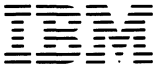


Personal Computer
Computer Language
Series

ASSEMBLER REFERENCE

for the UCSD p-System™ Version IV.0

Produced by SofTech Microsystems, Inc.

First Edition (January 1982)

Changes are periodically made to the information herein; these changes will be incorporated in new editions of this publication.

Products are not stocked at the address below. Requests for copies of this product and for technical information about the system should be made to your authorized IBM Personal Computer Dealer.

A Product Comment Form is provided at the back of this publication. If this form has been removed, address comment to: IBM Corp., Personal Computer, P.O. Box 1328-C, Boca Raton, Florida 33432. IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligations whatever.

PREFACE

This manual describes the UCSD p-System 8086/88/87 Assembler. It also describes the instruction set of the 8086/88 CPUs and the 8087 floating point processor. (This assembler was developed by Softech Microsystems to support these three Intel processors but this does not imply support of the 8087 on all configurations of the IBM Personal Computer.) The p-System Assembler is a powerful tool for creating assembly routines to be run within (or outside of) the UCSD p-System environment.

For a complete understanding of the 8086/88/87 assembly language, the Intel 8086 Family Users' Manual (Intel Corporation, Santa Clara, Calif., 1979) should be used in conjunction with this manual.

Chapter 1 of this manual describes the UCSD p-System assembler itself. It should be noted that the p-System assembler differs substantially from the Intel assembler.

Chapter 2 describes how you can simplify coding, reduce the chance of making errors, and ensure that standard sequences of instructions are coded by using the Assembler Directives.

Chapter 3 gives a brief overview of the 8086/88 CPU. The registers, flags, and addressing modes are discussed. For a more detailed description of the 8086/88 processor see the Intel manual.

Chapter 4 lists the 8086/88 and 8087 operations and gives a brief summary of their actions. Again, for more detailed information, the Intel manual should be referenced. Chapter 4 also describes differences between the standard Intel mnemonics and the mnemonics accepted by the UCSD p-System Assembler.

CONTENTS

NOTES

CHAPTER 1. UCSD p-SYSTEM 8086/88/87 ASSEMBLER

Contents

Introduction

This chapter describes the UCSD p-System 8086/88/87 Assembler. Definitions of technical terms and descriptions of assembler-related concepts are given. The assembler directives are detailed. Linking of assembled routines with host compilation units is described. Various other issues are addressed, such as assembled listings, error messages, and sharing of machine resources with the Interpreter.

Assembly Language Definition

An assembly language consists of symbolic names which can represent machine instructions, memory addresses, or program data. The main advantage of assembly language programming over machine coding is that programs can be organized in a more readable and hence easier to understand fashion.

An assembly language program (called source code) is translated by an assembler into a sequence of machine instructions (called object code). Assemblers can create either relocatable or absolute object code. Relocatable code includes information that allows a loader to place it in any available area of memory, while absolute code must be loaded into a specific area of memory. Symbolic addresses in programs that are assembled to relocatable object code are called relocatable addresses.

Assembly Language Applications

Users of the UCSD Pascal system are interested in developing assembly language programs for one of two purposes:

1) assembly language procedures running under the control of a host program in Pascal or FORTRAN.

2) stand-alone assembly language programs for use outside of the operating system's environment.

The UCSD p-System 8086/88/87 Assembler, in conjunction with the system linker and some support programs, has been designed to meet these needs.

General Programming Information Object Code Format

Byte Organization

A byte consists of eight bits. The bits may represent eight binary values, or a single character of data. The bits may also represent a one byte machine instruction or a number which is interpreted either as a signed two's complement number in the range of -128 to 127 or an unsigned number in the range of 0 to 255.

Word Organization

A word consists of sixteen bits, or two adjacent bytes in memory. A word may contain a one word machine instruction, any combination of byte quantities, or a number which may be interpreted either as a signed two's complement number in the range of -32,768 to 32,767 or an unsigned number in the range of 0 to 65,535.

Source Code Format

Character Set

The following characters are used to construct source code:

- upper and lower case alphabetic: A..Z, a..z
- numerals: 0..9
- special symbols: | @ # $ % ^ & * ( ) < > ~ [ ] . , / ; : " ' + - = ? _
- space (' ') character and tab character

Identifiers

Identifiers consist of an alphabetic character followed by a series of alphanumeric characters and/or underscore characters. The underscore character is not significant. This definition of identifiers is equivalent to the Pascal definition.

Identifiers are used in:

- label and constant definitions.
- machine instructions, assembler directives, and macro identifiers.
- label and constant references.

Example:

```
FormArray
FORM_ARRAY
formarray
```

... all denote the same item.

Predefined Symbols and Identifiers

Predefined identifiers are reserved by the assembler as symbolic names for machine instructions and registers. They may not be used as names for labels, constants, or procedures. Also, the dollar sign, "$", is predefined to specify the location counter. When used in an expression, the dollar sign represents the current value of the location counter in the program.

Character Strings

A character string is written as a series of ASCII characters delimited by double quotes. A string may contain up to eighty characters, but cannot cross source lines. A double quote may be embedded in a character string by entering it twice, for example, "This contains ""embedded"" double quotes". The assembler directive .ASCII requires a character string for its operand. Strings also have limited uses in expressions.

Constants

Binary Integer Constants

A binary integer constant is a series of bits or binary digits (0..1) followed by the letter T. The range of values is 0 to 11111111, or 0 to 1111, if a byte constant.

Examples: `0T` `01000100T` `11101T`

Decimal Integer Constants

A decimal integer word constant is written as a series of numerals (0..9) followed by a period. Its range of values is -32768 to 32767 as a signed two's complement number. As a byte constant, its range of values is -128 to 127 as a signed two's complement number or 0 to 255 as an unsigned number.

Examples:
```
000.    256.    -4096.
```

Hexadecimal Integer Constants

A hexadecimal integer word constant is written as a series of up to four significant hexadecimal numerals (0..9, A..F) followed by the letter H. The leading numeral of a hex constant must be a numeric character. The range of values is 0 to FFFF. These are examples of valid hex constants:

Examples:
```
0AH
100H
0FFFEH ; leading zero is required here
```

Byte constants possess similar syntax, but can have at most two significant hex numerals, with a range of 0 to FF.

Octal Integer Constants

An octal integer word constant is written as a series of up to six significant octal numerals (0..7) followed by the letter Q. Its range of values is 0 to 177777. Byte constants can have at most three significant octal numerals, with a range of 0 to 477.

Examples:
```
170     4570     177776Q
```

Default Radix Integer Constants

The radix of an integer constant lacking a trailing radix character is decimal on the p-System 8086/87 Assembler.

Character Constants

Character constants are special cases of character strings and may be used in expressions. The maximum length is two characters for a word constant, and one character for a byte constant.

Examples: **"A"** **"BC"** **"YA"**

Assembly Time Constants

An assembly time constant is written as an identifier that has been assigned a constant value by the .EQU directive (see "Data and Constant Definition Directives" in this chapter). Its value is completely determined at assembly time from the expression following the directive. Assembly time constants must be defined before they may be referenced.

Expressions

Expressions are used as symbolic operands for machine instructions and assembler directives. An expression can be:

- a label, which might refer to a defined address or an address further down in the source code (implying that the label is presently undefined), an externally referenced address, or an absolute address.
- a constant.
- a series of labels or constants separated by arithmetic or logical operators.
- the null expression, which evaluates to a constant of value 0.

Relocatable and Absolute Expressions

An expression containing more than one label is valid only if the number of relocatable labels added to the expression exceeds the number of relocatable labels subtracted from the expression by zero or one. The expression result is absolute if the difference is zero, and relocatable if the difference is one. Subexpressions that evaluate to relocatable quantities may not be used as arguments to a multiplication, division, or logical operation. Unary operators may not be applied to relocatable quantities.

In relocatable programs, absolute expressions may not be used as operands of instructions which require location-counter-relative address modes.

Linking and One Pass Restrictions

An expression may contain no more than one externally defined label, and its value must be added to the expression. An expression containing an external reference may not contain a forward referenced label, and the relocation sum of any other relocatable labels in the expression must be equal to zero.

An expression may contain no more than one forward referenced identifier. A forward referenced identifier is assumed to be a relocatable label defined further down in the source code; any other identifiers must be defined before they are used in an expression. An expression containing a forward referenced label may not contain an externally defined label.

Arithmetic & Logical Operators

The following operators are available for use in expressions:

unary operations:

+	plus
-	minus (two's complement negation)
~	logical not (one's complement negation)

binary operations:

+	plus
-	minus
^	exclusive or
*	multiplication
/	signed integer division (DIV)
//	unsigned integer division (DIV)
%	unsigned remainder division (MOD)
|	bitwise OR
&	bitwise AND

The following operators are available for use only with conditional assembly directives:

```
=    equal
<>   not equal
```

The following symbols may be used as alternatives to the single character definitions presented above. Occurrences of these alternative definitions require at least single blank characters as delimiters.

```
.OR    =   |
.AND   =   &
.NOT   =   ~
.XOR   =   ~
.MOD   =   %
```

The assembler performs left to right evaluation of expressions; there is no operator precedence. All operations are performed on word quantities. Usage of unary operators is limited to constants and absolute addresses. Angle brackets must enclose subexpressions which contain embedded unary operators.

Subexpression Grouping

Angle brackets (< and >) may be used in expressions to override the left to right evaluation of operands. Subexpressions enclosed in angle brackets are completely evaluated before inclusion in the rest of the expression.

The following are examples of valid expressions. The default radix is decimal.

Examples:

```
MARK+4              ; The sum of the value of
                    ; identifier MARK plus 4

BILL-2              ; The result of subtracting 2 from
                    ; the value of identifier BILL

2-BARRY             ; The result of subtracting the
                    ; value of identifier BARRY from
                    ; 2. BARRY must be absolute.

3*2+MACRO           ; The sum of the value of
                    ; identifier MACRO plus the
                    ; product of 3 times 2.

DAVID+3*2           ; 2 times the sum of the
                    ; identifier DAVID and 3. DAVID
                    ; must be absolute.

650/2-RICH          ; The result of dividing 650 by 2
                    ; and subtracting the value of
                    ; identifier RICH from the
                    ; quotient. RICH must be
                    ; absolute.
                    ; Null expression: constant 0

-4*12+<6/2>         ; evaluates to -45 (decimal)

85+2+<-5>           ; evaluates to 82 (decimal)

0!1&{~0}            ; evaluates to 1

0 .OR 1 .AND <.NOT 0>   ; is the same
                    ; expression (result is 1)
```

Source Statement Format

An assembly language source program consists of source statements which may contain machine instructions, assembler directives, comments, or nothing (a blank line). Each source statement is defined as one line of a textfile. Assembly language identifiers are restricted to upper case alphabetic characters, but lower case characters may be used in the comment field.

Label Field

The assembler supports the use of both standard labels and local (i.e., reuseable) labels. The label field begins in the leftmost character position of each source line. Macro identifiers and machine instructions must not appear in the start of the label field, but assembler directives and comments may appear there.

Standard label usage

A standard label is an identifier that appears in the label field of a source statement. It may be terminated by an optional colon character, which is not used when refrencing the label. As in Pascal, only the first eight characters of the label are important; the rest are ignored by the assembler. As in Pascal, the underscore character is not significant.

Example:

```
BIOS
L3456:              ; referenced as L3456
The_Kind
LONG_label          ; last character is ignored
```

A standard label is a symbolic name for a unique address or constant; it may be declared only once in a

source program. A label is optional for machine instructions and for many of the assembler directives. A source statement consisting of only a label is a valid statement; it has the effect of assigning the current value of the location counter to the label. This is equivalent to placing the label in the label field of the next source statement that generates object code. Labels defined in the label field of the .EQU directive (see "Data and Constant Definition Directives" in this chapter) are assigned the value of the expression in the operand field.

Local Label Usage

Local labels allow source statements to be labeled for reference by other instructions without taking up storage space in the symbol table. They can contribute to the cleanliness of source program design by allowing the creation of nonmnemonic labels for use by iterative and decision constructs, thus reserving the use of mnemonic label names for demarking conceptually more important sections of code.

Local labels must have $ in the first character position; the remaining characters must be digits. As in regular labels, only the first eight digits are significant. The scope of a local label is limited to the lines of source statements between the declaration of consecutive standard labels; thus, the jump to label $4 in the following example is illegal:

Example:

```
LABEL1
        ADC     AX, SI
$3      MOV     MEM, AX
        JC      $3      ; legal
        NOP     $4      ; illegal
        JNC
LABEL2
        ADC     AX, SI
$4      MOV     MEM, AX
```

Up to 21 local labels may be defined between 2 occurrences of a standard label. On encountering a standard label, the assembler purges all existing local label definitions; hence, all local label names may be redefined after that point. Local labels may not be used in the label field of the .EQU directive (see "Data and Constant Definition Directives" in this chapter).

Opcode Field

The opcode field begins with the first non-blank character following the label field, or with the first nonblank character following the leftmost character position when the label is omitted. It is terminated by one or more blanks. The opcode field contains an identifier which can be of the following types:

- machine instruction
- assembler directive
- macro call

Operand Field

The operand field begins with the first nonblank character following the opcode field, and is terminated by zero or more blanks. It can contain zero or more expressions, depending on the requirements of the preceding opcode.

Comment Field

The comment field can be preceded by zero or more blanks, begins with a semicolon (;), and extends to the end of the current source line. It may contain any printable ASCII characters. The comment field is listed on assembled listings, and has no other effect on the assembly process.

Source File Format

Assembly source files are generated using the system editor and saved as files of type TEXT. A source file is constructed from the following entities:

- assembly routines (procedures and functions).
- global declarations.

Assembly Routines

A source file may contain more than one assembly routine; in this case, a routine ends upon the occurrence in the source code of another program delimiting directive (e.g., the start of the following routine). Each routine in a source file is a separate entity; it contains its own relocation information and may be individually referenced by a Pascal host program during linking.

Assembly routines must begin with a .PROC, .FUNC, .RELPROC, or .RELFUNC directive. The last routine in the source file must be terminated by the .END directive. "Program Linking and Relocation" in this chapter gives a detailed description of these directives.

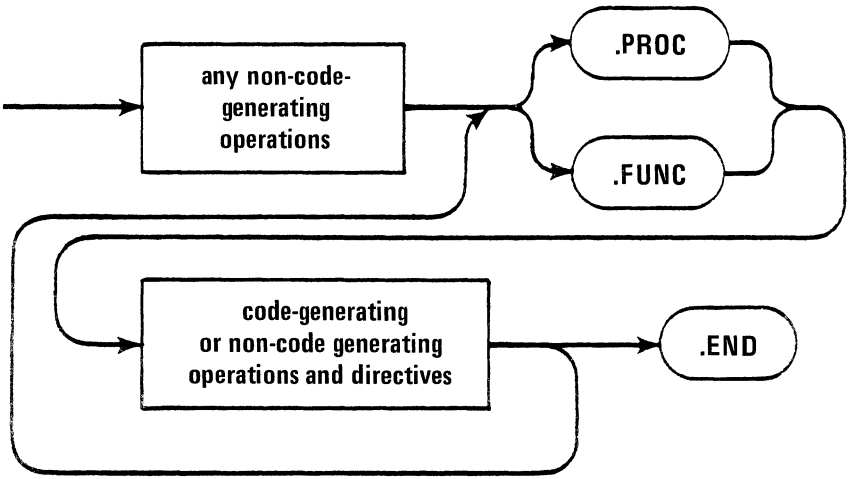
At the end of each routine, the assembler's symbol table is cleared of all but predefined and globally declared symbols, and the location counter (LC) is reset to zero.

Global Declarations

An assembly routine may not directly access objects declared in another assembly routine, even if the routines are assembled in the same source file; however, occasions arise when it is desirable for a set of routines to share a common group of

declarations. Therefore, the assembler allows global data declarations.

Any objects declared before the first occurrence of a .PROC or .FUNC directive in a source file may be referenced by all subsequent assembly routines. No code may be generated before the first procedure delimiting directive; hence, the global objectives are limited to the non-code-generating directives (.EQU, .REF, .DEF, .MACRO, .LIST, etc.).



Absolute Sections

Assembly language programmers often find it necessary to access absolute addresses in memory, regardless of where an assembly routine is loaded in memory. For instance, a program may need to access ROM routines. Absolute sections allow the user to define labels and data space using the standard syntax and directives, but with the extra ability to specify absolute (nonrelocatable) label addresses starting at any location in memory.

Absolute sections are initiated by the directive .ASECT (for absolute section) and terminated by the directive .PSECT (for program section, which is the default setting during assembly). When the .ASECT

directive is encountered, the absolute section location counter (ALC) becomes the current location counter. The .ORG directive can be used to set the ALC to any desired value. Label definitions are nonrelocatable and are assigned the current value of the ALC. The data directives .WORD, .BLOCK, and .BYTE cause the ALC (instead of the regular LC) to be incremented.

Data directives in an absolute section cannot place initial values in the locations specified as they can when used in the program section; thus, the absolute section serves as a tool for constructing a template of label-memory address assignments.

The equate directive (.EQU) may be used in an absolute section, but the labels are restricted to being equated only to absolute expressions. The only other directives allowed to occur within an absolute section are .LIST, .NOLIST, .END, and the conditional assembly directives.

Absolute sections may appear as global objects.

The following is a simple example of an absolute section:

```
.ASECT      ; start absolute section

.ORG ODFOOH     ; set ALC to DF00 hex

    ; note - no data values assigned
    ; label assignments below

DKSOUT      .BYTE       ; DSKOUT = DF00
DSKSTAT     .BYTE       ; DSKSTAT = DF01
CONS        .WORD       ; CONS = DF02
BLAGUE      .BLOCK 4    ; BLAGUE = DF04
                        ;    (4 bytes)
REMOUT      .WORD       ; REMOUT = DF08
OFFSET      .EQU        ; REMOUT + 2
                        ; OFFSET = DF0A

.PSECT
```

CHAPTER 2. ASSEMBLER DIRECTIVES

Contents

NOTES

Assembler directives (sometimes referred to as pseudo-ops) enable the programmer to supply data to be included in the program and exercise control over the assembly process. Assembler directives appear in the source code as predefined identifiers preceded by a period (.).

Metasymbols

The following metasymbols are used below in the syntax definitions for assembler directives:

- special characters and items in capital letters must be entered as shown.
- items that are in italics are defined by the user.
- items within square brackets ([]) are optional.
- the word 'or' indicates a choice between two items.
- items in lower case letters are generic names for classes of items.

The following terms are names for classes of items:

b = the occurrence of one or more blanks.

integer = any legal integer constant as defined in "Constants" in Chapter 1.

label = any legal label as defined in "Label Field" in Chapter 1.

expression = any legal expression as defined in "Expressions" in Chapter 1.

value = any label, constant, or expression. Its default value is 0.

value list = a list of zero or more values delimited by commas.

identifier = a legal identifier as defined in "Identifiers" in Chapter 1.

idlist = a list of one or more identifiers delimited by commas.

id:integer list = a list of one or more identifier-integer pairs separated by a colon and delimited by a comma. The colon:integer part is optional; its default value is 1.

comment = any legal comment as defined in "Comment Field" in Chapter 1.

character string = any legal character string as defined in "Character Strings" in Chapter 1.

file identifier = any legal name for a Pascal text file.

Example: [*label*] [b] .ASCII b *character string* [*comment*]

... indicates that a label may be included in the label field (but is not necessary), and that a character string must be included as an operand. These formal definitions should be thought of as being on one line even though they may appear on more than one line in this manual (because of textual space considerations).

Small examples are included after each definition to supply the user with a reference to the specific syntax of the directive.

Procedure-Delimiting Directives

Every source program (including those intended for use as stand-alone code files) must contain at least one set of procedure-delimiting directives. The most frequent use of the assembler is in assembling small routines intended to be linked with a host compilation unit. The directives .PROC and .FUNC identify and delimit assembly language procedures. .RELPROC and .RELFUNC identify and delimit dynamically relocatable procedures. Dynamically relocatable procedures may reside in the code pool, and are subject to more of the System's memory management strategies. "Program Linking and Relocation" in this chapter has a more detailed description of the use of these directives.

PROC

Identifies the beginning of an assembly language procedure. The procedure is terminated by the occurrence of another delimiting directive in the source file.

FORMAT: [b] .PROC b *identifier* [,*integer*] [*comment*]

identifier is the name associated with the assembly procedure.

integer indicates the number of words of parameters passed to this routine. The default is 0.

Example:

```
.PROC  DLDRIVE,2
```

Procedure-Delimiting Directives

.FUNC

Identifies the beginning of an assembly language function, which is expected (by the host compilation unit) to return a function result on top of the stack; otherwise, equivalent to the .PROC directive.

Format: [b] .FUNC b *identifier* [,*integer*] [*comment*]

identifier is the name associated with the assembly procedure.

integer indicates the number of words of parameters passed to this routine. The default is 0.

Example:

```
.FUNC RANDOM
```

.RELPROC

Identifies the begininng of a dynamically relocatable assembly language procedure. Such assembly procedures must be position-independent (see "Program Linking and Relocation" in this chapter). The procedure is terminated by the occurrence of another delimiting directive in the source file.

Format: [b] .RELPROC b *identifier* [. *integer*] [*comment*]

identifier is the name associated with the assembly procedure.

integer indicates the number of words of parameters passed to this routine. The default is 0.

Example:

```
.RELPROC POOF,3
```

.RELFUNC

Identifies the beginning of a dynamically relocatable assembly language function which is expected (by the host compilation unit) to return a function result on the stack; otherwise, equivalent to the .RELPROC directive.

Format: [b] .RELFUNC b *identifier* [, *integer*] [*comment*]

identifier is the name associated with the assembly function.

integer indicates the number of words of parameters passed to this routine. The default is 0.

Example:

```
.RELFUNC  POOOF
```

.END

Marks the end of an assembly source file.

Format: [*label*] [b] .END

Data and Constant Definition Directives

.ASCII

Converts character strings to a series of ASCII byte constants in memory. The bytes are allocated in the order that they appear in the string. An identifier in the label field is assigned the location of the first character allocated in memory.

Format: [*label*] [b] .ASCII b *character string* [*comment*]

character string is any string of printable ASCII characters delimited by double quotes.

Example:

```
.ASCII    "HELLO"
```

.BYTE

Allocates and initializes values in one or more bytes of memory. Values must be absolute byte quantities. The default value is zero. An identifier in the label field is assigned the location of the first byte allocated in memory.

Format: [*label*] [b] .BYTE b [valuelist] [*comment*]

Example:

```
TEMP .BYTE 4; code would
             ; be: 04 hex

TEMP1 .BYTE ; code would
             ; be: 00 hex
```

.BLOCK

Allocates and initializes a block of consecutive bytes in memory. A byte value must be an absolute quantity. The default value is zero. An identifier in the label field is assigned the location of the first byte/word allocated.

Format: [*label*] [b] .BLOCK b *length* [, *value*] [*comment*]

length is the number of bytes to allocate with the initial value *value.*

Example:

```
TEMP .BLOCK  4,6H
```

The output code would be:

```
06 06 06 06 ;four bytes with value 06 hex
```

Data and Constant Definition Directives

.WORD

Allocates and initializes values in one or more consecutive words of memory. Values may be relocatable quantities. The default value is zero. An identifier in the label field is assigned the location of the first word allocated.

Format: [*label*] [b] .WORD b *valuelist* [*comment*]

Example:

```
TEMP .WORD  0,2,,4
```

The output code would be:

```
0000
0002
0000   ; this is a default value.
0004

L1  .WORD  L2
```

The output code would be a word containing the address of the label L2.

.EQU

Equates a value to a label. Labels may be equated to an expression containing relocatable labels, externally referenced labels, and/or absolute constants. The general rule is that labels equated to values must be defined before use. The exception to this rule is for labels equated to expressions containing another label. Local labels may not appear in the label field of an equate statement.

Format: *label* [b] .EQU b *value* [*comment*]

Example:

```
BASE    .EQU    R6
```

Location Counter Modification Directives

These directives affect the value of the location counter (LC or ALC) and the location in memory of the code being generated.

.ORG

If used at the beginning of an absolute assembly program, .ORG initializes the location counter to *value.* Used anywhere else, .ORG will generate zero bytes until the value of the location counter equals *value.*

Format: [b] .ORG b *value* [*comment*]

Example:

```
.ORG    1000H
```

.ALIGN

Outputs sufficient zero bytes to set the location counter to a value which is a multiple of the operand value.

Format: [b] .ALIGN b *value* [*comment*]

Example:

```
.ALIGN  2
```

This aligns the LC to a word boundary.

These directives allow the user to exercise control over the format of the assembled listing file generated by the assembler. No code is generated by these directives, and their source lines do not appear on assembled listings. See "Assembler Output" in this chapter for a more detailed description of an assembled listing.

.TITLE

Changes the title printed on the top of each page of the assembled listing. The title may be up to 80 characters long. The assembler will change the title to SYMBOLTABLE DUMP when printing a symbol table; the title reverts back to its former value after the symbol table is printed. The default value for the title is ' '.

Format: [b] .TITLE b *character string* [*comment*]

Example:

```
.TITLE "MACROS"
```

Listing Control Directives

.ASCIILIST

Print all bytes generated by the .ASCII directive in the code field of the list file, creating multiple lines in the list file if necessary. Assembly begins with an implicit .ASCIILIST directive.

Format: [b] .ASCIILIST [*comment*]

Example: **.ASCIILIST**

.NOASCIILIST

Limit the printing of data generated by the .ASCII directive to as many bytes as will fit in the code field of one line in the list file.

Format: [b] .NOASCIILIST [*comment*]

Example: **.NOASCIILIST**

.CONDLIST

List source code contained in the unassembled sections of conditional assembly directives.

Format: [b] .CONDLIST [*comment*]

Example: **.CONDLIST**

.NOCONDLIST

Suppress the listing of source code contained in the unassembled sections of conditional assembly directives. Assembly begins with an implicit .NOCONDLIST directive.

Format: [b] .NOCONDLIST [*comment*]

Example:

```
.NOCONDLIST
```

.NOSYMTABLE

Suppress the printing of a symbol table after each assembly routine in an assembled listing.

Format: [b] .NOSYMTABLE [*comment*]

Example:

```
.NOSYMTABLE
```

.PAGEHEIGHT

Control the number of lines printed in an assembled listing between page breaks. Assembly begins with an implicit .PAGEHEIGHT 59 directive.

Format: [b] .PAGEHEIGHT *integer* [*comment*]

Example:

```
.PAGEHEIGHT
```

.NARROWPAGE

Limit the width of an assembled listing to 80 columns. The symbol table is printed in a narrow format, source lines are truncated to a maximum of 49 characters, and title lines on the page headers are truncated to a maximum of 40 characters.

Format: [b] .NARROWPAGE [*comment*]

Example:

```
.NARROWPAGE
```

.PAGE

Continue the assembled listing on the next page by sending an ASCII form feed character to the assembled listing.

Format: [b] .PAGE

Example:

```
.PAGE
```

.LIST

Enables output to the list file, if a listing is not already being generated. .LIST and .NOLIST can be used to examine certain sections of source and object code without creating an assembled listing of the entire program. Assembly begins with an implicit .LIST directive.

Format: [b] .LIST

Example:

```
.LIST
```

.NOLIST

Suppresses output to the list file, if it is not already off.

Format: [b] .NOLIST

Example:

```
.NOLIST
```

.MACROLIST

Specifies that all following macro definitions will have their macro bodies printed when they are invoked in the source program. Assembly begins with an implicit .MACROLIST directive. "Macro Language" in this chapter has a detailed description of macro language.

Format: [b] .MACROLIST

Example:

```
.MACROLIST
```

.NOMACROLIST

Specifies that all following macro definitions will not have their macro bodies printed when they are invoked in the source program. Only the macro identifier and parameter list are included in the listing.

Format: [b] .NOMACROLIST

Example:

```
.NOMACROLIST
```

.PATCHLIST

List occurrences of all back patches of forward referenced labels in the list file. Assembly begins with an implicit .PATCHLIST directive. "Assembler Output" in this chapter has a detailed description of back patches.

Format: [b] .PATCHLIST

Example:

```
.PATCHLIST
```

.NOPATCHLIST

Suppress the listing of back patches of forward references.

Format: [b] .NOPATCHLIST

Example:

```
.NOPATCHLIST
```

Program Linkage Directives

Linking directives enable communication between separately assembled and/or compiled programs. "Program Linking and Relocation" in this chapter has a detailed description of program linking.

.CONST

Allows access to globally declared constants in the host compilation unit by the assembly procedure.

Format: [b] .CONST b *idlist* [*comment*]

Each *id* is the name of a global constant declared in the Pascal host.

Example:

```
.CONST   LENGTH
```

.PUBLIC

Allows variables declared in the global data segment of the host compilation unit to be referenced by an assembly language routine.

Format: [b] .PUBLIC b *idlist* [*comment*]

Each *id* is the name of a global variable declared in the Pascal host.

Example:

```
.PUBLIC  I,J,LENGTH
```

.PRIVATE

Allows an assembly language routine to store variables in the global data segment of the host compilation unit that are accessible only to the assembly language routine.

Format: [b] .PRIVATE b *id:integer list* [*comment*]

Example:

```
.PRIVATE  PRINT,BARRAY:9
```

Each *id* is treated as a label defined in the source code. *integer* determines the number of words of space allocated for *id.*

.INTERP

Allows an assembly language procedure to access code or data in the P-code interpreter. .INTERP is a predefined symbol for a processor dependent location in the resident interpreter code; offsets from this base location may be used to access any code in the interpreter. Correct usage of this feature requires a knowledge of the interpreter's jump vector for this location. Its domain is generally restricted to systems applications.

Format: valid when used in *expression*

Example:

```
ERR .EQU 12
  ; hypothetical
  ; routine offset

BOMB .EQU .INTERP+ERR
    JMP   BOMBINT
```

Program Linkage Directives

.REF

Provides access to one or more labels defined in other assembly language routines.

Format: [b] .REF *idlist* [*comment*]

Example:

```
.REF    SCHLUMP
```

.DEF

Makes one or more labels to be defined in the current routine available to other assembly language routines for reference.

Format: [b] .DEF *idlist* [*comment*]

Example:

```
.DEF    FOON,YEEN
```

"Conditional Assembly" in this chapter has a detailed description of conditional assembly features.

.IF

Marks the start of a conditional section of source statements.

Format: [b] .IF b *expression* [= or <> *expression*] [*comment*]

Example:

```
.IF DEBUG
```

.ENDC

Marks the end of a conditional section of source statements.

Format: [b] .ENDC [*comment*]

Example:

```
.ENDC
```

.ELSE

Marks the start of an alternative section of source statements.

Format: [b] .ELSE [*comment*]

Example:

```
.ELSE
```

"Macro Language" in this chapter has a detailed description of macro language.

.MACRO

Indicates the start of a macro definition.

Format: [b] .MACRO b *identifier* [*comment*]

identifier is used to invoke the macro being defined.

Example:

```
.MACRO  ADDWORDS
```

.ENDM

Marks the end of a macro definition.

Format: [b] .ENDM [comment]

Example:

```
.ENDM
```

.INCLUDE

Causes the assembler to start assembling the file named as an argument of the directive; when the end of this file is reached, assembling resumes with the source code that follows the directive in the original file. This feature is useful for including a file of macro definitions or for splitting up a source program too large to be edited as a single text file. .INCLUDE may not be used in an included source file (i.e., nested use of the directive) and may not be used in a macro definition.

Format: [b] .INCLUDE b*file identifier* [*comment*]

The comment field of the .INCLUDE directive must be separated from the file identifier by at least one blank character.

Example:

```
.INCLUDE MYDISK:MACROS
```

.ABSOLUTE

Causes the following assembly routine to be assembled without relocation information. Labels become absolute addresses and label arithmetic is allowed in expressions. Usage is valid only before the occurrence of the first procedure delimiting directive. .ABSOLUTE must not be used when creating a Pascal external procedure. "Program Linking and Relocation" has a detailed description of absolute code files.

Format: [b] .ABSOLUTE [*comment*]

Example:

```
.ABSOLUTE
```

.ASECT

Specifies the start of an absolute section. "Absolute Sections" in Chapter 1 has a detailed description of .ASECT.

Format: [b] .ASECT [*comment*]

Example:

```
.ASECT
```

.PSECT

Specifies the start of a program section, and is used to terminate an absolute section. "Absolute Sections" in Chapter 1 has a detailed description of .PSECT.

Format: [b] .PSECT [*comment*]

Example:

```
.PSECT
```

.RADIX

Sets the current default radix to the value of the operand. Allowable operands are: 2 (binary), 8 (octal), 10 (decimal), and 16 (hexadecimal). "Constants" in Chapter 1 has a detailed description of radices. Initial defaults for each assembler version are listed in "Sharing Machine Resources with Interpreter".

Format: [b] .RADIX *integer* [*comment*]

Example:

```
.RADIX 10
                ; decimal
                ; default radix
```

Conditional Assembly

Conditional assembly directives are used to selectively exclude or include sections of source code at assembly time. Conditional sections are initiated with the .IF directive and terminated with the .ENDC directive, and may contain the .ELSE directive. Control over the inclusion of conditional sections is determined by the use of conditional expressions. Conditional sections may contain other conditional sections.

When the assembler encounters an .IF directive, it evaluates the associated expression to determine the condition value. If the condition value is false, the source statements following the directive are discarded until a matching .ENDC or .ELSE is reached. If the .ELSE directive is used in a conditional section, source code before the .ELSE is assembled if the condition is true, and source code after the .ELSE is assembled if the condition is false.

Overall syntax for a conditional section (using the metalanguage described in "Assembler Directives" in this chapter is as follows:

```
.IF conditional expression
        source statements
[.ELSE
        source statements]
.ENDC
```

Conditional Expressions

A conditional expression can take one of two forms: a single expression, or comparison of two character strings or expressions. The first form is considered false if it evaluates to zero; otherwise, it is considered true. The second form of conditional expression is comparison for equality or inequality (indicated by the symbols = and <>, respectively).

Example:

```
.IF LABEL1-LABEL2 ; arithmetic expression

        ; This code is assembled only if
        ; difference is zero

.IF %1="STUFF" ; comparison expression

        ; This code is assembled only if
        ; outer condition is true and
        ; text of first macro parameter
        ; is equal to "STUFF".

.ENDC ; terminate nested section

        ; This code is assembled if outer
        ; condition is true

.ELSE

        ; This code is assembled if first
        ; condition is false

.ENDC ; terminate outer section
```

The assembler supports the use of a macro language in source programs. A macro language allows the programmer to associate a set of source statements with an identifying symbol; when the assembler encounters this symbol (known as a macro identifier) in the source code, it substitutes the corresponding set of source statements (known as the macro body) for the macro identifier, and assembles the macro body as if it had been included directly in the source program. A carefully designed set of macro definitions can be used in all source programs to simplify the development of assembly language routines.

Macro language is enhanced by including a mechanism for passing parameters (known as macro parameters) to the macro body while it is being expanding, allowing a single macro definition to be used for an entire class of subtasks.

Example:

```
                        ; macro definition. . .
.MACRO STRING           ; macro identifier is
                        ;     STRING
                        ; Macro Body:
                        ; %1 and %2 are
                        ;     parameter
                        ;     declarations
.BYTE %2                ; 2nd parameter is
                        ;     length byte
.ASCII %1               ; 1st parameter is
                        ;     argument
.ENDM                   ; end macro definition
```

Further down in the source code...

```
STRING  "WRITE",5.      ; 1st macro call
                        ; parameters are
                        ;     "WRITE"
                        ;     and 5.

STRING "TYPE SPACE",10.
                        ; 2nd macro call
                        ; parameters are
                        ;     "TYPE SPACE"
                        ;     and 10.
```

This is what gets assembled...

```
.BYTE  5.  ; data string declarations
.ASCII  "WRITE"

.BYTE  10.
.ASCII  "TYPE SPACE"
```

Macro Definitions

Macro definitions may occur anywhere in a source program and are delimited by the directives .MACRO and .ENDM. The macro identifier must be unique to the source program, except when the programmer is redefining a predefined machine instruction name as a macro identifier. A macro definition may not include another macro definition; however, it may include macro calls. Macro calls may be nested to a maximum depth of five levels. A macro definition must occur before any calls to that macro are assembled, but macro calls may be forward referenced within the bodies of other macro definitions.

Macro Calls

Macro calls may occur anywhere in a source program that code may be generated. A macro call consists of a macro identifier followed by a list of parameters. The parameters are delimited by commas and terminated by a carriage return or semicolon. Upon encountering a macro call, source code is read from the text of the corresponding macro body. Macro parameters within the macro body are substituted with the text of the matching parameter listed after the macro identifier which initiated the call.

Parameter Passing

Macro parameters are referenced in a macro body by using the symbol %n in an expression, where n is a single nonzero decimal digit. Upon scanning this symbol, the assembler replaces it with the text of the n'th macro parameter. Please note that macro parameters are *not* expanded within the quotes of an ASCII data string.

Three cases are possible:

1) The parameter exists — make the substitution.

2) The n'th parameter doesn't exist in the parameter list being checked (less than n parameters were passed); a null string is substituted.

3) Another symbol of the form %m is encountered in the parameter list. If nested macro calls exist, the text of the m'th parameter at the next higher level of macro nesting is substituted; otherwise, the symbol itself is assembled.

Parameters are passed without leading and trailing blanks. All assembly symbols except macro calls may be passed as parameters.

The following is an example of parameter passing in macros:

```
.MACRO DOS
UNO     %2, UN
SAR     %1
.ENDM

.MACRO  UNO
MOV     %1, %2
SAL     %2
.ENDM
```

In a program, the macro call...

```
DOS  TROIS, DEUX
```

assembles as...

```
MOV  DEUX, UN     ; UNO got UN directly,
                  ; but had to use DOS's
                  ; 2nd param
SAL  DEUX
SAR  TROIS        ; DOS used its own 1st
                  ; param
```

Scope Of Labels In Macros

A problem arises in the use of macro language when the definition of a macro body requires the use of branch instructions and thus the presence of labels. Declaring a regular label in a macro body is incorrect if the macro is called more than once, for the label would be substituted twice into the source program and flagged by the assembler as a previously defined label. Location-counter-relative addressing can be used, but is prone to errors in nontrivial applications. The solution is to generate labels that are local to the macro body; the assembler's local labels have this capability.

Local label names declared in a macro body are local to that macro; thus, a section of code that contains a local label $1 and a macro call whose body also has the local label $1 will assemble without errors (contrast this with what happens when two occurrences of $1 fall between two regular labels). This feature allows local labels to be used freely in macros without fear of conflicts with the rest of the program.

> **Note:** The maximum of 21 local labels active at any instant still applies.

Local Labels As Macro Parameters

The passing of local labels as parameters has a special property. Unlike other macro parameters, local labels are not passed as uninterpreted text. The scope of a local label passed in a macro call does not change as it is passed through increasing levels of macro nesting, regardless of naming conflicts along the way. One use of this property is passing an address to a macro which simulates a conditional branch instruction.

The following is an example of passing local labels as macro parameters:

```
        .MACRO  EIN
        JE      $1
        JEN     %1
$1
        .ENDM
```

In a program, the code...

```
TWIE
        SUB   ICHI,NI
        EIN   $1
        RET
$1
        JMP   SAN
```

assembles as...

```
TWIE
        SUB   ICHI,NI
        JE    $1        ; this references macro
                        ;   local label
        JNE   $1        ; this references
                        ;   outside $1
$1                      ; macro local label

        RET
$1                      ; outside $1
        JMP   SAN
```

Program Linking and Relocation

The Adaptable Assembler produces either absolute or relocatable object code that may be linked as required to create executable programs from separately assembled or compiled modules.

Program linking directives generate information required by the System Linker to link modules.

Some of the advantages of linking are:

- Long programs can be divided into separately assembled modules to avoid a long assembly, reduce the symbol table size, and encourage modular programming techniques.
- Modules can be shared by other linked modules.
- Utility modules can be added to the System Library for use as external procedures by a large number of programs.
- Pascal programs can directly call assembly language procedures.

The assembler generates linker information in both relocatable and absolute code files. The System Linker accesses this information during the linking process and removes it from the linked code file.

Relocatable code includes information that allows a loader program to place it anywhere in memory, while absolute (also called core image) codefiles must be loaded into a specific area of memory to execute properly. Assembly procedures running in the Pascal system environment must always be relocatable; the loading and relocation process is performed by the interpreter at a load address determined by the state of the System.

Absolute code will not run under the p-System environment (under which high-level programs must run). Relocatable code *can* run under the p-System. Code segments which contain statically relocatable code remain in main memory throughout the lifetime of their host program (or unit), and are position-locked for that duration. Thus, relocatable code may maintain and reference its own internal data space (or spaces). In addition,

statically relocatable code saves some space because its relocation information does not have to remain present throughout the life of the program.

The directives .PROC and .FUNC designate statically relocatable routines; .RELPROC and .RELFUNC designate dynamically relocatable routines. Code segments which contain dynamically relocatable code do not necessarily occupy the same location in memory throughout their host's lifetime, but are maintained in the code pool along with other dynamic segments (mostly P-code), and may be swapped in and out of main memory while the host program (or unit) is running. Thus, dynamically relocatable code *cannot* maintain internal data spaces — data which is meant to last across different calls of the assembly routine must be kept in host data segments using .PRIVATEs and .PUBLICs. (It is the programmer's responsibility to make sure that this is the case.)

Examples: 1) Data space is embedded in the code, but the code does not move:

```
.PROC  FOON
.WORD  SPACE
  ...
.END
```

2) The code moves, but data space is allocated in the host compilation unit's global data segment:

```
.RELPROC  FOON
.PRIVATE  SPACE
  ...
.END
```

3) **Wrong:** The code moves, and the data is embedded in the code, so the data is destroyed:

```
.RELPROC    FOON
.WORD       SPACE
  ...
.END
```

Code pool management is described in the *Internal Architecture Guide.*

Program Linking Directives

This section describes overall usage of linking directives. All linking of assembly procedures involves word quantities; it is not possible to externally define and reference data bytes or assembly time constants. Arguments of these directives must match the corresponding name in the target module (a lower case Pascal identifier will match an upper case assembly name, and vice versa) and must not have been used before their appearance in the directive; all following references to the arguments are treated by the assembler as special cases of labels. These external references are resolved by the linker and/or interpreter by adding the link time and run time offsets to the existing value of the word quantity in question; thus, any intial offsets generated by the inclusion of external references and constants in expressions are preserved.

Host Communication Directives

The directives .CONST, .PUBLIC, and .PRIVATE allow the sharing of constants and data between an assembly procedure and its host compilation unit.

.CONST — Allows an assembly procedure to access globally declared constants in the host compilation unit. All references to arguments of .CONST are patched by the Linker with a word containing the value of the host's compile time constant.

.PUBLIC — Allows an assembly procedure to access globally declared variables in the host compilation unit. Note - this directive can be used to set up pointers to the start of multi-word variables in host programs; it is not limited to single word variables.

.PRIVATE — Allows an assembly procedure to declare variables in the global data segment of the host compilation unit that are inaccessible to the host. The optional length attribute of the arguments allows multi-word data spaces to be allocated; the default data space is one word.

External Reference Directives

The directives .REF and .DEF allow separately assembled modules to share data space and subroutines. See "Example of Linking to Pascal Host" in this chapter for examples.

.DEF	Declares a label to be defined in the current program as accessible to other modules. One restriction is imposed on usage - it is invalid to .DEF a label that has been equated to a constant expression or an expression containing an external reference.
.REF	Declares a label existing and .DEF'ed in another module to be accessible to the current program.

Program Identifier Directives

The directives .PROC, .FUNC, .RELPROC, .RELFUNC, and .END serve as delimiters for source programs. Every source program (relocatable or absolute) must contain at least one pair or delimiting directives (see "Assembly Routines" in this chapter.)

The identifier argument of the .PROC or .RELPROC directives serves two functions: it is referenced by the Linker when linking an assembly procedure to its corresponding host, and it can be referenced as an externally declared label by other modules. Specifically, the declaration:

```
.PROC FOON   ; procedure heading
```

... in a source program is functionally equivalent in the assembly environment to the following statements:

```
.DEF FOON    ; FOON may be externally
             ;    referenced
FOON         ; declare FOON as a label
```

This feature allows an assembly module to call other (external and eventually linked in) assembly modules by name. The .FUNC and .RELFUNC

directives are used when linking an assembly function directly to a Pascal host program; they are not intended for uses which involve linking with other assembly modules.

The optional integer argument after the procedure identifier is referenced by the Linker to determine if the number of words of parameters passed by the Pascal host's external procedure declaration matches the number specified by the assembly procedure declaration; it is not relevant when linking with other assembly modules.

Linking Program Modules

For information on linking with FORTRAN, refer to the FORTRAN manual.

Linking With A Pascal Host Program

External procedures and functions are assembly language routines declared in Pascal programs. In order to run Pascal programs with external declarations, it is necessary to compile the Pascal program, assemble the external procedure or function, and link the two codefiles. The linking process can be simplified by adding the assembled routine to the system library with the librarian program.

A host program declares a procedure to be external in a syntactically similar manner to a forward declaration. The procedure heading is given (with parameter list, if any), followed by the keyword EXTERNAL. Calls to the external procedure use standard Pascal syntax, and the Compiler checks that calls to the external procedure agree in type and number of parameters with the external declaration. All parameters are pushed on the stack in the order of their appearance in the parameter list of the

declaration; thus, the rightmost parameter in the declaration will be on the top of the stack. "Parameter Passing Conventions" in this chapter has a detailed description of parameter passing conventions.

It is the programmer's responsibility to assure that the assembly language routine maintains the integrity of the stack. This includes removing all parameters passed from the host, preserving the SS and SP registers, and making a clean return to the Pascal run time environment using the return address originally passed to it. The price of nonconformance in these matters is a potentially fatal system crash, as assembly routines are outside the scope of the Pascal environment's run time error facilities. "Sharing Resource with the Interpreter" in this chapter has a detailed description of Pascal/ assembly language protocols on the IBM Personal Computer.

An external function is similar to a procedure, but with some differences that affect the way in which parameters are passed to and from the Pascal runtime environment. First, the external function call will push one, two, or four words on the stack before any parameters have been pushed. Two or four words will be pushed for a function of type real (depending upon the real size that your IBM Personal Computer has been set up to run on). One word will be pushed for all other types of functions. The words are part of the P-machine's function calling mechanism, and are irrelevant to assembly language functions; the assembly routine must throw these away before returning the function's result. Second, the assembly routine must push the proper number of words (2 or 4 for type real, 1 otherwise) containing the function result onto the stack before passing control back to the host. "Sharing Resource with the Interpreter" in this chapter describes a very clean way to do all of this without ever using an actual POP or PUSH operation.

The ability of external procedures to pass any variables as parameters gives the assembly programmer complete freedom to access the machine dependent representations of machine independent Pascal data structures; however, with this freedom comes the responsibility of respecting the integrity of the Pascal run time environment. This section attempts to enumerate the P-machine's parameter passing conventions for all data types in order that the programmer may gain a better understanding of the Pascal/assembly language interface; it does not actually describe data representations.

Parameters may be passed either by value or by name (variable parameters). For purposes of assembly language manipulation, variable parameters are handled in a more straightforward fashion than value parameters.

The word tos is used in the following sections as an abbreviation for top of stack.

Variable Parameters

Variable parameters are referenced through a one word pointer passed to the procedure. Thus, the procedure declaration:

```
procedure pass_by_name (var i,j : integer;
                        var q : some_type); external;
```

... would pass 3 one word pointers on the stack; tos would be a pointer to q, followed by pointers to j and and i.

A Pascal external procedure declaration is allowed to contain variable parameters lacking the usual type declaration; this enables variables of different Pascal

types to be passed through a single parameter to an assembly routine. Untyped parameters are not allowed in normal pascal procedure declarations.

The procedure declaration:

```
procedure untyped_var (var i; var q:
                       some_type); external;
```

... contains the untyped parameter i.

Value Parameters

Value parameters are handled in a manner dependent upon their data type. The following types are passed by pushing copies of their current values directly on the stack: boolean, char, integer, real, subrange, scalar, pointer, set, and long integer. Other sections of the user manual describe the number of words per data type and the internal data format. For instance, the declaration:

```
procedure pass_by_value (i : integer; r : real);
                                        external;
```

... would pass 2 words on tos containing the value of the real variable r followed by one word containing the value of the integer variable i.

Variables of type record and array are passed by value in the same manner as variable parameters; pointers to the actual variable are pushed onto the stack. Variables of type PACKED ARRAY OF CHAR and STRING are passed by value with a segment pointer (see "Accessing Byte Array Parameters with a Segment Pointer" in this chapter.

Pascal procedures protect the original variables by using the passed pointer to copy their values into a local data space for processing; assembly procedures should respect this convention and not alter the contents of the original variables.

Accessing Byte Array Parameters with a Segment Pointer

A segment pointer consists of two words on the stack. The first word (tos) contains either NIL or a pointer to a segment environment record.

If the first word is NIL, then the second word (at tos-1) points to the parameter.

If the first word is not NIL, then to find the parameter it is necessary to chain through some records. The first word is a pointer and the second word is an offset. The first word points to a segment environment record. The second word of this record contains a pointer to a pointer to the base of the segment where the parameter resides. The exact location of the parameter is given by the second word on the stack (tos-1), which is an offset into the code segment.

This address chain may be described as follows (offsets are word offsets):

(first word + 1) + *contents of second_word*

A full description of these mechanisms may be found in the *Internal Architecture Guide.*

Example Of Linking To Pascal Host

Note that in the following example the host program passes control to the beginning of an assembly procedure whether or not machine instructions are present there; therefore, all data sections allocated in the procedure must either occur after the end of the machine instructions or have a jump instruction branch around them.

Example:

```
PROGRAM EXAMPLE;
                {Pascal host program}
const size = 80;
var   i,j,k : integer;
      lstl : array [0..9] of char;
    {PRT and LST2 get allocated here}

  procedure do_nothing; external;
  function null_func
             (xxyxx,z:integer)
             :integer; external;

begin
   k := 45;
   do_nothing;
   j := null_func(k,size);
end.
```

```
.PROC       DONOTHING ; underscores
                      ;  are not
                      ;  significant
                      ;  in Pascal

.CONST      SIZE      ; can get at size
                      ;  constant in host
.PUBLIC     I,LST1    ; and also these two
                      ; global vars
.DEF        TEMP1     ; this allows
                      ;  NULLFUNC
                      ;  to get at temp1
                      ; code starts here
POP         RETADR    ; return addr
                      ;  pushed on stack


; does nothing

PUSH        RETADR    ; set up stack for
                      ;  return
RETL
                      ; data area
RETADR      .EQU      TEMP1
TEMP1       .WORD
                      ; end of procedure
                      ; DONOTHING
```

```
.FUNC       NULLFUNC,2

.PRIVATE    PRT,LST2:9 ; 10 words of
                       ;  private data
.REF        TEMP1      ; references data
                       ;  temp in
                       ;  DONOTHING
                       ; code starts here

POP         RETURN     ; save return
                       ;  address

POP         PRT        ; get parameter z
POP         LST2+4     ; get parameter
                       ;  xxyxx

POP         TEMP1      ; toss 1 word of
                       ;  junk

; performs null action

PUSH        LST2+4     ; return xxyxx as
                       ;  result
PUSH        RETURN     ; restore subr link
RETL                   ; return to calling
                       ;  program
                       ; data starts here
RETURN      .WORD
                       ; end of assembly
            .END
```

Stand-alone Applications

The UCSD p-System 8086/88/87 Assembler has the capability to produce absolute (core image) codefiles for use outside of the p-System's runtime environment.

The p-System does not include a linking loader or an assembly language debugger, as the P-machine architecture is not conducive to running programs (whether high or low level) that must reside in a

dedicated area of memory. The user is responsible for loading and executing the object codefile; this can be done using the p-System, with the understanding that the existing runtime environment may be jeopardized in the process. "Loading and Executing Absolute Codefiles" in this chapter provides some ideas on how to create a Pascal loader program.

The utility COMPRESSOR is a much easier and more versatile way of doing this task. It allows for relocation and compaction of code. Refer to the *Users' Guide* for the UCSD p-System.

Assembling

The .ABSOLUTE and .ORG directives are used to create an object codefile suitable for use as an absolute core image. .ABSOLUTE causes the creation of nonrelocatable object code, and .ORG may be used to initialize the location counter to any starting value. A source file headed by .ABSOLUTE should not have more than one assembly routine; sequential absolute routines do not produce continuous object code and cannot be successfully linked with one another to produce a core image.

The codefile format consists of a 1 block codefile header followed by the absolute code, and is terminated by one block of linker info; thus, stripping off the first and last block of the codefile will leave a core image file. The use of .ABSOLUTE should be limited to one routine; though linker information is generated, it is difficult to link absolute codefiles so as to produce a correct core image file.

Loading And Executing Absolute Codefiles

The following section describes one method of loading and executing absolute codefiles using the UCSD p-System. The program outlined is not the only solution. It is also feasible to use the system intrinsics to read and/or move the codefile into the desired memory location, but this requires a knowledge of where the interpreter, operating system, and user program reside in order to prevent system crashes by accidentally overwriting them. The program outlined below allows the most freedom in loading core images; the only constraint is that the assembly code itself is not overwritten while being moved to its final location. This possibility can be detected before loading proceeds.

It must be emphasized that in most cases loading object code into arbitrary memory locations while a Pascal system is resident will adversely affect the system; the absolute assembly language program is then on its own, and rebooting may be necessary to revive the Pascal system.

The loader program consists of:

1) A Pascal host program that calls two external procedures.

2) One or more linkable absolute codefiles to be loaded. (.RELPROCs are not allowed.)

3) A small assembly procedure MOVE_AND_GO that moves the above object codefiles from their system load address to their proper locations and transfers control to them.

4) A small assembly language procedure LOAD_ADDRESS that returns the system load addresses of the aforementioned assembly code to the host program.

The absolute codefiles are assembled to run at their desired locations, and MOVE_AND_GO contains the desired load addresses of each core image. Both LOAD_ADDRESS and MOVE_AND_GO have external references to the core images; these are used to calculate the system load address and code size of each image file. The whole collection is linked and executed, with the Pascal host performing the following actions:

Print the result of calling LOAD_ADDRESS to determine whether the area of memory in which the Pascal system loaded the assembly code overlays the known final load address of the core images. Issue a prompt to continue, so that the program can be aborted if a conflict does arise.

Call MOVE_AND_GO.

Operation of the Assembler

The system assembler is invoked by typing A at the command level of the operating system. This command will execute the file named SYSTEM.ASSMBLER (note the missing E in the file name; this is required for conformance with the file system's restrictions on file name lengths); if this is not the name of the desired assembler version, be sure to save the existing file SYSTEM.ASSMBLER under a different name before changing the desired assembler's name to SYSTEM.ASSMBLER. Assemblers that are not in use are usually saved with the file name ASM8086.CODE.

Support Files

The UCSD p-System 8086/88/87 assembler has three associated support files: two opcodes files and an error file. These should always be stored along with the assembler code file.

In order for the assembler to run correctly, it is necessary that the proper opcode files be present on some on-line disk. The assembler will search all units in increasing order of the unit number until it finds them. The opcode files must have the names 8086.OPCODES and 8087.FOPS. The 8087.FOPS file is necessary only if your IBM Personal Computer is set up to run with the 8087 floating point processor. The opcode files contains all predefined symbols (instruction and register names) and their corresponding values for the associated assembly language. If the proper opfile is not on-line, the assembler will write *opfilename* not on any vol and abort the assembly.

The assembler also has an error file which contains a list of 8086/88/87 specific error messages. The error file must have the name 8086.ERRORS. The presence of the error file is not necessary for running the assembler, but it can greatly aid the chore of squeezing the syntax errors out of a freshly written program.

Setting Up Input And Output Files

When the assembler is first invoked from the prompt line, it will attempt to open the work file as its input file; if a work file exists, the first prompt will be the listing prompt described in "Responses to Listing Prompt" in this chapter and the generated code file will be named SYSTEM.WRK.CODE. If not, this prompt will appear.

Assemble what text?

Type in the file name of the input file followed by a carriage return. Typing only a carriage return will abort the assembly; otherwise, the next prompt will then appear:

To what codefile?

Type in the desired name of the output code file followed by a carriage return. Typing only a carriage return here will cause the assembler to name the output *SYSTEM.WRK.CODE, but typing $ will cause the code file to be created with the same filename prefix as the source file. The assembler will then display its standard listing prompt.

Responses To Listing Prompt

Before assembling begins, the following prompt will appear on the console:

```
8086 Assembler
Output file for assembled listing: CR for none
```

At this point, the user may respond with one of the following:

0) The escape key will abort the assembly and return the user to the operating system prompt.

1) CONSOLE: or #1: will send an assembled listing of the source program to the screen during assembly.

2) PRINTER: or #6: will send an assembled listing to the printer unit.

3) REMOUT: or #8: will send an assembled listing to the REMOTE unit.

4) A carriage return will cause the assembler to suppress generation of an assembled listing and ignore all listing directives.

5) All other responses will cause the assembler to write the assembled listing to a text file of that name; any existing textfile of that name will be removed in the process. For instance, the following responses will cause a list file named LISTING.TEXT to be created on disk unit 5:

```
#5:listing.text
#5:listing
```

In all cases, it is the responsibility of the user to ensure that the specified unit is on-line; the assembler will print an error message and abort if it is requested to open an off-line I/O unit.

Output Modes

If the user sends an assembled listing to the console, then that is what will be displayed on the screen during the assembly process; however, if the listing is sent to some other unit or if no listing is generated, the assembler writes a running account of the assembly process to the screen for the user's benefit. One dot is written to the screen for every line assembled; on every 50'th line, the number of lines currently assembled is written on the left hand side of the screen (delimited by angle brackets).

When an include file directive is processed by the assembler, the console displays the current source statement:

.INCLUDE ***file name***

This allows the user to keep track of which include file is currently being assembled.

At the end of the assembly, the console displays the total number of lines assembled in the source program and the total number of errors flagged in the source program.

Responses To Error Prompt

When the assembler uncovers an error, it will print the error number and the current source statement (if applicable to the error; this does not apply to undefined labels and system errors). It then attempts to retrieve and print an error message from the errors file. If the errors file cannot be opened (file is nonexistent or lack of memory), no message will appear. This is followed by the prompt:

E(dit, ***space, esc***

Typing an E will invoke the editor, a space will continue the assembly, and an escape character will abort the assembly. Some restrictions exist when either invoking the editor or attempting to continue:

1) In most cases, typing a space character restarts the assembly process with no problems; since assembly language source statements are independent of one another with respect to syntax, it is not a difficult task for the assembler to continue generating a code file. Thus, a code file will exist at the end of an assembly if the user types a space for every (nonfatal) error prompt that appears; of course, the code produced may not be a correct translation of the user's source program. Certain system errors are considered fatal by the assembler; these errors will abort the assembly regardless of the response given to the above prompt.

2) If an E is typed, the system automatically invokes the editor, which opens the file containing the offending error and positions the cursor at the location where the error occurred. This feature will always work correctly when the source program is wholly contained in one file; however, when include files are used, the user should set up the input and output files manually (see "Setting Up Input and Output Files" in this chapter) in order for the editor to position the cursor in the file that contains the error.

Miscellany

At the end of an assembly, an error message for each undefined label is printed. In some cases, occurrences of undefined labels can be ignored by the user if the labels in question are semantically irrelevant to the desired execution of the code file; the resulting code file will be perfectly valid, but the

references to the nonexistent labels will not be completely resolved.

In addition to generating a codefile, the assembler makes use of a scratch file, which is always removed from the disk upon normal termination of the assembly. Occasionally though, a system error may occur that will prevent the assembler from removing this file; if this happens, a new file may appear named LINKER.INFO. It may be removed without anxiety, as it is entirely useless outside of the assembler's domain. This should be a rare (if not nonexistent) phenomenon.

Assembler Output

The assembler can generate two varieties of output files. A codefile is always produced, but the user controls whether an assembled listing of the source file is produced.

An assembled listing displays each line of the source program, the machine code generated by that line, and the current value of the location counter. The listing may display the expanded form of all macro calls in the source program. Any errors that occur during the assembly process have messages printed in the listing file, usually immediately following the line of source code that caused the error. A symbol table is printed at the end of the listing; it serves as a directory for locating all labels declared in the source program.

An assembled listing of a source program printed on hard copy is one of the most effective debugging aids available for assembly language programs; it is equally useful for off-line, mental debugging and in conjunction with system debuggers.

A description of the codefile format is beyond the scope of this document. See the *Internal Architecture Guide.*

Source Listing

A paginated assembled listing is produced when the user responds to the assembler's listing prompt with a listfile name. The default listing is 132 characters wide and 55 lines per page. Each line of a source program is included in the assembled listing, except for source lines that contain list directives. Source statements that contain the equate directive .EQU have the resulting value of the associated expression listed to the left of the source line.

Macro calls are always listed, including the list of macro parameters and the comment field, if any. The macro is expanded by listing the body (with all formal parameters replaced by their passed values) if the macro list option was enabled when the macro was defined. Macro expansion text is marked in the assembled listing by the character # just to the left of the source listing. Comment fields in the definition of the macro body are not listed in macro expansions.

Source lines with conditional assembly directives are listed; however, source statements in an unassembled part of a conditional section are not listed.

Error Messages

Error messages in assembled listings have the same format as the error messages sent to the console (see "Operation of the Assembler" in this chapter), except that the user prompt is not included.

Code Listing

The code field lies to the left of the source program listing. It always contains the current value of the location counter, along with either code generated by the matching source statement or the value of an expression occurring in a statement that includes the equate directive .EQU; all are printed in the default list radix of the assembler version being used (either hex or octal - see "Sharing Resources with the Interpreter" in this chapter). Separately emitted bytes and words of code on the same line are delimited by spaces.

Forward References

When the assembler is forced to emit a byte or word quantity that is the result of evaluating an expression that includes an undefined label, it lists a * for each digit of the quantity printed (for example, an unresolved hex byte is listed as **, while an unresolved octal word appears as ******). If the .PATCHLIST directive is used, the assembler lists patch messages every time it encounters a label declaration that enables it to resolve all occurrences of a forward reference to that label. The messages (one for every backpatch performed) appear before the source statement that contains the label in question, and are of the form:

*location in codefile patched * patch value*

With this feature, the listing describes the contents of each byte or word of emitted code; if neatness of the assembled listing is more desirable, the .NOPATCHLIST directive will suppress the patch messages.

External References

When the assembler emits a word quantity that is the result of evaluating an expression that contains an externally referenced label, the value of that label (which cannot be determined until link time) is taken as zero; therefore, the emitted value will reflect only the result of any assembly time constants that were present in the expression.

Multiple Code Lines

Sometimes, it is possible for one source statement to generate more code than will fit in the code field; in most cases, the code is listed on successive lines of the code field (with corresponding blank source listing fields). Three exceptions are the .ORG, .ALIGN, and .BLOCK directives; because most uses of these directives generate large numbers of uninteresting byte values, the code field for these arguments is limited to as many bytes as will fit in the code field of one line.

Symbol Table

The symbol table is an alphabetically sorted table of entries for all symbols declared in the source program. Each entry consists of three fields; the symbol identifier, the symbol type, and the value assigned to that symbol. The symbol identifiers are defined in a dictionary printed at the top of the symbol table. Symbols equated to constants have their constant values in the third field, while program labels are matched with their location counter offsets; all other symbols have dashes in their value field, as they possess no values relevant to the listing.

Sharing Machine Resources with Interpreter

Calling and Returning

The p-System Interpreter invokes an assembly routine using the call long (CALLL) operator. Thus, the top of the stack contains a two word return address upon entrance into the routine. In order to return from an assembly routine the return long (RETL) operator should be used. (Alternatively, the return address could be popped and a jump long (JMPL) operation used.)

Accessing Parameters

The 8086/88 processor contains instructions which make accessing parameters passed to an assembly routine very easy. By moving the value of SP (which points to the P-machine stack) into BP, the parameters can be accessed by adding an offset of 4 bytes (to account for the two word return address). The first parameter (at the location 4 bytes above the top of the stack) is actually the last declared parameter in the host routine (the parameters are pushed in the order that they are declared).

If a function value is to be returned by a .FUNC assembly routine, it should be placed just above the last parameter using the same accessing scheme. The size of the returned function value is either 1, 2, or 4 words as described in "Linking with a Pascal Host Program" in this chapter.

The RETL operator may be given an operand which indicates how many bytes to cut the stack back after popping its two word return address. The size of the data space occupied by the parameters should be used. Thus, parameters may be accessed, and a clean return made without ever using a specific POP or PUSH instruction.

The following is an example of this scheme of accessing parameters and returning:

```
MOV     BP,SP
MOV     AX,(BP+4)       ; Last Param
MOV     BX,(BP+6)       ; Middle Param
MOV     CX,(BP+8)       ; First Param
                .
                .
                .
MOV     (BP+10),RSLT    ; Function return val
                        ; (if .FUNC)

RETL    6               ; Remove 3 params
```

Register Usage

All of the 8086/88 registers are available for use by user assembly routines (the Interpreter saves and restores the register values that it needs).

SS and SP must be preserved by the user, however. (The user may create and use a private stack if a minimum of 40 words are left available for stack expansion during interrupts. This is a very dangerous procedure, however, and is *not* recommended.)

Notes:

1. The integrity of the P-machine stack *must* be maintained. This is the programmer's responsibility and if this is not done, the results are unpredictable.

2. Upon entrance into the assembly routine, SS equals the P-machine stack pointer (SP). Also, DS, ES, and CS are all equal to the base of the p-System code segment.

3. Parameters which are passed as Pascal VAR variables are p-System pointers to actual data. These pointers are relative to SS.

4. .PRIVATE and .PUBLIC variables are also SS relative.

5. .BYTE quantities, .WORD quantities, and .REF'f labels are relative to CS, DS, or ES.

NOTES

CHAPTER 3. THE 8086/88 CPU

Contents

NOTES

INTRODUCTION

This chapter briefly describes the registers, flags, and addressing modes of the 8086/88 CPU. For more detailed information concerning the 8086/88 processor see the Intel *8086 Family User's Manual.*

General Registers

The 8086/88 CPU contains eight 16-bit general registers. The general registers are subdivided into two sets of four registers each: the data registers (sometimes called the H & L group for "high" and "low"), and the pointer and index registers (sometimes called the P & I group).

The data registers are unique in that their upper (high) and lower halves are separately addressable. This means that each data register can be used interchangeably as a 16-bit register, or as two 8-bit registers. The other CPU registers always are accessed as 16-bit units only. The data registers can be used without constraint in most arithmetic and logic operations. In addition, some instructions use certain registers implicitly, thus allowing compact yet powerful encoding.

The pointer and index registers can also participate in most arithmetic and logic operations. The P & I registers (except for BP) also are used implicitly in some instructions.

Data Register Group

Accumulator:	**AX (16 Bits)**
	AH (Bits 8-15)
	AL (Bits 0-7)
Base:	**BX (16 Bits)**
	BH (Bits 8-15)
	BL (Bits 0-7)
Count:	**CX (16 Bits)**
	CH (Bits 8-15)
	CL (Bits 0-7)
Data:	**DX (16 Bits)**
	DH (Bits 8-15)
	DL (Bits 0-7)

Pointer and Index Register Group Registers

SP	**Stack Pointer**
BP	**Base Pointer**
SI	**Source Index**
DI	**Destination Index**

Implicit Use of General Registers

AX	**Word Multiply,**
	Word Divide,
	Word I/O
AL	**Byte Multiply,**
	Byte Divide,
	Byte I/O
	Translate
	Decimal Arithmetic
AH	**Byte Multiply,**
	Byte Divide,
BX	**Translate**

CX	**String Operations, Loops**
CL	**Variable Shift and Rotate**
DX	**Word Multiply, Word Divide, Indirect I/O**
SP	**Stack Operations**
SI	**String Operations**
DI	**String Operations**

Segment Registers

The megabyte of memory addressable by the 8086/88 is divided into logical segments of up to 64K bytes each. (Memory segmentation is described in detail in the Intel *8086 Family User's Manual.*) The CPU has access to four segments at a time. Their base addresses (starting locations) are contained in the segment registers. The following table lists the segment registers:

Segment Registers

CS Code Segment

DS Data Segment

SS Stack Segment

ES Extra Segment

The CS register points to the current code segment; instructions are fetched from this segment. The SS register points to the current stack segment; stack operations are performed on locations in this segment. The DS register points to the current data

segment; it generally contains program variables. The ES register points to the current extra segment, which also is typically used for data storage.

The segment registers are accessible to programs and can be manipulated with several instructions. See the Intel *8086 Family User's Manual* for suggested guidelines concerning segment register use.

Flags

The 8086/88 has six 1-bit status flags that reflect certain properties of the result of an arithmetic or logic operation. A group of instructions is available that allows a program to alter its execution depending on the state of these flags, that is, depending upon the result of a prior operation. Different instructions affect the status flags differently; in general, however, the flags reflect the following conditions:

If **AF** (the auxiliary carry flag) is set, there has been a carry out of the low nibble (4 bits) into the high nibble, or a borrow from the high nibble into the low nibble of an 8-bit quantity. This flag is used by decimal arithmetic instructions.

If **CF** (the carry flag) is set, there has been a carry out of, or a borrow into, the high-order bit of the result (8- or 16-bit). The flag is used by instructions that add and subtract multibyte numbers. Rotate instructions can also isolate a bit in memory or a register by placing it in the carry flag.

If **OF** (the overflow flag) is set, an arithmetic overflow has occurred; that is, a significant digit has been lost because the size of the result exceeded the capacity of its destination location. An Interrupt On Overflow instruction is available that will generate an interrupt in this situation.

If **SF** (the sign flag) is set, the high-order bit of the result is a 1. Since negative binary numbers are represented in the 8086/88 in standard two's complement notation, SF indicates the sign of the result (0=positive, 1=negative).

If **PF** (the parity flag) is set, the result has even parity, an even number of 1-bits. This flag can be used to check for data transmission errors.

If **ZF** (the zero flag) is set, the result of the operation is 0.

Three additional control flags can be set and cleared by programs to alter processor operations:

Setting **DF** (the direction flag) causes string instructions to auto-decrement; that is, to process strings from high addresses to low addresses, or from "right to left". Clearing DF causes string instructions to auto-increment, or to process strings from "left to right".

Setting **IF** (the interrupt-enable flag) allows the CPU to recognize external (maskable) interrupt requests. Clearing IF disables these interrupts. IF has no affect on either non-maskable external or internally generated interrupts.

Setting **TF** (the trap flag) puts the processor into single-step mode for debugging. In this mode, the CPU automatically generates an internal interrupt after each instruction, allowing a program to be inspected as it executes instruction by instruction. The Intel *8086 Family User's Manual* contains an example showing the use of TF in a single-step and breakpoint routine.

The following table is a summary of the Flags:

CF Carry

PF Parity

AF Auxiliary Carry

ZF Zero

SF Sign

OF Overflow

IF Interrupt-Enable

DF Direction

TF Trap

Addressing Modes

The 8086/88 provides many different ways to access instruction operands. Operands may be contained in registers, within the instruction itself, in memory or in I/O ports. In addition, the addresses of memory can be calculated in several different ways. This section briefly describes these addressing modes. For a more complete description see the Intel *8086 Family User's Manual.*

Register and Immediate Operands

Instructions that specify only register operands are generally the most compact and fastest executing of all instruction forms. This is because the register "addresses" are encoded in instructions in just a few bits, and because these operations are performed entirely within the CPU (no bus cycles are run).

Registers may serve as source operands, destination operands, or both.

Immediate operands are constant data contained in an instruction. The data may be either 8 or 16 bits in length. Immediate operands can be accessed quickly because they are available directly from the instruction queue; like a register operand, no bus cycles need to be run to obtain an immediate operand. The limitations of immediate operands are that they may only serve as source operands and that they are constant values.

Direct Addressing

Direct addressing is the simplest memory addressing mode. No registers are involved. The effective address is taken directly from the displacement field of the instruction. (The effective address is the unsigned 16-bit number that expresses the operand's distance in bytes from the beginning of the segment in which it resides.) Direct addressing is typically used to access simple variables (scalars).

Register Indirect Addressing

The effective address may be taken directly from one of the base or index registers (BX, BP, SI, DI). One instruction can operate on many different memory locations if the value in the base or index register is updated appropriately. The LEA (Load Effective Address) and arithmetic instructions might be used to change the register value.

Any 16-bit general register may be used for register indirect addressing with the JMP or CALL instructions.

Based Addressing

In based addressing, the effective address is the sum of a displacement value and the content of register BX or register BP. Specifying BP as a base register directs the Bus Interface Unit (see Intel Manual) to obtain the operand from the current stack segment (unless a segment override prefix is present). This makes based addressing with BP a very convenient way to access stack data (the Intel manual contains examples of this).

Based addressing also provides a straightforward way to address structures which may be located at different places in memory. A base register can be pointed at the base of a structure, and elements of the structure addressed by their displacements from the base. A different location can be accessed by simply changing the base register.

Based Index Addressing

Based indexed addressing generates an effective address that is the sum of a base register, an index register, and a displacement. Based indexed addressing is a very flexible mode because two address components can be varied at execution time.

Based indexed addressing provides a convenient way for a procedure to address an array allocated on a stack. Register BP can contain the offset of a reference point on the stack, typically the top of the stack after the procedure has saved registers and allocated local storage. The offset of the beginning of the array from the reference point can be expressed by a displacement value, and an index register can be used to access individual array elements.

Arrays contained in structures and matrices (two-dimensional arrays) also can be accessed with based indexed addressing.

String Addressing

String instructions do not use the normal memory addressing modes to access their operands. Instead, the index registers are used implicitly. When a string instruction is executed, SI is assumed to point to the first byte or word of the source string, and DI is assumed to point to the first byte or word of the destination string. In a repeated string operation, SI and DI are automatically adjusted to obtain subsequent bytes or words.

NOTES

CHAPTER 4. 8086/88/87 INSTRUCTIONS

Contents

NOTES

INTRODUCTION

This chapter describes how the UCSD p-System 8086/88/87 Assembler differs from the Intel standard assembler.

Also, brief descriptions of each of the 8086/88 and 8087 operators are given. These descriptions are intended for quick reference use only. For detailed information concerning the instruction set, see the *Intel 8086 Family User's Manual.*

Assembler Differences from the Intel Standard

The UCSD p-System 8086/88/87 Assembler differs in some respects from the standard Intel assembler. These differences are listed in this chapter.

Assembler Directives

None of the Intel assembler directives or operators are implemented. Instead, the assembler directives described in Chapter 2 of this manual are available.

Specification of Code Segment Register

The default code segment register is CS. Many operations use this register, as a default, to indicate which 64K segment of memory to obtain an operand from. If it is desired that another segment register be used, that register may be specified, followed by a colon, followed by the operand (see Segment Override below). In addition to the forms DS:memop, and so forth, a separate mnemonic SEG followed by a segment register name may be written in a statement preceding the instruction mnemonic.

Examples: `MOV AX,ES:AVALUE`

is equivalent to

```
SEG ES MOV AX,AVALUE
```

Parenthesis

Index or base register references in a memory operand are enclosed in parentheses, not square brackets, for example, FIRST(BX) rather than FIRST[BX].

Immediate Byte

ADD immediate byte to memory operand is coded

```
ADDBIM memop,immedbyte
```

to distinguish it from the ADD memop, immedword which is the default. Similarly, MOVBIM, ADCBIM, SUBBIM, SBBBIM, CMPBIM, ANDBIM, ORBIM, XORBIM, and TESTBIM are added to the vocabulary.

Memory Byte

INC memory byte is coded:

```
INCMB memop
```

to distinguish it from INC memory word, which is the default. Similarly, DECMB, MULMB, IMULMB, DIVMB, IDIVMB, NOTMB, NEGMB, ROLMB, RORMB, RCLMB, RCRMB, SALMB, SHLMB, SHRMB, SARMB were added to the vocabulary to specify memory byte operands.

MUL and DIV Byte

In MUL, IMUL, DIV, IDIV the single memory operand form, for example:

```
MUL  memop
```

implies a word operation. To specify a byte operation, either MULMB memop may be used, or the form

```
MUL  AL,memop
```

The same holds true for IMUL, DIV, IDIV.

Note: DIV AL,memop is rather misleading, as the actual operation would be AX/ memory-byte.

MOV substitute for LEA

For LEA reg,label or LEA reg,label+const the assembler will substitute MOV reg,immed_val where immed_val = label or label+const. This saves four clock times (4 vs. 8).

IN and OUT

The normal form of IN and OUT is IN ac,port or IN ac,DX and OUT port,ac or OUT DX,ac where ac=AL denotes an 8-bit data path and ac=AX denotes a 16-bit path. Since the accumulator is the only possible register source/destination (DX specifies port=address in DX), single operand forms are also provided: INB and OUTB for byte data, and INW and OUTW for 16-bit data. The syntax is INB port or INB DX.

In the two-operand forms of IN and OUT, the order of the operands is not important; thus OUT ac,DX or OUT ac,port will be acceptable.

String Operations

The mnemonics for the string operations are suffixed with B or W to denote byte or word operations: thus MOVSB and MOVSW, CMPSB and CMPSW, SCASB and SCASW, LODSB and LODSW, and STOSB and STOSW are in the vocabulary, but MOVS ... STOS are not.

Segment Override

XLAT and the string instructions (9) have implied memory operands and nothing is required to be coded in the operand field. However, in order to permit the specification of a segment override prefix in the case of XLAT, MOVSB/MOVSW, CMPSB/CMPSW, and LODSB/LODSW, the assembler permits operand expressions for these instructions.

Note, however, that only the default segment for SI, namely DS, can be overridden. The segment for DI is ES and cannot be overridden. A segment override prefix of DS applied to SI does not generate a segment override prefix.

If these operations were written with operands, they would have this syntax:

```
XLAT        AL, (BX)
MOVS {B|W}  (DI), [seg:] (SI)
CMPS {B|W}  (DI), [seg:] (SI)
SCAS {B|W}  (DI),AX
LODS {B|W}  AX, [seg:] (SI)
STOS {b|W}  (DI),AX
```

The string instructions may be prefixed by a REP (repeat) instruction of some type. The assembler flags an error if both REP and a segment override are specified.

Long Jumps, Calls, and Returns

Intersegment CALL, RET, and JMP are implemented as follows:

1) The mnemonics CALLL, RETL, and JMPL specifically designate intersegment operations.

2) An indirect address (for example, (reg) or (label)) is assembled in standard fashion with a "mod op r/m" effective address byte possibly followed by displacement bytes. The memory location references must hold the new IP, and the next higher location must hold the new CS.

3) The direct address form must have two absolute operands:

   CALLL expr1,expr2

   where expr1 is the new IP and expr2 becomes the new CS. Constants or external symbols (for example, .REF definitions) qualify as absolute operands.

8087 Mnemonics

Mnemonics for the 8087 floating point operations are standard except for certain of the memory reference operations, where a letter suffix is appended to denote the operand size:

_D short real or short integer (double word)

_Q long real or long integer (quad word)

_W integer word

_T temporary real (ten byte)

The D and Q suffixes apply to the following real ops:

FADD, FCOM, FCOMP, FDIV, FDIVR, FMUL, FST, FSUB, GSUBR, FLD, FSTP

Example: **FADDD, FADDQ, etc.**

The T suffix applies only to FLD and FSTP.

The W and D suffixes apply to the following integer ops:

FIADD, FICOM, FICOMP, FIDIV, FIDIVR, FIMUL, FIST, FISUB, FISUBR, FILD, FISTP

The Q suffix for long integers applies only to FILD and FISTP.

The 8086/88 Instruction Set

The following are the 8086/88 opcode mnemonics recognized by the UCSD p-System 8086/88/87 Assembler. The differences between these mnemonics and the standard Intel mnemonics is discussed in "Assembler Differences from the Intel Standard" in this chapter. This is meant as a quick reference list only. For a detailed description of the 8086/88 operations see the Intel *8086 Family User's Manual.*

Note: The special case mnemonics (which are not Intel standard) such as ADDBIM are listed with the standard mnemonic to which they correspond, for example, ADD. This does not mean that the special case mnemonics indicate operations which take all of the addressing modes listed. For example, ADDBIM is meant for adding immediate bytes only. The mnemonic ADD is meant to take any of the other addressing modes listed (and will default to a word add if an immediate quantity is indicated). All of these special mnemonics are discussed in "Assembler Differences from the Intel Standard" in this chapter.

AAA (ASCII Adjust for Addition)

Format: AAA (no operands)

Flags:

O	D	I	T	S	Z	A	P	C
U				U	U	X	U	X

Operands: none

Example: **AAA**

AAD (ASCII Adjust for Division)

Format: AAD (no operands)

Flags:

O	D	I	T	S	Z	A	P	C
U				X	X	U	X	U

Operands: none

Example: **AAD**

AAM (ASCII Adjust for Multiply)

Format: AAM (no operands)

Flags:

O	D	I	T	S	Z	A	P	C
U				X	X	U	X	U

Operands: none

Example: **AAM**

AAS (ASCII Adjust for Subtraction)

Format: AAA (no operands)

Flags:

O	D	I	T	S	Z	A	P	C
U				U	U	X	U	X

Operands: none

Example: **AAS**

ADC (Add with carry)

ADCBIM (Add with carry Immediate Byte)

Format: ADC destination, source

Flags:

O	D	I	T	S	Z	A	P	C
X				X	X	X	X	X

Operands:
register, register
register, memory
memory, register
register, immediate
memory, immediate
accumulator, immediate

Example:

```
ADC AX,SI
```

ADD (Addition)

ADDBIM (Add Immediate Byte)

Format: ADD destination, source

Flags:

O	D	I	T	S	Z	A	P	C
X				X	X	X	X	X

Operands: register, register
register, memory
memory, register
register, immediate
memory, immediate
accumulator, immediate

Example: **ADD DI,(BX).ALPHA**

AND (Logical and)

ANDBIM (Logical and, immediate byte)

Format: AND destination, source

Flags:

O	D	I	T	S	Z	A	P	C
0				X	X	U	X	0

Operands:
register, register
register, memory
memory, register
register, immediate
memory, immediate
accumulator, immediate

Example:

```
AND CX,FLAG_WORD
```

CALL (Call a procedure)

CALLL (Long Call of a procedure)

Format: CALL target

Flags: O D I T S Z A P C

Operands: near-proc
far-proc
memptr 16
regptr 16
memptr 32

Example:
```
CALL NEAR_PROC
```


CBW (Convert byte to word)

Format: CBW (no operands)

Flags: O D I T S Z A P C

Operands: none

Example:
```
CBW
```

CLC (Clear carry flag)

Format: CLC (no operands)

Flags:

O	D	I	T	S	Z	A	P	C
								0

Operands: none

Example:

```
CLC
```

CLD (Clear direction flag)

Format: CLD (no operands)

Flags:

O	D	I	T	S	Z	A	P	C
	0							

Operands: none

Example:

```
CLD
```

CLI (Clear interrupt flag)

Format: CLI (no operands)

Flags:

O	D	I	T	S	Z	A	P	C
		0						

Operands: none

Example: **CLI**

CMC (Complement carry flag)

Format: CMC (no operands)

Flags:

O	D	I	T	S	Z	A	P	C
								X

Operands: none

Example: **CMC**

CMP (Compare destination to source)

CMPBIM (Compare immediate byte)

Format: CMP destination, source

Flags:

O	D	I	T	S	Z	A	P	C
X				X	X	X	X	X

Operands:
register, register
register, memory
memory, register
register, immediate
memory, immediate
accumulator, immediate

Example:

```
CMP (BP+2),SI
```

CMPSW (Compare string, wordwise)

CMPSB (Compare string, bytewise)

Format: CMPSB dest-string, source-string

Flags:

O	D	I	T	S	Z	A	P	C
X				X	X	X	X	X

Operands: dest-string, source-string
(repeat) dest-string, source-string

Example:

```
COMPSB BUFF1, BUFF2
```


CWD (Convert word to double word)

Format: CWD (no operands)

Flags:

O	D	I	T	S	Z	A	P	C

Operands: none

Example:

```
CWD
```

DAA (Decimal adjust for Addition)

Format: DAA (no operands)

Flags:

O	D	I	T	S	Z	A	P	C
X				X	X	X	X	X

Operands: none

Example:

```
DAA
```

DAS (Decimal adjust for Subtraction)

Format: DAS (no operands)

Flags:

O	D	I	T	S	Z	A	P	C
U				X	X	X	X	X

Operands: none

Example:

```
DAS
```

DEC (Decrement by one)

DECMB (Decrement memory byte)

Format: DEC destination

Flags:

O	D	I	T	S	Z	A	P	C
X				X	X	X	X	

Operands: reg16
reg8
memory

Example: **DEC AX**

DIV (Division, unsigned)

DIVMB (Division, unsigned, memory byte)

Format: DIV source

Flags:

O	D	I	T	S	Z	A	P	C
U				U	U	U	U	U

Operands: reg8
reg16
mem8
mem16

Example:

```
DIV TABLE[SI]
```

ESC (Escape)

Format: ESC external-opcode, source

Flags: O D I T S Z A P C

Operands: immediate, memory
immediate, register

Example: **ESC 20, AL**

HLT (Halt)

Format: HLT (no operands)

Flags: O D I T S Z A P C

Operands: none

Example: **HLT**

IDV (Integer Division)

IDIVMB (Integer Division, memory byte)

Format: IDIV source

Flags:

O	D	I	T	S	Z	A	P	C
U				U	U	U	U	U

Operands: reg8
reg16
mem8
mem16

Example:

```
IDIV (BX).DIVISOR_WORD
```

IMUL (Integer Multiplication)

IMULMB (Integer Multiplication memory byte)

Format: IMUL source

Flags:

O	D	I	T	S	Z	A	P	C
X				U	U	U	U	X

Operands: reg8
reg16
mem8
mem16

Example:

```
IMUL CL
```

IN (Input byte or word)

INB (Input byte)

INW (Input word)

Format: IN accumulator, port

Flags: O D I T S Z A P C

Operands: accumulator, immed8
accumulator, DX

Example:

```
IN AX, DX
```

INC (Increment by one)

INCMB (Increment memory byte)

Format: INC destination

Flags:

O	D	I	T	S	Z	A	P	C
X				X	X	X	X	

Operands: reg16
reg8
memory

Example:

```
INC CX
```


INT (Interrupt)

Format: INT interrupt-type

Flags:

O	D	I	T	S	Z	A	P	C
		0	0					

Operands: immed8 (type=3)
immed8 (type <> 3)

Example:

```
INT 3
```

INTR (external maskable interrupt)

Format: INTR (no operands)

Interrupt if INTR and IF=1

Flags:

```
O D I T S Z A P C
    0 0
```

Operands: none

Example: **not applicable**

INTO (Interrupt if overflow)

Format: INTO (no operands)

Flags:

```
O D I T S Z A P C
    0 0
```

Operands: none

Example: **INTO**

IRET (Interrupt return)

Format: IRET (no operands)

Flags:

O	D	I	T	S	Z	A	P	C
R	R	R	R	R	R	R	R	R

Operands: none

Example:

```
IRET
```

JA/JNBE (Jump if above/Jump if not below nor equal)

Format: JA short-label
JNBE short-label

Flags:

O	D	I	T	S	Z	A	P	C

Operands: short-label

Example:

```
JA ABOVE
```

JAE/JNB (Jump if above or equal/ Jump if not below)

Format: JAE short-label
JNB short-label

Flags: O D I T S Z A P C

Operands: short-label

Example:

```
JAE ABOVE_EQUAL
```

JB/JNAE (Jump if below/ Jump if not above nor equal)

Format: JB short-label
JNAE short-label

Flags: O D I T S Z A P C

Operands: short-label

Example:

```
JB BELOW
```

JBE/JNA (Jump if below or equal/ Jump if not above)

Format: JBE short-label
JNA short-label

Flags: O D I T S Z A P C

Operands: short-label

Example:

```
JNA NOT_ABOVE
```

JC (Jump if carry)

Format: JC short-label

Flags: O D I T S Z A P C

Operands: short-label

Example:

```
JC CARRY_SET
```

JCXZ (Jump if CX is zero)

Format: JCXZ short-label

Flags: O D I T S Z A P C

Operands: short-label

Example:

```
JCXZ COUNT_DONE
```

JE/JZ (Jump if equal/Jump if zero)

Format: JE short-label
JZ short-label

Flags: O D I T S Z A P C

Operands: short-label

Example:

```
JZ ZERO
```

JG/JNLE (Jump if greater/ Jump if not less nor equal)

Format: JG short-label
JNLE short-label

Flags: O D I T S Z A P C

Operands: short-label

Example:
```
JG GREATER
```

JGE/JNL (Jump if greater or equal/ Jump if not less)

Format: JGE short-label
JNL short-label

Flags: O D I T S Z A P C

Operands: short-label

Example:
```
JGE GREATER_EQUAL
```

JL/JNGE (Jump if less/Jump if not greater nor equal)

Format: JL short-label
JNGE short-label

Flags: O D I T S Z A P C

Operands: short-label

Example:

```
JL LESS
```

JLE/JNG (Jump if less or equal/ Jump if not greater)

Format: JLE short-label
JNG short-label

Flags: O D I T S Z A P C

Operands: short-label

Example:

```
JLE LESS_EQUAL
```

JMP (Jump)

JMPL (Jump Long)

Format: JMP target

Flags: O D I T S Z A P C

Operands:
short-label
near-label
far-label
memptr16
regptr16
memptr32

Example:
```
JMP NEAR_LABEL
```

JNC (Jump if not carry)

Format: JNC short-label

Flags: O D I T S Z A P C

Operands: short-label

Example:

```
JNC NOT_CARRY
```

JNE/JNZ (Jump if not equal/ Jump if not zero)

Format: JNE short-label
JNZ short-label

Flags: O D I T S Z A P C

Operands: short-label

Example:

```
JNE NOT_EQUAL
```

JNO (Jump if not overflow)

Format: JNO short-label

Flags: O D I T S Z A P C

Operands: short-label

Example: **JNO NO_OVERFLOW**

JNP/JPO (Jump if not parity/ Jump if parity odd)

Format: JNP short-label
JPO short-label

Flags: O D I T S Z A P C

Operands: short-label

Example: **JPO ODD_PARITY**

JNS (Jump if not sign)

Format: JNS short-label

Flags: O D I T S Z A P C

Operands: short-label

Example:

```
JNS POSITIVE
```

JO (Jump if overflow)

Format: JO short-label

Flags: O D I T S Z A P C

Operands: short-label

Example:

```
JO SIGNED_OVERFLOW
```

JP/JPE (Jump if parity/ Jump if parity even)

Format: JP short-label
JPE short-label

Flags: O D I T S Z A P C

Operands: short-label

Example:

```
JPE EVEN_PARITY
```

JS (Jump if sign)

Format: JS short-label

Flags: O D I T S Z A P C

Operands: short-label

Example:

```
JS NEGATIVE
```

LAHF (Load AH from flags)

Format: LAHF (no operands)

Flags: O D I T S Z A P C

Operands: none

Example:

```
LAHF
```

LDS (Load pointer using DS)

Format: LDS destination, source

Flags: O D I T S Z A P C

Operands: reg16, memptr32

Example:

```
LDS SI,DATA.SEG(DI)
```

LEA (Load effective address)

Format: LEA destination, source

Flags: O D I T S Z A P C

Operands: reg16, memptr16

Example: **LEA BX,(BP)(DI)**

LES (Load pointer using ES)

Format: LES destination, source

Flags: O D I T S Z A P C

Operands: reg16, memptr32

Example: **LES DI,(BX).TEXT_BUFF)**

LOCK (Lock bus)

Format: LOCK (no operands)

Flags: O D I T S Z A P C

Operands: none

Example:
```
LOCK XCHGFLAG,AL
```

LODSB (Load string bytewise)

LODSW (Load string wordwise)

Format: LODS source-string

Flags: O D I T S Z A P C

Operands: source-string
(repeat) source-string

Example:
```
REP LODS NAME
```

LOOP (Loop)

Format: LOOP short-label

Flags: O D I T S Z A P C

Operands: short-label

Example: **LOOP AGAIN**

LOOPE/LOOPZ (Loop if equal/ Loop if zero)

Format: LOOPE short-label
LOOPZ short-label

Flags: O D I T S Z A P C

Operands: short-label

Example: **LOOPE AGAIN**

LOOPNE/LOOPNZ (Loop is not equal/Loop if not zero)

Format: LOOPNE short-label
LOOPNZ short-label

Flags: O D I T S Z A P C

Operands: short-label

Example:

```
LOOPNE AGAIN
```

NMI
(external nonmaskable interrupt)

Format: NMI (no operands)

Interrupt if NMI=1

Flags:

O	D	I	T	S	Z	A	P	C
		0	0					

Operands: no operands

Example: **not applicable**

MOV (Move)

MOVBIM (Move immediate byte)

Format: MOV destination, source

Flags: O D I T S Z A P C

Operands:
memory, accumulator
accumulator, memory
register, register
register, memory
memory, register
register, immediate
memory, immediate
seg-reg, reg16
seg-reg, mem16
reg16, seg-reg
memory, seg-reg

Example:

```
MOV BP,STACK_TOP
```

MOVSB (Move string bytewise)

MOVSW (Move string wordwise)

Format: MOVS dest-string, source-string

Flags: O D I T S Z A P C

Operands: dest-string, source-string
(repeat) dest-string, source-string

Example:

```
MOVS LINE, EDIT_DATA
```


MOVSB/MOVSW (Move strng (byte/word))

Format: MOVSB/MOVSW (no operands)

Flags: O D I T S Z A P C

Operands: (none)
repeat (none)

Example:

```
REP MOVSW
```

MUL (Multiplication, unsigned)

MULMB (Multiplication, unsigned, memory byte)

Format: MUL source

Flags:

O	D	I	T	S	Z	A	P	C
X				U	U	U	U	X

Operands:
reg8
reg16
mem8
mem16

Example:

```
MUL CX
```

NEG (Negate)

NEGMB (Negate memory byte)

Format: NEG destination

Flags:

O	D	I	T	S	Z	A	P	C
X				X	X	X	X	1*

* 0 if destination=0

Operands: register
memory

Example: **NEG AL**

NOP (No operation)

Format: NOP

Flags: O D I T S Z A P C

Operands: none

Example: **NOP**

NOT (Logical not)

NOTBIM (Logical not, immediate byte)

Format: NOT destination

Flags: O D I T S Z A P C

Operands: register
memory

Example: **NOT AX**

OR (Logical inclusive or)

ORBIM (Logical inclusive or, immediate byte)

Format: OR destination, source

Flags:

O	D	I	T	S	Z	A	P	C
0				X	X	U	X	0

Operands:
register, register
register, memory
memory, register
accumulator, immediate
register, immediate
memory, immediate

Example:

```
OR FLAG_BYTE, CL
```

OUT (Output byte or word)

OUTB (Output byte)

OUTW (Output word)

Format: OUT port, accumulator

Flags: O D I T S Z A P C

Operands: immed8, accumulator
DX, accumulator

Example:

```
OUT DX, AL
```

POP (Pop word off stack)

Format: POP destination

Flags: O D I T S Z A P C

Operands: register
seg-reg (CS illegal)
memory

Example: **POP DX**

POPF (Pop flags off stack)

Format: POPF (no operands)

Flags:

O	D	I	T	S	Z	A	P	C
R	R	R	R	R	R	R	R	R

Operands: none

Example: **POPF**

PUSH (Push word onto stack)

Format: PUSH source

Flags: O D I T S Z A P C

Operands: register
seg-reg (CS legal)
memory

Example:

```
PUSH ES
```

PUSHF (Push flags onto stack)

Format: PUSHF (no operands)

Flags: O D I T S Z A P C

Operands: none

Example:

```
PUSHF
```

RCL (Rotate left through carry)

RCLMB (Rotate left through carry, memory byte)

Format: RCL destination, count

Flags:

O	D	I	T	S	Z	A	P	C
X								X

Operands:
register, 1
register, CL
memory, 1
memory, CL

Example:

```
RCL AL, CL
```

RCR (Rotate right through carry)

RCRMB (Rotate right through carry, memory byte)

Format: RCR destination, count

Flags:

O	D	I	T	S	Z	A	P	C
X								X

Operands:
register, 1
register, CL
memory, 1
memory, CL

Example: **RCR (BX).STATUS, 1**

REP (Repeat string operation)

Format: REP (no operands)

Flags: O D I T S Z A P C

Operands: none

Example: **REP MOVS DEST,SRCE**

REPE/REPZ (Repeat string operation while equal/while zero)

Format: REPE/REPZ (no operands)

Flags: O D I T S Z A P C

Operands: none

Example: **REPE CMPS DATA,KEY**

REPNE/REPNZ (Repeat string operation while not equal/not zero)

Format: REPNE/REPNZ (no operands)

Flags: O D I T S Z A P C

Operands: none

Example:

```
REPNE SCASW INPUT_LINE
```

RET (Return from procedure)

RETL (Return Long from procedure)

Format: RET optional pop value

Flags: O D I T S Z A P C

Operands: (intra-segment, no pop)
(intra-segment, pop)
(inter-segment, no pop)
(inter-segment, pop)

Example: **RET 4**

ROL (Rotate left)

ROLMB (Rotate left, memory byte)

Format: ROL destination, count

Flags:

O	D	I	T	S	Z	A	P	C
X								X

Operands:
register, 1
register, CL
memory, 1
memory, CL

Example:

```
ROL BX, 1
```

ROR (Rotate right)

RORMB (Rotate right, memory byte)

Format: ROR destination, count

Flags:

O	D	I	T	S	Z	A	P	C
X								X

Operands: register, 1
register, CL
memory, 1
memory, CL

Example:

```
ROR CMD_WORD, CL
```


SAHF (Store AH into flags)

Format: SAHF (no operands)

Flags:

O	D	I	T	S	Z	A	P	C
				R	R	R	R	R

Operands: none

Example:

```
SAHF
```

SAL/SHL (Shift arithmetic left/ Shift logical left)

SALMB/SHLMB (Shift left, memory byte)

Format: SAL/SHL destination, count

Flags:

O	D	I	T	S	Z	A	P	C
X								X

Operands: register, 1
register, CL
memory, 1
memory, CL

Example:

```
SAL AL, 1
```

SAR (Shift arithmetic right)

SARMB (Shift arithmetic right, memory byte)

Format: SAR destination, count

Flags:

O	D	I	T	S	Z	A	P	C
X				X	X	X	U	X

Operands:
register, 1
register, CL
memory, 1
memory, CL

Example: **SAR DI, CL**

SBB (Subtract with borrow)

SBBBIM (Subtract with borrow immediate byte)

Format: SBB destination, source

Flags:

O	D	I	T	S	Z	A	P	C
X				X	X	X	X	X

Operands:
register, register
register, memory
memory, register
accumulator, immediate
register, immediate
memory, immediate

Example:

```
SBB BX, CX
```

SCASB (Scan string, bytewise)

SCASW (Scan string, wordwise)

Format: SCASW dest-string

Flags:

O	D	I	T	S	Z	A	P	C
X					X	X	X	X

Operands: dest-string
(repeat) dest-string

Example:

```
REPNE SCASB BUFFER
```

SHR (Shift logical right)

SHRMB (Shift logical right, memory byte)

Format: SHR destination, count

Flags:

O	D	I	T	S	Z	A	P	C
X								X

Operands: register, 1
register, CL
memory, 1
memory, CL

Example:

```
SHR SI, 1
```

STC (Set carry flag)

Format: STC (no operands)

Flags:

O	D	I	T	S	Z	A	P	C
								1

Operands: none

Example: **STC**

STD (Set direction flag)

Format: STD (no operands)

Flags:

O	D	I	T	S	Z	A	P	C
	1							

Operands: none

Example: **STD**

STI (Set interrupt enable flag)

Format: STI (no operands)

Flags: O D I T S Z A P C
1

Operands: none

Example:
```
STI
```

STOSB (Store bytestring)

STOSW (Store word string)

Format: STOSB dest-string

Flags: O D I T S Z A P C

Operands: dest-string
(repeat) dest-string

Example:
```
REP STOSB DISPLAY
```

SUB (Subtraction)

SUBBIM (Subtraction, immediate byte)

Format: SUB destination, source

Flags:

O	D	I	T	S	Z	A	P	C
X				X	X	X	X	X

Operands:
register, register
register, memory
memory, register
accumulator, immediate
register, immediate
memory, immediate

Example: **SUB CX, BX**

TEST (Test or non-destructive logical and)

TESTBIM (Test, immediate byte)

Format: TEST destination, source

Flags:

O	D	I	T	S	Z	A	P	C
0				X	X	U	X	0

Operands:
register, register
register, memory
accumulator, immediate
register, immediate
memory, immediate

Example: **TEST SI, END_COUNT**

WAIT (Wait while TEST pin not inserted)

Format: WAIT (no operands)

Flags: O D I T S Z A P C

Operands: none

Example:
```
WAIT
```

XCHG (Exchange)

Format: XCHG destination, source

Flags: O D I T S Z A P C

Operands: accumulator, reg16
memory, register
register, register

Example:
```
XCHG AX, BX
```

XLAT (Translate)

Format: XLAT (source-table)

Flags: O D I T S Z A P C

Operands: source-table

Example: **XLAT ASCII_TAB**

XOR (Logical exclusive or)

XORBIM (Logical exclusive or, immediate byte)

Format: XOR destination, source

Flags:

O	D	I	T	S	Z	A	P	C
0				X	X	U	X	0

Operands:
register, register
register, memory
memory, register
accumulator, immediate
register, immediate
memory, immediate

Example: **XOR CL, MASK_BYTE**

8087 FLOATING POINT OPERATORS

The following is a reference list of the 8087 floating point operators. "Assembler Differences from the Intel Standard" in this chapter describes the differences between the UCSD p-System 8087 Assembler mnemonics suffixes and the standard Intel mnemonics.

Key to 8087 Exception Codes

I = Invalid Operand
Z = Zero Divide
D = Denormalized
O = Overflow
U = Underflow
P = Precision

Many instructions allow their operands to be coded in more than one way. For example, FADD (add real) may be written without operands, with only a source, or with a destination and a source. The instruction descriptions in this section employ the simple convention of separating alternative operand forms with slashes; the slashes, however, are not coded. Consecutive slashes indicate an option of no explicit operands. The operands for FADD are thus described as:

//source/destination,source

This means that FADD may be written in any of three ways:

FADD

FADD source

FADD destination,source

ST indicates the top of the stack. ST(i) indicates a stack element where i is a three bit quantity in the range 0 to 7. (See the Intel documentation for a complete description of this.)

FABS (Absolute value)

Format: FABS (no operands)

Operands: none

Exceptions: I

Example:

```
FABS
```

FADD (Add real)

Format: FADD //source/destination, source

Operands: //ST,ST(i)/ST(i),ST
short-real
long-real

Exceptions: I, D, O, U, P

Example:

```
FADD ST, ST(4)
```

FADDP (Add real and pop)

Format: FADDP destination, source

Operands: ST(i),ST

Exceptions: I, D, O, U, P

Example: **FADDP ST(2), ST**

FBLD (Packed decimal (BCD) load)

Format: FBLD source

Operands: packed decimal

Exceptions: I

Example: **FBLD YTD SALES**

FBSTP (Packed decimal (BCD) store and pop)

Format: FBSTP source

Operands: packed decimal

Exceptions: I

Example: **FBSTP (BX).FORCAST**

FCHS (Change sign)

Format: FCHS (no operands)

Operands: none

Exceptions: I

Example: **FCHS**

FCLEX/FNCLEX (Clear exceptions)

Format: FCLEX/FNCLEX (no operands)

Operands: none

Exceptions: none

Example: **FCLEX**

FCOM (Compare real)

Format: FCOM //source

Operands: //ST(i)
short-real
long-real

Exceptions: I, D

Example: **FCOM ST(1)**

FCOMP (Compare real and pop)

Format: FCOMP //source

Operands: //ST(i)
short-real
long-real

Exceptions: I, D

Example: **FCOMP ST(2)**

FCOMPP (Compare real and pop twice)

Format: FCOMPP (no operands)

Operands: none

Exceptions: I, D

Example: **FCOMPP**

FDECSTP (Decrement stack pointer)

Format: FDECSTP (no operands)

Operands: none

Exceptions: none

Example: **FDECSTP**

FDISI/FNDISI (Disable interrupts)

Format: FDISI/FNDISI (no operands)

Operands: none

Exceptions: none

Example: **FDISI**

FDIV (Divide real)

Format: FDIV (//source/destination,source)

Operands: //ST(i), ST
short-real
long-real

Exceptions: I, D, Z, O, U, P

Example: **FDIV ARC[DI]**

FDIVP (Divide real and pop)

Format: FDIVP destination, source

Operands: ST(i), ST

Exceptions: I, D, Z, O, U, P

Example: **FDIVP ST(4), ST**

FDIVR (Divide real reversed)

Format: FDIVR destination, source

Operands: //ST,ST(i)/ST(i), ST
short-real
long-real

Exceptions: I, D, Z, O, U, P

Example: **FDIVR ST(2), ST**

FDIVRP (Divide real reversed and pop)

Format: FDIVRP destination, source

Operands: ST(i), ST

Exceptions: I, D, Z, O, U, P

Example: **FDIVRP ST(1), ST**

FENI/FNENI (Enable interrupts)

Format: FENI/FNENI (no operands)

Operands: none

Exceptions: none

Example: **FENI**

FFREE (Free register)

Format: FFREE destination

Operands: ST(i)

Exceptions: none

Example: **FFREE**

FIADD (Integer add)

Format: FIADD source

Operands: word-integer
short-integer

Exceptions: I, D, O, P

Example: **FIADD DISTANCE TRAVELED**

FICOM (Integer compare)

Format: FICOM source

Operands: word-integer
short-integer

Exceptions: I, D

Example: **FICOM TOOL.N PASSES**

FICOMP (Integer compare and pop)

Format: FICOMP source

Operands: word-integer
short-integer

Exceptions: I, D

Example:

```
FICOMP N SAMPLES
```

FIDIV (Integer divide)

Format: FIDIV source

Operands: word-integer
short-integer

Exceptions: I, D, Z, O, U, P

Example:

```
FIDIV RELATIVE ANGLE(DI)
```

FIDIVR (Integer divide reversed)

Format: FIDIVR source

Operands: word-integer
short-integer

Exceptions: I, D, Z, O, U, P

Example: **FIDIVR FREQUENCY**

FILD (Integer load)

Format: FILD source

Operands: word-integer
short-integer
long-integer

Exceptions: I

Example: **FILD (BX).SEQUENCE**

FIMUL (Integer multiply)

Format: FIMUL source

Operands: word-integer
short-integer

Exceptions: I, D, O, P

Example: **FIMUL BEARING**

FINCSTP (Increment stack pointer)

Format: FINCSTP

Operands: none

Exceptions: none

Example: **FINCSTP**

FINIT/FNINIT (Initialize processor)

Format: FINIT

Operands: none

Exceptions: none

Example: **FNINIT**

FIST (Integer store)

Format: FIST destination

Operands: word-integer
short-integer

Exceptions: I, P

Example: **FIST OBS.COUNT(SI)**

FISTP (Integer store and pop)

Format: FISTP destination

Operands: word-integer
short-integer
long-integer

Exceptions: I, P

Example:

```
FISTP (BX).ALPHA_COUNT(SI)
```

FISUB (Integer subtract)

Format: FISUB source

Operands: word-integer
short-integer

Exceptions: I, D, O, P

Example:

```
FISUB BASE_FREQUENCY
```

FISUBR (Integer subtract reversed)

Format: FISUBR source

Operands: word-integer
short-integer

Exceptions: I, D, O, P

Example: **FISUBR BALANCE**

FLD (Load real)

Format: FLD source

Operands: ST(i)
short-real
long-real
temp-real

Exceptions: I, D

Example: **FLD ST(0)**

FLDCW (Load control word)

Format: FLDCW source

Operands: 2-bytes

Exceptions: none

Example: **FLDCW CONTROL WORD**

FLDENV (Load environment)

Format: FLDENV source

Operands: 14-bytes

Exceptions: none

Example: **FLDENV(BP+6)**

FLDLG2 (Load log (base 10) of 2)

Format: FLDLG2

Operands: none

Exceptions: I

Example: **FLDLG2**

FLDLN2 (Load log (base e) of 2)

Format: FLDLN2

Operands: none

Exceptions: I

Example: **FLDLN2**

FLDL2E (Load log (base 2) of e)

Format: FLDL2E

Operands: none

Exceptions: I

Example: **FLDL2E**

FLDL2T (Load log (base 2) of 10)

Format: FLDL2T

Operands: none

Exceptions: I

Example: **FLDL2T**

FLDPI (Load pi)

Format: FLDPI

Operands: none

Exceptions: I

Example: **FLDPI**

FLDZ (Load +0.0)

Format: FLDZ

Operands: none

Exceptions: I

Example: **FLDZ**

FLD1 (Load +1.0)

Format: FLD1

Operands: none

Exceptions: I

Example:

```
FLD1
```

FMUL (Multiply real)

Format: FMUL //source/destination, source

Operands: //ST(i),ST/ST,ST(i)
short-real
long-real

Exceptions: I, D, O, U, P

Example:

```
FMUL SPEED_FACTOR
```

FMULP (Multiply real and pop)

Format: FMULP destination, source

Operands: ST(i),ST

Exceptions: I, D, O, U, P

Example: **FMULP ST(1),ST**

FNOP (No operation)

Format: FNOP

Operands: none

Exceptions: none

Example: **FNOP**

FPATAN (Partial arctangent)

Format: FPATAN

Operands: none

Exceptions: U, P (operands not checked)

Example: **FPATAN**

FPREM (Partial remainder)

Format: FPREM

Operands: none

Exceptions: I, D, U

Example: **FPREM**

FPTAN (Partial tangent)

Format: FPTAN

Operands: none

Exceptions: I, P (operands not checked)

Example: **FPTAN**

FRNDINT (Round to integer)

Format: FRNDINT

Operands: none

Exceptions: I, P

Example: **FRNDINT**

FRSTOR (Restore saved state)

Format: FRSTOR source

Operands: 94-bytes

Exceptions: none

Example: **FRSTOR (BP)**

FSAVE/FNSAVE (Save state)

Format: FSAVE destination

Operands: 94-bytes

Exceptions: none

Example: **FSAVE (BP)**

FSCALE (Scale)

Format: FSCALE

Operands: none

Exceptions: I, O, U

Example: **FSCALE**

FSQRT (Square root)

Format: FSQRT

Operands: none

Exceptions: I, D, P

Example: **FSQRT**

FST (Store real)

Format: FST destination

Operands: ST(i)
short-real
long-real

Exceptions: I, O, U, P

Example:
```
FST MEAN READING
```

FSTCW/FNSTCW (Store control word)

Format: FSTCW destination

Operands: 2-bytes

Exceptions: none

Example:
```
FSTCW SAVE_CTRL
```

FSTENV/FNSTENV (Store environment)

Format: FSTENV destination

Operands: 14-bytes

Exceptions: none

Example: **FSTENV (BP)**

FSTP (Store real and pop)

Format: FST destination

Operands: ST(i)
short-real
long-real
temp-real

Exceptions: I, O, U, P

Example: **FSTP ST(2)**

FSTSW/FNSTSW (Store status word)

Format: FSTSW destination

Operands: 2-bytes

Exceptions: none

Example: **FSTSW SAVE_STATUS**

FSUB (Subtract real)

Format: FSUB //source/destination,source

Operands: //ST,ST(i)/ST(i),ST
short-real
long-real

Exceptions: I, D, O, U, P

Example: **FSUB BASE_VALUE**

FSUBP (Subtract real and pop)

Format: FSUBP destination, source

Operands: ST(i),ST

Exceptions: I, D, O, U, P

Example: **FSUBP ST(2),ST**

FSUBR (Subtract real reversed)

Format: FSUB //source/destination, source

Operands: //ST,ST(i)/ST(i),ST
short-real
long-real

Exceptions: I, D, O, U, P

Example: **FSUBR (BX).INDEX**

FSUBRP (Subtract real reversed and pop)

Format: FSUBRP destination, source

Operands: ST(i),ST

Exceptions: I, D, O, U, P

Example:

```
FSUBRP ST(2),ST
```

FTST (Test stack top against 0.0)

Format: FTST

Operands: none

Exceptions: I, D

Example:

```
FTST
```

FWAIT (CPU wait while 8087 is busy)

Format: FWAIT

Operands: none

Exceptions: none (CPU instruction)

Example: **FWAIT**

FXAM (Examine stack top)

Format: FXAM

Operands: none

Exceptions: none

Example: **FXAM**

FXCH (Exchange registers)

Format: FXCH //destination

Operands: //ST(i)

Exceptions: I

Example: **FXCH ST(2)**

FXTRACT
(Extract exponent and significand)

Format: FXTRACT

Operands: none

Exceptions: I

Example: **FXTRACT**

FYL2X (Y * Log (base 2) of X)

Format: FYL2X

Operands: none

Exceptions: P (operands not checked)

Example:

```
FYL2X
```

FYL2XP1 (Y * Log (base 2) of (X+1))

Format: FYL2XP1

Operands: none

Exceptions: P (operands not checked)

Example:

```
FYL2XP1
```

NOTES

APPENDIX A. ASSEMBLER ERROR MESSAGES

1: Undefined label

2: Operand out of range

3: Must have procedure name

4: Number of parameters expected

5: Extra garbage on line

6: Input line over 80 characters

7: Not enough ifs

8: Must be declared in ASECT before use

9: Identifier previously declared

10: Improper format

11: EQU expected

12: Must EQU before use if not to a label

13: Macro identifier expected

14: Word addressed machine

15: Backward ORG not allowed

16: Indentifier expected

17: Constant expected

18: Invalid structure

19: Extra special symbol

20: Branch too far

21: Variable not PC relative

22: Invalid macro parameter index

23: Not enough macro parameters

24: Operand not absolute

25: Invalid use of special symbols

26: Ill-formed expression

27: Not enough operands

28: Cannot handle this relative

29: Constant overflow

30: Invalid decimal constant

31: Invalid octal constant

32: Invalid binary constant

33: Invalid key word

34: Unexpected end of input - after macro

35: Include files must not be nested

36: Unexpected end of input

37: Bad place for an include file

38: Only labels & comments may occupy column one

39: Expected local label

40: Local label stack overflow

41: String constant must be on 1 line

42: String constant exceeds 80 chars

43: Invalid use of macro parameter

44: No local labels in ASECT

45: Expected key word

46: String expected

47: Bad block, parity error (CRC)

48: Bad unit number

49: Bad mode, invalid operation

50: Undefined hardware error

51: Lost unit, no longer on-line

52: Lost file, no longer in directory

53: Bad title, invalid file name

54: No room, insufficient space

55: No unit, no such volume on-line

56: No file, no such file on volume

57: Duplicate file

58: Not closed, attempt to open an open file

59: Not open, attempt to access a closed file

60: Bad format, error in reading real or integer

61: Nested macro definitions not allowed

62: = or <> expected

63: May not EQU to undefined labels

64: Must declare .ABSOLUTE before first .PROC

76: Had label, open parenthesis then invalid

77: Expected absolute expression

78: Both operands cannot be a segment register

79: Invalid pair of index registers

80: Have to use BX, BP, SI or DI

81: Invalid constant as first operand

82: The first operand is needed

83: The second operand is needed

84: Expected comma before 2nd operand

85: Registers stand alone except in indirect

86: Only 2 registers per operand

87: Expected label or absolute

88: Invalid to use BP indirect alone

89: Close parenthesis expected

90: Cannot POP CS

91: Cannot have exchange of r8 with r16

92: Segment registers not allowed

93: ESC external op on left must be const < 64

94: Only one of the operands can have segment override

95: Right operand must be a memory location

96: Left operand must be a 16-bit register

97: Left operand must be memory or register alone

98: Op cannot be a segment register or immediate

99: Count must be 1 or in CL

100: A byte constant operand is required

101: Operand must use ()or be a label

102: LOCK followed by something invalid

103: REP precedes only string operations

104: Not implemented

105: Expected a label

106: Not implemented

107: Open parenthesis expected

108: Expected register alone as right operand

109: Segovpre then regalone, that's invalid

110: Only one operand allowed

111: Operands are AL,op2 for byte MUL, etc.

112: SP can only be used with the SS segment

113: MOVBIM only for immediate to memory

114: BIMs must be immediate bytes to memory

115: Seg override on repeated instruction not ok

116: Segment register expected

117: (8087) Invalid two-operand format

118: (8087) Invalid single operand format

119: (8087) Improper operand field

120: (8087) Instruction has no operands

121: No override of ES on string destination

122: Interseg needs 2 constant or external operands

123: I/O port must be immediate byte or DX

124: I/O source/dest register must be AL or AX

INDEX

A

B

C

D

E

F

G

I

L

M

N

O

P

W

Personal Computer
Computer Language Series

Product Comment Form

Assembler Reference 6936562

Your comments assist us in improving our products. IBM may use and distribute any of the information you supply in anyway it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

Comments:

If you wish a reply, provide your name and address in this space.

Name _____

Address _____

City _____ State _____

Zip Code _____

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 123 BOCA RATON, FLORIDA 33432

POSTAGE WILL BE PAID BY ADDRESSEE

IBM PERSONAL COMPUTER
SALES & SERVICE
P.O. BOX 1328-C
BOCA RATON, FLORIDA 33432

Fold here

Please do not staple

Tape

)ntinued from inside front cover

)ME STATES DO NOT ALLOW THE
(CLUSION OF IMPLIED
ARRANTIES, SO THE ABOVE
(CLUSION MAY NOT APPLY TO
)U. THIS WARRANTY GIVES YOU
'ECIFIC LEGAL RIGHTS AND YOU
AY ALSO HAVE OTHER RIGHTS
'HICH VARY FROM STATE TO
:ATE.

IM does not warrant that the functions
ntained in the program will meet your
quirements or that the operation of the
'ogram will be uninterrupted or error
ee.

owever, IBM warrants the diskette(s) or
ssette(s) on which the program is fur-
shed, to be free from defects in materials
id workmanship under normal use for a
:riod of ninety (90) days from the date of
:livery to you as evidenced by a copy of
)ur receipt.

IMITATIONS OF REMEDIES

3M's entire liability and your exclusive
:medy shall be:

- the replacement of any diskette(s) or cassette(s) not meeting IBM's "Limited Warranty" and which is returned to IBM or an authorized IBM PERSONAL COMPUTER dealer with a copy of your receipt, or
- if IBM or the dealer is unable to deliver a replacement diskette(s) or cassette(s) which is free of defects in materials or workmanship, you may terminate this Agreement by returning the program and your money will be refunded.

N NO EVENT WILL IBM BE LIABLE
'O YOU FOR ANY DAMAGES,
NCLUDING ANY LOST PROFITS,
.OST SAVINGS OR OTHER
NCIDENTAL OR CONSEQUENTIAL
DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE SUCH PROGRAM EVEN IF IBM OR AN AUTHORIZED IBM PERSONAL COMPUTER DEALER HAS BEEN ADVISED OF THE POSSIBLITY OF SUCH DAMAGES, OR FOR ANY CLAIM BY ANY OTHER PARTY.

SOME STATES DO NOT ALLOW THE LIMITATION OR EXCLUSION OF LIABILITY FOR INCIDENTAL OR CONSEQUENTIAL DAMAGES SO THE ABOVE LIMITATION OR EXCLUSION MAY NOT APPLY TO YOU.

GENERAL

You may not sublicense, assign or transfer the license or the program except as expressly provided in this Agreement. Any attempt otherwise to sublicense, assign or transfer any of the rights, duties or obligations hereunder is void.

This Agreement will be governed by the laws of the State of Florida.

Should you have any questions concerning this Agreement, you may contact IBM by writing to IBM Personal Computer, Sales and Service, P.O. Box 1328-W, Boca Raton, Florida 33432.

YOU ACKNOWLEDGE THAT YOU HAVE READ THIS AGREEMENT, UNDERSTAND IT AND AGREE TO BE BOUND BY ITS TERMS AND CONDITIONS. YOU FURTHER AGREE THAT IT IS THE COMPLETE AND EXCLUSIVE STATEMENT OF THE AGREEMENT BETWEEN US WHICH SUPERSEDES ANY PROPOSAL OR PRIOR AGREEMENT, ORAL OR WRITTEN, AND ANY OTHER COMMUNICATIONS BETWEEN US RELATING TO THE SUBJECT MATTER OF THIS AGREEMENT.

International Business Machines Corporation

P.O. Box 1328-W
Boca Raton, Florida 33432

6936562

Printed in United States of America