =	<address of word>
<field number>	:: =	<decimal integer>
<term>	:: =	<constant>|<symbol>
<constant>	:: =	<decimal constant>|<hexadecimal constant>|<character constant>
<decimal constant>	:: =	<digit>|<integer>
<hexadecimal constant>	:: =	<hexadecimal integer>
<character constant>	:: =	<letter>|<digit>|<delimeter>
<letter>	:: =	A|B|C|D|E|F|G|H|I|J|K|L|M| N|O|P|Q|R|S|T|U|V|W|X|Y|Z
<digit>	:: =	0|1|2|3|4|5|6|7|8|9
<delimiter>	:: =	+|−|*|=|'|.|␣|/|(|)|.|:
<integer>	:: =	<number>

1 Format of source statements

A source module consists of a sequence of statements. The Assembler interprets each line as it is presented.
Statements can be divided in the following fields:

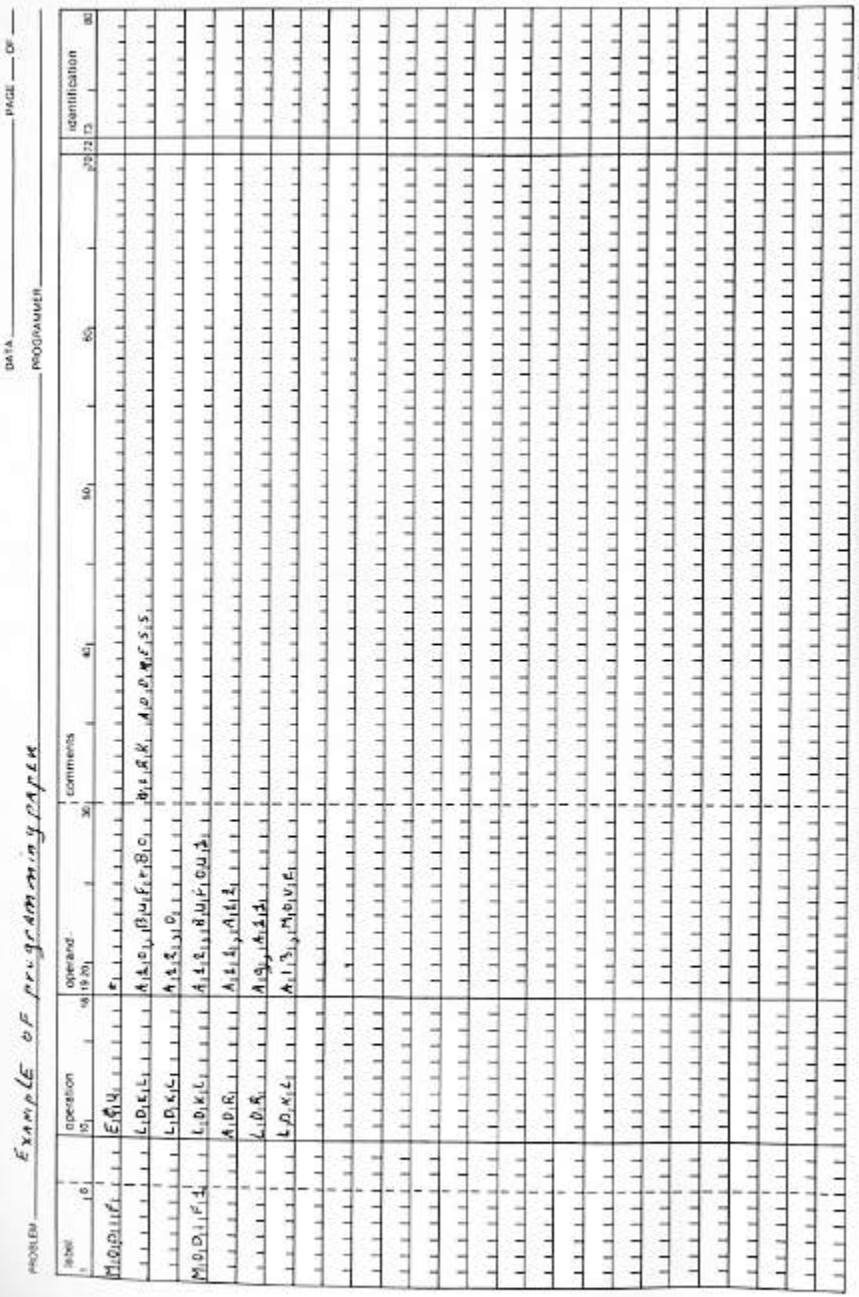
- label field
- operation field
- operand field
- comments field

<statement> :: =[<label>]␣<operation code>␣[<operand>]␣ [<comments>]
*[<comments>]

Each field has to be separated from the other by one (or more) blank character(s). Blanks may not appear in the fields themselves except when specified in a character constant or in a comments field. Instead of blanks a backslash may be used for separation (see page 1-16). One or more blanks at the beginning of a statement indicate that there is no label field.

If there are more than ten blanks after the operation field all following characters are considered to be belonging to the comments field.
An * (asterisk) at the beginning of a statement identifies that line as a comments line.

Statements punched on tape which are to be read by the ASR punched tape reader have to be terminated by LF XOFF CR, which switches the reader off, followed by a Null character, e.g. Rub-out, to allow for a proper reading and processing of the next usable character.



LABEL FIELD

< label > :: = < identifier >
< identifier > :: =
< letter > | < identifier > < letter > | < identifier > < digit > | < identifier >

Labels (or identifiers) in a module are used for reference purpose to other statements in a module.
The Assembler assigns, in most cases, 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, however, 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.

L.A.B.E.L. has the same meaning as LABEL.

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 which is not equated by an EQU directive to a label previously defined as relocatable).

OPERATION FIELD

< operation code > :: = < mnemonic > [S|(< end >)|L][*] < assembly directive >

where:

< **mnemonic** >

The operation field normally contains the mnemonic of a standard instruction. It is possible, however, to generate one's own instruction mnemonic by means of the FORM, XFORM and GEN directives (only with the monitor controlled Assembler).

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 word (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. CIR and CIRS instructions are to be considered as two different instructions.

NOTE: It is allowed to have the S preceded by a period sign though the Assembler does not take this sign into account.
e.g. AD.S␣ = ADS␣

(< **cnd** >):: = < numerical condition value > | < condition mnemonic >
< numerical condition value > :: = 0/1/ . . . . . /7
< condition mnemonic > :: =
Z|P|N|O|E|G|L|A|R|U|NA|NR|NZ|NP|NE|NG|NL|NN

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

. COND. REG CONTENTS	(<CND>)				
	GENERAL	ARITHM.		COMPARE	I/O
0	(0)	(Z) ZERO		(E) EQUAL	(A) ACCEPTED
1	(1)	(P) POS.		(G) GREATER	(R) REFUSED
2	(2)	(N) NEG.		(L) LESS	—
3	(3)	(O) OVERFL.		—	(U) UNKNOWN
		NOT - CONDITION			
≠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 instruction mnemonic of a constant instruction. It specifies that the operand is contained in 16 bits i.e. that the instruction must be assembled as a "long" instruction.

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

OPERAND FIELD

The operand field may contain an address expression, a register expression or constants associated with the current machine instruction or assembly directive or a combination of those.
The structure and meaning of the operand depends on the type of instruction and directive and is explained below.
All operand expressions must be separated by a comma.

Expression

< expression > :: = [+ | −]< term >[[+ | −]< term >[[+ | −]< term >]]
< term > :: = < constant > | < symbol >

NOTE: * is considered to be a symbol.

An expression may not refer to more than 2 symbols and may not refer to more than one register name. In the latter case it may not contain any other term.