HP 64873

V-Series Cross Assembler/ Macro Preprocessor

Reference

HEWLETT PACKARD

HP Part No. 64873-97007
Printed in U.S.A.
June 1991

Edition 3

Certification and Warranty

Certification

Hewlett-Packard Company certifies that this product met its published specifications at the time of shipment from the factory. Hewlett-Packard further certifies that its calibration measurements are traceable to the United States National Bureau of Standards, to the extent allowed by the Bureau's calibration facility, and to the calibration facilities of other International Standards Organization members.

Warranty

This Hewlett-Packard system product is warranted against defects in materials and workmanship for a period of 90 days from date of installation. During the warranty period, HP will, at its option, either repair or replace products which prove to be defective.

Warranty service of this product will be performed at Buyer's facility at no charge within HP service travel areas. Outside HP service travel areas, warranty service will be performed at Buyer's facility only upon HP's prior agreement and Buyer shall pay HP's round trip travel expenses. In all other cases, products must be returned to a service facility designated by HP.

For products returned to HP for warranty service, Buyer shall prepay shipping charges to HP and HP shall pay shipping charges to return the product to Buyer. However, Buyer shall pay all shipping charges, duties, and taxes for products returned to HP from another country. HP warrants that its software and firmware designated by HP for use with an instrument will execute its programming instructions when properly installed on that instrument. HP does not warrant that the operation of the instrument, or software, or firmware will be uninterrupted or error free.

Limitation of Warranty

The foregoing warranty shall not apply to defects resulting from improper or inadequate maintenance by Buyer, Buyer-supplied software or interfacing, unauthorized modification or misuse, operation

outside of the environment specifications for the product, or improper site preparation or maintenance.

No other warranty is expressed or implied. HP specifically disclaims the implied warranties of merchantability and fitness for a particular purpose.

Exclusive Remedies

The remedies provided herein are buyer's sole and exclusive remedies. HP shall not be liable for any direct, indirect, special, incidental, or consequential damages, whether based on contract, tort, or any other legal theory.

Product maintenance agreements and other customer assistance agreements are available for Hewlett-Packard products.

For any assistance, contact your nearest Hewlett-Packard Sales and Service Office.

Notice

Hewlett-Packard makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Hewlett-Packard shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance, or use of this material.

Hewlett-Packard assumes no responsibility for the use or reliability of its software on equipment that is not furnished by Hewlett-Packard.

UNIX is a registered trademark of UNIX System Laboratories Inc. in the U.S.A. and in other countries.

V20 and V30 are registered trademarks of NEC Corporation.

V25, V33, V35, V40, V50, V53, and V60 are trademarks of NEC Corporation.

Hewlett-Packard Company
Logic Systems Division
8245 North Union Boulevard
Colorado Springs, CO 80920, U.S.A.

RESTRICTED RIGHTS LEGEND Use, duplication, or disclosure by the U.S. Government is subject to restrictions set forth in subparagraph (C) (1) (ii) of the Rights in Technical Data and Computer Software Clause at DFARS 252.227-7013. Hewlett-Packard Company, 3000 Hanover Street, Palo Alto, CA 94304

Printing History

New editions are complete revisions of the manual. The date on the title page changes only when a new edition is published.

A software code may be printed before the date; this indicates the version level of the software product at the time the manual was issued. Many product updates and fixes do not require manual changes, and manual corrections may be done without accompanying product changes. Therefore, do not expect a one-to-one correspondence between product updates and manual revisions.

Edition 1	64873-97001, February 1990
Edition 2	64873-97004, July 1990
Edition 3	**64873-97007, June 1991**

Using This Manual

Your HP 64873 V-Series Cross Assembler/Linker documentation consists of three manuals:

- *64873 V-Series Cross Assembler/Macro Preprocessor Reference*
- *64873 V-Series Cross Linker/Librarian Reference*
- *A 64873 V-Series Cross Assembler/Linker User's Guide*

The Reference Manuals

The two *Reference* manuals are in the same binder. The *Reference* manuals document the basic features of the HP 64873 Cross Assembler/Linker (assembly language, assembler syntax, directives, macros, assembler controls, program segments, and relocation, linker and librarian commands, and so on).

The User's Guide

The *HP 64873 V-Series Cross Assembler/Linker User's Guide* contains information on how to start the HP 64873 product on your host computer. It also gives the command syntax and some short examples to get you started with the product.

In this Book

This documentation is written for the experienced program developer, and assumes a working knowledge of the V-Series family of microprocessors, and the Intel 8087 or NEC 72291 coprocessors.

Several useful and informative program examples and example fragments have been provided to clarify the references.

This manual is intended as a reference for the features of the HP 64873 Advanced Cross Assembler/Macro Preprocessor. However, this documentation does not describe the microprocessor itself, nor does it

teach you how to write working programs. For such information, refer to the following source:

- *NEC 70108(V20) Microprocessor User's Manual*.
- *NEC 70320 (V25) Microprocessor User's Manual*
- *NEC 70136 (V33) Microprocessor User's Manual*

For additional information call (800) 632-3531.

Manual Organization

This manual is organized in two sections. The first nine chapters of this manual describe the asv20/asv33 assembler. The last four chapters describe the apv20/apv33 macro preprocessor. There are several appendixes that contain information about both the assembler and macro preprocessor including a description of the acvtv20 porting tool. This nonsupported porting tool can help with translation of source files from the HP 64853 dialect to the HP 64873 dialect. Another nonsupported tool is intel2nec, which can assist in the translation of 8086 assembly source to NEC V20 assembly source. The information in this manual consists of the following topics:

Assembler Information:

- Functional description and list of features.
- Assembly language syntax, character set, symbols (including reserved words), and constants.
- Symbol and expression attributes.
- Alphabetical description of the assembler directives on pages specially formatted for quick reference.
- Thorough discussion of assembler expressions, operands, and a list of the V-Series, Intel 8087, and NEC 72291 instruction mnemonics with accepted operands.
- Description of assembler control statements, including primary and general controls.

- Description of assembler listings and symbol table listings.

Macro Preprocessor Information:

- Introduction to the macro preprocessor and macro functions.
- Discussion of the elements of macro expressions.
- Description and reference for the pre-defined apv20/apv33 macro functions. How to create user-defined macros and how they are treated by the macro preprocessor.

Notes

Contents

1 Assembler Introduction

Introduction 1-1
Instruction Set 1-1
Target Microprocessors 1-1
Assembler Operation 1-2
File Formats 1-2
  Input File Characteristics 1-2
  Output File Characteristics 1-3
asv20/asv33 Features 1-3
  Macro Preprocessor 1-4

2 Assembler Syntax

Introduction 2-1
Assembler Character Set 2-1
Symbols 2-2
  Symbol Formation 2-2
  Keywords 2-3
  Instruction Mnemonics 2-6
  Codemacro 2-6
  Label 2-6
  Variable 2-6
  Structure Name 2-7
  Structure Field Name 2-7
  Record Name 2-7
  Record Field Name 2-7
  Segment Name 2-7
  Group Name 2-8
  EQU Symbols 2-8
Constants 2-8
  Integer Constant 2-8
  Real Constant 2-10
  Character Constant 2-11
Delimiters 2-12
Assembler Statements 2-12

General Syntax 2-12
Comment 2-13
Continuation Lines 2-14

3 Symbol and Expression Attributes

Introduction 3-1
TYPE 3-2
OFFSET 3-2
BASE 3-3
INDEX 3-3
SEGMENT 3-4
SEGMENT RELOCATION 3-4
RELOCATION TYPE 3-4
SEGMENT ADDRESSABILITY 3-5
PS ADDRESSABILITY 3-6

4 Assembler Directives

Introduction 4-1
Syntax Conventions 4-1
EXTRN 4-2
Segmentation Directives 4-3
Program Segmentation 4-3
Default Segment - ??SEG 4-4
Data Definition Directives 4-5
Data Objects 4-6
Program Linkage Directives 4-6
Program Linkage 4-7
ASGNSFR 4-8
ASSUME 4-10
DB, DW, DD, DS, DQ, DL, DT 4-13
END 4-22
EQU 4-24
EVEN 4-28
EXTRN 4-29
V33 Considerations 4-32
GROUP 4-33
LABEL 4-36
NAME 4-38
ORG 4-39
PROC/ENDP 4-40
PUBLIC 4-43

PURGE 4-44
RECORD 4-46
Allocating Record Storage 4-48
SEGMENT/ENDS 4-50
Multiple Definitions of a Segment 4-53
SETIDB 4-56
STRUC/ENDS 4-58
Allocating Structure Storage 4-59

5 Expressions

Introduction 5-1
Reference Syntax Conventions 5-1
Expression Overview 5-2
Absolute Expression 5-2
Relocatable Expression 5-3
External Expression 5-3
Expression Operands 5-4
Numeric Values 5-4
Memory and Register Expressions 5-7
EQU 5-10
Expression Operators Introduction 5-11
Arithmetic Operators 5-11
Unary Plus, Unary Minus 5-11
Binary Addition, Subtraction 5-12
[ ] Square Brackets 5-13
. (Dot operator) 5-14
Multiplication, Division, Modulo 5-15
SHL, SHR 5-16
HIGH, LOW 5-17
Logical Operators 5-19
AND, OR, XOR 5-19
NOT 5-20
EQ, NE, LT, LE, GT, GE 5-20
Memory Operators 5-22
SHORT 5-22
THIS 5-22
PTR 5-23
Segment or Group Override 5-24
OFFSET 5-25
SEG 5-26
TYPE 5-27

LENGTH 5-28
SIZE 5-29
Record Operators 5-31
MASK 5-31
WIDTH 5-32
Segment and Group Operators 5-33
SMOFFSET 5-33
GROFFSET 5-33
SMSIZE 5-34
GRSIZE 5-35
Operator Precedence 5-36

6 Instructions and Operands

Introduction 6-1
Operand 6-1
Accepted Operands 6-1
Operand Positioning 6-3
Immediate Values 6-3
Registers 6-4
Memory Expressions and the MODRM Byte 6-8
Segment Addressability and Overrides 6-10
Addressability Checking 6-10
Default Segments 6-11
Segment Overrides 6-11
Improper Uses of Segment Overrides 6-12
Segment Override Byte 6-12
Overrides and Checking Against ASSUME 6-12
Segment Override Byte Generation 6-13
The V25 Family of Processors 6-14
The Instruction Set 6-21
asv20 Assembler Instruction Set 6-23

7 Assembler Controls

Introduction 7-1
General Syntax for Assembler Controls 7-2
Primary and General Controls 7-2
Controls on the Command Line 7-2
Control Conflicts 7-3
Controls and File Names 7-3
Control Abbreviations 7-3
Controls and Macro Preprocessor (apv20/apv33) 7-3

Primary Controls 7-4
[NO]CAPITALS 7-4
DATE(string) 7-4
[NO]DEBUG 7-4
[NO]ERRORPRINT (filename) 7-5
EXTERN_CHECK 7-5
GROUP_INFO 7-5
[NO]HLASSYM 7-6
[NO]MACRO(string) 7-6
MOD087 7-6
MOD287 7-6
MOD72291 7-6
MODV20 7-6
MODV25 7-7
MODV33 7-7
[NO]OBJECT (filename) 7-7
OPTIMIZE 7-7
PAGELENGTH(n) 7-8
PAGEWIDTH(n) 7-8
[NO]PAGING 7-8
[NO]PRINT(filename) 7-8
[NO]SYMBOLS 7-8
[NO]TYPE 7-9
[NO]UNREFERENCED_EXTERNALS 7-9
WARNING 7-9
WORKFILES(...) 7-9
[NO]XREF 7-9
General Controls 7-11
EJECT 7-11
[NO]GEN 7-11
GENONLY 7-11
INCLUDE(filename) 7-11
[NO]LIST 7-12
RESTORE 7-12
SAVE 7-13
TITLE(string) 7-13
Operational Differences in the Different Modes 7-14
V20 Mode 7-14
V25 Mode 7-14
V33 Mode 7-14
8087 Mode 7-14

80287 Mode 7-14

8 Assembler Listing Description

Introduction 8-1
Assembly Listing 8-1
Cross Reference and Symbol Table Format Description 8-5
Label 8-6
Type 8-6
Value 8-7
Cross Reference 8-8

9 Codemacros

Overview 9-1
Referencing Codemacros 9-1
Alphabetical Listing of the Codemacro Directives 9-2
Codemacro Directives 9-3
CODEMACRO 9-3
ENDM 9-6
Codemacro Matching 9-6
The Specmod Field 9-8
Range Specification 9-12
Examples: 9-13
Codemacro Matching Examples 9-14
Expressions in Codemacros 9-16
Syntax: 9-16
Directives within Codemacros 9-17
DB, DD, DW 9-18
MODRM 9-20
Syntax 9-20
NOSEGFIX 9-21
Record Name Initialization 9-22
RELB, RELW 9-23
RFIX, RFIXM, RNFIX, RNFIXM, RWFIX 9-24
SEGFIX 9-26

10 Macro String Preprocessor Introduction

Introduction 10-1
Input Source Characteristics 10-2
The Metacharacter '%' And The Call Pattern 10-2
Metacharacter Syntax 10-4

Literal Character * 10-5
Input Parsing 10-5
Output Buffering 10-6
Include Files 10-6

11 Elements Of Macro Expressions

Introduction 11-1
Character Set 11-2
Numbers 11-3
Symbols 11-3
Balanced Text String (baltex) 11-4
Expressions and Operators 11-4
HIGH, LOW 11-5
NOT 11-6
Add (+), Subtract (—) 11-6
Multiply (*), Divide (/), MOD 11-6
SHL, SHR 11-7
AND, OR, XOR 11-7
EQ, LE, LT, GE, GT, NE 11-7

12 Pre-Defined Macro Functions

Introduction 12-1
Pre-Defined Macro Functions 12-1
%' (Comment Function) 12-2
%n and %((Escape and Bracket Functions) 12-3
%EQS, %NES, %LTS, %LES, %GTS,%GES 12-5
%EVAL 12-6
%EXIT 12-6
%IF (Conditional Assembly Function) 12-7
%LEN 12-8
%MATCH 12-9
%METACHAR 12-11
%REPEAT 12-12
%SET 12-13
%SUBSTR 12-14
%WHILE 12-14
Example Problem 12-15

13 User-Defined Macros

Introduction 13-1

%DEFINE 13-2
Macro Reference 13-4
What is Output? 13-6
Referencing Macro-time Symbols 13-7

A Error Message Formats

Error Classes A-1
Warning A-1
Error A-1
Fatal Error A-1

B Assembler Error Messages

Introduction B-1
Syntax Errors B-1

C Macro String Preprocessor Error Messages

Error Codes and Messages C-1

D ASCII Codes

E Converting HP 64853 Assembly Language Programs

Introduction E-1
acvtv20 Introduction E-2
Assembler Differences E-2
IF E-3
EQU E-3
MACRO E-3
REPT E-4
SET E-4
External Declarations E-4
Porting Procedure— Main Files with INCLUDE Files E-6
acvtv20 Warnings, apv20 Errors, asv20 Errors E-7
Code Substitution E-8
BIN, DECIMAL, HEX, OCT E-10
BIN E-10
DECIMAL E-10
HEX E-10
OCT E-10
V25/35 Considerations E-11
Manual Macro Translations E-12

.IF, .GOTO, and .NOP Directives E-12
Looping Structures E-13
Numeric, String, and Null Comparisons E-13
Indexed Parameters E-14
Macro Calls E-15
acvtv20(1) Command Syntax E-16
Old and New List E-20
ASCII E-20
ALIGN E-20
ASSUME E-20
COMN E-21
DATA E-21
DBS E-21
DDS E-21
DWS E-21
<EOF> E-22
EQU E-22
EXPAND E-22
EXT E-22
GLB E-23
IF (Macro) E-23
INCLUDE Control E-23
LABEL Directive E-23
Label Field E-24
LIST E-24
MACRO E-24
MASK E-25
NAME E-25
NOLIST E-25
NOWARN E-25
Operator Field E-25
ORG E-26
PROC E-27
PROG E-28
REAL E-28
Reserved Words E-28
SPC E-28
SKIP E-29
TITLE E-29
WARN E-29
* (Comment) E-29

INTEL2NEC(1) E-30

F V-Series Instructions in Hexadecimal Order

G V-Series Instruction Set Summary

FOOTNOTES G-21

Index

Illustrations

Figure 2-1. Syntax for Decimal Real Without Exponent 2-10
Figure 2-2. Syntax for Decimal Real with Exponent 2-10
Figure 4-1. "Partial" Record Definition 4-47
Figure 4-2. Structure Definition and Allocation 4-61
Figure 5-1. SHL Operator 5-16
Figure 6-1. V20/25/33 Registers 6-5
Figure 8-1. Sample Assembler Listing 8-3
Figure 8-2. Cross Reference for Sample Listing 8-5
Figure 13-1. Syntax for User-Defined Macros 13-2

Tables

Table 2-1. Assembler Character Set 2-2
Table 2-2. asv20/asv33 Keywords and Instructions 2-4
Table 5-1. Binary Plus and Minus Results 5-12
Table 5-2. Operator Precedence 5-37
Table 6-1. RAM Register Bank Structure Definitions 6-15
Table 6-2. RAM and Special Function Register Mapping 6-16
Table 6-3. Operand Codes 6-22
Table 6-4. Assembler Instruction Set 6-23
Table 9-1. Codemacro Directives 9-2
Table 9-2. Specmods and Parameter Matches 9-8
Table 9-3. Absolute Number Conversion for Registers 9-12
Table 9-4. Arguments and Actual Parameters 9-14
Table 9-5. Directives within Codemacros 9-17
Table 11-1. Macro Preprocessor Character Set 11-2
Table 11-2. Operator Precedence 11-5
Table 12-1. Predefined Macro Functions 12-2
Table D-1. ASCII Codes D-2
Table F-1. V-Series and 8087 Instructions F-1
Table F-2. 72291 Instructions F-31

1

Assembler Introduction

Introduction

This chapter introduces the assembler by discussing the instruction set, target microprocessors, input and output file formats, and other similar information about the asv20/asv33 Advanced Cross Assembler. This chapter is primarily a brief overview, but it does highlight some important features of the asv20/asv33 assembler. The asv20 assembler is very similar to the asv33 assembler, but they differ in terms of their default targets.

Instruction Set

The asv20/asv33 assembler supports NEC instruction mnemonics, op codes, and syntax for the target microprocessors and thus is compatible with those used in NEC software and documentation.

The supported instruction set is listed in the chapter titled "Instructions and Operands." For further information about the instruction set, refer to the *70108(V20) Microprocessor User's Manual* mentioned in the "Using This Manual" section at the beginning of this manual. References for the V25 and V33 are also noted in that section.

Target Microprocessors

The asv20/asv33 assembler supports the NEC V20, V25, and V33 chip families. The V20 family includes the V20, V30, V40, and V50. The V25 family includes the V25, V35, V25+, and V35+. The V33 family includes the V33 and V53. The asv20 assembler defaults to the V20 and 8087 instruction set, while asv33 defaults to the V33 and 72291 instruction sets.

The asv20/asv33 assembler also translates instructions specific to the Intel 8087 or NEC 72291 floating-point coprocessors for coprocessor execution.

Assembler Operation

asv20/asv33 is a two pass assembler. On the first pass, labels, variables, and other user-defined symbols are examined and placed in an internal symbol table. Additionally, structure definitions are stored.

On the second pass, asv20/asv33 generates the object code, resolves symbolic addresses, and outputs the object module if the assembly was error free. If it was not error free, then asv20/asv33 displays errors on the output listing device and also a cumulative error count. In addition to the object module, asv20/asv33 can also output an HP 64000 format assembler symbol file for use in analysis tools.

The assembly listing produced during pass two contains information pertaining to the assembled program, including opcodes, assembled data, and the original source statements. Based on command line options, asv20/asv33 may also output a symbol table or cross reference table which gives further information not found in the standard assembly listing. Refer to the chapter titled "Assembler Listing Description" for a more complete explanation of the assembly listing and cross reference or symbol table information.

File Formats

Input File Characteristics

The source file input for the asv20/asv33 assembler is a text file containing V-Series instructions, assembler directives, and assembler controls. This file can be produced from an editor or the output file from another component of the HP 64873 package, the apv20/apv33 macro preprocessor.

Output File Characteristics

HP-OMF 86

asv20/asv33 produces a relocatable output object file in HP-OMF 86 format relocatable. HP-OMF 86 format relocatable is a superset of Intel Binary OMF relocatable. HP-OMF 86 format relocatable contains extensions to facilitate code integration and debugging. This format has not been verified to be strictly compatible with Intel Binary OMF relocatable. HP-OMF 86 format relocatable files, therefore, may not work correctly with tools or systems designed to consume Intel Binary OMF relocatable.

HP 64000 Assembler Symbol File

asv20/asv33 can optionally produce an HP 64000 format assembler symbol file. This file is used by analysis tools. The purpose of the assembler symbol file is to preserve the relationship between symbolic names that appeared in the original source file and the memory locations that they referenced.

asv20/asv33 Features

This final section lists some of the notable features of the asv20/asv33 Advanced Cross Assembler. The asv20/asv33 assembler

- generates code for the complete NEC V20, V25, and V33 instruction set
- supports Intel 8087 and NEC 72291 floating-point coprocessor instructions

- permits repeated definition of the same or of different code, data, and constants segments within a single source file
- has high-level-language-like data structures which permit the definition of structured data types and bit fields
- supports symbolic memory references via symbol names
- allows high degree of control over the assembly process (conditional assembly, structured control, listing and output control) through a flexible set of assembly control statements
- gives detailed, well-documented error messages
- produces extensive program listings that can include symbol table/cross reference information
- has a command line interface tailored to the host operating system
- as part of the HP 64873 V-Series Advanced Cross Assembler/Linker package, is well-integrated with the HP 64906 V-Series C Advanced Cross Compiler

Macro Preprocessor

The HP 64873 V-Series Advanced Cross Assembler/ Linker software package also includes a powerful, string-oriented macro preprocessor. The macro preprocessor adds even more flexibility to the assembler with its features (including support for recursive macros).

2

Assembler Syntax

Introduction

Assembly language, like other programming languages, has a character set, a vocabulary, rules of grammar, and conventions that allow for definition of new words or elements. The rules that describe the language are referred to as the "syntax" of the language. This chapter describes the basic elements of assembler language:

- the character set
- symbols
- constants
- delimiters

These basic elements, in turn, are put together to form assembler statements. This chapter also gives the general syntax of those statements.

Input source lines over 1024 characters in length will be truncated and an error message will be generated.

Assembler Character Set

The assembler recognizes the characters in Table 2-1. Any other characters, except those in a comment field, generate errors. Many of the special characters have no previously-defined meaning except as character constants. The characters are case sensitive by default. If case sensitivity is turned off, then all lower case alphabetic characters are treated as if they were upper case, unless they appear in quoted strings.

Table 2-1. Assembler Character Set

Alphabetic Characters

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

a b c d e f g h i j k l m n o p q r s t u v w x y z

Numeric Characters

0 1 2 3 4 5 6 7 8 9

Special Characters

	blank		horizontal tab	>	greater than
$	dollar sign	<	less than	*	asterisk
'	single quote	(	left parenthesis	,	comma
)	right parenthesis	+	plus sign	@	commercial at
-	minus sign	.	period	&	ampersand
:	colon	!	exclamation point	;	semicolon
"	double quote	=	equal sign	#	sharp
?	question mark	%	percent	_	underscore
[	left bracket	]	right bracket	\	back slash
`	accent grave	{	left brace	}	right brace
|	vertical bar	~	tilde	^	caret (uparrow)
		/	slash		

Symbols

Symbol Formation

A symbol is a sequence of characters. The first character must be

- A-Z or a-z (alphabetic)
- ? (question mark)
- @ (commercial at sign)
- _ (underscore)

The second and following characters can be any of these characters or the numerals 0-9. Symbols can be up to 255 characters in length, but only the first 31 characters are significant.

Symbols are used to represent arithmetic values, memory addresses, bit arrays (masks), and so on.

Examples of valid symbols:

```
LAB1
@mask
LOOP_NUM
L2345678901234567890123456789012345
;(entire symbol is stored, but only
31
;used for comparison)
```

Examples of invalid symbols:

```
ABORT* ;contains special character
1LAR   ;begins with a numeric
PAN N  ;embedded blank, symbol is PAN
```

Different symbols represent different kinds of data objects. In general, only a few kinds of symbols are allowed in any particular syntactic construct. Any of the following elements are considered to be symbols.

Keywords

Keywords (also called Reserved Words) are symbols pre-defined by the assembler which you can reference in certain acceptable constructs. Keyword symbols are not user-definable, nor can you create a user-defined symbol with a name that conflicts with a keyword. Keywords include directives and register names, among others. Keywords are not case-sensitive. The full list of assembler keywords appears in the following table. Although the keywords in the table are in upper case, there is no requirement that they appear in upper case in the source code.

Table 2-2. asv20/asv33 Keywords and Instructions

??SEG	BRKEM	DIR	FCMPA	FINCSTP
ABS	BRKV	DISPOSE	FCMPAE	FINIT
ADD4S	BRKXA	DIV	FCMPE	FINT
ADD	BTCLR	DIVU	FCOM	FIP3V
ADDC	BUSLOCK	DL	FCOMP	FIP4V
ADJ4A	BV	DQ	FCOMPP	FIST
ADJ4S	BW	DS0	FCOS	FISTP
ADJBA	BYTE	DS1	FCTW	FISUB
ADJBS	BZ	DS	FCVTDL	FISUBR
AH	CALL	DT	FCVTDS	FL0-FL7
AL	CH	DUP	FCVTLD	FLD1
AND	CHKIND	DW	FCVTLQ	FLD
ASGNSFR	CL	DWORD	FCVTLS	FLDCW
ASSUME	CLR1	EI	FCVTQL	FLDENV
AT	CMP4S	END	FCVTQS	FLDL2E
AW	CMP	ENDM	FCVTSD	FLDL2T
BC	CMPBK	ENDP	FCVTSL	FLDLG2
BCWZ	CMPBKB	ENDS	FCVTSQ	FLDLN2
BE	CMPBKW	EQ	FDECSTP	FLDPI
BGE	CMPM	EQU	FDIAG	FLDZ
BGT	CMPMB	ESC	FDISI	FLOGE
BH	CMPMW	EVEN	FDIV	FMOD
BL	CODEMACRO	EXT	FDIVP	FMOV
BLE	COMMON	EXTRN	FDIVR	FMOVCR
BLT	CVTBD	F2XM1	FDIVRP	FMOVRT
BN	CVTBW	FABS	FDWORD	FMUL
BNC	CVTDB	FACOS	FENI	FMULP
BNE	CVTWL	FADD	FEXPE	FNCLEX
BNH	CW	FADDP	FEXPEMI	FNDISI
BNL	CY	FAR	FEXPR	FNEG
BNV	DB	FASIN	FFREE	FNENI
BNZ	DBNZ	FATAN2	FIADD	FNINIT
BP	DBNZE	FATAN	FICOM	FNOP
BPE	DBNZNE	FBLD	FICOMP	FNSAVE
BPO	DD	FBSTP	FIDIV	FNSTCW
BR	DEC	FCHS	FIDIVR	FNSTENV
BRK	DH	FCLEX	FILD	FNSTSW
BRKCS	DI	FCMP	FIMUL	FPATAN

Table 2-2. asv20/asv33 Keywords and Instructions (Cont'd)

FPO1	FXAM	MOVBKB	PURGE	SHL
FPO2	FXCH	MOVBKW	PUSH	SHORT
FPOWER	FXTRACT	MOVSPA	QWORD	SHR
FPREM	FYL2X	MOVSPB	RECORD	SHRA
FPTAN	FYL2XP1	MUL	RELB	SIZE
FPTW	GE	MULU	RELW	SMOFFSET
FQWORD	GROFFSET	NAME	REP	SMSIZE
FR0-FR7	GROUP	NE	REPC	SP
FREM	GRSIZE	NEAR	REPE	SS
FRND	GT	NEG	REPNC	ST
FRNDINT	HALT	NIL	REPNE	STM
FRPOP	HIGH	NOP	REPNZ	STMB
FRPUSH	IN	NOSEGFIX	REPZ	STMW
FRSTOR	INC	NOT1	RET	STOP
FS0-FS7	INM	NOT	RETI	STRUC
FSAVE	INPAGE	NOTHING	RETRBI	SUB4S
FSCALE	INS	OFFSET	RETXA	SUB
FSIN	IX	OR	RFIX	SUBC
FSINCOS	IY	ORG	RFIXM	TBYTE
FSQRT	LABEL	OUT	RNFIX	TEST1
FST	LDEA	OUTM	RNFIXM	TEST
FSTCW	LDM	PAGE	ROL4	THIS
FSTENV	LDMB	PARA	ROL	TRANS
FSTP	LDMW	POLL	ROLC	TRANSB
FSTSW	LE	POP	ROR4	TSKSW
FSTW	LENGTH	PREFX	ROR	TYPE
FSUB	LOW	PREPARE	RORC	WIDTH
FSUBP	LT	PROC	RWFIX	WORD
FSUBR	MASK	PROCLEN	SEG	XCH
FSUBRP	MOD	PS	SEGFIX	XOR
FTAN	MODRM	PSW	SEGMENT	
FTST	MOV	PTR	SET1	
FWAIT	MOVBK	PUBLIC	SETIDB	

Instruction Mnemonics

A full set of instruction names (mnemonics) is pre-defined by the assembler. Instruction names can be removed from the symbol table with the PURGE directive and re-defined as something else. If you do this, the original meaning of the instruction is lost. There are six instructions (the operators AND, NOT, OR, SHL, SHR and XOR) that cannot be removed. A full list of the pre-defined instruction mnemonics, including the argument combinations acceptable for each, appears at the end of the chapter titled "Instructions and Operands."

Codemacro

A codemacro is a user-defined instruction or prefix to an instruction. The output generated from a codemacro can be a new instruction, a mixture of normal instructions, or just about anything that a customer might want (some assemblers define the normal instructions through the use of codemacros). A codemacro can be defined with the same name as an existing instruction or it can have a completely unique name that describes a new operation. Codemacros can be used anywhere that a predefined instruction can be used.

Label

A label is a user-defined symbol denoting the address of an instruction. Labels can be referenced only in the BR and CALL instructions and variations thereof. A label can be defined with the PROC directive or with the LABEL directive, but there is a another way to define a label that is used most often.

The most common way of defining a label is to place a name (followed by a colon) before an instruction mnemonic, which defines it as a label. Labels have certain attributes, but a discussion of those aspects of labels is left to the chapter titled "Symbol and Expression Attributes." Example:

```
THIS_IS_A_LABEL: MOV AW,2
```

Variable

A variable is a user-defined symbol denoting the address of a location to be used for data storage. Unlike many other assembly languages, asv20/asv33 distinguishes between a label and a variable. They are defined according to syntax and cannot be used interchangeably in expressions or instructions. However, when the LABEL directive is used with the keywords BYTE, WORD, DWORD, FDWORD, QWORD, FQWORD, TBYTE, or with a variable that is a structure name or record name, it defines a variable. When the LABEL directive is used with the type designator NEAR or FAR, it defines a label.

Variables have certain attributes, which are discussed in the chapter titled "Symbol and Expression Attributes."

Structure Name

A structure is a user-defined template describing the manner in which a block of storage is to be broken up into elements. A structure template does not have a storage area associated with it which means that a structure name, while it is still a symbol, is not a variable. A structure template name does not have attributes associated with it.

Structure Field Name

The individual elements of the structure template are called structure fields. Structure fields may be optionally assigned names, but again, since the structure template does not occupy storage, the structure field name is not a true variable. A structure field name, when a structure is allocated using the template, can be used with the dot operator to access an element of the structure, but the structure field name cannot be used alone. Structure field names do not have attributes associated with them.

Record Name

A record is a user-defined template describing how a one- or two-byte block of storage is to be broken up into bit fields. A record template does not have a storage area associated with it which means that a record name is not a variable. Record names do not have attributes associated with them.

Record Field Name

Each bit field describes a number of bits and has a name associated with it. Record field names are not variables, however, and do not have any attributes associated with them.

Segment Name

A segment is a user-defined logical division of the assembly source program. A logical segment can contain code, data, or stack information. Logical segments have names associated with them. These names are used to identify the logical segments to the assembler and loader so that they will eventually be placed together in the same physical segment in memory.

Group Name

A group name identifies a collection of logical segments gathered together because of some common factor. At load time, a group will be placed in memory such that any segment that is a member of the group

will be within 64k of the base of the group. Group names are also significant to the assembler and loader.

EQU Symbols

EQU symbols are names associated with other symbols or expressions through the use of the EQU assembler directive. EQU symbols are simply "replacement names" that can be used anywhere the symbols or expressions they replace could be used. Unlike symbols, however, EQU symbols are not variables and are not allocated storage.

Constants

A constant is an invariant quantity that can be either an arithmetic value or a character constant. Arithmetic values can be represented in either integer or floating-point format.

This section describes integer constants, real constants, and character constants.

Integer Constant

Decimal (base-10) constants can be defined as a sequence of numeric characters optionally preceded by a plus or a minus sign. If unsigned, the value is positive by default.

Internally, the assembler performs arithmetic on 17-bit quantities. A 17-bit value is 16-bit value with the 17th bit (the leftmost bit) as a sign bit. This value may range from -65535 to 65535 (-0FFFFH to 0FFFFH). However, integer constants are only allocated 16 bits when the assembler stores them in the output code. The 17-bit value can be interpreted as a signed or unsigned value and stored in one or two bytes.

A one byte constant can contain an unsigned number with a value from 0 to 255. A two byte unsigned number can range from 0 to 65535. When a constant is negative, its equivalent twos complement representation is generated and placed in the field specified. A 1-byte twos complement number can range from -128 to +127. A 2-byte twos complement number can range from -32768 to +32767. Whether or not a number is interpreted as a twos complement or an unsigned number is typically up to you.

Integer constants outside this range (-65535 to +65535) can appear only in the DD, DQ and DT directives, and on the right side of an EQU

directive. The legal range is different for each directive, as discussed in the chapter called "Assembler Directives."

Other Bases

Constants with bases other than decimal are defined by specifying a coded descriptor after the constant. In addition, the base may restrict or expand the accepted digits for the constant. The following list is of the available descriptors and their meanings and the range of acceptable digits for each kind of constant. If no descriptor follows a constant, the number is decimal by default.

- B - a binary constant - digits must be either 0 or 1
- O - an octal constant - digits are 0-7 inclusive
- Q - an octal constant - digits are 0-7 inclusive
- D - a decimal constant (the default if no descriptor appears) - digits are 0-9 inclusive
- H - a hexadecimal constant - digits are 0-9 inclusive and the letters A-F (or a-f — either are allowed regardless of case sensitivity)

Note

Hexadecimal constants may not begin with the letters A-F (a-f). In those cases, prefix the constant with a zero.

Examples of acceptable constants:

```
10011B   ;binary constant
25       ;defaults to decimal
constant
-0FFH    ;hex constant - notice
leading 0
1377Q    ;octal constant
255d900h ;hex constant
```

Real Constant

Real constants can only appear in DD, DQ, DT and EQU directives. There are three syntactically distinct ways of defining real numbers.

Decimal Real Without Exponent

See the following figure for the syntax diagram of decimal reals with exponents.

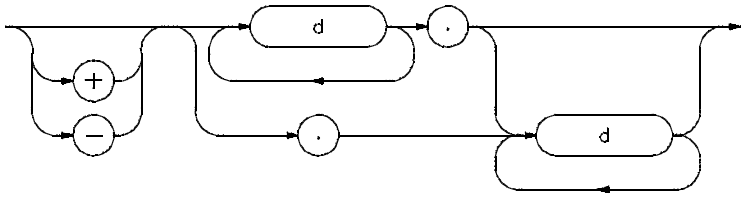


Figure 2-1. Syntax for Decimal Real Without Exponent

E

xamples:

```
1.234
.1234
1234.
```

Decimal Real With Exponent

See the following figure for the syntax diagram for decimal reals with exponents.

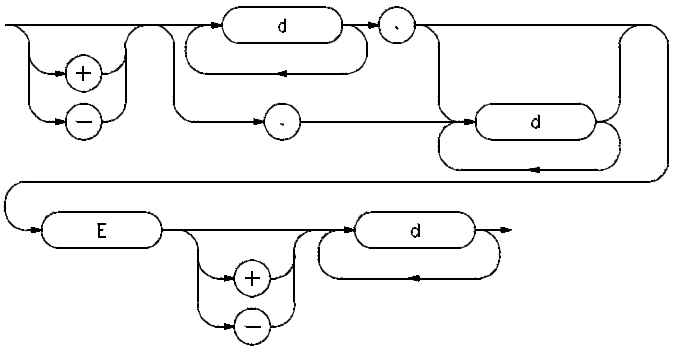


Figure 2-2. Syntax for Decimal Real with Exponent

This format is interpreted to mean that the number to the left of the E is multiplied by 10 raised to the power of the number to the right of the E. Examples:

```
3.14159E-27  ;means 3.14159 * 10^-27
-1e4         ;means -10000.
```

Hex Real

The syntax is 8, 16, or 20 hex digits followed by the letter R (or 9, 17, or 21 hex digits if a 0 must be prefixed to constants with leading hex digits of A-F).

Note that no sign is permitted. This format represents the actual bit pattern to be placed in a variable of type DWORD (8 or 9), QWORD (16 or 17), or TBYTE (20 or 21). (Intel's documentation describes the bit patterns used to represent real numbers.) Examples:

```
40490FDBR
0c0000000r
```

Character Constant

An ASCII character constant is specified by enclosing one or two characters within single or double quotation marks. The constant is encoded as a 16-bit number stored in different ways depending upon usage.

A character string of arbitrary length can be specified with the DB assembler directive.

A more complete discussion of character constants in contained in several of the chapters that follow.

Delimiters

The characters "blank" and "tab" are referred to as delimiters and are generally ignored by the assembler.

Note

There must be at least one delimiter between adjacent symbols and/or numeric constants to prevent them from being interpreted as a single item.

Delimiters are significant in character strings. Delimiters are not required between characters that have special meaning to the assembler (such as [, +, =, $, and so on).

Assembler Statements

General Syntax

The basic elements just described are put together to create statements and instructions that the assembler understands. The rules that govern the ways that statements may be formed are called syntax rules. The general syntax for an asv20/asv33 assembly language instruction statement is as follows:

```
[ label : ] [ prefix ] keyword [ operand [ , ... ] ] [;comment]
```

Each field in the general syntax has one or more of the delimiters discussed in the previous section between it and adjacent fields. Each field has a different purpose.

Label

The label is optional and, if present, identifies or marks the offset of the instruction. This label may be used as a destination in CALL, BR or conditional branch instructions. Notice the colon following the label. It must be present if the label is present.

Prefix

The prefix, if present, causes looping with string instructions or forces a bus lock during the instruction's execution. New prefixes can be defined through the use of codemacro definitions.

Keyword

Keywords can be any of the instruction mnemonics (a list of instruction mnemonics appears at the end of the chapter titled "Instructions and Operands"), codemacros defined by the user, or an EQU symbol set to an instruction or codemacro name.

Operand

An operand is an argument to the instruction in the keyword field. Commas separate multiple operands. Operands are discussed more completely in the chapter titled "Instructions and Operands."

Comment

The comment begins with a semicolon and continues until the end of the line. Comments are used to make "notations" about the assembly language code so that you or others may better understand the purpose of the code or how it works.

Comment

Comments can appear after instructions, assembler directives, control statements, macro definitions, or on lines by themselves. In fact, comments can appear anywhere in the assembly source file as long as they are preceded by semicolons. Comments are not processed by the assembler, but are passed through to the assembler listing.

When a comment is on a line by itself, a leading semicolon must be the first non-blank character (tabs are considered blank characters) on the line. The comment follows it. The comment is considered to continue to the end of the line.

Blank lines are also treated like comments, but they do not require semicolons to lead them. Blank lines appear in the output listing as blank lines. Blank lines may make the code more readable.

Continuation Lines

Some assembler statements will not fit on a single line. If a statement will not fit on a single line, it may be continued to the next line by beginning the next line with the ampersand (&) character. The ampersand must be in column one of the next line. Symbols, numbers, and strings cannot be broken across lines. It is not acceptable to use the ampersand to continue a comment line. In most cases, an error is likely to occur. Simply begin the new line with a semicolon to make it another comment line. Similarly, blank lines cannot be continued with the continuation character.

3

Symbol and Expression Attributes

Introduction

In the chapters that follow, frequent reference will be made to the attributes of variables, labels, and expressions. In order that those references may be understood, this chapter introduces attributes, identifies them, and explains their uses.

Symbols and expressions have certain attributes that determine where they may be used with an instruction and what object code will be generated if they are used. Most attributes are only important when a symbol or expression involves a relocatable or external value. Absolute values will not involve most attributes since absolute values are not modified by the loader.

There are nine possible attributes that a symbol or expression can have. They are

- TYPE
- OFFSET
- BASE
- INDEX
- SEGMENT
- SEGMENT RELOCATION
- RELOCATION TYPE
- SEGMENT ADDRESSABILITY
- PS ADDRESSABILITY

Not all attributes will apply in all cases, however. The following sections discuss the different attributes and how they affect symbols and expressions.

TYPE

The TYPE attribute may belong to either a variable, label, or memory expression. The fixed types are

- BYTE (1 byte)
- WORD (2 bytes)
- DWORD /FDWORD (4 bytes)
- QWORD /FQWORD (8 bytes)
- TBYTE (10 bytes)
- FAR (same or different segment)
- NEAR (same segment)

User-defined types are also possible and are created when a record or structure template is defined. See the chapter titled "Assember Directives" for more about records and structures.

It is possible for a memory expression to not have a type. Instead, the type is determined by using the expression. These explicitly typeless memory expressions are the so-called anonymous references.

OFFSET

The OFFSET attribute for a variable, label, or memory expression is the offset from the start of a segment or group. It is simply the number of bytes from the start of the segment or group. If the variable or label belongs to a noncombinable segment or if the expression was generated from a numeric value, the offset will be absolute. If the

variable or label belongs to a combinable segment or to a group, the offset will be relocatable.

BASE

The BASE register may be set as part of a memory reference. If a base register is used as part of an expression, the expression is known as a register expression, to set it apart from the simpler memory expression.

The base registers are BW and BP. Only one of these registers may be present in a any single register expression, although an index register may be present with the base register. If a base register *is* used in a memory expression, its contents are added to the memory offset at run-time to calculate a final offset for a memory location. If both a base and index register are present in the memory expression, then their values are first added together and then added to the offset to produce the memory reference. If the memory expression does not have a SEGMENT attribute (i.e., no variable, label, or segment override was used as part of the expression), then a default segment register will be used depending upon which base register appears in the register expression. If the BW register is used, DS0 is the default segment register. If BP is used, the default is SS. The default to SS for BP holds even if an index register is also present in the memory expression.

INDEX

The INDEX register may also be used as part of a memory reference. If an index register is used as part of an expression, either with or without a base register, then the expression is known as a register expression, to set it apart from the simpler memory expression.

The valid index registers are IX and IY. Only one index register can be present in a single register expression. It is also possible, of course, that no index register will be used. If an index register is used in a register expression, its contents are added, at run-time, to a memory offset to calculate a final offset for a memory location. If both an index and base register are used in a register expression, both registers are added to the offset to calculate the final offset. If the memory

expression does not have a SEGMENT attribute and no base register is used, then the DS0 segment register is used as a default.

SEGMENT

The SEGMENT attribute determines which segment a variable, label, or memory expression belongs to. The segment attribute is the base value of that segment. The base value is absolute if the segment has been placed using the AT keyword. Otherwise, it is a relocatable value until load time. (This attribute is also the value that is returned by using the SEG operator.)

SEGMENT RELOCATION

The SEGMENT RELOCATION attribute becomes important when a variable, label, or memory expression belongs to a group. In contrast to the SEGMENT attribute, this attribute determines which *group* the item belongs to. The SEGMENT attribute identifies which segment within the group the item belongs to. These two values must be known to correctly calculate offsets for a memory expression. Normally, this attribute is the same as the SEGMENT attribute unless the expression contains a group override. This attribute can be ignored unless groups are used.

RELOCATION TYPE

The RELOCATION TYPE is determined by a combination of the type of an expression and by operators that are applied to it. This value will be null if the expression can be completely determined at assembly time. This is true of offsets within non-combinable segments and for segment bases of segments that use the AT keyword. This value will be set, however, if the item is an offset from either a combinable segment or a segment base for a non-located segment or group. The possible types of relocation are:

- OFFSET: This type of relocation will generate the offset of a variable, label, or memory expression as part of the object

code. A 16-bit offset value will be calculated by the loader and inserted into the object code. The offset will be calculated relative to the base of the segment or, if a group override is used, relative to the base of the group. It is possible to add a 17-bit value to this offset.

- BASE: This type of relocation causes a 16-bit base value to be written directly to the object code. The base will be the base address of the segment that the variable, label, or memory expression belongs to unless a group override is used. In that event, the base will be the base address of the group. It is possible to add a 17-bit value to this base.

- HIGH: This type of relocation causes the upper 8-bit portion of an offset to be written to object code. The offset is calculated using the same rules as noted above, but only the high byte will be written out. It is possible to add an 8-bit value to this byte.

- LOW: This type of relocation causes the lower 8-bit portion of the offset to be written to object code. The offset is calculated using the same rules as noted above, but only the low byte will be written out. It is possible to add an 8-bit value to this byte.

SEGMENT ADDRESSABILITY

The SEGMENT ADDRESSABILITY of a memory location is determined by the segment the memory location belongs to and by any segment or group overrides applied. If a segment override is used to name a specific segment register, that register is used to address the memory location. Otherwise, the values found in the ASSUME directives must be tested. If the segment or group is found through the current ASSUME values, then that segment register is used to address that memory location. If no match is found, an error is generated, since the memory cannot be accessed.

It is possible to have a memory location that does not belong to a segment or group. This would be true of an anonymous memory reference, which looks like

```
[BW][IX]
; base and index registers
```

In such a reference, the segment addressability will be determined by using the default segment registers defined for the base and index registers. Recall that the default segment register will be DS0 unless the BP base register is used, in which case the default will be the SS segment register.

PS ADDRESSABILITY

The PS ADDRESSABILITY of a label is determined from both the current ASSUME value for the PS register, and any segment or group overrides that are applied to the label.

4

Assembler Directives

Introduction

This chapter describes the asv20/asv33 assembler directives. In an assembly language program, assembler directives are written as any other program statement might be, but directives are not translated into equivalent machine language instructions. Instead, assembler directives are interpreted as *instructions to the assembler* to control the program assembly process itself.

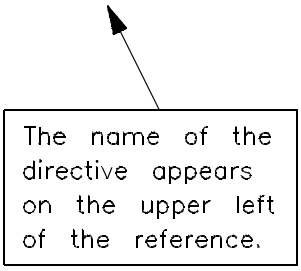
In this chapter, directives are organized in alphabetical order for easy reference. (The DB, DW, DD, DS, DQ, DL, and DT directives are described together because of their similarity.) However, assembler directives may also be grouped into three broad categories —Segmentation Directives, Data Definition Directives, and Program Linkage Directives— which identify the parts of the assembly process the different directives are designed to affect. Segmentation Directives help you to inform the assembler about the logical organization of your program. Data Definition Directives control the allocation and initialization of data, variables, and labels. Program Linkage Directives make it possible for you to create modular assembly language programs. The first sections of this chapter list the directives grouped by these three categories, briefly describe their functions, and more thoroughly discuss some concepts important to understanding how these directives work.

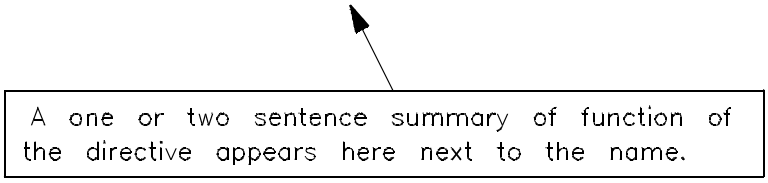
Syntax Conventions

This section gives the syntax conventions used in this chapter. Part of the EXTRN Directive reference follows with explanations of the different areas of the reference.

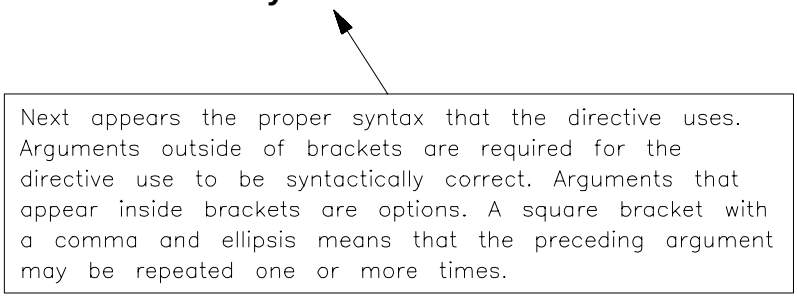
EXTRN

The EXTRN directive is used to declare certain symbols as external references.

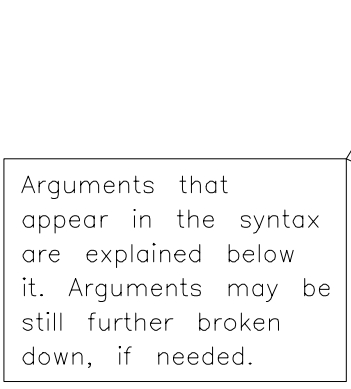




Syntax:



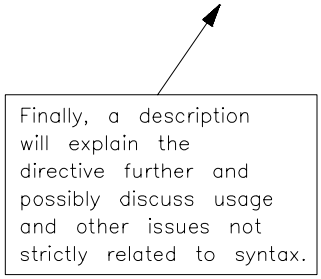
Where: **name** is a symbol.....



segment - unknown unless

Description

Symbols declared as **EXTRN** are not expected to be defined in the current module (they cannot be), but are passed to the loader to be matched against symbols declared **PUBLIC** in other modules. In asv20/asv33 the EXTRN directive will specify the name of the symbol and its associated type. The type declaration must...



Segmentation Directives

ASSUME informs the assembler of the contents of the segment registers.

GROUP combines several logical segments together.

SEGMENT/ENDS defines a logical segment in the assembly language program code.

These directives control program segmentation (the dividing of the assembly program into logical parts). To better understand program segmentation, read the following discussion.

Program Segmentation

The V20, V25, and V33 can directly address one megabyte of memory. (For the V33, there is only one megabyte of memory addressable at any specific moment.) This memory is viewed by the CPU through four segments, known as physical segments, each containing up to 64K bytes. The start of each segment is defined by a value, called a paragraph number, placed in one of the four special registers known as

segment registers. A paragraph number, or boundary, is located at a memory address which is divisible by 16 (that is, the least significant hexadecimal digit of the address is 0H). A physical segment is said to be *active* if one of the segment registers contains the base address of the start of the segment.

The four segments are classified as the code, data, stack, and extra segments. They are each pointed to by a separate segment register:

PS for code

DS0 for data

SS for stack

DS1 for extra

Executable instructions will be in a physical segment defined by the value in PS. Any stack operation will occur within the segment defined by SS. Data is generally found in the segment pointed to by DS0, but it can also be placed in any of the other segments. The segment accessed through the DS1 register will usually hold data also.

A logical segment is a segment as defined within a single assembly file. The linking loader can combine this logical segment with other segments of the same name to form a single physical segment. The size of the physical segment is limited to 64K, so the sum of the logical segments cannot exceed this limit. The collection of segments into a group is another form of physical segment.

Default Segment - ??SEG

All code and data within a source file must exist within some segment. Any code or data defined outside of segment directives within a source file will be assigned to a segment automatically created by the assembler. This segment is named ??SEG and exists in all object files. The ??SEG segment is defined to be public, so it is combined with all other ??SEG segments from other modules. It is also defined to be paragraph aligned.

Data Definition Directives

DB defines one byte of storage.

DW defines one word (two bytes) of storage.

DD defines one double word (four bytes) of storage.

DS defines one double word (four bytes) of storage (72291 data types).

DQ defines one quad word of storage (eight bytes - 8087 data types).

DL defines one quad word of storage (eight bytes - 72291 data types).

DT defines one tbyte (ten bytes - 8087 data types) of storage.

EQU assigns a particular value to a symbol.

EVEN aligns code or data with a word boundary.

ORG adjusts the location counter within the current segment.

PROC/ENDP assigns a label to a sequence of instructions.

PURGE causes a user-defined symbol to become undefined.

RECORD defines a record template.

STRUC/ENDS defines a structure template.

Data Definition Directives control the definition and initialization of data and/or storage as labels, variables, records, or structures. The short discussion on data objects that follows may help you to better understand the data definition directives.

Data Objects

The two most referenced data objects are variables and labels. With the Data Definition Directives, you may define these and other data objects in your program. Variables are data items, or areas of memory where

values are stored. Labels allow you to "mark" locations or sections in your code that may be BRed to or CALLed. One use of labels is to define "subroutine" locations in order to create structured programs. Unlike high-level language subroutines, however, scoping of names does not occur and you can "fall into" an embedded "subroutine."

Records and structures may also be defined by this category of directives. Records and structures are alike in that they are user-defined templates for storage allocation and initialization, they are not allocated storage at definition time, the assembler "remembers" what they look like, they can be referenced as often as you like, and each reference generates one or more copies of storage in the format of the template. At the time of the reference, records and structures may optionally have certain of their definition-time default values replaced.

Records and structures are different, however, in their basic makeups. When you define a structure, you specify how many bytes the template covers, how the bytes will be broken up into variables, and what default values will be placed into those bytes at allocation-time. In contrast, a record must be a one or two byte collection of bit fields. When defining a record, you specify how the record is to be broken up into bit fields, and any default values to be placed in the bit fields at allocation-time. The record size depends upon the sum of the number of bits in all the bit fields, which means the total may not exceed 16 bits.

Program Linkage Directives

ASGNSFR specifies which segment contains the V25 SFR and RAM registers.

END specifies the end of an assembly module.

EXTRN specifies symbols defined in other modules.

NAME assigns a name to an assembly module.

PUBLIC specifies which symbols are public.

SETIDB specifies the memory location where the V25 SFR and RAM registers are located.

Program Linkage Directives make it possible for you to create modular assembly language programs. Refer to the discussion of program linkage that follows to better understand the use of these directives.

Program Linkage

asv20/asv33 supplies the necessary directives to support multi-module programs. A program may be composed of many individual modules that can be separately assembled or compiled. Each module may define variables or labels that other modules may use. The Program Linkage Directives are the mechanisms in asv20/asv33 for communicating symbol information from module to module, for identifying those symbols within the current module that may be used by other modules, for stating what symbols (defined elsewhere) can be used within the current module, and for uniquely naming different object modules that are to be linked together. Using these directives, you may specify a "main module," that is, a module which contains the code that will be initially executed upon loading the program (the address the loader will use to initialize the start address of the program). At the same time, you may also supply initialization values for other segment registers.

The ASGNSFR and SETIDB directives are used for linking together multiple modules when the target processor is a member of the V25 family. These directives are used to inform the linker as to where in memory the V25 RAM and SFR registers are to be found. Any disagreements will result in error messages at link time.

ASGNSFR

The ASGNSFR directive is used to inform the assembler as to which segment contains the V25 SFR and RAM registers.

Syntax:

```
ASGNSFR segmentname
```

Where:

segmentname is the name of a segment in the assembly file.

Description:

The ASGNSFR directive must be used whenever the V25 SFR or RAM registers are to be accessed within an assembly file. This directive informs the assembler to pass information to the linker as to whether V25 registers were accessed in this file and which segment they were associated with. All references to the V25 SFR or RAM registers will result in relocatable values that are placed in the object file. The ldv20 linker will resolve these values by comparing the start address for the ASGNSFR segment with the SETIDB address of the SFR and RAM registers. Any reference to the SFR or RAM registers must be within 64k of the start address for the ASGNSFR segment.

The ASGNSFR directive allows the use of the V25 SFR and RAM keywords within the assembler. These keywords refer to the location of the various registers within the V25 register space. Any use of these keywords without an accompanying ASGNSFR directive will result in undefined values and error messages. The relocatable values generated by these V25 keywords will have offset values ranging from 0H to 1FFH. These offsets correspond to the address of the specified register within the V25 RAM and SFR register bank.

The ASGNSFR directive is only valid in the V25 mode. It is an error to use it in the V20 or V33 modes. Also, the ASGNSFR directive may only appear once within an assembly file. At link time, there can be many modules that contain ASGNSFR directives. No errors occur as long as these segments are placed within 64k of the SFR and RAM registers.

ASGNSFR (Cont'd)

An example of using the ASGNSFR directive follows:

```
SFRSEG SEGMENT
SFRSEG ENDS

ASGNSFR SFRSEG

CODE SEGMENT PUBLIC
ASSUME PS:CODE, DS0:SFRSEG

:
:
MOV TM0, 80H; DS0 register used
MOV MD0, 40H; to access these
registers
:
:

CODE ENDS
```

In the example, the SFRSEG segment is assumed to contain the V25 RAM and SFR registers. The instructions that use the V25 keywords will generate relocatable values that indicate an offset within the V25 RAM and SFR register bank. The ldv20 linker will check the placement of SFRSEG, as well as the placement of the V25 RAM and SFR register bank as specified by the SETIDB directive, to make sure that each reference to a V25 register is within 64k of the start of the segment. If this is not the case, a linktime error will be generated.

ASSUME

The ASSUME directive is used to inform the assembler of the contents of the segment registers.

Syntax:

```
ASSUME segreg:segpart [,...]
(or)
ASSUME NOTHING
```

Where:

segreg is one of the segment registers PS,DS0,DS1 or SS.

segpart is one of the following:

- **A segment name**. The base address of the segment is assumed to be in the named register. All data (or code) in the segment is addressable through this register.
  Example:

```
ASSUME PS:CODE, DS0:DATA
```

- **A group name** (must have been previously defined). The base address of the group is assumed to be in the named register. All code or data in all segments in the group are addressable through this register. Example:

```
ASSUME PS:CODEGRP, DS0:DATAGRP
```

- **A forward reference**. Forward references with ASSUME are only allowed for symbols which will be defined as segment names later in the program. When the segment name is later defined, then it may be used to address memory within the segment. Failure to define the segment name will cause an error to be reported.

- **The keyword SEG followed by the name of a previously-defined label, variable or external symbol.** The base address of the segment containing the symbol (which may not be known until link-time) is assumed to be in the

named register. The specified symbol and any other data known to be in the segment are addressable through the register. (For an external symbol defined outside a segment, no such data is known.) Example:

```
ASSUME PS:SEG START, DS0:SEG COUNT
```

- **The keyword NOTHING.** The register is assumed to contain garbage. The register will not be used to address any memory. The format

```
ASSUME NOTHING
```

is also legal; this is equivalent to

```
ASSUME PS:NOTHING,DS0:NOTHING,DS1:NOTHING,SS:NOTHING
```

Description: ASSUME is used by the assembler to

- determine if the code or data your program references is addressable
- decide whether a segment override byte should be generated.

Initially, the segment registers contain NOTHING (garbage) by default. The assembler assumes the contents of each segment register has not changed —since initialization or the last ASSUME— unless an ASSUME for that register is encountered. ASSUME itself, however, does not alter the value in the segment register. For example, the

ASSUME (Cont'd)

statement 'ASSUME DS0:DATA' does not alter the contents of DS0. You must, at some point, follow the ASSUME with a MOV instruction to DS0 in order to access data in the DATA segment without error.

PS register initialization, since it is done by the loader, does not require a MOV, but PS still requires an ASSUME before it may be used.

Note

There is an exception to the requirement that the PS register must have an ASSUME before it is used. When a BR instruction is used without a current PS-ASSUME value, the default is to ASSUME the current segment. The segment registers will not be checked. This only applies to NEAR references, since a BR to a FAR label requires that the PS register be updated.

DB, DW, DD, DS, DQ, DL, DT

The DB, DW, DD, DS, DQ, DL, and DT directives are used to define variables and/or initialize memory.

Syntax:

```
1 byte (Byte) initialization:
[name] DB init [,...]

2 byte (word) initialization:
[name] DW init [,...]

4 byte (dword) initialization:
[name] DD init [,...]

4 byte (fdword) initialization:
[name] DS init [,...]

8 byte (qword) initialization:
[name] DQ init [,...]

8 byte (fqword) initialization:
[name] DL init [,...]

10 byte (tbyte) initialization:
[name] DT init  [,...]

(or)

[name] Dx repeatval DUP(init,[,...])
     (where x is B, W, D, S, Q, L, T)
```

Where:

name is a unique asv20/asv33 symbol. Its associated attributes will be:

- **segment** - current segment

DB, DW, DD, DS, DQ, DL, DT (Cont'd)

- **offset** - current location counter
- **type** - type of data initialization unit

init may take on many possible values depending upon what type of initialization you wish to do. Init may be any of the following:

- **A constant expression.**
  - **DB** - 1 byte initialization. An integer constant or an expression which fit into 8 bits (either 0-extended or sign-extended) when stored in twos complement format. The range is -255 to +255. High and low relocatable numbers (created by the HIGH and LOW operators) are also acceptable scalars. Other relocatable numbers, such as the offset of a variable, are not acceptable.
    Examples:

```
DB 0
DB 65535  ;not accepted, out of range

DB -1     ;these are equivalent
DB 255    ;both generate hex FF
```

  - **DW** - 2 byte initialization. A constant or expression that evaluates to a number (either absolute or relocatable) which must fit into 16 bits (either 0-extended or sign-extended) when stored in twos complement format. The range is
    -65535 to +65535. Examples:

```
DW 0

DW 65536  ;not accepted, out of range

DW -1     ;these are equivalent
DW 65535  ;and generate hex FFFFH
```

DB, DW, DD, DS, DQ, DL, DT (Cont'd)

– **DD /DS** - 4 byte initialization. An integer constant or an expression that evaluates to an absolute number. The value must fit into 16 bits (either 0-extended or sign-extended). The range is -65535 to +65535. The 16-bit value is stored in the lower 2 bytes in twos complement format (least significant byte first) and the higher 2 bytes are sign-filled. Relocatable numbers are not permitted (it is impossible to determine how to fill the higher 2 bytes at assembly-time).

An integer constant in the range
-4 294 967 295 to +4 294 967 295
(from $-(2^{32}+1)$ to $+(2^{32}-1)$,
but not small enough to qualify for DW. Note that an expression cannot yield a value this large; all expressions evaluate to 17-bit numbers. The value is stored as a 32-bit twos complement integer, low byte first.

A decimal real. The valid range is roughly
-3.4E38 to -1.2E-38, 0, 1.2E-38 to 3.4E38.

A hex real of 8 digits (or 9 digits if its leading digit is 0).

Examples of the possibilities:

```
DD 0            ;yields 00000000
DD 65535        ;yields FFFF0000 (low byte first)
                ;in 16-bit range
DD -1           ;yields FFFFFFFF

DD 65537        ;yields 01000100 (low byte first)
DD -65537       ;yields FFFFFEFF (low byte first)

DD 0.0          ;a decimal real
DD 3.14159      ;another decimal real

DD 0C0000000R  ;a hex real
```

– **DQ /DL** - 8 byte initialization. An integer constant, or an expression whose value resolves to a 17-bit absolute number. The range of constants is $-(2^{64}+1)$ to $+(2^{64}-1)$.

DB, DW, DD, DS, DQ, DL, DT (Cont'd)

Such integer values are stored in 64-bit twos complement format.

A decimal real number which has an approximate legal range of values is
-1.7E308 to -2.3E-308, 0, 2.3E-308 to 1.7E308.

A hex real number consisting of 16 digits (or 17 digits if its leading digit is 0).

– **DT** - 10 byte initialization. An integer constant, or an expression that resolves to a 17-bit absolute number. The range of constants is $-(10^{18}+1)$ to $+(10^{18}-1)$. All integer values are stored in 80-bit signed-magnitude packed decimal (BCD) format, least significant byte in the lowest address.

A decimal real number that has an approximate range of -1.1E4932 to -3.4E-4932, 0, 3.4E-4932 to 1.1E4932.

A hex real number consisting of 20 digits (or 21 digits if its leading digit is 0). Examples:

```
DT 65535    ;generates 3555060000000000000H
            ;(low byte first)
DT -65535   ;generates 3555060000000000080H
            ;(low byte first)
```

- **The character "?" for indeterminate initialization.**
  – In situations where you wish to reserve storage but do not need to initialize the area to any particular value, use the special character "?" instead of a value. The area will be reserved with an indeterminate value. Examples:

```
ABYTE DB ?  ;reserve a byte
AWORD DW ?  ;reserve a word (2 bytes)
ADWORD DD ? ;reserve a double word
(4 bytes)
```

DB, DW, DD, DS, DQ, DL, DT (Cont'd)

```
AFDWORD DS ? ;reserve a double word
(4 bytes)
AQWORD DQ ? ;reserve a quad word (8
bytes)
AFQWORD DL ? ;reserve a quad word (8
bytes)
ATBYTE DT ? ;reserve a tbyte (10
bytes)
```

- **An address expression.**

Note

Assume registers are not checked when these directives are used with address expressions. Therefore, the only way to get a group-relative reference is to use a group override in the address expression.

– **DW** - 2 byte initialization. DW may be used with a variable name, a label name, a group name, or a segment name. Using DW with a variable or label name causes the offset of a variable or label (relative to its segment or, if a group override is used, to its group) to be stored. Using DW with a group or segment name causes the paragraph number of that group or segment to be stored. Examples:

```
DW COUNT     ;COUNT is a variable or label
             ;store offset of COUNT from its segment

DW DATAGRP :COUNT ;store offset of COUNT from its
             ;group (DATAGRP)

DW CODE      ;CODE is a segment or group name
             ;store the paragraph number
```

DB, DW, DD, DS, DQ, DL, DT (Cont'd)

– **DD /DS** - 4 byte initialization. DD may be used with a variable name, a label name, a group name, or a segment name. Using DD with a variable or label name causes the offset (relative to its segment or, if a group override is used, to its group) of the variable or label to be stored in the low order word and the segment or group base address for the label or variable to be stored in the high order word. Using DD with a group or segment name causes the paragraph number of that group or segment to be stored in the low order word. The high order word will be set to 00H. Using DD with a variable or label name is equivalent to storing a pointer to the variable or label address. Examples:

```
DD COUNT      ;COUNT is a variable or label, a
              ;pointer to it is stored
```

is equivalent to

```
DW COUNT      ;store offset of COUNT
DW SEG COUNT  ;store COUNT's segment
```

■ **Initialize with a string.**

– **DB -** 1 byte initialization. A string of up to 1024 characters may be specified with the DB directive. Each character in the string, left to right, is assigned one byte of memory, low address to high address. The string must be enclosed within single or double quotes. A single quote may be embedded in the string by using two consecutive quotes. Examples:

```
ALPHABET DB 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
WITHQUOTE DB 'THIS AIN''T HARD!'          ;inserting single
                                          ;quote in string
```

DB, DW, DD, DS, DQ, DL, DT (Cont'd)

– **DW, DD, DS, DQ, DL,** You may use these directives to code a string of 1 or 2 characters. Such a string is interpreted as a 17-bit number that is stored differently than it would be if DB were used. If two characters are stored, the second character in the string appears in the low byte of the storage and the first character appears in the next higher byte of the storage. If only one character is stored, the low byte of the storage contains the character. With either a 1 or 2 byte string, if any bytes of the storage remain unfilled, they are set to 00H. Using more than 2 characters in a string results in a warning message and only the first 2 characters are used.

– **DT** DT can also code a two character string, but it does it in a way different from the other directives. DT stores the string in BCD packed decimal format. If a single character is stored, its decimal ASCII value is stored in the low byte of storage. The remaining bytes are set to 00H. If two characters are stored, however, it becomes more complicated. It is done as follows:

  The 17-bit hexadecimal number representing the string is converted to its decimal equivalent. (The 17-bit hex number is formed by placing the ASCII hex value of the first character of the 2 character string in the leftmost byte of the 17-bit word and placing the ASCII hex value of the second character in the rightmost byte of the 17-bit word. The sign bit is zero.)

  Beginning with the rightmost digit of the resulting decimal value, the decimal representation is stored 2 digits per byte, working from right to left in the decimal value, until all digits are stored.

Any remaining bytes of storage are set to 00H.

DB, DW, DD, DS, DQ, DL, DT (Cont'd)

Examples:

```
DB '01' ;generates 3031H (shown low byte first)
DW '01' ;generates 3130H (shown low byte first)
DW '1'  ;generates 3100H (shown low byte first)
DD '01' ;generates 3130 0000H (shown low byte first)
DQ '01' ;generates 3130 0000 0000 0000H
        ;(shown low byte first)
DT '01' ;generates 3723 0100 0000 0000 0000H
        ;(shown low byte first)
```

Repeating value. The special construct, DUP, can initialize an area of memory with a repeated value or a repeated list of values.

- **repeatval** specifies the number of data initialization units (from 1 to 65535) to be filled (bytes, words, dwords, qwords, or tbytes depending upon whether Dx is DB, DW, DD, DS, DQ, DL, or DT).
- **init** (as an argument to DUP) may be a single occurrence of the possibilities that were acceptable for **init** in the non-repeating-value syntax, including another DUP, or **init** may be a *list* of these same values. DUPs may be nested to eight levels deep. Below are some examples:

```
WORD1 DB 2 DUP (?)    ;two consecutive bytes form word
DD 2 DUP ('01')       ;generates 3130000031300000H
NESTEDDUP DB 3 DUP (4 DUP (5 DUP (1, 6 DUP (0))))
                      ;60 occurrences of 1,6 DUP (0)
```

DB, DW, DD, DS, DQ, DL, DT (Cont'd)

If an indeterminate initialization is repeated, the memory reserved by that data directive will NOT be initialized to 0. Also, repeating a relocatable value (such as a location in memory) will result in only the first value being assigned correctly. So this practice is discouraged.

Description: The DB, DW, DD, DS, DQ, DL, and DT directives are used to define variables and/or initialize memory. The descriptions of the parts of the syntax adequately describe these directives.

The DD/DS and DQ/DL data directive pairs accept the same form of arguments and generate the same object code. The only differences between these forms are that the DS and DL directives are only usable on a word boundary and cannot appear inside a segment that is BYTE or INPAGE aligned. Errors are generated under those conditions.

The DS and DL directives may only be used when the assembler is in the 72291 mode.

END

The END directive is used to inform the assembler that the last source statement has occurred and to specify the load module starting address for the main module.

Syntax:

```
END [regint [,...]]
```

Where:

regint This field defines the contents for a segment register (and the PC and SP registers). To initialize the segment registers, the field may include some, or all, of the following symbols:

- **segname** is either a segment name or a group name. It identifies the paragraph number to be loaded into the segment register.
- **labelname** is the name of a label defined in the module. If it is used alone, its segment will be used to initialize the PS register and its offset will initialize the PC. If it is used with a segname, then just its offset will be used to initialize PC.
- **varname** is the name a variable defined in the module. Its offset will be used to initialize SP.

  The following examples show the proper syntax for initializing different segment registers.

PS (code) segment register initialization

```
END labelname              ;initializes PS and PC
                           ;(the segment part of the
                           ;label is used for PS)
(or)
END PS:labelname           ;same as 'labelname'
(or)
END PS:segname:labelname   ;the segment part (paragraph
                           ;number) to be loaded into
                           ;PS is taken from segname
```

END (Cont'd)

SS (stack) segment register initialization

```
END SS:segname          ;SP will be initialized to be
                        ;equal to the size of the
                        ;segment
```

(or)

```
END SS:segname:varname ;initializes SS and SP
                       ;(SP will be initialized to
                       ;the offset of varname)
```

DS0 (data) segment register initialization

```
END DS0:segname         ;initializes DS0
```

Description: An END directive is required for all assembly language programs. Any statements that follow the END directive will not be processed. Specifying a load address with the END directive also informs the loader that the current module is the main program. The main program defines the start of execution because execution begins at the address specified with the END directive for the main program. If multiple load modules are combined by the loader, only one module can specify a load address and therefore be considered the main program. (The loader issues warning messages if it encounters more than one main program.)

The END directive may also be used to define the initial contents of the DS0 and SS segment registers by specifying values to be placed in these registers by the linker/loader at load-time.

Note 

If the code is to be targeted for HP 64000 format absolute, you may only initialize the PS:PC register with END. Initialize the other registers explicitly within the code.

EQU

The EQU directive causes the assembler to assign a particular value to a symbol.

Syntax:

```
equate_symbol  EQU expression
```

Where:

equate__symbol is a mandatory symbol defined by this statement.

expression is one of the following items:

- **A numeric constant or expression.** The value of the expression must be determined at assembly time. Any symbols used in the expression must have been previously defined. See the Description section below for more discussion about real constants. Examples:

```
PI EQU 3.14159          ;real constant stored with
                        ;10 byte precision
DD PI                   ;4 byte floating point
DQ PI                   ;8 byte floating point
DT PI                   ;10 byte floating point

E1 EQU 2 + 3            ;numeric expression
E2 EQU E1 AND 4         ;E1 previously defined
E3 EQU (E1 - E2) / 12   ;E1 and E2 previously defined
```

EQU (Cont'd)

- **A variable or label name** (which may be a forward reference).

```
ALABEL EQU ALAB              ;ALAB not defined yet
ALAB: MOV AW, 0
```

- **A register name**, including ST and 72291 registers. Example:

```
COUNT  EQU CW
POINTER EQU BW
MOV COUNT, 10                ;CW = 10
MOV POINTER, OFFSET ARRAY    ;BW = offset of array
FREQ EQU ST(1)
FADD ST, FREQ
```

- **An instruction or codemacro name.**

```
BUMP  EQU INC                ;instruction name
BUMP AW                      ;same as INC AW
```

- **A register expression.** These may be single register expressions, or they may also include a segment override. This construct is useful when defining data items to be accessed on the stack. Refer to the Description section for a more information about the use of register expressions. Examples:

```
STACKWORD EQU WORD PTR SS:[BP + 2]
AVAR EQU [BW + 3]
ANEXTVAR EQU DS1:[BW]
```

EQU (Cont'd)

Description: The EQU directive in asv20/asv33 is more powerful than the EQU found in most other assemblers. All the various attributes of address expressions are stored, and any missing attributes may be added later with expression operators at the time the EQUed symbol is referenced.

Decimal real numbers are stored in a full 10-byte format to prevent a loss of precision; they may be used in DD, DS, DQ, DL, or DT directives later in your code. Hex real numbers, however, are stored in as many bytes as the specification indicates; they can be used later only in the single directive that accepts a hex real of that size.

It is possible for a symbol to appear as a forward reference before it is defined in an EQU. When this happens, the assembler assumes that the forward reference will resolve to a number, variable or label. If this turns out not to be the case, an error may occur on pass 2 if the assembler did not leave enough room for an instruction on pass 1.

Symbol chaining (defining a symbol in terms of another symbol which is in turn defined by another symbol) can be accomplished with the EQU directive, but the chain must eventually end as a numeric or address expression. An error occurs if the definition ends in a register or real number expression. Circular EQU definitions are also errors. Example:

```
A EQU  B
B EQU A          ;ERROR! circular
reference
```

A symbol defined by an EQU to an *address expression* consisting of more than one symbol (for example, BYTE PTR VBL) is stored as a variable or label, if possible. The entire EQU expression takes its attributes from the sub-expression on the right-side of the EQU. However, not all attributes will be set if attributes are missing from the right-side sub-expression. If that is the case, missing attributes must be supplied when the symbol on the left-side of the EQU is used elsewhere in an expression.

Examples:

EQU (Cont'd)

```
A EQU [BW][IX][5]       ;anonymous reference — type
                        ;information must be supplied
                        ;when A used elsewhere
B EQU WORD PTR 10       ;segment information must be
                        ;supplied later
```

EVEN

The EVEN directive causes the Location Counter to be aligned to an even value (a word boundary).

Syntax:

EVEN

Description:

The assembler brings about alignment by generating a NOP (90H) instruction if the current location counter contains an odd address value. The EVEN directive cannot be used in a byte aligned segment. Doing so will cause an error message to be generated.

EXTRN

The EXTRN directive is used to declare certain symbols as external references.

Syntax:

```
EXTRN name:type [,...]
```

Where:

name is a symbol, declared PUBLIC (see PUBLIC directive later in this chapter) in another module, to be defined as an external reference. Its associate attributes are the following:

- **segment** - unknown unless defined within a SEGMENT/ENDS pair
- **offset** - unknown
- **type** - type declared in **type** argument
- **relocation type** - external

type is one of the following:

- The keyword **BYTE, WORD, DWORD, FDWORD, QWORD, FQWORD,** or **TBYTE** for a variable which is one of these types.
- **A structure name.** Names a variable whose type is equal to the number of bytes allocated in a preceding structure definition.
- **A record name.** Names a variable whose type will be either byte or word depending on the preceding record definition.
- **NEAR** or **FAR**. A label of type near or far.
- **ABS.** A constant (17-bit number), always of type word.

EXTRN (Cont'd)

Description: Symbols declared as EXTRN are not expected to be defined in the current module (they cannot be), but are passed to the loader to be matched against symbols declared PUBLIC in other modules. In asv20/asv33, the EXTRN directive will specify the name of the symbol and its associated type. The type declaration must agree with the type of the symbol declared PUBLIC, but the loader does not do type-checking. It is your responsibility to maintain type compatibility.

The type ABS is used to declare a constant. Despite the mnemonic ABS, this number can prove to be offset relocatable or absolute when it is resolved depending upon how it was defined as a PUBLIC symbol. In either case, name can be used and treated like a constant value.

You must be careful in the placement of the EXTRN directive in relation to the definition of the program segment. If you know the segment in which the external symbol was defined as PUBLIC, place the EXTRN directive between a SEGMENT/ENDS pair that is ***identical*** to the SEGMENT/ENDS pair in which the object was defined in the other module. An external symbol defined in this manner will be addressable through the segment register containing the segment in question. In particular, a NEAR label defined EXTRN must be defined in segment identical to the one where it is defined PUBLIC because of the NEAR type restrictions. Example:

EXTRN (Cont'd)

In module "A"

```
DATA SEGMENT WORD PUBLIC
COUNT DB 0                ;declared as byte through DB
PUBLIC COUNT
DATA ENDS
```

In module "B"

```
DATA SEGMENT WORD PUBLIC  ;different module, but same
                          ;segment declaration
EXTRN COUNT:BYTE          ;typed as byte
DATA ENDS
```

If you do not know the segment in which the external symbol is defined, or if the segment in which it is defined is non-combinable, place the EXTRN directive outside of all SEGMENT/ENDS pairs in your program. To address the external symbol you must load the segment part (paragraph number) of the symbol into a segment register using the SEG operator and then either use an ASSUME directive to verify addressability or use a segment override for each use of that symbol.

Note

The V-Series linker does NOT verify that the definition of an external symbol matches the definition of its resolving public symbol. It is up to the user to make sure that external symbol definitions are placed within the correct segment or they should NOT be placed in a segment at all.

Example:

```
MOV AW, SEG COUNT
MOV DS1, AW          ;loads segment

(then)

ASSUME DS1:SEG COUNT ;verify
addressability
MOV DL, COUNT       ;use symbol
(or)
MOV DL, DS1:COUNT    ;use segment
override
```

V33 Considerations

In the V33 mode, there are three forms of external symbols that do not act the same as a normal external symbol. These externals are of the following forms: ?jump?MODULENAME?PROCNAME, ?addr?MODULENAME, and ?pgrn?MODULENAME. The MODULENAME refers to the module name for a V33 executable. The PROCNAME refers to a public procedure or label within the named module.

The ?jump?MODULENAME?PROCNAME external is a FAR external which refers to an address to begin execution. The ?addr?MODULENAME and ?pgrn?MODULENAME are ABS externals and represent 16-bit values that are used for initializing the V33 page table to map the 1-megabyte address space into the V33's 16-megabyte address space.

These externals are not resolved by the ldv33 linker, but are processed by the HP 64875 elv33 locator tool. The definitions of these externals are not affected by their relationship to segment definitions (that is, they do not belong to a segment, even if they are defined within one).

GROUP

The GROUP directive is used to specify several logical segments that are to be placed in the same physical segment.

Syntax:

```
name GROUP segpart [,...]
```

Where:

name is a mandatory, unique, user-defined name for the group.

segpart is one of the following:

- **A segment name.**
- **The keyword SEG followed by the name of a previously-defined variable, label, or external symbol.** This construct refers to the segment in which the specified symbol lies. For externals, this may not be discovered until link-time.
- **An undefined symbol** that must be defined later in the program as a segment name or the assembler reports an error.

Description:

At assembly-time you may specify that certain logical segments will all go in the same physical segment so the assembler will know that all such segments may be accessed from the same segment register. Such a collection of segments is called a *group*. The ordering of the segments in a GROUP directive will not necessarily control or represent the ordering of the segments in memory nor are the segments in a group necessarily adjacent in memory. GROUPing them only implies that they should lie within the same physical segment.

The total address space covered by all segments in a group must be less than or equal to 64K bytes. The size of the group is the equal to the sum of the sizes of all segments in the group. The assembler does not check whether the size of the group is greater than 64K bytes, but the loader does.

GROUP (Cont'd)

A group has a base address. The base address of a group refers to the lowest memory address of any segment in that group. The loader sets the group base address, and all segments in the group are addressable from this same group base address.

Forward references to group names are not allowed.

One of the uses of the group name is with the ASSUME directive. If, for example, you have defined many different data segments that you intend to form into one physical segment when the program is located in memory, you could combine these segments with the GROUP directive. Since the contents of all these data segments will be addressable through DS0 during the execution of the program, you may use the group name in the ASSUME and to initialize DS0. For example,

```
DATAGRP GROUP DATA1, DATA2 ;DATA1 and DATA2 not
                    ;defined yet

DATA1  SEGMENT
ABYTE  DB 0
DATA1  ENDS

DATA2 SEGMENT
AWORD DW 0
DATA2 ENDS

ASSUME DS0:DATAGRP, PS:CODE ;use group name in ASSUME
CODE SEGMENT
MOV AW, DATAGRP   ;AW = base address of group
MOV DS0, AW        ;initialize DS0
MOV AW, AWORD     ;addressable through DS0
.
.
.
CODE ENDS
```

Use of the OFFSET Operator With Groups

When using the OFFSET operator with a variable or label whose segment is in a group, you must use the group name as a segment override in an expression which references that variable or label, as in

```
MOV BW, OFFSET DATAGRP:COUNT
```

Also, if you wish to store the paragraph number of a variable or label defined with a group, you must use a group override. Otherwise, the paragraph number of the segment that contains the variable is stored instead. Example:

```
DW SEG DATAGRP:COUNT
DD DATAGRP:COUNT
```

LABEL

The LABEL directive is used to create a name for the current location of assembly, whether it is data or code.

Syntax:

```
name LABEL type
```

Where:

name is a unique user-defined symbol. Its associated attributes are the following:

- **segment** - current segment
- **PS-assume** - current PS-assume value (labels only)
- **offset** - current location counter
- **type** - as specified below

type is one of the following:

- The keyword **BYTE, WORD, DWORD, FDWORD, QWORD, FQWORD,** or **TBYTE** to create a variable which is one of these types.
- **A structure name** Creates a variable whose type is equal to the number of bytes allocated in a structure definition.
- **A record name** Creates a variable whose type will be either byte or word depending on the record definition**.**
- **NEAR** or **FAR** To create a label of type near or far.

LABEL (Cont'd)

Description: The LABEL directive and the idea of a "label" should not be confused. The LABEL directive creates a label or variable at the current location being assembled. A label is a name for a location in the code that can be BRed to or CALLed.

The LABEL directive is used primarily to address the same data item or same piece of code as different types. As a rule, asv20/asv33 requires that the type of a data reference match the type of the data definition (known as strong typing), which makes this dual addressing difficult. If you want to access a variable either as a word or as 2 bytes depending upon the context, the following would allow you to do so:

```
WORDNAME LABEL WORD
LOWBYTE DB 0
HIBYTE DB 0
```

The LABEL directive also allows you to define two labels of different types (for instance, both NEAR and FAR) but be careful that the right RET is executed for the type of CALL made. The following (potentially fatal) example illustrates this use:

```
AFARLABEL LABEL FAR
NEARLAB: MOV AW, BW
RET                    ;would be near, so some information
                       ;would be left on the stack
```

asv20/asv33 does not, in general, permit data storage at label locations—that makes writing self-modifying code difficult.

The FDWORD and FQWORD labels are only valid if the current offset is word aligned, and the current segment is not BYTE or INPAGE aligned. Errors are generated if either condition is not valid.

The FDWORD and FQWORD types may only be used when the assembler is in the 72291 mode.

NAME

The NAME directive is used to assign a name to an object module.

Syntax:

```
NAME module_name
```

Where:

module_name is a user-defined identifier. The name identifier can be any length, but only the first 40 characters are meaningful.

Description:

Every object module produced by asv20/asv33 has a name; if you do not provide one, the assembler issues a warning and gives the file a special name. The special name is the source file base name stripped of any path and suffix. A module name is not stored as a symbol. You can therefore duplicate a keyword or a user-defined label without conflict. Module names are not affected by the case control. They are always case-sensitive.

The linker does not require that modules have unique names, but it identifies its input files by module name on its listing map. For this reason, assign each module a unique name for clarity.

The librarian program does identify its modules by name. Every module used as input to the librarian must have a unique name or an error will result.

ORG

The ORG directive is used to alter the value of the Location Counter within the current segment.

Syntax:

```
ORG expression
```

expression evaluates to

- an absolute number (modulo 65536) that does not contain forward references or
- an offset relocatable number (modulo 65536) that is only relocatable from the current segment. Using the offset of '$' (dollar sign is the special character for the current location counter value) in a PUBLIC segment is an example of this form of ORG.

Description:

The ORG directive is used to locate code or data at a particular location (offset) within a segment. Using ORG with an absolute segment allows you to specify an actual memory location at which the code or data will be located.

Note 

Avoid expressions of the form

ORG OFFSET ($-1000)

since this particular expression will overwrite your last 1000 bytes of assembly (or will re-ORG high in the current segment if the expression evaluates to a negative number). An expression with the syntax "$+1000" will produce an error because this expression evaluates to a label, not to a number. To achieve what is intended, the expression "OFFSET ($+1000)" can be used.

PROC/ENDP

The PROC/ENDP directive pair is used to delimit a section of code which can then be CALLed from elsewhere in the program, much like a procedure in a high-level language.

Syntax:

```
name PROC [type]
.
.
(instructions)
.
.
name ENDP
```

Where:

name is a unique user-defined symbol providing a label for the beginning of the PROC. The name on the ENDP directive must match that on the most recently defined PROC for which an ENDP was not already encountered. The ENDP directive signals the end of a PROC definition to the assembler. The attributes of the PROC name are the following:

- **segment** - current segment
- **PS-assume** - current PS-assume
- **offset** - current location of PROC directive
- **type** - depends on type indicated
- **relocation type** - depends on enclosing segment

type is the type of the label defined at the beginning of the PROC. Type can be NEAR or FAR. NEAR is the default if no type is specified.

PROC/ENDP (Cont'd)

Description: The primary use of the PROC/ENDP pair is to give a type to the RET instruction enclosed by the pair. A RET instruction generates a NEAR return or a FAR return depending on whether the most recently defined PROC is NEAR or FAR. A RET or RETI outside of a PROC/ENDP pair or inside a pair which has no type specified is, by default, of type NEAR. Therefore, any code you wish to CALL FAR and then successfully RET from should be enclosed in a PROC/ENDP pair typed FAR.

Code execution begins at the instruction immediately following the PROC Directive when PROCs are CALLed or BRed to.

Nested PROCs

When a PROC is defined inside another (nested), it does not necessarily have the same type assigned to its RET or RETI instruction as does the enclosing PROC. For instance, an enclosing PROC may be typed FAR. When the next PROC occurs, it might be a NEAR. For the duration of that PROC until the ENDP, the type of any return instruction will be NEAR and not FAR. When the ENDP is found for the nested PROC, however, the type reverts to the type of the enclosing PROC, in this case FAR. Having a NEAR PROC inside a FAR PROC, then, does not affect the enclosing PROC.

Differences Between PROCs and "Subroutines"

The code in a PROC/ENDP pair is not a procedure in the same sense as it is in high-level languages. A few differences are of note:

- In contrast to the scoping of names in block-structured languages, all labels and variables within the PROC/ENDP pair are not local to the "subroutine", but are global to the entire file.
- It is possible for execution to "fall into" a PROC from the previous instruction; it is not necessary to CALL a PROC to

PROC/ENDP (Cont'd)

execute it. Executing a RET or a RETI from a "fallen into" PROC can cause unpredictable results.

- The ENDP does not function as a return-from-procedure; it marks the end of the PROC for the assembler. It is possible for execution to "fall out of" a PROC through the ENDP into the next instruction. To return from a CALL, a RET or RETI instruction must be used.

PUBLIC

The PUBLIC directive is used to specify symbols, defined in one module, that are available to other modules at link time.

Syntax: PUBLIC name [,...]

Where:

name is the name of the symbol defined in the current module.

Description: Symbols designated PUBLIC will be placed in the object file and used by the loader to resolve external references (made with the EXTRN directive) from other modules.

PUBLIC symbols must be variables, labels or 17-bit constants defined by using EQU; any other types will generate an error. A 17-bit constant can be absolute or offset relocatable only; other relocation types are not allowed.

PURGE

The PURGE directive places a flag on the specified user-defined symbol in the symbol table so that the symbol is no longer recognized.

Syntax:

```
PURGE symbol [,...]
```

symbol can be any keyword or user-defined symbol, *except*

- **register names**
- **segment names** (including ??SEG).
- **group names**
- **hands-off keywords** (see keyword list in chapter titled "Assembler Syntax")
- **any user-defined symbol that appears in a PUBLIC statement**

Description:

A PURGEd symbol can be redefined following the PURGE statement. A reference to the symbol following the PURGE statement, but before a re-definition, is treated as a forward reference to the second definition. If a PURGEd symbol is never redefined, references to the symbol following the PURGE statement are considered errors (reference to undefined symbol).

Purging symbols does not physically remove them from the symbol table and therefore PURGE cannot be used to deal with symbol table overflow.

If a variable or label that is defined in the current module but does not appear in a PUBLIC or EXTRN statement (that is, a local symbol) is purged, it will not appear in the object module. A PURGE directive, placed just before the END statement can —in combination with the $DEBUG assembler control statement— be used to pass on only a few selected symbols for debugging purposes.

PURGE (Cont'd)

Any variable, label or absolute number that was defined by an EXTRN statement can be purged, but the symbol will still appear in the object module as an external reference.

If a symbol is defined by an EQU to another symbol (not an expression), a PURGE on one of the symbols can cause unexpected results. The rule is that if a symbol in a EQU chain is PURGEd, it and all symbols that precede it to the beginning of the chain are also PURGEd.

Given the EQU chain that follows:

```
A EQU B
B EQU C
C DW 0      ;EQU chain resolving at C
```

The following PURGEs, which should not be considered as sequential code but as separate lines somewhere in the assembly source program, would have the described effects.

```
PURGE A    ;purges only A (B and C are still defined)

PURGE B    ;purges A and B (C still defined)

PURGE C    ;purges A, B, and C
```

RECORD

The RECORD directive defines a record template.

Syntax:

```
name RECORD recfieldname:nnn[=datum]
[,...]
```

Where:

name is a mandatory user-defined name for the record template.

recfieldname is a mandatory user-defined name for a bit field.

nnn is an integer constant, or an expression containing no forward references, that evaluates to an absolute number. The range of **nnn** is from 1 to 16, inclusive, and denotes how many bits will be in a bit field. Bits are counted from high bit to low bit within the full byte or word. Thus, the first bit field following the RECORD keyword is the most significant field of the record.

datum is an optional integer constant, or an expression containing no forward references which evaluates to an absolute number, specifying a default value for this bit field. This value can be overridden when the record is allocated. If no datum is present, zero is the default. If the datum is present, it must fit into the number of bits specified (nnn), zero-filled. For example, the legal default values for a 1-bit field are 0 and 1. Values that are either negative or too large are truncated to fit within a given field. A warning is also generated.

Description:

The RECORD directive always defines a record template of either 1 or 2 bytes in size. This definition only describes a record; it does not allocate any memory at definition time. The total number of bits in all bit fields within the record cannot exceed 16. When the template is used to define an occurrence of the record, memory is allocated in multiples of 8 bits (1 byte). If the total number of bits in a record template is one to eight inclusive, the unit used to allocate storage

when the record template is used is 1 byte. If the number of bits is 9 to 16 inclusive, then allocation is 2 bytes.

You might experience some confusion in those cases where the bit field allocation does not fill exactly 8 or 16 bits. While it is true that bit counting begins with the most significant bit in cases where the byte or word is completely filled, partially allocated records (the number of bits in the bit fields do not total exactly 8 or 16 bits) will have their bit fields right-justified in the byte or word and the remaining most significant bits will be zero-filled. This means that the first bit in the left-most bit field where counting begins will not be the left-most bit of the byte or word. The following definition

```
REC1 RECORD R1:3=7,R2:5    ;generates 11100000B or E0H
```

defines an 8-bit pattern which has all 8 bits filled. Note that R2, because it is not initialized, is set to zero by default. However, the definition

```
REC2 RECORD R3:3=7,R4:3=3  ;generates 00111011B or  3BH
```

leaves two bits remaining in an 8-bit byte. The two three-bit bit fields are right justified, and the remaining two bits, the two most significant bits, are zero-filled. The following figure illustrates how, for the above example of record template REC2, the partial record is defined by the RECORD directive.

Figure 4-1. "Partial" Record Definition

Similarly, the two 16-bit record definitions below illustrate what happens to 16-bit partial records.

```
REC3 RECORD R5:3=7,R6:13=4095
                    ;generates 1110111111111111B or  0EFFFH
REC4 RECORD R7:1=1,R8:8=127
                    ;generates 0000000101111111 or 017FH
```

Remember, the RECORD directive only defines a template, it does not allocate storage. To see how to allocate storage using a record template, read the next section.

Allocating Record Storage

After you have defined a record template, the template definition can be used in the following syntax to allocate storage:

Syntax:

```
[name] recname <[[datum],] [...]>
```

(or)

```
[name] recname repeatval DUP (<[[datum],] [...]>)
```

Where:

- **name** is an optional name to be declared as a variable with the following attributes:
  - **segment** - current segment being assembled
  - **offset** - current location counter value
  - **type** - total number of bytes in the record template (either 1 or 2)
- **recname** is the name assigned to a previously-defined record template **repeatval** is a 17-bit integer constant, or an expression containing no forward references and evaluating to a 17-bit absolute number, between 1 and 65535 inclusive. Repeatval specifies the number of copies of the record to allocate.

- **datum** is an optional value to be used instead of the default value provided in the template. All such values must be 17-bit integer constants, or expressions that evaluate to 17-bit absolute numbers. Relocatable values are not allowed.
  - The first datum replaces the default value of the first bit field within the record, the second datum replaces the default on the second bit field, etc. Null data items are permitted (separated by commas) to direct the assembler to use the default values; null data values are useful when a default value other than the first needs to be overridden. If a field is mmm bits wide, the least significant mmm bits of the twos complement representation of the datum are used. For example, if a 3-bit field is being overridden, values of 6, -2, and 14 will all generate the 3 bits 110. Examples (using the REC1 definition shown above):

```
FIRSTREC REC1 <>              ;no overrides to defaults,
                              ;generates 0E0H
SECNDREC REC1 <4>             ;overrides R1 - generates 080H
THIRDREC REC1 <,5>            ;overrides R2 - generates 0E5H
FIVERECS REC1 5 DUP (<>)      ;5 copies of default record
```

It is allowable to nest record allocations up to 10 deep.

SEGMENT/ENDS

The SEGMENT/ENDS directive pair is used to define a logical segment.

Syntax:

```
name SEGMENT [align-type][combine-type]['classname']
.
.
.
name ENDS
```

Where:

name is a mandatory user-defined name that cannot conflict with any other symbol.

align-type specifies what boundary the logical segment must be placed on. If the align-type is not specified, PARA is the default. Align-type may be any of the following keywords:

- **BYTE** - byte alignment. Segment can start anywhere.
- **WORD** - word alignment. The segment must start on an address divisible by 2. (An address which has a least significant bit of 0.)
- **PARA** - an address divisible by 16. (An address which has its least significant hexadecimal digit equal to 0H.)
- **PAGE** - page alignment. The segment must start on an address divisible by 256. (An address which has its two least significant hexadecimal digits equal to 00H.)
- **INPAGE** - inpage alignment. The entire logical segment cannot be more than 256 bytes long; it cannot cross a page boundary (an address divisible by 256). It will be moved to start on an address divisible by 256 only if movement is necessary to prevent the segment from crossing a page boundary.

SEGMENT/ENDS (Cont'd)

combine-type specifies the way in which the linking loader combines this segment with other logical segments of the same name to form a physical segment in memory. If combine-type is not specified, the logical segment will not be combined with any other logical segment, not even one with the same name from a different module. Combine-type can be any of the following keywords:

- **PUBLIC** - all segments of the same name defined to be PUBLIC will be concatenated to form a single physical segment. The loader controls the order of concatenation. The length of the resulting physical segment will be equal to the sum of the lengths of the segments that have been combined.

- **COMMON** - all segments of the same name defined to be COMMON will be overlapped, starting at the same physical address, to form a physical segment. The size of the resulting physical segment will be equal to the size of the largest segment of those overlapped.

- **STACK** - all segments of the same name defined to be STACK will be concatenated into a physical segment such that the combined segment will *end* at a certain physical address (overlaid against high memory) and will grow "downward." The length of the resulting segment will be the sum of the lengths of the combined segments. (STACK is not a true keyword. You can define a symbol to be STACK without conflicting with the usage in the SEGMENT directive.)

- **MEMORY** - all segments of the same name defined to be MEMORY will be combined so that the first memory segment encountered by the linker will be treated as the physical "memory" segment. In the list of linked modules, the first module that contains a "memory" segment will be used to define the physical "memory" segment. It will be located at an address above all other segments in the program. Any other

segments of the type memory that are encountered by the linker will be treated as common with the first segment. The length of the physical memory segment will be equal to the length of the first memory segment encountered (Memory, like Stack, is not a true keyword. You can define a symbol to be MEMORY without conflicting with the usage in the SEGMENT directive).

- **AT nnn** - this segment will be placed at the paragraph number specified. The expression nnn cannot contain forward references and must evaluate to an absolute number. Absolute segments are not aligned by the linker; the various align-type keywords are syntactically correct when used in combination with AT but are ignored. Note that **nnn** represents a paragraph number, not an actual address; therefore if AT 0444H is specified, the segment will start at address 04440H. A segment created with AT will be non-combinable with segments from other modules.

'classname' is a name that may be used to indicate that segments are to be located near each other in memory. When assigning physical addresses to these logical segments, the linking loader attempts to place logical segments with the same classnames close together. However, the classname cannot be used to combine segments such that they may be addressed through the same segment register.

The classname must be enclosed in single quotes, as shown, or in double quotes.

Classnames are not stored as symbols; they may duplicate symbol names (even keywords) without conflict. If a classname is to be assigned to a segment, assign it at the first occurrence of the segment in the source file.

Description: The SEGMENT/ENDS directive pair is used to define a logical segment. This segment may be combined with other segments of the same name defined in either the same module or in other modules.

SEGMENT/ENDS (Cont'd)

These logical segments will form the physical segments, located in memory, that are pointed to by the segment registers. Within a source module, each occurrence of an equivalent SEGMENT/ENDS pair (with the same name) is viewed as being one part of a single program segment.

Multiple Definitions of a Segment

The assembler keeps the value of the offset from the current segment (i.e., the most recent SEGMENT directive) in an internal location called the location counter. The assembler saves the location counter for each segment when it finds an ENDS for that segment, or if it finds a new SEGMENT directive. Later, if the assembler finds another SEGMENT directive which uses the name of that previously defined segment, the earlier location counter is retrieved and used. For this reason, a segment may be broken into pieces within a module, or between modules if it is combinable, and those pieces will still be placed in the same physical segment.

The align-type, combine-type and classname need not be included with the second and later SEGMENT directives for a segment of the same name. If they are absent, the assembler takes the segment's characteristics from the first definition. However, any keywords that are present must match the first definition, or an error is reported. If an absolute segment is broken into pieces and the AT keyword is used on a SEGMENT directive for the second or later piece, the absolute base address must match the first definition, even though the location counter is taken from the stored value. The second part of the segment will not start at the specified base address, but the AT value must match. Examples of breaking a segment:

SEGMENT/ENDS (Cont'd)

```
S1 SEGMENT PUBLIC
NOP                     ;relocatable location 0
S1 ENDS

S1 SEGMENT              ;assembler uses PUBLIC attribute
ADD AW,2                ;instruction at relocatable location 1
S1 ENDS


S2 SEGMENT AT 0444H
NOP                     ;instruction at absolute location 04440H
S2 ENDS

S2 SEGMENT AT 0444H
NOP                     ;instruction at absolute location 04441H
DB 14 dup(0)            ;skip 14 bytes
S2 ENDS
S2 SEGMENT AT 0445H     ;an error! Must use 0444H
NOP                     ;instruction at absolute location 04450H
S2 ENDS
```

Nested or Embedded Segments

It is legal to nest SEGMENT/ENDS pairs. Each ENDS must refer to the most recently-defined SEGMENT whose ENDS was not yet encountered. The fact that a segment is nested inside another does not mean that the code for the nested segment is placed inside the enclosing segment. The code is the same as it would be if no nesting occurred. Nesting helps you to define logical structures and makes programming easier. Example:

```
S1 SEGMENT PUBLIC
NOP                     ;goes into S1 segment
S2 SEGMENT PUBLIC
ADD AW,2                ;goes into S2 segment
S2 ENDS
SUB AW,3                ;goes into S1, S2 is "closed"
```

Improper Nesting:

```
S1 SEGMENT PUBLIC
NOP
S2 SEGMENT PUBLIC
ADD AW,2
S1 ENDS       ;ENDS does not match most recent SEGMENT
  SUB AW,3
S2 ENDS       ;ENDS does not match remaining SEGMENT
```

SEGMENT/ENDS (Cont'd)

Maximum Number of Segments

If you use the default HP-OMF 86 object file format, you may use an unlimited number of segments. The HP 64000 (.A) object file format allows only three named segments. Therefore, if you use the HP 64000 object file format (the **-h** command -line option), use three or fewer relocatable segments per module.

The first relocatable segment with code will be assigned the PROG segment. The first relocatable segment with data will be assigned the DATA segment, if that segment is not used for PROG. The next relocatable segment, whether it contains code or data, will be assigned the COMN segment.

SETIDB

The SETIDB directive indicates to the assembler and linker where in memory the V25 SFR and RAM registers are located.

Syntax:

```
SETIDB [ value ]
```

Where:

value is a constant numeric expression ranging from 0 to 0FFH. If the value is not present, the expression defaults to 0FFH.

Description:

The SETIDB directive is used to pass information to the ldv20 linker as to where in memory the V25 SFR and RAM registers are to be located. The value that is passed in to this directive indicates the upper 8 bits of the 20-bit physical address for these registers. If no value is given with the SETIDB directive, the default value of 0FFH is used. The V25 SFR and RAM registers starts at 0E00H bytes beyond the SETIDB address indicated. The physical address of any register in the V25 SFR and RAM registers can be obtained by shifting the SETIDB value to the left by 12, adding 0E00H to that value, and then adding the offset of the register to that value. This is, in fact the process used by the ldv20 linker. The linker will take the relocatable value for any references to V25 SFR and RAM registers and resolve them such that they refer to the register's physical location. This requires any ASGNSFR segment to be placed within 64k of the start of the V25 registers, so the resulting segment/offset pair can reach the correct physical address.

The SETIDB directive is optional for using the V25 SFR and RAM registers, but it must be used if these registers are to be placed at a location other than 0FFE00H in physical memory. The SETIDB directive generates object code to modify the V25 IDB register to contain the value used in this directive. The assembler also passes this value to the ldv20 linker so it can validate that the ASGNSFR segments are located at acceptable addresses. The object code generated by the SETIDB directive pushes some registers to the stack, so it should only be used after the SS and SP registers are initialized. It should also be used before any of the V25 SFR and RAM registers are accessed.

SETIDB (Cont'd)

The SETIDB directive can only be used once in any single module. Also, the ldv20 linker will check the modules that it is given so as to verify that only one of its input modules contains a SETIDB directive. Multiple SETIDB directives will result in a linktime error.

The SETIDB directive is only valid in the V25 mode. It is an error to use it in the V20 or V33 modes.

An example of using the SETIDB follows:

```
CODE SEGMENT PUBLIC
ASSUME PS:CODE

; Initialize the Stack registers
; so the stack can be used
MOV AW, STACKSEG
MOV SS,AW
MOV SP,0FFFFH

; Change the V25 IDB register to
; point to where I want the SFR
; and RAM registers to reside.
; In this example, the V25 registers
; will be at 080E00H through
; 080FFFH.
SETIDB 080H
:

CODE ENDS
```

STRUC/ENDS

The STRUC/ENDS directive pair is used to define a structure template.

Syntax:

```
name STRUC
.
.
<data directives>
.
.
name ENDS
```

Where:

name is a unique user-defined symbol that becomes the structure name. The name on the ENDS must match the name on the STRUC. Its type attribute is the following:

- **type** - number of bytes defined in structure data directives

Description:

The structure definition only describes a given structure and its contents; it does not allocate any memory at that time. All statements between the STRUC and ENDS directives must be one of the following: DB, DW, DD, DS, DQ, DL, or DT directives, comment lines, blank lines, or assembler controls. Any assembler controls that are included within the STRUC/ENDS pair are not stored as part of the template and therefore are not executed anew each time the structure is referenced. Any symbols referenced in the argument field of any of the included directives must have been previously defined. Forward references are not allowed within a structure definition.

You will notice that the ENDS directive is also used to terminate a SEGMENT definition. This is unambiguous, since an ENDS closing a SEGMENT is not legal within a structure definition.

If a DB or other directive within a structure definition has a name in its name field (which must be unique, and cannot previously have been the object of a forward reference), this name is known as a structure

STRUC/ENDS (Cont'd)

field. It is not the same as a variable, and it is not associated with any particular storage location or segment. Structure names and structure fields can be used in very few syntactic constructs. Forward references to structure names and structure fields are not allowed.

Structure field names do have associated attributes. They follow:

- **offset** - offset from the beginning of the structure definition
- **type** - type of data definition directive

Allocating Structure Storage

After you have defined a structure template, it can be used in the following syntax to allocate storage:

Syntax:

```
[name] strucname <[[datum],] [...]>
(or)
[name] strucname repeatval DUP ( <[[datum],] [...]> )
```

Where:

- **name** is an optional name to be declared as a variable with the following attributes:
  - **segment** - current segment being assembled
    offset - current location counter value
    type - total number of bytes in the structure template
- **strucname** is the name assigned to a previously defined structure template.
- **nnn** is a 17-bit integer constant, or an expression containing no forward references and evaluating to a 17-bit absolute number between 1 and 65535 (inclusive); it is the number of copies of the structure to allocate.

STRUC/ENDS (Cont'd)

- **datum** is an optional scalar to be used in place of the default value provided in the template. The first datum replaces the default value on the first data definition directive within the structure, the second datum replaces the default on the second data definition directive, etc.
  - Null data (separated by commas) is permitted and directs the assembler to use the default value; this is useful when a value other than the first occurring value must to be overridden. The legal values for these scalars are the same as in the data definition directive to which they apply, including the indeterminate-initialization keyword '?'. Note that repeated data (i.e., DUP expressions) cannot be used as an override.
  - Not every default value can be overridden. Default values can be replaced only if the template defined just one unit of data for the data definition directive (structure field) that is to be overridden, or the template defined a character string in a DB directive. These conditions mean that such defaults as DB 1,2 and DW 10 DUP (0) cannot be overridden.

The number of bytes used in a DB string is fixed when the structure is defined. Such a string can be overridden only by another string. If a longer string is used to override, it is truncated, and a warning message is given. If a shorter string is used to override, it is filled out, using the characters at the end of the default string.

Errors may occur upon allocation of a structure if the structure definition used DS or DL data directives, and the allocation of the structure occurs in either a BYTE or INPAGE aligned segment, or the location counter is not on a word boundary.

The figure on the following page illustrates structure definition and allocation.

STRUC/ENDS (Cont'd)

The structure definition

```
BLUEPRINT  STRUC
   FIRST      DW      0FFFEH
   SECOND  DW      BUFFER
   THIRD     DB      7, 5
   FOURTH  DB       'A'
   FIFTH      DB       ?
   SIXTH      DW      257
BLUEPRINT  ENDS
```

yields a structure template like this:

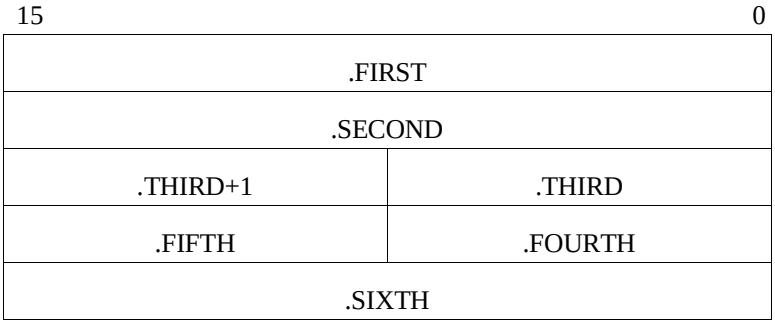


Figure 4-2. Structure Definition and Allocation

The instruction

B1 BLUEPRINT < >

allocates storage for B1 that looks like:

OFFSET (BUFFER)	
0 5	0 7
indeterminate	4 1
0 1 0 1	

The instruction

B2 BLUEPRINT <,0 . . . 255 >

allocates storage for B2 that looks like:

15 0

0 F F F E	
0 0 0 0	
0 5	0 7
F F	4 1
0 1 0 1	

Figure 4-2. Structure Definition and Allocation (Cont'd)

5

Expressions

Introduction

This chapter describes the syntax and semantics of expressions. The early part of the chapter explains the kinds of expressions and discusses expression operands. The latter part lists the different expression operators and their uses. The end of the chapter has a table showing the precedence ranking of the expression operators.

Reference Syntax Conventions

The sections that include the references about the expression operators follow certain conventions:

1. The name of the operator (or a descriptive term for the operator) appears in the lefthand column.

2. The proper assembler syntax appears next under a heading of "Syntax."

3. A short description follows the syntax. The description explains the syntax and any arguments appearing in the syntax. There may also be other information relating to the operator itself or to using the operator.

4. Some expression operators may affect the attributes (see chapter named "Attributes") of its operands. If that is so, a list of attributes and their values follows the description.

5. Some short examples that use the operator may follow the description or attributes sections.

Expression Overview

An expression is a simple or complex combination of operands that may be bound by operators. Operands can be numeric values or address expressions. Operators include conventional unary and binary arithmetic operators (+, -, *, /, MOD, etc.), logical operators (AND, OR, XOR, NOT), or special operators such as memory and record operators.

Expressions have certain attributes. Attributes are discussed thoroughly in the chapter named "Symbol and Expression Attributes."

Expressions are in turn used as operands to assembly language instructions and assembler directives. Expressions may be absolute, relocatable, or external.

Absolute Expression

An absolute expression is one whose value is known completely at assembly time. Assembly of absolute expressions results in object code that does not need to be further modified by the loader. An absolute expression will have an operand that is

- a numeric constant
- a constant memory expression (addresses which are known at assembly time)
- record allocation values
- a record bit field offset
- a segment base located during assembly time with the AT keyword (AT is discussed in the SEGMENT/ENDS directive in the "Assembler Directives" chapter)
- an offset for a variable or label from a segment which is non-combinable
- a register name

Relocatable Expression

A relocatable expression contains a relocatable operand as part of the expression. The value of a relocatable expression is not known at assembly time and must be assigned later by the loader. Relocatable expression values are 16-bit values unless modified by the HIGH or LOW operators to become 8-bit values. A relocatable expression will have an operand that is

- a segment base where the segment is combinable (including all groups, since their bases are not set until load time)
- a variable or label which belongs to a combinable segment

External Expression

An external expression is a relocatable expression which contains items that are not within the module being assembled. These expressions reference external variables, labels, or numbers. Their values must be assigned by the loader when the module containing the referenced item is available for relocating. External expressions, like relocatables, are assumed to be 16-bits in size, but may be modified with the HIGH or LOW operators to be 8-bit values. More information about external references appears in the chapter called "Assembler Directives."

During the assembly process, the assembler uses 17-bit numbers to perform arithmetic and other operations involving expressions. A 17-bit number is a 16-bit number with an additional sign bit. The 17-bit number is used within the assembler so that negative numbers with large absolute values (to -65535) may be used in calculations. When the value is coded, the sign bit is discarded and is not output, since only 16-bit values are used in the object code.

Expression Operands

An expression may consist of only an operand, or an operand or operands modified by an operator or operators. Operands are broadly divided into two groups: numeric values and memory or register expressions. A numeric value will be directly represented in the assembled code. A memory or register expression is an indirect value because the assembler is coding a reference —or reserving a space that will be filled later— which points to a location in memory where the actual data resides. Expressions involving the EQU directive can be either a numeric or memory expression.

Numeric Values

Numeric values result from a variety of different operands. Numeric constants, obviously, are numeric values, but other, less clearly numeric operands also produce numeric values. Any of the following operands can generate numeric values:

- **A constant.** There are several ways that an absolute number, or constant, may be represented to the asv20/asv33 assembler. The easiest and most straightforward way is to make the expression operand a decimal, octal, hexadecimal, or binary number. The various representations are as follows:
  - A decimal number is a series of digits, ranging from 0 to 9, that optionally ends with the character 'D'. Decimal numbers are base-10 and are the numbers people are most familiar with.
  - An octal number is a base-8 number represented by a series of digits, ranging from 0 to 7, and ending with either the character 'O' or 'Q'.
  - A hexadecimal number is a base-16 number represented by a series of digits, ranging from 0 to 9, or by characters, ranging from 'A' to 'F'. These numbers must end with the character 'H'. A hexadecimal number may not begin with a character; in those instances, place a leading zero in front of the hex number.
  - A binary number is a base-2 number represented by a series of digits, either 0 or 1, and ending with the character 'B'.

Examples of numeric constants:

```
MOV AW, 35        ;decimal number
MOV AW, 12D       ;decimal number with optional
                  ;following 'D'
MOV AW, 37O       ;octal number with the letter 'O'
MOV AW, 12Q       ;octal number with following 'Q'
MOV AW, 12H       ;hexadecimal number
MOV AW, 0A34H     ;hexadecimal number with leading 0
MOV AW, 0110101B  ;binary number
```

- **Quoted string.** A one or two character quoted string which is used as an expression operand will be stored as a hexadecimal number in a two byte word. Each byte contains the ASCII value of the character it stores. If two characters are stored in a word, the first character is represented in the high byte of the word and the second character is represented in the low byte. If only a single character is stored, it is represented in the low byte and the high byte is set to 00H. A quoted string always evaluates to a positive 17-bit value. This method of representing numbers is cumbersome and not very useful. It is also much more difficult to verify that the value is correct. Examples:

```
MOV AW, 'A#'      ;generates 04123H
MOV AL, HIGH 'B'  ;generates 00H
```

- **Record template.** The chapter named "Assembler Directives" discusses the record structure. A record is a series of bit fields which may be defined within a one or two byte structure called a template. Template definition does not allocate storage, but specifying an occurrence of a record can allocate memory, much like a DB (define byte) or a DW (define word) directive might allocate memory. A record template may also be used as an expression operand, but in this usage no memory is allocated. Instead, the operand is evaluated to be a positive 17-bit value and used the same as any number.

Examples:

```
R1 RECORD F1:3, F2:5, F3:2  ;the RECORD directive
                            ;defines record template
MOV AW, R1<>                ;value is 0 since
                            ;no defaults specified
                            ;in template definition
```

```
MOV AW, R1<2,14,3>          ;value is 0013BH
MOV AW, R1<2,14,3> + 5      ;value is 00140H
```

- **Record field.** You may also use a record field name by itself as an expression operand. If the field name is used without a MASK or WIDTH operator, then the assembler replaces the field name with a number which is the shift value required to move the lower bit of its bit field to the 0th bit position. For example, using the record template definition above, the value that would be replaced for F1 is 7 since there are 7 bits of data to the right of the field F1. The shift value, combined with the MASK operator described later in this chapter, may be used to extract field values from a record.

- **Segment or group name.** When used as an expression operand, the name becomes an immediate value that is the paragraph number for the segment or group. Since most segments and all groups are not assigned this value by the assembler, it will usually be relocatable. Only segments that use the AT keyword will have a fixed paragraph number known by the assembler. These values may be used as is —to initialize a segment register, for instance— or used wherever a relocatable number may be used (except with HIGH and LOW). Examples:

```
MOV AW, SEG1  ;load paragraph number for segment
MOV DS0, AW    ;initialize DS0 register
MOV AW, GRP1  ;load paragraph number for group
```

Memory and Register Expressions

There are several ways to reference memory in assembly source files. Memory might be referenced with operands which are any of the following:

- **Variables or labels.** Variables are defined through data directives and structure or record allocations. Labels are defined through assembly instructions or PROC directives. Either variables or labels may also be defined through EXTRN statements or LABEL directives. Given the variable and label definitions in the first three lines of the example below, the last two lines use those definitions as memory operands:

```
WMEM DW 2               ;word variable
R1 RECORD F1:3, F2:4    ;record template definition
U1 R1 <>                ;byte variable, from
                        ;a record allocation
L1: MOV AW, WMEM        ;NEAR label, using a word
                        ;variable

MOV AL, U1              ;uses byte variable as operand
BR L1                  ;uses NEAR label as operand
```

- **Variable with offset.** Variables used as memory operands may have offsets added to them in order to refer to memory locations near the memory location of the variable. The variable with offset operand may be expressed in two ways. Examples of both:

```
MOV AW, WMEM + 5         ;adds 5 to variable address
                         ;accesses memory 5 bytes higher
                         ;than location of variable WMEM
MOV AW, 5 + WMEM         ;same result from slightly different
                         ;way of expressing it
MOV AW, WMEM[5]          ;same result from very different
                         ;way of expressing it
```

- **Structure field.** Much the same as using an added offset to a variable, using a structure field name as part of a memory operand allows access to memory that is near a variable. Offset is from the variable named when storage using the structure template was allocated. Using a structure field name as a memory operand also changes the type of the memory expression to that of the field. Example:

```
ST1 STRUC
BFIELD DB ?       ;field offset value from ST1 is 0
WFIELD DW ?       ;field offset value from ST1 is 1
ST1 ENDS

MOV AW, BMEM.WFIELD  ;adds 1 to offset, word type
```

- **Register indirect reference.** The V20, V25, and V33 processors also allow an instruction to indirectly refer to memory by using base and/or index registers. The contents of these registers are added to a variable's offset at runtime, which means a memory address can be created that is not known when the assembly code is written. A register expression operand can contain one base register name, one index register name, or one base and one index register name. Additionally, constants may be part of the operand along with the registers.

  The valid base registers are BW and BP and the valid index registers are IX and IY.

  Base or index registers used this way must be enclosed in square brackets in a register expression, but there are several different ways to represent expressions given this restriction.

  – A base and index register may be added together explicitly by using a '+' sign within the brackets or added implicitly by enclosing each register name in separate, adjacent brackets.

  – A base or index register alone may have a constant added to it or subtracted from it in the same manner. (The '-' sign must be used for subtraction, since adjacent brackets are, by default, added.)

  – A base and index register added together may also have a constant added using either a '+' sign or adjacent brackets, or a constant may be subtracted by using a '-' sign within the brackets.

  – A base and index register cannot be subtracted from one another, however.

Examples:

```
MOV AW, WMEM[BW]           ;one base register,
                           ;no index register

MOV AW, WMEM[BP][IX]       ;these two slightly different
                           ;expressions are equivalent
MOV AW, WMEM[BP+IX]        ;both add one base register
                           ;and one index register

MOV AW, WMEM[IX]           ;no base register,
                           ;one index register

MOV AW, WMEM[5][BP]        ;both of these expressions use
MOV AW, WMEM[5+BP]         ;an index register with a
                           ;constant added


MOV AW, WMEM[BP-5]         ;one base register with
                           ;constant subtracted,
                           ;no index register
MOV AW, WMEM[BW][IY][5]    ;one base and one index
                           ;register added
MOV AW, WMEM[BW+IY+5]      ;with constant added also
                           ;both expressions equivalent
```

- **Anonymous reference.** This form of register expression operand contains only constants and registers and does not include a variable or label name. Because there is no variable or label name, no segment or type information is inherent in the expression.

  This expression may be given a type and segment, using the PTR and segment override operators. Otherwise, default values are assumed, depending upon the instruction and the registers that are used. If the base register BP is used, the default segment register is SS. Otherwise, the DS0 segment register is the default segment register.

  A default type value may be assumed if other operands to the instruction provide enough information to limit the type of the memory expression. Otherwise, an error is generated. For a constant to be used as a memory reference, it must be typed with the PTR operator so the assembler knows to treat the value as such. Otherwise, the constant is treated as an immediate value.

Examples:

```
MOV AW, [BW]              ;default is DS0 segment
MOV AW, [BP][IX]          ;default is SS segment
MOV AW, DS1:[BW]           ;segment is DS1
MOV AW, DS0: WORD PTR 5    ;segment is DS0
MOV AW, [BW].WFIELD       ;default is DS0 segment
```

EQU The EQU directive, discussed in the chapter titled "Assembler Directives," allows you to assign a value to a symbol. Some of the possible assignments include register names, variables, memory expressions, or constants. The symbol on the left side of the EQU directive may be used in an expression as an operand. The result is the same as if whatever appears on the right side of the EQU were used as an operand instead. Examples:

```
E1 EQU BW             ;V20 register
MOV AW, E1            ;register to register
MOV AW,BW             ;same as MOV AW, E1

E2 EQU WMEM           ;variable
E3 EQU BMEM[BP][IX]   ;register expression
E4 EQU 037B2H         ;constant
MOV AW, WMEM[E1]      ;register from memory
MOV AW, E2[E1]        ;register from memory
MOV AL, E3            ;register from memory
MOV AW, E4            ;immediate value into register
MOV AW, E4 / 5        ;immediate value into register
```

Expression Operators Introduction

Operators are functions that take one or more operands and return a new value. Operators are used to build expressions that cannot be defined strictly as simple operands. Use operators to add numbers, change the type of a memory expression, or to cause segment overrides. You may use a complex expression involving operators anywhere a simple operand may be used if the value returned by the complex expression is equivalent to the value of the simple operand.

Arithmetic Operators

The arithmetic operators conform to the commonly understood notions of these operators. Arithmetic involving these operators is done using the full 17-bit representation of the operands. Negative number results are stored, however, in twos complement form.

Unary Plus, Unary Minus

Syntax:

```
Unary Plus:   + operand
Unary Minus:  - operand
```

Description: The unary operators '+' and '-' each take a single operand and return a single value as the result. The '+' operator may be applied to an absolute or a relocatable value and the result will be an absolute or relocatable value. The '-' operator may only be applied to absolute values. The result will be the 2's complement of the value. These operators may be thought of as being the binary operators '+' and '-' with a lefthand operand of 0. Examples:

```
MOV AW, + 5     ;result is 5 or 00005H
MOV AW, - 2     ; result is -2 or 0FFFEH
MOV AW, + WMEM  ;result is memory expression
```

Binary Addition, Subtraction

Syntax:

```
Addition:     operand1 + operand2
Subtraction:  operand1 - operand2
```

Description: The binary operators '+' and '-' each take two operands and return a single value as the result. If memory addresses are used, the offset from the segment base is the value used as an operand. The types of operands that are allowed and the types of the results are shown in the following table.

The shorthand words in the table mean the following:

```
ABSNUM = absolute number, constant
RELOCNUM = relocatable number (OFFSET, external ABS, SEG)
ADDR = memory address, possibly relocatable or external
```

Table 5-1. Binary Plus and Minus Results

Operand 1	Operator	Operand 2	Result
ABSNUM	+, -	ABSNUM	ABSNUM
RELOCNUM	+, -	ABSNUM	RELOCNUM
ABSNUM	+	RELOCNUM	RELOCNUM
ADDR	+, -	ABSNUM	ADDR
ABSNUM	+	ADDR	ADDR
ADDR	-	ADDR	ABSNUM

Note that ADDR-ADDR is only valid if both memory addresses are either absolute or relocatable. They must also belong to the same segment so that their offsets are relative to the same base value. This allows the result to be absolute. Neither address may be of an external reference, since its offset is not known at assembly time. Examples:

```
EXTRN EXABS: ABS            ;declared labels -  variables
MEMSTART DB ?
WMEM DW 2
MEMEND DW ?

MOV AW, 5 + 15              ;result is 20 or 00014H
MOV AW, 3 - 12              ;result is -9 or 0FFF7H
MOV AW, WMEM + 5            ;result is offset of WMEM + 5
```

```
MOV AW, 4 + EXABS           ;result is external const + 4
MOV AW, MEMEND - MEMSTART   ;result is number of bytes
                            ;between MEMSTART and MEMEND
```

[ ] Square Brackets

Syntax:

```
address [ data_or_reg ]
```

Description: Square brackets give base and/or index attributes to an address expression or create a new address expression. The square brackets must occur in pairs. Such pairs cannot occur within angle brackets. However, more than one pair of square brackets can occur in a single expression.

The contents of the brackets are very limited. The only valid register names that can be used are BW, BP, IX, and IY. The first two, BW and BP, are base registers and only one of the two can be present within an entire expression. The IX and IY registers are index registers and, like base registers, only one of these registers can be present within an entire expression. It is valid to have both a base register and an index register in an expression. It is also possible to place numeric constants within the brackets.

The above items can appear singly within square brackets, as in:

```
mov AW, wmem[BW][IX][5]
```

It is also valid to replace '] [' pairs with a '+' sign, as in:

```
mov AW, wmem[BW+IX+5]
```

The only time a minus sign is valid within square brackets is to subtract a constant, as in:

```
mov AW, wmem[BW+IX-5]
```

The constant expression part of the square brackets modifies the offset value of any memory value that is also part of the expression. The base and index registers are used to denote indirect addressing as part of an expression. The contents of the indicated registers are added to any memory expression offset in the expression to create a final memory address.

A memory address is not required to be part of an expression which has square brackets as part of itself. For example, take the following expression:

```
mov AW, [BW][IX][5]
```

This expression represents a memory location that is 5 bytes past the sum of the contents of the BW and IX registers at the moment of execution for that instruction. The segment register used for this instruction would be the DS0 register. The SS register is used if the BP base register is part of the expression. It is also valid to specify a different segment register through the use of a segment override, such as:

```
mov AW, DS1: [BW][IX][5]
mov AW, SEG1: [BW][IX][5]
mov AW, GRP1: [BW][IX][5]
```

. (Dot operator)

Syntax:

```
address '.' struc_field
```

Description: This operand accepts an address expression as its left operand and a structure field as its right operand. The result of the operation is an address expression whose offset is equal to the offset attribute of the left operand plus the offset of the structure field within its structure template (in bytes). The type of the resulting memory expression is the type of the structure field. All other attributes are derived from the left operand. This operator is convenient for addressing fields within memory that contains one or more occurrences of a given structure. For example, suppose a structure was defined like this:

```
STRUCNAME STRUC
BYTEFLD DB 0
WORDFLD DW 5 DUP (3)
        DT 3.14159
STRINGFL DB 'DEFAULT'
STRUCNAME ENDS
```

The offset of BYTEFLD, WORDFLD, and STRINGFL within this structure template are 0,1, and 21, respectively. These structure field names can be used to reference fields within a structure in memory, as in:

```
DATABLOCK STRUCNAME<>
MOV AW, DATABLOCK.WORDFLD      ; WORD type
MOV CL, DATABLOCK.BYTEFLD      ; BYTE type

MOV IY, OFFSET DATABLOCK
MOV AW, [IY].WORDFLD           ; indirectly referencing memory
```

It is not valid to use the dot operator immediately after a digit, due to the possible confusion with a real number. Instead, the operator must be separated from the digit by parenthesis, such as:

```
(DATABLOCK + 2).WORDFLD        ; valid
DATABLOCK + 2.WORDFLD          ; illegal
```

Multiplication, Division, Modulo

Syntax:

```
Multiplication:  absval * absval
Division:        absval / absval
Modulo:          absval MOD absval
```

Description: These three operators each take two absolute values as operands and return a single absolute value. The '*' operator multiplies the two operands and returns the result. The '/' operator divides the first operand by the second operand. The MOD operator returns the value of the first operand modulo the second operand. Modulo division discards the integer quotient and returns a value that is only the remainder. For either straight division ('/') or modulo division, the righthand operand cannot have a value of 0 . Examples:

```
MOV AW, 5 * 3     ;result is 15 or 0000FH
MOV AW, (-2) * 5  ;result is -10 or 0FFF6H
MOV AW, 5 / 2     ;result is 2
MOV AW, 13 MOD 3  ;result is 1
```

SHL, SHR

Syntax:

```
absval SHL shiftvalue
absval SHR shiftvalue
```

Description: The SHL and SHR operators shift the first operand bitwise by the value of the second operand. Both operands must be absolute values; the result is also absolute. The SHL operator shifts bits to the left and SHR shifts bits to the right. Bits that are shifted to the left beyond the leftmost bit and bits that are shifted to the right beyond the rightmost bit are lost. Bits with a value of 0 are shifted in to fill. The sign bit of the 17-bit value can be modified as a result of a shift operation, since it is possible to shift 1's into or out of the sign bit. See the following figure for an example.

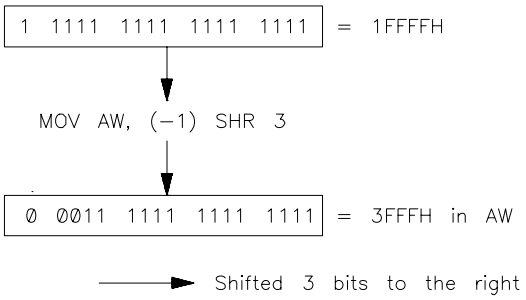


Figure 5-1. SHL Operator

Notice that the sign bit (the leftmost bit) of the argument in the above figure was shifted in when the shift right occurred.

Some other shifted values:

```
MOV AW, 5 SHL 2      ;result is 20 or 00014H
MOV AW, 13 SHR 2     ;result is 3
MOV AW, 44 SHL 11    ;result is 24576 or 06000H
MOV AW, (-54) SHR 3  ;result is 16377 or 3FF9H
```

HIGH, LOW

Syntax:

```
HIGH operand
LOW operand
```

Description: These operators take either an absolute value or relocatable memory expression as an argument and return a BYTE-sized value of the same type. HIGH returns the high byte of the operand, LOW returns the low byte.

If the operand is a memory expression, it cannot contain index or base register names. The value returned will be the HIGH or LOW byte of the memory location offset. Since this value is not always known during assembly, it may be relocatable and therefore set by the loader.

Attributes:

relocation type - high or low

Examples:

```
MOV AL, HIGH 01234H   ;result is 012H
MOV AL, LOW 01234H    ;result is 034H
MOV AH, HIGH WMEM     ;result is high byte of offset
MOV AL, LOW WMEM      ;result is low byte of offset

EXTRN EXTABS:ABS
MOV AL, HIGH EXTABS   ;result is high byte of
                      ;external number
```

The following identities apply to HIGH and LOW.

```
High (High X) = 0H
Low (Low X) = Low X
High (Low X) = 0H
Low (High X) = High X
```

Logical Operators

The logical operators return values that are the result of comparing operands. (NOT can be seen as an exception.) AND, OR, and XOR compare the bits of their operands while EQ, NE, ...,GE all compare the values of their operands.

AND, OR, XOR

Syntax:

```
absval AND absval
absval OR absval
absval XOR absval
```

Description: These operators each take two absolute values as operands and return a single absolute value. If n is used to identify any given bit of the result, bit n has its value set differently depending on the operator used. The following rules apply:

- The AND operator will set a bit n of the result to 1 if bit n of both operands is a 1; otherwise bit n is set to 0.
- The OR operator will set bit n of the result to 1 if bit n of either operand is a 1; otherwise bit n is set to 0.
- The XOR operator will set bit n of the result to 1 if bit n of each operand is different; bit n is set to 0 if both bits are the same.

The operations are performed on full 17-bit values. Examples:

```
MOV AW, 035H AND 3145H  ;result is 5
MOV AW, 035H OR 3145H   ;result is 3175H
MOV AW, 035H XOR 3145H  ;result is 3170H
```

NOT

Syntax:

```
NOT absval
```

Description: The NOT operator takes an absolute value as its operand and returns an absolute value that is the one's complement of the operand. The one's complement is derived by toggling the bits of the operand. If bit n of the operand is 1, then bit n of the result will be 0. Similarly, if bit n of the operand is 0, bit n of the result will be 1. The operation is performed on full 17-bit values. Examples:

```
MOV AW, NOT 1   ;result is 0FFFEH
MOV AW, NOT 55  ;result is 0FFC8H
```

EQ, NE, LT, LE, GT, GE

Syntax:

```
equal: operand1 EQ operand2
not equal: operand1 NE operand2
less than: operand1 LT operand2
less than or equal: operand1 LE
operand2
greater than: operand1 GT operand2
greater than or equal: operand1 GE
operand2
```

Description: These operators each compare their operands and return a value that depends upon the result of the comparison. The result will be 0 if the comparison is false and the value will be 0FFFFH if the comparison is true. The operands must both be absolute numbers, both be memory expressions, or both be segment base values. Memory expressions may not contain base or index register names, may not refer to externals, and must reside in the same segment. It is the offset portion of the memory addresses that are compared. Offsets and absolute values are compared using 17-bit arithmetic.

Examples:

```
MOV AW, 15 GT 3         ;result is 0FFFFH
MOV AW, WMEM EQ BMEM    ;result is 00000H
MOV AW, SEG WMEM EQ A   ;result depends upon whether
                        ;WMEM lies within segment A
```

Memory Operators

SHORT

Syntax:

```
SHORT label
```

Description: The SHORT operator takes a label as its operand. The SHORT operator assures the assembler that the label will be within 127 bytes of the current location counter. SHORT is mainly used with the BR instruction, where a forward reference to a label can result in either a one- byte or two-byte displacement. The SHORT operator informs the assembler that a one-byte displacement may be used (which only requires one byte of storage) where otherwise a two-byte displacement would result in extra object code size. It is up to you to ensure that the label is within 127 bytes because an error occurs if it is not. Example:

```
BR SHORT FWDLAB
```

THIS

Syntax:

```
THIS type
```

Description: The THIS operator takes a type name as an operator and returns a memory reference of the given type. The memory referenced will be for the current location and segment. The length of the memory will be 1. The valid types for the operand are BYTE, WORD, DWORD, FDWORD, QWORD, FQWORD, TBYTE, NEAR, and FAR. The result of this operator may be used as either the right-hand side of an EQU (in which case it acts the same as a LABEL directive) or as a memory reference in an instruction (which would be a rare use). Note that THIS NEAR is the same as '$'. (Dollar sign is the special character used to represent the location counter.) Also, use of FDWORD or FQWORD could result in errors if the current segment is BYTE or INPAGE aligned, if the current location counter is not on a word boundary, or if the assembler is not in the 72291 mode.

Attributes:

segment - current segment

offset - current location counter

type - as defined

relocation type - depends upon current segment

segment - current segment if defining variable

PS-assume - current PS assume value if defining label

Examples:

```
LAB2 EQU THIS FAR    ;create FAR label
LAB1: NOP
DATAW EQU THIS WORD  ;allow word accesses to bytes
DATABL DB 1
DATABH DB 2
```

PTR

Syntax:

```
type PTR operand
```

Description: The PTR operator is used to either set or change the type of its operand. The valid types that may be used are BYTE, WORD, DWORD, FDWORD, QWORD, FQWORD, TBYTE, NEAR, and FAR. The resulting expression will behave as a variable, label, memory expression, or register expression of the given type. Valid operands depend upon the type used. For instance, it is not possible to change the type of a register expression to a NEAR or FAR label.

Attributes:

type - as defined

Examples:

```
MOV AW, WORD PTR BMEM      ;access as word
BR NEAR PTR LABFAR        ;use far label as NEAR
MOV AL, BYTE PTR [BP]      ;typing an anonymous
                           ;memory reference
MOV DS0: WORD PTR 10, AW    ;absolute offset typing
```

Segment or Group Override

Syntax:

```
operand1 : operand2
```

Description: The segment override changes the segment attribute of the second operand to that of the first operand for the duration of the instruction statement. The first operand may be

- one of the segment registers (DS0, DS1, SS, or PS)
- the name of a segment
- the name of a group

The second operand must be a variable, label, memory expression, or register expression. If the first operand is a segment register, then the second operand's segment addressability attribute is changed to that of the segment register and no further testing is done. If the first operand is a segment name or group name, then the ASSUME values are checked to see if a segment register has been assumed to point to the segment or to the group. If one is found, the segment relocation and addressability attributes are changed to that of the matching segment register. If one is not found, it is an error. Remember, segment overrides only affect the current instruction; the ASSUME directive should be used for more global overrides.

The group override is useful when referring to variables or labels that belong to segments in the group. If no override is used, all offsets are relative to the base of the segment that the memory belongs to. The group override must be used to make the offset relative to the base of the group, which is probably a different value.

Attributes:

segment relocation - set to value of group or segment name used

segment addressability - set for variables

PS-assume - set for labels if group or segment name used

Examples:

```
MOV AW, DS0: WMEM      ;offset from DS0, base of segment
                       ;that WMEM belongs to
MOV AW, SEG1: WMEM     ;offset from base of SEG1, or group
                       ;that SEG1 belongs to, depending upon
                       ;order of ASSUMES
MOV AW, GRP1: WMEM     ;offset from base of GRP1
BR FARLAB             ;offset from base of segment
BR GRP1: FARLAB       ;offset from base of GRP1
```

OFFSET

Syntax:

```
OFFSET variable
OFFSET label
```

Description: The OFFSET operator takes a variable, label, or memory expression as its operand and returns the offset value from some base as the result. If no segment override appears in the operand, the offset will be from the beginning of the segment. If a group name is used as a segment override, then the offset will be from the group base. Remember that no checking is done against the ASSUME values for the registers. To get the offset from a group, an explicit group override must be used. In either case, the result is an immediate value, not a memory address. The value may be relocatable, depending upon whether the operand resides in a combinable segment or in a group. The result of an OFFSET operator occupies 2 bytes if it is a relocatable value. Otherwise, the number of bytes depends upon the value of the offset. Example:

```
MOV IX, OFFSET WMEM       ;offset from segment base
MOV IX, OFFSET GRP1:WMEM  ;offset from group base
```

SEG

Syntax:

```
SEG variable
SEG label
```

Description: The SEG operator takes a variable, label, or memory expression as its operand and returns a segment base as its result. The base may be relocatable, depending upon the type of the segment or group that the operand belongs to or on any overrides that have been applied to the operand. The memory expression may not contain index or base register names. Externals are allowed in the operand. The size of a relocatable segment base is always 2 bytes unless the segment definition used the AT keyword. In that instance, the number of bytes may be 1 or 2, depending upon the segment location.

The SEG operator should not be used with operands that belong to a group. Instead, a segment register should be initialized to the group base so that all memory addresses will be offset from that base. Otherwise, the group is not being used correctly.

Note that the SEG operator may also be used in the ASSUME directive. See the reference about the ASSUME directive in the chapter titled "Assembler Directives" for more discussion on how SEG may be used with ASSUME.

Note

The SEG operator will also accept a segment name or a group name as an operator. Since segment names and group names do not have segment attributes, SEG with a segment or group name does not perform any function. The assembler ignores the SEG operator and acts as if only the segment or group name were used.

Attributes:

relocation type - base

Example:

```
MOV AW, SEG WMEM; load base value into AW
MOV DS0, AW; initialize DS0 register
```

TYPE

Syntax:

```
TYPE variable
TYPE label
```

Description: The TYPE operator takes a variable, label, structure name, or memory expression as its operand. TYPE returns an absolute value that represents the type of the operand.

For most operands, the result is equal to the number of bytes allocated by a single occurrence of the operand. This value could then be used for incrementing a pointer into a data array, for example. The following are the returned values for variables or labels of a given type:

- BYTE - returns 1
- WORD - returns 2
- DWORD /FDWORD - returns 4
- QWORD /FQWORD - returns 8
- TBYTE - returns 10
- NEAR - returns -1 in two's complement form
- FAR - returns -2 in two's complement form
- record - returns number of bytes described by an occurrence of record
- structure - returns the sum of the sizes of the directives within the structure

Examples:

```
MOV AW, TYPE WMEM          ;result is 2
MOV AW, TYPE LABFAR        ;result is -2  in two's
                           ;complement form (FFFEH)
REC1 RECORD F1:3, F2:5     ;record definition with
                           ;RECORD directive
R1 REC1 <>                 ;storage allocation
                           ;using record template
MOV AW, TYPE REC1          ;result is 1
MOV AW, TYPE R1            ;result is 1



ST1 STRUC                  ;structure template
                           ;definition
  DB ?
  DW ?
ST1 ENDS
SU1 ST1 <>                 ;storage allocation using
                           ;structure template
MOV AW, TYPE ST1           ;result is 3
MOV AW, TYPE SU1           ;result is 3
```

LENGTH

Syntax:

```
LENGTH variable
```

Description: The LENGTH operator takes a variable as its operand. It returns an absolute value equal to the number of units that were defined with the variable. A unit may include several bytes allocated by a single occurrence of a type, but it still counts as just one unit. For instance, a single word allocation occupies two bytes, but from the point of view of LENGTH, it is one unit (in this case one word). The length of external symbols is always defined to be 1, regardless of how it is defined in a different file. LENGTH does not operate on structure or record templates. Examples:

```
L1 DB 1
MOV AW, LENGTH L1        ;result in AW is 1

 L2 DW 1,2
 MOV AW, LENGTH L2       ;result in AW is 2

L3 DB 5 DUP (2)
MOV AW, LENGTH L3        ;result in AW is 5

 L4 DW 1, 4 DUP (?)
MOV AW, LENGTH L4        ;result in AW is 5

REC1 RECORD F1:3, F2:5   ;record template definition
R1 REC1 <>               ;variable declared using record
                         ;template
MOV AW, LENGTH R1        ;result in AW is 1

 R2 REC1 5 DUP (<>)      ;another variable with record
                         ;template
MOV AW, LENGTH R2        ;result in AW is 5

 ST1 STRUC               ;structure template def.
  DB ?
  DW ?
ST1 ENDS
SU1 ST1 <>               ;variable declared
                         ;using structure template
MOV AW, LENGTH SU1       ;result in AW is 1
```

SIZE

Syntax:

```
SIZE variable
```

Description: The SIZE operator takes a variable, structure name, structure field, or record name as its operand and returns an absolute value equal to the total number of bytes defined by the operand. The size is generally equal to the length of the operand multiplied by the operand's type. Examples:

```
L1 DB 1
MOV AW, SIZE L1         ;result in AW is 1

L2 DW 1,2
MOV AW, SIZE L2         ;result in AW is 4

L3 DB 5 DUP (2)
MOV AW, SIZE L3         ;result in AW is 5

L4 DW 1, 4 DUP (?)
MOV AW, SIZE L4         ;result in AW is 10


REC1 RECORD F1:3, F2:5  ;record template definition
R1 REC1 <>              ;storage allocation using record
                        ; template
MOV AW, SIZE R1         ;result placed in AW is 1
MOV AW, SIZE REC1       ;result placed in AW is 1

ST1 STRUC               ;structure template def.
  DB ?
  STF1 DW ?
ST1 ENDS
SU1 ST1 <>              ;variable declared using
                        ;structure template
MOV AW, SIZE ST1        ;result placed in AW is 3
MOV AW, SIZE SU1        ;result placed in AW is 3
MOV AW, SIZE STF1       ;result in AW is 2
```

Record Operators

Record operators are used with record structure templates and record allocations to isolate bit fields of records and to find the actual number of bits in a record.

MASK

Syntax:

```
MASK recfield
```

Description: The MASK operator takes a record field as its operand. It returns an absolute number that will mask all the bits in a record except for those that belong to the record field operand. A mask is a number that will have 1's for all bits within the record field and have 0's for all other bits. It can be either a byte- or word-sized value, depending upon the size of the record and the positioning of the field within the record.

The MASK operator is useful when combined with the shift value (see Expression Operands in this chapter) for a record field. Together, they allow you to extract the value of a field. First, mask the record to isolate the bits that belong to the field. Then, shift the field so that its least significant bit is in the 0th bit position. The value of the result will now be equal to the value in the record field. Example:

```
R1 RECORD F1:5, F2:2
U1 R1 <14,3>
:
:
MOV AL,U1             ;load record into register
AND AL,MASK F1        ;mask out extra bits with MASK
                      ;operator and AND command
MOV CL, F1            ;put field shift value
                      ;in register
SHR AL,CL             ;shift field to lowest bit
                      ;position - AL now contains
                      ;value of record field
```

WIDTH

Syntax:

```
WIDTH operand
```

Description: The WIDTH operator takes a record name or record field as its operand. It returns an absolute number that is the number of bits defined in the operand. For a record name, the value will be the sum of the bits in the record fields, and will not include unused bits. For a record field, the value is the number of bits within that particular field. Examples:

```
R1 RECORD F1:5, F2:2
MOV AW, WIDTH R1      ;result in AW is 7
MOV AW, WIDTH F1      ;result in AW is 5
```

Segment and Group Operators

These operators return values that are only known at link-time. They generally refer to the size or address of segments and groups within a program.

SMOFFSET

Syntax:

```
SMOFFSET segmentname
```

Description: The SMOFFSET operator returns a value that is the offset of the indicated segment from the next-lowest paragraph boundary. This value is the same as the last hex-digit of the base address for the segment. If the segment is paragraph or page aligned or is at an absolute location, then this value will be 0. Otherwise, this value is a relocatable value that will be known at final link time. The value will be range from 0 to 15, but will be word-sized if it is relocatable. Example:

```
A SEGMENT BYTE

; LOAD PARAGRAPH VALUE FOR SEGMENT
MOV AW, A

; LOAD OFFSET OF SEGMENT FROM NEAREST
; PARAGRAPH.  TOGETHER, THEY FORM THE
; START LOCATION FOR THE SEGMENT
MOV BW, SMOFFSET A
```

GROFFSET

Syntax:

```
groupname GROFFSET segmentname
```

Description: The GROFFSET operator returns the offset of a segment's base from the start of a group that it belongs to. The segment must be defined as part of the group or this operator will result in an error. Since the offset within the group is not known until link time, this operator will result in a word-sized relocatable value. The linker will generate a value from 0 to 0FFFFH at link time, which will the offset of the segment's base from the start of the group. Example:

```
GRGRP GROUP A,B

; POINT DS0 AT GROUP
MOV AW, GRGRP
MOV DS0, AW

; SET UP POINTER TO START
; OF SEGMENT SO LOCATIONS
; WITHIN THE SEGMENT CAN BE
; REFERENCED FROM THE GROUP
; SELECTOR
MOV IX, GRGRP GROFFSET B
```

SMSIZE

Syntax:

```
SMSIZE segmentname
```

Description: The SMSIZE operator returns a word-sized value that is the size of the indicated segment. Since this size is not known (usually) at assembly time, this operator generates a word-sized relocatable value. The linker will generate a value from 0 to 0FFFFH at link time. Note that the linker will return the value 0 if the group size is 64k.

Example:

```
A SEGMENT PUBLIC

; LOAD SEGMENT SIZE.
; COULD BE USED TO MAKE
; SURE INDEX VALUES DON'T
; GO OUTSIDE OF A SEGMENT.
MOV AW, SMSIZE A
```

GRSIZE

Syntax:

```
GRSIZE groupname
```

Description: The GRSIZE operator returns a word-sized value that is the size of the indicated group. Since this size is not known at assembly time, this operator generates a word-sized relocatable value. The linker will generate a value from 0 to 0FFFFH at link time, which will be the size of the group. Note that the linker will return the value 0 if the group size is 64k. Examples:

```
GRGRP GROUP A,B

MOV AW, GRSIZE GRGRP
```

Operator Precedence

Complex expressions, or expressions that contain multiple operators, are evaluated according to operator precedence rules:

- Expressions enclosed within parentheses are evaluated from the innermost set of parenthesis to the outermost set. Within a set of parenthesis, operators conform to the other precedence rules below.
- Excluding parentheses, sub-expressions that have operators of higher precedence will be calculated before sub-expressions with operators of lower precedence. For example, a multiply operation is done before an addition operation.
- Excluding parentheses, sub-expressions which have operators of equal precedence (Operators that appear on the same line in the table below are of equal precedence.) are evaluated left-to-right. Left-to-right evaluation means that if two operators of equal precedence appear in the same expression, the operator which is closer to the leftmost end of the expression will be evaluated before an operator closer to the rightmost end. For instance, in the expression '6 * 5 / 3' the order of evaluation is to multiply 6 by 5 and then divide by 3. The result is 10.

The ranking of operators from higher to lower precedence is given in the following table.

Table 5-2. Operator Precedence

Precedence	Operators
Higher	(), [ ], <>, . ,LENGTH, SIZE, WIDTH, MASK , SMOFFSET, SMSIZE, GROFFSET, GRSIZE
	PTR, OFFSET, SEG, TYPE, THIS, Segment Override
	HIGH, LOW
↑	*, /, MOD, SHR, SHL
	Unary +, -
	Binary +, -
	EQ, NE, LT, LE, GT, GE
↓	NOT
	AND
	OR, XOR
Lower	SHORT

Notes

6

Instructions and Operands

Introduction

This chapter, in the early part, thoroughly discusses the operand field of the general assembly language statement syntax found in the "Assembler Statements" section of the chapter titled "Assembler Syntax." (No need to refer back to it. The syntax has been repeated at the beginning of the next section.)The latter part of this chapter contains a listing that specifies the recognized instructions for the asv20/asv33 assembler and also the acceptable operand combinations for each instruction.

Operand

You may recall that the general syntax of an assembler statement is as follows:

```
[ label : ] [ prefix ] keyword [ operand [ ,...] ] [;comment]
```

This section concentrates on the operand field of this syntax.

Accepted Operands

A list of assembly language instructions and the operand combinations acceptable for each instruction is at the end of this chapter. Each allowable combination has a limited range of values. Any other combination results in an error condition.

Compatible Types

In most instances, if an instruction takes more than one operand, the operands must be of the same type. For example, it is only possible to move a WORD-sized value into a WORD destination. A mismatch error occurs if an instruction attempts to move a WORD into a BYTE.

It is possible, however, to move a BYTE-sized immediate value into a WORD-sized destination. The immediate is either stored as a WORD or it is sign-extended during execution.

Some instructions allow operands to be of different types. It is best to check the list of instructions at the end of the chapter for allowable operand combinations.

Required Typing

Many instructions do require that the memory operand be typed. Instructions that take a single operand generate different object code depending upon the type of the operand. Or, perhaps the type of one operand does not restrict the valid type of the other operand. The assembler cannot decide what object code to output in these instances. The following instructions demonstrate some unacceptable operand combinations:

```
INC [BW]     ;generate byte or word instruction?
ESC 5,[BW]   ;5 doesn't restrict memory
MOV [BW], 2  ;2 fits in a byte or word storage
```

The INC instruction accepts both BYTE and WORD memory operands. In the above example, the assembler could not decide which instruction to generate.

The ESC instruction also accepts BYTE and WORD memory operands. The immediate value 5, in the example above, does not help limit the type of the memory operand since the value is independent of the memory type.

For the MOV instruction above, the immediate value 2 is small enough to fit in either a BYTE or a WORD. Again, the immediate operand does not restrict the type sufficiently.

When in doubt, type these ambiguous expressions to avoid possible error conditions.

Anonymous References

Most instructions are able to accept operands that do not have type information—references known as anonymous memory references. These references do not have a variable or any type information

associated with them, so the assembler must use other knowledge to guess the type. The assembler may type the anonymous operand to be the same as another operand in the instruction, or not require a type at all. The following examples are of typing the same as another operand:

```
MOV AW, [BW]   ;WORD since AW is a WORD-sized register
MOV [BW], AL   ;BYTE since AL is a BYTE-sized register
MOV [BW], 1000 ;WORD since 1000 can't be stored in BYTE
```

Assumed Type With Register

The assembler can easily guess the type of an anonymous reference if the other operand is an V20 register. Notice in the above example when AW and AL were used. Another example of an instruction not needing a type (since it handles all memory operands the same) is an 8087 floating point instruction. Example:

```
FLDCW [BW]
```

Operand Positioning

If an instruction takes a single operand, the operand position (other than it must be in the proper place) is not critical. Instructions which accept two operands generally treat the first operand as the destination operand and the second operand as the source operand. The movement of data is then from the second operand into the first. The instruction

```
MOV AW, BW
```

takes the contents of the BW register and places it in the AW register. There are exceptions. Some string instructions use the first operand as the source operand and the second operand as the destination operand. Check the usage of the operands when in doubt. The instruction list at the end of this chapter —and in the *NEC 70116 User's Manual*— includes information on data movement between operands.

Immediate Values

Immediate values are operands in many assembly language instructions. In most cases, the immediate value is a source operand. This value is stored directly in the destination operand or used to modify a value already stored elsewhere, say in a register or memory location.

Immediate values are not always numbers. Immediate values are also generated in many non-obvious ways as shown in the chapter titled "Expressions."

Range of Immediate Values

Immediate values can be absolute, relocatable, or external numbers. The size of the value is determined by the instruction used, by the value itself, and by what type is assumed for it.

An absolute immediate may range anywhere from -65535 to 65535 depending upon the instruction and the type of the operand. The BRK (interrupt) instruction, for instance, can only take a value from 0 to 255 since that is the range of interrupt values for the V20. A variable of type BYTE may take a value from -255 to 255. A variable of type WORD may take a value from -65535 to 65535.

A relocatable or external immediate is always assumed to be a 16-bit value unless modified with a HIGH or LOW operator.

Registers

A very common operand is a processor register. A processor register is a memory store that is internal to the V20, V25, and V33 processors, and the 8087 or 72291 co-processors. Internal registers can be source operands or destination operands for data. Some registers have special tasks which restrict their uses in programs. Since some instructions may indirectly use or modify these restricted registers, take care their contents are not accidentally modified or misused.

The figure below shows the general purpose and special registers for the V-Series processor. Following the figure is a more detailed description of the various processor registers.

DATA REGISTERS

7 0	7 0
AH (HIGH BYTE OF AW)	AL (LOW BYTE OF AW)
BH (HIGH BYTE OF BW)	BL (LOW BYTE OF BW)
CH (HIGH BYTE OF CW)	CL (LOW BYTE OF CW)
DH (HIGH BYTE OF DW)	DL (LOW BYTE OF DW)

POINTER AND INDEX REGISTERS

15 0
SP (STACK POINTER)
BP (BASE POINTER)
IX (SOURCE INDEX)
IY (DESTINATION INDEX)

SEGMENT REGISTERS

PS (CODE)
DS0 (DATA)

Figure 6-1. V20/25/33 Registers

16-bit Registers AW, BW, CW, DW, IY, IX, SP, BP

There are eight 16-bit (WORD-sized) general purpose registers located on the V20, V25, and V33 processors referenced by the unique register names AW, BW, CW, DW, IY, IX, BP and SP. AW, BW, CW, and DW are general purpose data registers. For most instructions that allow a register as an operand, these four registers are used. IY, IX, BW and BP are the index and base registers.

Some instructions explicitly use certain registers. The CW register, for instance, is used to control looping. Many string instructions use the IX as a source pointer and IY as a destination pointer. The SP register points to the top of stack and is modified whenever CALLs, PUSHs, or POPs occur. Data loss can occur through a side effect of these explicit usages. Be careful to protect the contents of these registers so they are not accidentally modified through the use of an instruction.

8-bit Registers AL, AH, BL, BH, CL, CH, DL, DH

There are also eight 8-bit (BYTE-sized) registers. The unique names given to them are AL, AH, BL, BH, CL, CH, DL, and DH. These registers are not separate registers; instead they are the byte-addressable upper and lower halves of the four 16-bit general-purpose data registers (AW, BW, CW, and DW). AW, for instance, is equivalent to AL+AH. (Not the value, but the register.)

The 'L' in AL means the low byte of AW and the 'H' in AH means the high byte AW. If you refer to AL, the assembler understands that you mean the low byte of AW. If you refer to AW, the assembler understands that you mean the entire 16 bits of AW.

You may load data into these registers either as a single 16-bit quantity or as two 8-bit quantities. The resulting value in the register is the same.

Segment Registers PS, DS0, SS, DS1

V-Series memory addresses are generated by offsetting from segment registers. To be able to address a particular location in memory, that address must be contained in one of the four, currently active physical segments. Each segment has a maximum size of 64K and each has a particular register that contains the base address (lowest memory location) of the segment. Each segment has a different purpose:

- Executable code (program code) is located in the Code segment and is addressable through the PS (Code Segment) register.

- Data is most often located in the Data segment (although it can be in any of the four segments) and is addressed through the DS0 (Data Segment) register.

- The program stack is located in the Stack segment and is addressed through the SS (Stack Segment) register.

- Data often is located in the Extra segment and is addressed via the DS1 (Extra Segment) register.

Memory Addressing A memory address is a 20-bit value —allowing the V20, V25, and V33 to address 1 megabyte of memory— that is calculated from the segment base address located in one of the segment registers, and an offset supplied either by the PC (instruction pointer), or by operands contained in the instruction itself. To calculate the memory address, the 16-bit value in a given segment register is first shifted to the left 4 bits. Then the offset value (either a 16-bit or 8-bit value) is added to the shifted value to generate the 20-bit address necessary to access memory.

Segment Register Use The four segment registers have restricted use. The only assembly instructions that may reference these registers as operands are the MOV, PUSH, and POP instructions.

Some Assembler Directives also use the register names as part of their syntax, but this use does not cause object code to be generated.

Other instructions indirectly reference the segment registers. CALLs and BRs, for example, change the PS register if the branch takes execution out of the current segment. Finally, as noted in the chapter titled "Expressions," segment register names may be used as overrides in memory operands.

8087 Floating Point Registers ST(0)...ST(7)

The 8087 co-processor has eight floating point stack registers. They are referenced by the names ST(0), ST(1), ST(2), ST(3), ST(4), ST(5),

ST(6), and ST(7). ST(0) may be referenced as just ST without the appended (0). These registers are only used with some 8087 floating point instructions.

They are not directly accessible to the V20, V25, and V33 processors. Instead, 8087 instructions make the contents of these registers available in memory. The 8087 floating point stack registers are 80 bits in size and store their values in IEEE floating point format.

72291 Floating point Registers FR0,..,FR7,FS0,..,FS7,FL0,..,FL7,FCTW,FPTW,FSTW

The 72291 co-processor has 8 stack registers and 3 status registers. The 8 stack registers, labeled FR0 through FR7, are each 96 bits in size when stored. The contents of each stack register can be either a short, 32-bit floating point value (FS0 through FS7) or a long, 64-bit floating point value (FL0 through FL7). It is invalid to mix types when performing floating point operations. The three status registers (FCTW, FPTW, and FSTW) store information about the current state of the 72291 co-processor.

These registers are not directly accessible to the V33 processor. Instead, 72291 instructions must be executed to make the contents of these registers available in memory.

Memory Expressions and the MODRM Byte

Memory expressions may be either simple memory references (using a variable name by itself) or a complex expressions involving register indirection or offsets within structures. A simple memory reference will always take the type of the variable, so that type must either be compatible with an instruction or it must be re-typed with the PTR operator. Examples:

```
MOV AW, WMEM                ;simple variable
MOV AW, [BW][IX]            ;indirect anonymous
                            ;memory reference
MOV AW, [BP].SFWORD         ;indirect anonymous memory
                            ;reference with offset
MOV AW, WMEM[BP][IY]        ;indirect memory reference
MOV AW, STR1.SFWORD         ;structure field reference
MOV AW, WORD PTR DMEM       ;typed memory reference
```

Physical Address Calculation

The processor must generate a physical address for each memory reference. The offset part of the address —the value which is added to

the shifted segment register address— may be coded into the instruction in one of four ways:

- As a direct 16-bit offset.
- As an indirect offset through a base register, BW or BP, optionally with an added (or subtracted) 8-bit or 16-bit displacement.
- As an indirect offset through an index register, IX or IY, optionally with an added (or subtracted) 8-bit or 16-bit displacement.
- As an indirect offset through the sum of one base register and one index register, optionally with an added (or subtracted) 8-bit or 16-bit displacement.

MODRM Byte

The information describing how the offset is derived is stored in the object code in a special byte called the MODRM byte. This byte has three fields:

1. The first field describes how many bytes are required to hold the displacement portion of the address. This field can specify that 0, 1, or 2 bytes are required. If the value is a relocatable or external value, two bytes are always required.
2. The second field contains a register code or part of the code for the instruction; it is not relevant to this section.
3. The third field contains information describing what base and index registers are used, if any, when generating the address.

The MODRM byte, along with any displacement value, determines the offset of the memory address referenced in an instruction. Remember, the value is just the offset of the memory address. The base from which to offset must still be decided.

Single Memory Expression per Instruction

Each memory expression is either a source or destination for the instruction. Most instructions allow only a single memory expression, since the MODRM byte can only describe one. Some string instructions may have two memory expressions as operands, but these instructions are special cases because the operands are only used to check for segment addressability. Their offsets are not emitted as object code. Instead, the IX and IY registers are used for addressing the memory.

Segment Addressability and Overrides

The V20, V25, or V33 processor generates a memory address by shifting the value from a segment register four bits to the left and then adding an offset to the shifted value. A segment of memory, up to a maximum of 64K bytes in size, is active only if one of the four segment registers points to that particular piece of memory.

Note that the segment is a physical segment, a physical piece of memory. These physical segments contain the logical segments of your assembly language program that you identified through SEGMENT/ENDS assembler directive pairs and other, similar means.

With the ASSUME assembler directive, you tell the assembler what values to assume as the base locations of the currently active segments. The ASSUME directive, then, lets you inform the assembler of the relationship between the logical segments you have defined in the program and the physical segments where they will eventually be located.

Addressability Checking

During assembly, if the assembler encounters an instruction that generates a memory reference, the assembler checks that reference against the value in the ASSUME for that segment. The assembler generates an error if the location in memory cannot be accessed through that particular segment register. The exception to checking against the ASSUME is when a memory reference contains a specific segment override.

NEAR and SHORT label references are also checked for addressability through the PS segment register to assure the assembler that the label

can be reached during execution. A segment or group name may be used to override a label if the PS segment register value will be different than that currently assumed.

Addressability checking is done so that the correct object code may be generated. Unless a memory reference contains a segment override, the instruction is not preceded by a segment override byte in the generated object code. If no segment override byte is coded with the instruction, then the instruction memory reference defaults to a certain segment, depending upon the nature of the instruction.

Default Segments

If a memory reference does not specifically name a segment register through a segment override, there are default segment registers for memory references. The PS register is the default for instruction fetching. The DS0 segment register is the default for most memory data references, unless BP (a base register) is specified for register indirection. The SS segment register is the default if BP is used. Some string instructions default to the DS1 segment register with certain operands.

Although there are default segment registers for references, you must still use the ASSUME directive to inform the assembler where the bases of these segments are located; again, to specify the relationship between logical and physical segments and to aid in addressability checking.

Segment Overrides

An instruction may override these default registers by including a segment override in the instruction operand. There are two reasons why a segment override might be included in a memory reference:

- The memory location accessed in not located in the default segment that would be used with a particular instruction.
- The memory location accessed is located within a group in a segment. In this instance, the base of the group must be used for memory access, not the base of the segment.

The override holds for the duration of the instruction only. Segment overrides do not alter the contents of segment registers or the values specified in ASSUME directives.

Improper Uses of Segment Overrides

The section on default segments mentions that some string instructions default to the DS1 register. For these string instructions, you may not use segment overrides for string operands. You may use segment overrides, however, for the other memory operands in those instructions.

These and other exceptions are noted in the listing of instructions at the end of this chapter.

Segment Override Byte

When the assembler generates code for an instruction containing a segment override, the assembler precedes the instruction code with a segment override byte. (Whether it will appear or not is discussed below.) This override byte, if present, causes a specific segment register to be used to address that memory, regardless of which segment the variable belongs to. In the segment override byte, specific values are associated with specific registers. Examination of these values can tell you which segment the override has been generated for. The values are

```
PS - 2EH
DS0 - 3EH
SS - 36H
DS1 - 26H
```

Overrides and Checking Against ASSUME

If a segment name is used to override the default segment value for a memory reference, then the ASSUME value for the override segment is checked to see if it has been set to either

- the segment named in the override, or
- to a group that contains the segment named in the override.

If a group name is used, then the group name must match exactly.

Examples of segment overrides:

```
MOV AW, SEG1: WMEM       ;matches segment or group
MOV AW, GRP1: WMEM       ;matches group only
```

Segment Override Byte Generation

A memory reference that includes a segment override generates a segment override byte depending upon the outcome of the following checks:

1. If the memory is addressable by the default segment register for that type of instruction and operand, then the instruction needs no override byte.

2. If this test fails, then the segment registers are checked in the following order: DS0, DS1, PS, and SS. If the memory expression is addressable by one of these registers, then an override byte is generated for that register.

3. If no register match occurs, an error is generated. The checks are specific. If the variable used in the memory expression was an external defined outside of a segment, it can only match an ASSUME segment that has been set to the SEG value of the external or to a group that includes that segment.

The V25 Family of Processors

The V25 Family of processors include the V25, V35, V25+, and V35+ microprocessors. These processors accept the same instruction set as the V20 Family, but they contain some additional instructions and peripherals. The biggest difference between the two families is that the V25 processors have a 512-byte block of memory mapped into register memory onboard the processors. This block of register memory defaults to starting at 0FFE00H in memory, but can be placed anywhere in memory by modifying the IDB register at the top of the register block.

The 512-byte block is broken up into 2 256-byte pieces. The first block contains 8 sets of register banks, numbered from 0 to 7. Each register bank contains a full complement of the registers needed by the V25 for normal operation. The existence of these register banks allows the V25 to perform context switching with very little overhead. This context switching is normally used during interrupts since the switching of register banks means that the registers do not need to be saved on stack during the processing of the interrupt. A new feature of the V25 is the task switch. There are new instructions that allow a task to switch register banks and begin processing at a new location in memory. Again, the advantage of the register bank switch is that registers do not need to be saved to stack and processing can be resumed at the previous location by simply returning from the context switch to the original register bank. While the various registers in the current register bank can be accessed through the normal V20 instructions, the contents of other register banks can be accessed through the use of memory addressing. There are 2 predefined variables and 23 predefined structure field names that allow accessing of these register values. The RAM and REGBANK variables refer to the beginning of the register bank. These variables can be further modifying by using the correct series of structure field names to reference a specific register. For example, to get the contents of the IX register in the 6th register bank, the following instruction could be used:

```
MOV AW, RAM.BK6.BIX
```

The structure field names are of type 'word', so the typing of the instruction operands is correct. Table 6-1 contains a list of the predefined structure field names and the offset values associated with these names.

The second piece of register memory contains the registers that control the many peripherals that exist on the V25 processors. These registers,

Table 6-1. RAM Register Bank Structure Definitions

Name	Type	Length	Offset	Meaning
bk0	word	16	0	Register Bank 0
bk1	word	16	32	Register Bank 1
bk2	word	16	64	Register Bank 2
bk3	word	16	96	Register Bank 3
bk4	word	16	128	Register Bank 4
bk5	word	16	160	Register Bank 5
bk6	word	16	192	Register Bank 6
bk7	word	16	224	Register Bank 7
bvpc	word	1	2	VPC register
bpsw	word	1	4	PSW register
bpc	word	1	6	PC register
bds0	word	1	8	DS0 register
bss	word	1	10	SS register
bps	word	1	12	PS register
bds1	word	1	14	DS1 register
biy	word	1	16	IY register
bix	word	1	18	IX register
bbp	word	1	20	BP register
bsp	word	1	22	SP register
bbw	word	1	24	BW register
bdw	word	1	26	DW register

called the Special Function Registers or SFRs, can be written to or read from the same as if they were memory locations. The V25 BTCLR instruction may also be used to access these memory locations directly. There are predefined mnemonics for each Special Function Register, so the user can reference the SFR directly instead of using a numeric value to reference a specific register. For example, to place a value in the Timer Interrupt Request Control register, the following instruction could be used:

```
MOV TMIC0, 03H
```

Table 6-2 contains a list of the predefined SFR register names and the offsets associated with these names.

The mnemonics for the RAM and SFR registers are valid only in the V25 mode of the assembler and preclude the use of these names as user-defined variables or codemacros in the assembly code. There are also some other conditions upon their use, which are described in the ASGNSFR and SETIDB sections of the "Assembler Directives" chapter.

More information on the V25 processor can be obtained from the NEC 70320 User's Manual.

Table 6-2. RAM and Special Function Register Mapping

SFR Name		Type	Offset Meaning
RAM	byte	0E00	Register Bank
RAMBANK	word	0E00	Register Bank
P0	byte	0F00	Port 0
PM0	byte	0F01	Port Mode 0
PMC0	byte	0F02	Port Mode Control 0
P1	byte	0F08	Port 1

Table 6-2. RAM and Special Func. Reg. Mapping (Cont'd)

SFR Name		Type	Offset Meaning
PM1	byte	0F09	Port Mode 1
PMC1	byte	0F0A	Port Mode Control 1
P2	byte	0F10	Port 2
PM2	byte	0F11	Port Mode 2
PMC2	byte	0F12	Port Mode Control 2
PT	byte	0F38	Port T
PMT	byte	0F3B	Port Mode T
INTM	byte	0F40	Interrupt Mode
EMS0	byte	0F44	External interrupt Macro Service 0
EMS1	byte	0F45	External interrupt Macro Service 1
EMS2	byte	0F46	External interrupt Macro Service 2
EXIC0	byte	0F4C	EXternal I/O request Control 0
EXIC1	byte	0F4D	EXternal I/O request Control 1
EXIC2	byte	0F4E	EXternal I/O request Control 2
RXB0	byte	0F60	Receive Buffer 0
TXB0	byte	0F62	Transmit Buffer 0
SRMS0	byte	0F65	Serial Receive Macro Service 0
STMS0	byte	0F66	Serial Transmit Macro Service 0
SCM0	byte	0F68	Serial Communication Mode 0
SCC0	byte	0F69	Serial Communication Control 0

Table 6-2. RAM and Special Func. Reg. Mapping (Cont'd)

SFR Name		Type	Offset Meaning
BRG0	byte	0F6A	Baud Rate Generator reg 0
SCE0/SCS0	byte	0F6B	Serial Communication Error/Status 0
SEIC0	byte	0F6C	Serial Error I/O request Control 0
SRIC0	byte	0F6D	Serial Receive I/O request Control 0
STIC0	byte	0F6E	Serial Transmit I/O request Control 0
RXB1	byte	0F70	Receive Buffer 1
TXB1	byte	0F72	Transmit Buffer 1
SRMS1	byte	0F75	Serial Receive Macro Service 1
STMS1	byte	0F76	Serial Transmit Macro Service 1
SCM1	byte	0F78	Serial Communication Mode 1
SCC1	byte	0F79	Serial Communication Control 1
BRG1	byte	0F7A	Baud Rate Generator reg 1
SCE1/SCS1	byte	0F7B	Serial Communication Error/Status 1
SEIC1	byte	0F7C	Serial Error I/O request Control 1
SRIC1	byte	0F7D	Serial Receive I/O request Control 1
STIC1	byte	0F7E	Serial Transmit I/O request Control 1
TM0	word	0F80	Timer Register 0
TM0L	byte	0F80	Timer Register 0 Low
TM0H	byte	0F81	Timer Register 0 High
MD0	word	0F82	Modulo register 0

Table 6-2. RAM and Special Func. Reg. Mapping (Cont'd)

SFR Name		Type	Offset Meaning
MD0L	byte	0F82	Modulo register 0 Low
MD0H	byte	0F83	Modulo register 0 High
TM1	word	0F88	Timer Register 1
TM1L	byte	0F88	Timer Register 1 Low
TM1H	byte	0F89	Timer Register 1 High
MD1	word	0F8A	Modulo register 1
MD1L	byte	0F8A	Modulo register 1 Low
MD1H	byte	0F8B	Modulo register 1 High
TMC0	byte	0F90	Timer Control 0
TMC1	byte	0F91	Timer Control 1
TMMS0	byte	0F94	Timer Macro Service 0
TMMS1	byte	0F95	Timer Macro Service 1
TMMS2	byte	0F96	Timer Macro Service 2
TMIC0	byte	0F9C	Timer I/O request Control 0
TMIC1	byte	0F9D	Timer I/O request Control 1
TMIC2	byte	0F9E	Timer I/O request Control 2
DMAC0	byte	0FA0	DMA Control 0
DMAM0	byte	0FA1	DMA Mode 0
DMAC1	byte	0FA2	DMA Control 1
DMAM1	byte	0FA3	DMA Mode 1

Table 6-2. RAM and Special Func. Reg. Mapping (Cont'd)

SFR Name		Type	Offset Meaning
DIC0	byte	0FAC	DMA I/O request Control 0
DIC1	byte	0FAD	DMA I/O request Control 1
SAR0L	byte	0FC0	DMA source address 0 Low
SAR0M	byte	0FC1	DMA source address 0 Middle
SAR0H	byte	0FC2	DMA source address 0 High
DAR0L	byte	0FC4	DMA destination address 0 Low
DAR0M	byte	0FC5	DMA destination address 0 Middle
DAR0H	byte	0FC6	DMA destination address 0 High
TC0	word	0FC8	DMA counter 0
TC0L	byte	0FC8	DMA counter 0 Low
TC0H	byte	0FC9	DMA counter 0 High
SAR1L	byte	0FD0	DMA source address 1 Low
SAR1M	byte	0FD1	DMA source address 1 Middle
SAR1H	byte	0FD2	DMA source address 1 High
DAR1L	byte	0FD4	DMA destination address 1 Low
DAR1M	byte	0FD5	DMA destination address 1 Middle
DAR1H	byte	0FD6	DMA destination address 1 High
TC1	word	0FD8	DMA counter 1
TC1L	byte	0FD8	DMA counter 1 Low
TC1H	byte	0FD9	DMA counter 1 High

The Instruction Set

This section contains the instruction set accepted by the asv20/asv33 assembler. All operand combinations are listed for each instruction. Some of these instructions or operand combinations are only valid in certain modes (such as V20 or V25). These restricted instructions are explained in the notes at the end of the list of instructions.

The V33 Family of processors includes the V33 and V53 microprocessors. These processors have the same instruction set as the V20 Family, with the addition of two new instructions. The two new instructions, BRKXA and RETXA, can be used when a complete V33 system is to be built. Normally, the V33 processor addresses only the first megabyte of memory, just like the V20 or V25. These new instructions give the user access to the full 16-megabyte address space of the V33. More information on the use of these instructions can be found in the *HP 64875 Extended Mode Locator User's Guide* or in the *NEC 70136 Microprocessor User's Manual*.

A special code denotes what operand patterns are allowed for each instruction. If no operands are noted, then none are expected for that instruction. Otherwise, each operand will have a name, indicating what the operand does, followed by a colon and a code indicating what type of operand is to be used. A list of these codes follows in a table. If an operand is restricted to certain values, then these values will be listed in parenthesis after the code. If more than one restricted value is possible, then they will be separated by commas. Numeric ranges will be denoted by their boundary values.

Table 6-3. Operand Codes

AB	AL only
AW	AW only
CB	SHORT label with current segment and within 127 bytes of current location
CD	FAR label, offset and base
CW	NEAR label, within current segment
D	17-bit immediate value
DB	1-byte immediate value, from -255 to 255
DW	2-byte immediate value, from -65535 to 65535
EB	either an 8-bit register or BYTE-type memory expression
ED	DWORD-type memory expression
EW	either a 16-bit register or WORD-type memory expression
F	8087 floating point stack register
I	various registers: R=all registers, PSW=processor status, CY=carry, DIR=direction flag
M	any type of memory expression
MB	BYTE-type memory expression
MW	WORD-type memory expression
MD	DWORD-type memory expression
MQ	QWORD-type memory expression
MT	TBYTE-type memory expression
RB	8-bit register
RW	16-bit register
S	segment register
SFR	V25 SFR register keyword
T	ST(0); top of 8087 floating point register stack
XB	BYTE-type, simple memory expression; no register indirection
V	72291 floating point register (FR0-FR7, FS0-FS7, FL0-FL7, FCTW, FPTW, FSTW)
XW	WORD-type, simple memory expression; no register indirection

Table 6-4. Assembler Instruction Set

ADD	dst:AB,src:DB	
ADD	dst:AW,src:DB	
ADD	dst:AW,src:DW	
ADD	dst:EB,src:DB	
ADD	dst:EB,src:RB	
ADD	dst:EW,src:DB(-128,127)	
ADD	dst:EW,src:DW	
ADD	dst:EW,src:RW	
ADD	dst:RB,src:EB	
ADD	dst:RW,src:EW	
ADD4S		
ADD4S	dst:M,src:M	(Note 1)
ADDC	dst:AB,src:DB	
ADDC	dst:AW,src:DB	
ADDC	dst:AW,src:DW	
ADDC	dst:EB,src:DB	
ADDC	dst:EB,src:RB	
ADDC	dst:EW,src:DB(-128,127)	
ADDC	dst:EW,src:DW	

Table 6-4. Assembler Instruction Set (Cont'd)

ADDC	dst:EW,src:RW	
ADDC	dst:RB,src:EB	
ADDC	dst:RW,src:EW	
ADJ4A		
ADJ4S		
ADJBA		
ADJBS		
AND	dst:AB,src:DB	
AND	dst:AW,src:DB	
AND	dst:AW,src:DW	
AND	dst:EB,src:DB	
AND	dst:EB,src:RB	
AND	dst:EW,src:DB	
AND	dst:EW,src:DW	
AND	dst:EW,src:RW	
AND	dst:RB,src:EB	
AND	dst:RW,src:EW	
BC	place:CB	
BCWZ	place:CB	
BE	place:CB	
BGE	place:CB	

Table 6-4. Assembler Instruction Set (Cont'd)

BGT	place:CB	
BH	place:CB	
BL	place:CB	
BLE	place:CB	
BLT	place:CB	
BN	place:CB	
BNC	place:CB	
BNE	place:CB	
BNH	place:CB	
BNL	place:CB	
BNV	place:CB	
BNZ	place:CB	
BP	place:CB	
BPE	place:CB	
BPO	place:CB	
BR	place:CB	
BR	place:CD	
BR	place:CW	
BR	place:EW	
BR	place:MD	
BRK	itype:DB(3)	

Table 6-4. Assembler Instruction Set (Cont'd)

BRK	itype:DB	
BRKCS	src:RW	(Note 3)
BRKEM	vector:DB	(Note 2)
BRKV		
BRKXA	vector:DB	(Note 4)
BTCLR	sfr:M,bit:DB(0,7),place:CB	(Note 3)
BUSLOCK	PREFX	
BV	place:CB	
BZ	place:CB	
CALL	addr:CB	
CALL	addr:CD	
CALL	addr:CW	
CALL	addr:ED	
CALL	addr:EW	
CHKIND	indx:RW,bptr:MD	
CHKIND	indx:RW,bptr:MW	
CLR1	carry:I(CY)	
CLR1	dst:EB,off:D(0,7)	
CLR1	dst:EB,off:RB(CL)	
CLR1	dst:EW,off:D(0,15)	
CLR1	dst:EW,off:RB(CL)	

Table 6-4. Assembler Instruction Set (Cont'd)

CLR1	updown:I(DIR)	
CMP	dst:AB,src:DB	
CMP	dst:AW,src:DB	
CMP	dst:AW,src:DW	
CMP	dst:EB,src:DB	
CMP	dst:EB,src:RB	
CMP	dst:EW,src:DB(-128,127)	
CMP	dst:EW,src:DW	
CMP	dst:EW,src:RW	
CMP	dst:RB,src:EB	
CMP	dst:RW,src:EW	
CMP4S		
CMP4S	dst:M,src:M	(Note 1)
CMPBK	IX__ptr:MB,IY__ptr:MB	(Note 1)
CMPBK	IX__ptr:MW,IY__ptr:MW	(Note 1)
CMPBKB		
CMPBKW		
CMPM	IY__ptr:MB	(Note 1)
CMPM	IY__ptr:MW	(Note 1)
CMPMB		
CMPMW		

Table 6-4. Assembler Instruction Set (Cont'd)

CVTBD		
CVTBW		
CVTDB		
CVTWL		
DBNZ	place:CB	
DBNZE	place:CB	
DBNZNE	place:CB	
DEC	dst:EB	
DEC	dst:EW	
DEC	dst:RW	
DI		
DISPOSE		
DIV	divisor:EB	
DIV	divisor:EW	
DIVU	divisor:EB	
DIVU	divisor:EW	
EI		
ESC	opcode:DB(0,63),addr:EB	
ESC	opcode:DB(0,63),addr:ED	
ESC	opcode:DB(0,63),addr:EW	
EXT	dst:RB,count:D(0,15)	

Table 6-4. Assembler Instruction Set (Cont'd)

EXT	dst:RB,src:RB	
F2XM1		(Note 5)
FABS		(Note 5)
FABS	dst:V(FL0),src:V(FL0,FL7)	(Note 7)
FABS	dst:V(FS0),src:V(FS0,FS7)	(Note 7)
FACOS	dst:V(FL0),src:V(FL0,FL7)	(Note 7)
FACOS	dst:V(FS0),src:V(FS0,FS7)	(Note 7)
FADD		(Note 5)
FADD	dst:F,src:T	(Note 5)
FADD	dst:T,src:F	(Note 5)
FADD	dst:V(FL0),src:MQ	(Note 7)
FADD	dst:V(FL0),src:V(FL0,FL7)	(Note 7)
FADD	dst:V(FS0),src:MD	(Note 7)
FADD	dst:V(FS0),src:V(FS0,FS7)	(Note 7)
FADD	memop:MD	(Note 5)
FADD	memop:MQ	(Note 5)
FADDP	dst:F,src:T	(Note 5)
FASIN	dst:V(FL0),src:V(FL0,FL7)	(Note 7)
FASIN	dst:V(FS0),src:V(FS0,FS7)	(Note 7)
FATAN	dst:V(FL0),src:V(FL0,FL7)	(Note 7)
FATAN	dst:V(FS0),src:V(FS0,FS7)	(Note 7)

Table 6-4. Assembler Instruction Set (Cont'd)

FATAN2	dst:V(FL0),src:V(FL0,FL7)	(Note 7)
FATAN2	dst:V(FS0),src:V(FS0,FS7)	(Note 7)
FBLD	memop:MT	(Note 5)
FBSTP	memop:MT	(Note 5)
FCHS		(Note 5)
FCLEX		(Note 5)
FCMP	dst:V(FL0),src:MQ	(Note 7)
FCMP	dst:V(FL0),src:V(FL0,FL7)	(Note 7)
FCMP	dst:V(FS0),src:MD	(Note 7)
FCMP	dst:V(FS0),src:V(FS0,FS7)	(Note 7)
FCMPA	dst:V(FL0),src:MQ	(Note 7)
FCMPA	dst:V(FL0),src:V(FL0,FL7)	(Note 7)
FCMPA	dst:V(FS0),src:MD	(Note 7)
FCMPA	dst:V(FS0),src:V(FS0,FS7)	(Note 7)
FCMPAE	dst:V(FL0),src:MQ	(Note 7)
FCMPAE	dst:V(FL0),src:V(FL0,FL7)	(Note 7)
FCMPAE	dst:V(FS0),src:MD	(Note 7)
FCMPAE	dst:V(FS0),src:V(FS0,FS7)	(Note 7)
FCMPE	dst:V(FL0),src:MQ	(Note 7)
FCMPE	dst:V(FL0),src:V(FL0,FL7)	(Note 7)
FCMPE	dst:V(FS0),src:MD	(Note 7)

Table 6-4. Assembler Instruction Set (Cont'd)

FCMPE	dst:V(FS0),src:V(FS0,FS7)	(Note 7)
FCOM		(Note 5)
FCOM	fpst:F	(Note 5)
FCOM	memop:MD	(Note 5)
FCOM	memop:MQ	(Note 5)
FCOMP		(Note 5)
FCOMP	fpst:F	(Note 5)
FCOMP	memop:MD	(Note 5)
FCOMP	memop:MQ	(Note 5)
FCOMPP		(Note 5)
FCOS	dst:V(FL0),src:V(FL0,FL7)	(Note 7)
FCOS	dst:V(FS0),src:V(FS0,FS7)	(Note 7)
FCVTD	dst:V(FL0),src:MD	(Note 7)
FCVTDS	dst:V(FS0),src:MD	(Note 7)
FCVTLD	dst:MD,src:V(FL0)	(Note 7)
FCVTLQ	dst:MQ,src:V(FL0)	(Note 7)
FCVTLS	dst:MD,src:V(FL0)	(Note 7)
FCVTLS	dst:V(FS0),src:MQ	(Note 7)
FCVTLS	dst:V(FS0),src:V(FL0,FL7)	(Note 7)
FCVTLS	dst:V(FS0,FS7),src:V(FL0)	(Note 7)
FCVTQL	dst:V(FL0),src:MQ	(Note 7)

Table 6-4. Assembler Instruction Set (Cont'd)

FCVTQS	dst:V(FS0),src:MQ	(Note 7)
FCVTSD	dst:MD,src:V(FS0)	(Note 7)
FCVTSL	dst:MQ,src:V(FS0)	(Note 7)
FCVTSL	dst:V(FL0),src:MD	(Note 7)
FCVTSL	dst:V(FL0),src:V(FS0,FS7)	(Note 7)
FCVTSL	dst:V(FL0,FL7),src:V(FS0)	(Note 7)
FCVTSQ	dst:MQ,src:V(FS0)	(Note 7)
FDECSTP		(Note 5)
FDIAG		(Note 7)
FDISI		(Note 5)
FDIV		(Note 5)
FDIV	dst:F,src:T	(Note 5)
FDIV	dst:T,src:F	(Note 5)
FDIV	dst:V(FL0),src:MQ	(Note 7)
FDIV	dst:V(FL0),src:V(FL0,FL7)	(Note 7)
FDIV	dst:V(FS0),src:MD	(Note 7)
FDIV	dst:V(FS0),src:V(FS0,FS7)	(Note 7)
FDIV	memop:MD	(Note 5)
FDIV	memop:MQ	(Note 5)
FDIVP	dst:F,src:T	(Note 5)
FDIVR		(Note 5)

Table 6-4. Assembler Instruction Set (Cont'd)

FDIVR	dst:F,src:T	(Note 5)
FDIVR	dst:T,src:F	(Note 5)
FDIVR	memop:MD	(Note 5)
FDIVR	memop:MQ	(Note 5)
FDIVRP	dst:F,src:T	(Note 5)
FENI		(Note 5)
FEXPE	dst:V(FL0),src:V(FL0,FL7)	(Note 7)
FEXPE	dst:V(FS0),src:V(FS0,FS7)	(Note 7)
FEXPEM1	dst:V(FL0),src:V(FL0,FL7)	(Note 7)
FEXPEM1	dst:V(FS0),src:V(FS0,FS7)	(Note 7)
FFREE	fpst:F	(Note 5)
FIADD	memop:MD	(Note 5)
FIADD	memop:MW	(Note 5)
FICOM	memop:MD	(Note 5)
FICOM	memop:MW	(Note 5)
FICOMP	memop:MD	(Note 5)
FICOMP	memop:MW	(Note 5)
FIDIV	memop:MD	(Note 5)
FIDIV	memop:MW	(Note 5)
FIDIVR	memop:MD	(Note 5)

Table 6-4. Assembler Instruction Set (Cont'd)

FIDIVR	memop:MW	(Note 5)
FILD	memop:MD	(Note 5)
FILD	memop:MQ	(Note 5)
FILD	memop:MW	(Note 5)
FIMUL	memop:MD	(Note 5)
FIMUL	memop:MW	(Note 5)
FINCSTP		(Note 5)
FINIT		
FINT		(Note 3)
FIP3V	dst:V(FL0),src:V(FL0,FL7)	(Note 7)
FIP3V	dst:V(FS0),src:V(FS0,FS7)	(Note 7)
FIP4V	dst:V(FL0),src:V(FL0,FL7)	(Note 7)
FIP4V	dst:V(FS0),src:V(FS0,FS7)	(Note 7)
FIST	memop:MD	(Note 5)
FIST	memop:MW	(Note 5)
FISTP	memop:MD	(Note 5)
FISTP	memop:MQ	(Note 5)
FISTP	memop:MW	(Note 5)
FISUB	memop:MD	(Note 5)
FISUB	memop:MW	(Note 5)
FISUBR	memop:MD	(Note 5)

Table 6-4. Assembler Instruction Set (Cont'd)

FISUBR	memop:MW	(Note 5)
FLD	fpst:F	(Note 5)
FLD	memop:MD	(Note 5)
FLD	memop:MQ	(Note 5)
FLD	memop:MT	(Note 5)
FLD1		(Note 5)
FLDCW	memop:M	(Note 5)
FLDENV	memop:M	(Note 5)
FLDL2E		(Note 5)
FLDL2T		(Note 5)
FLDLG2		(Note 5)
FLDLN2		(Note 5)
FLDPI		(Note 5)
FLDZ		(Note 5)
FLOGE	dst:V(FL0),src:V(FL0,FL7)	(Note 7)
FLOGE	dst:V(FS0),src:V(FS0,FS7)	(Note 7)
FMOD	dst:V(FL0),src:MQ	(Note 7)
FMOD	dst:V(FL0),src:V(FL0,FL7)	(Note 7)
FMOD	dst:V(FS0),src:MD	(Note 7)
FMOD	dst:V(FS0),src:V(FS0,FS7)	(Note 7)
FMOV	dst:MD,src:V(FS0)	(Note 7)

Table 6-4. Assembler Instruction Set (Cont'd)

FMOV	dst:MQ,src:V(FL0)	(Note 7)
FMOV	dst:V(FL0),src:MQ	(Note 7)
FMOV	dst:V(FL0),src:V(FL0,FL7)	(Note 7)
FMOV	dst:V(FL0,FL7),src:V(FL0)	(Note 7)
FMOV	dst:V(FS0),src:MD	(Note 7)
FMOV	dst:V(FS0),src:V(FS0,FS7)	(Note 7)
FMOV	dst:V(FS0,FS7),src:V(FS0)	(Note 7)
FMOVCR	dst:MD,src:V(FCTW)	(Note 7)
FMOVCR	dst:MD,src:V(FPTW)	(Note 7)
FMOVCR	dst:MD,src:V(FSTW)	(Note 7)
FMOVCR	dst:V(FCTW),src:MD	(Note 7)
FMOVCR	dst:V(FPTW),src:MD	(Note 7)
FMOVCR	dst:V(FSTW),src:MD	(Note 7)
FMOVRT	dst:M,src:V(FR0,FR7)	(Note 7)
FMOVRT	dst:V(FR0,FR7),dst:M	(Note 7)
FMUL		(Note 5)
FMUL	dst:F,src:T	(Note 5)
FMUL	dst:T,src:F	(Note 5)
FMUL	dst:V(FL0),src:MQ	(Note 7)
FMUL	dst:V(FL0),src:V(FL0,FL7)	(Note 7)
FMUL	dst:V(FS0),src:MD	(Note 7)

Table 6-4. Assembler Instruction Set (Cont'd)

FMUL	dst:V(FS0),src:V(FS0,FS7)	(Note 7)
FMUL	memop:MD	(Note 5)
FMUL	memop:MQ	(Note 5)
FMULP	dst:F,src:T	(Note 5)
FNCLEX		(Note 5)
FNDISI		(Note 5)
FNEG	dst:MD,src:V(FS0)	(Note 7)
FNEG	dst:MQ,src:V(FL0)	(Note 7)
FNEG	dst:V(FL0),src:MQ	(Note 7)
FNEG	dst:V(FL0),src:V(FL0,FL7)	(Note 7)
FNEG	dst:V(FL0,FL7),src:V(FL0)	(Note 7)
FNEG	dst:V(FS0),src:MD	(Note 7)
FNEG	dst:V(FS0),src:V(FS0,FS7)	(Note 7)
FNEG	dst:V(FS0,FS7),src:V(FS0)	(Note 7)
FNENI		(Note 5)
FNINIT		(Note 5)
FNOP		(Note 5)
FNSAVE	memop:M	(Note 5)
FNSTCW	memop:M	(Note 5)
FNSTENV	memop:M	(Note 5)
FNSTSW	dst:AW	(Note 6)

Table 6-4. Assembler Instruction Set (Cont'd)

FNSTSW	memop:M	(Note 5)
FPATAN		(Note 5)
FPO1	opcode:D(0,511)	
FPO1	opcode:D(0,63),addr:MB	
FPO1	opcode:D(0,63),addr:MD	
FPO1	opcode:D(0,63),addr:MQ	
FPO1	opcode:D(0,63),addr:MT	
FPO1	opcode:D(0,63),addr:MW	
FPO2	opcode:D(0,127)	
FPO2	opcode:D(0,15),addr:MB	
FPO2	opcode:D(0,15),addr:MD	
FPO2	opcode:D(0,15),addr:MQ	
FPO2	opcode:D(0,15),addr:MT	
FPO2	opcode:D(0,15),addr:MW	
FPOWER	dst:V(FL0),src:MQ	(Note 7)
FPOWER	dst:V(FL0),src:V(FL0,FL7)	(Note 7)
FPOWER	dst:V(FS0),src:MD	(Note 7)
FPOWER	dst:V(FS0),src:V(FS0,FS7)	(Note 7)
FPREM		(Note 5)
FPTAN		(Note 5)
FREM	dst:V(FL0),src:MQ	(Note 7)

Table 6-4. Assembler Instruction Set (Cont'd)

FREM	dst:V(FL0),src:V(FL0,FL7)	(Note 7)
FREM	dst:V(FS0),src:MD	(Note 7)
FREM	dst:V(FS0),src:V(FS0,FS7)	(Note 7)
FRND	dst:V(FL0),src:V(FL0,FL7)	(Note 7)
FRND	dst:V(FS0),src:V(FS0,FS7)	(Note 7)
FRNDINT		(Note 5)
FRPOP		(Note 7)
FRPUSH		(Note 7)
FRSTOR	memop:M	(Note 5)
FSAVE	memop:M	(Note 5)
FSCALE		(Note 5)
FSIN	dst:V(FL0),src:V(FL0,FL7)	(Note 7)
FSIN	dst:V(FS0),src:V(FS0,FS7)	(Note 7)
FSINCOS	dst:V(FL0),src:V(FL0,FL7)	(Note 7)
FSINCOS	dst:V(FS0),src:V(FS0,FS7)	(Note 7)
FSQRT		(Note 5)
FSQRT	dst:V(FL0),src:V(FL0,FL7)	(Note 7)
FSQRT	dst:V(FS0),src:V(FS0,FS7)	(Note 7)
FST	fpst:F	(Note 5)
FST	memop:MD	(Note 5)
FST	memop:MQ	(Note 5)

Table 6-4. Assembler Instruction Set (Cont'd)

FSTCW	memop:M	(Note 5)
FSTENV	memop:M	(Note 5)
FSTP	fpst:F	(Note 5)
FSTP	memop:MD	(Note 5)
FSTP	memop:MQ	(Note 5)
FSTP	memop:MT	(Note 5)
FSTSW	dst:AW	(Note 6)
FSTSW	memop:M	(Note 5)
FSUB		(Note 5)
FSUB	dst:T,src:F	(Note 5)
FSUB	dst:V(FL0),src:MQ	(Note 7)
FSUB	dst:V(FL0),src:V(FL0,FL7)	(Note 7)
FSUB	dst:V(FS0),src:MD	(Note 7)
FSUB	dst:V(FS0),src:V(FS0,FS7)	(Note 7)
FSUB	dstF,src:T	(Note 5)
FSUB	memop:MD	(Note 5)
FSUB	memop:MQ	(Note 5)
FSUBP	dst:F,src:T	(Note 5)
FSUBR		(Note 5)
FSUBR	dst:F,src:T	(Note 5)
FSUBR	dst:T,src:F	(Note 5)

Table 6-4. Assembler Instruction Set (Cont'd)

FSUBR	memop:MD	(Note 5)
FSUBR	memop:MQ	(Note 5)
FSUBRP	dst:F,src:T	(Note 5)
FTAN	dst:V(FL0),src:V(FL0,FL7)	(Note 7)
FTAN	dst:V(FS0),src:V(FS0,FS7)	(Note 7)
FTST		(Note 5)
FWAIT		(Note 5)
FXAM		(Note 5)
FXCH		(Note 5)
FXCH	dst:V(FL0),src:V(FL0,FL7)	(Note 7)
FXCH	dst:V(FS0),src:V(FS0,FS7)	(Note 7)
FXCH	fpst:F	(Note 5)
FXTRACT		(Note 5)
FYL2X		(Note 5)
FYL2XP1		(Note 5)
HALT		
IN	dst:AB,port:DB	
IN	dst:AB,port:RW(DW)	
IN	dst:AW,port:DB	
IN	dst:AW,port:RW(DW)	
INC	dst:EB	

Table 6-4. Assembler Instruction Set (Cont'd)

INC	dst:EW	
INC	dst:RW	
INM	IY__ptr:EB,port:RW(DW)	(Note 1)
INM	IY__ptr:EW,port:RW(DW)	(Note 1)
INS	dst:RB,count:D(0,15)	
INS	dst:RB,src:RB	
LDEA	dst:RW,src:M	
LDM	IX__ptr:MB	
LDM	IX__ptr:MW	
LDMB		
LDMW		
MOV	dst:AB,src:XB	
MOV	dst:AW,src:XW	
MOV	dst:EB,src:DB	
MOV	dst:EB,src:RB	
MOV	dst:EW,src:DB	
MOV	dst:EW,src:DW	
MOV	dst:EW,src:RW	
MOV	dst:EW,src:S	
MOV	dst:I(PSW),src:RB(AH)	
MOV	dst:RB(AH),src:I(PSW)	

Table 6-4. Assembler Instruction Set (Cont'd)

MOV	dst:RB,src:EB	
MOV	dst:RW,src:EW	
MOV	dst:S(DS1),src:EW	
MOV	dst:S(SS,DS0),src:EW	
MOV	dst:XB,src:AB	
MOV	dst:XW,src:AW	
MOV	seg:S(DS0),dst:RW,src:MD	
MOV	seg:S(DS1),dst:RW,src:MD	
MOVBK	IY__ptr:MB,IX__ptr:MB	(Note 1)
MOVBK	IY__ptr:MW,IX__ptr:MW	(Note 1)
MOVBKB		
MOVBKW		
MOVSPA		(Note 3)
MOVSPB	src:RW	(Note 3)
MUL	dst:RW,src1:EW,src2:DB(-128,127)	
MUL	dst:RW,src1:EW,src2:DW	
MUL	dst:RW,src2:DB(-128,127)	
MUL	dst:RW,src2:DW	
MUL	mplier:EB	
MUL	mplier:EW	
MULU	mplier:EB	

Table 6-4. Assembler Instruction Set (Cont'd)

MULU	mplier:EW	
NEG	dst:EB	
NEG	dst:EW	
NOP		
NOT	dst:EB	
NOT	dst:EW	
NOT1	carry:I(CY)	
NOT1	dst:EB,off:D(0,7)	
NOT1	dst:EB,off:RB(CL)	
NOT1	dst:EW,off:D(0,15)	
NOT1	dst:EW,off:RB(CL)	
OR	dst:AB,src:DB	
OR	dst:AW,src:DB	
OR	dst:AW,src:DW	
OR	dst:EB,src:DB	
OR	dst:EB,src:RB	
OR	dst:EW,src:DB	
OR	dst:EW,src:DW	
OR	dst:EW,src:RW	
OR	dst:RB,src:EB	
OR	dst:RW,src:EW	

Table 6-4. Assembler Instruction Set (Cont'd)

OUT	port:DB,dst:AB	
OUT	port:DB,dst:AW	
OUT	port:RW(DW),dst:AB	
OUT	port:RW(DW),dst:AW	
OUTM	port:RS(DW),IX_ptr:EW	
OUTM	port:RW(DW),IX_ptr:EB	
POLL		
POP	dst:EW	
POP	dst:I(PSW)	
POP	dst:I(R)	
POP	dst:S(DS0)	
POP	dst:S(SS,DS1)	
PREPARE	disp:D(0,0FFFFH),level:D(0,255)	
PUSH	src:DB(-128,127)	
PUSH	src:DW	
PUSH	src:EW	
PUSH	src:I(PSW)	
PUSH	src:I(R)	
PUSH	src:S	
REP	PREFX	
REPC	PREFX	

Table 6-4. Assembler Instruction Set (Cont'd)

REPE	PREFX	
REPNC	PREFX	
REPNE	PREFX	
REPNZ	PREFX	
REPZ	PREFX	
RET		
RET	src:DW	
RETI		
RETRBI		(Note 3)
RETXA	vector:DB	(Note 4)
ROL	dst:EB,count:DB(0,31)	
ROL	dst:EB,count:DB(1)	
ROL	dst:EB,count:RB(CL)	
ROL	dst:EW,count:DB(0,31)	
ROL	dst:EW,count:DB(1)	
ROL	dst:EW,count:RB(CL)	
ROL4	dst:EB	
ROLC	dst:EB,count:DB(0,31)	
ROLC	dst:EB,count:DB(1)	
ROLC	dst:EB,count:RB(CL)	
ROLC	dst:EW,count:DB(0,31)	

Table 6-4. Assembler Instruction Set (Cont'd)

ROLC	dst:EW,count:DB(1)	
ROLC	dst:EW,count:RB(CL)	
ROR	dst:EB,count:DB(0,31)	
ROR	dst:EB,count:DB(1)	
ROR	dst:EB,count:RB(CL)	
ROR	dst:EW,count:DB(0,31)	
ROR	dst:EW,count:DB(1)	
ROR	dst:EW,count:RB(CL)	
ROR4	dst:EB	
RORC	dst:EB,count:DB(0,31)	
RORC	dst:EB,count:DB(1)	
RORC	dst:EB,count:RB(CL)	
RORC	dst:EW,count:DB(0,31)	
RORC	dst:EW,count:DB(1)	
RORC	dst:EW,count:RB(CL)	
SET1	carry:I(CY)	
SET1	dst:EB,off:D(0,7)	
SET1	dst:EB,off:RB(CL)	
SET1	dst:EW,off:D(0,15)	
SET1	dst:EW,off:RB(CL)	
SET1	updown:I(DIR)	

Table 6-4. Assembler Instruction Set (Cont'd)

SHL	dst:EB,count:DB(0,31)	
SHL	dst:EB,count:DB(1)	
SHL	dst:EB,count:RB(CL)	
SHL	dst:EW,count:DB(0,31)	
SHL	dst:EW,count:DB(1)	
SHL	dst:EW,count:RB(CL)	
SHR	dst:EB,count:DB(0,31)	
SHR	dst:EB,count:DB(1)	
SHR	dst:EB,count:RB(CL)	
SHR	dst:EW,count:DB(0,31)	
SHR	dst:EW,count:DB(1)	
SHR	dst:EW,count:RB(CL)	
SHRA	dst:EB,count:DB(0,31)	
SHRA	dst:EB,count:DB(1)	
SHRA	dst:EB,count:RB(CL)	
SHRA	dst:EW,count:DB(0,31)	
SHRA	dst:EW,count:DB(1)	
SHRA	dst:EW,count:RB(CL)	
STM	IY__ptr:MB	(Note 1)
STM	IY__ptr:MW	(Note 1)
STMB		

Table 6-4. Assembler Instruction Set (Cont'd)

STMW		
STOP		(Note 3)
SUB	dst:AB,src:DB	
SUB	dst:AW,src:DB	
SUB	dst:AW,src:DW	
SUB	dst:EB,src:DB	
SUB	dst:EB,src:RB	
SUB	dst:EW,src:DB(-128,127)	
SUB	dst:EW,src:DW	
SUB	dst:EW,src:RW	
SUB	dst:RB,src:EB	
SUB	dst:RW,src:EW	
SUB4S		
SUB4S	dst:M,src:M	(Note 1)
SUBC	dst:AB,src:DB	
SUBC	dst:AW,src:DB	
SUBC	dst:AW,src:DW	
SUBC	dst:EB,src:DB	
SUBC	dst:EB,src:RB	
SUBC	dst:EW,src:DB(-128,127)	
SUBC	dst:EW,src:DW	

Table 6-4. Assembler Instruction Set (Cont'd)

SUBC	dst:EW,src:RW	
SUBC	dst:RB,src:EB	
SUBC	dst:RW,src:EW	
TEST	dst:AB,src:DB	
TEST	dst:AW,src:DB	
TEST	dst:AW,src:DW	
TEST	dst:EB,src:DB	
TEST	dst:EB,src:RB	
TEST	dst:EW,src:DB	
TEST	dst:EW,src:DW	
TEST	dst:EW,src:RW	
TEST	dst:RB,src:EB	
TEST	dst:RW,src:EW	
TEST1	dst:EB,off:D(0,7)	
TEST1	dst:EB,off:RB(CL)	
TEST1	dst:EW,off:D(0,15)	
TEST1	dst:EW,off:RB(CL)	
TRANS		
TRANS	table:MB	
TRANSB		
TSKSW	src:RW	(Note 3)

7

Assembler Controls

Introduction

Assembler controls are internal assembler switches which let you enable and disable certain aspects of the assembly process. This chapter describes assembler controls and control defaults.

The beginning of the chapter discusses the general syntax of assembler controls and other topics that are true about controls in the general sense.

The remainder of this chapter contains a list of controls, a description of their functions, and a discussion of the differences between the two possible processor specifications. The controls are divided into primary and general controls.

The control references, unlike most other references in this manual, do not list the syntax under a separate heading. Instead, the syntax is shown directly in the control name heading in the left column. If a [NO] appears in the heading, it indicates that the word NO can be prefixed to a control to make it do the opposite of what the control does normally. For example, LIST turns on the output listing, but NOLIST turns off printing of the listing. (The -L command line option causes a listing to be generated.)

Following the heading is the shorthand version of the command. Next is the default setting for a given control (if one makes sense). Finally, a short discussion of the control may be present.

General Syntax for Assembler Controls

The syntax of a control line in the source code is:

```
$control[(parameter)] [...]
```

The dollar sign may be preceded by tabs or blanks. Separators must be included between adjacent controls. Examples:

```
$XREF
$INCLUDE(filename) DEBUG SYMBOLS
$PRINT ERRORPRINT(FILENAME)
```

Primary and General Controls

Assembler controls are classified as either primary or general. Primary control statements occur only on the first few lines of the source program before any non-control statements (other than comments and blank lines). Primary controls are not processed when they occur after any statement other than a control line; their presence after any statement other than a control line causes an error. General controls, however, can be specified at any time in the source program. In most instances, an error in either kind of control line causes all remaining controls on the line to be ignored.

Controls on the Command Line

Assembler controls may also be included on the command line when the assembler is invoked. If a primary control is entered on both the command line and in the first lines of the source file being assembled, the control from the command line overrides the control in the file for that particular assembly.

If a general control is entered on both the command line and in the file (since general controls can appear anywhere in the file, the general control might be far, relatively, from the beginning of the file), then the control from the command line is in effect until the control in the source file is found. At that point, the source file control overrides the command line control for the rest of the assembly.

Control Conflicts

If a primary control conflicts with another primary control and both are in the source file, then the one that appears last takes effect. If the conflict is between a control on the command line and one in the file, then the control which appears on the command line overrides the one in the file.

If general controls conflict (whether both in the file or one on the command line and one in the file), then the control which appears last will be the one to take effect. Example:

```
$NOLIST
$LIST  ;this control is last, it will be the one
         ;to take effect
```

Controls and File Names

Certain controls accept a file name as a parameter. The file name parameters are optional, except with INCLUDE, and are ignored with all controls except INCLUDE. The [NO] form of these controls does not accept a file name.

Control Abbreviations

Each control can be abbreviated to a two-character or three-character equivalent; the abbreviations are listed with each control. Abbreviations may be negated if the full name of the control can be negated. Controls are not case-sensitive; upper-case and lower-case letters are equivalent. Remember that their arguments may be case sensitive, although the controls are not.

Controls and Macro Preprocessor (apv20/apv33)

Most controls are used only by the assembler. The INCLUDE control acts differently, however, if the source file is processed by the macro preprocessor before assembly. If the source file contains INCLUDE controls and does go through the macro preprocessor, then the macro preprocessor will expand the INCLUDEs. The output from the preprocessor will then contain the include files and will no longer contain the INCLUDE controls (not a problem, since they are no longer necessary). The macro preprocessor does not act on any other assembler controls.

Primary Controls

[NO]CAPITALS

shorthand = [NO]CA, [NO]CAP
default = NO CAPITALS

Causes symbols to be case insensitive. That is, upper and lower case letters are treated as the same character. If this control is negated, then all lower case characters in symbols will be treated as separate from the upper case characters. This control does not affect text within strings (except for class names).

Note 

All Intel-generated OMF will contain case insensitive symbols.

DATE(string)

shorthand = DA

(No default necessary.)

The DATE control has no effect. It is supplied for Intel compatibility, and its use will not generate an error. The date printed on the listing and placed in the object file is obtained from the operating system.

[NO]DEBUG

shorthand = [NO]DB , [NO]DBG
default = DEBUG

Causes symbolic debug and type information to be placed into the output object module. By default, only non-PURGEd variables, labels and numbers are placed into the object module.

[NO]ERRORPRINT (filename)

shorthand = [NO]EP , [NO]ERR
default= ERRORPRINT

This control causes error and warning information to be displayed on standard error. The filename, if present, is ignored and is only allowed for Intel compatibility. The noerrorprint control suppresses error and warning messages from being displayed on standard error. The nowarning control may be used to suppress only warning messages while allowing error messages to be displayed.

EXTERN_CHECK

shorthand = [NO]EC
default = extern_check

This control causes the use of external symbols to check that an ASSUME register has been defined such that the external symbol can be referenced from the ASSUME register. The noextern_check control causes the assembler to allow any use of an external symbol without verifying that the symbol is accessible through whatever assume register is used to reference that symbol. This then requires the user to make sure that segment registers are correctly set up to reference the segment that a given external symbol belongs to.

GROUP_INFO

shorthand = [NO]GI
default = group_info

This control causes the debug information emitted from the assembler to associate group information to all symbols that belong to segments that are members of a group. Only one group will be assigned to a given symbol, regardless of how many groups a given segment belongs to. The nogroup_info control will only associate group information to labels and procedures; variables will NOT have group information associated.

[NO]HLASSYM shorthand = [NO]HA
default = NOHLASSYM

Causes asv20/asv33 to generate low-level symbol information for static procedures, static data, and embedded assembly code. This option is useful when compiler-generated output is to be debugged in an emulator. If the output is to be debugged in AxDB or AxDE, then the negated form of this option is recommended.

[NO]MACRO(string) shorthand = [NO]MC, [NO]MAC

(No default necessary.)

Enables or disables macro assembly. Since macro processing is accomplished by a separate program, this control has no effect in either the assembler or the macro preprocessor. It is supplied for Intel compatibility, and its use will not generate an error.

MOD087 default: MOD087 for asv20, MOD72291 for asv33

Causes the Intel 8087 instruction set to be available. The output for these instructions may be linked to the Intel 8087 emulation library.

MOD287 default: MOD087 for asv20, MOD72291 for asv33

Causes the Intel 80287 instruction set to be available. This instruction set is nearly identical to the 8087 instruction set, with the addition of one new operand combination for FSTSW/FNSTSW.

MOD72291 default: MOD087 for asv20, MOD72291 for asv33

Causes the NEC 72291 instruction set to be available.

MODV20 shorthand = M2
default = MODV20 for asv20, MODV33 for asv33

Identifies the target microprocessor as V20. Causes V20/V30/V40/V50 instruction set to be recognized. Errors or warnings will be issued when instructions from conflicting instruction sets are encountered.

MODV25

shorthand = M5
default = MODV20 for asv20, MODV33 for asv33

Causes V25/V25+/V35/V35+ instruction set to be recognized. The ASGNSFR and SETIDB directives are also recognized. These directives allow the V25 RAM and SFR keywords to be used with the new V25 instructions or the normal V20 instructions.

MODV33

shorthand = M3
default = MODV20 for asv20, MODV33 for asv33

Causes V33 instructions to be recognized in addition to the V20 instruction set. It also allows DS and DL data directives to be used.

[NO]OBJECT (filename)

shorthand = [NO]OJ , [NO]OBJ

default = OBJECT

Generates an output object module, but the optional file name is ignored and only allowed for Intel compatibility. The assembler gives the object file the same root name as the source file, with a '.o' (dot lower case o) default file name extension.

OPTIMIZE

shorthand = OP

This control will cause the assembler to spend extra time processing the input file so the resulting object file has as few NOPs as possible. These NOPs are generated when forward references are used in expressions. The assembler does not always know how many bytes of output will be produced for a given instruction, so it saves extra space. If the instruction turns out to be shorter than that size, then the assembler pads the rest of the length with NOP bytes. This control will allow the assembler to spend time removing these NOPs when they are generated under these conditions. Note that this control will cause the assembler to run for a longer time than it otherwise would.

PAGELENGTH(n)

shorthand = PL , LEN
default = 55 lines per page

Specifies the page length of the listing as "n" lines, where n= 20 or more lines.

PAGEWIDTH(n)

shorthand = PW , WID
default= 132 characters per line

Specifies the listing page width in number of characters, where n is a number between 60 and 255, inclusive. Lines exceeding the current page width are wrapped to the next line.

[NO]PAGING

shorthand = [NO]PI
default= PAGING

Formats the output listing so as to have headers at the top of each page. By default, the headers supply the assembler name, title, and the date. If NOPAGING is specified, then the listing does not contain page headers or page ejects (except for an initial header on the first page). This option is only useful if a listing is produced.

[NO]PRINT(filename)

shorthand = [NO]PR , [NO]PRI
default = NOPRINT

Recognized for Intel compatibility only and has no effect. An error summary still goes to stdout and error messages go to stderr.

[NO]SYMBOLS

shorthand = [NO]SB , [NO]SYM
default= SYMBOLS

Prints an alphabetically sorted symbol table with the output listing. The listing will not contain cross-reference information. Cross-reference information is produced with the XREF control. If XREF is used, it will override this control and cross-reference information will be produced. This option is only useful if a listing is output.

[NO]TYPE

shorthand = [NO]TY
default = TYPE

This control is recognized for Intel compatibility only and its use will not have any effect. Whether type information is generated depends upon the DEBUG control being on.

[NO]UNREFERENCED_ EXTERNALS

shorthand = [NO]UE
default = NOUNREFERENCED_EXTERNALS

This control will cause all external symbols, including those that are unreferenced, to appear in the generated object file. In certain cases, these externals may be used to cause certain object files to be linked at link time. If this control is not present or if the NOUNREFERENCED_EXTERNALS control is used, any unreferenced externals will be removed from the resulting object file. This form of the control is useful when using inline functions in the AxLS C compiler. This will prevent unnecessary routines from being linked in that are being processed inline.

WARNING

shorthand = [NO]WA
default = WARNING

This control causes warning messages to be displayed along with any error messages that may appear on standard error. The nowarning control suppresses the warning messages so only error information is sent to standard error. The errorprint control overrides either form of this control in determining whether any information is sent to standard error or not.

WORKFILES(...)

shorthand = WF , WOR

(No default necessary.)

This control has no effect. It is supplied for Intel compatibility, and its use will not generate an error.

[NO]XREF shorthand = [NO]XR [NO]XRE
default = NOXREF

Prints a cross reference table on the output listing. If you use both the XREF and SYMBOLS controls, a cross reference table will be generated.

General Controls

EJECT

shorthand = EJ , EJE

(No default necessary.)

Advances the listing form to the beginning of the next page and prints a new header. This is only useful if a listing is being generated and paging is in effect.

[NO]GEN

shorthand = [NO]GE

(No default necessary.)

This control has no effect in either the assembler or macro preprocessor. It is supplied for Intel compatibility, and its use will not generate an error.

GENONLY

shorthand = GO

(No default necessary.)

This control has no effect in either the assembler or macro preprocessor. It is supplied for Intel compatibility, and its use will not generate an error.

INCLUDE(filename)

shorthand = IC , INC

(No default necessary.)

Indicates that the specified file should be included in the source input before the next line of the current source file is processed. Unlike other controls, INCLUDE must appear on a line by itself. No other controls, or other INCLUDEs, can be on the same line.

Note 

Include allows you to specify a different directory than the current working directory for include files by allowing a path name with the file name. However, if the file in the other path also has an INCLUDE control and that control does not have a path name as part of the file name, then the assembler will look back into the current working directory for the new include file. That means that if an include file in another directory needs another file to be complete, do not expect the assembler to pick this file up from that other directory along with the first include file.

[NO]LIST

shorthand = [NO]LI , [NO]LIS
default = LIST

Turns on assembly listing at any point in the program. If used in combination with NOLIST, you can list a portion of the source file. NOLIST overrides XREF and SYMBOLS. An error summary still goes to stdout and errors still go to stderr regardless of LIST setting.

RESTORE

shorthand = RS

(No default necessary.)

Restores, as the current settings, the most recently-saved settings for LIST/NOLIST that are on the stack. This control is used mainly to restore LIST/NOLIST settings after returning from INCLUDE files.

SAVE shorthand = SA

(No default necessary.)

Saves the current settings of LIST and NOLIST controls on a stack up to 64 deep. This control remains in effect until explicitly changed. SAVE is typically used with RESTORE where LIST/NOLIST settings are saved before an INCLUDE control switches the input source to another file. RESTORE can be used to restore the settings at the end of the include file or upon returning from the include file.

TITLE(string) shorthand = TT , TIT
default = module name

Enables you to define a title of up to 41 characters in a page header. Unquoted parentheses in "string" must be balanced. String may be quoted if "unusual" characters are used in the title. The length of the title is bound by PAGEWIDTH. If you want the title to appear on the first page, use the TITLE control on the first source line or the command line.

Operational Differences in the Different Modes

The asv20/asv33 operates in one of three modes depending upon the choice of control: MODV20, MODV25, or MODV33.

V20 Mode

The V20 Mode supports the full V20 instruction set, including BRKEM, FPO1, and FPO2. This mode is the default for asv20.

V25 Mode

The V25 Mode supports the V25/V25+/V35/V35+ instruction sets. These instruction sets include all of the V20 instructions except BRKEM. There are also several new instructions that are valid only in the V25 Mode. The ASGNSFR and SETIDB directives are recognized only in this mode. These directives allow the V25 RAM and SFR keywords to be used with the new V25 instructions or the normal V20 instructions.

V33 Mode

The V33 mode supports the full V20 instruction set, with the exception of the BRKEM instruction, as well as the two new V33 instructions, BRKXA and RETXA. This mode is the default for asv33.

There are also 3 separate modes for floating point instructions. These modes are set as part of the initial defaults, but can be changed through the use of the controls MOD087, MOD287, and MOD72291.

8087 Mode

The 8087 mode supports the full Intel 8087 floating point instruction set. This mode also generates NOPs or FWAIT instructions at the start of most instructions in order to allow for synchronization of processors during the execution of the floating point code. This is the default mode for asv20.

80287 Mode

The 80287 mode consists of the same instruction set as the 8087, with the addition of one new operand combination for the FSTSW/FNSTSW instruction. This mode does not generate the NOPs and FWAIT bytes at the start of the instructions. Also, the FDISI, FENI, FNDISI, and FNENI instructions do not generate any output.

These instructions may not be linked into an Intel 8087 emulation library for execution.

Notes

8

Assembler Listing Description

Introduction

This chapter contains a description of a sample assembler listing, including a description of the optional symbol table and cross reference format.

Assembly Listing

The asv20/asv33 Assembler uses a two-pass process. During the first pass, labels, variables and other user-defined symbols are examined and placed in the symbol table. Additionally, structures are stored internally.

During the second pass, the object code is generated, symbolic addresses are resolved, and a listing and object module are produced. Errors detected during the assembly process will be displayed on the output listing with a cumulative error count. At the end of the assembly process a symbol table or a cross reference table can be displayed.

The listing contains information pertaining to the assembled program, including op codes, assembled data and the original source statements. The listing can be used as a documentation tool by including comments and remarks that describe the function of the particular program segment.

A sample assembler listing is provided at the end of this chapter. Refer to the following points to examine and understand the listing.

1. The page headings on this sample show the time and date of the program run.

2. The column titled "Line" contains decimal numbers associated with the listing source lines. These numbers are referred to in the cross reference table.

3. The column titled "Offset" contains a value that represents the first memory address of any object code generated by this statement.

4. The columns under "Object-Bytes" show the object code generated by instructions and directives in the file. Bytes are output lowest address first.

5. To the right of the data bytes are the assembler relocation flags. The flags are ’R’ for relocatable operand, and ’E’ for external operand. If one operand is relocatable and the other is external, the ’E’ flag will be displayed.

6. The original source statements are reproduced to the right of the object-bytes field.

7. At the end of the listing the assembler prints the number of assembler errors. The assembler substitutes NOPs when it cannot translate a particular opcode and therefore provides room for patching the program.

A symbol table or cross reference table can be generated at the end of the assembly listing if the option specifying its output is used. All user-defined symbols, in alphabetic order, along with the symbol’s value type and attributes, are listed in the symbol table.

```
           Hewlett Packard ASV20 HP64873-19002 02.00 06Feb90 Unreleased Copr. HP 1990
Page 1 Fri Feb 16 10:48:23 1990
 SAMPLE           HP64873-19002 02.00 06Feb90 Unreleased Copr. HP 1990
Cmdline - asv20 -L asmexamv20.s
Line Offset Object-Bytes
1     0000                              $MODV20 XREF
2     0000                              ;
3     0000                              ;This small sample program is intended to show
                                         an example list file
4     0000                              ;and to show the some instructions in V20
                                         mode.  For a
5     0000                              ;more complete list of instructions, see the
                                         "Instructions and Operands" chapter.
6     0000                              NAME SAMPLE
7     0000                              DATA SEGMENT WORD PUBLIC 'DATA'
8     0000   10(                        ARRAY DB 10 DUP (0)
8     0000   00 )
9     000A                              DATA ENDS
10    0000
11    0000                              CODE SEGMENT WORD PUBLIC 'CODE'
12    0000                              ASSUME PS:CODE,DS0:DATA,DS1:DATA
13    0000   65 A4                      REPC MOVBKB
14    0002   64 A5                      REPNC MOVBKW
15    0004   0F 31 C1                   INS CL,AL
16    0007   0F 39 C1 0D                INS CL,13
17    000B   0F 33 C1                   EXT CL,AL
18    000E   0F 3B C1 0D                EXT CL,13
19    0012   0F 20                      ADD4S
20    0014   0F 20                      ADD4S DS1:WORD PTR ARRAY, WORD PTR ARRAY
21    0016   0F 22                      SUB4S
22    0018   0F 22                      SUB4S DS1:WORD PTR ARRAY, WORD PTR ARRAY
23    001A   0F 26                      CMP4S
24    001C   0F 26                      CMP4S DS1:WORD PTR ARRAY, WORD PTR ARRAY
25    001E   0F 28 06 00 00     R       ROL4 BYTE PTR ARRAY
26    0023   0F 2A C0                   ROR4 AL
27    0026   0F 10 C0                   TEST1 AL,CL
28    0029   0F 11 C0                   TEST1 AW,CL
29    002C   0F 18 C3 06                TEST1 BL,6
30    0030   0F 19 06 00 00 0D  R       TEST1 WORD PTR ARRAY,13
31    0036   0F 16 C2                   NOT1 DL,CL
32    0039   0F 17 06 00 00     R       NOT1 WORD PTR ARRAY,CL
33    003E   0F 1E C1 06                NOT1 CL,6
34    0042   0F 1F C3 0D                NOT1 BW,13
35    0046   0F 12 C2                   CLR1 DL,CL
36    0049   0F 13 C4                   CLR1 SP,CL
37    004C   0F 1A C2 06                CLR1 DL,6
38    0050   0F 1B 06 00 00 0D  R       CLR1 WORD PTR ARRAY,13
```

Figure 8-1. Sample Assembler Listing

```
39    0056   0F 14 C7                    SET1 BH,CL
40    0059   0F 15 C6                    SET1 IX,CL
41    005C   0F 1C C3 06                 SET1 BL,6
42    0060   0F 1D 06 00 00 0D  R        SET1 WORD PTR ARRAY,13
43    0066   0F FF 7F                    BRKEM 7FH
44    0069   66 2E 00 00        R        FPO2 5,BYTE PTR ARRAY
45    006D   66 2E 00 00        R        FPO2 5,WORD PTR ARRAY
46    0071   66 2E 00 00        R        FPO2 5,DWORD PTR ARRAY
47    0075   66 2E 00 00        R        FPO2 5,QWORD PTR ARRAY
48    0079   66 2E 00 00        R        FPO2 5,TBYTE PTR ARRAY
49    007D                               CODE ENDS
50    0000                               END
```

Figure 8-1. Sample Assembler Listing (Cont'd)

Cross Reference and Symbol Table Format Description

By default, the assembler produces a symbol table at the end of each listing. If you want the assembler to produce a cross reference table in place of the symbol table, use the XREF option.

If SYMBOLS and XREF are both specified, a cross reference table is produced. The cross reference table includes all the information present in the symbol table, but with line references noted for each symbol. The symbol table listing and cross reference features can be turned on only at the beginning of a program, and once on, cannot be turned off at a later point.

The remainder of this section describes parts of the following sample cross reference table listing.

```
            Hewlett Packard ASV20 HP64873-19002 02.00 06Feb90 Unreleased Copr. HP 1990
Page 2 Fri Feb 16 10:48:23 1990
 SAMPLE             HP64873-19002 02.00 06Feb90 Unreleased Copr. HP 1990
                          Cross  Reference

Label     Type     Value                                References
??SEG     SEGM     SIZE=0000 PUBLIC PARA
ARRAY     LOCAL    DATA:0000 BYTE         -8 20 20 22 22 24 24 25 30 32 38 42 44 45 46
                                           47 48
CODE      SEGM     SIZE=007D PUBLIC WORD CLASS 'CODE'  -11 12 -49
CODE      CLASS
DATA      SEGM     SIZE=000A PUBLIC WORD CLASS 'DATA'  -7 -9 12 12
DATA      CLASS


NO ASSEMBLY ERRORS
NO ASSEMBLY WARNINGS
```

Figure 8-2. Cross Reference for Sample Listing

Label In the symbol table or cross reference listing header, the Label field lists the symbol name.

Type The Type field describes the kind of symbol represented by the Label. This field may be any of the following:

SEGM	segment name
GROUP	group name
CLASS	class name
LOCAL	local variable
PUBLIC	public variable
EXTERN	external variable or label
LABEL	local far or near label
STRUC	structure definition
STR_FLD	structure field name
REC	record definition
REC_FLD	record field name
EQU	equate symbol
PROC	procedure name
UNDEF	undefined symbol

Value The Value field appears to the right of the Type field and is used to indicate attributes of the symbol. These attributes further describe what the symbol is or where the symbol resides. The specific attributes shown depend upon the Types above.

SEGM	Size of segment (in bytes), followed by combine type (PUBLIC/MEMORY/STACK/COMMON), followed by alignment (BYTE/WORD/PARA/PAGE/INPAGE/AT nnn), followed by classname, if present.
GROUP	List of segments that belong to the group. If a SEG EXTRN was used, then the name of the external will be displayed.
LOCAL, PUBLIC, EXTERN, LABEL, PROC	Segment name (if known), and offset within segment, followed by type (BYTE/WORD/ DWORD/QWORD/TBYTE/ NEAR/FAR/ABSOLUTE).
STRUC	Size of structure, followed by number of fields.
STR_FLD	Offset within structure, followed by type of field (BYTE/WORD/ DWORD/QWORD/TBYTE).
REC	Size of record, followed by number of fields, followed by width of record in bits.
REC_FLD	Bit offset within record, followed by width of field in bits.
EQU	If EQU'd to a register, the name of the register is shown. If EQU'd to a 17-bit value, NNNN. If EQU'd to a real number, REAL.

If EQU'd to an instruction, INSTRUCTION.

If EQU'd to a memory expression, EXPRESSION.

The UNDEF and CLASS types do not have any attributes.

Cross Reference

If a cross reference is being generated in addition to the symbol listing, then line references will appear to the right of the Value field. Each line reference will be separated from the next by a space.

The line on which the symbol is defined will have a minus sign placed before it. All other line numbers indicate references to the symbol. It is possible for there to be more than one definition of a symbol (for example, a segment). Also, purged symbols may appear more than once in the table.

9

Codemacros

Overview

Codemacros define V20, V25, and V33 instructions. A codemacro is a template for generating code, with certain bits fixed and other bits that are supplied when the codemacro is referenced (much as a record or structure). You must define the codemacro using the CODEMACRO directive before referencing it.

Referencing Codemacros

Formal arguments can be defined on the call line and then referenced in the body of the codemacro. Forward references to codemacros are illegal.

A codemacro is referenced by using its name in the opcode field of a source statement. You must provide actual parameters at this time, which must match the parameters as to the sort of entity described (number, WORD address expression, segment register, etc.). Matching is described in detail below. If matching is successful for all arguments, the codemacro is used to generate code. At this time, the formal arguments in the codemacro body will be replaced with data derived from the corresponding actual parameters.

Multiple codemacros with the same name are legal. When the name is referenced, each of the defined codemacros is checked to determine whether its formal arguments match the actual parameters you provide. The first codemacro whose arguments match is used to generate code. Multiple codemacros are checked in reverse order; the most recently-defined codemacro is checked first. This feature permits a single symbol to generate a variety of different code, depending on the arguments provided. When defined, asv20/asv33 compiles the codemacro into a compact internal form and stores it in virtual memory.

Alphabetical Listing of the Codemacro Directives

The following pages describe the codemacro directives. An alphabetical listing of the directives is provided in Table 9-1.

Table 9-1. Codemacro Directives

Directive	Function
CODEMACRO	Enters Codemacro Definition
ENDM	Terminates Codemacro Definition

Codemacro Directives

CODEMACRO Enters Codemacro Definition

Syntax

```
CODEMACRO cmac_name [formal:specmod[range]][,formal:specmod[range]]...
   ...
ENDM
```

or

```
CODEMACRO cmac_name PREFX
   ...
ENDM
```

Description

cmac_name	The name associated with the defined codemacro. It may have been previously defined as a codemacro, but not as anything else. This name is stored as a symbol and should not conflict with reserved words. Note that using an instruction name in this field is legal and results in an additional codemacro to be searched for that name.
formal	An arbitrary symbol defining a formal argument to the codemacro. Formals are not stored as symbols, and can duplicate keywords or even the cmac_name without conflict. Formals have no existence outside their codemacro and do not appear in the symbol table listing, although two formal parameters to the same codemacro cannot have the same name. A codemacro can have at most 255 formal arguments.

specmod — A letter or pair of letters describing the actual parameters that will match this formal parameter.

The legal values for specmod are:

A	Ab	Aw			
C	Cb	Cd	Cw		
D	Db	Dw			
E	Eb	Ed	Ew		
F					
G					
I					
M	Mb	Md	Mq	Mt	Mw
R	Rb	Rw			
S					
T					
X	Xb	Xd	Xq	Xt	Xw

Upper- and lower-case letters are interchangeable for these values. The convention of one upper-case letter followed by one lower-case letter is used in this chapter for clarity and to avoid confusion with the directives DB and DW. The first letter of the specmod is referred to as the specifier and the second letter as the modifier. The meaning of the various specmods is described in Table 9-2.

range — An optional field that follows a parameter. It describes a range of values that limits the acceptable modules for the parameters matching the formal argument. The first letter of the specmod must be A, D, G, I, R, or S. Any other type of specmod is not permitted to have a range field. The syntax and meaning of range fields is further described later in this section.

PREFX

A keyword that can appear instead of the formal arguments, indicating that the codemacro name cannot take parameters. Instead, it is used to precede another codemacro or instruction name. At the time the codemacro is referenced, an error is detected if another codemacro or instruction does not follow this one.

PREFX is associated with the codemacro name as a whole rather than separately with each codemacro. If one codemacro uses PREFX, another codemacro with the same name must also use PREFX. The last codemacro defined controls in case of conflict. A formal argument cannot be named PREFX.

The CODEMACRO directive lets you enter the codemacro definition mode and specifies the formal arguments associated with the new codemacro. The ENDM is used to terminate the codemacro definition mode. Each CODEMACRO directive must have a corresponding ENDM directive, and codemacro definitions cannot be nested.

Examples

```
CODEMACRO CMAC1 FORMAL1:Ew,FORMAL2:Db(10,20)
CODEMACRO CMAC2 FORMAL3:S
CODEMACRO CMAC3
CODEMACRO CMAC4 PREFX
```

ENDM Terminates Codemacro Definition

Syntax

```
ENDM
```

Description The ENDM directive terminates the codemacro definition mode. Each ENDM must correspond to a CODEMACRO directive. For more information on ENDM, see the description of the CODEMACRO directive in the previous section.

Codemacro Matching

This section describes the algorithm used for matching codemacro references to possibly one of several codemacro definitions. The assembler performs two passes on the input file.

1. During pass 1, all actual parameters are evaluated. Those containing undefined symbols constitute a special kind, called ‘‘forward references,’’ which are treated differently from other expressions. asv20/asv33 is much more liberal concerning what a forward reference can match than what a fully-evaluated expression can match. Forward references are considered to be typeless unless type information is specifically attached with PTR or SHORT.

2. The chain of codemacros corresponding to the instruction mnemonic is searched, beginning with the last one defined. asv20/asv33 looks for a codemacro with the same number of formal arguments as there are actual parameters, such that each actual parameter matches the corresponding formal as far as specmod and range goes. Matching is described in this chapter, in the section called Range Specification. The first codemacro that matches is used as described in #3 below. If none matches, an error is reported.

3. The number of bytes of object code is estimated by executing the codemacro and discarding the generated bytes. This estimate is used to update the location counter. By default, forward references do not require a segment override byte from the SEGFIX, RFIXM, and RNFIXM directives.

4. During pass 2, the codemacro chain search starts at the beginning again. Presumably, all forward references have now been resolved. If not, an error is issued and the absolute number 0 is substituted for the undefined symbol, which may in turn cause other errors. This resolution of forward references can cause a different codemacro to be matched than in pass 1. If none matches, an error is reported. If a codemacro matches in pass 1, it does not necessarily have to match in pass 2.

5. Code is generated using the matched codemacro. A different number of bytes of code can be generated than was called for in the estimate from pass 1. If more code is generated in pass 1 than in pass 2, the extra room allocated is filled with NOPs (90H). If more code is generated in pass 2 than in pass 1, an error message is issued and the entire space allocated is filled with NOPs.

The Specmod Field

The specmod field determines what actual parameters match each formal argument. In Table 9-2, ‘‘variable’’ is an address expression with type BYTE, WORD, DWORD, FDWORD, QWORD, FQWORD, TBYTE, a structure name, or a record name, and ‘‘label’’ is an address expression with type NEAR or FAR. For the purpose of matching, forward references during pass 1 are treated as a special kind of expression that match certain specmods. Specmods match actual parameters as shown in Table 9-2.

Table 9-2. Specmods and Parameter Matches

Specmod	Match
A	AW or AL.
Ab	AL.
Aw	AW.
C	Any label, or any forward reference of type NEAR or FAR or no_type.
Cb	Any NEAR label with the same segment definition attribute as the current assumed contents of PS via ASSUME and within the range -128 to +127 from the beginning of the code macro reference, or any forward reference with SHORT attached.
Cd	Any FAR label, or any forward reference without a type or of type FAR.
CW	Any NEAR label with the same segment definition attribute as the current assumed contents of PS via ASSUME but farther away from the beginning of the codemacro reference than -128 to +127, or any external NEAR label
D	Any 17-bit number, or any forward reference with no type.

Table 9-2. Specmods and Parameter Matches (Cont'd)

Db	Any absolute number between -256 and 255, inclusive, or any number of relocation type high or low
Dw	Any absolute number not between -256 and 255 inclusive, or any number of relocation type offset or base, or any forward reference with no type.
E	Any variable, or any address expression without a type, or any register except segment registers, or any forward reference, except for those typed NEAR
Eb	Any variable with type BYTE
Ed	Any variable with type DWORD or FDWORD, or any forward reference of type DWORD, FDWORD, or no type.
Ew	Any variable with type WORD, or any 16-bit register, except segment registers, or any forward reference of type WORD or notype.
F	The 8087 floating-point stack or any element thereof: ST
G	72291 status registers (FCTW, FPTW, FSTW)
I	V20 registers (R=all registers, PSW=flag word, DIR=direction flag, CY=carry flag). A range is required.
M	Any variable or any address expression without a type, or any forward reference except those of types NEAR
Mb	Any variable with type BYTE, or any forward reference of type BYTE or no type.
Md	Any variable with type DWORD or FDWORD, or any forward reference of type DWORD, FDWORD, or no type.
Mq	Any variable with type QWORD or FQWORD, or any forward reference of type QWORD, FQWORD, or no type.

Table 9-2. Specmods and Parameter Matches (Cont'd)

Mt	Any variable with type TBYTE, or any forward reference of type TBYTE or no type.
Mw	Any variable with type WORD, or any forward reference oftype WORD or no type.
R	Any register except segment registers.
Rb	Any 8-bit register (AH, AL, BH, BL, CH, CL, DH, DL).
Rw	Any 16-bit register except segment registers (AW, BW, CW, DW, SP, BP, IX, IY)
S	Segment registers (DS1, PS, SS, DS0)
T	The 8087 floating-point stack top: ST or ST(0) only.
X	Any variable or any address expression without a type, whose base and index attributes are null or any forward reference except those of types NEAR
Xb	Any variable of type BYTE whose base and index attributes are null, or any forward reference of type BYTE or no type.
Xd	Any variable of type DWORD or FDWORD whose base and index attributes are null
Xq	Any variable of type QWORD or FQWORD whose base and index attributes are null, or any forward reference of type QWORD, FQWORD, or no type.
Xt	Any variable of type TBYTE whose base and index attributes are null, or any forward reference of type TBYTE or no type.
Xw	Any variable of type WORD whose base and index attributes are null, or any forward reference of type WORD or no type.

In addition, typeless address expressions such as [BW] will sometimes match the specmods Eb, Ew, Mb, and Mw. There must be enough information for asv20/asv33 to infer the size of the operation. This condition is met if the codemacro has at least two formal arguments, and one or more of the actual parameters corresponding to the other argument(s) is not either another typeless address expression or a number that matches Db.

For example, suppose a codemacro has ARG1:Ew,ARG2:Ew as the formal arguments. The actual parameters [BW],AW match, since AW implies a WORD operation; however, the actual parameters [BW],[BW] do not match since the information to infer the size of the operation is insufficient. This condition means that any codemacro with a single formal parameter of specmod Eb, etc., cannot match a typeless address expression, including several of the built-in instructions (e.g., INC, FISUB, MUL).

A few built-in instructions (e.g., FLDENV) have the specmod M on their single formal parameter and, therefore, will accept a typeless address expression.

Range Specification

A codemacro range is a parenthesized list of one or two expressions separated by a comma. The syntax of a range specification is:

```
(value1[,value2])
```

Each value must be a register name or an expression evaluating to an absolute number (i.e. not an address). Registers are converted to absolute numbers according to Table 9-3.

Table 9-3. Absolute Number Conversion for Registers

Register	Number
AL, AW, DS1 , CY, FCTW	0
CL, CW, PS , DIR, FPTW	1
DL, DW, SS , PSW, FSTW	2
BL, BW, DS0 , R	3
AH, SP	4
CH, BP	5
DH, IX	6
BH, IY	7

Some codemacros have specific limits on the range of parameters that can be used. This pertains to formals using specifiers A, D, I, R, or S.

When codemacros are referenced, the actual parameter is checked against the specified range, converting actual registers according to Table 9-3. If the range field contained a single value, the actual parameter must match it. If the range field contained two values, the actual parameter must be greater than or equal to the first and less than or equal to the second. Otherwise, the actual parameter does not match. Relocatable actual parameters and forward references never match a formal with a range field.

Examples:

S(0,2)	Matches DS1, PS, or SS.
S(0)	Matches only DS1.
Db(2,-1)	Generates error - invalid range.
Db(-1,2)	Matches -1, 0, 1, or 2. 255 does not match (9-bit comparison).
Db(-1,DL)	Same as previous example.
Rw(DW)	Matches DW.
Rb(CL)	Matches CL.
Db(1)	Matches 1.

Codemacro Matching Examples

Table 9-4 shows a list of the arguments on some example codemacros for the MOV instruction, in the order they are searched, along with actual parameters that will match each. WORDVAR is a variable of type WORD, and BYTEVAR is a variable of type BYTE.

Table 9-4. Arguments and Actual Parameters

Codemacro Reference		Match
MOV WORDVAR,AW		MOV dst:Xw,src:Aw
MOV BYTEVAR,AL		MOV dst:xb,src:Ab
MOV AW,WORDVAR		MOV dst:Aw,src:Xw
MOV AL,BYTEVAR		MOV dst:Ab,src:Xb
MOV SS,WORDVAR		MOV dst:S(SS,DS0),src:Ew
MOV WORDVAR,PS		MOV dst:Ew,src:S
MOV CW,WORDVAR		MOV dst:Rw,src:Ew
MOV CL,BYTEVAR		MOV dst:Rb,src:Eb
MOV DS0:[BW],AW		MOV dst:Ew,src:Rw
MOV DS0:[BW],AL	;16-bit move	MOV dst:Eb,src:Rb
MOV CW,1000	;8-bit mov	MOV dst:Rw,src:Dw
MOV CW,20		MOV dst:Rw,src:Db
MOV CL,20	;16-bit move,0 fill	MOV dst:Rb,,src:Db
MOV WORDVAR,1000		MOV dst:Ew,src:Dw
MOV WORDVAR,20		MOV dst:Ew,src:Db
MOV BYTEVAR,20	;16-bit move, 0 fill	MOV dst:Eb,src:Db

The following is a list of some instructions that do not match the formal argument pairs in Table 9-4.

```
MOV PS,WORDVAR    ; PS is not between SS and DS0,
                  ; and not equal to DS1.
MOV DS1,BYTEVAR   ; No such 8-bit operation appears.
MOV WORDVAR,BL    ; In general, 8-bit and 16-bit operands
                  ; cannot mix.
MOV BL,WORDVAR    ; Mixed 8- and 16-bit operands.
MOV BL,1000       ; Mixed 8- and 16-bit operands. 1000 won't fit in BL.
MOV BYTEVAR,1000  ; Mixed 8- and 16-bit operands. 1000 won't fit in
                  ; BYTEVAR either.
```

Expressions in Codemacros

Only a small subset of the usual expressions is available within codemacro definitions. The following are allowed:

- Absolute numbers, and expressions which evaluate to absolute numbers. No forward references are allowed within such expressions.
- Segment registers.
- Formal argument names.
- Shifted formal arguments.

Syntax:

```
formal_name.recordfield
```

where

```
formal_name and recordfield
```

are symbols. This means to perform a right shift of the actual parameter corresponding to the formal_name at the time the codemacro is referenced, by the number of bits given by the shift count of the recordfield. The actual parameter must be an expression that evaluates to an absolute number. If the actual parameter is a relocatable number, an error is reported at the time the codemacro is referenced. The predefined ESC instruction uses this construct.

The keyword PROCLEN.

This has the value 255 if the most recently defined PROC at the time of codemacro reference was declared FAR. It has the value 0 otherwise. Thus, if the codemacro reference is not in a PROC, PROCLEN yields 0.

Directives within Codemacros

Only a few directives are legal within a codemacro definition, and these are listed below. Instructions are not allowed within a codemacro definition, but assembler controls and comments are; however, the assembler control is not considered part of the codemacro. None of these directives are allowable outside a codemacro definition unless so described elsewhere in this manual (e.g. DB, DW, DD, and record names).

The following pages describe directives within codemacros. An alphabetical listing of these directives is provided in Table 9-5.

Table 9-5. Directives within Codemacros

Directive	Function
DB	Generates byte of immediate data.
DD	Generates 4 bytes of immediate data.
DW	Generates 2 bytes of immediate data.
MODRM	Generates ModRM byte.
NOSEGFIX	Checks for addressability through a certain seg register.
recordname	Generates 1 or 2 bytes using the specified record template.
RELB	Generates 1-byte displacement.
RELW	Generates 2-byte displacement.
RFIX	Generates a WAIT (9BH) followed by the first 5 bits of an ESC(0D8H).
RFIXM	Generates a WAIT (9BH) followed by a segment override byte (if needed) followed by the first 5 bits of an ESC (0D8H).
RNFIX	Generates an NOP (90H) followed by the first 5 bits of an ESC(0D8H).
RNFIXM	Generates a NOP (90H) followed by a segment override byte (if needed) followed by the first 5 bits of an ESC (0D8H).
RWFIX	Generates a WAIT (9BH).
SEGFIX	Generates segment-override byte if needed.

DB, DD, DW Generates N-Bytes of Immediate Data

Syntax

```
DB absolute_numeric_expression
DB formal_name
DB formal_name.recordfield
DD absolute_numeric_expression
DD formal_name
DD formal_name.recordfield
DW absolute_numeric_expression
DW formal_name
DW formal_name.recordfield
```

Description

absolute_numeric _expression	An absolute numeric expression.
formal_name	A name that is a formal parameter to the codemacro.
formal_ name.recordfield	A name that is a formal parameter to a codemacro but shifted according to the recordfield.

The DB, DD, and DW directives are similar to their counterparts outside codemacros, but their legal operands are much more restricted.

Each consecutive appearance of a DB, DW, or DD directive within a codemacro generates one, two, or four bytes, respectively.

It is possible for a formal of specmod Dw to appear in a DB directive, where it will not fit, which will then cause an error at the time of codemacro reference.

A formal_name without a recordfield must be of specifier D for the DB directive and must be of specifier D, C, or X for the DW and DD directives. Specifiers C and X represent labels and variables, respectively; their appearance in a DW or DD has the same effect as described in the chapter Data Definition and Initialization.

A formal_name appearing with a recordfield must have specifier D.

MODRM Generates ModRM Byte

Syntax

```
MODRM formal_name2,formal_name1
```

or

```
MODRM number,formal_name1
```

Description

formal_name1	An effective-address parameter. It must have a specifier of E, M, R, X, A, or S.
formal_name2	A parameter, usually a register. It must have a specifier of D, R, A, or S.
number	An expression evaluating to an absolute number.

MODRM generates the ModRM byte, which can contain a wide variety of information: a register involved in the instruction, the base and index registers of an operand, the addressing mode (direct address, relative to the current location, immediate, register), a continuation of the opcode, etc. Most users do not need the details of this byte's construction. Those interested are referred to Appendix G for the meaning of each bit in this byte for each instruction.

asv20/asv33 derives 5 bits of information from formal_name1 in a highly packed encoded form, and 3 bits from the first parameter. If the first operand of MODRM is a number that is either a constant or a formal matching D, the low 3 bits are used in the generated byte. If the first operand is a register with a matching A, R, or S, the 3 bytes to use are taken from the numeric values corresponding to registers as described in the section on Range Specification.

NOSEGFIX checks for Addressability

Syntax

```
NOSEGFIX segreg , formal_name
```

Description

segreg	One of the segment registers DS1, PS, DS0, SS.
formal_name	A formal argument name whose specifier is E, M, or X (a memory parameter).

NOSEGFIX ensures that a parameter is addressable through a specific segment register. It is used in the built-in instruction set for the string instructions MOVBK, STM, CMPBK, CMPM, INM for which one operand must be addressable through DS1.

NOSEGFIX checks the segment addressability attribute of the actual parameter corresponding to the formal_name to ensure that the parameter is addressable through the specified segment register. If the actual parameter is a register (matching E), it is considered addressable. If the attribute is a segment register, it must match the register on the NOSEGFIX. If the attribute is null, it is not addressable. If the attribute is a segment or group, asv20/asv33 checks the assumed contents of the specified segment register through ASSUME, as it does for SEGFIX. NOSEGFIX never generates any code. It merely performs an error check. Note that this check is not performed at argument matching time. It is possible for the actual parameters to match the formal arguments of a codemacro that contains a NOSEGFIX directive and then get an error on the NOSEGFIX, even if another codemacro exists farther along in the codemacro chain that would not get this error. No codemacro in the built-in instruction set can do this.

Record Name Initialization

Syntax

```
recordname<[expression][,expression]...>
```

Description

recordname — The name of a previously-defined record.

expression — One of the following:

- An expression evaluating to an absolute number
- A formal argument
- A formal argument plus a .recordfield
- Null
- PROCLEN

The record initialization directive lets you control bit fields in codemacro definitions.

Formal arguments in either construct (with or without a .recordfield) must be of specifier D, and the corresponding actual parameter cannot be relocatable or an error will be reported when codemacros are expanded. The resulting byte or word is constructed just as the records described in the chapter Data Definition and Initialization.

Each expression must evaluate to an absolute number, and only the bits corresponding to the defined size of each .recordfield are used. Also, the least significant bits of the expression value are used, and more significant bits are discarded without any check. Null fields, as well as records outside codemacros, result in the use of the default value at the time the record was defined.

RELB, RELW **Generates N-byte Displacement**

Syntax

```
RELB formal_name
RELW formal_name
```

Description

formal_name — The name of a formal parameter to the codemacro with specmod type C.

The RELB and RELW directives generate a one- or two-byte displacement, respectively, denoting the distance from the location of the codemacro reference to a target which can only be a label. The displacement is measured from the location after the bytes generated by RELB or RELW. Specifically, if the target is the byte immediately following the generated displacement whether that is 1 or 2 bytes, the generated displacement will be 1. These directives take one operand, a formal argument that must be of specmod Cb or Cw. RELB and RELW do not concern themselves with segment addressability or the contents of PS.

During codemacro matching to Cb and Cw specmods, the assembler assumes that any RELB or RELW in the codemacro will follow exactly one generated byte and, as a result, the restriction of the displacement for Cb to -126 to +129 occurs. This assumption is correct for all codemacros in the built-in instruction set. You can write codemacros for which this assumption does not hold. For example, you can write one equivalent to several predefined instructions, but if this is done, the wrong match can be made at codemacro reference-time.

RFIX, RFIXM, RNFIX, RNFIXM, RWFIX

Generates WAIT or NOP

Syntax

```
RFIX      formal_or_number
RFIXM     formal_or_number, formal_name
RNFIX     formal_or_number
RNFIXM    formal_or_number, formal_name
RWFIX
```

Description

formal_or_number — A codemacro parameter with specifier type D or an absolute expression that evaluates to an absolute number.

formal_name — A codemacro parameter with specifier type E, M, or X.

These closely-related directives pertain to floating-point instructions. In all modes, they generate bytes as follows:

RFIX — WAIT (9BH) followed by the first word of an ESC (0D8H)

RFIXM — WAIT (9BH) followed by a segment override byte (if needed) followed by the first word of an ESC (0D8H)

RNFIX — NOP (90H) followed by the first word of an ESC (0D8H)

RNFIXM — NOP (90H) followed by a segment override byte (if needed) followed by the first word of an ESC (0D8H)

RWFIX — WAIT (9BH)

RFIX and RNFIX have one operand; RFIXM and RNFIXM have two operands; RWFIX has no operands. The first operand of each, except RWFIX, is either a formal parameter with specifier D or an expression evaluating to an absolute number. The least significant 3 bits of this operand are taken as the last 3 bits of the generated ESC. If the corresponding actual parameter is relocatable, an error is reported when codemacros are referenced.

The second operand of RFIXM and RNFIXM is a formal argument of specifier E, M, or X representing a memory address. The segment override byte is issued or not, depending on this parameter; the algorithm is exactly the same as that described under SEGFIX.

The preceding descriptions assume that the object code will be used on an 8087 chip. These directives are designed for use within floating-point instructions. However, if the linker references the 8087 emulator library instead, the WAIT and NOP instructions described are transmuted into instructions to the emulator. The linker performs this function by resolving external references generated by the R?FIX? directives. This is why, for instance, a codemacro uses RWFIX instead of DB 9BH.

Intel provides two libraries, one of which is used as input to its linker for any given absolute object module. One library is used if the code is destined for an 8087, and the other is used if the 8087 is to be emulated.

This use of built-in external references, which typically will not be of concern to you, also means that any codemacro employing one of these directives displays an E flag (i.e. external reference) on the output listing when referenced. This includes all the floating-point instructions in the built-in instruction set.

SEGFIX Generates Segment-Override Byte

Syntax

```
SEGFIX formal_name
```

Description

formal_name — A codemacro parameter with specifier type E, M, or X.

The SEGFIX directive generates a segment-override byte, if needed (either 26H, 2EH, 36H, or 3EH). This instructs the hardware to use a different segment register for the following instruction.

SEGFIX has one parameter which must be a formal argument name. This argument represents a memory address and, therefore, must have one of the specifiers (1st letter of the specmod) E, M, or X. A register (matching E) never generates a segment override. An address expression has its segment addressability attribute checked as follows:

- If this attribute is null, an error is reported.
- If the attribute is a segment register, that register is used for addressing.
- If the attribute is a group, the assumed contents of the segment registers via ASSUME are checked to see if one of them contains the group.
- If the attribute is a segment, the assumed contents of the segment registers via ASSUME are checked to see if one of them contains the segment or a group containing the segment.

In the last two cases, the segment registers are examined in this order:

1. The register implied by the base and index attributes of the actual parameter (DS0 or SS).
2. The other registers are examined in the order DS1, PS, SS, DS0.

The first register for which the check succeeds is used for addressing. If the actual parameter cannot be addressed through any segment register, an error is issued. Otherwise, once asv20/asv33 has determined which segment register to use for addressing, it determines whether that register is the default implied by the base and index attributes. If so, no override byte is generated; if not, a segment override byte corresponding to the segment register used for addressing is generated.

Notes

10

Macro String Preprocessor Introduction

Introduction

The Macro String Preprocessor (apv20/apv33) is a character string replacement program which performs pre-assembly processing of macros in assembly language source files. It searches the source code for macro calls, and then replaces those calls with the macro return values. The advantage of having the macro string preprocessor is to permit frequently-used segments of code to be used repeatedly by one or several users from a library, without having to re-write the code for each use. You can automatically insert a section of code into the source program by encoding a single line—the macro call.

At definition time, key constructs in the macro may be represented by formal parameters; actual parameters are later substituted for the formal ones. apv20/apv33 handles conditional assembly, assembly-time loops, and is also capable of recursion.

Note

The macro preprocessor is case sensitive by default. Upper and lower case characters are not equivalent to the preprocessor. The macro symbol MACSYM would not be the same as macSYM, MaCSYM, or macsym. Case sensitivity can, however, be turned off on the command line.

---

apv20/apv33 is implemented as a program separate from the assembler, thereby saving time for those who do not use macros. It is compatible with the NEC syntax for the V20, V25, and V33 macro languages. If you use macros in the source code, you must run the Macro Preprocessor to produce an output file for input to the assembler.

The two Macro Preprocessors, apv20 and apv33, are linked together, and perform the same operations. The two versions are provided for tool orthogonality.

Input Source Characteristics

apv20/apv33 views its input file as a stream of characters instead of a sequence of statements. All processing is character-oriented. The ends of lines are treated as if they ended with a <line feed>. This character is called 'end-of-line' or '<EOL>' in text that follows.

The Metacharacter '%' And The Call Pattern

The macro preprocessor searches the input source one character at a time, looking for a special character called the **metacharacter**. By default, this character is the percent sign ('%'), but it can be dynamically changed. Until the metacharacter is found, characters are passed to the output file without change. When the metacharacter is found, the macro preprocessor reads and interprets the characters following it, isolating a **call pattern**. The call pattern is interpreted as instructions to the macro preprocessor and is not passed to the output file. However, the macro preprocessor produces an expansion of the call pattern that is written to the output file in place of the call pattern. The call pattern can contain other metacharacters followed by call patterns; these will also be expanded. Expansions are stacked, analogous to nested subroutines. When the current expansion is complete, the stack is popped, and the next higher expansion resumes where it left off. The expansion of a call pattern is always a string of characters which can be null (zero characters) in some cases, but most often it is one or more characters. When the outermost expansion is completed, the macro preprocessor goes back to copying characters while scanning for the metacharacter.

The source code below has statements that contain macros.

```
NOP
asymbl EQU 2
DB %LEN( %SUBSTR(5 DUP (0),1,1)) ;note blank before
                                 ;%SUBSTR
ADD AW,2
```

The example source code is treated by the macro preprocessor in this way:

1. Everything up to the first "%" is passed to the output unchanged. The text has no significance to the macro preprocessor.

2. The first "%" invokes the pre-defined macro function LEN, which counts the characters in its argument. (LEN, SUBSTR, and other pre-defined macro functions used in these examples are described in detail in the chapter called "Pre-defined Macro Functions.")

   Everything up to but not including the balancing right parenthesis (in this example, the last parenthesis) is the argument to LEN.

3. The argument to LEN contains a call to another pre-defined macro function, SUBSTR, which extracts a substring from its first argument according to parameters in the second and third arguments. The expansion of the outer function LEN therefore pauses while SUBSTR is evaluated.

4. In this example, the result of SUBSTR is the single character '5'. After the evaluation, LEN resumes, in effect evaluating "%LEN(5)" (again, notice the space in front of the 5). This produces the string "02H," which is passed to the output.

The space between "%LEN(" and "%SUBSTR" is a significant part of the LEN argument, but is not part of the call to SUBSTR. Following "02H," apv20/apv33 puts out the <EOL>, which is the next character following the call pattern of LEN in the source file. Notice that <EOL> is not part of the call pattern. The assembler, therefore, sees the following line of text:

```
DB 02H<EOL>
```

Metacharacter Syntax

The metacharacter can be followed by

- a symbol
- a left parenthesis
- an apostrophe
- a decimal digit
- an asterisk (called the literal character), that in turn must be followed by a symbol.

Any other characters are not acceptable, particularly spaces and tabs. A symbol following the metacharacter (or the metacharacter-asterisk pair) must be one of three things:

- A pre-defined macro function.
- A call to a previously-defined user macro.
- A reference to a previously-defined macro-expansion-time symbol or, within a macro body, a formal argument or local symbol. The metacharacter is recognized anywhere in the source text, including within character strings.

Getting a line such as

```
DB  '20% inflation'
```

to pass through the macro preprocessor requires special handling. Getting these strings through the macro preprocessor is discussed in the "%n and %((Escape and Bracket Functions) in the chapter titled "Pre-defined Macro Functions."

Literal Character *

The literal character (*) specifies that metacharacters contained in the arguments to a function are not expanded. The literal character is placed between the metacharacter and the function or macro name, and spaces or other separators cannot precede or follow it. The literal character inhibits the expansion of all user macros, symbols, and pre-defined functions. It does not affect formal macro parameters, local symbols within macros, and the escape, comment and bracket functions. If one of the lines of code from the previous example were rewritten to contain the literal character before the LEN macro name,

```
DB  %*LEN(%SUBSTR(5 DUP (0),1,1))
```

then the SUBSTR call is not expanded. Instead, LEN counts the length of the string '%SUBSTR(5 DUP (0),1,1)' and returns the string "16H." Output to the assembler would then be

```
DB  16H <EOL>
```

If the literal character preceded SUBSTR instead of LEN, it would have no effect in this example because the argument to SUBSTR does not contain any metacharacters. Misuse of the literal character causes the macro preprocessor to pass strings containing a metacharacter on to the assembler, where they will usually be flagged as errors. The literal character is prohibited all together with some functions; other functions accept it, but ignore it. The literal character should almost always be used when defining a user-macro.

Input Parsing

The macro preprocessor recognizes only those keywords specifically mentioned in the "Expressions and Operators" section of the chapter titled "Elements of Macro Expressions." The macro preprocessor only understands symbols in specific constructs which are usually preceded by the metacharacter. Assembly-time user-defined symbols (labels, etc.), the location counter, and EQUs are all unknown to the macro preprocessor.

You must be careful that a macro call produces each <EOL> in the right place. Readable input to the macro preprocessor frequently results in a large number of output lines consisting only of blanks and end-of-lines. For user convenience and assembler speed, such lines are always omitted from the output. To create a blank line, deliberately use a blank comment line.

Output Buffering

The macro preprocessor buffers its output in an array that can hold 256 characters. When its buffer is full and another character (other than <EOL>) is received, apv20/apv33 breaks the output line into two pieces. The break occurs at the 256 character boundary and the remaining text is placed on the next line of output. This and all other lines created from the long input line will begin with a '&' so the assembler can recognize the line as a continuation. Since the break is made at a fixed location, it is likely that the result will cause a syntax error in the assembler. Thus, it is best if line lengths are restricted to less than 256 characters.

Include Files

INCLUDE is an assembler control command, but the macro preprocessor will act on INCLUDE also. INCLUDE statements cause the macro preprocessor to temporarily stop reading source statements from the current file. It begins reading source statements from the file specified by the INCLUDE. It continues reading from the include file until it finds the end-of-file for the include file or it finds another INCLUDE. When the preprocessor resolves all INCLUDEs and does find the end-of-file for the include file, it then returns to the file that contained the INCLUDE statement and again begins reading source statements immediately after the INCLUDE statement.

Note

The maximum depth that the macro preprocessor can handle nested INCLUDE controls is to a level of eight. The restriction on the assembler depends only upon the number of open files the operating system allows at one time.

Note

Include allows you to specify a different directory than the current working directory for include files by allowing a path name with the file name. However, if the file in the other path also has an INCLUDE control and that control does not have a path name as part of the file name, then the macro preprocessor will look back into the current working directory for the new include file. That means that if an include file in another directory needs another file to be complete, do not expect apv20/apv33 to pick this file up from that other directory along with the first include file.

The syntax for the INCLUDE statement:

```
$INCLUDE(filename)
```

The '$' must be in column 1 for the preprocessor to recognize it for processing.

Any INCLUDE starting in column 1 of a source statement, whether from a source file or an include file, is processed by the macro preprocessor when it is first read. An INCLUDE within a macro definition can be processed at assembly-time or at macro-expansion-time, depending on whether the '$' starts in column 1 in the definition. If an INCLUDE does have a '$' in column 1 in the definition, then it is expanded at definition time. Otherwise, INCLUDE is not processed at macro-expansion-time. Example:

```
%*DEFINE(MAC1) ($INCLUDE(filename))      ;assembly-time
%*DEFINE(MAC2) (
$INCLUDE(filename)                       ;macro-definition time
 )
%*DEFINE(MAC3(PARM1)) ($INCLUDE(%PARM1)) ;assembly-time
%*DEFINE(MAC4(PARM1)) (
 $INCLUDE(%PARM1)                        ;macro-definition time.
 )
```

Since %PARM1 is an improper filename, this causes an error.

However, expansions of MAC4 will be the expected:

```
$INCLUDE(value-of-%parm1-at-expansion-time)
```

This is the same as MAC3, but MAC3 does not produce an error message.

Any $INCLUDE processed at macro-expansion-time causes the remainder of its source line to be lost. If an error is detected while processing an INCLUDE, the error message is placed in the output file as usual and the line containing the INCLUDE is handled as ordinary text. If INCLUDE is misspelled or if the following left parenthesis is missing, no macro-expansion-time error is reported; the string is passed intact to the assembler.

11

Elements Of Macro Expressions

Introduction

This chapter discusses the basic elements of macro expressions. The discussion describes the accepted character set for the macro preprocessor, how numbers are handled, and how symbols are formed.

Some of the arithmetic and logical operators used by the assembler may also be used in macro expressions. This chapter lists and briefly describes them, but does not go into the detail that the earlier chapter titled "Expressions" does. Refer to that chapter for more specific information about these operators.

Macro expressions appear in some of the pre-defined instructions and are particularly important to the %SET macro function.

Character Set

The macro preprocessor recognizes the characters listed in the following table.

Table 11-1. Macro Preprocessor Character Set

Alphabetic Characters

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
```

Numeric Characters

```
0 1 2 3 4 5 6 7 8 9
```

Special Characters

```
   blank                   horizontal tab          >  greater than
$  dollar sign             <  less than            *  asterisk
'  single quote            (  left parenthesis     ,  comma
)  right parenthesis       +  plus sign            @  commercial at
-  minus sign              .  period               &  ampersand
:  colon                   !  exclamation point    ;  semicolon
"  double quote            =  equal sign           #  sharp
?  question mark           %  percent              _  underscore
[  left bracket            ]  right bracket        \  back slash
`  accent grave           {  left brace           }  right brace
|  vertical bar           ~  tilde                ^  caret (uparrow)
/  slash
```

Numbers

Numbers are stored in 17-bit form with a range of -65535 to +65535. Note that the sign bit is stored, therefore -1 is not the same as +65535 for purposes of macro-time operations (although they can be the same to the assembler). Integer constants in bases other than decimal are defined by placing a coded descriptor after the integer. The descriptors are as follows:

- B - binary
- O - octal
- Q - octal
- D - decimal (default)
- H - hexadecimal

Symbols

Symbols must begin with a letter or one of two special characters: the question mark ('?'), or the underscore ('_'). The second and following characters can be any letter, digit, question mark, or underscore. Only the first 31 characters of a symbol are used by the macro processor to define that symbol; any additional characters are only for documentation purposes.

Note

By default, the macro preprocessor is case sensitive. That means that upper and lower case letters are not equivalent in macro symbols. "ASYMBOL," according to the default, is not equivalent to "asymbol" or "ASYmBOL." Case sensitivity, however, can be turned off on the command line.

A macro symbol must be preceded by the metacharacter or the macro preprocessor will treat it as ordinary text. The exception is a string argument to a specific macro function.

The macro preprocessor does not recognize forward references because it makes only one pass through the source. Any symbol must be defined before it is used. Keywords are stored separately from symbols. Symbol names can therefore duplicate operator keyword names without conflict.

Macro symbols always have a string as a value. If the string happens to represent a valid numeric constant (such as '01Q' or '2'), the symbol can be used as the operand of an expression. Only macro-time symbols and 17-bit integer constants are valid macro expression operands. The macro preprocessor does not deal with relocatable numbers of any sort.

Balanced Text String (baltex)

A frequently-referenced concept is the balanced-text string ('baltex'), which is a string of characters containing balanced parentheses. Formally, baltex either contains no parentheses, or one or more sets of balanced parenthesis, as in

```
'baltex(baltex)baltex'
```

where each baltex is a balanced-text string (possibly null).

Expressions and Operators

Expressions consist of one or more operands, and zero or more operators. The recognized operator keywords and their relative precedence are in the following table: (Operators that appear on the same line in the table have the same relative precedence.)

Table 11-2. Operator Precedence

Precedence	Operators
Higher	HIGH, LOW
	*, /, MOD, SHR, SHL
↑	Unary and Binary +, -
	EQ, NE, LT, LE, GT, GE
↓	NOT
	AND
Lower	OR, XOR

Parentheses can be used to override the default precedence of these operators and are recommended for complex expressions.

HIGH, LOW

The HIGH and LOW operators extract the respective high or low byte from a word value. The sign bit is ignored and the twos complement form of negative numbers is used. Examples:

```
HIGH 1234H     ;yields 12H
HIGH -2        ;yields 0FFH
HIGH 65534     ;yields 0FFH
```

The following identities apply:

```
HIGH (HIGH x) = 0
LOW (LOW x) = LOW x
HIGH (LOW x) = 0
LOW (HIGH x) = HIGH x
```

NOT

The NOT operator produces the bitwise (ones) complement of the operand as a 17-bit operation. Since the bitwise complement of 0FFFFH is 10000H (-65536) (which is not a valid 17-bit value), NOT 0FFFFH is defined to be 0.

Add (+), Subtract (—)

The + and — operators can work on either a single operand (unary) or on two operands (binary). A unary operation is equivalent to 0 plus or minus the operand. Example:

```
-2
555+07FFH
```

Multiply (*), Divide (/), MOD

Multiply, divide and MOD accept 17-bit operands and return 17-bit results. The MOD operator yields its left operand modulo its right operand (the remainder after a division).

SHL, SHR

The logical shift operators are full 17-bit shifts including the sign bit, and work on the twos complement form of negative numbers. They shift their left operand the number of bits given by their right operand, using zero fill. For example, -2 SHR 2 is 7FFFH or +32767. It is possible for a shift to produce the invalid 17-bit number -65536 (10000H), which is automatically converted to 0.

If the count is negative, the shift is performed in the opposite direction. If the magnitude of the count is greater than 16, the result is 0.

AND, OR, XOR

The logical operators AND, OR, and XOR perform the indicated bitwise logical operation. They are full 17-bit operations, including the sign bit, and work on the twos complement form of negative numbers. Example:

```
-2 AND 32767   ;yields 7FFEH (32766)
-2 OR 32767    ;yields 1FFFFH (-1)
-2 XOR 32767   ;yields 18001H (-32767)
```

The logical operators can produce a result of 10000H (-65536), which is converted to 0.

EQ, LE, LT, GE, GT, NE

The relational operators, EQ, LE, LT, GE, GT, and NE compare their operands and return an absolute number. The return value 0 if the comparison is false or 0FFFFH (+65535) if it is true. Example:

```
3 EQ 0          ;yields 0 (false)
-2 EQ 0FFFEH    ;yields 0 (false - 17 bit comparison)
```

Notes

12

Pre-Defined Macro Functions

Introduction

This chapter provides a description of the pre-defined macro functions found in apv20/apv33.

Pre-defined macro functions are provided as building blocks so that you may create user-defined macros. It would be nearly impossible to duplicate many useful operations found in the pre-defined functions with equivalent user-defined macros.

Note

A user-defined macro may be re-defined in the source program at some point after the original user definition. Redefinition does not cause errors; it does cause the preceding macro definition to be lost. Pre-defined macro functions, however, may not be re-defined. It is an error to try to do so.

Pre-Defined Macro Functions

The pre-defined macro functions listed below are recognized by the macro preprocessor.

Note

The pre-defined macro functions %IN, %OUT, %CI and %CO are not supported by the apv20/apv33 macro preprocessor. These functions accept user input to macro functions.
The pre-defined macro function %DEFINE does not appear in this chapter because it is discussed in detail in the following chapter titled "User-Defined Macros." Discussion of %DEFINE is appropriate to that chapter because %DEFINE is used to create user-defined macros.

Table 12-1. Predefined Macro Functions

%’ (comment function)	%((bracket function)
%n (escape function)	%DEFINE
%EQS	%GES
%GTS	%LES
%LTS	%NES
%EVAL	%EXIT
%IF	%LEN
%MATCH	%METACHAR
%REPEAT	%SET
%SUBSTR	%WHILE

%’ (Comment Function)

Call Pattern:

```
%’ ...any text... ’ or end-of-line
```

Description: The comment function permits insertion of comments without being passed on to the assembler. Everything from the quote up to a matching closing quote or to an end-of-line is considered a comment. Metacharacters within the comment string are not expanded. In the output, the call pattern (*including the closing end-of-line*, if used) is replaced with the null string.

Example:

```
MOV AW,%ARG1      %' ARG1 is the loop counter'
MOV IX,0          %' Initialize index register
BR $-2
%SET(symbol,02H)  %' Initialize: %SET(symbol,03H)'
DB %symbol
```

The second line in this example will result in an assembly-time error because the end-of-line terminating the comment is removed along with the comment, so the assembler sees the two instructions

```
MOV IX,0  BR $-2
```

without an end-of-line between them. The fourth line shows that metacharacters inside a comment are not expanded; the last line expands to 'DB 02H' because the '%SET' was not executed within the comment. The literal character ('*') cannot be used with the comment function.

%n and %((Escape and Bracket Functions)

Call Pattern:

```
escape function:  %n[n-characters]
bracket function: %(baltex)
```

Escape Function

Description: n is a decimal (base-10) digit from 0 to 9 inclusive. The expanded value of the escape function pattern is the n-characters immediately following n itself. These will be passed to the assembler without being examined by the macro preprocessor. For example, '%1%' passes a '%' to the output. The pattern '%0' passes no characters.

Bracket Function

Description: The expanded value of the bracket function is the "baltex" that appears between the parenthesis. The bracket function inhibits the expansion of all macros and functions within its argument *except* the escape function, the comment function, and macro parameters. These are always expanded.

Escape and Bracket Functions (Generally)

Description: It is sometimes necessary to hide certain text from the macro preprocessor, such as when a percent sign (%) is desired in the output or when using strings involving unbalanced parentheses or commas as text. The escape and bracket functions serve this need.

The bracket function might be more flexible than the escape function, but it deals only with baltex, and the metacharacter is interpreted (although once a call pattern has been detected it cannot be expanded). Examples:

```
%(1,2,3)      ;1,2,3 is passed to the output (this might
              ;be used as the actual parameter to a
              ;macro to prevent the commas from being
              ;interpreted as delimiters)
%330%         ;30% is passed to the output
%(30%)        ;error – '%)' is not legal
%(%330%)      ;%330% is evaluated, then used as
              ;an argument to %()
%(30%1%)      ;same
%(%(30))      ;%(30) is passed to the output
DB '30%1%'    ;DB '30%' is passed to the output because
              ;quotes are ignored by preprocessor
```

The literal character ('*') is not accepted with the bracket or escape functions.

%EQS, %NES, %LTS, %LES, %GTS,%GES

Call Pattern:

```
%xxS(baltex1,baltex2)
```

In the above call pattern, xx represents the first two characters of any of the function names.

Description: The string relational functions all compare two strings, character by character, left to right, and expand to a *logical-valued string*: –1H for TRUE, and 00H for FALSE.

The first string cannot contain a comma unless the comma is protected by parentheses, the escape function, or the bracket function.

Comparison is on the basis of ASCII character values. A blank character has the value 20H, tab has the value 09H, and <EOL> has the value 0AH (<line feed>). The comparison is *true* if the first argument has the relationship to the second indicated by the function. (EQS is true if the two strings are equal. GTS is true if the first string is "greater" than or equal to the second string.)

If two strings are of different lengths, but are identical on all characters in the shorter string, the longer string is considered to be greater.

The literal character ***** is allowed, but it has no effect. Metacharacters in the argument strings are always expanded. Example:

```
%EQS(0,00H)   ;yields 00H (false), since comparison is
              ;of strings, not numeric values
%GTS(2,100H)  ;yields -1H same reason as above
%GTS(c,CBA)   ;yields -1H (true), since c>C (ASCII
              ;values), which ends comparison
```

%EVAL

Call Pattern:

```
%EVAL(expression)
```

Description: EVAL is used to evaluate an expression and it expands to a string representing the numeric value of the expression. The expanded string represents the value in hexadecimal. The first character of the expanded string is always a digit 0-9, the last character is always 'H', and the characters between are the hexadecimal digits 0-F. The expression is evaluated using 17-bit arithmetic, as always, but the expanded value is at most 16-bits. Negative numbers are shown in twos complement form. The expanded string can be 3, 4, 5 or 6 characters in length. Examples:

```
%EVAL(3+3)     ;yields 06H
%EVAL(3-3)     ;yields 00H
%EVAL(-2)      ;yields 0FFFEH

%SET(S1,44)    ;null (decimal value)
%SET(S2,333Q)  ;null (octal value)
%EVAL(%S1+%S2) ;yields 0107H
```

The call pattern %*EVAL is legal, but the literal character ('*') has no effect; metacharacters in the expression are always expanded.

%EXIT

Call Pattern:

```
%EXIT
```

Description: The EXIT function allows immediate exit from the most recently invoked %REPEAT, %WHILE, or a user-defined macro. The call pattern %EXIT has no argument; it ends with the character 'T'. Some common uses are to prevent a WHILE loop from going on forever and to allow multiple exit points from a user macro.

This macro illustrates the classic example of recursion, the factorial function:

```
%*DEFINE(FACTORIAL(X))
(%IF(%X LE 1) THEN (01H %EXIT) FI %EVAL((%X)*%FACTORIAL(%X-1)) )
```

The same result could also be accomplished by using %ELSE instead of %EXIT. In this simple case using an %ELSE might even be clearer, but in more complex examples the %IFs might be nested several levels deep, so %EXIT would be much easier.

The call pattern %*EXIT is legal, but the literal character ('*') has no effect.

%IF (Conditional Assembly Function)

Call Pattern:

```
%IF(expression) THEN (baltex1) [ELSE
(baltex2)] FI
```

Description: The IF function enables a user to decide at macro-time whether to assemble certain code or not. Doing this at macro-time has the advantage that the assembler (which may require more execution time than the macro preprocessor) sees only that code that is to be assembled.

The expanded value of %IF is the expanded value of either baltex1 or baltex2 (if present), but not both. The call pattern %IF first evaluates the *numeric* expression. If the low bit of the 17-bit value is 1, then the expression is considered true. Baltex1 is passed to the output as the expanded value of %IF. If the low bit of the 17-bit value is 0, then the expression is considered false and baltex2 becomes the expanded value of %IF (if baltex2 is present). If it is not present, the expanded value of %IF is null.

Typically, the expression will contain comparison operators (EQ, and so forth) or string comparison macro functions (%EQS, and so forth). These always return –1 for true and 0 for false, so %IF does what you would expect. However, any numeric value is acceptable.

The baltex that is not selected is also not expanded. Any %SETs in it, for instance, will not be executed.

The keywords THEN, ELSE, and FI are not stored as symbols, and user symbols can duplicate these names. Since the arguments are all baltex with parentheses as delimiters, there is no problem with ambiguity.

Call patterns (%IFs) can be nested; each FI (and ELSE, if present) is considered to go with the most recently defined IF. Example:

```
%*DEFINE(MAC(symbol))  (
%IF (%symbol LT 0)
THEN (  %'goes with LT if'
     DB 00H
) ELSE (   %'goes with LT if'
     %IF (%symbol GT 10)
     THEN (
          %set(newsymbol,%symbol-10)
          DB %newsymbol
     ) ELSE (   %'goes with GT if'
          DB %symbol
     ) FI   %'goes with GT if'
) FI   %'goes with LT if'
)
```

The literal character ('*') is legal with %IF and has the effect of suppressing metacharacter expansion in whichever baltex is selected to become the output. Metacharacters in the expression are always expanded.

%LEN

Call Pattern:

```
%LEN(baltex)
```

Description: The LEN function counts the characters in its argument and expands to a string representing the numeric value of the expression. The expanded string represents the value in hexadecimal. The first character of the expanded string is always a digit 0-9, the last character is always 'H', and the characters between are the hexadecimal digits 0-F. The expression is evaluated using 17-bit arithmetic, as always, but the expanded value is at most 16-bits. Negative numbers are shown in twos complement form. The expanded

string can be 3, 4, 5 or 6 characters. The literal character ('*') is legal and prevents the expansion of metacharacters in the baltex string. Example:

```
%LEN(countme)        ;yields 07H
%LEN(%EQS(ABC,abc))  ;depends on case sensitivity
%*LEN(%EQS(ABC,abc)) ;counts '%EQS(ABC,abc)'
                     ;and yields 0DH
%LEN() ;yields 00H
```

An <EOL> counts as one character (the line feed character). %LEN of a SET-symbol will produce a number between 3 and 7 inclusive. It is the number of characters of the internal string representation of the symbol value.

Note

The value is a full 17-bits, with a minus sign if needed (signed magnitude representation). Thus –2 is stored as '-02H' and 65534 is stored as '0FFFEH'. This is the only time (within a %LEN) that the value of a SET-symbol is not really stored as a number.

%MATCH

Call Pattern:

```
%MATCH(name1 delimiter name2)
(string)
```

Note

The spaces surrounding the delimiter in the syntax above are not a part of the call pattern; they are shown only for clarity. Spaces between the first and second pair of parentheses are acceptable. Spaces, tabs, or end-of-lines are skipped over if they appear there.

Description: Name1 and name2 are symbols (not necessarily previously defined) and delimiter is a single character separating them.

It can be any character that is *not* valid in symbols. It could be a space, tab, comma, end-of-line, parenthesis, or others.

MATCH divides a string into two parts at the first occurrence of the delimiter, and assigns each part to a symbol. Its expansion is the null string. MATCH is most commonly used in connection with loops, as described below.

MATCH searches the (expanded) string for the first occurrence of the delimiter. When it is found, all characters in the string preceding the delimiter are assigned as the value of name1. All characters following the delimiter are assigned as the value of name2. Either value can be null. If the delimiter is not present in string, the entire string is assigned to name1 and name2 receives the null string as its value. Examples:

```
%MATCH(NAME1,NAME2) (A,B,C)       ;NAME1='A', NAME2='B,C'
%MATCH(NAME1 NAME2) (A,B,C)       ;NAME1='A,B,C', NAME2=null
%MATCH(NAME1 , NAME2) (A,B,C)     ;Error – illegal spaces
                                  ;around comma (delimiter in this example)
```

The literal character ('*') is legal in conjunction with %MATCH and inhibits the expansion of any metacharacters in "string." Example:

```
%SET(sym,2)
%MATCH(VAR1,VAR2) (%sym,02H)  ;VAR1=02H, VAR2=02H
%*MATCH(VAR3,VAR4)(%SYM,02H)  ;VAR3=%SYM, VAR4=02H
%SET(SYM,3)
DB %VAR1                      ;yields DB 02H in the output
DB %VAR3                      ;yields DB 03H and %SYM is
                              ;expanded at reference time
DB %*VAR3                     ;yields DB %SYM and causes an
                              ;assembly-time error
```

The last example is case dependent and would not work if case sensitivity was not turned off.

The MATCH function is often used to extract similar fields out of a string one at a time. Suppose a string consists of several numbers separated by spaces. Such a string might be the expected value of a formal argument, for instance. To generate a DB for each number:

```
%MATCH(TEMPVAR^JUNK) (%FORMALARG)
%WHILE( %LEN(%TEMPVAR) GT 0 )
(%MATCH(NEXTNUM TEMPVAR) (%TEMPVAR)
```

```
DB %NEXTNUM
)
```

The first MATCH copies the formal argument to TEMPVAR, presuming there are no carets (^) in %FORMALARG (this is a trick to evade the fact that SET can assign only numeric values to a symbol; it cannot assign a string). The condition of the WHILE loop states that TEMPVAR must still be non-null. The MATCH inside the loop extracts the next number from TEMPVAR and stores the rest of the string back in TEMPVAR. The DB is then generated and we execute the WHILE test again.

%METACHAR

Call Pattern:

```
%METACHAR(baltex)
```

Description: The METACHAR function changes the metacharacter (% by default) to a different, user-specified character. These are the acceptable alternative metacharacters:

```
@ / + - # . __ = [ ] < > ! ’ " $ & , = % { } ~ ‘ | \ ^
```

The following characters cannot be used as a metacharacter:

```
the letters (A-Z, a-z)
the digits (0-9)
 _ ? * ( ) blank tab <EOL>
```

The new metacharacter is taken to be the first character of the expanded value of baltex, although baltex can be any number of characters long. The new metacharacter takes effect immediately at the first character following the right parenthesis delimiting the call pattern of METACHAR. The literal character (’*’) is accepted on METACHAR, but it has no effect, as the argument of METACHAR is always expanded.

Changing the metacharacter can have unforeseen catastrophic effects. For example, any previously defined macros probably have the default metacharacter (’%’) in the stored macro body. They will not expand correctly if the metacharacter changes. The expanded value of the METACHAR function is the null string.

%REPEAT

Call Pattern:

```
%REPEAT (expression) (baltex)
```

Description: The REPEAT function is one way to program a loop. REPEAT evaluates the 17-bit numeric expression and then baltex is expanded that many times. Note that the expression is expanded only once. If baltex alters macro symbols that are involved in the expression, it does not affect loop control. If the expression evaluates to be less than or equal to zero, baltex is expanded zero times (the expanded value of REPEAT is the null string). Example:

```
%REPEAT(5)      (SHL AW,1
)
```

Note

The <EOL> within baltex is necessary for correct expansion. Without the <EOL>; this REPEAT would produce

SHL AW,1SHL AW,1SHL AW,1SHL AW,1SHL AW,1

%*REPEAT is acceptable. The asterisk inhibits the expansion of metacharacters within baltex. Metacharacters in 'expression' are always expanded.

%SET

Call Pattern:

```
%SET(name,expression)
```

Description: SET defines the string "name" as a symbol, whether or not it was already defined, and gives it the value of "expression." Expression must result in a number, but the value of name is stored as a string (like all macro symbols). Generally, you can ignore this fact and treat name as if it were stored as a number. Multiple SET directives can reference the same name. The expanded value of the %SET call pattern is the null string.

The literal character ('*') makes no sense with SET, since its first argument must be a symbol and its second argument must evaluate to a number. Neither argument can contain metacharacters after expansion. If the macro preprocessor attempts to expand %*SET, it will report an error.

It is correct for the symbol-referencing construct to appear inside another SET for the same symbol. Example:

```
%SET(username,%username+1)
```

This increments the value of 'username' by one. However, the next example is incorrect:

```
%SET(username,username+1)
```

This example generates a macro-time error because the character string "username" is not a legal expression operand. Symbol-referencing is discussed in the chapter titled "User-Defined Macros."

%SUBSTR

Call Pattern:

```
%SUBSTR(baltex,exp1,exp2)
```

Description: The SUBSTR function extracts a substring from its first argument based on its second and third arguments.

In this pattern, exp1 and exp2 are numeric expressions. The expanded value of the pattern is a substring of baltex. The substring begins at character number exp1 and contains exp2 characters. If exp1 is less than or equal to 0, or greater than the number of characters in baltex, then the expanded value is null. If exp2 is less than or equal to 0, then the expanded value is null. If exp1 is of such a size that the expansion value will not be null, but exp2 implies more characters than remain in baltex, then the expanded value is all characters from character exp1 to the end of baltex, inclusive. Examples:

```
%SUBSTR(12345678,4,2)      ;yields   45
%SUBSTR(12345678,-1,2)     ;yields   null
%SUBSTR(12345678,10,2)     ;yields   null
%SUBSTR(12345678,2,-1)     ;yields   null
%SUBSTR(12345678,2,1000)   ;yields   2345678
```

The literal character ('*') is accepted with SUBSTR, but is ignored. Metacharacters in any of the arguments are always expanded.

%WHILE

Call Pattern:

%WHILE (expression) (baltex)

Description: The WHILE function programs macro-time loops. It works similarly to the WHILE construct in high level languages.

WHILE evaluates the 17-bit numeric expression each time through the loop. If the least significant bit of the expression is 0, the expanded value of WHILE is the null string. If the least significant bit of the expression is 1, then baltex is expanded and passed on as part of the expanded value of WHILE, and the expression is evaluated again. The

loop continues until the expression evaluates to false (least significant bit is 0).

For the loop to terminate, baltex must modify the value of expression or an EXIT function must be used. Otherwise the loop will never exit. WHILE is often used in conjunction with either SET or MATCH, either of which will update a macro symbol on each pass through the loop (see the example under MATCH).

The call pattern %*WHILE is not accepted, since preventing the expansion of baltex would result in an infinite loop. An error will be reported if %*WHILE is found.

Example Problem

This example shows the effects of an incorrect factorial macro.

```
%*DEFINE(FACTORIAL(X))
(%IF(%X LE 1) THEN (01H %EXIT) FI
%EVAL(%X*%FACTORIAL(%X-1))
)
```

The only difference between this example and the one shown with the %EXIT function reference is that this one is missing the pair of parentheses around the second %X. They are necessary, because the arguments of macros are strings, not numbers. The incorrect version above called with the actual parameter 4 expands successively to the following:

```
4*FACTORIAL(4-1)
  4-1*FACTORIAL(4-1-1)
    4-1-1*FACTORIAL(4-1-1-1)
      01H
     4-1-1*01H
    02H
   4-1*02H
  02H
 4*02H
08H
```

The %FACTORIAL in the next lower calling level is evaluated before the %EVAL in the one that called it is executed. That is as it should be

and the recursive property of this function is retained. The problem is that the normal rules of precedence govern *within* the enclosing parentheses of %EVAL. This means that the multiplication is done to just part of the intended value of %X, instead of the full value, at any level. The result is therefore less than it should be.

As a general guide, it is advisable to surround any macro-time symbol with either parentheses or %EVAL() if you expect to produce a numeric value. For this example, one fix is to put %EVAL() around %X-1 in the call to %FACTORIAL. This forces evaluation of the subtraction before the value is passed to the next lower calling level. Another fix is to put parentheses around the second %X—as has been discussed and was done in the example for %EXIT. This causes parentheses to be around the subtractions preceding the multiplication sign that then force the intended order of arithmetic evaluation. The corrected macro definition, using the %EVAL() fix, follows:

```
%*DEFINE(FACTORIAL(X))
(%IF(%X LE 1) THEN (01H %EXIT) FI
%EVAL(%X*%FACTORIAL(%EVAL(%X-1)))
)
```

The corrected macro definition called with the same parameter of 4 would expand as follows:

```
4*FACTORIAL(3)
  3*FACTORIAL(2)
    2*FACTORIAL(1)
      01H
     2*01H
    02H
   3*02H
  06H
 4*06H
018H
```

13

User-Defined Macros

Introduction

This chapter provides information about defining macros, including the syntax for defining them, and how macros are referenced. User-defined macros are created by using the %DEFINE macro function.

User-defined macros can be defined in terms of themselves which means they can invoke themselves within their own macro bodies. This ability is called recursion. Any macro that calls itself must include a terminating condition that causes the macro to "bottom out" eventually or the preprocessor can enter into an infinite loop.

%DEFINE

If you want to define a macro, you must use the DEFINE function.

Because the syntax for DEFINE is somewhat complicated, the following figure contains the syntax diagram for DEFINE.

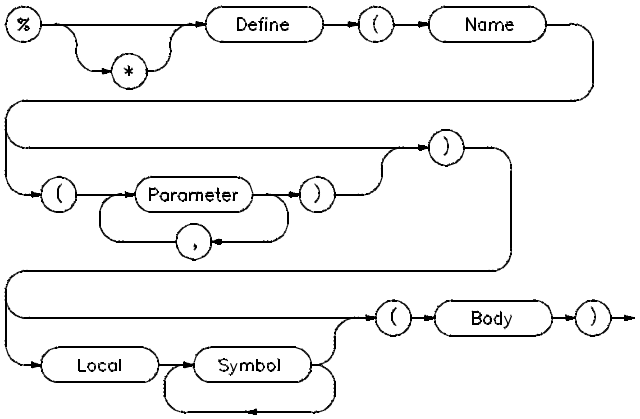


Figure 13-1. Syntax for User-Defined Macros

Where:

% is the current metacharacter (which is usually %).

***** is the optional literal character. This character should be used with most definitions. There are two reasons:

- It will inhibit the expansion of macro calls flagged by the current metacharacter (usually %) within the macro body at the time of macro definition. Instead, macro calls will be expanded at the time of macro reference.
- You must use the literal character with any macro that has formal parameters. Otherwise, the macro preprocessor will attempt to evaluate any references to the formal arguments within the macro body as symbols or other macro calls, which will result in errors.

Define is the pre-defined macro function for creating user-defined macros.

Name is the user-defined name to be associated with the macro. It cannot conflict with the predefined macro functions (listed in the previous chapter), but it can duplicate an earlier user-defined macro name or symbol. In the latter case, the previous meaning of the symbol is lost. The macro name should not be preceded by the current metacharacter (usually %).

Parameter is a formal parameter name. Formal parameters, if they exist, are replaced by actual parameters when the macro is invoked.

Note

Formal parameter names are not preceded by the metacharacter when they are being declared in the macroname argument list. To reference a formal parameter within the macro body, however, you must precede its name with the metacharacter (as in %ARGUMENT_NAME for the formal parameter ARGUMENT_NAME).

Parameter names must be distinct from one another within a macro, but they can duplicate other formal parameter names in other macros, since they have no existence outside the macro definition. They can also duplicate the names of other user macros or macro functions. If they do duplicate other macro function names, then the other macros or functions cannot be used within the macro body, since the duplicated name will refer instead to the parameter.

Local is the word that must precede the local parameter list.

Symbol is a local symbol name. Such symbols can be used only within the macro body. They are undefined outside of it.

The purpose of local symbols is to avoid multiply-defined symbols in the output of the macro processor. Each time the macro is referenced, each local symbol receives a unique two to five digit suffix. For example, if a local symbol LABEL were defined for use within a macro, then the first macro invocation might substitute LABEL00 and the second invocation might use LABEL01. This way, the assembler would not see a multiply-defined symbol. When locals are initially

being declared following the LOCAL keyword, they must not be preceded by the metacharacter. However, when referencing local symbols in the macro body, they must be preceded by the metacharacter. The symbol LOCAL is not reserved; a user symbol or macro can have this name.

Body is a balanced-text string. It can contain references to formal arguments and local symbols, if any, as described above. It can also include references to user-defined macros (including itself), to macro-expansion-time symbols (preceded by '%'), and to macro functions.

A macro should not redefine itself (%*DEFINE) within its body, however. The expanded value of DEFINE is the null string, but the macro body is stored internally for later use. A re-DEFINE in a macro body, then, is working at cross purposes.

Macro Reference

A macro is referenced by preceding its name with the metacharacter. If the macro was defined with formal arguments, the reference must include the same number of actual parameters, enclosed in parentheses and separated by commas. Actual parameters can be null, but the required delimiters must still be present between them. Each actual parameter is substituted for its corresponding formal parameter, wherever it appears in the macro body, on a string basis.

The literal character ('*') is acceptable in conjunction with references to user-defined macros. Normally, all metacharacters in the actual parameters are evaluated immediately when the macro reference is found and the resulting strings are stored. They are then substituted for the formal parameters as the macro body is copied. The literal character defers evaluation of actual parameters until they are found in the macro body, and they are re-evaluated each time they are found. It is possible, then, that the values of actual parameters might change between evaluations depending on what the macro body does.

Following are some sample macro definitions and references along with short discussions about each. Each new macro and discussion begins with the new %DEFINE, but an implied order of definition from

first to last is understood in order that some of the discussions make sense. Some of the macros are intentionally incorrect.

```
%*DEFINE(MAC1) ( DB 2 )
```

MAC1 will have the string value "DB 2" when invoked.

```
%*DEFINE(MAC2(ARG1)) (DB %ARG1)
```

MAC2 is stored as "DB %ARG1". %ARG1 is to be evaluated at the time of macro reference because of literal character ('*') precedes DEFINE.

```
%*DEFINE(ERR1(ARG1)) (DB ARG1)
```

ERR1 shows a common error. The '%' is omitted from the formal parameter in the macro body which means it will not be recognized. The assembler will be passed "DB ARG1" when the macro is invoked, which is not likely to be correct.

```
%*DEFINE(MAC3(ARG1)) (%MAC1
                      %MAC2(%ARG1))
```

MAC3 references the previously-defined macros MAC1 and MAC2. Since the evaluation of metacharacters in MAC3 is deferred (with *), this example would also work if the definitions of MAC1 or MAC2 followed that of MAC3 (as long as they are defined before MAC3 is invoked).

```
%DEFINE(ERR2(ARG1)) (%MAC1
                     %MAC2(%ARG1))
```

ERR2 shows another common error—the literal character was omitted. The metacharacters in the macro body are expanded immediately (at macro-definition time). Since there is a reference to a formal parameter, this cannot be done—there is no actual parameter to substitute for it. The macro preprocessor actually attempts to expand %ARG1 as a macro symbol or user-macro. In some cases this might be possible, although it is not likely to be what is expected.

```
%*DEFINE(ERR3(ARG1)) (%MAC1
%MAC2(%ARG1))
```

ERR3 shows another frequent user-error, a missing <EOL>. Since the body of neither MAC1 nor MAC2 includes an <EOL>, ERR3 should include one between their invocations (as MAC3 does). The invocation %ERR3(3) will yield "DB 2 DB 3" and cause an assembler

error. If MAC1 ended with an <EOL> or MAC2 began with an <EOL>, ERR3 would be correct.

```
%DEFINE(MAC4) (%MAC1
               %MAC2(4))
```

MAC4 shows an acceptable use of DEFINE without '*'. The stored body of MAC4 is shown in the following example, since the calls to MAC1 and MAC2 are evaluated immediately:

```
'DB 2
DB 4'
```

With the definitions of MAC1 and MAC2 shown above, %MAC4 is the same as %MAC3(4). But MAC1 and/or MAC2 might be redefined later on. In this case, MAC3 will reference the new values, while MAC4 will not.

```
%*DEFINE(MAC5(ARG1)) LOCAL LABEL (
     %LABEL:   MOV AW,%ARG1[IY]
               INC IY
               DBNZNE %LABEL)
```

MAC5 shows the use of a local symbol. Each invocation of MAC5 will create a unique assembler-time symbol from LABEL.

What is Output?

Note that the macro definitions above produce no output, since each DEFINE expands to the null string. Consider the macros as being defined sequentially without separating blank lines. The end-of-lines between the terminating right parenthesis of each macro body and the following metacharacter ('%') of the next macro result in blank lines that are not output. If the macro preprocessor did not remove blank lines, these examples would generate seven blank lines. This behavior is typical of readable macro code. All characters between the delimiting parentheses (including end-of-lines) are considered part of the macro body, which in turn is part of the syntax of DEFINE. Such characters are not considered for output.

Referencing Macro-time Symbols

Symbols are defined by the SET and MATCH functions. A symbol is referenced by preceding its name with the metacharacter, as in

```
%name
```

Without the metacharacter, the macro preprocessor treats "name" like any other character string. The call pattern of the symbol ends where the name ends (there is no argument in parentheses). The expanded value of this construct is the character string that had been assigned to it. (For instance: '01H'; or 'STRINGVALUE'; or the null string.)

The literal character ('*') is proper with a macro-time symbol. It inhibits the expansion of any metacharacters within the symbol value which otherwise would be expanded. For example, suppose the value of a symbol SYM is "%LEN(01H)." %SYM will expand to '03H', but %*SYM will expand to "%LEN(01H)". Generally, the literal character should be omitted.

The literal character similarly affects formal parameters of macros within the macro body. A formal parameter is not recognized if preceded by the literal character. This permits giving a formal parameter the same name as a macro function while still being able to access the function within the macro body. Example:

```
%*DEFINE(MAC(LEN)) (DB %LEN
                    DB %*LEN(%LEN))
```

The first %LEN is the formal argument LEN, as is the third. The second is not recognized as an argument because of the literal character, so it reverts to its normal meaning as a pre-defined function. The literal character has meaning to this particular function, so the inner %LEN is not expanded.

The literal character cannot be used with local symbols within a macro body.

This chapter concludes the reference for the apv20/apv33 Macro String Preprocessor. Error messages for the macro preprocessor can be found in the appendixes.

Notes

A

Error Message Formats

Error Classes

There are three classes of errors that may occur during assembler or macro preprocessor execution: warnings, errors, and fatal errors.

Warning

Warnings announce something that *might* be a problem in the output file. This may or may not indicate a problem with the program.

After a warning, the output files are written normally.

After a warning, asv20/asv33 and apv20/apv33 both return a code indicating "success" so that command files and "make" operations continue normally.

Error

Errors announce something that *is* wrong in the output file. For example, a reference to an unresolved symbol will cause problems at run-time.

After an error, the output files are written normally. The output files are complete and may be useful in subsequent operations.

After an error, asv20/asv33 and apv20/apv33 both return a code indicating "error" so that command files and "make" operations stop.

Fatal Error

A fatal error announces a condition that causes processing to be discontinued. After a fatal error, the output files are incomplete and corrupt. They are not useful for subsequent operations.

After a fatal error, asv20/asv33 and apv20/apv33 both return a code indicating "error" so that command files and "make" operations stop.

Notes

Assembler Error Messages

Introduction

When the assembler encounters a syntax error, it does not generate code for the instruction or directive on the line and any of its continuation lines where the error occurs. The error message is printed on the line below the error, with a caret (^) pointing to the offending syntax.

In some cases, the assembler issues a general syntax error that indicates there is something wrong at the place the caret points, but the specific nature of the error is not determined.

In the event of a syntax error, the assembler does not generate code, but continues processing with the next statement.

Syntax Errors

500

Expecting an expression. The assembler expected an expression, but found something different at the location pointed to by the caret.

501

Expecting an OR-level expression. The assembler expected an OR-level expression, but found something different at the location pointed to by the caret.

OR-level expressions include all the AND-level expressions plus the OR and XOR operators.

502

OR or XOR expected. The assembler expected an OR or XOR operator, but found something different at the location pointed to by the caret.

503

Expecting an AND-level expression. The assembler expected an AND-level expression, but found something different at the location pointed to by the caret.

AND-level expressions include all the NOT-level expressions, and the AND operator.

504

AND expected. The assembler expected an AND operator, but found something different at the location pointed to by the caret.

505

Expecting a NOT-level expression. The assembler expected a NOT-level expression, but found something different at the location pointed to by the caret.

506

Expecting a relational operator-level expression. The assembler expected a relational-level expression, but found something different at the location pointed to by the caret.

Relational-level expressions include all binary addition-level expressions plus the EQ, NE, LT, LE, GT, and GE operators.

507

Expecting a relational operator. The assembler expected a relational operator, but found something different at the location pointed to by the caret.

The relational operators are: EQ, NE, LT, LE, GT, and GE.

508

ENDS or constant definition directive expected.
The assembler expected to find an ENDS directive but found something different at the location pointed to by the caret.

509

Expecting an addition operator. The assembler expected an addition operator, but found something different at the location pointed to by the caret.

The addition operators are plus (+) and minus (—).

510

Expecting a multiplication-level expression. The assembler expected a multiplication-level expression, but found something different at the location pointed to by the caret.

Multiplication-level expressions include all

- byte-level expressions
- MOD, SHR, SHL
- multiplication and division operators
- base registers (BW, BP) and index registers (IX, IY)

511

Expecting a multiplication operator. The assembler expected a multiplication operator, but found something different at the location pointed to by the caret.

The multiplication operators are MOD, SHR, SHL, and multiplication (*) and division (/).

512

Expecting a valid argument to NAME. The assembler expected a valid module name argument to the NAME directive, but found something different at the location pointed to by the caret.

Byte-level expressions include all secondary-level expressions plus the HIGH and LOW operators.

513

Expecting a secondary-level expression. The assembler expected to find a secondary-level instruction but found something different at the location pointed to by the caret. Secondary-level expressions include all primary-level expressions, the segment override (colon), the PTR, OFFSET, SEG, and TYPE operators.

514

Expecting a primary-level expression. The assembler expected a primary-level expression, but found something different at the location pointed to by the caret.

Primary-level expressions include all expression primitives as well as the MASK, WIDTH, SIZE, and LENGTH operators, and the dot operator for structures.

516

Expecting a symbolic name. The assembler expected a symbolic name, but found something different at the location pointed to by the caret.

517

Expecting an integer constant. The assembler expected an integer constant, but found something different at the location pointed to by the caret.

520

Expecting a register. The assembler expected a register (such as AW, BW, BP, IX, and others) but found something different at the location pointed to by the caret.

521

Segment register expected. The assembler expected a segment register (PS, DS0, DS1, or SS) but found something different at the location pointed to by the caret.

522

NOTHING or segment register expected. The assembler expected the keyword NOTHING or a segment register (PS, DS0, DS1, or SS) but found something different at the location pointed to by the caret.

523

Expecting an identifier or integer constant. The assembler expected an identifier or integer constant, but found something different at the location pointed to by the caret.

524

Expecting identifier, directive, or colon. The assembler expected an identifier, directive, or colon, but found something different at the location pointed to by the caret.

525

Expecting an identifier or constant definition directive. The assembler expected an identifier or constant definition directive (such as DB, DW, DD, and others) but found something different at the location pointed to by the caret.

526

Expecting an identifier or type. The assembler expected an identifier or type, but found something different at the location pointed to by the caret.

527

SEGMENT expected. The assembler expected a segment, but found something different at the location pointed to by the caret.

528

PTR expected. The assembler expected a PTR operator, but found something different at the location pointed to by the caret.

529

DUP expected. The assembler expected DUP, but found something different at the location pointed to by the caret.

530

Expecting a comma. The assembler expected a comma, but found something different at the location pointed to by the caret.

531

Expecting a colon. The assembler expected a colon, but found something different at the location pointed to by the caret.

532

Expecting a period, left bracket, or left angle bracket. The assembler expected a period (.), left bracket ([), or left angle bracket (), but found something different at the location pointed to by the caret.

533

Expecting right bracket. The assembler expected a right bracket, but found something different at the location pointed to by the caret.

534

Expecting a left parenthesis. The assembler expected a left parenthesis, but found something different at the location pointed to by the caret.

535

Dollar sign expected. The assembler expected a dollar sign '$', but found something different at the location pointed to by the caret.

536

Expecting comma or right angle bracket. The assembler expected a comma or right angle bracket, but found something different at the location pointed to by the caret.

537

Expecting comma or right parenthesis. The assembler expected a comma or right parenthesis, but found something different at the location pointed to by the caret.

538

Expecting a left bracket. The assembler expected a left bracket, but found something different at the location pointed to by the caret.

539

Expecting a right parenthesis. The assembler expected a right parenthesis, but found something different at the location pointed to by the caret.

540

Expecting a label or a statement. The assembler expected a label or a statement, but found something different at the location pointed to by the caret.

541

Expecting an instruction mnemonic. The assembler expected an instruction mnemonic, but found something different at the location pointed to by the caret.

543

Assembler general control expected. The assembler expected a general control, but found something different at the location pointed to by the caret.

544

Expecting an assembler control. The assembler expected an assembler control, but found something different at the location pointed to by the caret.

545

Constant definition directive expected. The assembler expected a constant definition directive such as DB, DW, DD, and others, but found something different at the location pointed to by the caret.

546

Unexpected control or directive name, or missing END directive. An illegal primary control or directive was found at the location pointed to by the caret or an END directive was not found before the end of the source file.

547

Expecting a string. The assembler expected a string, but found something different at the location pointed to by the caret.

548

Expecting parenthesized text. The assembler expected a valid attribute to the SEGMENT directive, but found something different at the location pointed to by the caret.

549

Expecting valid attribute to the SEGMENT directive. The assembler expected to find an alignment type such as BYTE, PARA, INPAGE, and others, but found something different at the location pointed to by the caret.

550

Expecting a combine type. The assembler expected a combine type (PUBLIC, STACK, COMMON, and others) but found something different at the location pointed to by the caret.

551

Continuation line found where initial line was expected. The assembler found the continuation character (ampersand ['&']) as the first character on a line that it was expecting to *begin* rather than to *continue* with an assembly statement.

552

Logical end of program already encountered. Assembler statements, directives, or controls were found in a source file AFTER an END directive was encountered. The *only* legal input after an END directive are comment lines or blank lines.

554

Structure or record initialization expected. The assembler expected to encounter a left angle bracket, but found something different at the location pointed to by the caret.

555

Record field initialization expected. The assembler expected to encounter an equal sign, but found something different at the location pointed to by the caret.

556

Expecting a valid member of a GROUP. The assembler expected a valid member of a GROUP (such as a segment name), but found something different at the location pointed to by the caret.

557

Expecting an item which can be purged. The assembler expected an item that can be purged (such as symbolic names, instructions, and others), but found something different at the location pointed to by the caret.

558

Expecting a valid END initialization element. The assembler expected a valid END initialization element, but found something different at the location pointed to by the caret.

559

Expecting a valid ASSUME element. The assembler expected a valid ASSUME element, but found something different at the location pointed to by the caret.

561

Expecting valid CODEMACRO parameter information. The assembler expected to find valid CODEMACRO parameter information but found something different at the location pointed to by the caret.

562

Expecting a codemacro parameter specifier. The assembler expected to find a codemacro parameter specifier but found something different at the location pointed to by the caret.

563

This statement is not valid in a codemacro definition. The caret points to a statement that is not legal in the body of a codemacro definition. Refer to the chapter titled Codemacros for a list of valid statements.

564

Expecting a type. The assembler expected a Type (such as BYTE, WORD, DWORD, and others) but found something different at the location pointed to by the caret.

565

Unbalanced string delimiters. A string that was opened with an apostrophe or quotation mark does not have a closing apostrophe or quotation mark. Usually this is caused by failing to double occurrences of apostrophes or quotation marks that are contained in the text of the string.

566

Syntax error. In some cases, the assembler can determine that there is a syntax error, but can't determine exactly what the error is. In these cases, this general message is generated, with the caret indicating the point of the error.

567

Syntax error in command line options. Control options on the command line may only be delimited with spaces, tabs, or commas. Also, any arguments to controls must be delimited with parentheses.

568

Unbalanced parentheses. The number of right parentheses in the line does not match the number of left parenthesis. In complicated continued expressions, this could be due to the following line not having its continuation character in the first column.

569

Illegal operand for unary MINUS or NOT. Neither the unary minus nor the NOT operator can have a relocatable operand. The operand pointed to by the caret is relocatable.

570

Expecting a unary addition-level expression. The assembler expected to find a unary addition-level expression, but found something different at the location pointed to by the caret. Unary addition-level expressions include all of the multiplication level expressions as well as unary plus and minus.

571

Additional information encountered beyond end of statement.
After reaching what it thought was the logical end of a statement, the assembler found additional text at the location pointed to by the caret.

572

Expecting decimal or hexadecimal floating-point constant.
The assembler expected to find a decimal or hexadecimal floating-point constant at the location pointed to by the caret.

573

Expecting a signed integer constant. The assembler expected to find an integer constant with or without a leading unary plus or minus, but found something different at the location pointed to by the caret.

574

Expecting a SHORT-level expression. The assembler expected to find a SHORT-level expression but found something different at the location pointed to by the caret. SHORT-level expressions include all of the OR-level expressions as well as the SHORT operator.

575

Expecting an argument to an instruction or codemacro.
The assembler expected to find an argument to an instruction or

codemacro, but found something different at the location pointed to by the caret.

600

Illegal or mismatched argument. The caret points to the place where the operand type is incorrect for the instruction, or where the type doesn't match up correctly with another of the operands in the instruction.

601

Anonymous memory type. The size of the operand pointed to by the caret cannot be determined from the operand's expression, or from the content of other operands in the instruction.

602

Illegal type of expression. The expression pointed to by the caret is either not allowed in the directive or in the instruction in which it is specified, or the expression is not a valid expression.

603

Illegal type of argument in expression. The operator that precedes or follows the sub-expression being pointed to by the caret does not allow this type of sub-expression or one of its operands. Certain operators (such as * or /) allow only sub-expressions that resolve to an absolute number as an operand. Other operators only allow non-absolute expressions when certain conditions exist (see the description of '-' and relational operators).

604

Illegal or duplicate memory argument. Only one argument that references a memory location is allowed in any given instruction.

605

This instruction requires at least one operand. More than one operand had been supplied to this instruction, when only one operand is allowed.

606

This instruction requires at least two operands. Less than two operands (or more than two) have been supplied to this instruction; two are required.

607

This instruction requires three operands. Less than three operands have been supplied to this instruction; three are required.

608

Duplicate declaration of symbolic name. The symbolic name, pointed to by the caret, has already been declared in a previous statement.

609

Duplicate specification of module name. This message occurs when more than one NAME directive appears in the source program.

610

Duplicate occurrence of base register in register expression. Only one base register (BW or BP) may be used in any given register expression.

611

Duplicate occurrence of index register in register expression. Only one index register (IX or IY) may be used in any given register expression.

612

This symbol is not defined as a label. The caret points to a symbol, in a directive or expression, that must be a labl. The symbol pointed to by the caret is not a label.

613

This symbol is not defined as a segment or group. The caret points to a symbol, in a directive or expression, that must be a segment name or group name. The symbol pointed to by the caret is not a segment or group name.

614

This symbol is not defined as a variable. The caret points to a symbol, in a directive or expression, that must be a variable. The symbol pointed to by the caret is not a variable.

615

This symbol is not defined as a structure. The caret points to a symbol, in a directive or expression, that must be a structure. The symbol pointed to by the caret is not a structure.

616

This symbol is not defined as a structure field. The caret points to a symbol, in a directive or expression, that must be a structure field. The symbol pointed to by the caret is not a structure field.

617

This symbol is not defined as a structure or record. The caret points to a symbol, in a directive or expression, that must be a structure or record. The symbol pointed to by the caret is not a structure or record.

618

This symbol is not defined as a record field. The caret points to a symbol in a directive or expression that is required to be a record field in order to be valid. The symbol pointed to by the caret is not of this kind.

619

This symbol is not defined as a segment. The caret points to a symbol, in a directive or expression, that must be a segment. The symbol pointed to by the caret is not a segment.

620

Alignment type inconsistent. The alignment type specified in this SEGMENT directive is not the same as one specified in a previous segment directive for the same segment.

621

Combine type inconsistent. The combine type specified in this SEGMENT directive is not the same as one specified in a previous segment directive for the same segment.

623

Illegal or premature termination of segment. This error indicates improper nesting of segments or a misspelling of the segment name in either the SEGMENT or ENDS directives.

624

Segment nesting level exceeded. Segments can be nested to a level of 16 only.

625

Missing SEGMENT directive or previous segment nesting error. This ENDS directive has no associated SEGMENT directive, either due to omission or to a nesting error on its associated SEGMENT directive.

626

Expecting alignment type, combine type, or classname. The assembler expected an alignment type, combine type, or classname, but found something different at the location pointed to by the caret.

627

Classname inconsistent. The classname specified in this SEGMENT directive is not the same as one specified in a previous segment directive for the same segment.

628

Illegal type of symbol in this ASSUME. This error occurs when a symbol other than a segment or group is used in an ASSUME directive without being preceded by the SEG operator.

629

Initialization nest level exceeded. When using the DUP construct in conjunction with a data directive (DB, DW, DD, DS, DQ, DL, or DT), the maximum nesting level for DUPs is eight.

630

This symbol does not have a defined segment value, or segment not addressable. The symbolic name pointed to by the caret does not have a segment attribute in the list of legal attributes.

631

This argument does not have a defined offset value. The symbolic name pointed to by the caret does not have a offset attribute in the list of accepted attributes.

632

This argument does not have a defined type value. The symbolic name pointed to by the caret does not have a type attribute in the list of accepted attributes.

633

This argument does not have a defined length value. The symbolic name pointed to by the caret does not have a length attribute in the list of accepted attributes.

634

This argument does not have a defined size value. The symbolic name pointed to by the caret does not have a size attribute in the list of accepted attributes.

635

This argument does not have a defined field width value.
The symbolic name pointed to by the caret does not have a field width attribute in the list of accepted attributes.

636

This argument does not have a defined mask value.
The symbolic name pointed to by the caret does not have a mask attribute in the list of accepted attributes.

637

Immediate value overflow. The immediate value is not within the proper range for its context. Specifically, it is not within the range 0 to 0FFH for DB, 0 to 0FFFFH for DW or an instruction, and 0 to 0FFFFFFFFH for all others.

638

This expression must be absolute. The expression must resolve to an absolute number to be permissible in this context.

639

Item cannot be addressed by segment registers.
The segment associated with the variable pointed to by the caret is not currently ASSUMEd into any of the segment registers, nor has an explicit segment override been used.

641

Invalid floating point constant The floating point constant pointed to by the caret is not a valid floating point constant. No valid floating-point value can be stored for this constant.

642

Illegal operand in this register expression.
Register expressions may contain a base register (BW or BP), an index register (IX or IY), and any expression that evaluates to an absolute value. Expressions or symbols with relocatable results are not permitted.

643

Division by zero attempted. The divisor portion of this expression involving the division operator is itself an expression that evaluates to an absolute number with a value of 0.

645

This relational operator has an invalid operand or operands.
See the description of relational operators for what operands are valid.

648

Hexadecimal real constants are invalid in this context.
Hexadecimal real constants are allowed only in data definition statements or EQU definitions.

649

Illegal floating-point stack register (0-7 allowed).
A mnemonic representing an 8087 floating-point stack register was not in the legal list of mnemonics (ST,ST(0),ST(1),...,ST(7)).

650

Value too large for one-byte displacement. The number (or expression that evaluates to an absolute number) is pointed to by the caret is either less than -128 or greater than 255, and thus cannot be represented in just one byte.

651

Hex real constant size does not match with data directive. Hex real constants must be eight significant hex digits for the DD /DS directive, sixteen significant digits for the DQ /DL directive, and twenty significant digits for the DT directive.

653

This symbol cannot be purged. The following kinds of symbols cannot be purged:

- keywords
- segment names (including ??SEG)
- group names
- any user-defined symbol that has appeared in a PUBLIC statement

654

Symbol cannot be declared PUBLIC. PUBLIC symbols must be variables, labels or 17-bit constants; any other types will generate an error.

655

This symbol cannot be a member of a group. Only segments, externals, or variables may be used in a GROUP directive. Only a segment may be forward referenced.

656

Illegal statement in this context. This error is generated if a PROCLEN directive appears outside of a CODEMACRO definition, a STRUC statement appears within a structure definition, or if a structure initialization occurs within another structure initialization.

658

Illegal or premature termination of procedure. This error indicates improper nesting of procedure or a misspelling of the procedure name in either the PROC or ENDP directives.

659

Procedure nesting level exceeded. Procedures can be nested to a level of 16 only.

660

Illegal type in this context. This error is generated if a type other than NEAR or FAR appears in a PROC directive, or if a type other than a standard type (e.g. a structure or record name) appears as the argument to the THIS operator.

661

Illegal termination of structure. This error indicates a misspelling of the structure name in either the STRUC or ENDS directives.

662

Null initialization is not allowed in this context. Null (or default) initialization is permitted only in structure or record initialization, not in structure or record definition or data definition directives.

663

Invalid record field size. A given field within a record can be no larger than 16 bits, or no smaller than 1 bit.

664

Maximum record size exceeded. The size of a record is limited to 16 bits.

666

This variable is not defined as a record. The caret points to a symbol, in a directive or expression, that must be a record. The symbol pointed to by the caret is not a record.

667

Include file nesting limit exceeded. The limit for nested include files has been exceeded. This limit is operating system specific.

668

Cannot open include file. The filename specified in the preceding include control is misspelled, the associated file is not in the current directory, or the associated file cannot be opened.

669

Illegal type of EQU in this context. An example of this error is an EQU to a V20 instruction mnemonic as the expression portion of a data definition directive, such as DB. Many other similar conditions exist that will generate this error.

670

Too many arguments specified for this instruction. The particular instruction pointed to by the caret does not allow as many arguments as are specified. INC AW,BW, for example, has one too many arguments.

671

This type of segment override is illegal in this context. Certain types of expressions are not permitted to have a segment override operator (colon operator) as part of the expression. The expression pointed to by the caret is one such expression.

672

Illegal value for PAGELENGTH control. The minimum value in the PAGELENGTH control is 20 lines.

673

Illegal value for PAGEWIDTH control. The legal values for the PAGEWIDTH control fall in the range of 41 to 255 columns, inclusive.

674

Illegal value for TITLE control. The string for a TITLE control is limited to a length of 40 characters.

675

More than 64 levels of control saves. The $SAVE control cannot be nested to a depth greater than 64.

676

More than 64 levels of control restores. The $RESTORE control cannot be nested to a depth greater than 64.

677

This symbol is not a parameter to this codemacro. The symbol pointed to by the caret, which is contained within a codemacro definition, is not present in the CODEMACRO statement for the current codemacro. Therefore, the symbol cannot be a parameter to the current codemacro.

678

This symbol is not defined as a codemacro parameter. The caret points to a symbol in a directive or expression that is required to be a codemacro parameter in order for it to be valid. The symbol pointed to by the caret is not a codemacro parameter.

679

This codemacro parameter's specifier is invalid in this context. Certain directives within a codemacro definition allow only parameters that have specific types of codemacro specifiers. The codemacro parameter pointed to by the caret is not of the specific type needed for the directive in which it is used.

680

Illegal range expression in codemacro parameter definition. Either the range expression pointed to by the caret does not evaluate to an absolute number, or it is out of range according to the codemacro specifier with which it is associated.

681

This symbol is not a valid codemacro specifier. The symbol pointed to by the caret is not one of the valid codemacro specmod fields mentioned in the chapter titled Codemacros.

682

Duplicate definition of codemacro parameter. The symbol pointed to by the caret has appeared more than once in the same codemacro directive and is a duplicate definition.

683

This expression is illegal within a codemacro definition. Null initialization expressions, DUP expressions, and dot operator expressions that don't use a record field as their right operand are illegal within a codemacro definition.

684

This statement is not allowed in a codemacro definition. Only a limited number of types of statements is allowed in a codemacro definition. For a complete list, see the chapter titled Codemacros.

685

This instruction or codemacro has too many operands. asv20/asv33 limits the number of operands to 3 in an instruction and to 255 in a codemacro.

686

Duplicate use of NOSEGFIX directive in codemacro definition. Only one NOSEGFIX directive can be used in any given codemacro definition.

687

Duplicate use of SEGFIX directive in codemacro definition. Only one SEGFIX directive can be used in any given codemacro definition.

688

PREFX and non-PREFX codemacros cannot have the same name. The codemacro symbol being pointed to by the caret has been defined in codemacro directives both with and without the PREFX keyword. The last definition of the codemacro is the one that will be in effect.

689

Missing PROC directive or previous procedure nesting error. This ENDP directive has no matching PROC directive due to an omission or a nesting error involving its associated PROC directive.

690

This symbol has not been defined. During Pass 1, the assembler assumes that an undefined symbol is a forward reference. This message occurs when the symbol is still not defined in Pass 2. The assembler generates NOPs and continues assembly. You should modify the code to define the symbol, or the symbol will have no value.

691

PS cannot be destination register. PS can only be changed by using an ASSUME directive, a BR or CALL instruction to a FAR location, and a MOV or POP has been used to load the PS register.

692

Pass 1 estimate of instruction bytes insufficient. The number of bytes reserved for an instruction as a result of a forward reference in Pass 1 did not leave enough code space for the instruction in Pass 2.

693

This symbol is not defined as a group. The symbol before the GROFFSET operator or following the GRSIZE operator must be a group name. If it is not, then this error is generated.

694

Shift values greater than 31. A value for one of the shift or rotate instructions evaluated to a value that was greater than 31. Adjust the shift value and reassemble.

695

DS1 cannot be overridden in this string instruction. Certain types of string instructions (e.g. MOVBK) require that their second operand use the DS1:IY combination for their reference. In such instances, the DS1 register cannot be overridden. Modify the program to do such operations through the DS1 register, and reassemble.

697

Illegal character in numeric constant. An illegal character for a numeric constant was found in the constant pointed to by the caret. Remove the illegal character and reassemble.

698

Illegal DUP value. A negative or zero repeat count value for a DUP initialization was found at the location pointed to by the caret. Only positive repeat values are allowed. Correct and reassemble.

699

No forward references allowed in EQU expressions.
The expression pointed to by the caret contains an as-yet undefined symbol. Since this expression is being defined as an EQU symbol, such forward references are not allowed. Eliminate the forward reference by moving the definition of the as-yet undefined symbol in front of this EQU definition, and reassemble.

701

This construct is invalid in the current assembly mode.
Certain constructs that are accepted only by a given assembly mode (MODV20, MODV25, or MODV33) that aren't accepted in the current assembly mode will cause this error to be generated.

702

No module name specified. No NAME directive was found in the source program. The default name, which is the basename of the source file, will be used.

703

This symbol was previously declared public. The symbol pointed to by the caret previously appeared in this or another PUBLIC directive.

704

Too many initializations specified: remainder ignored. When re-initializing a structure or record at allocation time, this message is generated if more initialization values were specified than there were fields in the structure or record.

705

This field cannot be re-initialized: value not changed. Structure fields with many values or a DUP expression cannot be re-initialized at allocation time.

706

Illegal initialization value: not re-initialized. An attempt was made to initialize a structure or record field with an invalid value.

707

Location counter overflow. Addition of the current instruction or data definition directive causes the current segment's location counter to exceed the value 0FFFFH, i.e. the 64k limit of a segment. The location counter is set to the value MOD 65536.

Note 

This may cause previous code or data to be overwritten if this is ignored.

708

This EQU cannot be made public. Certain types of EQU symbols, such as those representing instructions or address expressions, are not permitted to be declared PUBLIC.

709

Floating point overflow: set to infinity. The number of bytes in a floating point value exceeds the limit of a DD /DS (32 bytes), DT (80 bytes) or DQ /DL (64 bytes) directive. Assembly continues; adjust the value to fit within the limit of the Data Directive used.

710

Floating point underflow: set to zero. The number of bytes in a floating point value is under the limit of a DD /DS (32 bytes), DT (80 bytes) or DQ /DL (64 bytes) directive. Assembly continues; adjust the value to fit within the limit of the Data Directive used.

711

BCD value exceeds 18 decimal digits. A packed decimal value (DT) can take 18 digits only; anything over 18 is truncated. Assembly continues; adjust the value to fit within the 18 digit limit.

712

Integer value exceeds 64-bit limit. This warning occurs when an integer constant used in a DQ or DL directive has a value outside the range 0 to FFFFFFFFFFFFFFFFH. Correct the value and reassemble.

716

This and future preprocessor statements will be ignored. Meta characters have not been preprocessed; assembly continues. The assembler does not process any lines with meta characters. Execute the macro string preprocessor before assembling.

717

Segment limit exceeded for this segment. The specified segment contains instructions and/or data that take up more than the maximum allowable 64K bytes of space. Break the segment into multiple segments or shrink the size of the segment, and reassemble.

718

Procedure not closed within this segment. A procedure (or procedures) whose PROC directive was defined in the segment having an ENDS directive which is currently being processed has not yet been closed. The procedure should be closed by inserting an ENDP directive at some point before the ENDS directive.

719

Segment not closed by end of module. One or more segments were open at the point where the assembler found the END directive. The segments should be closed at the appropriate point within the source file.

720

Procedure closed in segment other than the one it was defined in. The ENDP directive, which closes a procedure, appears in a different segment than the one in which the matching PROC directive appears. Make sure that the PROC and ENDP directives reside within the same segment.

722

String truncated to 2 characters before integer conversion. A string that appears anywhere other than in a DB directive must be either 1 or 2 characters long. If such a string is longer than 2 characters, it will be truncated to 2 characters and converted to an integer.

724

Record field overflow: 'value' modulo 'field width' used. If a record field initialization or reinitialization expression evaluates to a value that won't fit the specified record field, the appropriate modulo operation is performed in order to force the value to fit.

726

Illegal assembly mode. The instruction pointed to by the caret is not valid in this assembler.

727

Overriding string too large for field. If a string field in a structure is reinitialized and the string is too long for the specified field, the string is truncated and this warning message is displayed.

728

Source path names for debug have been truncated to 255 characters. If the assembly module was produced by the AxLS C compiler and the full path name for the source file or any include file is longer than 255 characters, the assembler will truncate the path name from the left, adding an ellipsis to the name to create a total length of 255 characters, and emit this message.

729

High-level block nesting limit exceeded: some variable scoping lost. Nesting of high-level procedure or code blocks is allowed up to a depth of 15. Any nesting beyond this depth will result in the loss of information about which block symbols belong to.

800

EVEN directive cannot be in a BYTE aligned segment. You cannot use the EVEN directive within a segment whose alignment attribute is BYTE. In such a segment, there is no need to force the

alignment to be on a word boundary as it will not be any more effective by doing so. Comment out or remove the unnecessary EVEN directive and reassemble.

801

PS-PC initialization required for main module. Some register initializations were provided on the END directive; however, this error message indicates that no initialization for the PS:PC registers was provided. If any register initializations are provided, an initialization for PS:PC must be provided as well. Add the appropriate initialization and reassemble.

802

Illegal initialization of SS register. It is illegal to initialize the SS register to anything other than a segment base. In particular, group bases are not allowed. Correct the initialization on the END directive and reassemble.

803

Circular chain of equates. EQU symbols in a list with a length of at least one were defined as other EQU symbols in such a way that the last symbol in the list was defined as the first symbol in the list. Usually, such a construct results from symbol spelling errors, or in larger programs, widely scattered EQU definitions. Correct the erroneous EQU definition and reassemble.

804

Illegal to use relocatables in DB, DQ, or DT. If a relocatable value appears in an expression for a DB, DQ, DL, or DT directive, this error is generated. Remove the relocatable value and reassemble.

805

Variables or Labels cannot be in DB, DQ, or DT.
An expression that contains a variable or label is not allowed in a DB, DQ, DL, or DT directive.

806

Illegal to use multiple INCLUDE controls on line. Only one INCLUDE control is allowed on any given line containing assembler controls. Split the control line into as many lines as necessary to obtain control lines with only one INCLUDE control per line, and reassemble.

807

Inconsistent AT value given for segment. A segment was specified in a previous SEGMENT directive with a different absolute paragraph number than is specified in the current SEGMENT directive. The paragraph values should be the same.

808

This codemacro specifier cannot have a range.
The codemacro specmod field being pointed to by the caret is not permitted to have an associated range. Only codemacro parameters with specifiers A, D, G, I, R, or S can have range values.

809

Duplicate specification of alignment type. A segment directive can only have a single alignment type as an option. This error is generated if more than one alignment type is detected in the segment directive.

810

Duplicate specification of combine type. A segment directive can only have a single combine type as an option. This error is

generated if more than one combine type is detected in the segment directive.

811

Duplicate specification of class name. A segment was specified in a previous SEGMENT directive with a different class name than is specified in the current SEGMENT directive. Both SEGMENT directives should use the same class name.

812

Maximum source line length exceeded. An input source line exceeded 1024 characters in length. The assembler will not accept lines longer than this length.

813

Maximum string length exceeded. A string was defined that exceeded 1024 characters in length. The assembler will not accept strings longer than this length.

814

Duplicate declaration of ASGNSFR directive. Only one ASGNSFR directive is allowed in a source file. Remove the duplicate entry.

815

Duplicate declaration of SETIDB directive. Only one SETIDB directive is allowed in a source file. Remove the duplicate entry.

816

SETIDB expression out of range. The SETIDB directive only allows constant expressions in the range of 0 to 0FFH. Modify the expression to fit within that range.

817

assembler not in MODV25 mode The SETIDB or ASGNSFR directives may only be used when the asv20/asv33 assembler is in MODV25 mode. Reassemble with the MODV25 control set.

818

This codemacro specifier must have a range. The 'G' codemacro specifier must always have a range as part of the specification. It is not possible to match all possible registers that map to this specification.

819

This codemacro range may only contain one argument. The 'G' codemacro specifier may only match a single register. As such, the range used with the specifier may only have a single register.

820

Relocatable numbers not allowed in DD. A relocatable value was used in a DD directive, which is not allowed. Only relocatable full addresses, segment, or group names may be used in a DD or DS directive.

821

DS or DL directives must be on a word boundary. The DS and DL data directives must be on a word boundary in order for 72291 floating point instructions to work correctly. It is invalid to use these directives on an odd counter location.

822

DS or DL directives cannot be in a BYTE or INPAGE aligned segment. The DS and DL data directives must be used in segments that cannot be relocated to an odd address boundary. As such, it is

invalid to use these directives in BYTE or INPAGE aligned segments since these segments can be placed on such a boundary.

823

FDWORD or FQWORD variables must be on a word boundary. FDWORD and FQWORD variables must be on a word boundary in order for 72291 floating point instructions to work correctly. It is invalid to define these variables at an odd counter location.

824

FDWORD or FQWORD variables cannot be in a BYTE or INPAGE aligned segment. FDWORD and FQWORD variables must be defined in segments that cannot be relocated to an odd address boundary. As such, it is invalid to define these variables in BYTE or INPAGE aligned segments since these segments can be placed on such a boundary.

825

Codemacro argument cannot be addressed by the required segment register. The codemacro requires that one of its arguments be addressable through a specific segment register. The current ASSUME contents for that register does not allow that argument to be reached, so this error is generated.

826

Iterated Data record offset is too large for a fixup. Fixups to object code can only occur within the first 1024 bytes of a record. In this instance, an iterated data record is being created that is larger than 1024 bytes and requires a fixup beyond that point. This cannot be represented in HP-OMF86 so this error is generated.

827

OMF record length exceeds maximum value. An HP-OMF86 record can only be 64k in size. Any attempt to generate more than 64k of text in a single HP-OMF86 record will result in this error message.

828

Codemacro instruction length exceeds 247 bytes. A single codemacro instruction can only generate up to 247 bytes of object code. Any instruction that generates more than that number of bytes will result in this error message.

996

Internal error.

997

Fatal Error.

998

***** Fatal Internal Error: Unimplemented Semantics ***.**

999

************ FATAL INTERNAL ERROR **********.**

C

Macro String Preprocessor Error Messages

The Macro Preprocessor produces numbered error messages. This appendix explains the meaning of the numeric codes. More than one message may appear for a given source line. Each message is printed immediately upon detection of the error (because the macro processor is character-oriented, not line-oriented). The usual effect is for a message to appear *before* any output from the source line that caused the error. Macro error messages appear as assembler comments in the output source file, like this:

```
; ***** ERROR 301
```

Error Codes and Messages

301

Undefined macro name. The text following a metacharacter (%) is not a recognized user function name or built-in macro function. The reference is ignored, not passed to the output file, and processing continues with the character following the name.

302

Illegal call to %EXIT. %EXIT is outside any user macros, WHILEs, or REPEATs. The call is ignored, %EXIT is not passed to the output file, and processing continues.

303

Illegal expression. A numeric expression was expected. There could be a missing % from a macro-time symbol or a syntax error, among others. This message is produced when apv20/apv33 is trying to evaluate an expression within EVAL, IF, WHILE, SUBSTR or REPEAT. The function call is aborted (any output from it is lost) and processing continues following the call pattern of the function. This message is also reported when an illegal character is detected in a string being compared with %EQS (or other string comparison functions).

304

Logical Expression Error

305

Missing "FI". Self-explanatory. This has no effect except to produce the message. However, the search for FI is character-by-character, so that if FE was present when FI was expected, the F would be removed from the output file. The E and subsequent characters would be passed on normally.

306

Missing "THEN". Self-explanatory. The call to IF is aborted and processing continues following the first character which failed to match. Thus the THEN and ELSE clauses, and the ELSE and FI keywords, will be treated as normal text and expanded normally. As with FI, the search for THEN is character- by-character.

307

Illegal attempt to redefine macro. A built-in function cannot be re-defined at any time. It is not possible to re-define a macro formal parameter within the macro body or a macro name within its own body.

309

Missing balanced string. In a call to a built-in function, a required balanced-text string delimited by parentheses is not present. This error can also be generated when the leading left parenthesis is not found where expected. The function call is aborted and scanning continues from the point at which the error was detected.

310

Missing list item. A list item (delimited by commas) is missing. The function or macro call is aborted and scanning continues from the point where the error was detected.

311

Missing delimiter. A delimiter required when scanning of a user-defined macro or built-in function (a comma, usually) is not present. The macro function call is aborted and scanning continues from the point at which the error was detected.

312

Premature EOF. The end of the input file occurred while the call to the macro was being scanned. This usually occurs when a right parenthesis is omitted, causing the Macro Preprocessor to scan to the end of the file searching for it. Note that even if the closing parenthesis of a macro call is given, this error may occur if any preceding commas are missing, since the Macro Preprocessor searches for delimiters one by one.

313

Macro stack overflow. The macro context stack MSTAK has overflowed. This stack is 64 deep and contains an entry for each symbol preceded by the metacharacter. The cause of this error is excessive recursion in macro calls or expansions; a likely source is a user-programmed infinite loop. When this error is encountered, the stack is emptied and all pending output destroyed; scanning continues at the next character in the input file. This message can also be produced to indicate that INCLUDEs were nested too deeply.

314

Nested macro error.

315

String buffer overflow. The string buffer used in conjunction with the macro stack to save intermediate results from nested macro calls has overflowed.

318

Illegal metacharacter. Self-explanatory. The current metacharacter remains unchanged.

319

Unbalanced right parenthesis. During the scan of a call to a user-defined macro, an unmatched right parenthesis was encountered. This is frequently because of a missing argument (the right parenthesis terminating the macro call was found when a comma was expected). The call is aborted and scanning continues from the point at which the error was detected.

338

Invalid symbol. A symbol (not preceded by the metacharacter) is required in certain contexts, such as the MATCH, DEFINE and SET functions. This symbol was not valid.

340

Literal character on SET or WHILE. The constructs %*SET and %*WHILE make no sense and produce this message. The * is ignored, and the Macro Preprocessor attempts to expand SET or WHILE normally.

401

Bad or missing parameter. The parameter to a control is not correctly formed, or a control that requires a parameter does not have one. Typographical errors often lead to this message.

414

Unable to open include file. Self-explanatory.

901

Scan stack overflow. This error indicates that the stack used for evaluating complex expressions has overflowed. This will not occur for any expression likely to be useful in practice. Break the expression into smaller ones.

906

Macro symbol table exhausted. The macro-time symbol table is full. This table contains symbol names plus the string values of SET and MATCH symbols.

Notes

D

ASCII Codes

The Assembler will recognize the following characters. The equivalent codes are expressed in hexadecimal notation.

Table D-1. ASCII Codes

Char.	ASCII	Char.	ASCII	Char	ASCII
blank	20	@	40	‘	60
!	21	A	41	a	61
"	22	B	42	b	62
#	23	C	43	c	63
$	24	D	44	d	64
%	25	E	45	e	65
&	26	F	46	f	66
’	27	G	47	g	67
(	28	H	48	h	68
)	29	I	49	i	69
*	2A	J	4A	j	6A
+	2B	K	4B	k	6B
,	2C	L	4C	l	6C
-	2D	M	4D	m	6D
.	2E	N	4E	n	6E
/	2F	O	4F	o	6F
0	30	P	50	p	70
1	31	Q	51	q	71
2	32	R	52	r	72
3	33	S	53	s	73
4	34	T	54	t	74
5	35	U	55	u	75
6	36	V	56	v	76
7	37	W	57	w	77
8	38	X	58	x	78
9	39	Y	59	y	79
:	3A	Z	5A	z	7A
;	3B	[	5B	{	7B
<	3C	\	5C	|	7C
=	3D	]	5D	}	7D
>	3E	^	5E	~	7E
?	3F	_	5F		

Converting HP 64853 Assembly Language Programs

Introduction

This appendix documents the changes that must be made to source files written for the HP 64853 assembler so that they can be assembled with the HP 64873 assembler. Not everything that appears in the HP 64853 format source files can be translated into something that the HP 64873 assembler will recognize, but most can. Translation is done in two ways. Some translations must be done manually. Most translations, however, can be done by the acvtv20 translation program also described in this appendix.

Note

The program acvtv20 automatically performs most of the transformations described here. acvtv20 is an unsupported porting tool. acvtv20 is not a part of the 64873 product and is distributed at no cost. Hewlett-Packard makes no warranty on its quality or fitness for a particular purpose.

The first section of this appendix discusses the acvtv20 porting tool and issues that are caused by the differences between the two assemblers. The next section describes the manual translations to macros that must be done because the porting tool cannot perform some macro translations. The third section gives the command syntax for acvtv20. The final section is an old and new list. This section is arranged alphabetically according to keywords in HP 64853 assembly language. It gives a side-by-side comparison between the old and new syntax and shows you how acvtv20 transforms particular HP 64853 constructs.

acvtv20 Introduction

This section describes the way that acvtv20 approaches the conversion process, what it produces, and its limitations. It also describes the sequence you should follow to translate files that contain include files. This section will give you a better understanding of what you will have to do to complete the translation process.

Note

The first line of an HP 64853 program identifies the target processor. HP 64873 assembly language supports only the 70108/70116/70208/70216 and 70320/70330/70325/70335 processors. The HP 64873 assembler does not support the 8086, 80186, 8089, or 80286 microprocessors. Therefore, the following target processor identifiers are not recognized: "8086", "8088", "80186", "80188", "8089_86", "8089_88", and "80286".

Assembler Differences

The HP 64873 assembler is really two programs: the preprocessor, apv20; and the assembler, asv20. The preprocessor implements the following features of the old HP 64853 assembler: SET directives, REPT directives, MACRO definitions and expansions, and IF/ELSE/ENDIF conditional assembly directives. The assembler then completes the process by assembling the file produced by the preprocessor. acvtv20 translates these features in the following way.

IF

```
IF <expr>
lines
ELSE
lines
ENDIF
```

translates to

```
%IF((<expr>)NE 0)
THEN
(lines)
ELSE
(lines)
FI
```

EQU

```
id EQU <expr>
```

translates to

```
id EQU <expr>
```

If <expr> is a constant expression, acvtv20 generates %SET(id,<expr>) and acvtv20 also stores id in its symbol table. Later, when id is referenced in preprocessor expressions, acvtv20 recognizes it and translates it to %id.

MACRO

```
id MACRO &P1,&P2
lines
MEND
```

translates to

```
%*DEFINE(id(P1,P2))
(lines)
```

acvtv20 also stores id in its symbol table. Later, when id is referenced, acvtv20 recognizes it and translates it to %id.

REPT

```
REPT <expr>
 next line
```

translates to

```
%REPEAT(<expr>)
(
 next line
)
```

SET

```
id SET <expr>
```

translates to

```
%SET(id, <expr>)
```

acvtv20 stores id in its symbol table. Later, when id is referenced, acvtv20 recognizes it and translates it to %id.

Note 

Sometimes the constant expressions in IFs or REPTs cannot be translated. HP 64853 calculates its constant expressions using 32 bit numbers. apv20 uses only 17 bit numbers. HP 64853 allows constant expressions to be formed by subtracting two relocatable symbols. apv20 cannot do this because it has no knowledge of the value of relocatable symbols.

External Declarations

HP 64853 allows an external identifier to be associated with a segment register in the EXT directive. For example:

```
EXT DS1:X1 WORD
MOV AW,X1   ;references X1 using DS1
```

If you use X1 in certain memory reference operands, the HP 64853 assembler will automatically generate an DS1: segment override for the instruction.

The HP 64873 assembler does not have an equivalent capability. Instead, an external identifier can be associated with a segment by placing the EXTRN directive inside a SEGMENT/ENDS pair. The segment may then be associated with a segment register through an ASSUME directive.

Since it would be difficult to automatically perform this kind of rearrangement, acvtv20 instead does the following:

- When an external declaration with an associated segment register is found, acvtv20 stores the identifier and segment register in its symbol table.

- When the external identifier is referenced, acvtv20 will generate (when appropriate) an explicit segment override. For example, the instructions shown above would be translated to the following.

```
EXTRN X1:WORD %'DS1:D:X1'
MOV AW,DS1:X1   ;references X1 using
DS1
```

The preprocessor comment %'DS1:D:X1' records the information in the original EXT directive. If, in a subsequent translation, acvtv20 sees this comment when reading a translated include file, it can update its symbol table just as if it saw the original declaration.

Note

You should not do SET definitions, constant EQU definitions, or "EXT segreg:id" declarations using MACRO parameters. For example:

PSWORD MACRO &P1
EXT PS:&P1 WORD
MEND
PSWORD X1

While this arrangement works perfectly well in the HP 64853 assembler, acvtv20 cannot tell that the variable X1 will have the implied PS: override quality. It may not translate references to X1 correctly.

Porting Procedure— Main Files with INCLUDE Files

Here is a procedure for translating a main file and its INCLUDE files. This sequence gives acvtv20 its most complete symbol table and allows it to do the most accurate translation.

1. Translate the include files first. Use the -c option to specify the main file as the context file. This allows definitions in the main program to be used when translating the include file. Furthermore, as each include file is translated, its definitions are available for translating subsequent include files.

2. Make manual corrections to translated include files. Typically, this means rewriting .IF, .GOTO, etc., directives in MACROs.

3. Translate the main file(s). Make corrections to the main file(s).

4. Assemble the main file(s) using the HP 64873 preprocessor/assembler. Correct preprocessor and assembly errors.

For example, here are three files: prog.S, inc1, and inc2

Main File prog.S

```
" 70116 "
;prog.S
  EXT  DS1:X1 WORD
  INCLUDE inc1
  INCLUDE inc2
  M2 ;defined in inc2
END
```

Include File inc1

```
;inc1
DISP SET 6
```

Include File inc2

```
;inc2
M2 MACRO
  MOV AW,X1   ;X1 defined in prog.S
  ADD AW,DISP ;DISP defined in inc1
MEND
```

First, translate inc1 as follows.

```
$ acvtv20 -c prog.S inc1 >  inc1.h
```

Second, translate inc2. Because of the "-c prog.S" option and because we have already created inc1.h, acvtv20 will correctly translate the references to X1 and DISP.

```
$ acvtv20 -c prog.S inc2 >  inc2.h
```

Finally, translate prog.S. Because inc2.h exists, acvtv20 will correctly translate the reference to macro M2.

```
$ acvtv20 prog.S >  prog.s
```

acvtv20 Warnings, apv20 Errors, asv20 Errors

To do a successful port, you must pay attention to messages from three sources.

acvtv20

acvtv20 issues warnings when it detects something that may need your attention. For example, it issues a warning when a MACRO call has more actual parameters than it has formal parameters in the MACRO definition. As previously explained, the two assemblers operate differently in this situation. Depending on how your MACRO is written, you may or may not need to change this statement.

apv20

After translating your files, you must understand and correct preprocessor errors. For example, errors may result from using constant expressions whose value is too large for the 17 bit preprocessor expression limit.

asv20

Finally, you must understand and correct asv20 errors. Assembler errors have numerous causes. For example, HP 64853 allowed user labels to duplicate instruction mnemonics (e.g. TEST). HP 64873 does not allow this and produces a syntax error. In this case, you should change the name of the offending label.

Code Substitution

acvtv20 has a feature that allows HP 64873 code to coexist with HP64853 code in an untranslated assembly source file. This feature is useful when, instead of doing a one-time port, you want to maintain a single, untranslated source file and then use acvtv20 as necessary to obtain translated source.

acvtv20 treats the comment ";sub64873;" in a special way. When acvtv20 sees that comment, it does the following:

- Discards all the text before the ;sub64873; comment. Any warnings generated by this text are also discarded. Note that acvtv20's symbol table is still updated normally if the discarded text contains certain directives.
- Writes any text following the ;sub64873; comment to standard output without any changes.

In the example below, we want to substitute legal HP 64873 code for the .IF and .NOP directives that acvtv20 does not translate

```
.IF &P1.GE.0 LAB  ;sub64873;%IF(%P1
LT 0) THEN (
      DW  &P1
  LAB .NOP        ;sub64873;) FI
```

acvtv20 will produce the following output for the preceding text

```
%IF(%P1 LT 0) THEN (
    DW  %P1
 ) FI
```

Note

acvtv20 only recognizes the substitution string ;sub64873; at the beginning of the comment field. In the example below, acvtv20 will not make a substitution because comment text precedes the ;sub64873; string.

```
DW &&P1 ;indexed parameter;sub64873;
DW %P1
```

BIN, DECIMAL, HEX, OCT

These four HP 64853 directives generate data with bytes that are reversed from the normal V-Series convention. When translating, you must adjust the value of operands to these directives to compensate for this. Examples:

BIN

HP 64853	HP 64873
BIN 10110	DW 10110 0000 0000B

DECIMAL

HP 64853	HP 64873
DECIMAL 999	DW -6397

HEX

HP 64853	HP 64873
HEX ABCD	DW 0CDABH

OCT

HP 64853	HP 64873
OCT 777	DW 1 774 001Q

V25/35 Considerations

The HP 64853 allows the predefined Special Function Register (SFR) names only for the 'BTCLR' instructions, with no setup for the special function register area. The HP 64873 allows the predefined SFR names even for memory references. But, setup for the SFR area must be done before using SFR names, even when only the 'BTCLR' instructions refer to the predefined SFR names.

SFR Area Declaration

acvtv20 automatically adds the following three lines at the beginning of the "70320" and "70330" HP 64853 programs:

```
SFR           SEGMENT PUBLIC
SFR           ENDS
ASGNSFR       SFR
```

You can remove these, if there is no SFR name reference in the program. Otherwise, you must make sure to locate the segment 'SFR' within 64k of the location of the internal RAM area at link time. The location of the internal RAM area is defined by the SETIDB directive. For example, if you have specified the location of the internal RAM (and SFR) area in the following way:

```
SFR           SEGMENT PUBLIC
SFR           END
ASGNSFR       SFR
    :
SETIDB        12H
    :
```

You must locate the SFR segment at an address before 12E00H at link time. The internal RAM and SFR registers start at that address, so any references to these registers require the SFR segment to be within 64k of that address.

Manual SFR-name Translation

If you have defined the SFR area in your HP 64853 programs, you should remove the definition and modify your programs to use the predefined SFR names.

Manual Macro Translations

acvtv20 automatically translates simple MACRO definitions (i.e. those without .IF, .SET, .GOTO, or .NOP directives and without indexed "&&PNO" parameters). For example:

```
M1   MACRO &P1
  TEST  &P1,AW
  BNE   L1&&&&
  MOV AW,1
L1&&&&
MEND
```

translates to

```
%*DEFINE(M1(P1)) LOCAL L1 (
  TEST  %P1,AW
  (BNE)   %L1
  MOV   AW,#1
%L1:
)
```

More complicated structures must be translated manually. Generally, this can always be done except when .IF or .SET expressions use symbol values which cannot be calculated at preprocessor time. The following sections illustrate techniques for translating typical structures.

.IF, .GOTO, and .NOP Directives

The HP 64853 .IF, .GOTO, and .NOP conditional assembly directives must be manually translated into the HP 64873 %IF preprocessor directives. For example:

```
.IF &P1.LT.0 L1
.IF &P1.GT.10 L1
  MOV AW,&P1
.GOTO L2
L1  .NOP
  MOV AW,#1
L2  .NOP
```

translates to:

```
%IF(%P1 GE 0 AND %P1 LE 10) THEN
( MOV AW,%P1
 ) ELSE
(    MOV AW,1
 ) FI
```

If the branches in your MACRO do not define a block structure, you must rearrange the MACRO to conform to the IF/THEN/ELSE structure of apv20.

Looping Structures

Macros branches which do loops can be translated into %REPEAT or %WHILE structures. For example

```
COUNT .SET &CNT
  LOOP  .NOP
DW   &VALUE
  COUNT .SET COUNT-1
  .IF  COUNT.GT.0 LOOP
```

translates to

```
%REPEAT(%CNT)
(  DW   %VALUE
)
```

Numeric, String, and Null Comparisons

The HP 64853 .IF directive performs either numeric or string comparisons depending on the operands being compared. Numeric comparisons must translate into apv20 numeric expressions; string comparisons must use the apv20 preprocessor string comparison functions. Examples:

A numeric comparison.

```
HP 64853                HP 64873
.IF &P1.EQ.0 L1         %IF(%P1 NE
0) THEN
```

A string comparison.

```
HP 64853                HP 64873
.IF "&P1".EQ."0"
%IF(%NES(%P1,0)) THEN
```

HP 64853 allowed a null parameter either to be the null string ("") or to be omitted entirely (except for a comma placeholder). Here is how

to test for an omitted or null macro parameter. Check for both of these possibilities in your translated .IF directive. Example:

```
HP 64853                    HP 64873
.IF &P1 .EQ. "" L1
%IF(%NES(%P1,) AND

%NES(%P1,'')) THEN
```

Indexed Parameters

HP 64853 allows MACRO parameters to be referenced by number. HP 64873 has no equal facility. Two translation techniques can be used.

1. Use %*DEFINE to make a new identifier which has the value of the indexed parameter. For example:

```
HP 64853
M1  MACRO &P1,&P2
IP  .SET  1
 .IF   &P1.EQ."" L1
IP  .SET  2
L1  .NOP
 DW &&IP
 MEND
```

```
HP 64873
%*DEFINE(M1(P1,P2)) (
%IF(%NES(%P1,) AND %NES(%P1,'')) THEN
(%*DEFINE(IP) (%P1)) ELSE
(%*DEFINE(IP) (%P2)) FI

DW    IP
)
```

2. Sometimes a MACRO indexes an indefinite number of parameters. This can be handled with the %MATCH function. For example, the following MACRO defines one word for each actual parameter. It stops on the first null parameter or at the end of the list.

```
HP 64853
WORDS MACRO &P1
INDEX .SET 1
LOOP  .IF  &&INDEX.EQ."" E1
      DW &&INDEX
INDEX .SET INDEX+1
      .GOTO LOOP
E1    MEND
```

```
HP 64873
 %*DEFINE(WORDS(P1)) (
 %MATCH(NEXT,REST) (%P1)
 %WHILE(%NES(%NEXT,) AND %NES(%NEXT,''))
 ( DW %NEXT
 %MATCH(NEXT,REST) (%REST)
 )
 )
```

Macro Calls

Sometimes, a MACRO call specifies a different number of actual parameters than formal parameters in the MACRO definition. acvtv20 records the number of formal parameters in a MACRO definition. It automatically handles the first two of three situations described below. The third situation usually requires a manual change.

1. If you specify fewer actual parameters than there are formal parameters, apv20 will error and not expand the macro. To prevent this, acvtv20 automatically generates additional null parameters on the macro call.

2. If you specify actual parameters and no formal parameters were declared, apv20 does not consume the actual parameter list and they eventually cause a syntax error. To prevent this, acvtv20 suppresses the actual parameter list.

3. If you specify more actual parameters than formal parameters, apv20 acts as follows: the value of the last formal parameter is equal to the value of its corresponding actual parameter concatenated with all the additional actual parameters and comma delimiters. Any reference to the last formal parameter will generate a different value than it did in the HP 64853 assembler. acvtv20 issues a warning in this case. You should either eliminate the extra actual parameters or rewrite the macro to preserve its original function.

acvtv20(1)
Command Syntax

Note

The program acvtv20 automatically performs most of the transformations described here. acvtv20 is an unsupported porting tool. acvtv20 is not a part of the 64873 product and is distributed at no cost. Hewlett-Packard makes no warranty on its quality or fitness for a particular purpose.

Name

acvtv20 - converts 8086 assembly programs from HP 64853 format to HP 64873 format

Synopsis

```
/usr/contrib/bin/acvtv20 [-dsw][-a
align]
          [-c context] [-h
suffix][file]
```

Description

acvtv20 translates assembly source programs from one dialect to another. It assumes the input file is a legal 70108, 70116, 70320, or 70330 assembly program for the HP 64853 assembler. The output may be assembled with the HP 64873 assembler.

acvtv20 does not translate "8086", "8088", "80186", "80188", "8089_86", "8089_88", or "80286" programs that were accepted by the HP 64853 assembler. Programs for these microprocessors are also not accepted by the HP 64873 assembler.

acvtv20 reads from standard input or the named file. It writes the translated assembly to standard output. It writes warnings about functional differences between the input and output to standard error.

acvtv20 supports a one-time porting of assembly programs from one product to another. The objective is to obtain the same (or functionally equal) bits from the HP 64873 assembler as from the HP 64853 assembler. acvtv20 changes directives, delimiters, operators, and so on to achieve this goal. However, because of differences between the two assemblers, this porting process cannot be entirely automatic or trivial.

acvtv20 makes two passes over its input file. The first pass builds a symbol table of certain identifiers (MACROS, externals, etc.) that will effect the translation; the second pass performs the translation.

acvtv20 may look at other files to supplement its symbol table. The **-c contextfile** option incorporates the definitions from **contextfile** in the present translation. Typically, a **contextfile** is a main, untranslated assembly module while the present file is an INCLUDE file of **contextfile.** Whenever acvtv20 encounters an INCLUDE directive (either in **contextfile** or the present input), it attempts to open the already translated include file and read its definitions. (See the -h suffix option for include file naming conventions.)

acvtv20 has a code substitution feature. It allows HP 64873 code to coexist with HP 64853 code in the same untranslated file. Refer to the section "Code Substitution" for more information.

acvtv20 was implemented with lex(1) and yacc(1). The source code is available in /usr/contrib/src/acvtv20/.

Options

-c context	Scan the context file (and translated INCLUDE files mentioned in it) for definitions to use when translating file. This option is useful when translating INCLUDE files. Specifying a context allows acvtv20 to accurately translate references to certain identifiers (MACROS, externals, etc.) that were defined in the main "context" file or its (translated) INCLUDE files.
-a align	Align is one of the HP 64873 align-types of BYTE, WORD, PARA, PAGE, INPAGE. Specify the align-type used in segment directives for relocatable segments. The

default align-type of BYTE duplicates the alignment behavior of the HP 64853 assembler. However, the HP 64873 assembler errors when an EVEN directive occurs within a BYTE aligned segment. If EVEN directives will be used, use the -a WORD option.

-d — (differences) acvtv20 writes pairs of input/output lines only when they are different. This output is not suitable for subsequent assembly.

-h suffix — Specifies the suffix (default .h) which is added to file names in INCLUDE directives to form the name of the "translated" include file. If the file name in the INCLUDE directive has a suffix (i.e. contains a period) then suffix replaces the original suffix. Otherwise, suffix is appended to the original file name.

For example, suppose an HP 64853 program contained the following directive.

```
INCLUDE file.H
```

acvtv20 would translate this to the following HP 64873 control.

```
$INCLUDE(file.h)
```

It would also assume that file.H had already been translated into file.h and attempt to read file.h before continuing with the present translation.

-s — (silent) Suppress warnings to standard error.

-w — (warn) Include warning messages (as comment lines) in the standard output following the appropriate translated line.

Files

/usr/contrib/bin/acvtv20

Executable file for assembly language porting tool.

/usr/contrib/src/acvtv20/*

Source code files and make file for assembly language porting tool.

See Also

HP 64873 V-Series Advanced Cross Assembler/Macro Preprocessor Reference Manual, Cross Assembler/Linker 8086/8088 Series and 70108/70116/70320, apv20(1), asv20(1), asm(1).

Diagnostics

acvtv20 returns non-zero if errors occur while performing I/O operations or while parsing the command line. Otherwise it returns zero.

Warning messages and the source lines which caused them are written to standard error.

Bugs

acvtv20 performs a limited set of transformations. Errors may occur when assembling the output. The object code from the 64853 assembly may not be the same as from the 64873 assembly.

acvtv20 may detect a syntax error reading a legal HP 64853 program. The syntax of the HP 64853 assembly language is irregular. Occasionally, a legal assembler statement will be unacceptable to the translator. acvtv20 will issue a warning when it detects a syntax error. The offending statement must be translated manually.

Old and New List

This section provides a side-by-side comparison of the HP 64853 constructs with the HP 64873 constructs. It is organized alphabetically by the HP 64853 keywords for the most part. There are some instances of general classifications, such as "Reserved Words" or "Operator Field." acvtv20 performs most of the conversions shown in this section.

ASCII

HP 64853	HP 64873
`ASCII 'ABC'`	`DB 'ABC'`

ALIGN

HP 64853	HP 64873
`label ALIGN`	`EVEN` `label:`

In HP 64873 assembly language, EVEN directives cause errors if they appear in segments with align-types of BYTE. Use an align-type of WORD if you want to use the EVEN directive. Any label may appear on the following line.

ASSUME

HP 64853	HP 64873
`ASSUME segreg:ORG segreg:abs_segname`	`ASSUME`

Most ASSUME directives need not be changed when moving to the HP 64873 assembler. However, when referring to absolute (for instance, ORGed) segment, you must do things differently. Briefly, when translating the ORG directive, you must create a named absolute segment using the SEGMENT directive. The ASSUME directive should then refer to this segment name. (See ORG for more information.)

COMN

HP 64853	HP 64873
label COMN	<prevproc> END <prevseg> ENDS COMN SEGMENT BYTE
COMMON	label:

Issue an ENDP to end the previous PROC, if necessary. Issue an ENDS directive to end the previous segment, if necessary. Any label must appear on the line following the directive.

DATA

HP 64853	HP 64873
label DATA	<prevproc> ENDP <prevseg> ENDS DATA SEGMENT BYTE
PUBLIC	label:

Issue an ENDP to end the previous PROC, if necessary. Issue an ENDS directive to end the previous segment, if necessary. Any label must appear on the line following the directive.

DBS

HP 64853	HP 64873
DBS <expr>	DB <expr> DUP (?)

DDS

HP 64853	HP 64873
DDS <expr>	DD <expr> DUP (?)

DWS

HP 64853	HP 64873

```
DWS <expr>          DW <expr> DUP (?)
```

<EOF>

HP 64853	HP 64873
<EOF>	<prevproc> ENDP <prevseg> ENDS END <EOF>

Add an END directive to the module if not already present. Also, issue ENDP and ENDS directives if necessary.

EQU

HP 64853	HP 64873
id EQU <expr>	id EQU <expr> %SET(id, <expr>)

If an EQU label is ever referenced in a *preprocessor expression* (IF, REPT, or SET), then you must define that label for the preprocessor using the %SET directive. References to id in preprocessor expressions must be changed to %id.

EXPAND

HP 64853	HP 64873
EXPAND	;EXPAND

The EXPAND function cannot be translated.

EXT

HP 64853	HP 64873
EXT id	EXTRN id:NEAR
EXT id type	EXTRN id:type
EXT segreg:id type	EXTRN id:type

The HP 64853 declaration "EXT segreg:id" causes an automatic segment override when id is used in a memory reference operand. The

HP 64873 assembler does not have an equal feature. Two approaches can be used to obtain the same code. You can either find all the references to id and add an explicit segment override to the operand when appropriate, or, place all the EXTRN directives with a particular associated segment register inside a segment. In the second case, you then must make sure an ASSUME directive is in effect for the proper segment register when the external identifiers are used.

GLB

HP 64853	HP 64873
GLB id	PUBLIC id

IF (Macro)

HP 64853	HP 64873
IF <expr> 0) THEN	%IF((<expr>) NE
lines	(lines
ELSE	) ELSE
lines	(lines
ENDIF	) FI

The constant expression in the IF directive must be calculated by the preprocessor. All identifiers that appear in the expression must be defined in %SET directives and referenced as %id.

INCLUDE Control

HP 64853	HP 64873
INCLUDE "file"	$INCLUDE(file)

LABEL Directive

HP 64853	HP 64873
id LABEL	id LABEL NEAR

Most LABEL directives need no changes. The exception is those which omit the implied NEAR type.

Label Field

HP 64853	HP 64873
`label: directive`	`label directive`
`label instruction`	`label: instruction`
`label macroname operands`	`label: %macroname(operands)`

Colons following labels are now significant. With the HP 64853 assembler, a colon following a label was optional. HP 64873 assembler prohibits a colon on a label for an assembler directive. HP64873 assembler requires a colon on a label for a blank line, an instruction, and a macro definition.

LIST

HP 64853	HP 64873
`LIST`	`$LIST`
`LIST n`	`$PAGELENGTH(n)`
`LIST`	

Note

PAGELENGTH is a primary control. It, and other HP 64873 primary controls, must be placed at the beginning of the file before any executable statements.

MACRO

HP 64853	HP 64873
`M1 MACRO &P1`	`%*DEFINE(M1(P1))`
`(`	
`anything &P1`	`anything %P1`
`MEND`	`)`

Translate MACRO definitions and references to macro parameters as shown.

MASK

HP 64853	HP 64873
MASK	;MASK

The MASK function cannot be translated. You must find any ASC directives which are affected and change the operands.

NAME

HP 64853	HP 64873
NAME	;NAME

The NAME function, which puts a comment in the relocatable object module, cannot be translated.

NOLIST

HP 64853	HP 64873
NOLIST	$NOLIST

NOWARN

HP 64853	HP 64873
NOWARN	;NOWARN

The NOWARN function cannot be translated.

Operator Field

HP 64853	HP 64873
.AN.	AND
.EQ.	EQ
.GE.	GE
.GT.	GT

.LE.	LE
.LT	LT
.NE.	NE
.NT.	NOT
.OR.	OR
.SL.	SHL
.SR.	SHR

HP 64853	**HP 64873**
#1234	1234

Remove the pound sign before literal operands.

Within a string, make the following translations.

- A quote (') becomes two quotes in series ('').
- To the macro preprocessor, the percent sign, left parenthesis, and right parenthesis are special characters. You should add a preprocessor escape sequence to percent and to unbalanced parentheses to avoid processor errors.
- HP 64873 string delimiters are different.

HP 64853	**HP 64873**
"string"	'string'
^string^	'string'
'string'	'string'

ORG

HP 64853	**HP 64873**
label ORG	<prevproc> ENDP <prevseg> ENDS abs_seg SEGMENT AT
PARA_VAL	ORG
OFFSET_VAL	label:

The HP 64853 ORG directive begins an absolute segment. Translate as follows.

- Issue an ENDP to end the previous PROC, if necessary.
- Issue an ENDS to end the previous segment, if necessary.
- The upper 16 bits of the ORG expression represents the segment value and the lower 16 bits represent the offset. You must extract the paragraph value and the offset manually because the HP 64873 does not do 32 bit arithmetic.
- Start an absolute segment, using the AT keyword, at the paragraph value.
- Set the offset using the ORG directive.
- Any label must follow the ORG to retain its original value. It is not necessary to create a new absolute segment for every ORG directive. Several ORGed sections (with the same segment values) may be combined. The HP 64873 ORG directive may be used to set the offset with the absolute segment.

PROC

HP 64853	HP 64873
label PROC type	<prevproc> ENDP label PROC type
PROC FAR	<prevproc> ENDP dummy PROC FAR

Issue an ENDP to end the previous procedure if necessary.

An unlabeled PROC directive is only useful for its effect on subsequent RET instructions. If the unlabeled PROC has type FAR, create a dummy PROC to retain the same behavior. This dummy procedure is unnecessary if the type of the unlabeled PROC is NEAR because HP 64873, by default, creates NEAR return instructions when RETs appear outside of any procedure.

PROG

HP 64853	HP 64873
`label PROG`	`<prevseg> ENDS`
	`PROG SEGMENT BYTE`
`PUBLIC`	
	`label:`

Issue an ENDS directive to end the previous segment if necessary. Any label must appear on the line following the directive.

REAL

HP 64853	HP 64873
`REAL number`	

The REAL directive cannot be translated. REAL is not useful because the byte order of its numbers is opposite the V-Series convention. Use DD, DQ, or DT to create useful real numbers.

Reserved Words

HP 64853	HP 64873
`TEST EQU 0`	`TESTx EQU 0`

HP 64873 assembler recognizes more reserved identifiers. HP 64853 assembly language allowed you to define labels that were spelled the same as either instruction mnemonics or assembler directives. HP 64873 assembler does not allow reserved word duplication. Change the spelling of identifiers that duplicate reserved words.

SPC

HP 64853	HP 64873
`SPC`	

The SPC function can only be translated into an equal number of empty source lines.

SKIP

HP 64853	HP 64873
SKIP	$EJECT

TITLE

HP 64853	HP 64873
TITLE 'string'	$TITLE(string)

WARN

HP 64853	HP 64873
WARN	;WARN

The WARN function cannot be translated.

* (Comment)

HP 64853	HP 64873
* comment	; comment
instr operand comment	instr operand
;comment	

HP 64853 sometimes allows comments to begin with an asterisk and sometimes does not require any delimiter. HP 64873 requires all comments to begin with semicolon. Blank lines do not need to begin with a semicolon.

INTEL2NEC(1)

Name

intel2nec - convert Intel 8086/186 assembly source into NEC V20 assembly source.

Synopsis

/usr/contrib/bin/intel2nec < intel_src_file > nec_src_file

Note

This utility program is unsupported. It is not a part of any HP product and is provided at no cost. Hewlett-Packard makes no warranty on its quality or fitness for a particular purpose.

Description

Intel2nec accepts input from standard input and writes a translated version to standard output. The input is expected to be Intel 8086/80186 assembly language. The intel2nec translator converts the 8086/80186 assembly instructions into their equivalent NEC V20 assembly instructions. The translator also converts those controls that are defined differently. The output should be usable in the HP 64873 V-Series assembler, asv20.

The intel2nec translator is implemented using the lex(1) program. The lex source file for the translator is shipped with the product, so the translator can be modified as needed to meet your needs. It is, however, fairly robust for normal use.

See Also

asv20(1), ldv20(1), V-Series Assembler/Linker Reference Manual.

Diagnostics

Intel2nec always returns zero.

Bugs

Intel2nec also translates comment text. While this is normally acceptable, it can create odd results.

F

V-Series Instructions in Hexadecimal Order

Table F-1. V-Series and 8087 Instructions

Hex Binary	MODRM Byte	Instruction	Parameters	Function
00 00000000	MOD REG R/M	ADD	EA,REG	BYTE ADD (REG) TO EA
01 00000001	MOD REG R/M	ADD	EA,REG	WORD ADD (REG) TO EA
02 00000010	MOD REG R/M	ADD	REG,EA	BYTE ADD (EA) TO REG
03 00000011	MOD REG R/M	ADD	REG,EA	WORD ADD (EA) TO REG
04 00000100		ADD	AL,DATA8	BYTE ADD DATA TO REG AL
05 00000101		ADD	AW,DATA16	WORD ADD DATA TO REG AW
06 00000110		PUSH	DS1	PUSH (DS1) ON STACK
07 00000111		POP	DS1	POP STACK TO REG DS1
08 00001000	MOD REG R/M	OR	EA,REG	BYTE OR (REG) TO EA
09 00001001	MOD REG R/M	OR	EA,REG	WORD OR (REG) TO EA
0A 00001010	MOD REG R/M	OR	REG,EA	BYTE OR (EA) TO REG
0B 00001011	MOD REG R/M	OR	REG,EA	WORD OR (EA) TO REG
0C 00001100		OR	AL,DATA8	BYTE OR DATA TO REG AL
0D 00001101		OR	AW,DATA16	WORD OR DATA TO REG AW

Hex Binary	MODRM Byte	Instruction	Parameters	Function
0E 00001110		PUSH	PS	PUSH (PS) ON STACK
0F 00001111	00010000	TEST1	EA8,CL	Test bit, store result
0F 00001111	00010001	TEST1	EA16,CL	Test bit, store result
0F 00001111	00010010	CLR1	EA8,CL	Clear bit
0F 00001111	00010011	CLR1	EA16,CL	Clear bit
0F 00001111	00010100	SET1	EA8,CL	Set bit
0F 00001111	00010101	SET1	EA16,CL	Set bit
0F 00001111	00010110	NOT1	EA8,CL	Invert bit
0F 00001111	00010111	NOT1	EA16,CL	Invert bit
0F 00001111	00011000	TEST1	EA8,DATA3	Test bit, store result
0F 00001111	00011001	TEST1	EA16,DATA4	Test bit, store result
0F 00001111	00011010	CLR1	EA8,DATA3	Clear bit
0F 00001111	00011011	CLR1	EA16,DATA4	Clear bit
0F 00001111	00011100	SET1	EA8,DATA3	Set bit
0F 00001111	00011101	SET1	EA16,DATA4	Set bit
0F 00001111	00011110	NOT1	EA8,DATA3	Invert bit
0F 00001111	00011111	NOT1	EA16,DATA4	Invert bit
0F 00001111	00100010	SUB4S		Subtract Nibble String
0F 00001111	00100101	MOVSPA		Move Stack Pointer
0F 00001111	00100110	CMP4S		Compare Nibble String
0F 00001111	00101000	ROL4	EA8	Rotate Nibble Left
0F 00001111	00101010	ROR4	EA8	Rotate Nibble Right

F-2 V-Series Instructions in Hexadecimal Order

Hex Binary	MODRM Byte	Instruction	Parameters	Function
0F 00001111	00110001	INS	REG8,REG8	Insert Bit Field
0F 00001111	00110011	EXT	REG8,REG8	Extract Bit Field
0F 00001111	00111001	INS	REG8,IMM4	Insert Bit Field
0F 00001111	00111011	EXT	REG8,IMM4	Extract Bit Field
0F 00001111	10010001	RETRBI		Return From Bank Switch
0F 00001111	10010010	FINT		Finish Interrupt
0F 00001111	10010100	TSKSW	REG16	Perform Task Switch
0F 00001111	10010101	MOVSPB	REG16	Move Stack Before Switch
0F 00001111	10011100	BTCLR	SFR,DATA3, DISP8	Branch if True and Clear
0F 00001111	10011101	BRKCS	REG16	Break with Context Switch
0F 00001111	10011110	STOP		Enter Stop Mode
0F 00001111	11100000	BRKXA	imm8	entered extended-memory mode
0F 00001111	11110000	RETXA	imm8	return from extended-memory mode
0F 00001111	11111111	BRKEM		Enter 8080 emulation mode
10 00010000	MOD REG R/M	ADDC	EA,REG	BYTE ADD (REG) W/ CARRY TO EA
11 00010001	MOD REG R/M	ADDC	EA,REG	WORD ADD (REG) W/ CARRY TO EA
12 00010010	MOD REG R/M	ADDC	REG,EA	BYTE ADD (EA) W/ CARRY TO REG
13 00010011	MOD REG R/M	ADDC	REG,EA	WORD ADD (EA) W/ CARRY TO REG

Hex Binary	MODRM Byte	Instruction	Parameters	Function
14 00010100		ADDC	AL,DATA8	BYTE ADD DATA W/CARRY TO REG AL
15 00010101		ADDC	AW,DATA16	WORD ADD DATA W/ CARRY TO REG AW
16 00010110		PUSH	SS	PUSH (SS) ON STACK
17 00010111		POP	SS	POP STACK TO REG SS
18 00011000	MOD REG R/M	SUBC	EA,REG	BYTE SUB (REG) W/ BORROW FROM EA
19 00011001	MOD REG R/M	SUBC	EA,REG	WORD SUB (REG) W/ BORROW FROM EA
1A 00011010	MOD REG R/M	SUBC	REG,EA	BYTE SUB (EA) W/ BORROW FROM REG
1B 00011011	MOD REG R/M	SUBC	REG,EA	WORD SUB (EA) W/ BORROW FROM REG
1C 00011100		SUBC	AL,DATA8	BYTE SUB DATA W/ BORROW FROM REG AL
1D 00011101		SUBC	AW,DATA16	WORD SUB DATA W/ BORROW FROM REG AW
1E 00011110		PUSH	DS0	PUSH (DS0) ON STACK
1F 00011111		POP	DS0	POP STACK TO REG DS0
20 00100000	MOD REG R/M	AND	EA,REG	BYTE AND (REG) TO EA
21 00100001	MOD REG R/M	AND	EA,REG	WORD AND (REG) TO EA
22 00100010	MOD REG R/M	AND	REG,EA	BYTE AND (EA) TO REG
23 00100011	MOD REG R/M	AND	REG,EA	WORD AND (EA) TO REG
24 00100100		AND	AL,DATA8	BYTE AND DATA TO REG AL

Hex Binary	MODRM Byte	Instruction	Parameters	Function
25 00100101		AND	AW,DATA16	WORD AND DATA TO REG AW
26 00100110		DS1:		SEGMENT OVERIDE W/ SEGMENT REG DS1
27 00100111		ADJ4A		DECIMAL ADJUST FOR ADD
28 00101000	MOD REG R/M	SUB	EA,REG	BYTE SUBTRACT (REG) FROM EA
29 00101001	MOD REG R/M	SUB	EA,REG	WORD SUBTRACT (REG) FROM EA
2A 00101010	MOD REG R/M	SUB	REG,EA	BYTE SUBTRACT (EA) FROM REG
2B 00101011	MOD REG R/M	SUB	REG,EA	WORD SUBTRACT (EA) FROM REG
2C 00101100		SUB	AL,DATA8	BYTE SUBTRACT DATA FROM REG AL
2D 00101101		SUB	AW,DATA16	WORD SUBTRACT DATA FROM REG AW
2E 00101110		PS:		SEGMENT OVERIDE W/ SEGMENT REG PS
2F 00101111		ADJ4S		DECIMAL ADJUST FOR SUBTRACT
30 00110000	MOD REG R/M	XOR	EA,REG	BYTE XOR (REG) TO EA
31 00110001	MOD REG R/M	XOR	EA,REG	WORD XOR (REG) TO EA
32 00110010	MOD REG R/M	XOR	REG,EA	BYTE XOR (EA) TO REG
33 00110011	MOD REG R/M	XOR	REG,EA	WORD XOR (EA) TO REG
34 00110100		XOR	AL,DATA8	BYTE XOR DATA TO REG AL

Hex Binary	MODRM Byte	Instruction	Parameters	Function
35 00110101		XOR	AW,DATA16	WORD XOR DATA TO REG AW
36 00110110		SS:		SEGMENT OVERIDE W/ SEGMENT REG SS
37 00110111		ADJBA		ASCII ADJUST FOR ADD
38 00111000	MOD REG R/M	CMP	EA,REG	BYTE COMPARE (EA) WITH (REG)
39 00111001	MOD REG R/M	CMP	EA,REG	WORD COMPARE (EA) WITH (REG)
3A 00111010	MOD REG R/M	CMP	REG,EA	BYTE COMPARE (REG) WITH (EA)
3B 00111011	MOD REG R/M	CMP	REG,EA	WORD COMPARE (REG) WITH (EA)
3C 00111100		CMP	AL,DATA8	BYTE COMPARE DATA WITH (AL)
3D 00111101		CMP	AW,DATA16	WORD COMPARE DATA WITH (AW)
3E 00111110		DS0:		SEGMENT OVERIDE W/ SEGMENT REG DS0
3F 00111111		ADJBS		ASCII ADJUST FOR SUBTRACT
40 01000000		INC	AW	INCREMENT (AW)
41 01000001		INC	CW	INCREMENT (CW)
42 01000010		INC	DW	INCREMENT (DW)
43 01000011		INC	BW	INCREMENT (BW)
44 01000100		INC	SP	INCREMENT (SP)

Hex Binary	MODRM Byte	Instruction	Parameters	Function
45 01000101		INC	BP	INCREMENT (BP)
46 01000110		INC	IX	INCREMENT (IX)
47 01000111		INC	IY	INCREMENT (IY)
48 01001000		DEC	AW	DECREMENT (AW)
49 01001001		DEC	CW	DECREMENT (CW)
4A 01001010		DEC	DW	DECREMENT (DW)
4B 01001011		DEC	BW	DECREMENT (BW)
4C 01001100		DEC	SP	DECREMENT (SP)
4D 01001101		DEC	BP	DECREMENT (BP)
4E 01001110		DEC	IX	DECREMENT (IX)
4F 01001111		DEC	IY	DECREMENT (IY)
50 01010000		PUSH	AW	PUSH (AW) ON STACK
51 01010001		PUSH	CW	PUSH (CW) ON STACK
52 01010010		PUSH	DW	PUSH (DW) ON STACK
53 01010011		PUSH	BW	PUSH (BW) ON STACK
54 01010100		PUSH	SP	PUSH (SP) ON STACK
55 01010101		PUSH	BP	PUSH (BP) ON STACK
56 01010110		PUSH	IX	PUSH (IX) ON STACK
57 01010111		PUSH	IY	PUSH (IY) ON STACK
58 01011000		POP	AW	POP STACK TO REG AW
59 01011001		POP	CW	POP STACK TO REG CW
5A 01011010		POP	DW	POP STACK TO REG DW

Hex Binary	MODRM Byte	Instruction	Parameters	Function
5B 01011011		POP	BW	POP STACK TO REG BW
5C 01011100		POP	SP	POP STACK TO REG SP
5D 01011101		POP	BP	POP STACK TO REG BP
5E 01011110		POP	IX	POP STACK TO REG IX
5F 01011111		POP	IY	POP STACK TO REG IY
60 01100000		PUSH	R	PUSH ALL DATA
61 01100001		POP	R	POP ALL DATA
62 01100010	MOD REG R/M	CHKIND	REG,EA	CHECK INDEX IN REG AGAINST BOUNDS AT EA
63 01100011		(not used)		
64 01100100		(not used)		
65 01100101		(not used)		
66 01100110		FP02		Floating Point Instructions
67 01100111		(not used)		
68 01101000		PUSH	DATA16	PUSH WORD DATA ON STACK
69 01101001	MOD REG R/M	MUL	REG,EA, DATA16	MULTIPLY (EA) BY WORD DATA; SIGNED
6A 01101010		PUSH	DATA8	PUSH BYTE DATA ON STACK; SIGN-EXTEND
6B 01101011	MOD REG R/M	MUL	REG,EA, DATA8	MULTIPLY (EA) BY BYTE DATA; SIGNED
6C 01101100		INM	DST8	BYTE INPUT
6D 01101101		INM	DST16	WORD INPUT

Hex Binary	MODRM Byte	Instruction	Parameters	Function
6E 01101110		OUTM	DST8	BYTE OUTPUT
6F 01101111		OUTM	DST16	WORD OUTPUT
70 01110000		BV	DISP8	JUMP ON OVERFLOW
71 01110001		BNV	DISP8	JUMP ON NOT OVERFLOW
72 01110010		BC/BL	DISP8	JUMP ON BELOW/NOT ABOVE OR EQUAL
73 01110011		BNC/BNL	DISP8	JUMP ON NOT BELOW/ABOVE OR EQUAL
74 01110100		BE/BZ	DISP8	JUMP ON EQUAL/ZERO
75 01110101		BNE/BNZ	DISP8	JUMP ON NOT EQUAL/NOT ZERO
76 01110110		BNH	DISP8	JUMP ON BELOW OR EQUAL/NOT ABOVE
77 01110111		BH	DISP8	JUMP ON NOT BELOW OR EQUAL/ABOVE
78 01111000		BN	DISP8	JUMP ON SIGN
79 01111001		BP	DISP8	JUMP ON NOT SIGN
7A 01111010		BPE	DISP8	JUMP ON PARITY/PARITY EVEN
7B 01111011		BPO	DISP8	JUMP ON NOT PARITY/PARITY ODD
7C 01111100		BLT	DISP8	JUMP ON LESS/NOT GREATER OR EQUAL
7D 01111101		BGE	DISP8	JUMP ON NOT LESS/GREATER OR EQUAL

Hex Binary	MODRM Byte	Instruction	Parameters	Function
7E 01111110		BLE	DISP8	JUMP ON LESS OR EQUAL/NOT GREATER
7F 01111111		BGT	DISP8	JUMP ON NOT LESS OR EQUAL/GREATER
80 10000000	MOD 000 R/M	ADD	EA,DATA8	BYTE ADD DATA TO EA
80 10000000	MOD 001 R/M	OR	EA,DATA8	BYTE OR DATA TO EA
80 10000000	MOD 010 R/M	ADDC	EA,DATA8	BYTE ADD DATA W/CARRY TO EA
80 10000000	MOD 011 R/M	SUBC	EA,DATA8	BYTE SUB DATA W/BORROW FROM EA
80 10000000	MOD 100 R/M	AND	EA,DATA8	BYTE AND DATA TO EA
80 10000000	MOD 101 R/M	SUB	EA,DATA8	BYTE SUBTRACT DATA FROM EA
80 10000000	MOD 110 R/M	XOR	EA,DATA8	BYTE XOR DATA TO EA
80 10000000	MOD 111 R/M	CMP	EA,DATA8	BYTE COMPARE DATA WITH (EA)
81 10000001	MOD 000 R/M	ADD	EA,DATA16	WORD ADD DATA TO EA
81 10000001	MOD 001 R/M	OR	EA,DATA16	WORD OR DATA TO EA
81 10000001	MOD 010 R/M	ADDC	EA,DATA16	WORD ADD DATA W/CARRY TO EA
81 10000001	MOD 011 R/M	SUBC	EA,DATA16	WORD SUB DATA W/ BORROW FROM EA
81 10000001	MOD 100 R/M	AND	EA,DATA16	WORD AND DATA TO EA
81 10000001	MOD 101 R/M	SUB	EA,DATA16	WORD SUBTRACT DATA FROM EA
81 10000001	MOD 110 R/M	XOR	EA,DATA16	WORD XOR DATA TO EA

Hex Binary	MODRM Byte	Instruction	Parameters	Function
81 10000001	MOD 111 R/M	CMP	EA,DATA16	WORD COMPARE DATA WITH (EA)
82 10000010	MOD 000 R/M	ADD	EA,DATA8	BYTE ADD DATA TO EA
82 10000010	MOD 001 R/M	(not used)		
82 10000010	MOD 010 R/M	ADDC	EA,DATA8	BYTE ADD DATA W/ CARRY TO EA
82 10000010	MOD 011 R/M	SUBC	EA,DATA8	BYTE SUB DATA W/ BORROW FROM EA
82 10000010	MOD 100 R/M	(not used)		
82 10000010	MOD 101 R/M	SUB	EA,DATA8	BYTE SUBTRACT DATA FROM EA
82 10000010	MOD 110 R/M	(not used)		
82 10000010	MOD 111 R/M	CMP	EA,DATA8	BYTE COMPARE DATA WITH (EA)
83 10000011	MOD 000 R/M	ADD	EA,DATA8	WORD ADD DATA TO EA
83 10000011	MOD 001 R/M	(not used)		
83 10000011	MOD 010 R/M	ADDC	EA,DATA8	WORD ADD DATA W/ CARRY TO EA
83 10000011	MOD 011 R/M	SUBC	EA,DATA8	WORD SUB DATA W/ BORROW FROM EA
83 10000011	MOD 100 R/M	(not used)		
83 10000011	MOD 101 R/M	SUB	EA,DATA8	WORD SUBTRACT DATA FROM EA
83 10000011	MOD 110 R/M	(not used)		
83 10000011	MOD 111 R/M	CMP	EA,DATA8	WORD COMPARE DATA WITH (EA)

Hex Binary	MODRM Byte	Instruction	Parameters	Function
84 10000100	MOD REG R/M	TEST	EA,REG	BYTE TEST (EA) WITH (REG)
85 10000101	MOD REG R/M	TEST	EA,REG	WORD TEST (EA) WITH (REG)
86 10000110	MOD REG R/M	XCH	REG,EA	BYTE EXCHANGE (REG) WITH (EA)
87 10000111	MOD REG R/M	XCH	REG,EA	WORD EXCHANGE (REG) WITH (EA)
88 10001000	MOD REG R/M	MOV	EA,REG	BYTE MOVE (REG) TO EA
89 10001001	MOD REG R/M	MOV	EA,REG	WORD MOVE (REG) TO EA
8A 10001010	MOD REG R/M	MOV	REG,EA	BYTE MOVE (EA) TO REG
8B 10001011	MOD REG R/M	MOV	REG,EA	WORD MOVE (EA) TO REG
8C 10001100	MOD 0SR R/M	MOV	EA,SR	WORD MOVE (SEGMENT REG SR) TO EA
8C 10001100	MOD 1-- R/M	(not used)		
8D 10001101	MOD REG R/M	LDEA	REG,EA	LOAD EFFECTIVE ADDRESS OF EA TO REG
8E 10001110	MOD 0SR R/M	MOV	SR,EA	WORD MOVE (EA) TO SEGMENT REG SR
8E 10001110	MOD -- R/M	(not used)		
8F 10001111	MOD 000 R/M	POP	EA	POP STACK TO EA
8F 10001111	MOD 001 R/M	(not used)		
8F 10001111	MOD 010 R/M	(not used)		
8F 10001111	MOD 011 R/M	(not used)		
8F 10001111	MOD 100 R/M	(not used)		
8F 10001111	MOD 101 R/M	(not used)		

Hex Binary	MODRM Byte	Instruction	Parameters	Function
8F 10001111	MOD 110 R/M	(not used)		
8F 10001111	MOD 111 R/M	(not used)		
90 10010000		XCH	AW,AW	EXCHANGE (AW) WITH (AW)
91 10010001		XCH	AW,CW	EXCHANGE (AW) WITH (CW)
92 10010010		XCH	AW,DW	EXCHANGE (AW) WITH (DW)
93 10010011		XCH	AW,BW	EXCHANGE (AW) WITH (BW)
94 10010100		XCH	AW,SP	EXCHANGE (AW) WITH (SP)
95 10010101		XCH	AW,BP	EXCHANGE (AW) WITH (BP)
96 10010110		XCH	AW,IX	EXCHANGE (AW) WITH (IX)
97 10010111		XCH	AW,IY	EXCHANGE (AW) WITH (IY)
98 10011000		CVTBW		BYTE CONVERT (AL) TO WORD (AW)
99 10011001		CVTWL		WORD CONVERT (AW) TO DOUBLE WORD
9A 10011010		CALL	DISP16,SEG16	DIRECT INTER SEGMENT CALL
9B 10011011		POLL		WAIT FOR TEST SIGNAL
9C 10011100		PUSH	PSW	PUSH FLAGS ON STACK
9D 10011101		POP	PSW	POP STACK TO FLAGS
9E 10011110		MOV	PSW,AH	STORE (AH) INTO FLAGS
9F 10011111		MOV	AH,PSW	LOAD REG AH WITH FLAGS
A0 10100000		MOV	AL,ADDR16	BYTE MOVE (ADDR) TO REG AL

Hex Binary	MODRM Byte	Instruction	Parameters	Function
A1 10100001		MOV	AW,ADDR16	WORD MOVE (ADDR) TO REG AW
A2 10100010		MOV	ADDR16,AL	BYTE MOVE (AL) TO ADDR
A3 10100011		MOV	ADDR16,AW	WORD MOVE (AW) TO ADDR
A4 10100100		MOVBK	DST8,SRC8	BYTE MOVE, STRING OP
A5 10100101		MOVBK	DST16,SRC16	WORD MOVE, STRING OP
A6 10100110		CMPBK	IXPTR,IYPTR	COMPARE BYTE, STRING OP
A7 10100111		CMPBK	IXPTR,IYPTR	COMPARE WORD, STRING OP
A8 10101000		TEST	AL,DATA8	BYTE TEST (AL) WITH DATA
A9 10101001		TEST	AW,DATA16	WORD TEST (AW) WITH DATA
AA 10101010		STM	DST8	BYTE STORE, STRING OP
AB 10101011		STM	DST16	WORD STORE, STRING OP
AC 10101100		LDM	SRC8	BYTE LOAD, STRING OP
AD 10101101		LDM	SRC16	WORD LOAD, STRING OP
AE 10101110		CMPM	IYPTR8	BYTE SCAN, STRING OP
AF 10101111		CMPM	IYPTR16	WORD SCAN, STRING OP
B0 10110000		MOV	AL,DATA8	BYTE MOVE DATA TO REG AL
B1 10110001		MOV	CL,DATA8	BYTE MOVE DATA TO REG CL
B2 10110010		MOV	DL,DATA8	BYTE MOVE DATA TO REG DL

Hex Binary	MODRM Byte	Instruction	Parameters	Function
B3 10110011		MOV	BL,DATA8	BYTE MOVE DATA TO REG BL
B4 10110100		MOV	AH,DATA8	BYTE MOVE DATA TO REG AH
B5 10110101		MOV	CH,DATA8	BYTE MOVE DATA TO REG CH
B6 10110110		MOV	DH,DATA8	BYTE MOVE DATA TO REG DH
B7 10110111		MOV	BH,DATA8	BYTE MOVE DATA TO REG BH
B8 10111000		MOV	AW,DATA16	WORD MOVE DATA TO REG AW
B9 10111001		MOV	CW,DATA16	WORD MOVE DATA TO REG CW
BA 10111010		MOV	DW,DATA16	WORD MOVE DATA TO REG DW
BB 10111011		MOV	BW,DATA16	WORD MOVE DATA TO REG BW
BC 10111100		MOV	SP,DATA16	WORD MOVE DATA TO REG SP
BD 10111101		MOV	BP,DATA16	WORD MOVE DATA TO REG BP
BE 10111110		MOV	IX,DATA16	WORD MOVE DATA TO REG IX
BF 10111111		MOV	IY,DATA16	WORD MOVE DATA TO REG IY
C0 11000000	MOD 000 R/M	ROL	EA,DATA8	BYTE ROTATE EA LEFT DATA8 BITS

Hex Binary	MODRM Byte	Instruction	Parameters	Function
C0 11000000	MOD 001 R/M	ROR	EA,DATA8	BYTE ROTATE EA RIGHT DATA8 BITS
C0 11000000	MOD 010 R/M	ROLC	EA,DATA8	BYTE ROTATE EA LEFT THRU CARRY DATA8 BITS
C0 11000000	MOD 011 R/M	RORC	EA,DATA8	BYTE ROTATE EA RIGHT THRU CARRY DATA8 BITS
C0 11000000	MOD 100 R/M	SHL	EA,DATA8	BYTE SHIFT EA LEFT DATA8 BITS
C0 11000000	MOD 101 R/M	SHR	EA,DATA8	BYTE SHIFT EA RIGHT DATA8 BITS
C0 11000000	MOD 110 R/M	(not used)		
C0 11000000	MOD 111 R/M	SHRA	EA,DATA8	BYTE SHIFT SIGNED EA RIGHT DATA8 BITS
C1 11000001	MOD 000 R/M	ROL	EA,DATA8	WORD ROTATE EA LEFT DATA8 BITS
C1 11000001	MOD 001 R/M	ROR	EA,DATA8	WORD ROTATE EA RIGHT DATA8 BITS
C1 11000001	MOD 010 R/M	ROLC	EA,DATA8	WORD ROTATE EA LEFT THRUCARRY DATA8 BITSCARRY DATA8 BITS
C1 11000001	MOD 011 R/M	RORC	EA,DATA8	WORD ROTATE EA RIGHT THRU CARRY DATA8 BITS
C1 11000001	MOD 100 R/M	SHL	EA,DATA8	WORD SHIFT EA LEFT DATA8 BITS
C1 11000001	MOD 101 R/M	SHR	EA,DATA8	WORD SHIFT EA RIGHT DATA8 BITS
C1 11000001	MOD 110 R/M	(not used)		

Hex Binary	MODRM Byte	Instruction	Parameters	Function
C1 11000001	MOD 111 R/M	SHRA	EA,DATA8	WORD SHIFT SIGNED EA RIGHT DATA8 BITS
C2 11000010		RET	DATA16	INTRA SEGMENT RETURN
C3 11000011		RET		INTRA SEGMENT RETURN
C4 11000100	MOD REG R/M	MOV	DS1,REG, EA	WORD LOAD REG AND SEGMENT REG DS1
C5 11000101	MOD REG R/M	MOV	DS0,REG,EA	WORD LOAD REG AND SEGMENT REG DS0
C6 11000110	MOD 000 R/M	MOV	EA,DATA8	BYTE MOVE DATA TO EA
C6 11000110	MOD 001 R/M	(not used)		
C6 11000110	MOD 010 R/M	(not used)		
C6 11000110	MOD 011 R/M	(not used)		
C6 11000110	MOD 100 R/M	(not used)		
C6 11000110	MOD 101 R/M	(not used)		
C6 11000110	MOD 110 R/M	(not used)		
C6 11000110	MOD 111 R/M	(not used)		
C7 11000111	MOD 000 R/M	MOV	EA,DATA16	WORD MOVE DATA TO EA
C7 11000111	MOD 001 R/M	(not used)		
C7 11000111	MOD 010 R/M	(not used)		
C7 11000111	MOD 011 R/M	(not used)		
C7 11000111	MOD 100 R/M	(not used)		
C7 11000111	MOD 101 R/M	(not used)		
C7 11000111	MOD 110 R/M	(not used)		

Hex Binary	MODRM Byte	Instruction	Parameters	Function
C7 11000111	MOD 111 R/M	(not used)		
C8 11001000		PREPARE	DATA16, DATA8	PERFORM ENTER SEQUENCE
C9 11001001		DISPOSE		PERFORM LEAVE SEQUENCE
CA 11001010		RET	DATA16	INTER SEGMENT RETURN
CB 11001011		RET		INTER SEGMENT RETURN
CC 11001100		BRK	3	TYPE 3 INTERRUPT
CD 11001101		BRK	TYPE	TYPED INTERRUPT
CE 11001110		BRKV		INTERRUPT ON OVERFLOW
CF 11001111		RETI		RETURN FROM INTERRUPT
D0 11010000	MOD 000 R/M	ROL	EA,1	BYTE ROTATE EA LEFT 1 BIT
D0 11010000	MOD 001 R/M	ROR	EA,1	BYTE ROTATE EA RIGHT 1 BIT
D0 11010000	MOD 010 R/M	ROLC	EA,1	BYTE ROTATE EA LEFT THRU CARRY 1 BIT
D0 11010000	MOD 011 R/M	RORC	EA,1	BYTE ROTATE EA RIGHT THRU CARRY 1 BIT
D0 11010000	MOD 100 R/M	SHL	EA,1	BYTE SHIFT EA LEFT 1 BIT
D0 11010000	MOD 101 R/M	SHR	EA,1	BYTE SHIFT EA RIGHT 1 BIT
D0 11010000	MOD 110 R/M	(not used)		
D0 11010000	MOD 111 R/M	SHRA	EA,1	BYTE SHIFT SIGNED EA RIGHT 1 BIT
D1 11010001	MOD 000 R/M	ROL	EA,1	WORD ROTATE EA LEFT 1 BIT

Hex Binary	MODRM Byte	Instruction	Parameters	Function
D1 11010001	MOD 001 R/M	ROR	EA,1	WORD ROTATE EA RIGHT 1 BIT
D1 11010001	MOD 010 R/M	ROLC	EA,1	WORD ROTATE EA LEFT THRU CARRY 1 BIT
D1 11010001	MOD 011 R/M	RORC	EA,1	WORD ROTATE EA RIGHT THRU CARRY 1 BIT
D1 11010001	MOD 100 R/M	SHL	EA,1	WORD SHIFT EA LEFT 1 BIT
D1 11010001	MOD 101 R/M	SHR	EA,1	WORD SHIFT EA RIGHT 1 BIT
D1 11010001	MOD 110 R/M	(not used)		
D1 11010001	MOD 111 R/M	SHRA	EA,1	WORD SHIFT SIGNED EA RIGHT 1 BIT
D2 11010010	MOD 000 R/M	ROL	EA,CL	BYTE ROTATE EA LEFT (CL) BITS
D2 11010010	MOD 001 R/M	ROR	EA,CL	BYTE ROTATE EA RIGHT (CL) BITS
D2 11010010	MOD 010 R/M	ROLC	EA,CL	BYTE ROTATE EA LEFT THRU CARRY (CL) BITS
D2 11010010	MOD 011 R/M	RORC	EA,CL	BYTE ROTATE EA RIGHT THRU CARRY (CL) BITS
D2 11010010	MOD 100 R/M	SHL	EA,CL	BYTE SHIFT EA LEFT (CL) BITS
D2 11010010	MOD 101 R/M	SHR	EA,CL	BYTE SHIFT EA RIGHT (CL) BITS
D2 11010010	MOD 110 R/M	(not used)		
D2 11010010	MOD 111 R/M	SHRA	EA,CL	BYTE SHIFT SIGNED EA RIGHT (CL) BITS

Hex Binary	MODRM Byte	Instruction	Parameters	Function
D3 11010011	MOD 000 R/M	ROL	EA,CL	WORD ROTATE EA LEFT (CL) BITS
D3 11010011	MOD 001 R/M	ROR	EA,CL	WORD ROTATE EA RIGHT (CL) BITS
D3 11010011	MOD 010 R/M	ROLC	EA,CL	WORD ROTATE EA LEFT THRU CARRY (CL) BITS
D3 11010011	MOD 011 R/M	RORC	EA,CL	WORD ROTATE EA RIGHT THRU CARRY (CL) BITS
D3 11010011	MOD 100 R/M	SHL	EA,CL	WORD SHIFT EA LEFT (CL) BITS
D3 11010011	MOD 101 R/M	SHR	EA,CL	WORD SHIFT EA RIGHT (CL) BITS
D3 11010011	MOD 110 R/M	(not used)		
D3 11010011	MOD 111 R/M	SHRA	EA,CL	WORD SHIFT SIGNED EA RIGHT (CL) BITS
D4 11010100	00001010	CVTBD		ASCII ADJUST FOR MULTIPLY
D5 11010101	00001010	CVTDB		ASCII ADJUST FOR DIVIDE
D6 11010110		(not used)		
D7 11010111		TRANS	TABLE	TRANSLATE USING (BW)
D8 11011---	MOD --- R/M	ESC	EA	ESCAPE TO EXTERNAL DEVICE
D8 11011000	MOD 000 R/M	FADD	Short-real	ADD 4-BYTE EA TO ST
D8 11011000	MOD 001 R/M	FMUL	Short-real	MULTIPLY ST BY 4-BYTE EA
D8 11011000	MOD 010 R/M	FCOM	Short-real	COMPARE 4-BYTE EA WITH ST

Hex Binary	MODRM Byte	Instruction	Parameters	Function
D8 11011000	MOD 011 R/M	FCOMP	Short-real	COMPARE 4-BYTE EA WITH ST AND POP
D8 11011000	MOD 100 R/M	FSUB	Short-real	SUBTRACT 4-BYTE EA FROM ST
D8 11011000	MOD 101 R/M	FSUBR	Short-real	SUBTRACT ST FROM 4-BYTE EA
D8 11011000	MOD 110 R/M	FDIV	Short-real	DIVIDE ST BY 4-BYTE EA
D8 11011000	MOD 111 R/M	FDIVR	Short-real	DIVIDE 4-BYTE EA BY ST
D8 11011000	1 1 000 (i)	FADD	ST,ST(i)	ADD ELEMENT TO ST
D8 11011000	1 1 001 (i)	FMUL	ST,ST(i)	MULTIPLY ST BY ELEMENT
D8 11011000	1 1 010 (i)	FCOM	ST(i)	COMPARE ST(i) WITH ST
D8 11011000	1 1 011 (i)	FCOMP	ST(i)	COMPARE ST(i) WITH ST AND POP
D8 11011000	1 1 100 (i)	FSUB	ST,ST(i)	SUBTRACT ELEMENT FROM ST
D8 11011000	1 1 101 (i)	FSUBR	ST,ST(i)	SUBTRACT ST FROM STACK ELEMENT
D8 11011000	1 1 110 (i)	FDIV	ST,ST(i)	DIVIDE ST BY ELEMENT
D8 11011000	1 1 111 (i)	FDIVR	ST,ST(i)	DIVIDE ST(i) BY ST
D9 11011001	MOD 000 R/M	FLD	Short-real	PUSH 4-BYTE EA TO ST
D9 11011001	MOD 001 R/M	(not used)		
D9 11011001	MOD 010 R/M	FST	Short-real	STORE 4-BYTE REAL TO EA
D9 11011001	MOD 011 R/M	FSTP	Short-real	STORE 4-BYTE REAL TO EA AND POP

Hex Binary	MODRM Byte	Instruction	Parameters	Function
D9 11011001	MOD 100 R/M	FLDENV	14 BYTES	LOAD 8087 ENVIRONMENT FROM EA
D9 11011001	MOD 101 R/M	FLDCW	2-BYTES	LOAD CONTROL WORD FROM EA
D9 11011001	MOD 110 R/M	FSTENV	14-BYTES	STORE 8087 ENVIRONMENT INTO EA
D9 11011001	MOD 111 R/M	FSTCW	2-BYTES	STORE CONTROL WORD INTO EA
D9 11011001	1 1 000 (i)	FLD	ST(i)	PUSH ST(i) ONTO ST
D9 11011001	1 1 001 (i)	FXCH	ST(i)	EXCHANGE ST AND ST(i)
D9 11011001	1 1 010 000	FNOP		STORE ST IN ST
D9 11011001	1 1 010 001	(not used)		
D9 11011001	1 1 010 01-	(not used)		
D9 11011001	1 1 010 1--	(not used)		
D9 11011001	1 1 011 (i)	*(1)		
D9 11011001	1 1 100 000	FCHS		CHANGE SIGN OF ST
D9 11011001	1 1 100 001	FABS		TAKE ABSOLUTE VALUE OF ST
D9 11011001	1 1 100 01-	(not used)		
D9 11011001	1 1 100 100	FTST		TEST ST AGAINST 0.0
D9 11011001	1 1 100 101	FXAM		EXAMINE ST AND REPORT CONDITION CODE
D9 11011001	1 1 100 11-	(not used)		
D9 11011001	1 1 101 000	FLD1		PUSH +1.0 TO ST

Hex Binary	MODRM Byte	Instruction	Parameters	Function
D9 11011001	1 1 101 001	FLDL2T		PUSH log 2 10 TO ST
D9 11011001	1 1 101 010	FLDL2E		PUSH log 2 e TO ST
D9 11011001	1 1 101 011	FLDPI		PUSH Pi TO ST
D9 11011001	1 1 101 100	FLDLG2		PUSH log 10 2 TO ST
D9 11011001	1 1 101 101	FLDLN2		PUSH log e 2 TO ST
D9 11011001	1 1 101 110	FLDZ		PUSH ZERO TO ST
D9 11011001	1 1 101 111	(not used)		
D9 11011001	1 1 110 000	F2XM1		CALCULATE 2 x - 1
D9 11011001	1 1 110 001	FYL2X		CALCULATE FUNCTION Y*log 2 X
D9 11011001	1 1 110 010	FPTAN		CALCULATE TAN OF 0 AS A RATIO
D9 11011001	1 1 110 011	FPATAN		CALCULATE ARCTAN OF 0
D9 11011001	1 1 110 100	FXTRACT		EXTRACT EXPONENT AND SIGNIFICAND FROM ST VALUE
D9 11011001	1 1 110 101	(not used)		
D9 11011001	1 1 110 110	FDECSTP		DECREMENT STACK POINTER IN STATUS WORD
D9 11011001	1 1 110 111	FINCSTP		INCREMENT STACK POINTER IN STATUS WORD
D9 11011001	1 1 111 000	FPREM		MODULO DIVISION OF ST BY ST(1)
D9 11011001	1 1 110 001	FYL2XP1		CALCULATE VALUE OF Y*log 2 (X + 1)

Hex Binary	MODRM Byte	Instruction	Parameters	Function
D9 11011001	1 1 111 010	FSQRT		CALCULATE SQUARE ROOT OF ST
D9 11011001	1 1 111 011	(not used)		
D9 11011001	1 1 111 100	FRNDINT		ROUND ST TO INTEGER
D9 11011001	1 1 111 101	FSCALE		ADD ST(1) TO EXPONENT OF ST
D9 11011001	1 1 111 11-	(not used)		
DA 11011010	MOD 000 R/M	RIADD	Short-integer	ADD 4-BYTE INTEGER EA TO ST
DA 11011010	MOD 001 R/M	FIMUL	Short-integer	MULTIPLY ST BY 4-BYTE INTEGER EA
DA 11011010	MOD 010 R/M	FICOM	Short-integer	CONVERT 4-BYTE INTEGER EA, AND COMPARE WITH ST
DA 11011010	MOD 011 R/M	FICOMP	Short-integer	CONVERT 4-BYTE INTEGER EA, COMPARE WITH ST, POP
DA 11011010	MOD 100 R/M	FISUB	Short-integer	SUBTRACT 4-BYTE INTEGER EA FROM ST
DA 11011010	MOD 101 R/M	FISUBR	Short-integer	SUBTRACT ST FROM 4-BYTE INTEGER EA
DA 11011010	MOD 110 R/M	FIDIV	Short-integer	DIVIDE ST BY 4-BYTE INTEGER EA
DA 11011010	MOD 111 R/M	FIDIVR	Short-integer	DIVIDE 4-BYTE INTEGER EA BY ST
DA 11011010	1 1 -- ---	(not used)		
DB 11011011	MOD 000 R/M	FILD	Short-integer	
DB 11011011	MOD 001 R/M	(not used)		

Hex Binary	MODRM Byte	Instruction	Parameters	Function
DB 11011011	MOD 010 R/M	FIST	Short-integer	STORE ROUNDED ST IN 4-BYTE INTEGER EA
DB 11011011	MOD 011 R/M	FISTP	Short-integer	STORE ROUNDED ST IN 4-BYTE INTEGER EA, POP
DB 11011011	MOD 100 R/M	(not used)		
DB 11011011	MOD 101 R/M	FLD	Temp-real	PUSH 10-BYTE EA ONTO ST
DB 11011011	MOD 110 R/M	Reserved		
DB 11011011	MOD 111 R/M	FSTP	Temp-real	STORE ST INTO 10-BYTE EA, POP
DB 11011011	1 1 0-- ---	Reserved		
DB 11011011	1 1 100 000	FENI		ENABLE INTERRUPT
DB 11011011	1 1 100 001	FDISI		DISABLE INTERRUPTS
DB 11011011	1 1 100 010	FCLEX		CLEAR EXCEPTIONS
DB 11011011	1 1 100 011	FINIT		INITIALIZE PROCESSOR
DB 11011011	1 1 100 1--	Reserved		
DB 11011011	1 1 101 ---	Reserved		
DB 11011011	1 1 11- ---	Reserved		
DC 11011100	MOD 000 R/M	FADD	Long-real	ADD 8-BYTE EA TO ST
DC 11011100	MOD 001 R/M	FMUL	Long-real	MULTIPLY ST BY 8-BYTE EA
DC 11011100	MOD 010 R/M	FCOM	Long-real	COMPARE ST WITH 8-BYTE EA
DC 11011100	MOD 011 R/M	FCOMP	Long-real	COMPARE ST WITH 8-BYTE EA

Hex Binary	MODRM Byte	Instruction	Parameters	Function
DC 11011100	MOD 100 R/M	FSUB	Long-real	SUBTRACT 8-BYTE EA FROM ST
DC 11011100	MOD 101 R/M	FSUBR	Long-real	SUBTRACT ST FROM 8-BYTE EA
DC 11011100	MOD 110 R/M	FDIV	Long-real	DIVIDE ST BY 8-BYTE EA
DC 11011100	MOD 111 R/M	FDIVR	Long-real	DIVIDE 8-BYTE EA BY ST
DC 11011100	1 1 000 (i)	FADD	ST(i), ST	ADD ST TO ELEMENT
DC 11011100	1 1 001 (i)	FMUL	ST(i), ST	MULTIPLY ELEMENT BY ST
DC 11011100	1 1 010 (i)	*(2)		
DC 11011100	1 1 011 (i)	*(3)		
DC 11011100	1 1 100 (i)	FSUBR	ST(i), ST	SUBTRACT ST FROM ELEMENT
DC 11011100	1 1 101 (i)	FSUB	ST(i), ST	SUBTRACT ELEMENT FROM ST
DC 11011100	1 1 110 (i)	FDIVR	ST(i), ST	DIVIDE ST(i) BY ST
DC 11011100	1 1 111 (i)	FDIV	ST(i), ST	DIVIDE ST BY ST(i)
DD 11011101	MOD 000 R/M	FLD	Long-real	PUSH 8-BYTE EA ONTO ST
DD 11011101	MOD 001 R/M	Reserved		
DD 11011101	MOD 010 R/M	FST	Long-real	STORE ST INTO 8-BYTE EA
DD 11011101	MOD 011 R/M	FSTP	Long-real	STORE ST INTO 8-BYTE EA
DD 11011101	MOD 100 R/M	FRSTOR	94-BYTES	RESTORE 8087 STATE FROM EA
DD 11011101	MOD 101 R/M	Reserved		
DD 11011101	MOD 110 R/M	FSAVE	94-BYES	SAVE 8087 STATE TO EA

Hex Binary	MODRM Byte	Instruction	Parameters	Function
DD 11011101	MOD 111 R/M	FSTSW	2-BYTES	STORE 8087 STATUS WORD TO 2-BYTE EA
DD 11011101	1 1 000 (i)	FFREE	ST(i)	SET STACK TAG TO "EMPTY"
DD 11011101	1 1 001 (i)	*(4)		
DD 11011101	1 1 010 (i)	FST	ST(i)	STORE ST INTO ST(i)
DD 11011101	1 1 011 (i)	FSTP	ST(i)	STORE ST INTO ST(i), POP
DD 11011101	1 1 1-- ---	Reserved		
DE 11011110	MOD 000 R/M	FIADD	Word-integer	ADD 2-BYTE INTEGER EA TO ST
DE 11011110	MOD 001 R/M	FIMUL	Word-integer	MULTIPLY ST BY 2-BYTE INTEGER EA
DE 11011110	MOD 010 R/M	FICOM	Word-integer	COMPARE 2-BYTE EA INTEGER WITH ST
DE 11011110	MOD 011 R/M	FICOMP	Word-integer	COMPARE 2-BYTE INTEGER EA WITH ST, POP
DE 11011110	MOD 100 R/M	FISUB	Word-integer	SUBTRACT 2-BYTE INTEGER EA FROM ST
DE 11011110	MOD 101 R/M	FISUBR	Word-integer	SUBTRACT ST FROM 2-BYTE INTEGER EA
DE 11011110	MOD 110 R/M	FIDIV	Word-integer	DIVIDE ST BY 2-BYTE INTEGER EA
DE 11011110	MOD 111 R/M	FIDIVR	Word-integer	DIVIDE 2-BYTE INTEGER EA BY ST
DE 11011110	1 1 000 (i)	FADDP	ST(i), ST	ADD ST TO ELEMENT
DE 11011110	1 1 001 (i)	FMULP	ST(i), ST	MULTIPLY ST BY ELEMENT, POP

Hex Binary	MODRM Byte	Instruction	Parameters	Function
DE 11011110	1 1 010 ---	*(5)		
DE 11011110	1 1 011 000	Reserved		
DE 11011110	1 1 011 001	FCOMPP		COMPARE ST WITH ST(1), POP TWICE
DE 11011110	1 1 011 01-	Reserved		
DE 11011110	1 1 011 1--	Reserved		
DE 11011110	1 1 100 (i)	FSUBRP	ST(i), ST	SUBTRACT ST FROM ELEMENT, POP
DE 11011110	1 1 101 (i)	FSUBP	ST(i), ST	SUBTRACT ST(i) FROM ST, POP
DE 11011110	1 1 110 (i)	FDIVRP	ST(i), ST	DIVIDE STACK ELEMENT BY ST, POP
DE 11011110	1 1 111 (i)	FDIVP	ST(i), ST	DIVIDE ST BY STACK ELEMENT, POP
DF 11011111	MOD 000 R/M	FILD	Word-integer	CONVERT 2-BYTE EA AND PUSH ONTO STACK
DF 11011111	MOD 001 R/M	Reserved		
DF 11011111	MOD 010 R/M	FIST	Word-integer	ROUND ST AND STORE IN 2-BYTE INTEGER EA
DF 11011111	MOD 011 R/M	FISTP	Word-integer	ROUND ST, STORE IN 2-BYTE INTEGER EA, POP
DF 11011111	MOD 100 R/M	FBLD	Packed decimal	LOAD BCD TO ST
DF 11011111	MOD 101 R/M	FILD	Long-integer	CONVERT 8-BYTE INTEGER EA AND PUSH ONTO STACK
DF 11011111	MOD 110 R/M	FBSTP	Packed decimal	CONVERT ST, STORE IN 10-BYTE BCD EA, POP

Hex Binary	MODRM Byte	Instruction	Parameters	Function
DF 11011111	MOD 111 R/M	FISTP	Long-integer	ROUND ST, STORE IN 8-BYTE INTEGER EA, POP
DF 11011111	1 1 000 (i)	*(6)		
DF 11011111	1 1 001 (i)	*(7)		
DF 11011111	1 1 010 (i)	*(8)		
DF 11011111	1 1 011 (i)	*(9)		
DF 11011111	1 1 --- ---	Reserved		
E0 11100000		DBNZNE	DISP8	LOOP (CW) TIMES WHILE NOT ZERO/NOT EQUAL
E1 11100001		DBNZE	DISP8	LOOP (CW) TIMES WHILE ZERO/EQUAL
E2 11100010		DBNZ	DISP8	LOOP (CW) TIMES
E3 11100011		BCWZ	DISP8	JUMP ON (CW)=0
E4 11100100		IN	AL,PORT	BYTE INPUT FROM PORT TO REG AL
E5 11100101		IN	AW,PORT	WORD INPUT FROM PORT TO REG AW
E6 11100110		OUT	PORT,AL	BYTE OUTPUT (AL) TO PORT
E7 11100111		OUT	PORT,AW	WORD OUTPUT (AW) TO PORT
E8 11101000		CALL	DISP16	DIRECT INTRA SEGMENT CALL
E9 11101001		BR	DISP16	DIRECT INTRA SEGMENT JUMP
EA 11101010		BR	DISP16,SEG16	DIRECT INTER SEGMENT JUMP

Hex Binary	MODRM Byte	Instruction	Parameters	Function
EB 11101010		BR	DISP8	DIRECT INTRA SEGMENT JUMP
EC 11101010		IN	AL,DW	BYTE INPUT FROM PORT (DW) TO REG AL
ED 11101010		IN	AW,DW	WORD INPUT FROM PORT (DW) TO REG AW
EE 11101010		OUT	DW,AL	BYTE OUTPUT (AL) TO PORT (DW)
EF 11101010		OUT	DW,AW	WORD OUTPUT (AW) TO PORT (DW)
F0 11110000		BUSLOCK		BUS LOCK PREFIX
F1 11110001		(not used)		
F2 11110010		REPNZ/REPNE		REPEAT WHILE (CW) not equal to 0 AND (ZF) = 0
F3 11110011		REPZ/REPE/ REP		REPEAT WHILE (CW) not equal to 0 AND (ZF) = 1
F4 11110100		HALT		HALT
F5 11110101		NOT1	CY	COMPLEMENT CARRY FLAG
F6 11110110	MOD 000 R/M	TEST	EA,DATA8	BYTE TEST (EA) WITH DATA
F6 11110110	MOD 001 R/M	(not used)		
F6 11110110	MOD 010 R/M	NOT	EA	BYTE INVERT EA
F6 11110110	MOD 011 R/M	NEG	EA	BYTE NEGATE EA
F6 11110110	MOD 100 R/M	MULU	EA	BYTE MULTIPLY BY (EA), UNSIGNED
F6 11110110	MOD 101 R/M	MUL	EA	BYTE MULTIPLY BY (EA), SIGNED

Table F-2. 72291 Instructions

Hex Binary	MODRM Byte	Instruction	Parameters	Function
66 01100110	MOD REG RM	FMOVRT	MEM,FRn	move into reg
66 01100110	11 000 RM	FSIN	FS0,FSn	sine of short
66 01100110	11 001 RM	FCOS	FS0,FSn	cosine of short
66 01100110	11 010 RM	FTAN	FS0,FSn	tangent of short
66 01100110	11 011 RM	FSINCOS	FS0,FSn	sine/cosine of short
66 01100110	11 100 RM	FASIN	FS0,FSn	inverse sine of short
66 01100110	11 101 RM	FACOS	FS0,FSn	inverse cosine of short
66 01100110	11 110 RM	FATAN	FS0,FSn	inverse tangent of short
66 01100110	11 111 RM	FXCH	FS0,FSn	exchange short regs
67 01100111	MOD REG RM	FMOVRT	FRn,MEM	move reg contents
67 01100111	11 000 RM	FSIN	FL0,FLn	sine of long
67 01100111	11 001 RM	FCOS	FL0,FLn	cosine of long
67 01100111	11 010 RM	FTAN	FL0,FLn	tangent of long
67 01100111	11 011 RM	FSINCOS	FL0,FLn	sine/cosine of long
67 01100111	11 100 RM	FASIN	FL0,FLn	inverse sine of long
67 01100111	11 101 RM	FACOS	FL0,FLn	inverse cosine of long
67 01100111	11 110 RM	FATAN	FL0,FLn	inverse tangent of long
67 01100111	11 111 RM	FXCH	FL0,FLn	exchange long regs
D8 11011000	MOD 000 RM	FMOVCR	MEM32,FCTW	store control word
D8 11011000	MOD 001 RM	FMOVCR	MEM32,FSTW	store control word

Hex Binary	MODRM Byte	Instruction	Parameters	Function
D8 11011000	MOD 010 RM	FMOVCR	MEM32,FPTW	store control word
D8 11011000	MOD 011 RM	FCVTLS	VS32,FL0	convert long into short
D8 11011000	MOD 100 RM	FMOV	VS32,FS0	move short reg
D8 11011000	MOD 101 RM	FNEG	VS32,FS0	negate
D8 11011000	MOD 110 RM	FCVTSD	MEM32,FS0	convert short into int32
D8 11011000	MOD 111 RM	FCVTLD	MEM32,FL0	convert long into int32
D8 11011000	11 000 RM	FIP3V	FS0,FSn	vector product
D8 11011000	11 001 RM	FIP4V	FS0,FSn	vector product
D8 11011000	11 110 RM	FABS	FS0,FSn	absolute value
D9 11011001	MOD 011 RM	FCVTSL	VS64,FS0	convert short into long
D9 11011001	MOD 100 RM	FMOV	VS64,FL0	move long reg
D9 11011001	MOD 101 RM	FNEG	VS64,FL0	negate
D9 11011001	MOD 110 RM	FCVTSQ	MEM64,FS0	convert short into int64
D9 11011001	MOD 111 RM	FCVTLQ	MEM64,FL0	convert long into int64
D9 11011001	11 000 RM	FIP3V	FL0,LSn	vector product
D9 11011001	11 001 RM	FIP4V	FL0,LSn	vector product
D9 11011001	11 110 RM	FABS	FL0,LSn	absolute value
DA 11011010	MOD 000 RM	FCMP	FS0,VS32	compare short
DA 11011010	MOD 001 RM	FCMPE	FS0,VS32	compare short
DA 11011010	MOD 010 RM	FCMPA	FS0,VS32	compare absolute short
DA 11011010	MOD 011 RM	FCMPAE	FS0,VS32	compare absolute short
DA 11011010	11100000	FRPUSH		push float stack down

Hex Binary	MODRM Byte	Instruction	Parameters	Function
DA 11011010	11101000	FRPOP		pop off float stack
DA 11011010	11110000	FDIAG		run 72291 diagnostics
DA 11011010	11111000	FINIT		initialize 72291
DB 11011011	MOD 000 RM	FCMP	FL0,VS64	compare long
DB 11011011	MOD 001 RM	FCMPE	FL0,VS64	compare long
DB 11011011	MOD 010 RM	FCMPA	FL0,VS64	compare absolute long
DB 11011011	MOD 011 RM	FCMPAE	FL0,VS64	compare absolute long
DC 11011100	MOD 000 RM	FMOVCR	FCTW,MEM32	load control word
DC 11011100	MOD 001 RM	FMOVCR	FSTW,MEM32	load control word
DC 11011100	MOD 010 RM	FMOVCR	FPTW,MEM32	load control word
DC 11011100	MOD 011 RM	FCVTSL	FL0,VS32	convert short into long
DC 11011100	MOD 100 RM	FMOV	FS0,VS32	move short into register
DC 11011100	MOD 101 RM	FNEG	FS0,VS32	negate
DC 11011100	MOD 110 RM	FCVTDS	FS0,MEM32	convert int32 into short
DC 11011100	MOD 110 RM	FPOWER	FS0,VS32	raise to a power
DC 11011100	MOD 111 RM	FCVTDL	FL0,MEM32	convert int32 into long
DC 11011100	11 000 RM	FEXPE	FS0,FSn	natural exponential
DC 11011100	11 001 RM	FLOGE	FS0,FSn	natural logarithm
DC 11011100	11 010 RM	FEXPEM1	FS0,FSn	natural exponential -1
DC 11011100	11 110 RM	FSQRT	FS0,FSn	square root
DC 11011100	11 111 RM	FRND	FS0,FSn	round to integer
DD 11011101	MOD 011 RM	FCVTLS	FS0,VS64	convert long into short

Hex Binary	MODRM Byte	Instruction	Parameters	Function
DD 11011101	MOD 100 RM	FMOV	FL0,VS64	move long into register
DD 11011101	MOD 101 RM	FNEG	FL0,VS64	negate
DD 11011101	MOD 110 RM	FCVTQS	FS0,MEM64	convert int64 into short
DD 11011101	MOD 110 RM	FPOWER	FL0,VS64	raise to a power
DD 11011101	MOD 111 RM	FCVTQL	FL0,MEM64	convert int64 into long
DD 11011101	11 000 RM	FEXPE	FL0,FLn	natural exponential
DD 11011101	11 001 RM	FLOGE	FL0,LSn	natural logarithm
DD 11011101	11 010 RM	FEXPEM1	FL0,FLn	natural exponential -1
DD 11011101	11 110 RM	FSQRT	FL0,LSn	square root
DD 11011101	11 111 RM	FRND	FL0,LSn	round to integer
DE 11011110	MOD 000 RM	FADD	FS0,VS32	add shorts
DE 11011110	MOD 001 RM	FSUB	FS0,VS32	subtract short
DE 11011110	MOD 010 RM	FMUL	FS0,VS32	multiply short
DE 11011110	MOD 011 RM	FDIV	FS0,VS32	divide short
DE 11011110	MOD 100 RM	FMOD	FS0,VS32	modulo function
DE 11011110	MOD 101 RM	FREM	FS0,VS32	remainder
DE 11011110	11 111 RM	FATAN2	FS0,FSn	inverse tangent of short
DF 11011111	MOD 000 RM	FADD	FL0,VS64	add longs
DF 11011111	MOD 001 RM	FSUB	FL0,VS64	subtract long
DF 11011111	MOD 010 RM	FMUL	FL0,VS64	multiply long
DF 11011111	MOD 011 RM	FDIV	FL0,VS64	divide long
DF 11011111	MOD 100 RM	FMOD	FL0,VS64	modulo function

FLAGS REGISTER CONTAINS:

```
X:X:X:X:(OF):(DF):(IF):(TF):(SF):(ZF):X:(AF):X:(PF):X:(CF)
```

*The marked encodings are *not* generated by the language translators. If however, the 8087 encounters one of these encodings in the instruction stream, it will execute it as follows:

1. FSTP ST(i)
2. FCOM ST(i)
3. FCOMP ST(i)
4. FXCH ST(i)
5. FCOMP ST(i)
6. FFREE ST(i) and pop stack
7. FXCH ST(i)
8. FSTP ST(i)
9. FSTP ST(i)

V-Series Instruction Set Matrix

b	= byte operation
d	= direct
f	= from CPU reg
i	= immediate
ia	= immed.to accum.
ib	= immediate byte
id	= indirect
is	= immed. byte sign ext.
iw	= immediate word
l	= long ie. intersegment
m	= memory
r	= register
r/m	= EA is second byte
IX	= short intrasegment
sr	= segment register
t	= to CPU reg
v	= variable
w	= word operation
z	= zero

REG IS ASSIGNED ACCORDING TO THE FOLLOWING TABLE.		
16-BIT (W=1)	**8-BIT (W=0)**	**SEGMENT REG**
000 AW	000 AL	00 DS1
001 CW	001 CL	01 PS
010 DW	010 DL	10 SS
011 BW	011 BL	11 DS0
100 SP	100 AH	
101 BP	101 CH	
110 IX	110 DH	
111 IY	111 BH	

EA IS COMPUTED AS FOLLOWS: (DISP8 SIGN EXTENDED TO 16 BITS)	
00 000 (BW) + (IX)	DS0
00 001 (BW) + (IY)	DS0
00 010 (BP) + (IX)	SS
00 011 (BP) + (IY)	SS
00 100 (IX)	DS0
00 101 (IY)	DS0
00 110 DISP16 (DIRECT ADDRESS)	DS0
00 111 (BW)	DS0
01 000 (BW) + (IX) + DISP8	DS0
01 001 (BW) + (IY) + DISP8	DS0
01 010 (BP) + (IX) + DISP8	SS
01 011 (BP) + (IY) + DISP8	SS
01 100 (IX) + DISP8	DS0
01 101 (IY) + DISP8	DS0
01 110 (BP) + DISP8	SS
01 111 (BW) + DISP8	DS0
10 000 (BW) + (IX) + DISP16	DS0

EA IS COMPUTED AS FOLLOWS: (DISP8 SIGN EXTENDED TO 16 BITS) (Cont'd)	
10 001 (BW) + (IY) + DISP16	DS0
10 010 (BP) + (IX) + DISP16	SS
10 011 (BP) + (IY) + DISP16	SS
10 100 (IX) + DISP16	DS0
10 101 (IY) + DISP16	DS0
10 110 (BP) + DISP16	SS
10 111 (BW) + DISP16	DS0
11 000 REG AW / AL	
11 001 REG CW /CL	
11 010 REG DW /DL	
11 011 REG BW /BL	
11 100 REG SP / AH	
11 101 REG BP /CH	
11 110 REG IX /DH	
11 111 REG IY /BH	

V20/V25 Instruction Set Matrix								
	LO							
Hi	**0**	1	2	3	4	5	6	7
0	ADD b.f.r/m	ADD w.f.r/m	ADD b.t.r/m	ADD w.t.r/m	ADD b.ia	ADD w.ia	PUSH DS1	POP DS1
1	ADDC b.f.r/m	ADDC w.f.r/m	ADDC b.t.r/m	ADDC w.t.r/m	ADDC b.ia	ADDC w.ia	PUSH SS	POP SS
2	AND b.f.r/m	AND w.f.r/m	AND b.t.r/m	AND w.t.r/m	AND b.ia	AND w.ia	SEG DS1	ADJ4A
3	XOR b.f.r/m	XOR w.f.r/m	XOR b.t.r/m	XOR w.t.r/m	XOR b.ia	XOR w.ia	SEG SS	ADJBA
4	INC AW	INC CW	INC DW	INC BW	INC SP	INC BP	INC IX	INC IY
5	PUSH AW	PUSH CW	PUSH DW	PUSH BW	PUSH SP	PUSH BP	PUSH IX	PUSH IY
6	PUSH R	POP R	CHKIND R.R/M					
7	BV	BNV	BL/ BC	BNL/ BNC	BE/ BZ	BNE/ BNZ	BNH	BH
8	Immed b.r/m	Immed w.r/m	Immed b.r/m	Immed is.r/m	TEST b.r/m	TEST w.r/m	XCH b.r/m	XCH w.r/m
9	NOP	XCH CW	XCH DW	XCH BW	XCH SP	XCH BP	XCH IX	XCH IY
A	MOV m $\rightarrow$ AL	MOV m $\rightarrow$ AW	MOV AL $\rightarrow$ m	MOV AW $\rightarrow$ m	MOVBK b	MOVBK w	CMPBK b	CMPBK w

V20/V25 Instruction Set Matrix								
	LO							
Hi	**0**	1	2	3	4	5	6	7
B	MOV i → AL	MOV i → CL	MOV i → DL	MOV i → BL	MOV i → AH	MOV i → C	MOV i → DH	MOV i → BH
C	Shift b.r/m.i	Shift w.r/mi	RETI (i - SP)	RETI	MOV DS1	MOV DS0	MOV b.i.r/m	MOV w.i.r/m
D	Shift b	Shift w	Shift b.v	Shift w.v	CVTBD	CVTDB		TRANS
E	DBNZNE	DBNZE	DBNZ	BCWZ	IN b	IN w	OUT b	OUT w
F	BUSLOCK		REP	REPZ	HALT	NOT1 CY	Grp 1 b.r/m	Grp 1 w.r/m

V20/V25 Instruction Set Matrix								
	LO							
Hi	**8**	9	A	B	C	D	E	F
0	OR b.f.r/m	OR w.f.r/m	OR b.t.r/m	OR w.t.r/m	OR b.ia	OR w.ia	PUSH PS	
1	SUBC b.f.r/m	SUBC w.f.r/m	SUBC b.t.r/m	SUBC w.t.r/m	SUBC b.ia	SUBC w.ia	PUSH DS0	POP DS0
2	SUB b.f.r/m	SUB w.f.r/m	SUB b.t.r/m	SUB w.t.r/m	SUB b.ia	SUB w.ia	SEG PS	ADJ4S
3	CMP b.f.r/m	CMP w.f.r/m	CMP b.t.r/m	CMP w.t.r/m	CMP b.ia	CMP w.ia	SEG DS0	ADJBS

V20/V25 Instruction Set Matrix								
	LO							
Hi	**8**	9	A	B	C	D	E	F
4	DEC AW	DEC CW	DEC DW	DEC BW	DEC SP	DEC BP	DEC IX	DEC IY
5	POP AW	POP CW	POP DW	POP BW	POP SP	POP BP	POP IX	POP IY
6	PUSH iw	MUL r.iw.r/m	PUSH is	MUL r.is.r/m	INM b	INM w	OUTM b	OUTM w
7	BN	BP	BPE	BPO	BLT	BGE	BLE	BGT
8	MOV b.f.r/m	MOV w.f.r/m	MOV b.t.r/m	MOV w.t.r/m	MOV sr.f.r/m	LDEA	MOV sr.t.r/m	POP r/m
9	CVTBW	CVTWL	CALL i.d	POLL	PUSH PSW	POP PSW	SAHF	LAHF
A	TEST b.i	TEST w.i	STM b	STM w	LDM b	LDM w	CMPM b	CMPM w
B	MOV i → AW	MOV i → CW	MOV i → DW	MOV i → BW	MOV i → SP	MOV i → BP	MOV i → IX	MOV i → IY
C	ENTER iw.ib	DISPOSE	RETI I . (i - SP)	RETI I	BRK Type 3	BRK (Any)	BRKV	RETI
D	ESC 0	ESC 1	ESC 2	ESC 3	ESC 4	ESC 5	ESC 6	ESC 7
E	CALL d	BR d	BR i.d	BR si.d	IN v.b	IN v.w	OUT v.d	OUT v.w

V20/V25 Instruction Set Matrix								
	LO							
Hi	**8**	9	A	B	C	D	E	F
F	CLR1 CY	STC	DI	EI	CLR1 DIR	STD	Grp 2 b.r/m	Grp 2 w.r/m

Where								
mod r/m	000	001	010	011	100	101	110	111
Immed	ADD	OR	ADDC	SUBC	AND	SUB	XOR	CMP
Shift	ROL	ROR	ROLC	RORC	SHL	SHR	SHL	SHRA
Grp 1	TEST		NOT	NEG	MULU	MUL	DIVU	DIV
Grp 2	INC	DEC	CALL id	CALL I id	BR id	BR I id	PUSH	

Notes

V-Series Instruction Set Summary

Function	Format	V20 Clock Cycles	Comments
DATA TRANSFER **MOVE = Move:**			
Register to Register/Memory	1000100w mod reg r/m	2/9-13	
Register/Memory to register	1000101w mod reg r/m	2/11-15	
Immediate to register/memory	1100011w mod 000 r/m data data if w = 1	11-15	8/16-bit
Immediate to register	1011w reg data data if w = 1	4	8/16-bit
Memory to accumulator	1010000w addr-low addr-high	10-14	
Accumulator to memory	1010001w addr-low addr-high	9-13	
Register/memory to segment register	10001110 mod 0 reg r/m	2/11-15	

Function	Format	V20 Clock Cycles	Comments
Segment/register to register/memory	10001100 mod 0 reg r/m	10/14	
PUSH = Push: Memory	11111111 mod 110 r/m	26	
Register	01010 reg	12	
Segment register	000 reg 110	12	
Immediate	011010s0 data data if s = 010	12	
PUSH R = Push All	01100000	67	
POP = Pop: Memory	10001111 mod 000 r/m	11	
Register	01011 reg	12	
Segment register	000 reg 111 (reg $\neq$ 01)	12	
POP R = Pop All	01100001	75	
XCH = Exchange:			
Register/memory with register	1000011w mod reg r/m	3/16-24	
Register with accumulator	10010 reg	3	

Function	Format	V20 Clock Cycles	Comments
IN = Input from:			
Fixed port	1110010w port	13	
Variable port	1110110w	12	
OUT = Output to:			
Fixed port	1110011w port	12	
Variable port	1110111w	12	
TRANS = translate byte to AL	11010111	9	
LDEA = Load EA to register	10001101 mod reg r/m	4	
MOV = Load pointer to DS0	11000101 mod reg r/m (mod ≠ 11)	26	
MOV = Load pointer to DS1	11000100 mod reg r/m (mod ≠ 11)	26	
MOV = Load AH with PSW	10011111	2	
MOV = Store AH into PSW	10011110	3	
PUSH PSW = Push flags	10011100	12	

Function	Format	V20 Clock Cycles	Comments
POP PSW = Pop flags	10011101	12	
ARITHMETIC ADD = Add:			
Reg/memory with register to either	000000dw mod reg r/m	2/24	
Immediate to register/memory	100000sw mod 000 r/m data data if s w - 01	2/26	
Immediate to accumulator	0000010w data data if w - 1	4	8/16-bit
ADDC = ADD with carry:			
Reg/memory with register to either	000100dw mod reg r/m	2/24	
Immediate to register/memory	100000sw mod 010 r/m data data is s w - 01	2/26	
Immediate to accumulator	0001010w data data if w - 1	4	8/16-bit

Function	Format	V20 Clock Cycles	Comments
INC = Increment:			
Register/memory	1111111w mod 000 r/m	2/24	
Register	01000 reg	2	
SUB = Subtract:			
Reg/memory and register to either	001010dw mod reg r/m	2/24	
Immediate from register/memory	100000sw mod 101 r/m data data if s w 01	4/26	
Immediate from accumulator	0010110w data data if w - 1	4	8/16-bit
SUBC = Subtract with borrow:			
Reg/memory and register to either	000110dw mod reg r/m	2/24	
Immediate from register/memory	100000sw mod 011 r/m data data if s w - 01	4/26	
Immediate from accumulator	0001110w data data if w - 1	4	8/16-bit

Function	Format	V20 Clock Cycles	Comments
DEC = Decrement:			
Register/memory	1111111w mod 001 r/m	2/24	
Register	01001 reg	2	
CMP = Compare:			
Register/memory with register	0011101w mod reg r/m	2/15	
Register with register/memory	0011100w mod reg r/m	2/15	
Immediate with register/memory	100000sw mod 111 r/m data data if s w - 01	4/17	
Immediate with accumulator	0011110w data data if w - 1	4	8/16-bit
NEG = Change sign	1111011w mod 011 r/m	2/24	
ADJBA = ASCII adjust for add	00110111	3	
ADJ4A = Decimal adjust for add	00100111	3	
ADJBS = ASCII adjust for subtract	00111111	7	

Function	Format	V20 Clock Cycles	Comments
ADJ4S = Decimal adjust for subtract	00101111	7	
MULU = Multiply (unsigned):	1111011w mod 100 r/m		
Register-Byte		21-22	
Register-Word		29-30	
Memory-Byte		27-28	
Memory-Word		39-40	
MUL = Integer multiply (signed):	1111011w mod 101 r/m		
Register-Byte		33-39	
Register-Word		41-47	
Memory-Byte		39-45	
Memory-Word		51-57	
MUL = Integer immediate multiply (signed)	011010s1 mod reg r/m data data if s= 0	28-34/46-52	
DIVU = Divide (unsigned):	1111011w mod 110 r/m		

Function	Format	V20 Clock Cycles	Comments
Register-Byte		19	
Register-Word		25	
Memory-Byte		25	
Memory-Word		35	
DIV = Integer divide (signed):	1111011w mod 111 r/m		
Register-Byte		29-34	
Register-Word		38-43	
Memory-Byte		35-40	
Memory-Word		48-53	
CVTBD = ASCII adjust for multiply	11010100 00001010	15	
CVTDB = ASCII adjust for divide	11010101 00001010	7	
CVTBW = Convert byte to word	10011000	2	
CVTWL = Convert word to double word	10011001	5	

Function	Format	V20 Clock Cycles	Comments
LOGIC Shift/Rotate Instructions:			
Register/Memory by 1	1101000w mod TTT r/m	2/24	
Register/Memory by CL	1101001w mod TTT r/m	7+n/27+n	
Register/Memory by Count	1100000w mod TTT r/m count	7+n/27+n	
	TTT Instruction 000 ROL 001 ROR 010 ROLC 011 RORC 100 SHL 101 SHR 111 SHRA		
AND = And:			
Reg/memory and register to either	001000dw mod reg r/m	2/24	
Immediate to register/memory	1000000w mod 100 r/m data data if w = 1	4/26	
Immediate to accumulator	0010010w data data if w = 1	4	8/16-bit

Function	Format	V20 Clock Cycles	Comments
TEST = And function to flags, no result:			
Register/memory and register	1000010w mod reg r/m	2/14	
Immediate data and register/memory	1111011w mod 000 r/m data data if w = 1	4/15	
Immediate data and accumulator	1010100w data data if w = 1	4	8/16-bit
OR = Or:			
Reg/memory and register to either	000010dw mod reg r/m	2/24	
Immediate to register/memory	1000000w mod 001 r/m data data if w = 1	4/26	
Immediate to accumulator	0000110w data data if w = 1	4	8/16-bit
XOR = Exclusive or:			
Reg/memory and register to either	001100dw mod reg r/m	2/24	
Immediate to register/memory	1000000w mod 110 r/m data data if w = 1	4/26	

Function	Format	V20 Clock Cycles	Comments
Immediate to accumulator	0011010w data data if w = 1	4	8/16-bit
NOT = Invert register/memory	1111011w mod 010 r/m	2/24	
STRING MANIPULATION			
MOVBK = Move byte/word	1010010w	14	
CMPBK = Compare byte/word	1010011w	22	
CMPM = Scan byte/word	1010111w	15	
LDM = Load byte/wd to AL/AW	1010110w	12	
STM = Star byte/wd from AL/AW	1010101w	10	
INM = Input byte/wd from DW port	0110110w	18	
OUTM = Output byte/wd to DW port	0110111w	18	
Repeated by count in CW			

Function	Format	V20 Clock Cycles	Comments
MOVBK - Move string	11110010 1010010w	11 + (8/16)n	
CMPBK - Compare string	1111001z 11010011w	7 + 22n	
CMPM = Scan string	1111001z 11010111w	7 + 14n	
LDM - Load string	11110010 1010110w	7 + 13n	
STM - Store string	11110010 1010101w	7 + 8n	
INM = Input string	11110010 0110110w	8 + 8n	
OUTM = Output string	11110010 0110111w	8 + 8n	
CONTROL TRANSFER			
CALL = Call: Direct within segment	11101000 disp-low disp--high	20	
Register memory indirect within segment	11111111 mod 010 r/m	18/31	
Direct intersegment	10011010 segment offset segment selector	29	
Indirect intersegment	11111111 mod 011 r/m (mod ≠11)	47	

Function	Format	V20 Clock Cycles	Comments
BR = Unconditional jump:			
Short long	11101011 disp-low	12	
Direct within segment	11101001 disp-low disp-high	12	
Register/memory indirect within segment	11111111 mod 100 r/m	11/24	
Direct intersegment	11101010 segment offset segment selector	15	
Indirect intersegment	11111111 mod 101 r/m (mod ≠11)	35	
RET = Return from CALL:			
Within segment	11000011	19	
Within seg adding immed to SP	11000010 data-low data-high		
Intersegment	11001011	24	

Function	Format	V20 Clock Cycles	Comments
Intersegment adding immediate to SP	11001010 data-low data-high	29	
BE Jump on equal zero	01110100 disp	4/14	14 if BR
BLT = Jump on less not greater or equal	01111100 disp	4/14	taken
BLE = Jump on less or equal not greater	01111110 disp	4/14	4 if BR
BL = Jump on below not above or equal	01110010 disp	4/14	not taken
BNH = Jump on below or equal not above	01110110 disp	4/14	
BPE = Jump on parity even	01111010 disp	4/14	
BV = Jump on overflow	01110000 disp	4/14	
BN = Jump on sign	01111000 disp	4/14	
BNE = Jump on not equal not zero	01110101 disp	4/14	
BGE = Jump on not less greater or equal	01111101 disp	4/14	

Function	Format	V20 Clock Cycles	Comments
BGT = Jump on not less or equal greater	01111111 disp	4/14	
BNL = Jump on not below above or equal	01110011 disp	4/14	
BH = Jump on not below or equal above	01110111 disp	4/14	
BPO = Jump on not parity odd	01111011 disp	4/14	
BNV = Jump on not overflow	01110001 disp	4/14	
BP = Jump on not sign	01111001 disp	4/14	
DBNZ = Loop CW times	11100010 disp	5/13	
DBNZE = Loop while zero equal	11100001 disp	5/14	
DBNZNE = Loop while not zero equal	11100000 disp	5/14	BR taken/
BCWZ = Jump on CW zero	11100011 disp	5/13	BR not taken
PREPARE = Enter Procedure	11001000 data-low data-high L		

Function	Format	V20 Clock Cycles	Comments
L = 0 L=1 L>>1		16 25 23+16 (n-1)	
DISPOSE = Leave Procedure	11001001	10	
BRK = Interrupt:			
Type specified	11001101 type	50	
Type 3	11001100	50	if INT.taken/
BRKV = Interrupt on overflow	11001110	52	if INT.not taken
RETI = Interrupt return	11001111	39	
CHKIND = Detect value out of range	01100010 mod reg r/m	73-76	
PROCESSOR CONTROL			
CLR1 CY = Clear carry	11111000	2	

Function	Format	V20 Clock Cycles	Comments
NOT1 CY = Complement carry	11110101	2	
SET1 CY = Set carry	11111001	2	
CLR1 DIR = Clear direction	11111100	2	
SET1 DIR = Set direction	11111101	2	
DI = Clear interrupt	11111010	2	
EI = Set interrupt	11111011	2	
HALT = Halt	11110100	2	
POLL = Wait	10011011	2	if test = 0
BUSLOCK = Bus lock prefix	11110000	2	
ESC = Processor Extension Escape	10011TTT mod LLL r/m	6	
(TTT LLL are opcode to processor extension)			
INS: Insert Bit Field			
Reg8 and Reg8	00001111 00110001 11 reg reg	35-113	
Reg8 and Imm.	00001111 00111001 11 000 mem data	75-103	

Function	Format	V20 Clock Cycles	Comments
EXT: Extract Bit Field			
Reg8 and Reg8	00001111 00110011 11 reg reg	34-59	
Reg8 and Imm.	00001111 00111011 11 000 mem data	25-52	
ADD4S: Add Nibble String	00001111 00100000	7+19n	
SUB4S: Subtract Nibble String	00001111 00100010	7+19n	
CMP4S: Compare Nibble String	00001111 00100110	7+19n	
ROL4: Rotate Left Nibble	00001111 00101000 mod 000 r/m	25/28	
ROR4: Rotate Right Nibble	00001111 00101010 mod 000 r/m	29/33	

Function	Format	V20 Clock Cycles	Comments
TEST1: Test One Bit			
Register/Memory,CL	00001111 0001000w mod 000 r/m	3/12	
Register/Memory,imm.	00001111 0001100w mod 000 r/m	4/13	
NOT1: Not One Bit			
Register/Memory,CL	00001111 0001011w mod 000 r/m	4/18	
Register/Memory,imm.	00001111 0001111w mod 000 r/m	5/19	
CLR1: Clear One Bit			
Register/Memory,CL	00001111 0001001w mod 000 r/m	5/14	
Register/Memory,imm.	00001111 0001101w mod 000 r/m	6/15	
SET1: Set One Bit			
Register/Memory,CL	00001111 0001010w mod 000 r/m	4/13	
Register/Memory,imm.	00001111 0001110w mod 000 r/m	5/14	
BRKEM: Break for 8080 Emulation	00001111 1111111w data	50	

Function	Format	V20 Clock Cycles	Comments
FINT: Finish Interrupt	00001111 10010010	2	
MOVSPA: Move Stack Pointer After Bank Switch	00001111 00100101	16	
MOVSPB: Move Stack Pointer Before Bank Switch Register	00001111 10010101 11110 reg	11	
BRKCS: Break with Context Switch Register	00001111 00101101 11001 reg	15	
RETRBI: Return from Register Bank Switching Interrupt	00001111 10010001	12	

FOOTNOTES

The effective Address (EA) of the memory operand is computed according to the mod and r/m fields:

if mod = 11 then r/m is treated as a REG field

if mod = 00 then DISP = 0*, disp-low and disp-high are absent

if mod = 01 then DISP = disp-low sign-extended to 16-bits, disp-high is absent

if mod = 10 then DISP = disp-high: disp-low

if r/m = 000 then EA = (BW) + (IX) + DISP

if r/m = 001 then EA = (BW) + (IY) + DISP

if r/m = 010 then EA = (BP) + (IX) + DISP

if r/m = 011 then EA = (BP) + (IY) + DISP

if r/m = 100 then EA = (IX) + DISP

if r/m = 101 then EA = (IY) + DISP

if r/m = 110 then EA = (BP) + DISP*

if r/m = 111 then EA = (BW) + DISP

DISP follows 2nd byte of instruction (before data if required)

*except if mod = 00 and r/m = 110 then EA = disp-high: disp-low.

SEGMENT OVERRIDE PREFIX

001 reg 110

reg is assigned according to the following:

reg	Segment Register
00	DS1
01	PS
10	SS
11	DS0

REG is assigned according to the following table:

16-Bit (w = 1)	8-Bit (w = 0)
000 AW	000 AL
001 CW	001 CL
010 DW	010 DL
011 BW	011 BL
100 SP	100 AH
101 BP	101 CH
110 IX	110 DH
111 IY	111 BH

The physical addresses of all operands addressed by the BP register are computed using the SS segment register. The physical addresses of the destination operands of the string primitive operations (those addressed by the IY register) are computed using the DS1 segment, which may not be overridden.

Index

A

absolute expression,**5-2**
acvtv20 translation tool,**E-16**
adding base and index register
  in expression,**5-8**
addition operator
  binary,**5-12**
  unary,**5-11**
allocating record storage,**4-48**
allocating structure storage,**4-59**
AND operator,**5-19, 11-7**
anonymous reference,**6-2**
  with expression,**5-9**
arithmetic operator,**5-11**
ASCII codes,**D-1**
assembler
  character set,**2-1**
  control general syntax,**7-2**
  cross reference format,**8-5**
  directive,**4-1**
  error messages,**B-1**
  general controls,**7-2**
  listing,**8-1**
  operation,**1-2**
  primary controls,**7-2**
  statement syntax,**2-12**
  symbol table format,**8-5**
assembler controls
  CAPITALS,**7-4**
  DATE,**7-4**
  DEBUG,**7-4**
  EJECT,**7-11**
  ERRORPRINT,**7-5**
  GEN,**7-11**
  general,**7-11**
  GENONLY,**7-11**
  INCLUDE,**7-11**

assembler controls (continued)
INCLUDE with macro preprocessor,**10-6**
LIST,**7-12**
MACRO,**7-6**
MODV20,**7-7**
MODV25,**7-7**
OBJECT,**7-7**
OPTIMIZE,**7-7**
PAGELENGTH,**7-8**
PAGEWIDTH,**7-8**
primary,**7-4**
PRINT,**7-8**
RESTORE,**7-12**
SAVE,**7-13**
SYMBOLS,**7-8**
TITLE,**7-13**
TYPE,**7-9**
UNREFERENCED_EXTERNALS,**7-9**
WORKFILES,**7-9**
XREF,**7-10**
assembler syntax
blank line,**2-13**
comment,**2-13**
continuation line,**2-14**
keyword,**2-13**
label,**2-12**
operand,**2-13**
prefix,**2-13**
symbol,**2-2**
assembly source translation
acvtv20 tool,**E-16**
assembly source translation
HP 64853 to HP 64873,**E-1**
ASSUME directive,**4-10**
assumed,**6-3**
* operator,**5-15, 11-6**
attribute
BASE,**3-3**
INDEX,**3-3**
OFFSET,**3-2**
PS ADDRESSABILITY,**3-6**

attribute (continued)
RELOCATION TYPE,**3-4**
SEGMENT,**3-4**
SEGMENT ADDRESSABILITY,**3-5**
SEGMENT RELOCATION,**3-4**
TYPE,**3-2**

B
balanced text string,**11-4**
baltex,**11-4**
BASE attribute,**3-3**
base register
in expression,**5-8**
binary minus,**5-12**
binary plus,**5-12**
blank line in syntax,**2-13**
bracket macro function,**12-3**

C
CAPITALS assembler control,**7-4**
caret,**B-1**
case insensitivity
assembler controls,**7-4**
case sensitivity
macro preprocessor,**10-1, 11-3**
character constant,**2-11**
character set
assembler,**2-1**
macro preprocessor,**11-2**
code translation
acvtv20 tool,**E-16**
HP 64853 to HP 64873,**E-1**
colon
with label,**2-6**
comment in syntax,**2-13**
comment macro function,**12-2**
constant,**2-8**
character,**2-11**
integer,**2-8**
real,**2-10**
continuation line in syntax,**2-14**
controls, assembler
CAPITALS,**7-4**
DATE,**7-4**

controls, assembler (continued)
DEBUG,**7-4**
EJECT,**7-11**
ERRORPRINT,**7-5**
GEN,**7-11**
general,**7-2, 7-11**
GENONLY,**7-11**
INCLUDE,**7-11**
LIST,**7-12**
MACRO,**7-6**
MODV20,**7-7**
MODV25,**7-7**
OBJECT,**7-7**
OPTIMIZE,**7-7**
PAGELENGTH,**7-8**
PAGEWIDTH,**7-8**
primary,**7-2, 7-4**
PRINT,**7-8**
RESTORE,**7-12**
SAVE,**7-13**
SYMBOLS,**7-8**
TITLE,**7-13**
TYPE,**7-9**
UNREFERENCED_EXTERNALS,**7-9**
WORKFILES,**7-9**
XREF,**7-10**
creating macros,**13-2**
cross reference format,**8-5**
CS ADDRESSABILITY attribute,**3-6**

D
data definition directive,**4-5**
data object,**4-6**
DATE assembler control,**7-4**
DB directive,**4-13**
with string,**4-18**
DD directive,**4-13**
DEBUG assembler control,**7-4**
default
PROC directive,**4-40**
segment,**4-4**
segment register,**4-11**
segments for memory addressing,**6-11**

DEFINE macro function,**13-2**
defining macros,**13-2**
differences between processor modes,**7-14**
directive
ASGNSFR,**4-8**
assembler,**4-1**
ASSUME,**4-10**
data definition,**4-5**
DB,**4-13**
DB with string,**4-18**
DD,**4-13**
DL,**4-13**
DQ,**4-13**
DS,**4-13**
DT,**4-13**
DW,**4-13**
DW, DD, DQ, DT with string,**4-19**
END,**4-23**
ENDP,**4-40**
ENDS (segments),**4-50**
ENDS (structures),**4-58**
EQU,**4-24**
EXTRN,**4-29**
GROUP,**4-33**
LABEL,**4-36**
NAME,**4-38**
ORG,**4-39**
PROC,**4-40**
program linkage,**4-7**
PUBLIC,**4-43**
PURGE,**4-44**
RECORD,**4-46**
SEGMENT,**4-50**
segmentation,**4-3**
STRUC,**4-58**
directive:SETIDB,**4-56**
division operator,**5-15**
DL directive,**4-13**
DQdirective,**4-13**
DS directive,**4-13**
DT directive,**4-13**

DW directive,**4-13**
DW, DD, DS, DQ, DL, DT directive
with string,**4-19**

E EBCDIC codes,**D-1**
EJECT assembler control,**7-11**
END directive,**4-23**
ENDP directive,**4-40**
ENDS directive,**4-50**
ENDS directive (structures),**4-58**
EQ operator,**5-20, 11-7**
EQS macro function,**12-5**
EQU directive,**4-24**
EQU symbols defined,**2-8**
error messages
assembler,**B-1**
error messages
formats,**A-1**
macro preprocessor,**C-1**
ERRORPRINT assembler control,**7-5**
escape macro function,**12-3**
EVAL macro function,**12-6**
EXIT macro function,**12-6**
expression
absolute,**5-2**
anonymous,**5-9**
base register in,**5-8**
with EQU directive,**5-10**
external,**5-3**
generally,**5-2**
group name operand,**5-6**
index register in,**5-8**
label name operand,**5-7**
in macro preprocessor,**11-4**
numeric operand,**5-4**
operand,**5-4, 6-1**
operator,**5-11**
operator, arithmetic,**5-11**
operator, logical,**5-19**
operator, record,**5-31**
record field operand,**5-6**
record operand,**5-5**

expression (continued)
register indirect,**5-8**
relocatable,**5-3**
segment name operand,**5-6**
string operand,**5-5**
structure field operand,**5-7**
variable name operand,**5-7**
external expression,**5-3**
EXTRN directive,**4-29**

F
formats for error messages,**A-1**
function
%((bracket) macro,**12-3**
bracket macro,**12-3**
%’ (comment) macro,**12-2**
comment macro,**12-2**
DEFINE macro,**13-2**
EQS macro,**12-5**
%n (escape) macro,**12-3**
escape macro,**12-3**
EVAL macro,**12-6**
EXIT macro,**12-6**
GES macro,**12-5**
GTS macro,**12-5**
IF macro,**12-7**
LEN,**10-3**
LEN macro,**12-8**
LES macro,**12-5**
LTS macro,**12-5**
MATCH macro,**12-10**
METACHAR macro,**12-11**
NES macro,**12-5**
REPEAT macro,**12-12**
SET macro,**12-13**
SUBSTR,**10-3**
SUBSTR macro,**12-14**
WHILE macro,**12-14**

G
GE operator,**5-20, 11-7**
GEN assembler control,**7-11**
general assembler controls,**7-2, 7-11**
general syntax,**2-12**

GENONLY assembler control,**7-11**
GES macro function,**12-5**
group,**4-33**
OFFSET operator with,**4-35**
override operator,**5-24**
GROUP directive,**4-33**
group name
defined,**2-8**
as expression operand,**5-6**
GT operator,**5-20, 11-7**
GTS macro function,**12-5**

H HIGH operator,**5-17, 11-5**
HP 64853 to HP 64873 translation,**E-1**

I IF macro function,**12-7**
immediate,**6-3**
immediate value
See also numeric value
INCLUDE assembler control,**7-11**
with macro preprocessor,**10-6**
incorrect macro example,**12-15**
INDEX attribute,**3-3**
index register
in expression,**5-8**
initialization
record,**4-48**
segment register,**4-22**
structure,**4-59**
instruction mnemonic defined,**2-6**
instruction set,**1-1**
assembler,**6-21**
V-Series,**G-1**
V-Series in hexadecimal order,**F-1**
integer constant,**2-8**

K keyword defined,**2-3**
keyword in syntax,**2-13**

L label,**4-6**
in syntax,**2-12**
LABEL directive,**4-36**

label name
defined,**2-6**
as expression operand,**5-7**
LE operator,**5-20, 11-7**
LEN function,**10-3**
LEN macro function,**12-8**
LENGTH operator,**5-28**
LES macro function,**12-5**
LIST assembler control,**7-12**
listing, assembler,**8-1**
literal (*) character,**10-5**
logical operator,**5-19**
logical segment,**4-4**
LOW operator,**5-17, 11-5**
LT operator,**5-20, 11-7**
LTS macro function,**12-5**

M MACRO assembler control,**7-6**
macro example (incorrect),**12-15**
macro function
bracket,**12-3**
comment,**12-2**
DEFINE,**13-2**
EQS,**12-5**
escape,**12-3**
EVAL,**12-6**
EXIT,**12-6**
GES,**12-5**
GTS,**12-5**
IF,**12-7**
LEN,**10-3, 12-8**
LES,**12-5**
LTS,**12-5**
MATCH,**12-10**
METACHAR,**12-11**
NES,**12-5**
REPEAT,**12-12**
SET,**12-13**
string relational,**12-5**
SUBSTR,**10-3, 12-14**
WHILE,**12-14**

macro preprocessor,**10-1**
balanced text string (baltex),**11-4**
character,**11-2**
error messages,**C-1**
INCLUDE file,**10-6**
input parsing,**10-5**
input source characteristics,**10-2**
literal character,**10-5**
metacharacter (%),**10-2**
number in,**11-3**
output buffering,**10-6**
symbol in,**11-3**
with expressions,**11-4**
with operators,**11-4**
MASK operator,**5-31**
MATCH macro function,**12-10**
memory addressing,**6-7, 6-9**
METACHAR macro function,**12-11**
microprocessors,**1-1**
—
binary,**5-12**
unary,**5-11**
with base and index register,**5-8**
MOD operator,**5-15, 11-6**
MODRM byte,**6-8**
MODV20 assembler control,**7-7**
MODV25 assembler control,**7-7**
multiple segment definition,**4-53**

N

NAME directive,**4-38**
NE operator,**5-20, 11-7**
NES macro function,**12-5**
nesting segments,**4-54**
NOT operator,**5-20, 11-6**
number
macro preprocessor,**11-3**
17-bit,**5-3**
numeric constant
other bases,**2-9**
numeric value
character constant,**2-11**
constant,**2-8**

numeric value (continued)
as expression operand,**5-4**
immediate value,**6-3**
integer constant,**2-8**
real constant,**2-10**

O

OBJECT assembler control,**7-7**
OFFSET attribute,**3-2**
OFFSET operator,**5-25**
with group,**4-35**
operand
in syntax,**2-13**
positioning,**6-3**
required typing,**6-2**
operation differences, processor modes,**7-14**
operation of assembler,**1-2**
operator
AND,**5-19, 11-7**
/,**5-15, 11-6**
EQ,**5-20, 11-7**
GE,**5-20, 11-7**
GT,**5-20, 11-7**
HIGH,**5-17, 11-5**
LE,**5-20, 11-7**
LENGTH,**5-28**
logical,**5-19**
LOW,**5-17, 11-5**
LT,**5-20, 11-7**
macro preprocessor,**11-4**
MASK,**5-31**
—, unary,**5-11 to 5-12**
MOD,**5-15, 11-6**
*,**5-15, 11-6**
NE,**5-20, 11-7**
NOT,**5-20, 11-6**
OFFSET,**5-25**
OR,**5-19, 11-7**
+, unary,**5-11 to 5-12, 11-6**
PTR,**5-23**
record,**5-31**
SEG,**5-26**
SHL,**5-16, 11-7**

operator (continued)
SHORT,**5-22**
SHR,**5-16, 11-7**
SIZE,**5-29**
THIS,**5-22**
TYPE,**5-27**
WIDTH,**5-32**
XOR,**5-19, 11-7**
operator precedence,**5-36**
OPTIMIZE assembler control,**7-7**
OR operator,**5-19, 11-7**
ORG directive,**4-39**
override
group,**5-24**
segment,**5-24, 6-11**
segment override checked against ASSUME,**6-12**

P

% (metacharacter),**10-2, 12-2 to 12-3**
PAGELENGTH assembler control,**7-8**
PAGEWIDTH assembler control,**7-8**
physical segment,**4-3**
+,**11-6**
binary,**5-12**
unary,**5-11**
with base & index register,**5-8**
position of operand,**6-3**
pre-defined macro function,**12-1**
precedence
of operators,**5-36**
prefix in syntax,**2-13**
primary assembler controls,**7-2, 7-4**
PRINT assembler control,**7-8**
PROC directive,**4-40**
default,**4-40**
processor mode
differences,**7-14**
V20,**7-14**
V25,**7-14**
program linkage,**4-7**
program linkage directive,**4-7**
program segmentation,**4-3**
PTR operator,**5-23**

PUBLIC directive,**4-43**
PURGE directive,**4-44**

Q quoted string
as expression operand,**5-5**

R real constant,**2-10**
record
differences from structure,**4-6**
as expression operand,**5-5**
initialization,**4-48**
name defined,**2-7**
similarities to structure,**4-6**
RECORD directive,**4-46**
record field
as expression operand,**5-6**
name defined,**2-7**
record operator,**5-31**
register
16-bit,**6-6**
8-bit,**6-6**
8087,**6-8**
assumed type,**6-3**
base,**6-6**
floating point,**6-8**
index,**6-6**
segment,**4-4, 6-6 to 6-7**
register indirect expression,**5-8**
relocatable expression,**5-3**
RELOCATION TYPE attribute,**3-4**
REPEAT macro function,**12-12**
RESTORE assembler control,**7-12**

S 17-bit number,**5-3**
SAVE assembler control,**7-13**
SEG operator,**5-26**
segment
addressability,**6-10**
default,**4-4**
logical,**4-4**
maximum number,**4-55**
nesting,**4-54**
override operator,**5-24**

segment (continued)
register,**4-4**
SEGMENT ADDRESSABILITY attribute,**3-5**
SEGMENT attribute,**3-4**
SEGMENT directive,**4-50**
segment name
defined,**2-7**
as expression operand,**5-6**
segment override,**6-11**
checked against ASSUME,**6-12**
segment register
default value,**4-11**
initialization,**4-22**
SEGMENT RELOCATION attribute,**3-4**
segmentation
directive,**4-3**
multiple segment definition,**4-53**
of program,**4-3**
SET macro function,**12-13**
SETIDB directive,**4-56**
SGNSFR directive,**4-8**
SHL operator,**5-16, 11-7**
SHORT operator,**5-22**
SHR operator,**5-16, 11-7**
SIZE operator,**5-29**
/ operator,**5-15, 11-6**
string
as expression operand,**5-5**
with DB directive,**4-18**
with DW, DD, DQ, DT directive,**4-19**
string relational macro function,**12-5**
STRUC directive,**4-58**
structure
differences from record,**4-6**
initialization,**4-59**
name defined,**2-7**
similarities to record,**4-6**
structure field
as expression operand,**5-7**
name defined,**2-7**
SUBSTR macro function,**10-3, 12-14**

subtraction operator
binary,**5-12**
unary,**5-11**
supported instruction set,**1-1**
supported microprocessors,**1-1**
symbol
EQU symbols,**2-8**
group name,**2-8**
instruction mnemonic,**2-6**
keyword,**2-3**
label,**2-6**
label with colon,**2-6**
macro preprocessor,**11-3**
record field name,**2-7**
record name,**2-7**
segment name,**2-7**
structure field name,**2-7**
structure name,**2-7**
variable,**2-6**
symbol in syntax,**2-2**
symbol table format,**8-5**
SYMBOLS assembler control,**7-8**
syntax
blank line,**2-13**
comment,**2-13**
continuation line,**2-14**
keyword,**2-13**
label,**2-12**
operand,**2-13**
prefix,**2-13**
symbol,**2-2**

T

THIS operator,**5-22**
TITLE assembler control,**7-13**
translation
acvtv20 tool,**E-16**
HP 64853 to HP @#!(PRODNUM),**E-1**
TYPE assembler control,**7-9**
TYPE attribute,**3-2**
TYPE operator,**5-27**

U unary minus,**5-11**
unary plus,**5-11**
UNREFERENCED_EXTERNALS assembler control,**7-9**
user-defined macro,**13-2**
user-defined macros,**13-1**

V V20 processor mode,**7-14**
V25 procossor mode,**7-14**
variable,**4-6**
variable name
defined,**2-6**
as expression operand,**5-7**

W WHILE macro function,**12-14**
WIDTH operator,**5-32**
WORKFILES assembler control,**7-9**

X XOR operator,**5-19, 11-7**
XREF assembler control,**7-10**