HP 64853
8086/88, 70108/116
Cross Assembler/Linker
User's Guide/Reference

Edition1

64853-90910
E0189
Printed in U.S.A. January 1989

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.

© Copyright 1987, 1989, Hewlett-Packard Company.

This document contains proprietary information, which is protected by copyright. All rights are reserved. No part of this document may be photocopied, reproduced or translated to another language without the prior written consent of Hewlett-Packard Company. The information contained in this document is subject to change without notice.

AdvanceLink, Vectra and HP are trademarks of Hewlett-Packard Company.

MS-DOS is a trademark of Microsoft Corporation.

UNIX is a registered trademark of AT&T.

VAX/VMS is a registered trademark of Digital Electronics Corporation.

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

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 or update was issued. Many product updates and fixes do not require manual, 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	**January 1989**	**64853-90910**	**E0189**
	(replaces	64853-90908	E0486
		64853-90909	E0288
	)		

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.

Using This Manual

This 8086/8088 Series Assembler/Linker manual is task/problem oriented. Tasks that you will perform with the assembler/linker are shown below:

Organization

Chapter 1 Overviews the HP 64000 Assembler/Linker and provides a brief introduction to the 8086/8088 Assembler/Linker.

Chapter 2 Provides a quick example by stepping through the process of assembling and linking three example program modules.

Chapter 3 Discusses how to use the assembler. Chapter 3 contains a description of the command and command line options that allows you to assemble your program modules. This chapter also describes the output files which are created by the assembler.

Chapter 4 Discusses how to link assembly language programs. Chapter 4 contains a description of the command and command line options that allow you to link your program modules. This chapter also describes the input files to the linker and the ouput files that are created by the linker.

Chapter 5. Discusses source file format and expressions. Chapter 5 contains information on the label, operation, operand and comment fields of an assembly language program source file, as well as information on symbolic terms, numeric terms, string constants, expression operators, and relocatable expressions.

Chapter 6 Discusses programming considerations. Chapter 6 contains information how and when to use special 8086/8088 pseudo instructions and keyword operators.

Chapter 7 Discusses pseudo instructions. Chapter 7 contains syntax descriptions of the HP 64000 and special 8086/8088 pseudo instructions.

Chapter 8 Discusses macros instructions. Chapter 8 contains a discussion of advantages and disadvantages of using macros and information on how to use macros.

Appendix A Contains a summary of the 8086/8088, 80186/80188, and 80286 instruction set.

Appendix B Contains information for 80286 protected mode assembler programming. This appendix includes descriptions of the special 80286 pseudo instructions, and a protected mode 80286 example program.

Appendix C Contains information on the 70116/70108 assembler, including microprocessor architecture, programming considerations, and the 70116/70108 instruction set summary.

Appendix D Contains information on 8087 architecture, programming considerations, special 8087 pseudo instructions, and the 8087 instruction set summary.

Appendix E Contains information on 8089 architecture, programming considerations, the special 8089 pseudo instruction, and the 8089 instruction set summary.

Appendix F Contains information on the 70320/70330 assembler, including microprocessor architecture, programming considerations, and the 70320/70330 instruction set summary.

Appendix G Contains a list of the assembler error messages, and gives a brief description of each.

Appendix H Contains descriptions of fatal and nonfatal link errors, why the error occurred, and how to correct it.

Appendix I Contains ASCII Conversion Table.

Index Contains topics of interest for quick location.

Contents

Chapter 1 Assembler/Linker Introduction

Chapter 2 A Quick Example

Chapter 3 Assembling Your Programs

Chapter 4 Linking Your Programs

Chapter 5 Source File Format And Expressions

Chapter 6 Programming Considerations

Chapter 7 Pseudo Instruction Summary

Chapter 8 Using Macro Instructions

Appendix A 8086/8088 Series Instruction Set Summary

Appendix B 80286 Programming

Appendix C 70108/70116 Programming And Instruction Set Summary

Appendix D 8087 Programming and Instruction Set Summary

Appendix E 8089 Programming and Instruction Set Summary

Appendix F 70320/70330 Programming And Instruction Set Summary

Appendix G Assembler Error Messages

Appendix H Linker Error Messages

Illustrations

Tables

1

Assembler/Linker Introduction

HP 64000 Assembler

The HP 64000 Assembler is a table-driven assembler to convert the users source program into relocatable data which can then be linked into executable machine language. The assembler is capable of producing code for virtually any microprocessor. Main assembler functions are the same regardless of the microprocessor being specified. Additional information is added for individual microprocessors in the form of tables. Tables are used to interpret processor-specific instructions and mnemonics.

Functional Description

The assembler covers the interactions required with the host system. Functions include reading and parsing the source program. All of the input and output file operations required by the source program, the resulting relocatable code and list files are handled by the assembler. The
assembler also:

1. Parses each line of the source program identifying the instruction for the specific processor.
2. Maintains a symbol table whose contents contain file symbols along with the associated values and symbol types.
3. Checks operand fields and flags errors if the syntax and/or address rules are not followed.

Assembler Operation

The HP 64000 Assembler reads the first line of the source file and looks for an assembler directive indicating which processor language is in the file that follows. The assembler then reads another file that contains the table for the indicated processor.

A simple interpreter is part of the assembler that handles the table code. The interpreter takes the specially coded table information and decodes it into instructions for the assembler. These instructions call up assembler functions, such as expression handlers, and object code generation. Instructions also allow for arithmetic operations and testing for Boolean results.

HP 64000 Linker

The linker is table-driven. Relocatable object modules are combined into one absolute file and executed in an emulation environment or used for programming PROM's (see Figure 1-1).

Table driven architecture allows the linker to support a variety of processors. The assembler directive in each relocatable file is used to identify the required processor tables. Each supported processor has a linker table used by the linker for configuration.

Linker tables contain two types of information: general information (such as word width and addressing space), and tables or sequences of instructions for the linker. The different instruction types and addressing modes allowed in the target processor correspond to the entry points in the linker table.

Functional Description

In preparing object code modules for the HP 64000 load program, the linker performs two functions: relocation and linking. These two functions are discussed in the following paragraphs.

Relocatable Code Areas

Several relocatable areas are provided by the HP 64000 assembler and linker. Assembler pseudos ORG, PROG, DATA and COMN define the relocatability of code. ORG defines code to be absolute or nonrelocatable. PROG and DATA are general purpose relocatable counters that allow user partitioned code to the loaded at different memory locations. For example, pseudos can load all program in ROM and all data in RAM. COMN specifies that the data be relocated to the same starting address as the COMN data from all other relocatable modules. When relocatable modules are linked, the user provides the starting addresses for the PROG, DATA and COMN relocatable code.

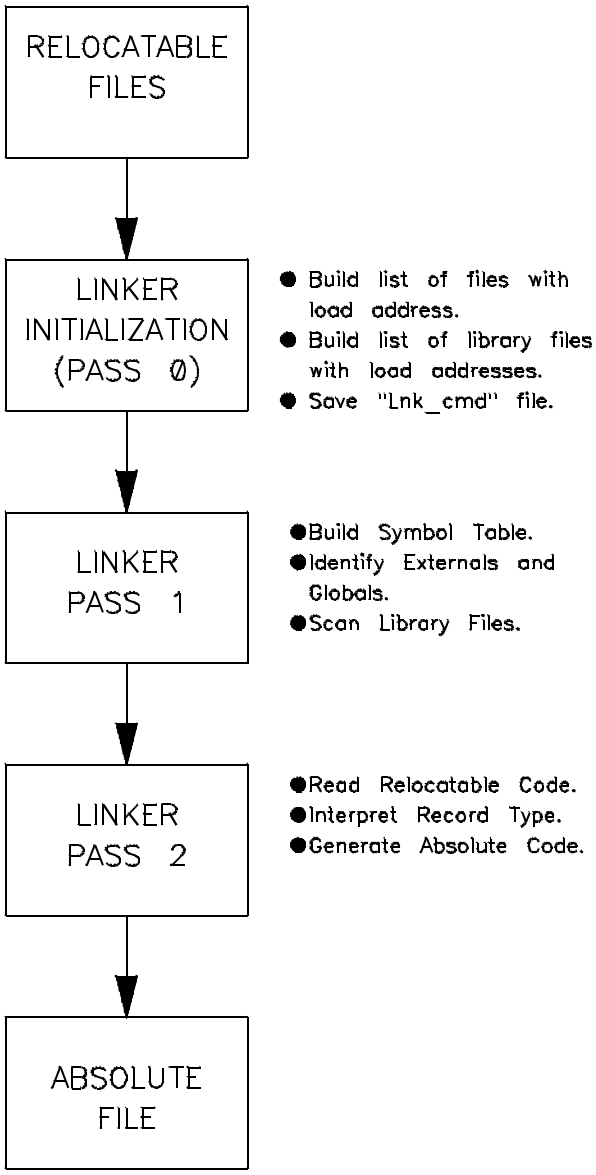

Figure 1-1. Linker Module Functions

Linking Relocatable Files For Emulation

Relocatable modules are linked and absolute and symbol files generated for use in emulation by the linker. During emulation the user may debug the program using symbols from the source program. The user does not have to know where in memory the linker stored the relocatable code. Any location in memory may be referred to by its symbolic name or by its absolute address.

The linker also creates a global symbol file for every link operation. This file is used by the emulator, along with assembler symbol tables, to provide symbolic debugging. It may also be used in subsequent links to preload the linker symbol table. This feature may be used in overlays and in reducing linking and download time for large pieces of software.

Introduction To The 8086/8088 Assembler/Linker

8086, 8088, 80186, 80188, 8089_86, and 8089_88 assemblers each support an address space of 20 bits. The 80186 assembler supports the 80286 microprocessor instruction set for the 8086 compatible mode, i.e., 20 bit address space. All of the above assemblers use the 8086 or 8088 linkers. Load addresses specified in assembler ORG statements or linker load address statements are input as 32 bit logical addresses (i.e., segment/offset). The upper 16 bits are the segment. The lower 16 bits are the offset. No delimiter exist between segment and offset. No procedure exists with this Assembler/Linker to input the physical address. The user must use the logical address.

80286 Protected Mode

The 80286 assembler supports the 80286 microprocessor instruction set in the protected mode. This means the assembler uses the 24 bit address space. Both in assembler ORG and linker address statements, the program must specify the 24 bit physical address of the object code. There is no procedure to input a logical address of selector and offset.

The 80286 microprocessor, upon reset, starts in the 20 bit address 8086 compatible mode. Transition from the 8086 compatible mode to the 80286 protected mode is performed program-

matically, i.e. the descriptor tables must be initialized and the machine status word set to the protected mode. Further, once invoked by the program, an exit cannot be done except using "RESET".

Note

The HP 64853's "80286" assembler was designed as an early support tool for the 80286 microprocessor running in the Protected Virtual Address Mode. In that mode there are some known limitations. (For instance, the "80286" assembler is not compatible with the HP 64228 - 80286 Emulator.) We encourage 80286 users to use the HP 64859 Cross Assembler/Linker instead. The HP 64859 product supports the 80286 in both the Real Address Mode and the Protected Virtual Address Mode.

Processor Directives

When developing an assembly language program, whether a coprocessor is involved or not, select the appropriate directive from: "8086", "8088", "80186", "80188", "80286", "8089_86", "8089_88", "70108", "70116", "70320", or "70330". The directive number should be that of the processor being used.

Host-Specific Issues

The preceeding sections have discussed the HP 64000 assembler and linker in a general fashion about features of the assembler that are the same for the different hosts. There are some host-specifics that make the assembler appear differently on the different hosts. These host-specific issues have to do not with the operation and function of the assembler and linker, but instead of the machine-specific command line interface and file name conventions. Throughout the rest of the manual, these differences will be a described where appropriate. For quick reference, however, refer to the following table for a summary of the interface differences.

COMMANDS			
HP–UX	**MS–DOS**	**HP 64000**	**VAX/VMS**
asm lnk	asm lnk	assemble link	asm lnk
OPTIONS			
HP–UX	MS–DOS	HP 64000	VAX/VMS
-o -l -n -e -x -t -c -v	/o /l /n /e /x /t /c /v	listfile list nolist expand xref nocode no_overlap (no option)	/OUTPUT /LIST /NOLIST /EXPAND /XREF /NOCODE /NOMEM_OVRLP /VERBOSE
FILE NAMES			
HP–UX	MS–DOS	HP 64000	VAX/VMS
filename.S filename.R filename.A filename.X filename.L filename.K filename.O	filename.S filename.R filename.A filename.X filename.L filename.K filename.O	Filename:source Filename:reloc Filename:asmb_sym Filename:absolute Filename:link_sym Filename:link_com Filename:listing	FILENAME.S FILENAME.R FILENAME.A FILENAME.X FILENAME.L FILENAME.K FILENAME.LIS

2

A Quick Example

Introduction

This chapter contains a quick step-by-step example of the process of assembling and linking several program modules. You will be acquainted with the HP 64000, 8086/8088 Cross Assembler/Linker and shown the steps in assembling and linking program modules.

Objectives of the Example Program

Topics listed below are covered by the example program. They are provided to indicate why the example program is written the way it is. The example program shows some of the basic features of the 8086/8088 Cross Assembler/Linker. The example program will:

- Contain 8086/8088 assembly language instructions written to use PROG, DATA, and COMN relocatable code areas (segments).
- Contain a few of the most used special 8086/8088 pseudo instructions.
- Show how the HP 64000 relocatable code areas (PROG, DATA, and COMN) are used in linking relocatable program modules.
- Contain an example of a simple macro definition.
- Show how to link two or more program modules.
- Show how to link relocatable files from a library file.

Note

The example programs in this chapter (and in the rest of the manual) are not resident in the assembler/linker software. Therefore, if you wish to step through the examples on your system exactly as they are shown in the manual, you must enter the program module source files.

Description of the Example Program

The example program moves data from three different memory locations to a fourth memory location. The program is divided into three modules to show how several program modules are linked together.

The MOV_MESG program module is made up of a data table containing the messages to be transferred. The main program defines a macro and calls "transfer" and "delay" subroutines. A memory storage location to which the messages will be transferred is also included.

The TRANSFER program module contains the TRANSFER subroutine which is called by the main program. The TRANSFER subroutine transfers data between two memory locations to the destination memory location. The subroutine will get the addresses of the beginning and end of the message to be transferred by reading the contents of two "parameter passing" memory locations. These "parameter passing" memory locations are defined in the MOV-MESG program module.

The DELAY program module contains the DELAY subroutine called by the main program. The DELAY subroutine causes a delay whose length is specified by the contents of a "parameter passing" memory location. The parameter passing memory location is defined in the MOV_MESG program module. The DELAY program module will be placed in an example library file to show how to link relocatable files from a library file.

The MOV_MESG Program Module

The example program provided will move three messages contained in a data table to another memory location. The messages are labeled MESG0, MESG1, and MESG2. Ends of each message are also labeled so that the program will know how many words of data to transfer. The destination memory location is labeled VIDEO_RAM.

The example program will (1) move the first message to VIDEO_RAM, where it will be displayed for a short length of time. The example program will then (2) move the second message to VIDEO_RAM, where is is displayed for a shorter length of time. Finally, the example shows a (3) movement of the third message to VIDEO_RAM, where it is displayed for an even shorter length of time. The program will then loop back to display the second and third messages, one after the other, repeatedly. The MOV_MESG source file is shown below.

```
"8086"
                GLB     MESG_OFFSET,MESG_END_OFFSET
                GLB     DELAY_COUNT,VIDEO_RAM,START
                EXT     TRANSFER,DELAY

                DATA
DATA_TABLE
MESG_OFFSET             DWS     1
MESG_END_OFFSET                 DWS     1
DELAY_COUNT             DWS     1
MESG0   DB                      "THIS EXAMPLE PROGRAM "
                DB              "MOVES SEVERAL MESSAGES "
                DB              "FROM A DATA TABLE TO A "
                DB              "MEMORY LOCATION"
MESG0_END
MESG1           DB              "THE FIRST MESSAGE IS "
                DB              "DISPLAYED FOR A MEDIUM "
                DB              "LENGTH OF TIME       "
                DB              "           "
MESG1_END
MESG2   DB              "THE SECOND MESSAGE IS "
                DB              "DISPLAYED FOR A SHORTER "
                DB              "LENGTH OF TIME     "
                DB              "           "
MESG2_END
```

Figure 2-1. The MOV_MESG Source File

2-4 A Quick Example

```
                PROG
                ASSUME CS:PROG,DS:DATA,ES:COMN
START           MOV             AX,SEG DATA_TABLE
                MOV             DS,AX
                MOV             AX,SEG VIDEO_RAM
                MOV             ES,AX
SET_UP  MACRO   &MESG_NO,&MESG_NO_END,&DELAY_COUNT
                MOV             MESG_OFFSET,OFFSET &MESG_NO
                MOV             MESG_END_OFFSET,OFFSET &MESG_NO_END

                MOV     DELAY_COUNT, &DELAY_COUNT
                CALL    TRANSFER

                CALL    DELAY
                MEND
                SET_UP  MESG0,MESG0_END,# 5FFH

REPEAT  SET_UP  MESG1,MESG1_END,# 4FFH
                SET_UP  MESG2,MESG2_END,# 3FFH
                JMP             REPEAT
                COMN
VIDEO_RAM       DDS             40H
                END
```

Figure 2-1. The MOV_MESG Source File (Cont'd)

Declaring Global Symbols

The MOV_MESG program module first declares global symbols using the GLB pseudo instruction. The labels MESG_IOFFSET, MESG_END_OFFSET, and VIDEO_RAM are declared as global symbols because they are to be used by the TRANSFER subroutine module. MESG_OFFSET, and MESG_END_OFFSET are memory locations which contain offset addresses of the beginning and end of the messages to be transferred to VIDEO_RAM.

The label DELAY_COUNT is declared as a global symbol because the DELAY library subroutine will use the contents of the DELAY_COUNT memory location as a count value. START is declared global because it labels the starting address for program execution.

Declaring External Symbols

The external (EXT) pseudo instruction allows use of labels which are defined in other program modules. In the MOV_MESG program module, CALL DELAY and CALL TRANSFER instructions use labels defined in the DELAY and TRANSFER program modules, respectively. Therefore, DELAY and TRANSFER must be declared as external symbols.

DATA Relocatable Code Area

The DATA code area (segment) contains memory locations through which values are passed to the DELAY and TRANSFER subroutines. The DWS special 8086/8088 pseudo instruction reserves 1 word for memory for each of the labels MESG_OFFSET, MEST_END_OFFSET, and DELAY_COUNT.

The DATA code area also contains the three messages to be displayed in the VIDEO_RAM memory location. ASCII string messages are defined with the DB (Define Byte) special 8086/8088 pseudo instruction. These three messages are labeled MESG0, MESG1, and MESG2. Ends of the three messages are labeled MESG0_END, MESG1_END, and

MESG2_END to allow the program to determine how many words to transfer.

PROG Relocatable Code Area

Program code for all three modules appears in the PROG code area (segment). The ASSUME pseudo instruction tells the assembler that the segment values of the addresses corresponding to the HP 64000 code areas PROG, DATA, and COMN are in segment registers CS, DS, and ES respectively. As an example, when we load the MESG_OFFSET location with data, the assembler will assume that DS contains the segment value of the MESG_OFFSET address (because MESG_OFFSET is in the DATA code area).

The program begins by loading the processors segment registers with values corresponding to the DATA and COMN segments. It is not necessary to initialize the CS register since all program instructions will be linked to the same PROG segment. Register CS will not be used in calculating addresses of memory references in the PROG segment.

SET_UP Macro Definition

Next the program must load the "parameter passing" memory locations with values for the TRANSFER and DELAY subroutines. Since these memory locations are loaded three times, each time using different offset addresses, a macro definition eliminates the need for writing the same set of instructions three times.

Macro parameters, &MESG_NO, &MESG_NO_END, and &DELAY_COUNT, allows variables to be created whose values are assigned when the macro instruction is used. Since the TRANSFER and DELAY subroutines are called each time after the MESG_OFFSET, MESG_END_OFFSET, and DELAY_COUNT "parameter passing" locations are loaded, the CALL
instructions are also included in the macro definition.

After the SET_UP macro has been defined, the program uses the macro instruction three times to transfer the three messages to the VIDEO_RAM memory location.

COMN Relocatable Code Area

The special 8086/8088 DDS (Define Double-Word Storage) pseudo instruction reserves 40H double-words of memory (256 bytes). VIDEO_RAM labels the start of this destination memory location.

The 'TRANSFER " Program Module

The TRANSFER program module contains the subroutine called by the main program. The TRANSFER subroutine moves data from the address whose offset is in location MESG_OFFSET through the address whose offset is in location MESG_END_OFFSET on then to the destination memory location VIDEO_RAM. THE TRANSFER program module source file is shown below.

```
"8086"
                GLB             TRANSFER
                EXT             DS:MESG_OFFSET WORD
                EXT             DS:MESG_END_OFFSET WORD
                EXT             ES:VIDEO_RAM

TRANSFER        MOV             CX,MESG_END_OFFSET
                SUB             CX,MESG_OFFSET
                SHR             CX,1
                CLD
                MOV             SI,MESG_OFFSET
                LEA             DI,VIDEO_RAM
                REP MOVSW
                RET
```

Figure 2-2. The TRANSFER Source File

Notice TRANSFER program module does not specify a PROG, DATA or COMN segment. The default segment is the PROG relocatable code area. For this reason, the instructions in this subroutine are actually in the PROG program area.

Notice also there is no ASSUME pseudo instruction in this module. That is because all the data location referencing labels are external. Because we can include segment information in EXT declarations, it is not necessary to use the ASSUME pseudo instruction in this module.

You may also specify the "type" of an external label with the EXT pseudo instruction. The "type" WORD is specified for external symbols MESG_OFFSET and MESG_END_OFFSET to tell the assembler that the memory locations are 16-bits wide.

The "DELAY" Program Module

The DELAY program module contains the DELAY subroutine called by the main program. The DELAY

subroutine displays the various messages for the length of time specified by the contents of the DELAY_COUNT memory location. The DELAY program module source file is shown below.

As with the TRANSFER program module, the only data loca-

```
"8086"

                GLB             DELAY
                EXT             DS:DELAY_COUNT WORD

DELAY           MOV             CX,DELAY_COUNT
OVER            MOV             BX,DELAY_COUNT
UNDER       DEC         BX
                JNZ             UNDER
                DEC             CX
                JNZ             OVER
                RET
```

Figure 2-3. The DELAY Source File

tion referencing operand is an external label. The segment to be assumed with the label DELAY_COUNT, and the "type" are defined in the EXT pseudo instruction operand.

Also, since no relocatable code area is defined, these instructions are in the PROG code area by default.

Assembling Program Module Source Files

Assembling program module source files creates relocatable files, assembly symbol files, and optionally, an assembler listing file. The commands to assemble source files for the different hosts are shown below:

HP-UX

asm -oe movemesg.S > movmesg.O

MS-DOS

asm /oe movmesg.S > movmesg.O

HP 64000

assem ble MOV_MESG **listfile** MOV_MESG
options expand

VAX/VMS

asm/expand/output= movmesg.lis movmesg.s

The assemble commands used in this example specify that an assembly "listfile" be created. The commands also specify that the "listfile" be an expanded listing, i.e., a listing that shows all the instructions caused by using a macro instruction and all the object code generated by the
assembler.

TRANSFER and DELAY program modules for each host are assembled in the same way.

Viewing Assembler Listing Files

Using the "listfile" option when assembling programs causes assembler error messages to be listed in the listing file instead of the standard output. The commands which
allows you to look at the assembler listing file are shown below:

HP-UX

cat movmesg.O

MS-DOS

type movmesg.O

HP 64000

edit MOV_MESG:**listing**

VAX/VMS

type movmesg.s

Assembler listing files for the TRANSFER and DELAY program modules may be viewed in the same way.

Program Module Assembly Listings

Listfiles for the MOV_MESG, TRANSFER, and DELAY program modules follow. The column captions are not produced by the assembler. They were put there as an aid in viewing the listings.

```
LOCATION  OBJECT CODE     SOURCE  LINE

                          1 "8086"
                          2           LIST    50
                          3           GLB     MESG_OFFSET,MESG_END_OFFSET
                          4           GLB     DELAY_COUNT,VIDEO_RAM,START
                          5           EXT     TRANSFER,DELAY
                          6
                          7           DATA
0000                      8           DATA_TABLE
0000                      9            MESG_OFFSET              DWS     1
0002                      10          MESG_END_OFFSET          DWS     1
0004                      11          DELAY_COUNT              DWS     1
                          12
0006    5448495320        13          MESG0   DB      "THIS EXAMPLE PROGRAM"
000B    4558414D50
0010    4C45205052
0015    4F4752414D
001A    20
001B    4D4F564553       14          DB      "MOVES SEVERAL MESSAGES"
0020    2053455645
0025    52414C204D
002A    4553534147
002F    455320
0032    46524F4D20       15          DB      "FROM A DATA TABLE TO"
0037    4120444154
003C    4120544142
0041    4C4520544F
0046    204120
0049    4D454D4F52       16          DB      "MEMORY LOCATION"
004E    59204C4F43
0053    4154494F4E
0058    2020202020
005D    20
005E                     17          MESG0_END
                         18
```

Figure 2-4. The MOV_MESG Assembly Listing

```
LOCATION OBJECT CODE    SOURCE  LINE
005E    5448452046      19 MESG1        DB      "THE FIRST MESSAGE IS"

0063    4952535420
0068    4D45535341
006D    4745204953
0072    20
0073    444953504C      20              DB      "DISPLAYED FOR A MEDIUM"
0087    554D20
008A    4C454E4754      21              DB      "LENGTH OF TIME"
008F    48204F4620
 0078   4159454420
094 5   4494D4520
0099    2020202020
009E    202020
00A1    2020202020      22              DB      "                "
00A6    2020202020
00AB    2020202020
00B0    2020202020
00B5    20
00B6                    23 MESG1_END
                        24
00B6    5448452053      25 MESG2        DB      "THE SECOND MESSAGE IS"
00BB    45434F4E44
00C0    204D455353
00C5    4147452049
00CA    5320
00CC    444953504C      26              DB      "DISPLAYED FOR A SHORTER"
00D1    4159454420
00D6    464F522041
00DB    2053484F52
00E0    54455220
00E4    4C454E4754      27              DB      "LENGTH OF TIME"
00E9    48204F4620
007D    464F522041
0082    204D454449
```

Figure 2-4. The MOV_MESG Assembly Listing (Cont'd)

2-14 A Quick Example

```
LOCATION  OBJECT CODE    SOURCE  LINE
00EE      54494D4520
00F3      2020202020
00F8      2020202020     28                DB              "                "
00FD      2020202020
0102      2020202020
0107      2020202020
010C      20
010D                     29 MESG2_END
                         30
                         31                PROG
                         32                ASSUME CS:PROG,DS:DATA,ES:COMN
                         33
0000      B80000         34 START          MOV             AX,SEG DATA_TABLE
0003      8ED8           35                MOV             DS,AX
0005      B80000         36                MOV             AX,SEG VIDEO_RAM
0008      8EC0           37                MOV             ES,AX
                         38
                         39 SET_UP         MACRO   &MESG_NO,&MESG_NO_END,
                                                           &DELAY_COUNT
                         40                MOV             MESG_OFFSET,OFFSET &MESG_NO
                         41                MOV             MESG_END_OFFSET,OFFSET
                                                           &MESG_NO_END
                         42                MOV             DELAY_COUNT, &DELAY_COUNT
                         43                CALL            TRANSFER
                         44                CALL            DELAY
                         45                MEND
                         46
000A                     47                SET_UP  MESG0,MESG0_END,# 5FFH
000A      C7060000       +                 MOV             MESG_OFFSET,OFFSET MESG0
000E      0006
0010      C7060002       +                 MOV             MESG_END_OFFSET,OFFSET
                                                           MESG0_END
0014      005E
0016      C7060004FF     +                 MOV             DELAY_COUNT, # 5FFH
001B      05
```

Figure 2-4. The MOV_MESG Assembly Listing (Cont'd)

```
LOCATION OBJECT CODE    SOURCE LINE

01C E   80000           +               CALL            TRANSFER
001F    E80000          +               CALL            DELAY
0022                    48REPEAT        SET_UP  MESG1,MESG1_END,# 4FFH
0022    C7060000        +               MOV             MESG_OFFSET,OFFSET MESG1
0026    005E
0028    C7060002        +               MOV             MESG_END_OFFSET,OFFSET
                                                        MESG1_END
002C    00B6
002E    C7060004FF      +               MOV             DELAY_COUNT, # 4FFH
0033    04
0034    E80000          +               CALL            TRANSFER
0037    E80000          +               CALL            DELAY
003A                    49              SET_UP  MESG2,MESG2_END,# 3FFH
003A    C7060000        +               MOV             MESG_OFFSET,OFFSET MESG2
003E    00B6
0040    C7060002        +               MOV             MESG_END_OFFSET,OFFSET
                                                        MESG2_END
0044    010D
0046    C7060004FF      +               MOV             DELAY_COUNT, # 3FFH
004B    03
004C    E80000          +               CALL            TRANSFER
004F    E80000          +               CALL            DELAY
0052    EBCE            50              JMP             REPEAT
                        51
                        52              COMN
0000                    53 VIDEO_RAM    DDS             40H
                        54              END
Errors=  0
```

Figure 2-4. The MOV_MESG Assembly Listing (Cont'd)

2-16 A Quick Example

```
LOCATION  OBJECT CODE    SOURCE LINE

                         1 "8086"
                         2          GLB             TRANSFER
                         3          EXT             DS:MESG_OFFSET WORD
                         4          EXT             DS:MESG_END_OFFSET WORD
                         5          EXT             ES:VIDEO_RAM
                         6
0000      8B0E0000       7 TRANSFER MOV CX,MESG_END_OFFSET
0004      2B0E0000       8          SUB             CX,MESG_OFFSET
0008      D1E9           9          SHR             CX,1
000A      FC             10         CLD
000B      8B360000       11         MOV             SI,MESG_OFFSET
000F      8D3E0000       12         LEA             DI,VIDEO_RAM
0013      F3A5           13         REP MOVSW
0015      C3             14         RET
Errors= 0
```

Figure 2-5. The TRANSFER Assembly Listing

```
LOCATION  OBJECT CODE   SOURCE  LINE

                        1 "8086"
                        2                GLB             DELAY
                        3                EXT             DS:DELAY_COUNT WORD
                        4
0000      8B0E0000      5 DELAY  MOV                     CX,DELAY_COUNT
0004      8B1E0000      6 OVER           MOV             BX,DELAY_COUNT
0008      4B            7 UNDER          DEC             BX
0009      75FD          8                JNZ             UNDER
000B      49            9                DEC             CX
000C      75F6          10               JNZ             OVER
000E      C3            11               RET
Errors=  0
```

Figure 2-6. The DELAY Assembly Listing

Creating an Example Library File

One of the goals of this example chapter was to show how to link library files and ordinary relocatable files. We also decided that the DELAY module would be put into a library file whose name is host-dependent. The following are the various host commands to create the library file.

HP-UX

cat delay.R > > exlib.R

MS-DOS

type delay.R > > exlib.R

HP 64000

library DELAY **to** EX_LIB

VAX/VMS

append/new delay.R exlib.R

Linking Program Module Relocatable Files

Linking is the process in which our three program modules will be joined together to form a single program. The result of linking relocatable program modules is an absolute file which contains object code to be executed by the microprocessor.

The linker permits combining any number of relocatable files, no-load files, and linker symbol files into an absolute file. It also allows specification of load addresses of the relocatable program areas in program modules.

Linker questions and answers are explained below.

Calling the Linker

The host-specific commands to access the linker are shown below:

HP-UX

llnk -o > demo.O

MS-DOS

lnk /o > demo

HP 64000

link listfile DEMO

VAX/VMS

lnk /output= demo.lisl

Answering Linker Questions

This section will answer the linker questions for the example program, explaining the reason each answer.

Object files? movmesg.R,transfer.R (HP-UX, MS-DOS)

MOV_MESG, TRANSFER (HP 64000)

movmesg,transfer (VAX/VMS)

Answer the object files question with the names of the relocatable program modules. Answering "two" relocatable files causes the linker to join back-to-back relocatable code program module areas. If you prefer to specify the load addresses of relocatable files individually, only "one" file at a time should be answered for this question. Press < RETURN> . Next question:

Library files? exlib.R (HP-UX, MS-DOS)

EX_LIB (HP 64000)

exlib (VAX/VMS)

The library files question gives you the opportunity to specify a library of relocatable program modules. The linker will attempt to find modules containing labels from the program modules that have not, as yet, been defined in the files answered in the first question. In our example, the linker will search the EX_LIB library for any relocatable module which defines the label DELAY. The relocatable file fitting that definition is relocatable program module DELAY (included earlier in the EX_LIB library).

If the label DELAY happens to be defined in two of the library's relocatable program modules, a link error will

occur. Press < RETURN> . Next question:

Load addresses: PROG,DATA,COMN=

000001000H,000002000H,000003000H

The load address question allows specification of addresses of the relocatable segments of PROG, DATA, and COMN. (Any ORG pseudo instructions in the relocatable program modules defines the address of the ORG absolute code area.) Press < RETURN> . Next question:

More files? no

Answer "no" to the more files question. We have already specified all the relocatable files to be linked. If we had answered the object files question above with only the MOV_MESG relocatable module, we would have to answer this question with "yes" to provide the linker information for the TRANSFER relocatable program module. Press < RETURN> . Next question:

list,xref,overlap_check,comp_db=

on off on off

Default answers for this question are sufficient for this example. Just press the < RETURN> key in response to this question. Next question:

Absolute file name= demo.X (HP-UX, MS-DOS)

DEMO (HP 64000)

demo (VAX/VMS)

You must answer this question with a valid file name. We will use **demo** as an absolute file name. The linker will then(1) create an absolute file with the name with a host-specific extension; (2) create a linker command file (whose contents are the answers just given); and (3) create a linker symbol file with the

same name (demo) and host-specific extension.

Linker Listing File

To see the results of the link we just specified, let's look at the linker listing file shown below.

```
FILE/PROG NAME          PROGRAM      DATA       COMMON
MOV_MESG:USERID         0000 1000    00002000   00003000 Tue, 19 Mar1985
TRANSFER:USERID         0000 1054                          Tue, 19 Mar 1985
next address            0000 106A    000210D    00003100

Libraries
EX_LIB:USERID
DELAY:USERID            0000 06A                      Tue, 19 Mar 1985
next address            0000 1079    0000210D   00003100
XFER address=  00000000 Defined by DEFAULT
No. of passes through libraries= 1
absolute & link_com file name= DEMO:USERID
Total#  of bytes loaded=  00000286
```

Figure 2-7. The DEMO Linker Listfile

Notice in Figure 2-7 above that the PROG, DATA and COMN areas of the MOV_MESG relocatable file have been linked to the addresses specified in the load addresses linker question. Also notice that the TRANSFER and DELAY program modules have been linked at the PROG addresses immediately following PROG memory space taken up by the program MOV_MESG. This linker listing file shows that library files are linked behind any other object files that have been specified.

This now completes this quick example of assembler/linker program modules.

Notes

3

Assembling Your Programs

Introduction

This chapter provides a description of the HP 64000 assembler and its operation. A description of the assembler options and their use is provided.

Functional Components Of The Assembler

The assembler has five major functional components: initialization, pass 1, pass 2, error/asm_sym generation (pass 3), and cross-reference listing (pass 4). These functional components are used by the assembler to make source code files for specific processors and produce relocatable object code.

Initialization

The assembler initialization function acquires the necessary information for setting the proper configuration and specific personality for the assembler. The information is

- input file
- listing file
- options (list, nolist, expand, nocode, xref)
- assembler personality (e.g. directive "8086")

Pass 1

Pass 1 performs the standard assembler pass 1 functions of reading source files, keeping program counters, and building the symbol table.

Pass 2

Pass 2 performs the standard pass 2 functions of reading source files, keeping program counters, using the symbol table, and generating relocatable code.

Pass 3

Pass 3 performs the functions of printing error text, the fatal error segment, and generating a sorted assembler symbol file.

Pass 4

Pass 4 generates a cross-reference map if required.

Input/Output Files

Source Input File

Input to the assembler is a source file. Source file filenames take different forms depending upon the host computer.

HP-UX, MS-DOS, VAX/VMS	filename.S (The *.S* extension need not be specified. Avoid filenames that result in confusing output filenames. *open.asm* is an example.)
HP 64000	Filename.source (The extension *:source* is the default and need not be specified.)

Source files consist of the following:

Example	**Description**
"8086"	Assembler directive.
Source Code	Consisting of source statements and pseudo instructions; refer to chapter 7.

Assembler Output Files

The assembler produces files stored under the same name as the source file, with host-dependent extensions. The assembler produces three files: a relocatable file, an assembly symbol file, and an optional listing file. If any of these three files exist before assembly of the source file, the assembler will replace them with new files.

Relocatable File

filename.R (HP-UX, MS-DOS, VAX/VMS)
Filename:reloc (HP 64000)

The relocatable object module is in a format that can be processed by the linker. If the relocatable file already exists, it will be overwritten.

Assembly Symbol File

filename.A (HP-UX, MS-DOS, VAX/VMS)
Filename:asmb_sym (HP 64000)

The assembly symbol file contains all local symbols defined in the source file. The assembly symbol file can be used for symbolic debugging. If the file already exists, it will be overwritten.

ListFile (Optional)

filename.O (HP-UX, MS-DOS, VAX/VMS)
Filename:listing (HP 64000)

Listfile is an optional listing. It can be directed to a line printer, stored in a file, or displayed on your terminal. If a listfile already exists, it will be overwritten. The listing can include:

- Source statements with object code.
- Error messages.
- Summary of errors with a description list.

- Symbol cross-reference list composed of all symbols except local macro labels and parameters. The symbol table format is discussed in the chapter titled "Linking Your Programs."

The cross-reference list is alphabetically sorted by symbol name.

Specifying Page Length of Assembler Output Listing

Assembler output listing can be controlled to limit the number of lines appearing on each page of the output. The following rules apply.

1. Output listing syntax is: **LIST < limit>**
2. Effective values for < limit> are 5 thru 127. If a number less than five is used, the first page of output will have six lines, and succeeding pages will have five lines.
3. The instruction cannot be included in the list options on the directive line. Rather, it must be treated as an opcode with an operand. The instruction will not be accepted by the assembler if it is entered from the keyboard.

Assembling The Program

Once a source file exists, it can be assembled using the host-specific command for invoking the assembler. A syntax description follows for assembler activation on the various hosts.

asm (HP-UX)

Syntax asm [-l] [-n] [-x] [-e] [-t] [-o] < file>

Syntax Definition

Definition for syntactical term and output default are as follows:

< file>	Source file to be assembled.
output default	Listing files are not produced unless list-file output is specified by the [-o] option. In this case, the listfile appears on 'stdout'. To direct output into a file, use the shell redirection "> filename".

Option Defintions

asm recognizes the following options, the first of which must be preceded by a dash (-); however, options can be concatenated (for instance, -ox):

-o	Listfile on (default is off).
-l	Overrides all list and nolist pseudos in the source file and forces listing of all lines.
-n	Overrides all list and nolist pseudos in the source file and forces no listing of all lines.
-e	Overrides all list and expand pseudos in the source file and forces expanded list of all areas selected for listing in source file.
-t	Causes assembly with no object code generation or relocatable file creation.

asm (HP-UX) Cont'd

-x	Causes a cross-reference to be printed to the < list destination> .

Examples

asm dat1.S

Assembles source file dat1.S; no output listing.

asm -ox dat1.S > dat1.O

Assembles source file dat1.S; output listing to file dat1.O with a symbol cross-reference table.

asm -t dat1.S

Assembles source file dat1.S; producing no relocatable file and listing only errors to the display.

Note

asm resides in public directory **/usr/hp64000/bin**. If /usr/hp64000/bin is in the user's directory path, the assembly can be run using only the command "**asm**". The assembler personality tables are located in **/usr/hp64000/tables**.

asm (MS-DOS)

Syntax asm [/l] [/n] [/x] [/e] [/t] [/o] < file>

Syntax Definition

Syntax definition of terms and output defaults are as follows:

< file>	Source file to be assembled.
output default	Listing files are not produced unless list-file output is specified by the [/0] option. In this case, the listfile appears on **'stdout'.** To direct output into a file, use the command parser redirection "**> filename**".

Option Definitions

asm recognizes the following optiions which must be preceded by a slash (/). In addition, options can be concatenated (e.g., /ox).

/o	Listfile **on** (default is off).
/l	Overrides all list and nolist pseudos in the source file and forces listing of all lines.
/n	Overrides all list and nolist pseudos in the source file and forces no listing of all lines.
/e	Overrides all list and expand pseudos in the source file and forces expanded list of all areas selected for listing in source file.

asm (MS-DOS) Cont'd

/t	Causes assembly with no object code generation and no relocatable file creation.
/x	Causes a cross-reference to be printed to the < list destination>

Examples

asm dat1.s

Assembles source file dat1.s; no output listing

asm /ox dat1.S> dat1.O

Assembles source file dat1.S; output listing to file dat1.O with a symbol cross-reference table.

asm /t dat1.S

Assembles source file dat1.S; producing no relocatable file and listing only errors to the display.

Note

The assembler resides in directory **\HP64700\BIN**. If \HP64700\BIN is in the user's directory path, the assembly can be run using only the command "**asm**". The assembler personality tables are located in **\HP64700\TABLES.**

assemble (HP 64000)

Syntax assemble < FILE> [listfile < list destination>]

options [list | nolist] [expand] [nocode] [xref]

Syntax Definition

< FILE>	Source file to be assembled. Selects file or device for listing output.
< list destination>	See < list destination> under "defaults" heading.

Option Definitions

Allows user to override listing options specified in the source file.

list	Overrides all list and nolist pseudos in the source file and forces listing of all lines.
nolist	Overrides all list and nolist pseudos in the source file and forces no listing except errors.
expand	Overrides all list and expand pseudos in the source file and forces expanded list of all areas selected for listing in source file.
nocode	Causes assembly with no object code generation or relocatable file creation.
xref	Causes a cross-reference to be printed to the < list destination> .

Default Values

< list destination>

By default, listing output is sent to the listfile default specified in last userid command. If no listfile default was specified in the last userid command, the listfile default is null.

options

If "options" is not selected, all listings occur as per pseudo instructions specified in the source file. If "options" is selected, and nothing else, then

- An output listing of the source program with object codes and error messages will be made.
- No expansion of macros and multiple-byte pseudo instructions will occur.
- No symbol cross-reference listing will be made.

Examples

assemble SAM

Assembles source file SAM; output listing to specified listfile default.

assemble SAM listfile CHARLEY

Assembles source file SAM; output listing to file CHARLEY of type listing.

assemble SAM listfile display options nolist nocode

Assembles source file SAM; producing no relocatable file and listing only errors to the display.

asm (VAX/VMS)

Syntax asm [options] < file>

Syntax Definition

Definiton for syntactical term and output default are as follows:

< file>	Source file to be assembled.
output default	Listing files are not produced unless list-file output is specified by the option /output [= < file>].

Option Definitions

asm recognizes the following options which must be preceded by a slash (/); however, options can be concatenated (for instance, /nolist/nocode filename).

/output [= < file>]

Listfile on (default is off). If the option "/output" is used with no "= ", the listfile will be placed in a file of the same basename with a .lis extension.

/list Overrides all list and nolist pseudos in the source file and forces listing of all lines.

/nolist Overrides all list and nolist pseudos in the source file and forces no listing except errors.

/expand Overrides all list and expand pseudos in the source file and forces expanded list of all areas selected for listing in source file.

asm (VAX/VMS) Cont'd

/nocode	Causes assembly with no object code generation or relocatable file creation.
/xref	Causes a cross-reference to be printed to the < list destination> .

Examples

asm dat1.s

Assembles source file dat1.s; no output listing.

asm/output dat1.s

Assembles source file dat1.s; output listing to file dat1.lis.

asm/nocode dat1.s

Assembles source file dat1.s; producing no relocatable file and listing only errors to the display.

Output Listing

An example of an assembler output listing is given in Figure 3-2, using the source program example listed in Figure 3-1. Figure 3-3 shows an assembler output listing that contains error messages.

```
"8086"    XREF

          GLB     INIT,SET,POOL,STOP
INIT              MOV     AX,0000H ; The INIT portion initializes registers.
                  MOV     BX,0005H;SET increments registers AX, BX and DX.
                  MOV     CX,0005H;SET compares register CX to zero. CX is
                  MOV     DX,0000H;compared to zero by the LOOP instruction.
SET               INC     AX        ;If CX is not zero it jumps to SET. After
                  INC     BL        ;CX becomes zero, the high byte of register
                  INC     DX        ;B (BH) is incremented and compared to five.
                  LOOP    SET       ;While BH is < five, POOL is repeated. When
POOL              INC     BH        ;BH equals five, the program is stopped.
                  CMP     BH, 05H   ;The program can be resumed without going
                  JNZ     POOL      ;into the weeds, because the next instruction
STOP              HLT               ;is an unconditional jump to INIT.
                  JMP     INIT
```

Figure 3-1. Source Program Example

```
FILE:     KW86                          HEWLETT-PACKARD:     8086 Assembler

LOCATION OBJECT CODE   LINE   SOURCE LINE                          COMMENTS

                       1      "8086"          XREF
                       2
                       3
                       4              GLB             INIT,SET,POOL,STOP
                       5
                       6
0000     B80000        7      INIT    MOV             AX,0000H     ;
0003     BB0000        8              MOV             BX,0000H     ;
0006     B90500        9              MOV             CX,0005H     ;
0009     BA0000        10             MOV             DX,0000H     ;
000C     40            11     SET     INC             AX
000D     FEC3          12             INC             BX
000F     42            13             INC             DX
0010     E2FA          14             LOOP            SET
0012     FEC7          15     POOL                INC                  BH
0014     80FF05        16             CMP             BH,05H
0017     75F9          17             JNZ             POOL
0019     F4            18     STOP    HLT
001A     EBE4          19             JMP             INIT

Errors =  0

LINE #          SYMBOL                  TYPE            REFERENCES

7               INIT                    P                       4,19
15              POOL                    P                       4,17
11              SET                     P                       4,14
18              STOP                    P                       4
```

Figure 3-2. Assembler Output Listing

Note

In the cross-reference table, the letter listed under the TYPE column has the following definition:

A = Absolute
C = Common (COMN)
D = Data (DATA)
E = External
P = Program (PROG)
U = Undefined

```
File: KW86                    HEWLETT-PACKARD:                    8086 Assembler
LOCATION OBJECT CODE   LINE    SOURCE LINE
                       1  "    8086"     XREF
                       2
                       3
                       4        GLB INIT,SET,POOL,STOP
                       5
                       6
0000     B80000        7        INIT      MOV          AX,0000H
0003     B80000        8                  MOV          BX,0000H
                       9                  MVO          CX,0005H
ERROR - UO                                             ^
0009     BA0000        10                 MOV           DX,0000H
000C     40            11       SET       INC          AX
000D     FEC3          12                 INC          BX
000F     42            13                 INC          DX
0010     E2FA          14                 LOOP         SET
0012     FEC7          15       POOL        INC           BH
0014     80FF05        16                 CMP          BH,05H
0017     75F9          17                 JNZ                        POL
ERROR - US, see Line 9                                 ^
0019     F4            18       STOP      HLT
001A     EBE4          19                 JMP          INIT
Errors =  2, previous error at line 17
US - Undefined Symbol.      The indicated symbol is not defined as a Label or declared
                                 as an external.
UO - Unidentified Opcode.   Opcode encountered is not defined for this microprocessor.

FILE: KW86                  CROSS REFERENCE TABLE
        LINE #              SYMBOL          TYPE                REFERENCES

        7                   INIT            P                   4,19
        ***                 POL             U                   17
        15                  POOL            P                   4
        11                  SET             P                   4,14
        18                  STOP            P                   4
```

Figure 3-3. Assembler Output Listing With Errors

Note 

Error messages are inserted immediately following the statement where the error occurs. All error messages (after the first error message) will contain a pointer to the statement where the last error occurred. At the end of the source program listing, an error summary statement will be printed. The summary will contain a statement as to the total number of errors noted, along with a line reference to the previous error. It will also define all error codes listed in the source program listing. Refer to Appendix C for a listing of all error codes.

Notes

4

Linking Your Programs

Introduction

A system application program, referred to as the linker, combines relocatable object modules into one absolute file. This absolute file can be loaded and executed in an emulation system or used for programming PROMs. Interaction between the user and the linker remains basically the same for any microprocessor assembler supported.

The linker prepares object code modules for emulation on the HP 64000, by performing two functions:

- Relocation: allocates memory space for each program relocatable module and relocates operand addresses to correspond to relocatable code.
- Linking: symbolically links relocatable modules.

Linker Functional Components

The linker has four major functional components: initialization, pass1, pass2, and cross-reference generation.

Initialization

The initialization function requires the following information from user keyboard input or as a command file:

- File names of all object files to be loaded.
- File names of libraries to be searched.
- Relocation information.
- Listing options.
- File name for the command/absolute/linker symbol file.

Pass 1

Pass 1 relocates all object file global symbols. If unresolved differences remain after processing all of the object files, then libraries are searched during Pass 1.

Pass 2

Pass 2 generates the absolute linker symbol and load map files. If memory overlaps are found, they will be flagged during Pass 2.

Cross-reference

Cross reference generation builds a table listing all global symbols, relocatable object modules that define global symbols, and relocatable modules that reference the symbols.

Linker
Input/Output Files

Linker Input Files

The linker processes two types of files: (1) relocatable files created by assembling source programs); and (2) linker symbol files (created previously by the linker). Filenames and extensions for the various hosts are shown below.

Relocatable Files

filename.R (HP-UX, MS-DOS, VAX/VMS)
Filename:reloc (HP 64000)

Linker Synbol Files

filename.L (HP-UX, MS-DOS, VAX/VMS)
Filename:link_sym (HP 64000)

Linker Output Files

Linking relocatable files produces four output files: (1) an absolute file ; (2) a linker symbol file ; (3) a linker command file ; and (4) an optional load map listfile . Filenames and file extensions for the various hosts follow.

Absolute File (object code) *ame:absolute*

filename.X (HP-UX, MS-DOS, VAX/VMS)
Filename:absolute (HP 64000)

Linker Symbol File

filename.L (HP-UX, MS-DOS, VAX/VMS)
Filename:link_sym (HP 64000)

Linker Command File *ame:link_com*

filename.K (HP-UX, MS-DOS, VAX/VMS)
Filename:link_com (HP 64000)

Listfile (optional) *ame:listing*

filename.O (HP-UX, MS-DOS, VAX/VMS)
Filename:listing (HP 64000)

Specifying Relocatable Files to be Linked

Files to be linked (and their respective load addresses) are specified by: (1) answering the linker questions; or (2) using a linker command file.

Answering the linker questions builds a linker command file. This linker command file may then be used to link the files that were specified in previous answers to the linker questions without having to answer the questions again.

Linker command files may also be edited. Edited linker command files may be used to specify: (1) new relocatable files to be linked; (2) different load addresses for the same relocatable files; or (3) both new relocatable files and different load addresses.

Answering Linker Questions

The commands to access the linker question is as follows:

lnk < RETURN> (HP-UX, MS-DOS, VAX/VMS)

link< RETURN> (HP 64000)

The questions that will be asked and the expected responses to build a linker command file are

Object files? {User types names of relocatable files to be linked}

Library files? {User types names of library files required for linking}

Load addresses: PROG,DATA,COMN {User specifies proper addresses}

More files? (y or n) {Enter either "y" or "n". If "y", then link editor reprompts again from 'object files' question. If "n", then link editor continues to the next question.}

Absolute file name= {User enters absolute file identifier}

Note 

Always terminate the last entry on a line with a comma if, during any question by the linker, entries are of such length that two or more lines are needed. Comma termination indicates to the linker that more entires follow. If any question (except the 'library files' question) is answered improperly or not answered at all (if no default values are shown), the link editor will request the proper information to be entered before it proceeds to the next question.

Explanation of Link Editor Questions

Object files?

You are asked for the name of each of the files that are to be linked. Object files that are listed after the "object file" question may contain relocatable modules, no-load files, and/or linker symbol files (for global symbol references).

No-load files No-load files are differentiated from normal relocatable files by enclosing the no-load files in parentheses: (filename). Parentheses indicate to the linker that no code is to be generated for the file. Relocation and linking occurs in the same manner as if the file was a load file. Note that the absolute image file generated by the linker does not contain the object code for the no-load file. No-load files are useful in linking to existing ROM code or in the design of software systems requiring memory overlays.

Linker Symbol Files Linker symbol files are included in the object file list when relocatable files contain references to global symbol locations in program modules already linked. An example of "object file?" response is shown below.

Object files? **file1.R,(file2.R,file3.R),file4.L** (HP-UX, MS-DOS, VAX/VMS)
File1,(File2,File3),File4:link_sym (HP 64000)

Library Files?

The library files question is the same as for object files. After all object files have been linked, the linker determines if any external symbols remain undefined. The linker searches the library files for object modules that define these symbols. The linker relocates and links only those relocatable modules that satisfy external references.

If a library file name is given as a response to the "object files?" question, then all the relocatable modules in the library file will

be relocated and linked. If a library file name is given as a response to the "library files?" question, then only those relocatable modules that define the unsatisfied externals will be relocated and linked. The remaining relocatable modules in the library file will be ignored.

It is also possible to combine relocatable files into a library by using the HP 64000 library command.

An example answer to the "library files?" question is:

Library files? **/usr/hp64000/lib/ns8086** (HP-UX)
\user\hp64000\lib\ns8086 (MS-DOS)
LIB:NS8086 (HP 64000)
HP$DISK:[HP64000.NS8086] (VAX/VMS)

Load Addresses:PROG,DATA,COMN

This question requires you to select separate, relocatable memory areas for the different modules of the program. Logical addresses (i.e., segment:offset) are entered unless your source files contain the "80286" directive. If the "80286" directive is present, then 24 bit physical addresses are entered. For example, if the following entries were made:

Load addresses:PROG,DATA,COMN=
00001000H,00002000H,00003000H

The linker would relocate the PROG file module in memory location starting at address 1000H. The DATA module relocates to memory location starting at address 2000H. The COMN module relocates to memory location starting at address 3000H.

Note 

Load addresses may be entered using any number base (binary, octal, decimal, or hexadecimal). However, the addresses listed in the load map are given in hexadecimal only.

More files? You now determine if more files are to be linked. If yes ("y"), then the linker begins interrogation again, allowing additional object and library files to be specified with new load addresses. You may continue with the previously relocatable area by typing "CONT" in the appropriate field when specifying new relocatable areas. The relocatable area is treated as if no new address was assigned. An example of the use of the "CONT" notation is as follows:

Load addresses:PROG,DATA,COMN=
0FF000BCCH,**CONT**,00003FFCH

Absolute File Name? You now assign a name to the command/absolute image file about to linked. The absolute file created by the linker is always associated with a link command file and a global symbol file of the same name.

Using Linker Command Files

The linker produces up to three files: (1) an absolute file ; (2) a linker symbol file ; and, if none exists, (3) a command file . Once linker command files have been created, they may be used to re-link the same relocatable files without answering the linker questions a second time. Linker command files are highly useful when modifications are made to assembly language programs and when these programs must then be reassembled and re-linked.

Linker command files may also be edited. Edited linker command files can link different relocatable files, or specify different load addresses, or both.

On the HP 64000, linker command files are edited by entering the following commands:

link < CMDFILE> **options edit**

You may now step through and change your previous answers to the linker questions by modifying the entries.

On the HP-UX, MS-DOS, and VAX/VMS systems, you must edit the linker command file like any other text file because the linker command file is an ASCII text file.

The command file format for HP-UX, MS-DOS, and VAX/VMS is shown in the following figure.

```
segment                     (begin a new segment)
object files                < FILE1> [,< FILE2> ,... ,< FILEn> ]
library files               [< LIB1> ,< LIB2> ,...< LIBn> ]
load addresses              < PROG> , < DATA> , < COMN>
[segment
        .
        .                   (Include as many segments as needed)
        .
        .]
absolute file name < ABSFILE>
```

Figure 4-1. Example Linker Command File

Running the Linker

The following pages describe link syntax for the different hosts and explain the procedure to link relocatable modules.

lnk (HP-UX)

Syntax lnk [-n] [-x] [-o] [-c] < file>

Syntax Definitions

Definitions for syntactical terms are as follows:

< file>	A variable representing the linker command file name. The syntax for < file> :

< file> = > < filename.K>

The file type must be a linker command file that ends in the .K file extension; no other file type can be specified with the lnk command.

output default	Listing files are not created unless the **-o option** is invoked, in which case the list-file is written to **stdout**. To direct the output into a file, use HP-UX syntax > **file-name** with the -o option.

Option Definitions

lnk recognizes the following options, the first of which must be preceded by a dash (-); however, options may be concatenated (e.g. -nxoc):

-n	Do not produce a load map listing.
-x	Produce a symbol cross-reference listing.
-o	Cause the listing to be created.
-c	Do not check for memory overlap.

lnk (HP-UX) Cont'd

Example Here are two examples of the lnk command:

lnk -xo reg8.K > reg9
or
lnk -xo reg8.K | lpr

In the first example above, the output listing with cross-reference table will be put in a file "reg9".

Note 

To save the error output with the output listing, redirect stderr. Example:
lnk -xo reg8.K > reg9 2> &1

In the second example above, the output listing with cross-reference table will be output to the line printer.

Note 

The linker is contained in public directory **/usr/hp64000/bin**. If /usr/hp64000/bin is in the user's directory path, a link can be run by using only the command "**lnk**". The personality tables are in **/usr/hp64000/tables**.

---

lnk (MS-DOS)

Syntax

lnk /n /x /o /c < file>

Syntax Definitions

Syntax definitions of terms include:

< file>	A variable representing the linker command file name. The syntax for < file> is:

< file> = > < filename.K>

The file type must be a linker command file that ends in the .K file extension. No other file type extension can be specified with the **lnk** command.

output default	Listing files are not created unless the **/o** option is invoked. If invoked, the listfile is written to **stdout.** To direct the output into a file, use the command parser redirection **> filename** with the **/o** option.

Option Definitions

lnk recognizes the following options only when preceded by a slash (/). Options may also be concatenated (e.g., /nxoc):

/n	Do not produce a load map listing.
/x	Produce a symbol cross-reference listing.
/o	Cause the listing to be created.
/c	Do not check for memory overlap.

lnk (MS-DOS) Cont'd

Examples

Two examples of the lnk command are:

lnk /xo reg8.K:reg9
or
lnk /xo reg8.K

The output listing with cross-reference table in the first example will be put in a file "reg9".

Note 

To save the error output with the output listing, redirect stderr. Example:
lnk /xo re3g8.K > reg9 2:&1

The output listing with cross-reference table in the second example will be output to the screen.

Note 

The linker is contained in public directory **\HP64700\BIN.** If **\HP64700\BIN** is in the user's directory path, a link can be run by using only the command "**lnk**". The personality tables are in **\HP64700\TABLES**.

---

link (HP 64000)

Syntax

link [< FILE>] [listfile < list destination>]

options [edit][nolist][xref][no_overlap_check][comp_db]

Syntax Definitions

< FILE>
: A file of type link_com to be used to direct the linker as to relocatable and relocation addresses.

< list destination>
: File or device to which listing output is sent.

Options Definitions

Allows you to override options specified in the linker command file.

nolist
: Overrides the list option specified in the linker command file and suppresses output of a load map.

xref
: Overrides no xref option specified in the linker command file and forces output of a global symbol cross-reference table.

edit
: Allows you to edit the current link_com file.

no_overlap_check
: Overrides overlap_check option specified in the linker command file and suppresses errors caused by

link (HP 64000) Cont'd

	memory overlaps. Default condition for overlap_check is ON.
comp_db	This file is created by the linker when requested and is a data base containing information from all of the comp_sym files associated with relocatables in an absolute file.

Note 

If previous link commands have specified the comp_db option, and new link commands do not specify the comb_db option, then old comp_db files will not be purged.

Default Values

< FILE>	If no linker command file is specified, the default allows creation of a new file of type link_com.
< list destination>	Defaults to user specified listfile default.
options	If options is not entered, listing defaults to options specified in the linker command file. If options is specified, but no option is selected, a load map listing with no cross-reference is made.

link (HP 64000) Cont'd

Examples

link

Requests the linker to create a new linker command file. Listing output will go to the listfile default.

link KW86

Links the absolute file KW86 containing files in linker command file KW86. The listing output will go to the listfile default and any options in the KW86:link_com file are in effect.

link KW86 **options edit**

This requests the linker command file KW86 options edit for the purpose of viewing or editing. Any listing output will go to the listfile default.

lnk (VAX/VMS)

Syntax

lnk < file>

Options	Default Values
/[no]map	/map
/[no]xref	/noxref
/[no]mem_ovlp	/mem_ovlp
/[no]output= [< file>]	/nooutput

Syntax Definitions

Definitions for syntactical terms are as follows:

< file> — A variable representing the linker command file name. The syntax for < file> is:

< file> = > < filename.K>

The extension does not have to be specified; it automatically defaults to .K.

Options Definitions

lnk recognizes the following options which must be preceded by a slash (/). All of the options can be negated by placing a "no" in front of the option; for example, /nomap.

/map — Produces a load map listing.

/xref — Produces a symbol cross-reference listing.

/mem_ovlp — Checks for memory overlap.

lnk (VAX/VMS) Cont'd

/output[= < file>]

If you specify /output, then a listing file will be generated. If < file> is omitted, then the absolute file name will be used for the listing file. The default extension for listing files is .LIS.

Examples

Here are two examples of the lnk command:

lnk reg8
or
lnk/xref/output= reg reg8

In the first example above, reg8.K will be used as a linker command file with no output. In the second example, reg8.K will be used as the linker command file, and a load map listing and symbol cross-reference listing will be put in file reg.LIS.

Linker Output

Linker listings may be output to the terminal CRT, line printer, or any file. The following information may be included in the linker output listing:

- Listing (Load Map).
- Cross-reference table.
- Error messages.

Note 

Certain error messages containing more than 80 characters will be viewed as wrapped around or truncated on many terminals. Complete error messages will be printed when using the line printer or a list file for listings.

Listing (Load Map)

A load map is a listing of the memory areas allocated to each relocatable file. The listing begins with the first file linked and proceeds to list all other linked files with their allocated memory locations. An example of a load map listing that will be printed on the system printer is as follows:

```
FILE/PROG NAME   PROGRAM          DATA COMMON ABSOLUTE  DATE
reg4                                                   00010000-00010037 Mon, 26 Mar 1984
          # REG4
reg7                          00000000                                          Mon, 26 Mar 1984
next address                   0000003A

XFER address =       0000000  Defined by DEFAULT
Current working directory =  /users/bobg
Absolute file name =  reg8:absolute
Total number of bytes loaded = 72

FILE/PROG NAME          PROGRAM          DATA COMMON     ABSOLUTE        DATE TIME COMMENTS
KYBD:SAVE               0000                                             Thu,5, Jun 1982 11:37
EXCT:SAVE                                                0B00-0B34       Thu, 5 Jun 1982 10:38
DSPL:SAVE                        A100                                     Thu, 5 Jun 1982 11:38
next address            0021      A121

REG1:SAVE               B000                                             Thu, 5 Jun 1982 11:52
REG2:SAVE               B103                                             Thu, 5 Jun 1982 11:53
REG3:SAVE               B206                                             Thu, 5 Jun 1982 11:58
next address            B30C

Libraries
PARAMETER:SAVE 0021                                                      Thu, 5, Jun 1982 11:43
MULTEQUAT:SAVE          0221                                             Thu, 5 Jun 1982 11:45
next address            0421      A121

XREF address= 0B00 Defined by EXCT
No. of passes through Libraries= 1
absolute & Link_com file name= SETAG1:SAVE
Total#  of bytes Loaded= 0782
```

Figure 4-2. Example Load Map Listing

A brief description of each column in the listing follows:

File/Prog Name This column contains the name of the files that are linked (reg4 and reg7). If the source name differs from the relocatable name, the source name is indented and printed below the relocatable file name (# REG4).

If library files are referenced, the master library will be listed. Subsections of the master library referenced will also be listed beneath the library file name. Subsections will be indented to indicate that they are part of the main library file. No-load files will be displayed in parentheses (...).

Program This column indicates the first address (hexadecimal) of a memory block that contains the PROG relocatable code in the file listed in the FILE/PROG NAME column.

Data This column indicates the first address (hexadecimal) of a memory block that contains the DATA relocatable code in the file listed in the FILE/PROG NAME column.

Common This column indicates the first address (hexadecimal) of a memory block containing the COMN relocatable code in the file listed in the FILE/PROG NAME column.

Absolute This column indicates the hexadecimal addresses of a memory block containing the absolute code assigned by the file listed in the FILE/PROG NAME column.

Note 

The "next address" statement in the load map listing indicates the next available hexadecimal address in PROG, DATA or COMN memory areas. This statement may also be used to determine the number of bytes (words for 16-bit processors) that are contained in each area (next address less starting address= total bytes).

Date This column indicates the date that the file in the FILE/PROG NAME column was assembled.

Time This column indicates the time that the file listed in the FILE/PROG NAME column was assembled.

Comments User comments may be entered in this column during assembly by the assembler pseudo NAME instruction.

XFER address The starting address in memory for program execution is XFER. XFER address can be assigned using the END pseudo in one of the relocatable files.

Current working directory Indicates the current MS-DOS directory being used.

Absolute file name Indicates the absolute file name assigned to the linked files.

Total bytes loaded Total number of bytes loaded during this link is indicated here.

Cross-Reference Table

The cross-reference table lists all global symbols, relocatable object modules that define them, and relocatable modules that reference them. An example of a cross-reference listing that will be listed on the system printer is as follows:

SYMBOL	R	VALUE	DEF BY	REFERENCES
DATA16	A	00007ABC	reg4	reg7
DATA32	A	000F423F	reg4	reg7
DATA8	A	0000007E	reg4	reg7

Figure 4-3. Sample Cross Reference Table

Each column in the cross-reference listing represents:

Symbol All global symbols will be listed in this column.

R(Relocation) A letter identifies the type of program module in this column. Available letters and their definitions are:

A=	Absolute
C=	Common (COMN)
D=	Data (DATA)
P=	Program (PROG)
U=	Undefined

Value Value of the relocated address of the symbol is in this column.

Def by A file name that defines the global symbol is in this column.

References This column lists the file names that reference the global symbol.

This concludes discussion of the Linker.

5

Source File Format And Expressions

Introduction

The HP Model 64000 Assembler recognizes three types of source statements: microprocessor instructions, assembler pseudo opcodes, and macro definitions or calls. This chapter describes the coding rules and conventions that must be followed when using the assembler.

Source Statement Format Rules

Each microprocessor instruction, assembler pseudo opcode, or macro call is divided into four fields: the label field, the operation field, the operand field, and the comment field. Format rules to be followed when constructing a line of source program follow:

Field Sequence

Field sequence cannot be changed. The correct order of field sequence is:

Label	Operation	Operand	Comment
SAVE	EQU	EXEC1	;SAVE ;EQUATES ;TO EXEC1

Note 

You are recommended to have each field in the source statement start at a fixed position (column) in the source line. This format may be defined using the tab setting capabilities of the system editor to specify each field's starting position. The presentation of the program listing in a fixed format improves readability.

Delimited Fields

One or more spaces (blanks) must separate the fields in a source statement.

Note 

Because of the way the assembler parser works, white space may be treated as the end of a statement when in fact the end of the statement has not been reached. If the assembler sees what it considers to be a complete, syntactically aceptable statement before encountering white space, it may stop at that white space without reading the remainder of the statement. The resulting generated code will be different than the code for the actual statement. This difference may not be discovered until execution time. Some examples of this problem follow:

```
"8086"
        SIGN ORG 10H
        MOV CX, BX OFFSET SIGN
```

Although OFFSET SIGN is illegal, the assembler does not recognize it as illegal because the parser stopped at the valid instruction "MOV CX, BX." Using an EQU to replace BX OFFSET SIGN will not solve the problem because EQU's are expanded before they are parsed.

```
"8086"
        DATA
VALUE   DB      10      DUP (?)
```

DUP is not supported by this assembler. However, the code in this example will not cause an error because the parser saw "VALUE DB 10 as a valid instruction.

Label Field Position

A label field, if used, must begin in column 1 of the source statement. If column 1 is blank, the assembler assumes that the label field is omitted.

Additional rules and conventions governing source statement length and fields are given in the following paragraphs.

Statement Length

A source statement may contain up to 110 characters (including spaces). A statement is terminated by a carriage return < RETURN> . Any statement containing more than 110 characters will be truncated to 110 characters.

Blank lines will not affect the object modules and may be introduced to improve readability of the source program listing.

Label Field

Labels may be used in all microprocessor instructions, some assembler pseudo opcodes, and macro calls. Since the label assigned identifies that particular statement and may be used as a reference point by other statements in the program, **every label must be unique within each source program.**

Note

Some specific symbols are predefined and cannot be used as labels. Predefined symbols will depend upon the microprocessor being supported.

The label field starts in column 1 of the source statement and must be terminated by a space or a colon (:).

Note

A colon (:) cannot be used to terminate a macro label. Refer to chapter 8 for construction of Macros.

A valid label may contain any number of characters. **The first character in the label must be an upper case alphabetic character.** Remaining label characters may be either alphabetic or numeric. The alphanumeric character set includes the letters of the alphabet (upper and lower case), the underline symbol (_), and the numeric digits 0 through 9. Invalid symbols

shown below include the dollar sign ($), the question mark (?) and beginning a label with a number (4).

Valid Symbols	Invalid Symbols
Ab_cd	ab.cd?
AB_CD	$BCDEF
A5rHi	4UVWXY

If more than fifteen characters are entered in the label field, the assembler will print all characters in the output listing but will use only the first 15 characters for label identification. Therefore, the assembler will recognize:

STATEMENTLABELA1

and

STATEMENTLABELA2

as being identical 15 character long labels. A duplicate-symbol error message will then be issued.

Statements requiring labels are macro definitions and EQU pseudo instructions. Assignment of a label is optional for all other statements.

Operation Field

The operation field contains: a mnemonic code for a microprocessor instruction; an assembler pseudo opcode (refer to chapter 7); or a macro call (refer to chapter 8). The assembler pseudo opcode specifies the operation or function to be performed. The operation field follows the label field and is separated from it by a least one space, a tab, or a colon(:). If there is no label, the pseudo opcode may begin in any column position following column 1.

Operation field termination is done by one or more spaces, or by a tab. If no operand field follows, the operation field can also be terminated either by a carriage return, or by a semicolon(;) indicating the start of the comment field.

Assembler pseudo and control statements provide the following capabilities:

- Assembler control.
- Object program linkage.
- Address and Symbol definitions.
- Constant definition.
- Assembly listing control.
- Storage allocation.

A label will be assigned to the current program counter location if the label is specified and the operation field does not contain a microprocessor instruction, an assembler pseudo opcode, or a macro call.

Operand Field

Values or locations required by the microprocessor instruction, assembler pseudo opcode, or macro call are specified by the operand field. The microprocessor uses various modes of addressing for obtaining the operands and saving the results of program execution.

The mnemonic instruction and the information in the operand field determines the addressing mode. Each instruction determines proper operand type and sequence. The operand field, if present, follows the operation field and must be separated from it by at least one space ( ).

An operand may contain an expression consisting of a single symbolic term, a single numeric term, or a combination of symbolic and numeric terms, enclosed in parentheses, and joined by the expression operators + , -, *, and /.

Comment Field

An optional comment field may contain any information that the user deems necessary to identify portions of the program. The delimiter for the comment field is the semicolon (;), a tab, or a space following the operand field. A semicolon in any column of the source statement will invoke the comment field (except when used in an ASCII string). In situations where more than one line of programming is needed for the comment field, an asterisk (*) in column 1 of a source statement indicates the following information is part of a comment field and should not be acted on as if it were part of the program.

Delimiters

Certain delimiting characters are restricted and are used to indicate the end of fields or labels, and the beginning of other fields or labels. Delimiters should not be used as ordinary characters. For example, **a space cannot be used as part of a label name.** A description of delimiters in Table 5-1 follows:

Table 5-1. Delimiters

Delimiter	Use
Space	Separates fields or operands; ends a label.
Tab	Separates fields; ends a label.
Semicolon (**;**)	Indicates start of comment field.
Asterisk (*****)	When used in column one of source statement indicates that comment field will follow.
Carets (**^ ...^**)	Indicates a character string.
Colon (**:**)	Indicates end of label field.
Parentheses(**(...)**)	Used in expression for precedence.
Single Quotes (**'...'**)	Indicates a character string.
Ampersand (&)	Indicates macro parameters.
Double Ampersand(&&)	Indexes macro parameters.
Quad Ampersand (&&&&)	Identifies a Macro unique number.
Quotation Marks (**"..."**)	Indicates a character string.

Symbolic Terms

A symbol used in the operand field must be a one that has been defined in the program, such as a symbol in the label field, a machine instruction, or a symbol in the label field of an EQU pseudo instruction. (Note that the EQU label field must be defined prior to referencing).

A symbol may be either absolute or relocatable. Either type depends on the type of assembly selected. The assembler assigns a value to a symbol when encountered in a label field of a source statement. If the program is to be loaded in absolute form, the values assigned by the assembler remain fixed. If the program is to be relocated, the actual value of a symbol will be established by the linker (refer to chapter 4 for linker processing).

A symbolic term may be preceded by a plus (+) or minus (-) sign. If preceded by a plus (+) or no sign, the symbol refers to its associated value. If preceded by a minus (-) sign, the symbol refers to the 2's complement of its associated binary value.

Program Counter ($)

($) is a symbolic term used to indicate the current value of the program counter. ($) can be used any place that symbolic references are legal (for example: $+ 2).

Numeric Terms

Numeric terms may be binary, octal, decimal, or hexadecimal. A binary term must have the suffix "B" (for example: 101101B). Octal values must have either an "O" or a "Q" suffix (for example: 26O, or 26Q). A hexadecimal term must have both the suffix "H" and a number prefix (using 0,2, or 3 for example: 0BBH, 2CDH, 36H). **When no suffix is assigned decimal value is assumed**.

Note 

It is necessary to **start a hexadecimal term with a decimal digit** since the assembler identifies a term starting with an alphabetic character as a symbolic reference. **All alphabetic hexadecimal digits must be capital letters** for compatability with the HP 64000 system
assembler.

String Constants

In addition to numeric and symbolic constants, an operation may also contain string constants. String constants are produced by using ASCII (American Standard Code for Information Interchange) characters (See appendix H for ASCII values.). String constants, combined with other symbols and constants, are written by enclosing ASCII characters within quotation marks ("..."), single quote marks ('...') or carets (^ ...^).

The numeric value of a string is defined as follows:

Null String

A null string (" ") (' ') or (^ ^) has a numerical value of zero.

One Character String

A one character string is stored in the high order byte of the low word (if more than one word is used). The value that appears there is the hexadecimal value of the ASCII representation of the character. The low order byte has the value 00H. Example:

'C' = "C" = 00000000B = 00H = Low order byte
01000011B = 43H = High order byte

Two Character String

A two character string is stored in the low word (if more than one word is used). The hex ASCII value of thefirst character of the string is stored in the high order byte of the word and the hex ASCII value of the second character of the string is stored in the low order byte. If any words remain, their bytes contain 00H. Example:

'AB' = "BA" = B = 01000010B = 42H = Low order byte
A = 01000001B = 41H = High order byte

Note 

The MASK pseudo instruction allows the user to alter ASCII strings. Refer to the MASK pseudo description in Chapter 7.

Strings Longer Than Two Characters

There are restrictions as to where strings longer than two characters may be used. For strings longer than two characters, the hex ASCII value of each character in the string is stored in byte order. That means the ASCII value of the first character is stored at the lowest byte address for the string and the last character is stored at the highest byte address for the string.
Example:

'BCDE'= "BCDE"= B = 01000010B = 42H = Low byte
C = 01000011B = 43H = 2nd byte
D = 01000100B = 44H = 3rd byte
E = 01000101B = 45H = High byte

Expression Operators

The assembler contains two groups of expression operators that permit the following operations:

Arithmetic Operators

The arithmetic operators are:

Operator	Interpretation
+	Addition
-	Subtraction
*	Multiplication
/	Division

Examples

The following expressions generate the bit pattern for ASCII character W (0101011B):

```
1+ 28*2
1+ (-28*-2)
1+ (84/3)*2
```

Logical Operators

Logical operators are used to form logical expressions. A logical expression may be used any place an expression can legally be used. The logical operators are as follows:

Operator	Interpretation
.AN.	Logical AND
.NT.	Logical one's complement
.OR.	Logical OR
.SL.	Shift left
.SR.	Shift right

Examples

```
EXEC1.SL.1
NT.CHAR
EXEC1.OR.EXEC2
```

Operator Precedence

Operators have a descending order of precedence defining which operator is evaluated first or next in an expression. Operators are listed below in descending order of precedence.

Parentheses (...) override all precedence.

```
.NT.
.SL.,.SR.
.OR.,.AN.
*,/
+ ,-
```

Relational Comparison (Macros Only)

When the assembler processes an ".IF " instruction, the logical expression in the operand field is evaluated. Relational operators are:

Operator	Interpretation
.EQ.	equal
.NE.	not equal
.LT.	less than
.GT.	greater than
.LE.	less than or equal
.GE.	greater than or equal

Relocatable Expressions

Three program counters are provided for identifying relocatable code areas. The three areas are identified as data (DATA), program (PROG), and common (COMN). These areas can be changed from one relocatable area to another using assembler pseudo codes. (Refer to chapter 7 for more information.) Rules governing use of relocatable expressions are given in the following paragraphs.

The value of a relocatable term will be assigned during the linking process. The assigned value will depend upon:

- The relocatable areas (PROG, DATA, or COMN) to which it is assigned, and;
- Where the area is located in memory during the link operation.

Expressions may be formed from absolute and relocatable terms using arithmetic operators and parentheses. Expressions resulting from this type of operation must be either absolute or one of the three relocatable types.

Absolute Terms

Absolute terms are expressions having values not dependent upon the location of the program module in memory. Formation of absolute expressions requires that:

- Each absolute term or constant is an absolute expression.
- If AD and BD are relocatable symbols in the same relocatable area, then (AD-BD) is designated an absolute expression. (This designation is absolute because the difference between AD and BD remains constant regardless of the relocation factor of the program. That is, if the program is relocated, the values of AD and BD are offset by the same amount.)
- If A2 and B2 are absolute symbols, then:

  (A2+ B2)
  (A2* B2)
  (A2-B2)
  and (A2/B2)

  are absolute expressions.

Relocatable Terms

Relocatable terms are expressions having values undefined at link time. Formation of relocatable expressions requires that:

- Any relocatable term is a relocatable expression.
- If DA is an absolute expression and DR is a relocatable expression, then:

  (DA+ DR)
  (DR+ DA)
  and (DR-DA)

  are relocatable expressions and are the only relationship permitted. An absolute expression may be subtracted from a relocatable expression. A relocatable expression may not be subtracted from an absolute expression.

Invalid Relocatable Terms

Use of relocatable terms in certain ways makes them invalid and will generate error messages. A valid example (c) is provided below along with two invalid relocatable terms (a & b) that generate error messages:

a. Two relocatable symbols - same area (PROG, DATA, or COMN). If DA and DB are two relocatable symbols, then:

```
(DA+ DB)
(DA*DB)
and (DA/DB)
```

are **invalid expressions**. The assembler does not recognize where these symbols are being stored in memory.

b. Two relocatable symbols - different areas (PROG, DATA, or COMN). If DA and DB are two relocatable symbols, then:

```
(DA+ DB)
(DA-DB)
and (DA*DB)
```

are **invalid expressions**. The assembler does not recognize where these symbols are being stored in memory.

c. Relocatable symbols in different areas (PROG, DATA, COMN) can be combined if the expression results in one relocatable type. For example, if relocatable symbols DA and DB are PROG type and relocatable symbol DC is DATA type, the expression:

```
(DA+ DC-DB)
```

is a **valid expression** since (DA-DB) is an absolute offset to DC.

This concludes the discussion of source file format and expressions.

Notes

6

Programming Considerations

Introduction

This chapter will help you to write assembly language programs by describing the HP 64000 8086/8088 assembler. If you know how the assembler works, what is expected in an assembly language program, and how the assembler generates code, then programming will be easier. If you follow the guidelines in this chapter, your code will generally have fewer errors at first writing. Errors will be easier to identify and can be corrected if and when they do occur.

This chapter contains special 8086/8088 pseudo instructions and keyword operators. It is important to understand pseudo instructions and keyword operators because together they tell the assembler how to generate code.

Key Concepts to Understanding the 8086/8088 Assembler

The two most important concepts to know when using the 8086/8088 Assembler are:

- The concept of a **segmented architecture.** This requires using segment and offset values in assembly language memory location references.
- The concept that **one assembly language mnemonic can specify several types of operations**. Types of operations relate to the size of data that is assembled or linked, or to the distance in memory of program transfers.

Impact of Segmented Architecture on Programming

8086/8088 Segmented Architecture

8086/8088 microprocessors are designed with a segmented architecture. Given a segmented architecture with 20 address lines, these processors can address 220 bytes (1 megabyte) of physical memory. Memory addresses inside the 8086/8088 microprocessors are calculated with two 16-bit quantities: a segment and an offset. Figure 6-1 shows how physical memory addresses are calculated with segments and offsets.

Physical Addresses vs. Logical Addresses (Segment:Offset)

Logical Addresses

A logical address is a 32-bit (segment:offset) quantity whose upper and lower 16-bit offset values are used to calculate a 20-bit physical address. The assembler recognizes only logical addresses.

Absolute Addresses

Logical addresses must be used when absolute addresses are specified in your assembly language programs, (for example, in the ORG pseudo instruction operand). 32-bits of information must be supplied in logical addresses. The first 16-bits specify the segment value of the address. The next 16-bits specify the offset value.

Different Logical Addresses Can Specify the Same Physical Address

Physical Addresses

It is possible for two different pairs of "segment:offset" values to specify the same physical address. Be aware that **two different pairs of segment and offset values can specify the same physical address.** For example, the instructions ORG 10002345H and ORG 12000345H specify the same physical address:

Contents of reg. DS =	1000H	Contents of reg. ES =	1200H
Contents of reg. BP =	+ 2345H	Contents of reg. BX =	+ 0345H
Address DS:[BP] =	12345H	Address ES:[BX] =	12345H

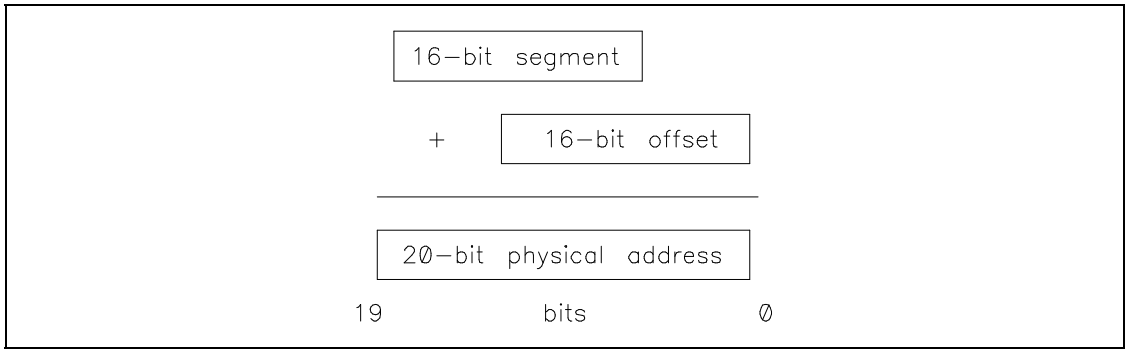

Figure 6-1. Calculating Physical w/Logical Addresses

Specifying Segments for Memory Referencing Operands

Every assembly language program memory reference must refer to one of the processors four segment registers. Contents of the segment register will be the 16-bit segment value of the logical address.

The offset value of the logical address will be specified by a program label, a base register, an index register, or a combination of two or three of the above, depending on the addressing mode. There are two ways in which the segment register can be specified in assembly language instructions:

- **Segment registers can be specified explicitly, by including the segment register name in the instructions memory referencing operand.**
- **Segment registers can be specified implicitly, by using the special 8086/8088 ASSUME pseudo instruction.**

Specifying Segment Registers Explicitly

Specifying which segment register is to be used in calculating an address requires including it in the assembly language instruction operand. The following instructions demonstrate this technique. (Note in the example below that DW is a special 8086/8088 pseudo instruction. It defines and initializes a word of memory.)

```
LOCATION OBJECT CODE          SOURCE  LINE

                              1 "8086"
0000    383C                  2 LABL    DW      3C38H
                              3
0002    2EA30000              4         MOV     CS:LABL,AX
0006    899F0000              5         MOV     DS:LABL[BX],BX
000A    26894E00              6         MOV     ES:[BP],CX
000E    2E89800000            7         MOV     CS:LABL[BX][SI],AX
0013    8903                  8         MOV     SS:[BP][DI],AX

Errors= 0
```

Specifying Segment Registers Implicitly

Another way to specify the segment portion of the logical address is to let the assembler "ASSUME" the segment register for you. ASSUME is a special 8086/8088 pseudo instruction which allows you to relate one of the processors segment registers to one of the HP 64000 code areas.

HP 64000 Code Areas

The HP 64000 defines three relocatable code areas: PROG, DATA, and COMN. One absolute code area is also defined (ORG). Locate parts of your assembly language program in each of these four code areas by using the appropriate HP 64000 pseudo instruction: PROG, DATA, COMN, or ORG. The default code area is PROG. The assembler maintains a program counter for each of these code areas. Assign the actual addresses of these relocatable code areas when linking your programs. The address of the ORG absolute code area program counter is specified in the ORG pseudo instructions operand.

HP 64000 code areas can be thought of as segments of physical memory.

Using the ASSUME Pseudo Instruction

The ASSUME pseudo instruction allows you to relate one of the microprocessors segment registers to one of the HP 64000 code areas. When memory references are made by your assembly language instructions, the assembler assumes which segment register should be used to calculate the physical address. When using the ASSUME pseudo, assembly language instructions may be written in the form shown below.

```
LOCATION OBJECT CODE   SOURCE LINE

                       1     "8086"
                       2           DATA
0000     383C          3     LABL  DW    3C38H
0002                   4     DEST  DWS   1
                       5
                       6           PROG
                       7           ASSUME DS:DATA,CS:PROG
0000     B80000        8           MOV    AX,SEG LABL     ;These two
0003     8ED8          9           MOV    DS, AX          ;instructions
                                                         ;initialize reg.DS
                       10
0005     A30000        11          MOV    LABL,AX  ;DS:LABL,AX
                                                         ;is assumed
0008     899F0000      12          MOV    LABL[BX],BX     ;DS:LABL[BX],BX
                                                         ;is assumed
000C     894E00        13          MOV     [BP],CX  ;SS:[BP],CX
                                                         ; is assumed
000F     89800000      14          MOV    LABL[BX][SI],AX
                                                         ;DS:LABL[BX][SI],AX
0013     8903          15          MOV     [BP][DI],AX     ;SS:[BP][DI],AX
                                                         ;is assumed
0015     A4            16          MOVS    DEST,LABL
ERROR-IO                                   ^

Errors=  1,          previous error at line          16
IO - Invalid Operand, Invalid or unexpected operand encountered or operand is missing
```

The assembler assumes segment registers based on two things: (1) The operand of the ASSUME pseudo instruction, and (2) the code area in which the program labels appear. In the example program above, LABL appears in the DATA code area. The assembler assumes that any memory references containing the label LABL should use register DS as the segment because it appears in the DATA code area.

When memory references do not contain labels, the assembler assumes that the SS register should be used as the segment value in calculating the physical address.

The example indicates an error on the last line. The error indicates that the assembler expected the destination operand of the MOVS instruction to be in the ES segment. The example did not assume anything about the ES register. Adding ",ES:DATA" to the ASSUME pseudo instruction operand will correct the error and allow the last instruction to assemble correctly.

Note 

In 8086 string instructions, the assembler always expects the destination operand to be associated with segment register ES.

Forward References

Since the assembler cannot know what segment a forward referenced variable will reside in prior to its definition, a segment override byte will be generated for all forward referenced variables during Pass One. This will be done if a segment override was not explicitly specified. An extra byte may result for these instructions, but will produce working code. You may wish to consider this when laying out your programs. Placing the data definition sections prior to the data referencing sections will produce fewer bytes of code.

Segment Overrides

When using the ASSUME pseudo it is possible to explicitly tell the assembler which of the processor's segment registers to use in calculating the physical address. **Adding segment overrides to memory referencing operands tells the assembler which segment register to use.** Segment overrides "CS:" and "ES:" cause the assembler to generate code identical to the code generated by assembling the first example program in the following example:

```
LOCATION OBJECT CODE    SOURCE  LINE
                        1 "8086"
                        2       DATA
0000    383C            3 LABL  DW              3C38H
                        4
                        5       PROG
                        6       ASSUME DS:DATA,CS:PROG
0000    B80000          7       MOV             AX,SEG LABL     ;These two
0003    8ED8            8       MOV             DS,AX           ;instructions initialize
                                                                ;reg.DS
                        9
0005    2EA30000        10      MOV             CS:LABL,AX
0009    899F0000        11      MOV             LABL[BX],BX     ;DS:LABL[BX],BX
                                                                ;is assumed
000D    26894E00        12      MOV             ES:[BP],CX
0011    2E89800000      13      MOV             CS:LABL[BX][SI],AX
0016    8903            14      MOVS            [BP][DI],AX     ;SS:[BP][DI],AX
                                                                ;is assumed
Errors= 0
```

Turning Off the "ASSUME" Pseudo

Specifying "NOTHING" in the pseudo instruction's operand field turns off the ASSUME pseudo instruction. NOTHING will cause the assembler to expect segment registers to be explicitly stated in memory referencing operands.

You may also "assume nothing" about a specific segment register by specifying NOTHING in the code area portion of the ASSUME pseudo instruction operand, e.g., ASSUME CS:NOTHING. If a memory referencing operand is to use a segment register for which NOTHING is assumed, then that segment register must be stated explicitly in the operand.

Types of Operations

Different types of operations may be specified with the same 8086/8088 assembly language mnemonic. Types of operations specify the size of the data that is operated on. Data size may be a byte, word, or a double-word sized piece of information. Types of operations refer also to the distance of program transfers to a memory location. Types of operations refer to the same 64K segment or to a memory location in another 64K segment.

Five "Types" Associated with Program Symbols

The 8086/8088 assembler associates a "type" with every program symbol (label) in order to aid the assembler in generating object code. Program symbols (labels) appear at two kinds of memory locations: data locations or instruction locations.

MEMORY LOCATION	TYPE
Data locations	- BYTE (8-bits wide) - WORD (16-bits wide) - DWORD (32-bits wide)
Instruction locations	- NEAR (within _+ 32K bytes) - FAR (beyond_+ 32K bytes)

How "Types" Are Associated with Memory Locations

The assembler associates "types" with memory locations and identifies the "types" associated with its program labels. Note in the example below that DB (Define Byte), DW (Define Word), and DD (Define Doubleword) are special 8086/8088 pseudo instructions which define and initialize memory.

```
LOCATION OBJECT CODE    SOURCE  LINE

                        1       "8086"
                        2               DATA
0000    C415354         3       NAME    DB              "LAST"
0004    383CFFFF80      4       LABL    DW              3C38H,0FFFFH,80H
0009    00
000A    002000F28F      5       VALU    DD              0F2002000H,120.0E-3
000F    C2F53D
                        6
                        7               PROG
                        8               ASSUME CS:PROG,DS:DATA,ES:COMN
0000    B80000          9               MOV             AX,SEG NAME
0003    8ED8            10              MOV             DS,AX
0005    B80000          11              MOV             AX,SEG DELAY
0008    8EC0            12              MOV             ES,AX
000A    E80000          13              CALL            DELAY
000D    E90000          14              JMP             DONE
0010    EBFE            15      DONE    JMP             DONE
                        16
                        17              COMN
0000    A10007          18      DELAY   MOV             AX,LABL+ 3
0003    48              19      AGN     DEC             AX
0004     75FD           20              JNZ             AGN
0006    C3              21              RET

Errors=  0
```

"Types" Associated With Data Locations

The "types" of data location labels are determined by the implied size of pseudo instructions DB, DW, and DD.

- The "type" associated with the label NAME is BYTE.
- The "type" associated with the label LABL is WORD.
- The "type" associated with the label VALU is DWORD.

Note that any expression involving these labels (NAME+ 1, LABL+ 2, or VALU-4) will be of the same type as the label.

"Types" Associated With Instruction Locations

The remaining labels in the example program above appear at instruction locations, and are associated with the either type NEAR or type FAR.

By default the assembler assigns "type" NEAR to all instruction location labels. Therefore, in the example program above, DELAY in the CALL DELAY instruction and DONE in the first JMP DONE instruction default to "type" NEAR.

In the example the last JMP DONE instruction has one less byte of code generated than the first. The assembler generates a "short" jump for this instruction because the label DONE had already been recognized and evaluated to be within the value of -128 to + 127 bytes. **JMP instructions whose operand labels have been previously defined in the program allow the assembler to generate the most efficient code possible.**

Three Conditions to Remember About "Types" When Writing Programs

Assembly language instructions operate on byte, word, and double-word size quantities. In most cases, the type of operation of an assembly language instruction is determined by the "types" associated with the operands of that instruction.

Three conditions must be remembered about "types" associated with operands:

Condition 1

If an assembly language instruction has two operands and each is associated with a "type", then the "types" of those operands must agree with each other.

Condition 2

If an assembly language instruction has two operands, and a "type" is only associated with one of the operands, then the operation will be of that same "type".

Condition 3

If no "type" is implied in an instruction's operand(s), and different "type" operations are allowed for the instruction, then a "type" must be specified in the operand.

When Instructions Have Two Operands, and Both Imply A "Type"

When an assembly language instruction has two operands, and there is a "type" associated with both operands, both "types" must agree with each other. For example, if the following instruction were added to the program above, an ET (Expression Type Invalid) error would occur.

```
MOV             AL,LABL
```

The error occurs because the "type" implied by AL is BYTE and the "type" associated with the label LABL is WORD. However, the following instruction will not cause an error

because the types associated with the operands do agree with each other.

```
MOV             AL,NAME+ 1
```

When "Types" Associated with Operands Disagree

It is possible to move the byte at location LABL to register AL by doing one of two things:

- Use a type override.
- Create a new label, whose "type" is BYTE, for the same memory location.

These two subjects are further discussed in the Using Keyword Operators section which follows.

When Instructions Have Two Operands, And Only One Is Associated With A "Type"

When an assembly language instruction has two operands, and only one of these operands has an associated "type", the operation will be of that "type". For example:

```
MOV         AX,[BX]
MOV         AL,[BX]
```

Both of these instructions assemble with no errors because the "type" of the operation is implied in only one operand. The first instruction moves a word of memory from the location addressed by register BX because the type associated with AX is WORD. The second instruction moves a byte because the "type" associated with register AL is BYTE.

When No 'Types" Are Associated With Instruction

Assembly language instructions whose operations may be of different "types", and whose operands imply no "type" will cause the assembler to generate error messages. For example, consider the following instructions.

```
POP     [BP]
MOV     AX,# 0003
MUL     [BP]
MOV     [BP],# 4
```

Neither the first, third, or fourth instruction above appears to imply a "type". Only the last two instructions cause error messages. No error message occurs for the first instruction because type of operation is implied by the instruction POP. Only words may be popped from the stack.

On the other hand, the MUL instruction may be either an 8-bit multiply or a 16-bit multiply. No "type" is implied by the MUL operand. This instruction will cause an ET (Expression Type Invalid) error message.

In the last instruction, the assembler doesn't evaluate whether the immediate value is supposed to placed in a memory location for byte width, word or double-word.
Consequently, this instruction will cause an IO (Invalid Operand) error message to occur.

Assigning "Types" to Operands Which Imply No "Type"

In the MUL and MOV instructions above, the assembler needs more information to evaluate what code to generate. Keyword operators must be added to the operand to direct the assembler to the correct and expected "type" of operation.

Using Keyword Operators

Keyword operators are necessary in some assembly language instructions to give further information to the assembler. The following actions are accomplished in assembly language instructions using keyword operators:

- Specify "types" in operands which imply no type.
- Override the "type" associated with a program label.
- Associate more than one "type" to a memory location.
- Create immediate operands whose values are determined by characteristics of program labels.

Fourteen keyword operators are defined by the 8086/8088 assembler. The keyword operators are briefly summarized in table 6-1.

Table 6-1. Keyword Operators

Keyword Operator	Description
BYTE	Defines operation to be byte type (1 byte long).
WORD	Defines operation to be word type(2 bytes long).
DWORD	Defines operation to be double-word type (4 bytes long).
NEAR	Informs the assembler that the label associated with the call or forward jump will be in the same segment.
FAR	Informs the assembler that the label associated with the call or forward jump will be in another segment.
PTR	Used in conjunction with BYTE, WORD, DWORD, NEAR, and FAR keyword operators (e.g., BYTE PTR, WORD PTR, etc.) in assembly language instruction operands to override the "type" associated with a label, or to specify the type of an operation if none is implied.

Table 6-1. Keyword Operators (Cont'd)

Keyword Operator	Description
SHORT	Informs the assembler that the label which appears in the operand of a forward JMP instruction is within + 127 bytes.
THIS	Used with the EQU pseudo instruction to create a label (with type BYTE, WORD, DWORD, NEAR, or FAR) for the instruction that follows (e.g., LABL EQU THIS WORD).
HIGH	Creates an assembly language instruction immediate operand whose value is the high-order byte of a label's offset value. label's offset value.
LOW	Creates an assembly language instruction immediate operand whose value is the low-order byte of a label's offset value.
OFFSET	Creates an assembly language instruction immediate operand whose value is the offset (from the segment base) of a label's address.
SEG	Creates an assembly language instruction immediate operand whose value is the segment of a label's address.
SIZE/TYPE	Creates an assembly language immediate operand whose value is a number associated with the "type" of a label. The size values of the various types are: BYTE 1 NEAR 0 WORD 2 FAR 7 DWORD 4
LENGTH	Creates an immediate operand whose value is 1.

Assigning "Types" to Operands Which Imply None

Let's return now to the previous example instruction in which the assembler could not evaluate the size of the operation. The instruction was:

```
MUL     [BP]
```

To correct the ET (Expression type invalid) error that occurs when assembling this instruction, you must specify in the operand whether the multiply should be 8-bit or 16-bit. Adding the BYTE PTR keyword operators to the memory operand, [BP], will indicate to the assembler that the multiplication should be 8-bit. Adding the keyword operators WORD PTR will indicate that the operation should be 16-bit. Either of the following instructions will be assembled without error messages:

```
MUL     BYTE PTR [BP]
MUL     WORD PTR [BP]
```

Type Overrides

In the LAST program example we could move a byte of memory from location NAME+ 1 to register AL. Now suppose you want to move the first two bytes at memory location NAME into register AX. To do this change the instruction to:

```
MOV     AX,NAME
```

This instruction causes an ET (Expression type invalid) error to occur during assembly. The error occurs because we attempt moving a BYTE sized memory operand into a WORD sized register. **Size of assembly language operands must agree.**

To cause the assembler to accept this instruction, override the type associated with the label NAME. A type override will change the "type" of a program label in that instruction's operand only. To override the BYTE type of the NAME label, add the keyword operators WORD PTR to

the memory operand NAME as follows:

```
MOV          AX,WORD PTR NAME
```

This instruction causes assembler generated code that will move the first two bytes, or the first word, at location NAME into register AX.

Using Near Type Overrides

NEAR "types" associated with instruction location referencing operands as a default may also be overridden.
Instructions whose memory operands include references to instruction locations are JMP and CALL. Consider the JMP and CALL usage in examples below:

```
LOCATION   OBJECT CODE          SOURCE   LINE
                                1        "8086"
0000       E80900               2                CALL            DELAY
0003       E80009               3                CALL            DELAY2
0006       E90A00               4                JMP             OVER
0009       E90000               5                JMP             OVER2
                                6
000C       B80300               7        DELAY   MOV             AX,# 3
000F       48                   8        AGN             DEC     AX
0010       75FD                 9                JNZ             AGN
0012       C3                   10               RET
                                11
0013       E8F6FF               12       OVER    CALL            DELAY
0016       E80009               13               CALL            DELAY2
0019       EBFE                 14       DONE    JMP             DONE
                                15
                                16               COMN
0000       E90013               17       OVER2   JMP             OVER
0003       E90019               18               JMP             DONE
0006       E8000C               19               CALL            DELAY
                                20
0009       B80300               21       DELAY2  MOV             AX,3
000C       48                   22       AGN2    DEC             AX
000D       75FD                 23               JNZ             AGN2
000F       C3                   24               RET
Errors= 0
```

From the instructions above note that the assembler associates the "type" NEAR with all the CALL and JMP instruction operands, except for backward jumps to labels in the same segment. (The assembler generates three bytes of code for NEAR instructions above.)

Using FAR PTR Type Overrides

Suppose that two program segments will ultimately be linked at addresses which are separated by more than 64K bytes. In this condition, specify that any calls or jumps between the two segments are of type FAR. Using FAR PTR type overrides accomplishes this. Adding type overrides to intersegment JMP and CALL instruction operands, will result in the code shown below.

```
LOCATION   OBJECT CODE          SOURCE LINE
                                1       "8086"
0000       E80F00               2               CALL    DELAY
0003       9A0000000F           3               CALL    FAR PTR DELAY2
0008       E90E00               4               JMP     OVER
000B       EB0C                 5               JMP     SHORT OVER
000D       EA00000000           6               JMP     FAR PTR OVER2
                                7
0012       B80300               8       DELAY   MOV     AX,# 3
0015       48                   9       AGN     DEC     AX
0016       75FD                 10              JNZ                     AGN
0018       C3                   11              RET
                                12
0019       E8F6FF               13      OVER    CALL    DELAY
001C       9A0000000F           14              CALL    FAR PTR DELAY2
0021       EBFE                 15       DONE   JMP     FAR PTR DONE
                                16
                                17              COMN
0000       EA00000019           18      OVER2   JMP     FAR PTR OVER
0005       EA00000021           19              JMP     FAR PTR DONE
000A       9A00000012           20              CALL    FAR PTR DELAY
                                21
000F       B80300               22      DELAY 2 MOV     AX,3
0012       48                   23      AGN 2   DEC     AX
0013       75FD                 24              JNZ     AGN2
0015       C3                   25              RET
Errors=  0
```

Notice that the FAR PTR keyword operators have no effect on the last JMP instruction in the PROG segment (PROG is initially the default code area). **When JMP instruction operands contain labels whose addresses are backward relative to the current program counter address, the assembler will generate code for the shortest possible JMP instruction, regardless of any attempted type overrides.** This is demonstrated when the assembler generates code for a short (within 0FFH bytes) jump in the JMP FAR PTR DONE instruction in the example above.

Using the SHORT Keyword Operator

The SHORT in the previous programs JMP SHORT OVER instruction is yet another keyword operator. The SHORT keyword operator is used when the label in a JMP instruction's operand is a forward reference, i.e., the label is defined later on in the program, and within 0FFH bytes.

Using the LABEL Pseudo Instruction

An alternative exists to issuing type overrides in assembly language instructions as a means of changing the "type" associated with a memory location. You can assign more than one type to the same memory location with the LABEL or EQU pseudo instructions. Both LABEL and EQU are special 8086/8088 pseudo instructions. The LABEL pseudoinstruction is equivalent to a combination of the EQU pseudo instruction and the THIS keyword operator. Consider the following instructions:

```
LOCATION   OBJECT CODE          SOURCE  LINE

                                   1        "8086"
                                   2
                                   3         DOUBLE    LABEL   DWORD
                                   4        UPWORD     LABEL   WORD
0000       43                      5        BYTE3      DB      43H
0001       22                      6        BYTE2      DB      22H
                                   7         LOWORD    LABEL   WRD
0002       CC                      8        BYTE 1     DB      0CCH
0CCH       0003 8A                 9         BYTE 0    DB      8AH
                                   10
                                   11                   PROG
                                   12                   ASSUME  CS:PROG,DS:DATA
0000       B80000                  13                   MOV     AX,SEG DOUBLE
0003       8ED8                    14                   MOV     DS,AX
                                   15
0005       C53E0000                16                   LDS     DI,DOUBLE
0009       C53E0000                17                   LDS     DI,DWORD PTR UPWORD
000D       C53E0000                18                   LDS     DI,DWORD PTR BYTE3
                                   19
0011       FF2E0000                20                   JMP     DOUBLE
0015       FF2E0000                21                   JMP     DWORD PTR UPWORD
0019       FF2E0000                22                   JMP     DWORD PTR BYTE3
                                   23
001D       A10000                  24                   MOV     AX,UPWORD
0020       A10000                  25                   MOV     AX,WORD PTR BYTE3
                                   26
0023       A00003                  27                   MOV     AL,BYTE0
0026       A00003                  28                   MOV     AL,BYTE PTR LOWORD+ 1

Errors=  0
```

Here, with four bytes of memory, DOUBLE is defined which is assigned type DWORD. UPWORD and LOWORD are assigned type WORD. BYTE3, BYTE2, BYTE1, and BYTE0 are assigned type BYTE. Equivalent instructions, using different labels, are written in the PROG segment to show that different labels refer to the same memory locations.

Using the THIS Keyword Operator

The same program repeated below uses the EQU pseudo instruction in conjunction with the THIS keyword operator.

```
LOCATION   OBJECT CODE          SOURCE  LINE

                                1       "8086"
                                2                       DATA
                    < 0000>     3       DOUBLE          EQU     THIS DWORD
                    < 0000>     4       UPWORD          EQU     THIS WORD
0000        43                  5       BYTE 3          DB      43H
0001        22                  6       BYTE 2          DB      22H
                    < 0002>     7       LOWORD          EQU     THIS WORD
0002        CC                  8       BYTE 1          DB      0CCH
0003        8A                  9       BYTE 0          DB      8AH
                                10
                                11                      PROG
                                12                      ASSUME  CS:PROG,DS:DATA
0000        B80000              13                      MOV     AX,SEG DOUBLE
0003        8ED8                14                      MOV     DS,AX
                                15
0005        C53E0000            16                      LDS     DI,DOUBLE
0009        C53E0000            17                      LDS     DI,DWORD PTR UPWORD
000D        C53E0000            18                      LDS     DI,DWORD PTR BYTE3
                                19
0011        FF2E0000            20                      JMP     DOUBLE
0015        FF2E0000            21                      JMP     DWORD PTR UPWORD
0019        FF2E0000            22                      JMP     DWORD PTR BYTE3
                                23
001D        A10000              24                      MOV     AX,UPWORD
0020        A10000              25                      MOV     AX,WORD PTR BYTE 3
                                26
0023        A00003              27                      MOV     AL,BYTE0
0026        A00003              28                      MOV     AL,BYTE PTR LOWORD+ 1

Errors=  0
```

Creating labels with different "types" also applies to instruction location labels. To illustrate how different "types" may be assigned to the same instruction location, the next program makes type FAR all intersegment jumps and calls by associating the same instruction location with different "types".

```
LOCATION   OBJECT CODE          SOURCE  LINE
                                1       "8086"
0000       E80900               2                   CALL    DELAY
0003       E8000F               3                   CALL    DELAY2 ERROR-IO
0006       E90A00               4                   JMP     OVER
0009       E90000               5                   JMP     OVER2
ERROR-IO, see line3                                 ^
                                6
                                7       FAR_DELAY   PROC    FAR
000C       B80300               8       DELAY       MOV     AX,# 3
000F       48                   9       AGN         DEC     AX
0010       75FD                 10                  JNZ     AGN
0012       CB                   11                  RET
                                12
                                13      FAR_OVER    LABEL   FAR
0013       E8F6FF               14      OVER        CALL    DELAY
0016       E8000F               15                  CALL    DELAY2
ERROR-IO,see line 5                                 ^
                                16      FAR_DONE    LABEL   FAR
0019       EBFE                 17       DONE       JMP     DONE
                                18
                                19                  COMN
                                20      OVER 2      LABEL   FAR
0000       EA00000013           21                  JMP     FAR_OVER
0005       EA00000019           22                  JMP     FAR_DONE
000A       9A0000000C           23                  CALL    FAR_DELAY
                                24
                                25       DELAY2     PROC    FAR
000F       B80300               26                  MOV     AX,3
0012       48                   27      AGN 2       DEC     AX
0013       75FD                 28                  JNZ     AGN 2
0015       CB                   29                  RET

Errors=  3,previous error at line15
IO - Invalid Operand, Invalid or unexpected operand encountered or operand is missing
```

Using the PROC Pseudo Instruction

PROC is a new special 8086/8088 pseudo instruction used in the program above. PROC pseudo instruction operates in the same way as the LABEL pseudo except that only the "types" NEAR or FAR may be associated with the next instruction location.

Notice that intersegment jumps and calls in the COMN segment all have the "type" FAR associated with their operands. (The assembler generates five bytes of code for FAR jumps or calls in the program above.)

Error messages are caused by intersegment JMP and CALL instructions in the PROG segment (initially the default code area). Errors occur because intersegment JMP and CALL operands contain labels that are defined later on in the program. Forward references to labels that are assigned "type" FAR must contain type overrides. Adding the FAR PTR keyword operators to the forward referencing JMP and CALL instruction operands will result in the code shown below.

```
LOCATION  OBJECT CODE          LINE      SOURCE        LINE

                               1         "8086"
0000      E80D00               2                       CALL      DELAY
0003      9A0000000F           3                       CALL      FAR PTR DELAY2
0008      E90C00               4                       JMP       OVER
000B      EA00000000           5                       JMP       FAR PTR OVER2
                               6
                               7         FAR_DELAY     PROC      FAR
0010      B80300               8         DELAY         MOV       AX,# 3
0013      48                   9          AGN          DEC       AX
0014      75FD                 10                      JNZ       AGN
0016      CB                   11                      RET
                               12
                               13        FAR_OVER      LABEL     FAR
0017      E8F6FF               14        OVER          CALL      DELAY
001A      9A0000000F           15                      CALL      FAR PTR DELAY2
                               16        FAR_DONE      LABEL     FAR
001F      EBFE                 17        DONE          JMP       DONE
                               18
                               19                      COMN
                               20         OVER2        LABEL     FAR
0000      EA00000017           21                      JMP       FAR_OVER
0005      EA0000001F           22                      JMP       FAR_DONE
000A      9A00000010           23                      CALL      FAR_DELAY
                               24
                               25        DELAY 2       PROC      FAR
000F      B80300               26                      MOV       AX,3
0012      48                   27        AGN 2         DEC       AX
0013      75FD                 28                      JNZ       AGN2
0015      CB                   29                      RET

Errors= 0
```

Other Keyword Operators

HIGH, LOW, OFFSET, SEG, SIZE, and TYPE are all keyword operators used with program labels to create assembly language instruction immediate operands. (See table 6-1 for the individual descriptions of these keyword operators.)

Predefined Symbols

When writing assembly language programs you need to be aware that certain symbols have been predefined and may not be used as symbols (labels) in your programs. Predefined symbols include register names, and special operands for pseudo instructions. The predefined symbols are shown in table 6-2.

Table 6-2. Predefined Symbols

Microprocessor Register Names				
AH	BP	CX	ES	
AL	BX	DH	IP	
AX	CH	DL	SI	
BH	CL	DS	SP	
BL	CS	DX	SS	
Keyword Operators				
BYTE	LOW	PTR	BYTE	
DWORD	NEAR	QWORD	THIS	
FAR	NOTHING	SEG	TYPE	
HIGH	OFFSET	SHORT		WORD
		SIZE		
Segment Names				
ORG	PROG	DATA	COMN	
Pseudo Instruction Operands				
EO	DPL0	DPL2	RO	
ER	DPL1	DPL3	RW	
Miscellaneous Symbols				
ABS	LENGTH	MODULE_NUMBER	ST	

Operands

Register Operands

Forms register operands may take are:

16-BIT GENERAL REGISTERS	8-BIT REGISTERS	SEGMENT REGISTERS
AX	AH- High byte of AX	CS
BX	AL - Low byte of AX	DS
CX	BH - High byte of BX	ES
DX	BL - Low byte of BX	SS
SP	CH - High byte if CX	
BP	CL - Low byte of CX	
SI	DH - High byte of DX	
DI	DL - Low byte of DX	

The accumulator may be either AX or AL.

Default Register Operands

The default register AL is for byte operations. The following instructions accomplish identical results:

```
MOV     BYTE PTR[BX]
MOV     AL,BYTE PTR[BX]
```

The default register AX is for word operations. The assembler will generate identical code for the following instructions:

```
ADD     WORD PTR[SI],
ADD     WORD PTR[SI],AX
```

The comma in the first instruction is necessary. Otherwise, the assembler will interpret AX as the destination operand.

Immediate Operands

Forms that immediate operands may take are:

16-BIT IMMEDIATE OPERANDS	8-BIT IMMEDIATE OPERANDS
# 0 - # 0FFFFH	# 0 - # 0FFH
SEG ABEL	HIGH ABEL
OFFSET ABEL	LOW ABEL
	SIZE ABEL
	TYPE ABEL

Unless one of the keyword operators is used to create an immediate operand, immediate operands must be prefixed by the pound (#) symbol. If operands are not prefixed by a keyword operator or a pound sign, the first character in each operand must be a digit.

The "type" WORD will be associated with operands containing the SEG and OFFSET keyword operators. Likewise, the "type" BYTE will be associated with the operands containing the HIGH, LOW, SIZE, and TYPE keyword operators. (No "type" is implied with number operands.)

Valid forms for program labels (identified by < LABEL> above) are discussed in the source file format chapter 5.

Memory Operands

Forms that memory operand may take are shown below:

ADDRESSING MODE	MEMORY OPERAND FORMS
Direct	ABEL
Register Indirect	[BX]
	[BP]
	[SI]
	[DI]
Based	< DISP> [BX]
	< DISP> [BP]
Indexed	< DISP> [SI]
	< DISP> [DI]
Based Indexed	< DISP> [BX] [SI]
	< DISP> [BX] [DI]
	< DISP> [BP] [SI]
	< DISP> [BP] [DI]

All memory operands may contain segment and type override prefixes.

In the based indexed addressing mode, the order in which the base and index registers appear does not matter.

Optional displacement value < DISP> can be a program label (whose format is discussed in the source file format chapter 5), or an immediate value (containing the pound (#) prefix only).

Base or index registers enclosed in brackets may be followed by plus (+) or minus (-) constant expressions as shown below:

```
[BX+ VALU]
[SI-7+ 3]
[BP+ 80H]
[DI+ VALU-4]
```

Label VALU in the operands shown above must be equivalent to a constant expression.

String Operations

All string operations use source index register (SI) to address the source operands, which are assumed to be in the current data segment (segment contained in DS). (The source segment may be overridden.)

Destination index register (DI) is used to address the destination operands, which are assumed to be located in the current extra segment (segment contained in ES). (The destination segment must always be in ES.)

If the direction flag (DF) is cleared, the operand pointers are incremented after each operation: once for byte operations and twice for word operations. If the DF flag is set, the operand pointers are decremented after each operation.

.

7

Pseudo Instruction Summary

Introduction

This chapter describes the HP Model 64000 assembler pseudo instructions. The pseudo instructions are used for listing control, program counter, linkage control, and constant definitions.

An assembler pseudo may be either an instruction to the assembler or a request for so special service. Most pseudos require no memory space because, unlike microprocessor instructions, they produce no object code.

The pseudo instruction descriptions are organized alphabetically in this chapter. A summary of the pseudoinstructions is shown below.

HP 64000 Pseudo Instructions

Pseudo	Function
ASC	Stores data in memory in ASCII format.
BIN	Stores data in memory in binary format.
COMN	Assigns common block of data or code to a specific location in memory.
DATA	Assigns data to a specific location in memory.
DECIMAL	Stores data in memory in decimal format.
END	Terminates the logical end of a program module. Operand field can be used to indicate the starting address in memory for program execution.
EXPAND	Causes an output listing of all source and macro generated codes.
GLB	Defines symbol that is used globally (referenced by other program modules).
HEX	Stores data in memory in hexadecimal format.
IF	Allows sections of code to be conditionally assembled.
INCLUDE	Allows a secondary file to be included in the source input stream.
LIST	Used to modify output listing of program.
MASK	Performs AND/OR logical operations on designated ASCII string.
NAME	Permits used to add comments for reference in the load map.
NOLIST	Suppresses output listings (except error messages).
OCT	Allows user to store data in octal format.
ORG	Sets program counter to specific memory address for absolute programming.
PROG	Assigns source statements to a specific location in memory. Assembler default condition is PROG storage area.
REAL	Converts real decimal numbers to IEEE binary floating point constants.
REPT	Enables user to repeat a source statement any number of times.
SET	Defines label field symbol with operand field value. Symbol can be redefined.
SKIP	Enables user to skip to a new page to continue program listing.
SPC	Enables user to generate blank lines within program listing.
TITLE	Enables user to create a test line at the top of each page listing for the source program.
WARN/ NOWARN	Turn warning message in source listing ON or OFF.

Special 8086/8088 Pseudo Instructions

Pseudo	Function
ALIGN	Increments the current program counter address, if odd.
ASSUME	Associates a segment register with a segment name (PROG, DATA, COMN, or ORG).
DB	Defines and initializes BYTE length memory locations.
DBS	Reserves memory space in bytes.
DD	Defines and initializes DWORD (double-word) length memory locations. DD can also be used to define real numbers.
DDS	Reserves memory space in double-words.
DW	Defines and initializes WORD length memory locations.
DWS	Reserves memory space in words.
EQU	Equates label field symbol with operand value. Operands may be the same as any regular instruction operand. The label field symbol cannot be redefined.
EXT	Defines external symbols. May also associate the symbol with a segment register and a TYPE.
LABEL	Creates and assigns a TYPE to the label field symbol. The address of the symbol will be the next memory location.
PROC	Creates a procedure label of type FAR or NEAR.
SET	Allows symbols to be defined and assigned values. A symbol defined with the SET pseudo may have its value changed later on in the program with another SET pseudo instruction.

Pseudo Instruction Syntax

The following descriptions list and define each assembler and control instruction in detail. They are listed alphabetically.

Note

Special 80286, 8087, and 8089 pseudo instruction descriptions are found in their respective appendices.

ALIGN

Align to Word Boundary
(Special 8086/8088 Series Pseudo)

Syntax

Label	Operation	Operand
	ALIGN	

Description

If the current program counter address is odd, ALIGN will increment the program counter by one. This optimizes word data storage, since access time is less when an address is on a word boundary.

Example

```
LOCATION  OBJECT CODE  LINE  SOURCE LINE
                       1     "8086"
0000      0001         2     NAME1   DB      0,1
0002      02           3     NAME2   DB      2
                       4             ALIGN
0004      0000         5     NAME3   DW      NAME1
0006      0002         6     NAME4   DW      NAME2

Errors= 0
```

ASC

Store ASCII Data in Memory

Syntax

Label	Operation	Operand
[symbol]	ASC	string expression
	or	
[symbol]	ASCII	string expression

Description

The ASC pseudo instruction allows the user to store ASCII text in memory using quotation marks, apostrophes or carets (^) as delimiters. The first delimiter must be used as the terminating delimiter.

The ASCII character(s) specified in the operand field may be in the form of a string expression.

Example

```
LOCATION  OBJECT CODE   LINE  SOURCE LINE
                        1     "8086"
0000      4142436162    2               ASC     "ABCabc"
0005      63
0006      4141274127    3               ASCII   "AA'A'AA"
000B      4141
000D      4242224141    4               ASCII   'BB"AA"BB'
0012      224242
Errors= 0
```

ASSUME

Assume Segment Location
(Special 8086/8088 Series Pseudo)

Syntax

Label	Operation	Operand
	ASSUME	segreg:segnam [,...]
	or	
	ASSUME NOTHING (default)	

Description

The ASSUME instruction informs the assembler of the addresses contained in the segment registers. The instruction is required prior to any memory references which do not explicitly name a segment register.

The ASSUME declaration associates a segment register with a segment name. All references to items in the named segment cause segment override prefixes to be generated if necessary.

The ":segnam" portion of the syntax statement must be one of the following:

PROG
DATA
COMN
ORG
NOTHING

The ASSUME NOTHING instruction removes all former assumptions as to which base addresses were in which segment registers. This turns off the implicit generation of segment-overrides. **Initially, all segment registers are assumed to NOTHING.**

ASSUME (Cont'd)

Example

```
LOCATION    OBJECT CODE    LINE    SOURCE LINE
                           1       "8086"
0000                       2       LABEL       DWS     2
0004        2EA10000       3                   MOV     AX,DS:LABEL
0008        A1             4                   MOV     AX,LABEL
ERROR-IO
                           5                   ASSUME  CS:PROG
0009        2EA10000       6                   MOV     AX,LABEL

Errors= 1,previous error at Line4
IO-Invalid Operand, Invalid or unexpected operand encountered or operand is missing.
```

BIN

Store Word Length Binary Data in Memory

Syntax

Label	Operation	Operand
[symbol]	BIN	binary number(s)
	or	
[symbol]	BINARY	binary number(s)

Description

The BIN pseudo instruction allows the user to store data in binary format in memory.

The number(s) specified in the operand field is (are) written in binary format. If more than one operand is specified, each must be separated from the other by a comma. Each operand specifies a 16 bit word.

Example

```
LOCATION  OBJECT CODE  LINE  SOURCE LINE
                       1     "8086"
0000      0008         2                BIN     1000
0002      0006000C     3                BINARY  110,1100
0006      00030006     4     LBL        BIN     011,0110
Errors= 0
```

COMN/DATA/PROG Designated Memory Storage Area

Syntax

Label	Operation	Operand
	COMN or DATA or PROG	

Description

Three program counters are used to identify areas of relocatable code. The areas are designated as data (DATA), program (PROG), and common (COMN). You can change from one relocatable area to another using these pseudoinstructions.

PROG and DATA instructions function identically. They are merely two names that identify two separate, relocatable memory areas. Common (COMN) allows construction of a common block of data used by different program modules. The default area is PROG.

Normally, the default memory area (PROG) will be used when constructing a source program. The DATA memory area might occupy another part of memory. DATA can be used for storing data, tables, instructions, etc.

The COMN pseudo can be used to group information that is common to a number of program modules. Assigning these type of items to a specific area in memory facilitates program modification and referencing.

COMN DATA PROG (Cont'd)

Note 

All information assigned to the COMN area in memory must be grouped in one program file. If two or more files assign information to the COMN area, the linker will overlay the first data stored with the second block of data assigned, thereby erasing the first block of data. However, this feature may be useful in the design of software systems requiring overlays.

Example

```
LOCATION  OBJECT CODE  LINE  SOURCE LINE
                       1     "8086"
                       2                COMN
0000      4558414D50   3                ASC       "EXAMPLE"
0005      4C45
                       4                DATA
0000      00030008     5     LBEL       BIN       011,1000,011
0004      0003
                       6                PROG
0000      BB0000       7                MOV       BX,# LBEL
Errors= 0
```

DB

Define Byte
(Special 8086/8088 Series Pseudo)

Syntax

Label	Operation	Operand
[Name]	DB	[expression,...]

Description

The DB instruction may be used to accomplish the following:

- Initialize memory locations.
- Define the type characteristic of variables.

When used with a variable expression in the label field, the DB instruction defines the variable to be type "BYTE". The DB instruction cannot be used to initialize memory storage using an address expression in the operand field. To initialize storage with characters, enclose the characters in quotation marks or apostrophes. The DB instruction is the only legal instruction for strings that contain more than two characters. Each character in the string requires one byte of memory.

DB (Cont'd)

Example

```
LOCATION  OBJECT CODE  LINE  SOURCE LINE
                       1     "8086"
0000      0001         2     V1        DB      0,1
0002      05           3     V2        DB      5
0003      4142434445   4     V3        DB      "ABCDE"
0000      8A00000      5               MOV     AL,DS:V1
000B      8A1E0001     6               MOV     BL,DS:V1+ 1
000F      8A2E0002     7               MOV     CH,DS:VS

Errors= 0
```

DBS

Define Byte Storage
(Special 8086/8088 Series Pseudo)

Syntax

Label	Operation	Operand
Name	DBS	expression

Description

The DBS instruction reserves space in bytes. Expression must be a valid assembly time expression with no forward references. DBS causes the program counter to be incremented by the value of expression. It will define NAME as a valid byte variable.

Example

```
LOCATION  OBJECT CODE  LINE  SOURCE LINE
                       1     "8086"
0000                   2     TABLE   DBS   10
000A      00           3     V1      DB    0
Errors= 0
```

DD

Define Double-Word
(Special 8086/8088 Series Pseudo)

Syntax

Label	Operation	Operand
[Name]	DD	[expression,...]

Description

The DD instruction may be used to accomplish the following:

- Initialize memory locations.
- Define the type characteristic of variables.

When used with a variable expression in the label field, the DD instruction defines the variable to be type double-word.

When an address expression is used in the operand field, the DD instruction will initialize two words of memory with the segment and offset of the variable.

When using character strings to initialize memory storage, the length of the character string is restricted to a maximum of two characters. The characters are swapped and placed in the low order word, with the high word being zero. If only a single character is used, it will be placed in the high byte of the low word.

The DD operation can also be used to define real-number pseudos. Real numbers are always expressed in decimal values. Be sure to include the decimal point. You may use either the normal decimal or the scientific form of the expression. You may also specify either positive or negative numbers and exponents (+ /-n.mE+ /-x). Positive numbers are assumed if you do not specify.

DD (Cont'd)

Example

LOCATION	OBJECT CODE	LINE	SOURCE LINE		
		1	"8086"		
0000	00000000	2	P1	DD	P1
0004	0000001C	3	P2	DD	L
0008	00000000	4	P3	DD	P1
000C	0000E442	5	J1	DD	114.0
0010	0000C842	6	J2	DD	1.0E2
0014	00401CC6	7	J3	DD	-1.0E4
0018	8FC2F53D	8	J4	DD	120.0E-3
001C	C51E0000	9	L	LDS	BX,DS:P1
0020	C4E60008	10		LES	SI,DS:P3
0024	2EFF2E0004	11		JMP	CS:P2

Errors= 0

DDS

Define Double-Word Storage
(Special 8086/8088 Series pseudo)

Syntax

Label	Operation	Operand
Name	DDS	expression

Description

The DDS instruction reserves space in double words. Expressions must be valid assembly expressions with no forward references. DDS causes the program counter to be incremented by four times the value of the expression. It will define NAME as a valid double word expression.

Example

```
LOCATION  OBJECT CODE  LINE  SOURCE LINE
                       1     "8086"
0000                   2     TABL      DDS     4
0010      FFFF00000    3     P1        DD      65535
Errors=  0
```

DW

Define Word
(Special 8086/8088 Series Pseudo)

Syntax

Label	Operation	Operand
[Name]	DW	[expression, ...]

Description

The DW instruction may be used to accomplish the following:

- Initialize memory locations.
- Define the type characteristic of variables.

When used with a variable expression in the label field, the DW instruction defines the variable to be type word.

When an address expression is used in the operand field, the DW instruction will initialize a word of memory with the offset of the variable.

When using character strings to initialize memory storage, the length of the character string is restricted to two characters. The characters will be swapped in memory. If a single character is specified, it will be placed in the byte with the higher numbered address.

DW (Cont'd)

Example

```
LOCATION  OBJECT CODE  LINE  SOURCE LINE
                       1     "8086"
0000      0000424102   2     X       DW      0,"AB",2
0005      00
0006      0000         3     Y       DW      X
0008      8B1E006      4             MOV     BX,DS:Y
000C      A10004       5             MOV     AX,DS:X+ 4
000F      8B07         6             MOV     AX, [BX]
0011      A30002       7             MOV     DS:X+ 2, AX

Errors=  0
```

DWS

Define Word Storage
(Special 8086/8088 Series Pseudo)

Syntax

Label	Operation	Operand
Name	DWS	expression

Description

The DWS instruction reserves space in words. Expressions must be valid assembly time expressions with no forward references. DWS causes the program counter to be incremented by twice the value of expression. It will define NAME as a valid word variable.

Example

```
LOCATION  OBJECT CODE  LINE  SOURCE LINE
                       1     "8086"
0000                   2     STRG    DWS     0AH
0014      FFFF         3     P1      DW      65535
Errors= 0
```

DECIMAL

Decimal Constant

Syntax

Label	Operation	Operand
[symbol]	DECIMAL	decimal number

Description

The DECIMAL pseudo instruction allows the user to store data in decimal format in memory.

The number(s) specified in the operand field is (are) written in decimal format. If more than one operand is specified, each one must be separated from the other by a comma.

Note 

The DECIMAL pseudo instruction can be used in the form DEC.

Example

```
LOCATION  OBJECT CODE  LINE  SOURCE LINE
                       1     "8086"
0000      007F00FF     2                 DECIMAL   127,255
0004      0000         3                 DECIMAL   65536
Errors= 0
```

END

Program Module Termination

Syntax

Label	Operation	Operand
	END	[expression]

Description

The END instruction terminates the logical end of a program module. It is optional. If omitted, the program will be automatically terminated after the last statement in the program module being edited.

The optional expression in the operand field represents the starting address in memory for program execution. This address initializes the program counter when the file is loaded during emulation. The expression must be an absolute or relocatable value (not an external symbol reference).

Example

```
LOCATION  OBJECT CODE  LINE  SOURCE LINE
                       1     "8086"
0000      007F00FF     2           DECIMAL   127,255
0004      0000         3           DECIMAL   65536
                       4           END

Errors= 0
```

EQU

Equate
(Special 8086/8088 Series Pseudo

Syntax

Label	Operation	Operand
[symbol]	EQU	expression

Description

The EQU instruction is used to establish a relationship between a symbol and an expression. The symbol in the label field acquires the same value as the expression in the operand field. Redefinition of the symbol is not permitted. If the operand field of an EQU instruction contains another symbol, it must be defined previously in the source program.

Example

```
LOCATION  OBJECT CODE  LINE  SOURCE LINE
                       1     "8086"
0000      0000         2     X          DB      0,0
          < 0002>      3     TWO        EQU     # 2
          < 0001>      4     X1         EQU     DS:WORD PTR
                                                X [BP+ 1] [DI]
          < 0000>      5     XH         EQU     HIGH X
0002      6602060000   6                ADD     XX
ERROR-ET                                        ^
ERROR-ET, see Line 6                            ^
ERROR - IO, see Line 6                          ^
          < 0000>      7     XX         EQU     AX
0007      03C0         8                ADD     AX

Errors =  3, previous error at line 6
ET - Expression Type, The type of expression is not valid or the operand is not valid
IO - Invalid Operand, Invalid or unexpected operand encountered or operand is missing
```

EXPAND

Listing of Macro Expansions

Syntax

Label	Operation	Operand
	EXPAND	

Description

The EXPAND instruction can be used in the assembler directive statement or embedded in the source program. If embedded in the source program, it will generate, within the output listing, all macro and data expansions that follow it.

You may exit the EXPAND output listing mode by embedding the LIST directive in the proper location within the source program.

Example

```
LOCATION  OBJECT CODE     LINE  SOURCE LINE
                          1     "8086"
                          2                EXPAND
0000      4142436162      3                ASC       "ABCabc"
0005      63
0006      4141274127      4                ASCII     "AA'A'AA"
000B      4141
000D      4242224141      5                ASCII     'BB"AA"BB'
0012      224242
                          6                LIST
0015      4142436162      7                ASC       "ABCabc"
001B      414274127       8                ASCII     "AA'A'AA"
0022      4242224141      9                ASCII     'BB"AA"BB'
Errors= 0
```

EXT

Define External Symbols
(Special 8086/8088 Series Pseudo)

Syntax

Label	Operation	Operand
	EXT	SYMBOL1,SYMBOL2
	or	
	EXTRN	SYMBOL1,SYMBOL2
	or	
	EXTERNAL	SYMBOL1,SYMBOL2

Note

Do not use the "EXT" short form of "EXTERNAL" in the "70108", "70116", "70320", and "70330" microprocessor modes. The NEC processors recognize an instruction with the mnemonic name of "EXT." Using the "EXT" form of the EXTERNAL pseudo op will cause a conflict with the instruction mnemonic "EXT" and cause an "IO - Invalid Operand" error at the pseudo op location. Use either "EXTRN" or "EXTERNAL" to refer to this pseudo op when in the NEC microprocessor modes.

Description

The EXT instruction permits the optional listing of segment register (segreg) and type required for the 8086 and 8088 microprocessors. The EXT instruction also provides a list of symbols referenced in this program module but defined in another program module. **When multiple symbols are listed in the operand field, they must be separated by commas.**

The optional TYPE, when required, may be BYTE, WORD, DWORD, NEAR, FAR, or ABS. (The ABS type allows you to declare constant numbers, which have been declared global

EXT (Cont'd)

in other modules, as externals.) If no TYPE is declared, the assembler assigns NEAR by default. If no segreg is declared, the assembler assigns it to NOTHING by default.

Example

```
LOCATION  OBJECT CODE   LINE  SOURCE LINE
                        1     "8086"
                        2             EXT     CS:SAM,
                                              CS:GEORGE
                        3             EXT     DS:PETE, WORD
                                              FRED
0000      A00000        4             MOV     AL,PETE
0003      26A10000      5             MOV     AX,ES:FRED
0007      E90000        6             JMP     SAM
000A      9A00000000    7             CALL    FAR PTR GEORGE

Errors= 0
```

GLB

Define Global Symbols

Syntax

Label	Operation	Operand
	GLB	SYMBOL1,SYMBOL2
	or	
	GLOBAL	SYMBOL1,SYMBOL2

Description

Symbols that are defined in one program module and referenced by other program modules must be declared global in the program module where they are defined.

Example

```
LOCATION  OBJECT CODE  LINE  SOURCE LINE
                       1     "8086"
                       2                GLB       LBEL
                       3                GLOBAL    TABLE
0000                   4     TABLE      DDS        3
          < 2000>      5     LBEL       EQU       2000H
Errors= 0
```

HEX

Store Hexadecimal Data in Memory

Syntax

Label	Operation	Operand
[symbol]	HEX	hexadecimal number

Description

The HEX pseudo instruction allows the user to store data in hexadecimal format. The number(s) specified in the operand field is (are) written in hexadecimal format. If more than one operand is specified, each one must be separated from the other by a comma.

Example

```
LOCATION  OBJECT CODE  LINE  SOURCE LINE
                       1     "8086"
0000      00FFFFFF     2                HEX   FF,0FFFF
0004      000A00FE     3     EXEC       HEX   A,FE,05,5F,7,81
000C      00070081
0010      FFFF         4                HEX   FFFFFF
Errors= 0
```

IF

Conditional Assembly
(Special 8086/8088 Series Pseudos

Syntax

Label	Operation	Operand
	IF	< absolute expression>
		.
	ELSE	.
		.
	IFEND or ENDIF	
	or	
	IF	< absolute expression>
		.
		.
		.
	IFEND or ENDIF	

Description

The IF pseudo instruction allows sections of code to be conditionally assembled. Sections of code are assembled or skipped based on an absolute expression. This expression is treated as a Boolean function with either a TRUE or a FALSE value.

The IF instruction evaluates an absolute expression as a logical function with the value zero FALSE and a nonzero value TRUE. When the expression evaluates to a nonzero (TRUE) condition, the code following the IF instruction is assembled until an ELSE or IFEND or ENDIF instruction is encountered. If the expression evaluates to zero (FALSE), then the ELSE part of the IF instruction is assembled until an IFEND or ENDIF is found. The expression type must be absolute (type= 0). All symbolic references must be defined before being used with a IF instruction. The lower 32 bits of the expression value are used to determine the true or false condition. The IFEND or ENDIF instructions are used to terminate the IF instruction. They must either follow the

IF (Cont'd)

ELSE instruction or the IF instruction if no ELSE portion is desired.

Conditional IF instructions can be nested up to 20 levels deep. If the nesting levels exceed 20, then an I0 (invalid operand) error will be flagged on the IF instruction. If an error is flagged on an ELSE or IFEND/ENDIF instruction, a nesting level error has occurred. One of these three instructions was encountered before an IF instruction or more IFEND or ENDIF instructions were found than IF
instructions. The end of the assembly source is treated as an IFEND or ENDIF instruction and no error is flagged if the assembler is currently in an IF instruction.

Example

```
LOCATION  OBJECT CODE  LINE  SOURCE LINE
                       1     "8086"
          < 0058>      2     TRUE      EQU     58H
          < 000>       3     FALSE     EQU     0
                       4
                       5               IF      TRUE
0000      B85800       6               MOV     AX,TRUE
                       7               ELSE
                       8               MOVE    AX,FALSE
                       9               ENDIF
                       10
                       11              IF      FALSE
                       12              MOVEW   AX,TRUE
                       13              ELSE
0003      B80000       14              MOV     AX, FALSE
                       15              ENDIF

Errors= 0
```

INCLUDE

Include Secondary File in Source Input

Syntax

Label	Operation	Operand
	INCLUDE	< Host-Specific File Naming Conventions>

Description

The INCLUDE pseudo instruction allows a secondary file to be included in the source input stream. Only one level of inclusion is allowed. Nested INCLUDE files will result in an error message.

Example

```
LOCATION  OBJECT CODE  LINE  SOURCE LINE
                       1     "8086"
                       2                 INCLUDE   C:/USERID/BIN
                       +     "8086"
ERROR-IS               ^
0000      000B         +                 BIN       1011
0002      0006000C     +                 BINARY    110,1100
0006      00030006     +     LBEL        BIN       011,0110
Errors=  1, previous error at line 2
IS - Illegal Symbol, Syntax expected an identifier and encountered an invalid character or term
```

LABEL

Label
(Special 8086/8088 Series Pseudo)

Syntax

Label	Operation	Operand
Name	LABEL	[type]

Description

The LABEL instruction may be used to create a symbol name and assign a type to that symbol. Available types that may be assigned to a symbol are: BYTE, WORD, DWORD, NEAR, and FAR. **If no type is specified, the default is NEAR.**

Example

```
LOCATION  OBJECT CODE     LINE  SOURCE LINE
                          1     "8086"
                          2     MEM_WORD    LABEL   WORD
0000      2214            3     MEM_BYTE    DB      22H,14H
0002      A10000          4                 MOV     AX,DS:MEM_WORD
0005      A10000          5                 MOV     AX,DS:WORD PTR
                                                    MEM_BYTE
0008      A00000          6                 MOV     AL,DS:MEM_BYTE
000B      E90A00          7                 JMP     DONE_NEAR
000E      EA00000018      9                 JMP     FAR PTR DONE_FAR
00113     9A00000018      10                CALL    FAR PTR   DONE_FAR
                          11    DONE_FAR    LABEL   FAR
0018      EBFE            12    DONE_NEAR   JMP     DONE_NEAR
001A      EBFC            13                JMP     FAR PTR DONE_FAR
001C      9A100000018     14                CALL    DONE_FAR

Errors= 0
```

LIST

Format Assembler Listfile

Syntax

Label	Operation	Operand	Comment
	LIST or LIST	< decimal number>	;controlled ;listing

Description

The LIST instruction can be used in the assembler directive statement or embedded in the source program. If embedded in the source program, it will generate one line of output for each line of source code that follows it.

The output listing can be controlled so that a desired number of lines per output page can be achieved. See the ASSEMBLING YOUR PROGRAMS (chapter 3) for information on specifying the page length of assembler output listings.

Note

All LIST instructions embedded in the source program will be overridden if any list option is specified in the assembler directive statement. (Refer to chapter 2 for assembler directive statement definition.)

LIST (Cont'd)

Example

```
LOCATION  OBJECT CODE   LINE  SOURCE LINE
                        1     "8086"
                        2           LIST    56
                        3           LIST
0000      4142436162    4           ASC     "ABCabc"
0006      4141274127    5           ASCII   "AA'A'AA"
000D      4242224141    6           ASCII   'BB"AA"BB'
                        7           EXPAND
0015      4142436162    8           ASC     "ABCabc"
001A      63
001B      4141274127    9           ASCII   "AA'A'AA"
0020      4242224141    10          ASCII   'BB"AA"BB'
0027      224242

Errors= 0
```

MASK

Set Mask

Syntax

Label	Operation	Operand
	MASK	(AND),(OR)

Description

The MASK instruction permits masking of ASCII strings. The instruction affects ASCII strings only and will produce a logical 'AND' operation with each ASCII character followed by a logical 'OR' operation. (The OR operand is optional. However, the 'OR' operation is always performed.)

The initial MASK conditions are:

AND = OFFH, and OR= 00H

Note

When MASK is used with two operands, and then later with only one, the previous second operand is still active.

MASK (Cont'd)

Example

```
LOCATION  OBJECT CODE  LINE  SOURCE LINE
                       1     "8086"
0000      41424344     2           ASC     "ABCD"
                       3           MASK    22H
0004      00020200     4           ASC     "ABCD"
                       5           MASK    OFFH
0008      41424344     6           ASC     "ABCD"
                       7           MASK    OFFH,55H
000C      55575755     8           ASC     "ABCD"
                       9           MASK    22H
0010      55575755     10          ASC     "ABCD"
Errors= 0
```

NAME

Add Comments to Load Map Listing

Syntax

Label	Operation	Operand	Comment
	NAME	"SALPHA"	;character ;string

Description

The NAME instruction is used to add comments to the object module for reference on the load map listing. The name string may contain any combination of characters, numbers, or special characters. NAME is limited to a maximum of 22 characters.

Example

```
"8086"
                NAME        "UP TO 22 CHARACTERS"
                DB          OFFH
```

Example Linker (Load Map) Listing

PROGRAM DATA COMMON ABSOLUTE	DATE	TIME	COMMENTS
000000	Tue, 10 Mar 1985 ,	1:35	UP TO 22 CHARACTERS
000001			
1			

NOLIST

No Output Listing

Syntax

Label	Operation	Operand
	NOLIST	

Description

The NOLIST instruction can be used in the assembler directive statement or embedded in the source program. If embedded in the source program, it will suppress the output listing of all source statements following it. If used in the assembler directive statement, it will suppress all output listings except error messages.

Example Source File

```
"8086"
            NOLIST
            ASC         "ABCabc"
            ASCII       "aa'A'AA"
            ASCII       'BB"AA"BB'
            EXPAND
            ASC         "ABCabc"
            ASCII       "AA'A'AA"
            ASCII       'BB"AA"bb'
```

NOLIST (Cont'd)

Example Assembler Listing

```
LOCATION  OBJECT CODE     LINE  SOURCE LINE
                          1     "8086"
                          6                EXPAND
0015      4142436162      7                ASC       "ABCabc"
001A       63
001B      4141274127      8                ASCII     "AA'A'AA"
0020       4141
0022      4242224141      9                ASCII     'BB"AA"BB'
0027      224242

Errors= 0
```

OCT

Store Octal Data in Memory

Syntax

Label	Operation	Operand
[symbol]	OCT	octal number
	or	
[symbol]	OCTAL	octal number

Description

The OCT pseudo instruction allows the user to store data in octal format.

The number(s) specified in the operand field is (are) written in octal format. If more than one operand is specified, each one must be separated from the other by a comma.

Example

```
LOCATION  OBJECT CODE  LINE  SOURCE LINE

                       1     "8086"
0000      07770077     2                 OCT     777,77
0004      00050067     3                 OCT     5,67,03
0008      0003
000A      7777         4                 OCT     77777777
Errors= 0
```

ORG

Absolute Code Area

Syntax

Label	Operation	Operand
	ORG	address

Description

The ORG instruction is used for absolute programming. It sets the contents of the location counter to the address entered in the operand field. The next statement, following the ORG instruction, will be located at the address specified.

Note

The ORG instruction cannot be used to alter the relocatable area counters associated with the DATA, PROM, and COMN instructions. The relocatable area instructions do not contain operands. Their associated counters start at zero and are initialized at linking time.

When using the ORG directive care should be taken to ensure that the assigned memory location will not result in memory overlap during the link operation.

A label symbol is generally not used in the operand field of this instruction. However, if a symbol is entered, it must be defined in a label field of a prior statement in the source program. The symbol must be an absolute expression.

PROC

Procedure Definition
(Special 8086/8088 Series Pseudo)

Syntax

Label	Operation	Operand
[Name]	PROC	[type]

Description

The PROC instruction tells the assembler that the codes following will be designated as type FAR or NEAR. The PROC instruction remains in effect until another PROC instruction is given. If no PROC statement is used without designating a type, the assembler assumes NEAR.

The value of a label will be the current location of the program counter and it will be of the type listed in the operand field.

The PROC instruction (explicit or implicit) has the following effects:

- Associates the label with the current value of the program counter and types the label.
- Determines whether the corresponding RET instruction will be coded as an intersegment (between segments) return or as an intrasegment (within segment) return.
  - If a PROC FAR definition is used, then the corresponding RET is coded as intersegment.
  - If a PROC NEAR is used (or in the default case), then the corresponding RET is coded as intrasegment.

The ENDP statement is illegal.

PROC (Cont'd)

Example

```
LOCATION  OBJECT CODE   LINE  SOURCE LINE
                        1     "8086"
0000      E80800        2           CALL    P1
0003      9A0000000D    3           CALL    P2

0008      40            4     P1    INC     AX
000C      C3            5           RET
                        6     P2    PROC    NEAR
000D      43            7           INC     BX
000E      CB            8           RET
Errors= 0
```

The first call (to P1) is an implicit PROC because of the RET instruction. It defaults to NEAR. The second call (to P2) is explicitly identified as a PROC NEAR. If the NEAR was not present in this definition, then the default would still be NEAR.

PROC (Cont'd)

Note 

An error can occur if you try to make far procedure call to a routine within the same file. As an example, the following program will generate a Legal Range error because the DWS instruction has pushed out of the 64K segment limit. A more common case (and more easily fixed) might be where a single file contains a large amount of program code that causes the segment limit to be exceeded. Changing the type of the PROC to FAR does not solve the problem;
instead, a different kind of error is caused. To avoid these errors, do not create code that will require a FAR CALL within the same file.

```
                        1   "8086"
0000        E80000      2               CALL    P1
ERROR - LR                                      ^
0003                    3   USESPCE     DWS     65536
                        4   P1          PROC
0003        40          5               INC     AX
0004        C3          6               RET

Errors=         1, previous error at line 2
```

LR - Legal Range, Address or displacement is out of range of the instructions's addressing capability.

REAL

Real Number

Syntax

Label	Operation	Operand
	REAL	real decimal number

Description

The REAL instruction converts real decimal numbers to IEEE binary floating point constants. Short (32-bit) or long (64-bit) IEEE binary floating point values can be generated by the REAL instruction.

A real decimal number must start with a decimal digit(s), followed by a decimal point, and end with a decimal digit(s). Powers of 10 are added after the decimal digit with an "E" or "L" qualifier. Real decimal numbers specified with an "E" qualifier or with no qualifier are converted to short real binary floating point (32 bits). The "L" qualifier indicates a long real number.

Numbers are converted to the IEEE standard for real numbers and stored high to low; where, the highest byte (containing the sign bit) is stored at the lowest address value and the lowest byte is stored at the highest address.

REAL (Cont'd)

Example

```
LOCATION  OBJECT CODE  LINE  SOURCE LINE
                       1     "8086"
0000      3F8000000    2     REAL    1.0       ;= 1
0004      43C80000     3     REAL    1.0E2     ;= 100.
0008      3C2eD70A     4     REAL    1.0E-2    ;= 0 .01
000C      BC23D70A     5     REAL    -1.0E-2-  ;= -0.01
0010      40590000     6     REAL    -1.0L2    ;LONG REAL=  100
0014      00000000

Errors= 0
```

REPT

Repeat Next Source Statement

Syntax

Label	Operation	Operand
	REPT	number

Description

The REPT instruction is used to repeat the next source statement any given number of times.

Example

```
LOCATION  OBJECT CODE  LINE  SOURCE LINE

                       1     "8086"
                        2                REPT    4
0000      48           +                DEC     AX
0001      48           +                DEC     AX
0002      48           +                DEC     AX
0003      48           3                DEC     AX

Errors=  0
```

SET

Define Symbol
(Special 8086/8088 Series Pseudo)

Syntax

Label	Operation	Operand
symbol	SET	expression

Description

The SET pseudo instruction allows a symbol to be defined and assigned a value. It is similar to the EQU pseudo, except with SET the value can be changed during the assembly process. The expression used must be absolute (type = 0) and all symbolic references must be defined before they are used.

Example

```
LOCATION  OBJECT CODE  LINE  SOURCE LINE
                       1     "8086"
          < 1000>      2     EXEC        SET     1000H
          < 1004>      3     EXEC        SET     EXEC+ 4
Errors= 0
```

SPC

Line Space

Syntax

Label	Operation	Operand
	SPC	[number]

Description

Whenever a SPC instruction is encountered in the source program, the assembler will space downward (line feed) a specified number of lines. The number of line feeds required is indicated in the operand field. If the operand field is left blank, the assembler will generate one blank line. The SPC instruction is printed in the output listing only if an error exists in the operand field.

SPC (Cont'd)

Example Source File

```
        "8086"
EXEC    SET     1000H
        SPC     3
EXEC    SET     EXEC+ 4
```

Example Listing

```
LOCATION  OBJECT CODE  LINE  SOURCE LINE
                       1  "  8086"
          < 1000>      2     EXEC        SET     1000H


          < 1004>      4     EXEC        SET     EXEC+ 4
Errors = 0
```

TITLE

Change Listfile Title

Syntax

Label	Operation	Operand
	TITLE	"Name"

Description

The TITLE instruction will initiate a page eject and create a "Name" line at the top of each page listing for the source program that follows. The title may be 70 characters in length and may be changed any number of times during the program.

This statement, if inserted as the second statement in the source program (directly after the assembler directive), will cause the title to be printed on the first page listing the source program and thereafter on the top of each page. Alternatively, if the TITLE instruction is inserted in the program at some place other than the second statement of the source program, the instruction will initiate a page eject and the new title will be printed at the top of the new page and each page thereafter.

Example

```
Hewlett Packard:          Sample Title:

LOCATION    OBJECT CODE   LINE    SOURCE LINE


Errors = 0
```

WARN/NOWARN

Warning/No Warning

Syntax

Label	Operation	Operand
	WARN or NOWARN	

Description

The NOWARN instruction turns off the warning message in the source line. The WARN instruction restores it.

8

Using Macro Instructions

Introduction

This chapter discusses macro directives, their use and construction. Using macro definitions (macros) eliminates the repetitious writing of the same sequence of instruction during source code construction.

Any legitimate sequence of instructions may be incorporated into a macro. This process is called "macro definition". Once defined, a single macro call may be used at any point in the source program to insert a sequence of instructions defined by the macro definition. The insertion of a sequence of instructions is referred to as "macro expansion".

Advantages of Using Macros

A macro definition provides a means of producing, at program assembly time, a commonly used sequence of assembler statements as many times as needed. The sequence of statements is specified just once as a macro. Thereafter, at any point in the program where these statements are to be produced, a single macro call will cause the sequence to be generated.

Using macros properly will serve to:

- Simplify program coding.
- Significantly reduce programming errors otherwise caused by rewriting similar instructions throughout the program.
- Ensure that common functions are performed by standard routines.
- Improve program readability.
- Reduce duplication of effort among programmers assigned to the project.

Disadvantages of Using Macros

Variables used in a macro are only known within it. Such variables are local rather than global. This can create unecessary confusion. Other disadvantages of macros are:

- Repetition of the same macro may create many instructions.
- Possible effects on registers and flags that may not be clearly stated.

Macros –vs– Subroutines

In some situations, a subroutine, rather than multiple in-line macro statements, can reduce overall program size. Subroutines require branching, then returning, from another part of the program. Subroutines usually increase program execution time. **Variables in a subroutine are evaluated only during program execution further slowing program execution. Macro parameters are evaluated at assembly time and do not slow down execution as much as subroutines.**

Macro Format

A macro definition consists of four parts that must appear in the order given below:

1. Header statement.
2. Macro definition name.
3. Macro definition body.
4. Trailer statement.

Header Statement

First the header statement occurs to specify both the name of the new macro instruction and the formal arguments (parameters) that will be used in the macro instruction. General macro header syntax is:

```
Name        MACRO       [optional parameters]
```

Macro Definition Name

Next the name of the macro definition is written in the label field of the source statement and must not be terminated by a colon (:). To avoid multiple-label conflicts, the assembler treats labels within macros as local labels, applying only to that particular macro. MACRO is written in the operation field of the source statement. The optional parameters follow in the operand field of the source statement.

Macro Definition Body

Next the body of a macro definition must define the action of the macro instruction. **There is no limit to the number of instructions that may appear in a macro definition body. Fields within the macro body are the same as those of an assembler instruction. Rules for forming a macro statement resemble the rules for forming an assembler instruction.**

Note

Macro definition bodies may contain the names of other macros. In other words, macros may be defined in terms of other macros. Macro bodies may not contain nested macros definitions. A nested macro definition would be a completely new macro defined within the macro body of another macro. Nested macros are not allowed.

Macro Trailer Statement

The last consideration is the trailer statement must consist of a single line. The operand field of the line contains the word MEND (macro end).

An example of a macro instruction is as follows:

Example

```
Label     Operation     Operand     Command

SAVE      MACRO
          OPC           EXEC1
          OPC           SAVEA
          OPC           EXEC2
          OPC           SAVEB
          MEND
```

Note 

The opcode symbol (OPC) listed in the operation field will take the form of a mnemonic instruction for the specific microprocessor being programmed.

Calling Macros

To call the SAVE macro, insert the macro name in the operation field of the source statement and the code in the body of the macro will be generated in the program as if it had been typed there. The generated instructions will be printed in the listing of the program (only if the expand list option is specified).

Example

```
SAVE
OPC       EXEC1
OPC       SAVEA
OPC       EXEC2
OPC       SAVEB
```

Optional Parameters

Formal parameters of a macro definition are often referred to as symbolic variables. Macro symbolic parameters (as distinguished from ordinary labels or symbols) are those symbols that may be assigned different values by the programmer. **When assembler instructions are generated according to the macro definition, dummy parameters are replaced by values that have been assigned to them. Three simple rules must be followed when forming dummy parameters:**

- The first character of the dummy parameter must be an ampersand (&).
- The second character of the dummy parameter must be a letter of the alphabet. All remaining characters, if any, can be letters or numbers.
- Any number of parameters or parameters of various lengths may be entered in the operand field of a macro definition. However, the entire line length must not exceed 110 characters (not including a carriage return). In addition, after arguments are substituted for parameters in a macro call, the lines resulting from the macro expansion must not exceed 110 characters. If the 110 character length is exceeded, an error message is issued.

Symbolic Parameters

Symbolic parameters used in the macro definition are assigned values by the programmer in each macro call referencing that particular macro. An example of the general syntax for symbolic parameters is:

```
Label       Operation     Operand

ADDS        MACRO         &SUBNAM,&PARAM
            JP            &SUBNAM
            DEF           &PARAM
            MEND
```

Assigning parameters to the ADDS macro develops:

```
ADDS    ADD,SUM+ 27
JP      ADD
DEF     SUM+ 27
```

Text Replacement and Concatenation

Macros may also be used for text replacement. Macros can also perform concatenation of a parameter to generate a new word. Consider the following macro instruction:

```
Label   Operation   Operand

SAVE    MACRO       &EXEC4,&PARM1,&PARM2
        LD&EXEC4    &PARM1
        ST&EXEC4    &PARM2
```

You may now call this simple macro instruction, assign your own parameters, and produce the following insert into your program:

```
        SAVE    A,EXEC2,EXEC3
        LDA     R0,EXEC2
        LD      EXEC3,R0
```

Note the substitution of actual parameters of call A, EXEC2, EXEC3 - for dummy parameters in the macro heading (&EXEC4, &PARM1, and &PARM2). Note further that the sequence of call parameters interchange directly with the sequence of the dummy parameters.

Note

A macro does not necessarily produce the same source code each time it is called. Changing the parameters in a macro call will change the source code that the macro generates.

Unique Label Generation

The macro assembler generates unique local labels each time a macro is called by using four ampersand characters in a label (&&&&). When a macro is called, &&&& is replaced by four decimal digits. Note, this four-digit constant is incremented every time a macro is called, even if the ampersand characters are not in the macro label. With this labeling, a macro can be called more than once in a program (no duplication of label).

Example

```
1                "8086"
2
3                TEXT MACRO &STRING
4
5                L1_&&&& DB L2_&&&&-L1_&&&&-1        ;Length of string.
6                        ASC &STRING
7                L2_&&&&
8
9                MEND
10
11               TEXT "STRING # 1"
+
+                L1_0001 DB L2_0001-L1_0001-1        ;Length of string.
+                        ASC "STRING # 1"
+                L2_0001
+
12
13               TEXT "STRING # 2"
+
+                L1_0002 db L2_0002-L1_0002-1  :     ;Length of string.
+                        ASC "STRING # 2"
+                L2_0002
+
```

Conditional Assembly

Four conditional assembly instructions are available for use with the HP 64000 Assembler. When inserted among the statements in the body of a macro definition, they provide the means for instructing the assembler to branch and loop among the statements of the executable program. These conditional assembly instructions will not be printed in the listing of the program (unless they contain an error). Only their effects can be seen in the generated object code. The four conditional instructions are:

.SET
.IF
.GOTO
.NOP

.SET Instruction

The .SET instruction provides a way to assign or modify an expression value of a macro local. This instruction assigns the value of the operand field to the name specified in the label field. When the label is encountered subsequently in the macro program, the assembler substitutes its new value. This value remains unchanged until altered by a subsequent .SET instruction. The general format of a .SET instruction is as follows:

Label	Operation	Operand
name	.SET	expression

An example of a .SET instruction is as follows:

```
GENTABLE        MACRO       &COUNT
LOOP_COUNT      .SET        &COUNT
LOOP_TOP        .NOP
                DEF         1
                DEF         2
                DEF         3
LOOP_COUNT      .SET        LOOP_COUNT-1
                .IF.        LOOP_COUNT.GT.0 LOOP_TOP
                MEND
```

Call expansion:

```
GENTABLE    3
DEF         1
DEF         2
DEF         3
DEF         1
DEF         2
DEF         3
DEF         1
DEF         2
DEF         3
```

.IF Instruction

The .IF instruction is the conditional-branch instruction using six types of relational operators. These operators are:

.EQ.	= = =	equal
.NE	= = =	not equal
.LT.	= = =	less than
.GT.	= = =	greater than
.LE.	= = =	less than or equal
.GE.	= = =	greater than or equal

Note  All relational operator comparisons are 32 bits unsigned.

An .IF instruction has the following format:

Operation	Operand
.IF	Exp .(Relational Operator). Exp Label

The .IF instruction directs the assembler to relationally compare two expressions. If the value of this comparison is true, a branch is taken to the statement named by the label symbol in the operand field. Otherwise, the statement immediately following the .IF instruction is processed by the assembler.

.GOTO Instruction

The .GOTO statement is the unconditional-branch instruction. It has the following format:

Operation	Operand
.GOTO	Label

The .GOTO instruction directs the assembler to branch, unconditionally, to the statement named by the label symbol in the operand field.

.NOP Instruction

A .NOP instruction is a no-operation instruction. This instruction is useful with .IF and .GOTO instructions when branching is required to sections of the program that are not labelled. The .NOP instruction format is as follows:

Label	Operation
Label	.NOP

When a branch is taken to a .NOP instruction, the effect is the same as if a branch were taken to the statement immediately following it.

Note 

Conditional assembly instructions generate no source code. The sole function of the .SET, .IF, .GOTO, and .NOP instructions are to conditionally alter the sequence in which the assembler processes the source program or macro definition instructions.

Example

```
CONDITION   MACRO   &P1,&P2,&P3
            .IF     &P1 .EQ. 1      LOAD
            .IF     &P1 .EQ. 2       STORE
            .GOTO   DONE
LOAD        .NOP
            OPC     &P2
            OPC     &P3
            .GOTO   DONE
STORE       .NOP
            OPC     &P3
            OPC     &P2
DONE        .NOP
            MEND
```

Some call expansion examples are as follows:

```
CONDITION    1,EXEC2,BLUE
OPC          EXEC2
OPC          BLUE
CONDITION    2,EXEC2,BLUE
OPC          BLUE
OPC          EXEC2
CONDITION    0
             < NOCODE>
```

Checking Macro Definition Parameters

When using macro calls, you may want to omit specific parameters defined in the macro definition. This is accomplished by using the null symbol ("") or a comma (,). For example:

Macro definition:

```
EXEC2        MACRO        &P1,&P2,&P3,&P4
```

Macro call:

```
EXEC2        ,EXEC3,"",0FCH
```

In the above example, &P2 is assigned a value of EXEC3 and &P4 a value FCH. &P1 and &P3 parameters are omitted.

An example of a macro expansion is as follows:

```
CALLSUB     MACRO     &SUB,&P1,&P2,&P3
            JP        &SUB
            .IF       &P1 .EQ. '"' DONE
            DEF       &P1
            .IF       &P2 .EQ. '"' DONE
            DEF       &P2
            .IF       &P3 .EQ. '"' DONE
            DEF       &PS
DONE        .NOP
            MEND
```

Note

When testing for null parameters, if a WARNING statement is generated, enclose the macro parameter designator in quotation marks (see "&SP" below) and compare it with the null parameter indicator (single quotation marks) enclosed in quotation marks. For example:

```
IF    "&SP" .EQ. "'" DONE
```

Three expansion call examples are as follows:

```
(a.)    CALLSUB   ADD,PARAM
        JP        ADD
        DEF       PARAM

(b.)    CALLSUB   ADD
        JP        ADD

(c.)    CALLSUB   ADD,IN,OUT,RESULT
        JP        ADD
        DEF       IN
        DEF       OUT
        DEF       RESULT
```

Indexing Parameters

The assembler can, when instructed, index through a parameter list to determine if all or certain parameters are present. Indexing is accomplished by using a macro local symbol prefaced with two ampersands (&&). The following macro directive is an example to index parameters:

```
1.   CALLSUB    MACRO   &P1,&P2,&P3,&P4
2.   JMP        &P1
3.   PARAM      .SET    2
4.   PARAM_LOOP .NOP
5.              .IF     &&PARAM.EQ.'"'"JUMP
                        _OUT MERGE
6.              DEF     &&PARAM
7.   PARAM      SET     PARAM+ 1
8.              .GOTO   PARAM_LOOP
9.   JUMP_OUT   .NOP
10.             MEND
```

A line-by-line explanation of the above macro example follows:

Line 1.	Defines the macro directive and CALL-SUB with its dummy parameters &P1, &P2, &P3, &P4.
Line 2.	Completes a subroutine call designated by parameter &P1.
Line 3.	Sets name PARAM to a value of 2.
Line 4.	Assigns a .NOP statement the name PARAM_LOOP.
Line 5.	Since the PARAM label has been assigned the value 2 (see line 3), the .IF statement checks to see if the second parameter of the macro call statement has been omitted. If it has, the .IF statement causes the program to branch to the JUMP_OUT statement.

Note

During each iteration of the PARAM_LOOP, the value of PARAM is increased by 1 (see line 7). The iterations continue until the .IF statement is satisfied.

Line 6.	Updates the value of PARAM to the current value assigned.
Line 7.	Adds 1 to the current value of PARAM.
Line 8.	Loops to PARAM_LOOP.
Line 9.	Uses a .NOP statement to exit the PARAM_LOOP iteration.
Line 10.	Ends the macro.

Three macro expansions of the previous macro example are as follows:

```
(a.)    CALLSUB   ADD
        JP        ADD

(b.)    CALLSUB   ADD,LOC1,LOC2
        JP        ADD
        DEF       LOC1
        DEF       LOC2

(c.)    CALLSUB   ADD,P1,P2,P3
        JP        ADD
        DEF       P1
        DEF       P2
        DEF       P3
```

This concludes the discussion concerning the use of macros in assembly programs.

Notes

A

8086/8088 Series Instruction Set Summary

Introduction

This appendix contains a summary of the 8086/8088/ 80186/80286 instruction sets. Included is a table describing the operands which appear in the instruction set summary. The instruction set summary is presented first in table A-2. The table of operand forms (table A-3) follows the instruction set summary.

Table A-1. Conditional Jump Flags

JA/JNBE	(CForZF) = 0
JAE/JNB	CF = 0
JB/JNAE	CF = 1
JBE/JNA	(CForZF) = 1
JC	CF = 1
JE/JZ	ZF = 1
JG/JNLE	((SFxorOF)orZF) = 0
JGE/JNL	(SFxorOF) = 0
JL/JNGE	(SFxorOF) = 1
JLE/JNG	((SFxorOF)orZF) = 1
JNC	CF = 0
JNE/JNZ	ZF = 0
JNO	OF = 0
JNP/JPO	PF = 0
JNS	SF = 0
JO	OF = 1
JP/JPE	PF = 1
JS	SF = 1

Table Convention

In Table A-2, the information is organized in the following manner:

Mnemonic	General Operand	Short Description
	Specific Operands	A more complete description that may include discussion of any specific operands that might be valid for the instruction.

Table A-2. Instruction Set Summary

AAA	**(no operands)**	**ASCII adjust AL after addition**
		Used after the ADD instruction, this instruction will, if the lower nibble of AL is greater than 9 or if the auxiliary carry flag is 1, add 6 to AL, increment AH, and set the carry and auxiliary carry flags. Otherwise, the carry and auxiliary carry flags are reset. The upper nibble of AL will always be 0 after this instruction. Register AL will contain the decimal digit result.
AAD	**(no operands)**	**ASCII adjust AX before division.**
		This instruction will prepare two unpacked BCD digits in AH and AL for division by adding 10xAH to AL and setting AH to zero.
AAM	**(no operands)**	**ASCII adjust AX after multiply.**
		Used after two unpacked BCD digits have been multiplied, this instruction will unpack the result (found in AL) of the unpacked BCD multiplication, leaving the most significant digit in AH and the least significant digit in AL.
AAS	**(no operands)**	**ASCII adjust AL after subtraction.**
		Used after the subtraction of one BCD digit from another whose byte result is in AL, this instruction will, if the lower nibble of AL is greater than 9 or if the auxiliary carry flag is 1, then decrement AL by 6, decrement AH, and set the carry and auxiliary carry flags. Otherwise, the carry and auxiliary carry flags are reset.The upper nibble of AL will always be 0 after instruction. Register AL will contain the decimal digit result.

Table A-2. Instruction Set Summary (Cont'd)

ADC	destination,source	Add with carry.
	register,register register,memory memory,register register,immediate memory,immediate accumulator,immediate	This instruction adds the source operand and the value of the carry flag to the destination operand. The destination will contain the result of the operation.
ADD	**destination,source**	**Addition.**
	register,register register,memory memory,register register,immediate memory,immediate accumulator,immediate	This instruction adds the source operand to the destination operand. The destination will contain the result of the operation.
AND	**destination,source**	**Logical AND.**
	register,register register,memory memory,register register,immediate memory,immediate accumulator,immediate	This instruction performs a logical AND operation on the operands. The destination will contain the result of the AND operation

Table A-2. Instruction Set Summary (Cont'd)

ARPL (80286)	**destination,source**	**Adjust RPL field of selector.**
	mem16,reg16 reg16,reg16	This 80286 Protected Mode instruction will compare the RPL field (bottom two bits) of the operands. If the RPL field of the destination operand is less than the RPL field of the source operand, then the zero flag is set to 1 and the RPL field of the first operand is increased to that of the source operand. If not,the zero flag is set to 0 and the destination RPL field is not changed.
BOUND (80186/80286)	**destination,source**	**Check array index against bounds**
	reg16,mptr32	This 80186/188, 80286 instruction compares the destination operand to two words in memory. The destination operand must be greater than or equal to the first memory word and less than or equal to the second memory word. An INT 5 will occur if the destination operand does not meet the condition above.
CALL	**target**	**Call a procedure.**
	near-proc far-proc memptr16 regptr16 memptr32 call-gate task-gate TSS	This instruction calls a procedure. The procedure may be in the current code segment (near-proc, memptr16, and regptr16 operands), or in another code segment (far-proc, and memptr32 operands). For the 80286 Protected Mode, calls may also be made to call gates, task gates, and to Task State segments (TSSs) using the far-proc operand.

Table A-2. Instruction Set Summary (Cont'd)

CBW	**(no operands)**	**Convert byte to word.**
		This instruction extends the signed byte in AL to a signed word in AX.
CLC	**(no operands)**	**Clear carry flag.**
		This instruction sets the carry flag to 0.
CLD	**(no operands)**	**Clear the direction flag.**
		This instruction sets the direction flag to 0. When the direction flag is 0, string instructions will cause the contents of the index register(s)to be incremented.
CLI	**(no operands)**	**Clear the interrupt flag.**
		This instruction sets the interrupt enable flag to 0 (if the current privilege level is at least as privileged as the IOPL in 80286 Protected Mode). External, maskable interrupts are disabled after this instruction.
CLTS (80286)	**(no operands)**	**Clear the task switched flag**
		This 80286 Privilege Level = 0 instruction clears the task switched flag in the Machine Status Word. The TS flag is set every time a task switch occurs.
CMC	**(no operands)**	**Complement the carry flag.**
		This instruction sets the carry flag if it is cleared, or clears the carry flag if it is set.
CMP	**destination,source**	**Compare destination to source.**

Table A-2. Instruction Set Summary (Cont'd)

	register,register register,memory memory,register register,immediate memory,immediate accumulator,immediate	This instruction subtracts the source operand from the destination operand and sets the flags accordingly. The result of the subtract operation is discarded
CMPS	**source-string,dest-string**	**Compare string.**
	source-string,dest-string	This instruction subtracts the source string, at [SI], to the destination string, at ES:[DI]. The flags are affected by this operation, but neither string is changed. This instruction may be preceded with the REPE or REPZ prefixes, in which case the instruction will be repeated until CX is 0. The CMPS instruction may also be preceded with the REPNE or REPNZ prefixes, in which case the instruction will be repeated until CX or the zero flag is 0. In the repeat operations, CX is decremented;SI and DI are either both incremented (if direction flag = 0) or both decremented (if direction flag = 1).
CWD	**(no operands)**	**Convert word to doubleword.**
		This instruction extends the signed word in AX to a signed doubleword in DX:AX.

Table A-2. Instruction Set Summary (Cont'd)

DAA	**(no operands)**	**Decimal adjust AL after addition.**
		Used after an ADD instruction whose result is a two digit BCD byte in AL, this instruction converts the result in AL into two BCD digits. If the lower nibble of AL is greater than 9 or if the auxiliary carry flag is 1, AL is incremented by 6 and the auxiliary carry flag is set. Otherwise, the auxiliary carry flag is reset. Next, if the upper nibble of AL is greater than 9 or if the carry flag is set, AL is incremented by 60H and the carry flag is set. Otherwise, the carry flag is cleared.
DAS	**(no operands)**	**Decimal adjust AL after subtraction.**
		Used after a subtraction instruction whose result is a two digit BCD byte in AL, this instruction converts the result in AL into two BCD digits. If the lower nibble of AL is greater than 9 or if the auxiliary carry flag is 1, AL is decremented by 6 and the auxiliary carry flag is set. Otherwise, the auxiliary carry flag is reset. Next, if the upper nibble of AL is greater than 9 or if the carry flag is set, AL is decremented by 60H and the carry flag is set. Otherwise, the carry flag is cleared.
DEC	**destination**	**Decrement operand by 1.**
	register memory	This instruction decrements the operand by 1. The carry flag is not affected by this instruction.

Table A-2. Instruction Set Summary (Cont'd)

DIV	**source**	**Unsigned division.**
	register memory	Unsigned values in AX or DX:AX are divided by the source operand byte or word, respectively. When AX is divided by the source operand byte, the quotient is stored in AL and the remainder is stored in AX. When DX:AX is divided by the source operand word, the quotient is stored in AX and the remainder is stored in DX.
ENTER (80186/80286)	**# bytes,nesting-level**	**Make stack frame for procedure parameters.**
	immediate,0 immediate,1 immediate,immed8	This 80186/188, 80286 instruction creates a stack frame. The destination operand specifies the number of bytes to be allocated for the procedure's stack. The source operand specifies the lexical nesting level of the procedure.
ESC	**external-opcode,source**	**Escape.**
	immediate,memory immediate,register	This instruction provides a way for coprocessors to obtain opcodes and memory operands from the 8086/88 or 80186/188 microprocessors. The external-opcode is an immediate value (0 - 63). The memory source operand allows the coprocessor to read the memory location.The register source operand causes the microprocessor to do nothing.
HLT	**(no operands)**	**Halt.**
		This instruction causes the processor to enter the halt state. The processor remains in the halt state until a RESET or an external interrupt occurs.

Table A-2. Instruction Set Summary (Cont'd)

IDIV	**source**	**Signed division.**
	register memory	The signed values in AX or DX:AX are divided by the source operand byte or word, respectively. When AX is divided by the source operand byte, the quotient is stored in AL and the remainder is stored in AH. When DX:AX is divided by the source operand word, the quotient is stored in AX and the remainder is stored in DX. The remainder has the same sign as the dividend (AX or DX:AX).
IMUL	**source**	**Signed multiplication.**
	reg8 reg16 memory	The source operand may be either a byte or word quantity. When the source operand is a byte, it is multiplied by AL, and the signed result is placed in AX. Carry and overflow flags are set to 0 if AH was initially a sign extension of AL. Otherwise, the carry and overflow flags are 1. When the source operand is a word, it is multiplied by AX, and the signed result is placed in DX:AX. Carry and overflow flags are set to 0 if DX was initially a sign extension of AX. Otherwise,the carry and overflow flags are 1.

Table A-2. Instruction Set Summary (Cont'd)

IMUL (80186/80286)	**dest,source**	**Signed multiplication.**
	reg16,immed8 reg16,mem16,immediate reg16,reg16,immediate	The 80186 and 80286 allow multiple operands. The two operand instruction is the same as the byte instruction above except that the result is placed in the 16 bit register specified. In the three operand instructions, the second operand is multiplied by the immediate value, and the result is placed in the register specified in the first operand. The carry and overflow flags are set to 0 if the signed result is less than -32768 or greater than 32767.
IN	**accumulator,port**	**Input from port.**
	accumulator,immed8 accumulator,DX	This instruction will cause a byte (accumulator = AL) or word (accumulator = AX) to be input from the port whose address is specified by an 8 bit immediate value or is in register DX.
INC	**destination**	**Increment operand by 1.**
	register memory	This instruction increments the operand by 1. The carry flag is not affected by this instruction.
INS (80186/80286)	**dest-string,port**	**Input from port to string.**
	dest-string,DX	This instruction transfers data from the input port specified by the contents of register DX to to the destination memory location ES:[DI]. This instruction may be preceded by the REP prefix described later in this table.

Table A-2. Instruction Set Summary (Cont'd)

INSB (80186/80286)	**(no operands)**	**Input byte(s) from port to string.**
		No operands are required with this 80186/80286 instruction because the dest-string type BYTE, and the segment override "ES:" are implied in this instruction mnemonic. The REP prefix is also allowed with this instruction.
INSW (80186/80286)	**(no operands)**	**Input word(s) from port to string.**
		No operands are required with this 80186/80286 instruction because the dest-string type WORD, and the segment override "ES:" are implied in this instruction mnemonic. The REP prefix is also allowed with this instruction.
INT	**interrupt-type**	**Call interrupt procedure.**
	immed8	This instruction calls an interrupt procedure. The immediate operand multiplied by four specifies the address of the interrupt pointer. The interrupt pointer contains the segment:offset address of the interrupt service routine. In the 80286 Protected Mode, the immediate operand is the index number of the service routine's gate descriptor in the Interrupt Descriptor Table (IDT).
INTO	**(no operands)**	**Interrupt on overflow.**
		This instruction is the same as the INT instruction except that immediate operand is implicitly 4, and the overflow flag must be set for the interrupt to be taken.

Table A-2. Instruction Set Summary (Cont'd)

IRET	**(no operands)**	**Return from interrupt.**
		This instruction pops the IP, CS, and flag registers and returns program execution to the point where it was interrupted. In the 80286 Protected Mode, the return from interrupt will cause a task switch to occur if the nested task flag is set. When a task switch occurs on an interrupt return, the service routine TSS is updated, and if the service routine task is re-entered, the code following the IRET will be executed.
JA/JNBE	**short-label**	**Jump if above/if not below or equal.**
	short-label	This instruction will cause a jump to an address within + 127 or -128 from the next IP if: (carry flag OR zero flag) = 0.
JAE/JNB	**short-label**	**Jump if above or equal/if not below.**
	short-label	This instruction will cause a jump to an address within + 127 or -128 from the next IP if: carry flag = 0.
JB/JNAE	**short-label**	**Jump if below/if not above or equal.**
	short-label	This instruction will cause a jump to an address within + 127 or -128 from the next IP if: carry flag = 1.
JBE/JNA	**short-label**	**Jump if below or equal/if not above.**
	short-label	This instruction will cause a jump to an address within + 127 or -128 from the next IP if: (carry flag OR zero flag) = 1.

Table A-2. Instruction Set Summary (Cont'd)

JC	**short-label**	**Jump if carry flag is set.**
	short-label	This instruction will cause a jump to an address within + 127 or -128 from the next IP if: carry flag = 1.
JCXZ	**short-label**	**Jump if CX is zero.**
	short-label	This instruction will cause a jump to an address within + 127 or -128 from the next IP if register CX = 0.
JE/JZ	**short-label**	**Jump if equal/if zero.**
	short-label	Instruction will cause jump to an address within + 127 or -128 from the next IP if: zero flag = 0.
JG/JNLE	**short-label**	**Jump if greater/if not less or equal.**
	short-label	This instruction will cause a jump to an address within + 127 or -128 from the next IP if: [(sign flag XOR overflow flag) OR zero flag] = 0.
JGE/JNL	**short-label**	**Jump if greater or equal/if not less.**
	short-label	This instruction will cause a jump to an address within + 127 or -128 from the next IP if: (sign flag XOR overflow flag) = 0.
JL/JNGE	**short-label**	**Jump if less/if not greater or equal.**
	short-label	This instruction will cause a jump to an address within + 127 or -128 from the next IP if: (sign flag XOR overflow flag) = 1.

Table A-2. Instruction Set Summary (Cont'd)

JLE/JNG	**short-label**	**Jump if less or equal/if not greater.**
	short-label	This instruction will cause a jump to an address within + 127 or -128 from the next IP if: (sign flag XOR overflow flag) OR zero flag= 1.
JMP	**target**	**Jump.**
	short-label near-label memptr16 regptr16 far-label memptr32 call-gate task-gate TSS	This instruction transfers program execution to the address specified by the operand. The target may be in the current code segment (short-label, near-label, memptr16, and regptr16 operands), in another code segment (far-label and memptr32 operands). In the 80286 Protected Mode, jumps may also be made to call gates, task gates, and to Task State Segments (TSSs) using the far-proc operand.
JNC	**short-label**	**Jump if carry flag is reset.**
	short-label	This instruction will cause a jump to an address within + 127 or -128 from the next IP if: carry flag = 0.
JNE/JNZ	**short-label**	**Jump if not equal/if not zero.**
	short-label	This instruction will cause a jump to an address within + 127 or -128 from the next IP if: zero flag = 0.
JNO	**short-label**	**Jump if not overflow.**
	short-label	This instruction will cause a jump to an address within + 127 or -128 from the next IP if: overflow flag = 0.

Table A-2. Instruction Set Summary (Cont'd)

JNP/JPO	**short-label**	**Jump if not parity/if parity odd.**
	short-label	This instruction will cause a jump to an address within + 127 or -128 from the next IP if: parity flag = 0.
JNS	**short-label**	**Jump if not sign.**
	short-label	This instruction will cause a jump to an address within + 127 or -128 from the next IP if: sign flag = 0.
JO	**short-label**	**Jump if overflow.**
	short-label	This instruction will cause a jump to an address within + 127 or -128 from the next IP if: overflow flag = 1.
JP/JPE	**short-label**	**Jump if parity/if parity even.**
	short-label	This instruction will cause a jump to an address within + 127 or -128 from the next IP if: parity flag = 1.
JS	**short-label**	**Jump if sign.**
	short-label	This instruction will cause a jump to an address within + 127 or -128 from the next IP if: sign flag = 1.
LAHF	**(no operands)**	**Load flags into register AH.**
		This instruction will load register AH with the low byte of the flag word.

Table A-2. Instruction Set Summary (Cont'd)

LAR (80186/80286)	**destination,source**	**Load access rights byte.**
	reg16,reg16 reg16,mem16	This 80286 Protected Mode instruction will load the access rights byte from the descriptor, whose selector is the source operand, into the high byte of the destination 16 bit register. The low byte of the 16 bit register is set to zero. If the descriptor cannot be accessed from the current privilege level or the selector RPL, the load is not performed, and the zero flag is cleared. The zero flag is set if the load is performed.
LDS/LES	**destination,source**	**Load doubleword pointer.**
	reg16,memory	This instruction will load the first word from the source memory location (offset value) into the destination register operand. The second word from the source memory location (segment or selector value) is loaded into DS or ES. When a selector value is loaded (80286 Protected Mode), the cache from the selector's associated descriptor is also loaded.
LEA	**destination,source**	**Load effective address offset.**
	reg16,memory	This instruction will load the offset of the source memory operand into the 16 bit register destination.

Table A-2. Instruction Set Summary (Cont'd)

LEAVE (80186/80286)	**(no operands)**	**High level procedure exit.**
		This 80186/188, 80286 instruction will copy BP to SP and POP BP, thereby releasing a procedure's stack space. LEAVE is the complementary operation to ENTER.
LGDT/LIDT (80286)	**memory**	**Load GDT/IDT Register.**
	memory	This 80286, Privilege Level = 0 instruction will load 6 bytes from the memory address into the Global or Interrupt Descriptor Table register. The first memory word will be the LIMIT of the GDT or IDT register, the next three bytes are the BASE, and the last byte is ignored.
LLDT (80286)	**selector**	**Load LDT Register.**
	mem16 reg16	This 80286 Protected Mode, PL = 0 instruction's selector operand should point to an LDT descriptor in the Global Descriptor Table. The LDT descriptor is loaded into the LDT register.
LMSW (80286)	**source**	**Load Machine Status Word.**
	mem16 reg16	This 80286, Privilege Level = 0 instruction will load the source word into the Machine Status Word. When this instruction is used to switch to protected mode, it must be immediately followed by an intrasegment jump instruction to flush the instruction queue. This instruction cannot be used to switch back to the real address mode.

Table A-2. Instruction Set Summary (Cont'd)

LOCK	**(no operands)**	**Assert BUS LOCK signal.**
		This instruction prefix causes the BUS LOCK signal to be asserted for duration of the instruction it prefixes. The bus is not locked for all cycles during the following instructions: CMPS, SCAS, STOS, LODS, PUSHA, POPA, CALL, RET, IRET, ENTER, BOUND, PUSH, POP, or any ESC.
LODS	**source-string**	**Load string operand.**
	source-string	This instruction loads AL or AX with the byte or word at location [SI]. The source-string operand will specify whether the operation is of type BYTE or WORD. After the load, SI is incremented if the direction flag = 0 or decremented if the direction flag = 1. Increments or decrements will be either by 1 for byte operations or 2 for word operations. This instruction may be preceded with the REP prefix, which is described later in this table.
LODSB	**(no operands)**	**Load byte string operand.**
		This instruction is the same as the LODS instruction except that no operand is required because the instruction implies a byte operation and the "DS:" segment override is assumed. The byte at DS:[SI] is loaded into AL.

Table A-2. Instruction Set Summary (Cont'd)

LODSW	**(no operands)**	**Load word string operand.**
		This instruction is the same as the LODS instruction except that no operand is required because the instruction implies a word operation and the "DS:" segment override is assumed. The word at DS:[SI] is loaded into AL.
LOOP	**short-label**	**Loop control with CX counter.**
	short-label	This instruction will decrement register CX and transfer program control to within + 127 or -128 bytes from the next IP if: CX does not equal 0.
LOOPE/LOOPZ	**short-label**	**Loop if equal/if zero.**
	short-label	This instruction will decrement register CX and transfer program control to within + 127 or -128 bytes from the next IP if: CX not equal to 0 and zero flag = 1.
LOOPNE/NZ	**short-label**	**Loop if not equal/if not zero.**
	short-label	This instruction will decrement register CX and transfer program control to within + 127 or -128 bytes from the next IP if: CX not equal to 0 and zero flag = 0.

Table A-2. Instruction Set Summary (Cont'd)

LSL (80286)	**destination,source**	**Load segment limit.**
	reg16,reg16 reg16,mem16	This 80286 Protected Mode instruction will load the limit value from the segment descriptor whose selector is the source operand into the high byte of the destination 16 bit register. If the descriptor cannot be accessed from the current privilege level or the selector RPL, the load is not performed, and the zero flag is cleared. The zero flag is set if the load is performed.
LTR (80286)	**source**	**Load task register.**
	reg16 memory register	This 80286 Protected Mode Privilege Level = 0 instruction will load the task register with the TSS selector source operand.The loaded TSS is marked busy; however, no task switch occurs.
MOV	**destination,source**	**Move.**
	memory,accumulator accumulator,memory register,register register,memory memory,register register,immediate memory,immediate seg-reg,reg16 seg-reg,mem16 reg16,seg-reg memory,seg-reg	Transfers bytes or words from the source operand to the destination operand.

Table A-2. Instruction Set Summary (Cont'd)

MOVS	**dest-string,source-string**	**Move data from string to string.**
	dest-string,source-string	This instruction moves data from the source string [SI] to the destination string ES:[DI]. The string operands are used to specify either BYTE or WORD operation. After the load, SI and DI are incremented if the direction flag = 0 or decremented if the direction flag = 1. Increments or decrements will be either by 1 for byte operations or 2 for word operations. This instruction may be preceded with the REP prefix, which is described later in this table.
MOVSB	**(no operands)**	**Move byte string.**
		This instruction is the same as the MOVS instruction except that no operand is required because the instruction implies a byte operation and the "DS:" segment override is assumed for the source operand. The byte at DS:[SI] is moved to ES:[DI].
MOVSW	**(no operands)**	**Move word string.**
		This instruction is the same as the MOVS instruction except that no operand is required because the instruction implies a word operation and the "DS:" segment override is assumed for the source operand. The word at DS:[SI] is moved to ES:[DI].

Table A-2. Instruction Set Summary (Cont'd)

MUL	**source**	**Unsigned multiplication.**
	register	The unsigned values in AL or AX are multiplied memory by the source operand byte or word, respectively. When AL is multiplied by the source operand byte, the result is stored in AX and the carry and overflow flags are set unless AH = 0, in which case they are reset. When AX is multiplied by the source operand word, the result is stored in DX:AX and the carry and overflow flags are set unless DX = 0, in which case they are reset.
NEG	**destination**	**Two's complement negation.**
	register memory	This instruction replaces the destination memory operand with its two's complement. The carry is set unless the value of the destination operand is zero, in which case it is reset.
NOP	**(no operand)**	**No operation.**
		This one byte instruction (opcode 90H) performs no operation.
NOT	**destination**	**One's complement negation.**
	register memory	This instruction replaces the destination operand with its one's complement (logical NOT).

Table A-2. Instruction Set Summary (Cont'd)

OR	**destination,source**	**Logical inclusive OR.**
	register,register register,memory memory,register register,immediate memory,immediate accumulator,immediate	This instruction performs a logical OR operation on the operands. The destination will contain the result of the OR operation.
OUT	**port,accumulator**	**Output to port.**
	immed8,accumulator DX,accumulator	This instruction will cause a byte (accumulator = AL) or word (accumulator = AX) to be output from the port whose address is specified by an 8 bit immediate value or is in register DX.
OUTS (80186/80286)	**DX,source-string**	**Output string to port.**
	DX,source-string	This 80186/80286 instruction transfers data from the source string [SI] to the port whose address is specified by the contents of register DX. The type operation (BYTE or WORD) is specified in the source-string operand. This instruction may be preceded by the REP prefix which is described later in this table.
OUTSB (80186/80286)	**(no operands)**	**Output byte string to port.**
		This instruction is the same as the OUTS instruction except that no operands are required because a byte operation is implied in the instruction mnemonic and the segment override "DS:" is assumed for the source operand. The byte at DS:[SI] will be output to the port whose address is in DX.

Table A-2. Instruction Set Summary (Cont'd)

OUTSW (80186/80286)	**(no operands)**	**Output word string to port.**
		This instruction is the same as the OUTS instruction except that no operands are required because a word operation is implied in the instruction mnemonic and the segment override "DS:" is assumed for the source operand. The word at DS:[SI] will be output to the port whose address is in DX.
POP	**destination**	**Pop word off stack.**
	reg16 seg-reg(CS illegal) mem16	This instruction will pop the word at the top of the stack (SS:SP) and place it in the destination. SP is incremented by 2. In the 80286 Protected Mode, if the destination is a segment register, the word at the top of the stack must be a selector.
POPA (80186/80286)	**(no operands)**	**Pop all general registers.**
		This 80186/80286 instruction will pop the eight general purpose registers in the following order: DI, SI, BP, SP, BX, DX, CX, AX. The SP value popped is discarded.
POPF	**(no operands)**	**Pop into flags register.**
		This instruction will pop the word from the top of the stack (SS:SP) into the flags register. In the 80286, the I/O privilege level will only be altered if executing at PL = 0. The interrupt enable flag will be altered only when executing at a privilege level equal to the I/O privilege level or higher.

Table A-2. Instruction Set Summary (Cont'd)

PUSH	**source**	**Push word onto stack.**
	reg16 seg-reg mem16 immediate (80286)	This instruction will push the source operand word onto the top of the stack (SS:SP). SP is decremented by 2. Immediate source operands are allowed for the 80286, and PUSH SP will push the value of SP before the instruction. (In the 8086, PUSH SP will push the value of SP after the instruction.
PUSHA (80186/80286)	**(no operands)**	**Push all general registers.**
		This 80186/80286 instruction will push the eight general purpose registers in the following order: AX, CX, DX, BX, original SP, BP, SI, DI. The SP value pushed is the SP value before the PUSHA instruction.
PUSHF	**(no operands)**	**Push the flags register.**
		This instruction will push the flag register onto the stack (SS:SP). SP is decremented by 2.
RCL	**destination,count**	**Rotate left through carry.**
	register,1 register,CL memory,1 memory,CL	This instruction will rotate the destination operand left, though the carry flag, the number of times specified by the count operand (either 1 or the contents of CL). The carry flag is rotated into the destination operand's low order bit.

Table A-2. Instruction Set Summary (Cont'd)

RCR	**destination,count**	**Rotate right through carry.**
	register,1 register,CL memory,1 memory,CL	This instruction will rotate the destination operand right, though the carry flag, the number of times specified by the count operand (either 1 or the contents of CL). The carry flag is rotated into the destination operand's high order bit.
REP		**Repeat string instructions.**
		The REP instruction prefix will cause the string instruction to be repeated until CX = 0.Register CX is decremented after every execution of the string instruction. The REP prefix may be used with the following string instruction families: INS, MOVS, OUTS, and STOS.
REPE/REPZ		
		The REPE/REPZ instruction prefix will cause the string instruction to be repeated until CX = 0 or until the zero flag = 1. Register CX is decremented after every execution of the string instruction. The REPE/REPZ prefix may be used with the following string instruction families: CMPS and SCAS.
REPNE/REPNZ		
		The REPNE/REPNZ instruction prefix will cause the string instruction to be repeated until CX = 0 or until the zero flag = 0. Register CX is decremented after every execution of the string instruction. The REPNE/REPNZ prefix may be used with the following string instruction families:CMPS and SCAS.

Table A-2. Instruction Set Summary (Cont'd)

RET	**pop-value (optional)**	**Return from procedure.**
	pop-value	This instruction transfers control back to a return address which was pushed onto the stack at the time of the procedure call. The optional pop-value allows you to release additional bytes from the stack; the pop-value is added to SP. In 80286 Protected Mode, intersegment returns are made through the return selector to a code segment of equal or less privilege.
ROL	**destination,count**	**Rotate left.**
	register,1 register,CL memory,1 memory,CL	This instruction will rotate the destination operand left the number of times specified by the count operand (either 1 or the contents of CL). The destination operand's high order bit is rotated into the low order bit.
ROR	**destination,count**	**Rotate right.**
	register,1 register,CL memory,1 memory,CL	This instruction will rotate the destination operand right the number of times specified by the count operand (either 1 or the contents of CL). The destination operand's low order bit is rotated into the high order bit.
SAHF	**(no operands)**	**Store AH into the flag register.**
		This instruction will store register AH into the low byte of the flag word.

Table A-2. Instruction Set Summary (Cont'd)

SAL/SHL	**destination,count**	**Shift arithmetic/logical left.**
	register,1 register,CL memory,1 memory,CL	This instruction will shift the destination operand left the number of times specified by the count operand (either 1 or the contents of CL). Zeroes are shifted into the low order bit. The overflow flag is cleared if the sign bit is the same at the end of the operation.
SAR	**destination,count**	**Shift arithmetic right.**
	register,1 register,CL memory,1 memory,CL	This instruction will shift the destination operand right the number of times specified by the count operand (either 1 or the contents of CL). Shifted into the high order bit are bits equal to the original high order bit, so that the sign of the operand is preserved.
SBB	**destination**	**Source Integer subtraction with borrow.**
	register,register register,memory memory,register register,immediate memory,immediate accumulator,immediate	This instruction subtracts the source operand and the carry flag from the destination operand. The destination operand will contain the result of the operation. Byte sized immediate values are sign-extended before subtraction.

Table A-2. Instruction Set Summary (Cont'd)

SCAS	**dest-string**	**Scan string operand.**
	dest-string	This instruction subtracts the byte or word destination string (ES:[DI]) from AL or AX, respectively. The flags are set as a result of the subtraction, but the result is discarded. Register DI is incremented if the direction flag = 0 or decremented if the direction flag = 1. Increments or decrements will be either by 1 for byte operations or 2 for word operations. This instruction may be preceded with the REPE or REPNE prefixes, which are described under the REP entry in this table.
SCASB	**(no operands)**	**Scan byte string operand.**
		This instruction is the same as the SCAS instruction except that no operand is required because the instruction implies a byte operation and the ES:[DI] destination location is assumed, as it was for the SCAS instruction.
SCASW	**(no operands)**	**Scan word string operand.**
		This instruction is the same as the SCAS instruction except that no operand is required because the instruction implies a word operation and the ES:[DI] destination location is assumed, as it was for the SCAS instruction.

Table A-2. Instruction Set Summary (Cont'd)

SGDT/SIDT (80286)	**memory**	**Store GDT/IDT Register.**
	memory	This 80286, Privilege Level = 0 instruction will store 6 bytes into memory from the Global or Interrupt Descriptor Table register. The first memory word will be the LIMIT of the GDT or IDT register, the next three bytes are the BASE, and the last byte is undefined.
SLDT (80286)	**destination**	**Store LDT Register.**
	mem16 reg16	This 80286, Protected Mode, Privilege Level = 0 instruction stores the LDT register, which contains a selector pointing to an LDT descriptor in the Global Descriptor Table, into a word length register or memory location.
SMSW (80286)	**destination**	**Store Machine Status Word.**
	mem16 reg16	This 80286 instruction will store the Machine Status Word into a word length register or memory location.
STC	**(no operands)**	**Set carry flag.**
		This instruction sets the carry flag to 1.
STD	**(no operands)**	**Set the direction flag.**
		This instruction sets the direction flag to 1. When the direction flag is 1, string instructions will cause the contents of the index register(s) to be decremented.

Table A-2. Instruction Set Summary (Cont'd)

STI	**(no operands)**	**Set the interrupt enable flag.**
		This instruction sets the interrupt enable flag to 1 (if the current privilege level is at least as privileged as the IOPL, in 80286 Protected Mode). External, maskable interrupts are enabled after the executing the next instruction.
STOS	**dest-string**	**Store string operand.**
	dest-string	This instruction loads AL or AX into the byte or word at location ES:[SI]. The dest-string operand will specify whether the operation is of type BYTE or WORD. After the load, DI is incremented if the direction flag = 0 or decremented if the direction flag = 1. Increments or decrements will be either by 1 for byte operations or 2 for word operations. This instruction may be preceded with the REP prefix, which is described later in this table.
STOSB	**(no operands)**	**Store byte string operand.**
		This instruction is the same as the STOS instruction except that no operand is required because the instruction implies a byte operation and the "ES:" segment override is assumed. The byte in AL is stored into ES:[DI].
STOSW	**(no operands)**	**Store word string operand.**
		This instruction is the same as the STOS instruction except that no operand is required because the instruction implies a word operation and the "ES:" segment override is assumed. The word in AX is stored into ES:[DI].
STR	**destination**	**Store Task Register. (80286)**

Table A-2. Instruction Set Summary (Cont'd)

	mem16 reg16	This 80286, Protected Mode instruction stores the Task Register into a word length register or memory location.
SUB	**destination,source**	**Subtraction.**
	register,register register,memory memory,register register,immediate memory,immediate accumulator,immediate	This instruction subtracts the source operand from the destination operand.memory,register The destination operand will contain the result of the operation. Byte-sized immediate values are sign-extended before subtraction if destination is word sized.
TEST	**destination,source**	**Logical compare.**
	register,register register,memory memory,register register,immediate memory,immediate accumulator,immediate	This instruction performs a logical AND on the two operands and sets the flags accordingly. The result is discarded.
VERR (80286)	**selector**	**Verify a segment for reading**
	mem16 reg16	This 80286, Protected Mode instruction tests if the segment to which the selector points is readable from the current privilege level. The zero flag is set to 1 if the segment is readable: to 0 if it is not.

Table A-2. Instruction Set Summary (Cont'd)

VERW (80286)	**selector**	**Verify a segment for writing.**
	mem16 reg16	This 80286, Protected Mode instruction tests if the segment to which the selector points is writable from the current privilege level. The zero flag is set to 1 if the segment is writable: to 0 if it is not.
WAIT	**(no operands)**	**Wait for signal level.**
		This instruction causes the 8086/80188 processor to enter the wait state until the TEST pin is active. In the 80286, the WAIT instruction suspends execution until the BUSY pin, driven by the 80287 numeric processor, is inactive (high).
XCHG	**destination,source**	**Exchange memory/register with register.**
	accumulator,reg16 reg16,accumulator memory,register register,memory register,register	This instruction exchanges the source and destination operands. In the 80286, the BUS LOCK signal is asserted during the exchange regardless of a LOCK prefix or IOPL.
XLAT	**source-table**	**Table look-up translation.**
	source-table	Before this instruction is executed, AL should contain the unsigned index to the table at DS:[BX]. This instruction will move the byte at location DS:[BX + AL] to AL.

Table Convention In Table A-3, information is laid out in the following way:

Operand	
	Forms this operand may take and possibly further explanation of those forms.

Table A-3. Operand Forms

register	
	Includes forms of "reg16" and "reg8" operands below.
reg8	
	AH, AL, BH, BL, CH, CL, DH, DL.
reg16	
	AX, BX, CX, DX, SP, BP, SI, DI.
seg-reg	
	CS, DS, SS, ES.
accumulator	
	AX or AL.
immediate	
	# 0 thru # 0FFFFH SEG < LABEL> OFFSET < LABEL> also "immed8" operands below.
immed8	
	# 0 thru # 0FFH HIGH < LABEL> LOW < LABEL> SIZE < LABEL> TYPE < LABEL>

Table A-3. Operand Forms (Cont'd)

memory	
	The form of a memory operand will depend on the addressing mode. (Segment overrides, CS:, DS:,ES:, and SS: allowed in all modes.)
ADDRESSING MODES:	
Direct	< LABEL>
Register Indirect	[BX], [BP], [SI], [DI]
Based	< LABEL> [BX] and < LABEL> [BP]
Indexed	< LABEL> [SI] and < LABEL> [DI]
Based Indexed	< LABEL> [BX][SI] < LABEL> [BX][DI] < LABEL> [BP][SI] < LABEL> [BP][DI]
NOTE: '] [' is the same as using "+ ". That is, [BX] [SI] is equivalent to [BX+ SI], etc.	
mem8	
	The form of this operand is the same as the "memory" operand except that the operand must be associated with the type BYTE. Either the label must be of type BYTE, or the BYTE PTR type override must precede the memory operand.
mem16	
	The form of this operand is the same as the "memory" operand except that the operand must be associated with the type WORD. Either the label must be of type WORD, or the WORD PTR type override must precede the memory operand.

Table A-3. Operand Forms (Cont'd)

source-table	
	This operand will be a label located at the beginning of a translation table. This operand must be of type BYTE. The assembler assumes that BX contains the address of the beginning of the translation table.
source-string	
	This operand will be a label. The assembler uses this operand to determine whether a string operation is a byte operation (in which case the label will be of type BYTE), or a word operation (in which case the label will be of type WORD). The assembler will assume that SI contains the label's offset address and that DS contains the segment address (or segment selector in 80286 Protected Mode) for the label's segment unless a segment override is used.
dest-string	
	This operand will be a label. The assembler uses this operand to determine whether a string operation is a byte operation (in which case the label will be of type BYTE), or a word operation (in which case label will be of type WORD). The assembler will assume that DI contains the label's offset address and that ES contains the segment address (or segment selector in 80286 Protected Mode) for the label's segment. The assembler will always assume that ES points to the destination string's segment. Segment overrides are not allowed.

Table A-3. Operand Forms (Cont'd)

short-label	
	This operand must reference a label within -128 or + 127 bytes from the next instruction pointer location.
near-label	
	This operand must reference a label in the current code segment.
far-label	
	This operand will reference a label in another code segment. Since the assembler will assume all jumps to be near, the type FAR must be associated with label, either in the instruction operand by preceding the label with FAR PTR, or the label's external declaration (with the EXT pseudo instruction). In the 80286 Protected Mode, the far-label operand may be used to jump to code segments, call gates, task gates, or Task State Segments (TSSs). Labels must be associated with the type FAR (either in the instruction operand or in the external (EXT) declaration).
near-proc	
	This operand must reference a label in the current code segment.

Table A-3. Operand Forms

far-proc	
	This operand will reference a label in another code segment. Since the assembler will assume all calls to be near, the type FAR must be associated with the label, either in the instruction operand by preceding the label with FAR PTR, or the label's external declaration (with the EXT pseudo instruction). In the 80286 Protected Mode, the far-label operand may be used to jump to code segments, call gates, task gates, or Task State Segments (TSSs). The labels must be associated with the type FAR (either in the instruction operand or in the external (EXT) declaration).
memptr16	
	This operand takes the same form as the "memory" operand. The memory location to which this operand points will contain the offset address to which program control will be transferred (in the current code segment).

Table A-3. Operand Forms (Cont'd)

mptr32	
	This operand takes the same form as the "memory" operand except that the operand must be associated with the type DWORD. The memory location to which this operand points will contain the segment:offset (selector:offset in the 80286 Protected Mode) address to which program control will be transferred. The offset portion of the target location is contained in the low address memory word, and segment (or selector) portion of the target location is contained in the high address memory word.
regptr16	
	This operand takes the same form as the "reg16" operand. The register will contain the offset address to which program control will be transferred (in the current code segment).

Notes

B

80286 Programming

Introduction

This appendix contains information on how to write 80286 protected mode programs. The 80186 assembler supports the 80286 microprocessor instruction set for the real address mode (8086 compatible). The 8086/8088 pseudo instructions and keyword operators also apply to the 80286 with exceptions noted below.

The processor directive that must appear in the first column of the first line of your assembly language program source files is "80286".

The additional instructions that make up the 80286 instruction set have been included in the 8086/8088 instruction set summary.

Using the "80286" directive allows your programs to contain the special 80286 pseudo instructions whose descriptions appear on the following pages.

Note

The HP 64853's "80286" assembler was designed as an early support tool for the 80286 microprocessor running in the Protected Virtual Address Mode. As such, there are some known limitations. (For instance, the "80286" assembler is not compatible with the HP 64228 - 80286 Emulator.) We encourage 80286 users to use the HP 64859 Cross Assembler/Linker instead. The HP 64859 product supports the 80286 in both the Real Address Mode and the Protected Virtual Address Mode.

The "SEG" Keyword Operator In 80286 Programs

The SEG pseudo instruction only works in the 8086 compatible mode. SEGE translates the 24 bit address into an 8086 type segment offset, and generates the segment. This is implemented so that the user can initialize data structures while the processor is still in the 8086 compatible mode, and the user is using the 80286 protected mode assembler. A physical address of UVWXYZ hex is translated to a 80286 logical address of V000WXYZ hex. Note the most significant nibble of address has no meaning to the 8086. The segment is always generated in this fashion. Since this assembler does not use the 80286 logical address of selector and offset, the SEG pseudo instruction will generate bad code if the microprocessor is in the protected mode. The user must load immediate numbers equivalent to the appropriate selector in order to initialize data structures while in the microprocessor protected mode.

80286 Pseudo Instructions

The pseudo instructions SEGMENT, ENDS, and STACKSEG allow the use of identifying labels. These segment labels are only local, however, to the module in which they are defined. Segment labels will not be usable with the instructions SEG_DES, TSS_DES, and LDT_DES if they are declared to be EXTERNAL.

Any descriptor tables desired must be created by the program. Table creation is not automatic. The instructions SEG_DES, TSS_DES, and LDT_DES create complete segment descriptor entries. Instructions CALL_GATE, TASK_GATE, INTR_GATE, and TRAP_GATE will create complete gate descriptor entries.

All selector references must be "immediate" values becasue only physical addresses are used by the assembler..

The following pseudo instructions are only applicable to 80286 assembly language programs.

The DD Pseudo Instruction in 80286 Programs

The DD pseudo instruction works identically for the 8086 and the 80286, except if the value field of the DD opcode is a relocatable label. In the 8086 the pseudo instruction would initialize memory with the 8086 logical address of the relocatable label. Since for the 80286, a 24 bit physical address is input, the 80286 assembler generates an 8086 logical address in the manner described for the SEG pseudo instruction, and initializes memory with this value. Since the 80286 assembler does not use the 80286 logical address of selector and offset, if you desire memory to be initialized, you must directly specify the immediate value of the selector and offset.

CALL_GATE
TASK_GATE
INTR_GATE
TRAP_GATE

Define Call Gate Descriptor
Define Task Gate Descriptor
Define Interrupt Gate Descriptor
Define Trap Gate Descriptor
(Special 80286 Pseudos)

Syntax

Label	Operation	Operand
CALL_GATE	DPL,SELECTOR	< ,OFFSET, < WORD_COUNT>
TASK_GATE	DPL,SELECTOR	< ,OFFSET >
INTR_GATE	DPL,SELECTOR	< ,OFFSET >
TRAP_GATE	DPL,SELECTOR	< ,OFFSET >

Description

Gate descriptors are used only for transfer of control from the instructions in one segment to the instructions in another segment. Gates provide some segment protection in that access to other level tasks must reference a gate. These pseudo instructions, therefore, allow the creation of gate descriptor data structures. They each require a data privilege level, i.e., DPL0,.., DPL3, and an immediate selector. The offset from the selector is optional, and for CALL_GATE, the word_count term is optional.

Example

For an example of this pseudo instruction and how it relates to the other 80286 pseudo instructions, see the 80286 example program at the end of this appendix.

JMP
CALL

Unconditional, Intersegment Jump
Unconditional, Intersegment Call
(Special 80286 Pseudos)

Syntax

Label	Operation	Operand
	JMP	SELECTOR< ,OFFSET>
	CALL	SELECTOR< ,OFFSET>

Description

Because the 80286 assembler does not use 80286 logical addresses (i.e., selector and offset), if the user attempts to do an intersegment JMP or CALL, incorrect code will be generated. This expansion of the JMP and CALL instructions allows the user to specify the immediate value of the SELECTOR of the segment to jump to, and optionally the OFFSET in the segment.

Example

For an example of this pseudo instruction and how it relates to the other 80286 pseudo instructions, see the 80286 example program at the end of this appendix.

SEGMENT

Create New Logical Segment
(Special 80286 Pseudo)

Syntax

Label	Operation	Operand
Name	SEGMENT	ATTR
Name	ENDS	

Description

This pseudo instruction creates a new logical segment, within the current 64K PROG, DATA, COMN, or ORG segment. These segments cannot be nested, and will all be created sequentially within the current segment. The total length of all the logical segments within PROG, DATA, COMN, or ORG segments must not exceed 64K. The SEGMENT pseudo is provided for the user, to have many separate logical segments within one file, with the provision that together they are less than 64K of memory. This SEGMENT pseudo instruction is to be utilized with the descriptor table building pseudo instructions. Each SEGMENT pseudo instruction must have a corresponding ENDS pseudo instruction.

The attributes (ATTR) assignable to a segment are four: Executable Only (EO), Executable and Readable (ER), Readable Only (RO), and Readable and Writable (RW).

Example

For an example of this pseudo instruction and how it relates to the other 80286 pseudo instructions, see the 80286 example program at the end of this appendix.

SEG_DES
TSS_DES
LDT_DES

Create Segment Descriptor
Create Task Segment Descriptor
Create Local Descriptor Table Descriptor
(Special 80286 Pseudos)

Syntax

Label	Operation	Operand
SEG_DES	SEG_NAME	< ,DPL,LENGTH>
TSS_DES	SEG_NAME	< ,DPL,LENGTH>
LDT_DES	SEG_NAME	< ,DPL,LENGTH>

Description

This pseudo instruction creates a descriptor data structure. It must be given an operand which was defined to be a SEGMENT. Optionally, the user can include the data privilege number i.e., DPL0 .. DPL3, and length of the segment in bytes.

Example

For an example of this pseudo instruction and how it relates to the other 80286 pseudo instructions, see the 80286 example program at the end of this appendix.

STACKSEG

Create Logical Stack Segment
(Special 80286 Pseudo)

Syntax

Label	Operation	Operand
Name	STACKSEG	LENGTH

Description

This pseudo instruction creates a logical stack segment with length LENGTH bytes. This pseudo does not require a SEGMENT or ENDS pseudo instruction.

Example

For an example of this pseudo instruction and how it relates to the other 80286 pseudo instructions, see the 80286 example program at the end of this appendix.

The 80286 Example Program

This example program initializes all system data segments required for a simple three task system and then schedules the tasks as follows: task 1 will start, followed by task 2, followed by task 1 in a repetitive fashion. If any internal exceptions or external interrupts are generated, task 3 will be invoked halting the processor. The system data segments will be temporarily setup in ROM and transferred to RAM using a string move.

80286 Example Program

```
"80286"
                GLOBAL      IDT,GDT,LDT1,LDT2

LIMIT_FF        EQU         00FFH
LIMIT_100       EQU         0100H
OFFSET_00       EQU         0000H
BASE_00         EQU         00H
REG_00          EQU         0000H
INTEL_RESV_B    EQU         00H
FLAG_WORD       EQU         00H
SEL_GDTA        EQU         0008H
SEL_LDT_1       EQU         0010H
SEL_LDT_1A      EQU         0018H
SEL_LDT_2       EQU         0020H
SEL_LDT_2A      EQU         0028H
SEL_LDT_3       EQU         0030H
SEL_LDT_3A      EQU         0038H
SEL_TSS_1       EQU         0040H
SEL_TSS_1A      EQU         0048H
SEL_TSS_2       EQU         0050H
SEL_TSS_2A      EQU         0058H
SEL_TSS_3       EQU         0060H
SEL_TSS_3A      EQU         0068H
ATTR_TG_DPL0    EQU         085H
```

```
ATTR_TG_DPL1        EQU         0A5H
ATTR_TG_DPL2        EQU         0C5H
ATTR_TG_DPL3        EQU         0E5H
ATTR_DS_DPL0        EQU         092H
ATTR_DS_DPL1        EQU         0B2H
ATTR_DS_DPL2        EQU         0D2H
ATTR_DS_DPL3        EQU         0F2H
ATTR_LDT            EQU         082H
ATTR_TS_DPL0        EQU         081H
ATTR_TS_DPL1        EQU         0A1H
ATTR_TS_DPL2        EQU         0C1H
ATTR_TS_DPL3        EQU         0E1H
ATTR_CS_DPL0        EQU         09AH
ATTR_CS_DPL1        EQU         0BAH
ATTR_CS_DPL2        EQU         0DAH
ATTR_CS_DPL3        EQU         0FAH
```

The following macro is used for defining a task state segment. The parameters passed are the virtual address of the stack pointer (SS_SEL and SP), the flag word (FLAGS), the code segment selector (CS_SEL), the instruction pointer (IP), the task ldt selector (LDT_SEL), the data segment selector (DS_SEL), and the extra data segment selector (ES_SEL). The stack pointer passed is used temporarily for all 4 stacks. all task state segments created will have a blank back link selector, and have all registers cleared except for the registers passed as parameters.

```
TSS_SEG     MACRO       &SP,&SS_SEL,&FLAGS,&CS_SEL,&IP,
                         &LDT_SEL,&DS_SEL,&ES_SEL
            DW          0000H
            REPT        3
            DW          &SP,&SS_SEL
            DW          &IP
            DW          &FLAGS
            DW          REG_00,REG_00,REG_00,REG_00
```

```
        DW          &SP

        DW          REG_00,REG_00,REG_00
        DW          &ES_SEL,&CS_SEL,&SS_SEL,&DS_SEL
        DW          &LDT_SEL
        MEND

        DATA
        ASSUME      DS:DATA
IDT     SEGMENT     RW
        TASK_GATE   DPL0,SEL_TSS_3              ;IDT
        TASK_GATE   DPL0,SEL_TSS_3
        TASK_GATE   DPL0,SEL_TSS_3
        TASK_GATE   DPL0,SEL_TSS_3
        TASK_GATE   DPL0,SEL_TSS_3
        TASK_GATE   DPL0,SEL_TSS_3
        TASK_GATE   DPL0,SEL_TSS_3
        TASK_GATE   DPL0,SEL_TSS_3
        TASK_GATE   DPL0,SEL_TSS_3
        TASK_GATE   DPL0,SEL_TSS_3
        TASK_GATE   DPL0,SEL_TSS_3
        TASK_GATE   DPL0,SEL_TSS_3
        TASK_GATE   DPL0,SEL_TSS_3
        TASK_GATE   DPL0,SEL_TSS_3
        TASK_GATE   DPL0,SEL_TSS_3
        TASK_GATE   DPL0,SEL_TSS_3
        TASK_GATE   DPL0,SEL_TSS_3
        TASK_GATE   DPL0,SEL_TSS_3
        TASK_GATE   DPL0,SEL_TSS_3
        TASK_GATE   DPL0,SEL_TSS_3

        TASK_GATE   DPL0,SEL_TSS_3
                    DPL0,SEL_TSS_3
        TASK_GATE   DPL0,SEL_TSS_3
        TASK_GATE   DPL0,SEL_TSS_3
        TASK_GATE   DPL0,SEL_TSS_3
        TASK_GATE   DPL0,SEL_TSS_3
        TASK_GATE   DPL0,SEL_TSS_3
```

```
        TASK_GATE   DPL0,SEL_TSS_3
        TASK_GATE   DPL0,SEL_TSS_3
        TASK_GATE   DPL0,SEL_TSS_3
        TASK_GATE   DPL0,SEL_TSS_3
        TASK_GATE   DPL0,SEL_TSS_3
        TASK_GATE   DPL0,SEL_TSS_3
IDT     ENDS
GDT     SEGMENT     RW
        SEG_DES      GDT                    ;GDT. FIRST DES IS
        SEG_DES     GDT                     ;FOR THE NULL SELECTOR
        LDT_DES     LDT1
        SEG_DES     LDT1
        LDT_DES     LDT2
        SEG_DES     LDT2
        LDT_DES     LDT3
        SEG_DES     LDT3
        TSS_DES     TSS1
        SEG_DES     TSS1
        TSS_DES     TSS2
        SEG_DES     TSS2
        TSS_DES     TSS3
        SEG_DES     TSS3
        REPT        18
        DD          00000000H,00000000H
GDT     ENDS
LDT1    SEGMENT     RW
        SEG_DES      TASK1,DPL0,LIMIT_100   ;LDT1
        SEG_DES     STACK
        REPT        30
        DD          00000000H,00000000H
LDT1    ENDS
LDT2    SEGMENT     RW
        SEG_DES     TASK2,DPL0,LIMIT_100   ;LDT2
        SEG_DES     STACK
        REPT        30
        DD          00000000H,00000000H
LDT2    ENDS
LDT3    SEGMENT     RW
```

```
            SEG_DES     TASK3,DPL0,LIMIT_100   ;LDT3
            SEG_DES     STACK
            REPT        30
            DD          00000000H,00000000H
LDT3        ENDS
TSS1        SEGMENT     RW
            TSS_SEG     00FEH,000CH,FLAG_WORD,0004H,OFFSET_00,
                        SEL_LDT_1,000CH,000CH          ;TSS1

TSS1        ENDS
TSS2        SEGMENT     RW
            TSS_SEG     00FEH,000CH,FLAG_WORD,0004H,
                        OFFSET_00,SEL_LDT_2,000CH,000CH
                                                       ;TSS2
TSS2        ENDS
TSS3        SEGMENT     RW
            TSS_SEG     00FEH,000CH,FLAG_WORD,0004H,
                        OFFSET_00,SEL_LDT_3,000CH,000CH
                                                       ;TSS3
TSS3        ENDS

IDT_LIMIT_BASE          DBS         6
GDT_LIMIT_BASE          DBS         6
            ORG         00000700H
            ALIGN
STACK       STACKSEG    254
STACK       SEGMENT     RW
            DWS         127
STACK_TOP   DW          0000H
STACK       ENDS
            PROG
            ASSUME      CS:PROG

INITIALIZE
            MOV         AX,SEG STACK_TOP
            MOV         SS,AX
            MOV         SP,OFFSET STACK_TOP


LD_IDT
```

```
                MOV             AX,SEG IDT
                MOV             DS,AX
                MOV             WORD PTR IDT_LIMIT_BASE,# LIMIT_FF
                MOV             WORD PTR IDT_LIMIT_BASE[2],OFFSET IDT
                MOV             BYTE PTR IDT_LIMIT_BASE[4],# BASE_00
                MOV             BYTE PTRIDT_LIMIT_BASE[5],# INTEL_RESV_B


                LIDT            WORD PTR IDT_LIMIT_BASE

LD_GDT
                MOV             WORD PTR GDT_LIMIT_BASE,# LIMIT_FF
                MOV             WORD PTR GDT_LIMIT_BASE[2],OFFSET GDT
                MOV             BYTE PTR GDT_LIMIT_BASE[4],# BASE_00
                MOV             BYTE PTR GDT_LIMIT_BASE[5],# INTEL_RESV_B
                LGDT            WORD PTR GDT_LIMIT_BASE
SET_P_MODE
                MOV             AX,# 00000101B          ;SET PE AND EM BITS IN MSW
                LMSW            AX
                JMP             LD_TR                   ;DUMMY JUMP TO FLUSH QUEUE
LD_TR
                MOV             AX,# SEL_TSS_3
                LTR             AX
START_TASK1
```

Opcode for a jump using direct virtual address dword (VADW NAMES TSS1 SELECTOR)

```
                JMP             SEL_TSS_1               ;THIS REPRESENTS
                                                        ;THE SELECTOR
```

```
                                TASK 1 jumps to TASK2

                ASSUME          CS:ORG
                ORG             0000FF000H

                                Fix TSS3 IP,SP,ES,CS,SS,DS within TASK1. TSS3 was faulty
                                when TASK1 was first invoked.

TASK1           SEGMENT         ER
                MOV             AX,SEL_TSS_3A
                MOV             DS,AX
                MOV             WORD PTR [14],# 0000H
                MOV             WORD PTR [26],# 00FEH
                MOV             WORD PTR [34],# 000CH
                MOV             WORD PTR [36],# 0004H
                MOV             WORD PTR [38],# 000CH
                MOV             WORD PTR [40],# 000CH

                                Opcode for a jump using direct virtual address dword (VADW
                                NAMES TSS2 SELECTOR)

TASK1_LOOP      JMP             SEL_TSS_2
                JMP             TASK1_LOOP
TASK1           ENDS

                                Jumps back to TASK 1.

                ASSUME          CS:ORG
                ORG             0000FF100H
```

```
                                    Opcode for a jump using direct virtual address dword (VADW
                                    NAMES TSS1 SELECTOR)
```

```
TASK2           SEGMENT             ER
TASK2_LOOP      JMP                 SEL_TSS_1
                                    JMP
                                    TASK2_LOOP
TASK2           ENDS
```

```
                                    TASK3 halts the processor.
```

```
                ASSUME              CS:ORG
                ORG                 0000FF200H
TASK3           SEGMENT             ER
                HLT
                JMP                 TASK3
TASK3           ENDS
```

Set up the restart vector to jump to INITIALIZE.

```
ASSUME    CS:ORG
ORG       0000FFFF0H
JMP       FAR PTR INITIALIZE
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
```

Notes

C

70108/70116 Programming And Instruction Set Summary

This appendix contains general information. Architecture, operands, and condition flags are briefly discussed. For detailed descriptions of the microprocessors, refer to the manufacturers users manual.

Programming Considerations

Sixteen-bit operands may be assigned to even or odd address locations. For data and address operands, the least significant byte of the word will be stored in the lower-valued address. The most significant byte will be stored in the next higher address. The 70116 microprocessor automatically performs the required number of memory accesses: one if the word operand begins on an even byte address, and two if it begins on an odd byte address. The 70108 always performs two memory accesses for each 16-bit operand.

Note

See the "EXT" pseudo op in the chapter titled "Pseudo Instruction Summary" about "EXT" conflicts with NEC processors.

Modes Of Operation

Two modes of operation are possible with the microprocessors: Native Mode and 8080 Mode or emulation mode. The processor in Native Mode executes 8086/8088 compatible instructions, while in 8080 Mode the 8080 set of instructions is emulated. The mode flag of the program status word is set (1) for native mode execution, and cleared (0) for 8080 mode.

Two instructions BRKEM (Break for Emulation) and RETEM (Return from Emulation) control entry into and out of emulation mode from native mode.

Two instructions CALLN (Call native routine) and RETI (Return from Interrupt) will switch operation from 8080 mode to native mode and back to 8080 mode.

As the assembler directive, use "70108" or "70116" for native mode, "70108_80" or "70116_80" for 8080 mode.

Addressing Capabilities

In general, memory operands may be addressed directly, using a16-bit offset, or indirectly, using a base and/or index register added to an optional 8- or 16-bit displacement value.

Instruction Set Summary

All mnemonic instructions are summarized in table C-1. The instruction set is arranged in alphabetical order. Refer to the manufacturer's users guide for more detailed information.

Figure C-2 shows the typical machine instruction format. The location of an operand in a register or memory will be specified by up to three files in each instruction format. These fields are the mode field (mod), the register field (reg), and the register/memory field (r/m). When used, they occupy the second byte of the instruction format. The mode field occupies the two most significant bits of the byte and specifies how the r/m field will be used in locating the operand. The reg field occupies the next three bits following the mode field and specifies either an 8-bit register or a 16-bit register where an operand will be located.

Note, bytes three through six of an instruction are optional fields that usually contain the displacement (DISP) value of a memory operand and/or the actual value of an immediate constant operand. The effective address (EA) of the memory

operand will be computed according to the mode and r/m fields as follows:

Register operands may be indicated within the instruction format by the reg field which will represent the selected register. Operands may be indicated by an encoded field, in which case EA will represent the register selected by the r/m field. Instructions without a "W" bit in their format always refer to 16-bit registers; those with a "w" bit in their format refer to either 8- or 16-bit registers according to the following reg field assignments:

	16-bit (W= 1)	8-bit (W= 0)
reg field:	000 = reg AW	000 = reg AL
	001 = reg CW	001 = reg CL
	010 = reg DW	010 = reg DL
	011 = reg BW	011 = reg BL
	100 = reg SP	100 = reg AH
	101 = reg BP	101 = reg CH
	110 = reg IX	110 = reg DH
	111 = reg IY	111 = reg BH

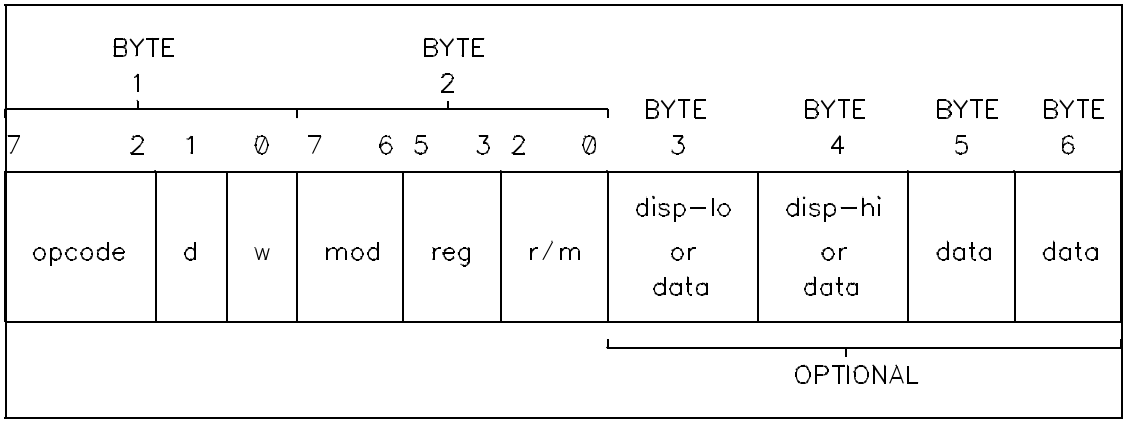

Figure C-1. Typical Instruction Format

d:	direction to (1) or from (0) register
w:	byte (0) or word (1) operation
mod and r/m:	addressing mode -register of memory
reg:	register select

The SEGMENT OVERRIDE PREFIX takes the form of: 001reg110 in which the register is assigned in the following manner:

reg	Segment register
00	DS1
01	PS
10	SS
11	DS0

70116/70108 Register Names

The following are reserved symbols. They have special meaning to the assembler and cannot appear as user-defined symbols. The 8086/8088 register counterparts are shown in parentheses.

SYMBOL	(8086/8088)	DESCRIPTION
AH	(AH)	High-order byte of register A
AL	(AL)	Low-order byte of register A
AW	(AX)	16-bit register A
BH	(BH)	High-order byte register B
BL	BL)	Low-order byte register B
BP	(BP)	Base pointer
BW	(BX)	16-bit register B
CH	(CH)	High-order byte register C
CL	(CL)	Low-order byte register C
CW	(CX)	16-bit register C
DH	(DH)	High-order byte register D
DL	(DL)	Low-order byte register D
DS0	(DS)	Data segment 0 register
DS1	(ES)	Data segment 1 register
DW	(DX)	16-bit register D
IX	(SI)	Source index register
IY	(DS)	Destination index register
PC	(IP)	Program counter
PS	(CS)	Program segment register
SP	(SP)	Stack pointer
SS	(SS)	Stack segment register

Instruction Set Symbols

Symbols used in table C-1, Instruction Set Summary, are as follows:

SYMBOL	DESCRIPTION
addr	Address (16 bits)
addr-hi	Most significant byte of address
addr-lo	Least significant byte of address
d	One-bit field identifying direction to (1) or from (0) register
data	Immediate operand (8- or 16-bit)
disp	8- or 16-bit displacement from end of current instruction
disp-hi	Most significant byte of 16-bit displacement
disp-lo	Least significant byte of 16-bit displacement
imm	3, 4 or 8-bit immediate operand
mod	Two-bit field defining addressing mode
offset-hi	Most significant byte in 16-bit offset destination address of target instruction
offset-lo	Least significant byte in 16-bit offset destination address of target instruction
reg	Field that defines the register used
r/m	Three-bit field, in conjunction with the mod and "W" fields defines EA
seg	Segment register
seg-hi	Most significant byte in 16-bit segment destination address of target instruction
seg-lo	Least significant byte in 16-bit segment destination address of target instruction
port	Number of I/O port
s:w	Sign-extended byte indicator
v	Interrupt: defines variable type (v= 1), or type 3 (v= 0) Shift or Rotate: variable number of bits to shift or rotate (v= 1), or one bit (v= 0)
w	One-bit field identifying byte (0) or word (1) instruction
z	Instruction being repeated terminates when zero flag is equal to z

Table C-1. 70116/70108 Instruction Set Summary

Mnemonic	Byte 1	Byte 2	Byte 3	Byte 4
ADD Add				
Memory or Register Operand with Register Operand	000000dw	mod reg r/m		
Immediate Operand to Memory or Register Operand	100000sw	mod 000 r/m	data	data if s:w= 01
Immediate Operand to Accumulator	0000010w	data	data if w= 1	
ADDC Add with Carry				
Memory or Register Operand with Register Operand	000100dw	mod reg r/m		
Immediate Operand to Memory or Register Operand	100000sw	mod 010 r/m	data	data if s:w= 01
Immediate Operand to Accumulator	0001010w	data	data if w= 1	
ADD4S Add Nibble String	00001111	00100000		
ADJBA Adjust Byte Add	00110111			
ADJBS Adjust Byte Subtract	00111111			
ADJ4A Adjust Nibble Add	00100111			

Table C-1. 701116/70108 Instruction Set Summary (Cont'd)

Mnemonic	Byte 1	Byte 2	Byte 3	Byte 4
ADJ4S Adjust Nibble Subtract	00101111			
AND And Logically				
Memory or Register Operand with Register Operand	001000dw	mod reg r/m		
Immediate Operand to Memory or Register Operand	1000000w	mod 100 r/m	data	data if w= 1
Immediate Operand to Accumulator	0010010w	data	data if w= 1	
BC/BL Branch if Carry/ Lower	01110010	disp		
BCWZ Branch if CW Equals Zero	11100011	disp		
BE/BZ Branch if Equal/ Zero	01110100	disp		
BGE Branch if Greater Than or Equal	01111101	disp		
BGT Branch if Greater Than	01111111	disp		
BH Branch if Higher	01110111	disp		
BLE Branch if Less Than or Equal	01111110	disp		
BLT Branch if Less Than	01111100	disp		

Table C-1. 70116/70108 Instruction Set Summary (Cont'd)

Mnemonic	Byte 1	Byte 2	Byte 3	Byte 4
BN Branch if Negative	01111000	disp		
BNC/BNL Branch if Not Carry/Not Lower	01110011	disp		
BNE/BNZ Branch if Not Equal/Not Zero	01110101	disp		
BNH Branch if Not Higher	01110110	disp		
BNV Branch if Not Overflow	01110001	disp		
BP Branch if Positive	01111001	disp		
BPE Branch if Parity Even	01111010	disp		
BPO Branch if Parity Odd	01111011	disp		
BR Branch				
Intrasegment or Intragroup Direct	11101001	disp-lo	disp-hi	
Intrasegment Direct Short	11101011	disp-lo		
Intrasegment or Intragroup Indirect	11111111	mod 100 r/m		
Intersegment Direct	11101010	offset-lo	offset-hi	seg-lo Byte 5= seg-hi

Table C-1. Instruction Set Summary (Cont'd)

Mnemonic	Byte 1	Byte 2	Byte 3	Byte 4
Intersegment Indirect	11111111	mod 101 m	(mod 11)	
BRK Break	1100110v	imm (if v= 1)		
BRKEM Break for Emulation	00001111	11111111	imm	
BRKV Break if Overflow	11001110			
BUSLOCK Bus Lock Prefix	11110000			
BV Branch if Overflow	01110000	disp		
CALL Call				
Direct Intrasegment or Intragroup	11101000	disp-lo	disp-hi	
Indirect Intraseg-ment or Intragroup	11111111	mod 010 r/m		
Direct Intersegment	10011010	offset-lo	offset-hi	seg-lo Byte 5= seg-hi
Indirect Inter-segment	11111111	mod 011 m	(mod11)	
CALLN Call Native	11101101	11101101	imm (effective for 8080 mode also)	
CHKIND Check Index	01100010	mod reg m		

Table C-1. Instruction Set Summary (Cont'd)

Mnemonic	Byte 1	Byte 2	Byte 3	Byte 4
CLR1 Clear Bit				
Bit CL of Memory or Register Operand	00001111	0001001w	mod 000 r/m	
Bit imm of Memory or Register Operand	00001111	0001101w	mod 000 r/m	
Carry Flag	11111000			
Direction Flag	11111100			
CMP Compare Operands				
Memory or Register Operand with Register Operand	0011101w	mod reg r/m		
Register Operand with Memory	0011100w	mod reg m		
Immediate Operand with Memory or Register Operand	100000sw	mod 111 r/m	data	data if s:w= 01
Immediate Operand with Accumulator	0011110w	data	data if w= 1	
COMPBK/CMPBKB/CMPBKW Compare Block/ Compare Block Byte/ Compare Block Word	1010011w			

Table C-1. Instruction Set Summary (Cont'd)

Mnemonic	Byte1	Byte 2	Byte 3	Byte 4
COPM/CMPMB/CMPMW Compare Multiple/ Compare Multiple Byte/ Compare Multiple Word	1010111w			
CMP4S Compare Nibble String	00001111	00100110		
CVTBD Convert Binary to Decimal	11010100	00001010		
CVTBW Convert byte to Word	10011000			
CVTDB Convert Decimal to Binary	11010101	00001010		
CVTWL Convert Decimal to Long Word	10011001			
DBNZ Decrement and Branch if Not Zero	11100010	disp		
DBNZE Decrement and Branch if Not Zero and Equal	11100001	disp		
DBNZNE Decrement and Branch if Not Zero and Not Equal	11100000	disp		

Table C-1. Instruction Set Summary (Cont'd)

Mnemonic	Byte1	Byte 2	Byte 3	Byte 4
DEC Decrement Operand by One				
Memory or Register Operand	1111111w	mod 001 r/m		
Word Register Operand	01001 reg			
DI Disable Interrupt	1111010			
DISPOSE Dispose a Stack Frame	11001001			
DIV Divide Signed	1111011w	mod 111 r/m		
DIVU Divide Unsigned	1111011w	mod 110 r/m		
EI Enable Interrupt	11111011			
EXT Extract Bit Field				
Register	00001111	00110011	11 reg reg	
Immediate	00001111	00111011	11000 reg	imm
FPO1 Floating Point Operation 1				
Register	11011xxx	11yyyzzz		
Memory	11011xxx	mod yyy m		
FPO2 Floating Point Operation 2				
Register	0110011x	11yyyzzz		
Memory	0110011x	mod yyy m		
HALT Halt	11110100			

Table C-1. Instruction Set Summary (Cont'd)

Mnemonic	Byte1	Byte 2	Byte 3	Byte 4
IN Input Byte and Input Word from Fixed Port	1110010w	port		
Variable Port	1110110w			
INC Increment Operand by One Memory or Register Operand	1111111w	mod 000 r/m		
Register Operand (Word)	01000reg			
INM Input Multiple	0110110w			
INS Insert Bit Field Register	00001111	00110001	11 reg reg	
Immediate	00001111	00111001	11000 reg	imm
LDEA Load Effective Address to Register	10001101	mod reg m		
LDM/LDMB/LDMW Load Multiple/ Load Multiple Byte/ Load Multiple Word	1010110w			

Table C-1. Instruction Set Summary (Cont'd)

Mnemonic	Byte1	Byte 2	Byte 3	Byte 4
MOV Move				
Memory or Register Operand to Register Operand	1000101w	mod reg r/m		
Register Operand to Memory	1000100w	mod reg m		
Immediate Operand to Memory Operand	1100011w	mod 000 m	data	data if w= 1
Immediate Operand to Register	1011wreg	data	data if w= 1	
Memory Operand to Accumulator	1010000w	addr-lo	addr-hi	
Accumulator to Memory Operand	1010001w	addr-lo	addr-hi	
Memory or Register Operand to Segment Register	10001110	mod 0seg r/m		
Segment Register to Memory or Register Operand	10001100	mod 0seg r/m		
32 Bit Memory to Data Segment 0 Register	11000101	mod reg m		

Table C-1. Instruction Set Summary (Cont'd)

Mnemonic	Byte 1	Byte 2	Byte 3	Byte 4
32 Bit Memory to Data Segment 1 Register	11000100	mod reg m		
PSW to AH	10011111			
AH to PSW	10011110			
MOVBK/MOVBKB/MOBBKW Move block/Move Block Byte/Move Block Word	1010010w			
MUL Multiply Signed				
Multiply Accumulator by Register or Memory	1111011w	mod 101 r/m		
Immediate	011010s1	mod reg r/m	data	data if s= 0
MULU Multiply Unsigned				
Accumulator by Register or Memory	1111011w	mod 100 r/m		
NEG Negate, or Form 2's Complement	1111011w	mod 011 r/m		
NOP No Operation	10010000			
NOT, or Form 1's Complement	1111011w	mod 010 r/m		

Table C-1. Instruction Set Summary (Cont'd)

Mnemonic	Byte 1	Byte 2	Byte 3	Byte 4
NOT1 Not Bit				
Bit CL of Memory or Register Operand	00001111	0001011w	mod 000 r/m	
Bit imm of Memory or Register Operand	00001111	0001111w	mod 000 r/m	imm
Carry Flag	11110101			
OR Inclusive OR				
Memory or Register Operand with Register Operand	000010dw	mod reg r/m		
Immediate Operand to Memory or Register Operand	1000000w	mod 001 r/m	data	data if w= 1
Immediate Operand to Accumulator	0000110w	data	data if w= 1	
OUT Output Byte Output Word				
Fixed Port	1110011w	port		
Variable Port	1110111w			
OUTM Output Multiple	0110111w			
POLL Poll and wait	10011011			

Table C-1. Instruction Set Summary (Cont'd)

Mnemonic	Byte 1	Byte 2	Byte 3	Byte4
POP Pop Word off Stack into Destination				
Memory Operand	10001111	mod 000 m		
Register Operand	01011reg			
Segment Register	000seg111	(reg 01)		
Pop Flags off Stack	10011101			
All General Registers	01100001			
PREPARE Prepare New Stack Frame	11001000	data-lo	data-hi	data
PUSH Push Word onto Stack				
Memory Operand	11111111	mod 110 m		
Register Operand (Word)	01010reg			
Segment Register	000seg110			
Push Flags onto Stack	10011100			
Push All General Registers	01100000			
Immediate	011010s0	data	data if s= 0	

Table C-1. Instruction Set Summary (Cont'd)

Mnemonic	Byte 1	Byte 2	Byte 3	Byte4
REP/REPE/REPZ/REPNE REPNZ Repeat String Operation	1111001z			
RPC Repeat While Carry	01100101			
REPNC Repeat While Not Carry	01100100			
RET Return from Procedure				
Intrasegment	11000011			
Intrasegment and Add Immediate to Stack Pointer	11000010	data-lo	data-hi	
Intersegment	11001011			
Intersegment and Add Immediate to Stack Pointer	11001010	data-lo	data-hi	
RETEM Return from Emulation	11101101	11111101	(effective for 8080 mode also)	
RETI Return from Interrupt	11001111			
ROL Rotate Left	110100vw	mod 000 r/m		
by count	1100000w	mod 000 r/m	count	

Table C-1. Instruction Set Summary (Cont'd)

Mnemonic	Byte 1	Byte 2	Byte 3	Byte4
ROLLC				
Rotate Left through Carry	110100vw	mod 010 r/m		
by count	1100000w	mod 010 r/m	count	
ROL4 Rotate Left Nibble				
8 bit Memory or Register Operand	00001111	00101000	mod 000 r/m	
ROR				
Rotate Right	110100vw	mod 001 r/m		
by count	1100000w	mod 001 r/m	count	
RORC				
Rotate Right through Carry	110100vw	mod 011 r/m		
by count	1100000w	mod 011 r/m	count	
ROR4 Rotate Right Nibble				
8 bit Memory or Register Operand	00001111	00101010	mod 000 r/m	
SET1 Set Bit				
Bit CL of Memory or Register Operand	00001111	0001010w	mod 000 r/m	
Bit imm of Memory of Register Operand	00001111	0001110w	mod 000 r/m	imm
Carry Flag	11111001			
Direction Flag	11111101			

Table C-1. Instruction Set Summary (Cont'd)

Mnemonic	Byte 1	Byte 2	Byte 3	Byte4
SHL Shift Left				
Shift Arithmetic Left and Shift Logical Left	110100vw	mod 100 r/m		
by count	1100000w	mod 100 r/m	count	
Shift Logical Right	110100vw	mod 101 r/m		
by count	1100000w	mod 101 r/m	count	
SHRA				
Shift Right Arithmetic	110100vw	mod 111 r/m		
by count	1100000w	mod 111 r/m	count	
STM/STMB/STMW Store Multiple/ Store Multiple Byte/ Store Multiple Word	1010101w			
SUB Subtract				
Memory or Register Operand and Register Operand	001010dw	mod reg r/m		
Immediate Operand from Memory or Register Operand	100000sw	mod 101 r/m	data	data if s:w= 01
Immediate Operand from Accumulator	0010110w	data	data if w= 1	

Table C-1. Instruction Set Summary (Cont'd)

Mnemonic	Byte 1	Byte 2	Byte 3	Byte4
SUBC Subtract with Carry				
Memory or Register Operand and	000110dw	mod reg r/m		
Register Operand				
Immediate Operand from Memory Register Operand	100000sw	mod 011 r/m	data	data if or w= 1
Immediate Operand from Accumulator	0001110w	data	data if w= 1	
SUB4S Subtract Nibble String	00001111	00100010		
TEST Test, Logical AND				
Memory or Register Operand with Register Operand	1000010w	mod reg r/m		
Immediate Operand with Memory or Register Operand	1111011w	mod 000 r/m	data	w= 1 data if
Immediate Operand with Accumulator	1010100w	data	data if w= 1	
TEST1 Test Bit				
Bit CL of Memory or Register Operand	00001111	0001000w	mod 000 r/m	
Bit imm of Memory or Register Operand	00001111	0001100w	mod 000 r/m	imm

Table C-1. Instruction Set Summary (Cont'd)

Mnemonic	Byte 1	Byte 2	Byte 3	Byte4
TRANS/TRANSB Translate Byte	110101			
XCH Exchange				
Memory or Register Operand with Register Operand	1000011w	mod reg r/m		
Register Operand with Accumulator	10010reg			
XOR Exclusive OR				
Memory or Register Operand with Register Operand	001100dw	mod reg r/m		
Immediate Operand Memory or Register Operand	1000000w	mod 110 r/m	data	data if w= 1
Immediate Operand to Accumulator	0011010w	data	data if w= 1	
Segment Override Prefix	001sreg110			

D

8087 Programming and Instruction Set Summary

Introduction

The 8087 can act as a coprocessor with the host microprocessor or as a numeric data processor. As a coprocessor, the 8087 shares the same instruction stream and can perform parallel executions. In the memory addressing mode, 8086/8088/80186 ESCAPE instructions will cause the 8086/8088/80186 to calculate an address and read its contents. The 8086/8088/80186 ignores the contents at this address. Meanwhile, the 8087 has been monitoring the instruction stream. When an ESCAPE instruction is detected, the 8087 starts processing. The 8087 latches the instruction. If an address was calculated, it is captured. The data is then read by the 8086/8088/80186 at this location. The instruction is decoded by the 8087 to determine how many more words are needed from memory. After fetching all the data required, the 8087 releases the bus and begins calculating. The 8086/8088/80186 then continues executing the instruction stream.

In numeric processing the 8087 has four rounding modes selected by the rounding control (RC) field in the control word (refer to figure D-2). Rounding occurs when the format of the destination cannot exactly represent the true result in arithmetic and store operations. The precision control (PC) field selects the precision of the result: 24, 53, or 64 bits; default is 64 bits. Real numbers can be closed by either of two models of infinity: projective or affine. The infinity control (IC) field selects the type of closure. Default is projective.

The 8087 represents data and final results of calculations between + /- 2.3 x 10308 to + /- 1.7 x 10308 (at double precision). This is not an exact representation. Remember that arithmetic on real numbers is inherently approximate. However, the 8087 does perform exact arithmetic on integers. An operation on two integers returns an exact integer result (providing it is within range).

Note

Since the 8087 is a coprocessor, it uses the host processor directive, i.e., "8086", "8088", "80186", "80188".

8087 Architecture

The programmer can access the 8087 floating-point stack, the seven words that specify the 8087 environment, and the seven data types addressed by the 8087. A description of these features follows.

Floating Point Stack

This stack has eight elements with sign, exponent, and significand fields. Each of the registers in the stack is 80 bits wide. The field format used in all stack calculations is the temporary real data format described later under Data Types.

The current top element in the floating point stack is the stack top (ST) field in the status word (described in the next section). A load (push) operation decrements the stack pointer by one

then loads a value into the new stack top. For example, FLDLG2 loads log102 into the new stack top. An operation that pops the stack increments the stack pointer by one. For example, FADDP ST[3],ST adds the stack top to element 3, replaces element 3 with this sum, and pops the stack.

8087 Environment

Status word, control word, tag word, two-word instruction address, and two-word data address define the 8087 environment.

Status Word

Status word can be inspected by storing it in memory with an 8087 instruction and then examining it with 8086/8088 CPU code. The format of the status word is shown in figure D-1.

15	14	13	12	11	10	9	8	7	6	5
B	C3	ST	ST	ST	C2	C1	C0	IR	*	PE
			4	3	2	1	0			
			UE	OE	ZE	DE	JE			

B: Busy field shows if 8087 is executing (1) or idle (0). C3-C0: Condition code, used mainly for conditional branching.
ST: Points to 8087 stack element that is current stack top.
IR: Interrupt request, latched to record pending interrupt to 8086/8088 CPU.

Figure D-1. Status Word Format

The remaining six bits are exception flags set when an exception occurs during instruction execution. For more information see the next section concerning the Control Word.

PE:	Precision
UE:	Underflow
OE:	Overflow
ZE:	Zero divide
DE:	Denormalized operand
IE:	Invalid operation

Control Word

The control word is made up of the exception masks, an interrupt enable mask, and control bits. The format of the control word is shown in figure D-2.

15	14	13	12	11	10	9	8	7
*	*	*	IC	RC	RC	PC	PC	RC
		6	5	4	3	2	1	0
		*	PM	UM	OM	ZM	DM	IM

*Not used

IC: Infinity control: affine = 1; projective = 0 (the default).
RC: Rounding control: 00 = to nearest or even (the default); 01 = down; 10 = up; 11 = truncate toward zero.
PC: Precision control: 00 = 24 bits; 01 = reserved; 10 = 53 bits; 11 = 64 bits (the default).
IEM: Interrupt-enable mask: 0 = enabled; 1 = disabled (masked).
PM: Precision Mask: masked (1) = return rounded result; unmasked (0) = return rounded result, request interrupt.
UM: Underflow Mask: masked (1) = denormalize result; unmasked (0) = (for register destination) adjust exponent, store result, request interrupt; (for memory destination) request interrupt.
OM: Overflow Mask: masked (1) = return properly signed ; unmasked (0) = (for register destination) adjust exponent, store result, request interrupt; (for memory destination) request interrupt.
ZM: Zerodivide Mask: masked (1) = return y signed with EXCLUSIVE OR of operand signs; unmasked(0) = request interrupt.
DM: Denormalized Operand Mask: masked (1) = (for memory operand) proceed as usual; (for register operand) convert to valid unnormal, then reevaluate for exceptions; unmasked (0) = request interrupt.
IM: Invalid Operation Mask: masked (1) = if one operand is NAN, return it; if both are NANS, return NAN having the larger absolute value; if neither is NAN, return indefinite; unmasked (0) = request interrupt.

Figure D-2. Control Word Format

Tag Word Tag fields TAG(0) through TAG(7) describe the status of stack

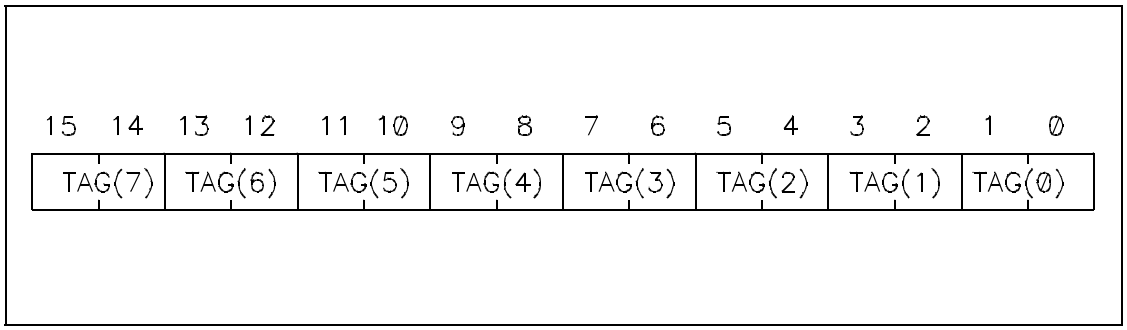

Figure D-3. Tag Word Format

elements 0 through 7, respectively. The format is shown in figure D-3 above.

Tag Field Values:

00 = Values (Normal or Unnormal)
01 = Zero (True)
10 = Xpecial (Not-A-Number, infinity, or Denormal)
11 = Empty

Exception Pointers

Exception pointers are available for user written exception handlers. When the 8087 executes an instruction, the instruction address and opcode are saved in the exception pointers. If the instruction references a memory operand, the operand address is also saved. An exception handler can be written to store these pointers in memory and obtain information concerning the instruction that caused an error. The exception pointers format is shown in figure D-4.

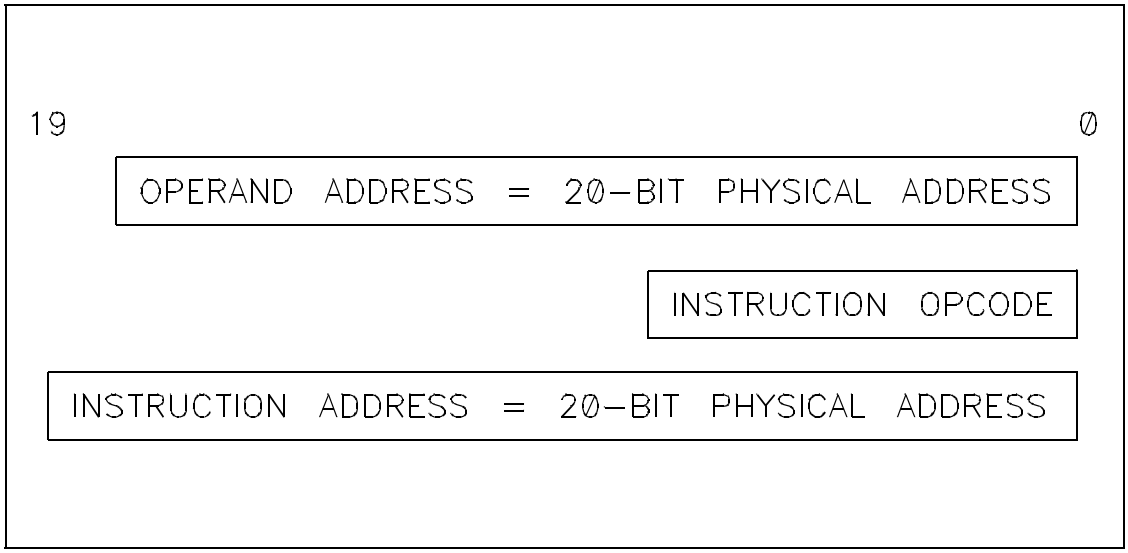

Figure D-4. Exception Pointers Format

Instruction Opcode

The instruction opcode is defined in the 11 least significant bits; the five most significant bits are always the 8087 hook (11011B), i.e., the CPU ESCAPE bits.

Data Types

The 8087 can address seven different data types with all of the 8086 addressing modes. Table D-1 lists the seven addressable 8087 data types.

Table D-1. 8087 Data Types

Data Type	Bits	Significant Decimal Digits	Approx. Decimal Range
Word Integer	16	4-5	-32768 < _N< _+ 32767
Short Integer	32	9	-2x109 < _N< _+ 2x109
Long Integer	64	18	-9x1018< _N< _+ 9x1018
Packed Decimal	80	18	-9...9 < _N< _+ 9...9
Short Real	32	6-7	0, 1.2x10-38 < _!N!< _+ 3.4x1038
Long Real	64	15-16	0, 2.3x10-308 < _!N!< _1.7x10308
Temporary Real	80	19-20	0,3.4x10-4932 < _!N!< _1.1x104932

The data formats are shown in figure D-5.

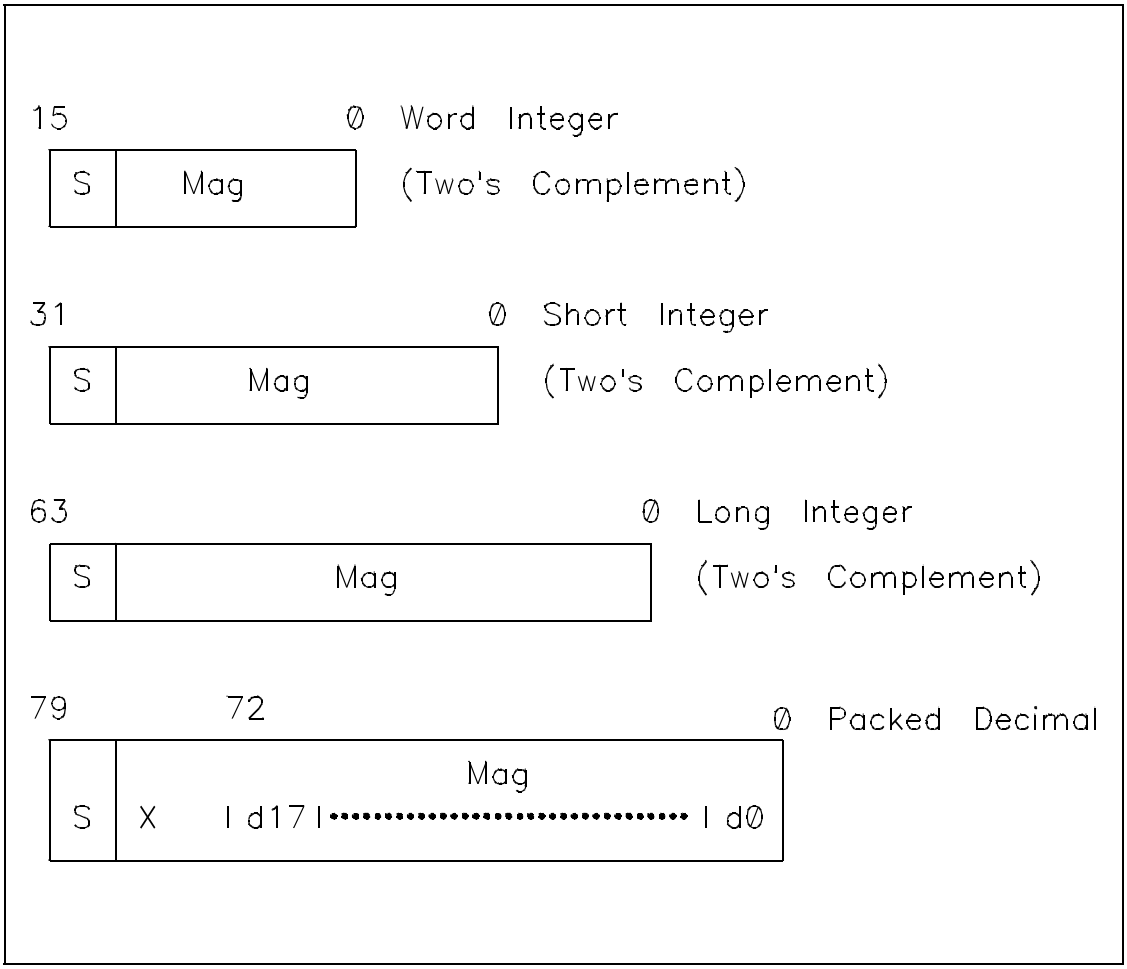

Figure D-5. Data Formats

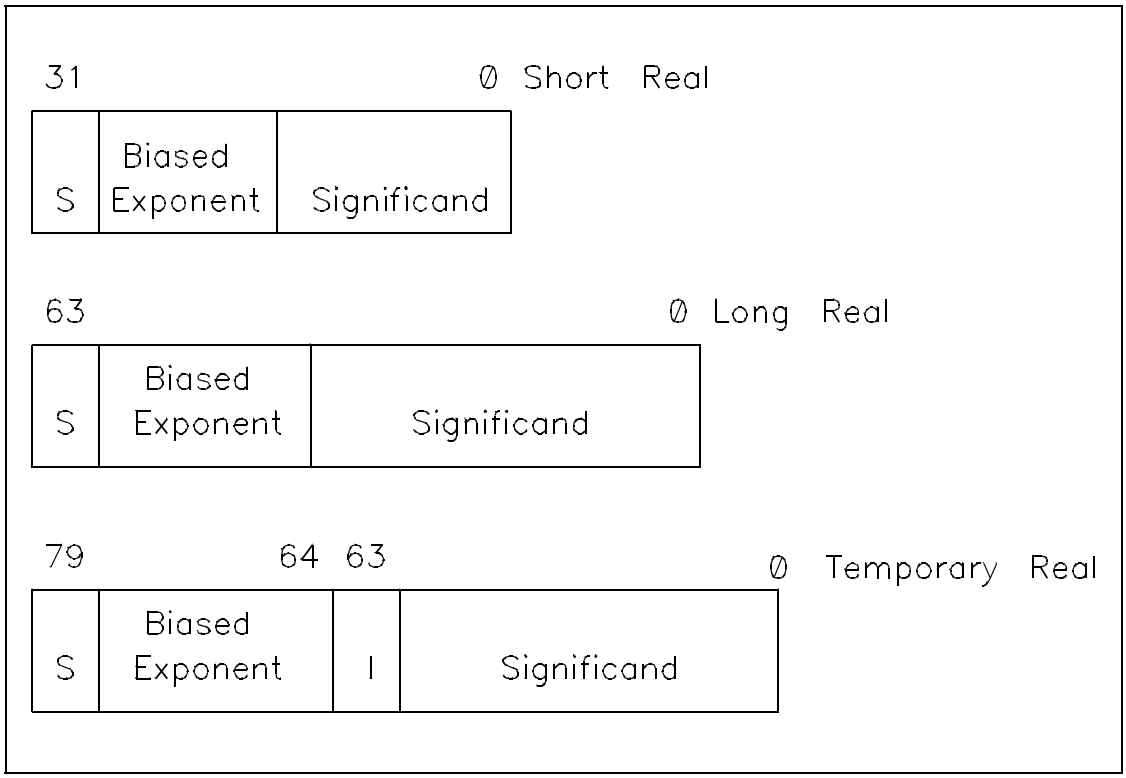

Figure D-5. Data Formats (Cont'd)

S:	Sign bit (0= positive, 1= negative).
Mag:	Magnitude
dn:	Decimal digit- two per byte
X:	Don't care; 8087 ignores when loading, zeros when storing.
I:	Integer bit of significand; stored in temporary real, implicit in short real and long real.
^ :	Implicit binary point location.
Biased Exponent:	Short Real 7FH Long Real 3FHH Temporary Real 3FFFH

Rules and Conventions

The 8087 instructions use either floating point stack elements or variables in memory as operands. The 8087 instructions cannot use labels, 8086 registers, or immediate values as operands. All of the instructions are summarized at the end of this chapter.

Data Transfer Instructions

The data transfer instructions move operands between stack elements or between the stack top and memory. Each of the seven data types can be converted to temporary real and loaded onto the stack or stored in memory in one operation. The 8087 tag word is automatically updated by data transfer instructions to show stack contents after instruction execution. The data transfer instructions are: load real (FLD), store real (FST), store real and pop (FSTP), exchange registers (FXCH), integer load (FILD), integer store (FIST), integer store and pop (FISTP), packed decimal (BCD) load (FBLD), and packed decimal (BCD) store and pop (FBSTP).

Examples

```
FLD1          ;Load 1 on top of stack.
FST  ST[4]    ;Transfer top of stack to stack element 4.
```

Arithmetic Instructions

8087 arithmetic instructions have many variations on basic add, subtract, multiply and divide operations. Operands can be located in stack elements or memory. Results can be deposited in any of the stack elements. Operands can be any of the following data types: word integer, short integer, short real, or long real. Five instruction forms using these instructions include: classical stack, stack element, stack element and pop, real memory, and binary integer. These forms are explained in detail in the following pages. The forms, mnemonics, and operand forms are summarized in table D-2.

Table D-2. Arithmetic Instructions

Instruction Form	Mnemonic	Operand
Classical Stack	Fxx	ST[1],ST
Stack Element	Fxx	ST[i],ST or ST,ST[i]
Stack Element and Pop	FxxP	ST[i],ST
Real Memory	Fxx	ST,short-real/long-real
Binary Integer	FIxx	ST,word-integer/short-integer

Implicit operands are shown in italics; they are not coded.

xx =	ADD	destination < --destination + source
	DIV	destination < -- destination / source
	DIVR	destination < -- source / destination
	MUL	destination < -- destination . source
	SUB	destination < -- destination - source
	SUBR	destination < -- source - destination

Assembler code generation for the arithmetic instructions FMUL, FMULP, FADD, FADDP, FDIV, FDIVP, FDIVR, FSUB, FSUBP, and FSUBPR takes place in one of two ways (subject to the issue/revision date of the assembler software). The pseudo instruction "NEW_8087" should be used if the software is dated 1 February, 1984, or later and a program has been written to be compatible with the latest processor revision. The pseudo instruction "OLD_8087" can be used if the machine code is to be compatible with the old software. OLD_8087 is the default if neither instruction is specified.

The pseudo instruction need be used only once, but must precede any 8087 instructions.

Classical Stack

In this form, the 8087 operates like a classical stack machine. Only the opcode is coded. The 8087 takes the source operand from the top of the stack and the destination from the next stack element. The operation is completed, the stack is popped, and the result of the operation is returned to the new stack top. This effectively replaces the operands with the result. The instructions that can be used with this form are: FADD, FSUB, FSUBR, FMUL, FDIV, and FDIVR.

Example

```
FSUB        ;Subtract stack top from next stack
            ;element, pop stack and return
            ;difference to floating point stack.
```

Stack Element

In this form, the stack top (ST) is one operand and any stack element is the other operand. The instructions that can be used with this form are: FADD, FSUB, FSUBR, FMUL, FDIV, and FDIVR.

Example

```
FADD ST,ST[4]     ;Add the stack top to stack element 4,
                  ;replace stack element 4 with the
                  ;sum, and pop the floating point stack.
```

Stack Element and POP

In cases where the stack top is only needed for one operation, this form picks up the stack top for the source operand and then discards it by popping the floating point stack. Instructions that can be used with this form are: FADDP, FSUBP, FSUBRP, FMULP, FDIVP, and FDIVRP.

Example

```
FDIVP ST[3],ST    ;Divide stack element 3 by the stack
                  ;top, replace element 3 with the
                  ;quotient, and pop the floating
                  ;point stack.
```

Real Memory

In this form, a real number is used directly as a source operand. The instructions that can be used with this form are: FADD, FSUB, FSUBR, FMUL, FDIV, and FDIVR.

Example

```
FMUL RANGE        ;Multiply the stack top by the
                  ;value in memory for RANGE and
                  ;replace the stack top with the
                  ;product.
```

Binary Integer

A binary integer is used directly as a source operand. Instructions used with this form are: FIADD, FISUB, FISUBR, FIMUL, FIDIV, and FIDIVR.

Example

```
FIMUL TBL_5  ;Multiply the integer memory operand
             ;TBL_5 times the stack top and return
             ;the product to the stack top.
```

Comparison Instructions

8087 comparison instructions analyze the stack top, usually in relationship to another operand. The result is reflected in the status word condition code. This can be inspected by transferring the condition code to memory with the store status word instruction (FSTSW). The compare instructions are: compare real (FCOM), compare real and pop (FCOMP), compare real and pop twice (FCOMPP), integer compare (FICOM), integer compare and pop (FCOMP), test (FTST), and examine (FXAM).

Example:

```
FCOM ST[3]    ;Compare stack top with stack element 3.
```

Transcendental Instructions

Transcendental instructions perform the core calculations for all the common trigonometric, inverse trigonometric, hyperbolic, inverse hyperbolic, logarithmic, and exponential functions. Prologue and epilogue software can be used to reduce arguments to a range accepted by the instructions and to adjust the result to correspond to the original arguments, if necessary. These instructions operate on the top one or two stack elements. The results are returned to the stack. Operands must be valid and the timing within the range for the instruction. An op-

erand to a transcendental must be normalized to be valid. Denormals, unnormals, infinities, and NANs are considered invalid. The transcendental instructions are: partial tangent (FPTAN), partial arctangent (FPATAN), calculate 2x-1 (F2XM1), calculate $Y*\log_2 X$ (FYL2X), and calculate $Y*\log_2(X+1)$ (FYL2XP1).

Constant Instructions

All of these instructions push a constant onto the stack. These constants have full temporary real precision (64 bits) and are accurate to about 19 decimal digits. The constant instructions are: load + 0.0 (FLDX), load + 1.0 (FLD1), load pi (FLDPI), load $\log_{10}2$, (FLDL2T), and load $\log_e 2$ (FLDL2E).

Processor Control Instructions

Most of these instructions are used in system-level activities such as initialization, exception handling, and task switching. Some of the instructions have alternate mnemonics with a second character N inserted. This instructs the assembler to precede the instruction with a CPU NOP instead of a CPU WAIT. **The alternate mnemonic should be used if there is the danger of an endless wait with the CPU WAIT instruction.** An endless wait could ensue if, for example, the CPU interrupt enable flag was cleared during a time when the 8087 was expected to generate an interrupt external to the CPU. Program execution would be inhibited because the interrupt would go unanswered. A list of the control instructions follows:

FINIT/FNINIT	Initialize processor
FDISI/FNDISI	Disable interrupts
FENI/FNENI	Enable interrupts
FLDCW	Load control word
FSTCW/FNSTCW	Store control word
FSTSW/FNSTSW	Store status word
FCLEX/FNCLEX	Clear exceptions
FSTENV/FNSTENV	Store environment
FLDENV	Load environment
FSAVE/FNSAVE	Save state
FRSTOR	Restore state
FINCSTP	Increment stack pointer
FDECSTP	Decrement stack pointer
FFREE	Free register
FNOP	No operation
FWAIT	CPU wait

Special 8087 Pseudo Instructions

This information supplements chapter 6 of this manual, the PSEUDO INSTRUCTION SUMMARY. Only instructions applicable to the 8087 processor are included here.

Pseudos DD and DW are the same as those for the 8086 and are explained in chapter 6 of this manual. The 8087 pseudo instructions DQ and DT are explained on the following pages.

8087 Instruction Set Summary

The instruction set is summarized in table D-3 in alphabetical order. Refer to the manufacturer's users manual for details. An explanation of the symbols used in the table follows. Note, d, mod, and r/m are the same as symbols used in the 8086.

- d: Destination; 0 = top of stack, 1 = one of stack elements.
- [i]: Three-bit field identifying stack element; 0 = top of stack, 1 = element next to top, and so on.
- m: One-bit field for data type length; real, 0 = short real 1 = long real; integer, 0 = short integer 1 = word integer.
- mod: Two-bit field defining addressing mode.
- r/m: mod, m, and this three-bit field define EA.

The 8087 instruction set summary, Table D-3, follows the pseudo instructions.

DQ

Define Quadword
(Special 8087 Pseudo)

Syntax

Label	Operation	Operand
Name	DQ	expression [,...]

Description

The DQ instruction can be used to accomplish the following:

- Initialize memory locations.
- Define the type characteristic of variables.

When used with a variable expression in the label field, the DQ instruction defines the variable to be type quadword.

The DQ instruction can also be used to define long integer or long real data types. An expression used to define long integers can only include integer numbers. Only 32 bits of expression are used to define the lower two words of the long integer. The upper two words of the expression are the sign extension. Real numbers are always expressed in decimal values. Be sure to include the decimal point. You may use either the normal decimal form of the expression or the scientific form. You may also specify either positive or negative numbers and exponents (+ /-n.mE+ /-x). Positive numbers are assumed if you do not specify.

DQ (Cont'd)

Example

```
FILE: DQ:USERID                          HEWLETT-PACKARD: 8086 Assembler
LOCATION     OBJECT CODE          LINE     SOURCE          LINE

                                    1     "8086"
0000         96000000               2     LONG_INTEGER    DQ      123+ 27
0004         00000000
0008         0000000000             3      LONG_REAL      DQ      123.0
000D         C05E40

Errors= 0
```

DT

Define Tenbyte
(Special 8087 Pseudo)

Syntax

Label	Operation	Operand
Name	DT	expression [,...]

Description

The DT instruction can be used to accomplish the following.

- Initialize memory locations.
- Define the type characteristic of variables.

When used with a variable name in the label field, the DT instruction defines the variable to be type "tenbyte".

The DT instruction can also be used to define temporary real and packed decimal data types. Real numbers are always expressed in decimal values. Be sure to include the decimal point. You can use either the normal decimal form of the expression or the scientific form. You can also specify either positive or negative numbers and exponents (+ /-n.mE+ /-x). Positive numbers are assumed if you do not specify. Range is -1.1E+ 4932 to -3.4E-4932, + 1.1E+ 4932 to + 3.4E-4932. Packed decimals can be up to 18 decimal characters without decimal point.

DT (Cont'd)

Example:

```
FILE: DT:USERID                    HEWLETT-PACKARD: 8086 Assembler
LOCATION        OBJECT CODE LINE   SOURCE LINE

                                   1 "8086"
0000            0000000000         2 TEMP_REAL          DT      124.0
0005            0000F80540
000A            2401000000         3 PACKED_DECIMAL     DT      124
000F            0000000000

Errors=  0
```

Table D-3. 8087 Instruction Set Summary

Mnemonic	Byte 1	Byte 2	Byte 3
F2XMI Calculate 2X -1	10011011	11011001	11110000
FABS Take absolute value of top of stack	10011011	11011001	11100001
FADD/FADDP Add real or add real and pop stack			
Stack top and stack element	10011011	11011d00	110000[i]
Stack top and memory operand	10011011	11011m00	mod000r/m
Pop stack	10011011	11011110	11000{ i}
FBLD Load packed decimal (BCD) onto top of stack	10011011	11011111	mod100r/m
FBSTP Store packed decimal (BCD) and pop stack	10011011	11011111	mod110r/m
FCHS CHange sign of the top stack element	10011011	11011001	11100000
FCLEX/FNCLEX Clear exceptions	10011011	11011011	11100010
FCOM Compare real			
Compare stack top and stack element	1001 011	11011000	11010[i]
Compare stack top and memory operand	100-11011	11011m00	mod010r/m

Table D-3. 8087 Instruction Set Summary (Cont'd)

Mnemonic	Byte 1	Byte 2	Byte 3
FCOMP Compare real and pop stack			
Compare stack top and and stack element and pop	10011011	11011000	11011[i]
Compare stdack to and	10011011	11011m00	mod011r/m
FCOMP Compare real and memory and pop	10011011	11011110	11011001
FCOMPP Compare real and pop stack twice	10011011	11011110	11011001
FDECSTP Decrement stack top pointer	10011011	11011001	11110110
FDISI/FNDISI Disabole interrupts	10011011	11011011	11100001
FDIV/FDIVP Divide real or divide real and pop stack			
Stack top and stack element	10011011	11011d00	11111[i]
Stack top and memory operand	10011011	11011m00	mod111r/m
Pop stack	10011011	11011110	11111[i]
FENI/FENI Enable interrupts	10011011	11011011	11100000
FIADD Add integer	10011011	11011m10	mod000r/m

Table D-3. 8087 Instruction Set Summary (Cont'd)

Mnemonic	Byte 1	Byte 2	Byte 3
FICOM/FICOMP Integer compare or integer compare and pop stack			
Compare integer	10011011	11011m10	mod010r/m
Compare integer and pop stack	10011011	11011m10	mod011r/m
FDIV Integer divide	10011011	11011m10	mod110r/m
FIDIVR Reversed integer divide	10011011	11011m10	mod111r/m
FILD Load integer onto top of sdtack			
Integer memory to top of stack	10011011	11011m11	mod000r/m
Long integer memory to top of stack	10011011	11011111	mod101r/m
FIMUL Integer multiply	10011011	11011m10	mod001r/m
FINCSTP Increment stack	10011011	11011011	11100011
FINIT/FNINIT Initialize 'processor	10011011	11011011	11100011
FIST Store integer	10011011	11011m11	mod010r/m

Table D-3 8087 Instruction Set Summary (Cont'd)

Mnemonic	Byte 1	Byte 2	Byte 3
FISTP Store integer and pop stack			
Top of stack to intger memory and pop stack	10011011	11011m11	mod011r/m
Top of tack to long integer and pop stack	10011011	11011111	mod111r/m
FISUB Integer subtract	10011011	11011m10	mod100r/m
FISUBR Reversed integer subtract	10011011	11011m10	mod101r/m
FLD Load real onto top of stiack			
Stack element to stack top	10011011	11011001	11000[i]
Real memory operand to stack top	10011011	11011m01	mod000r/m
Temporary real memory operand to stack top	10011011	11011011	mod101r/m
FLD1 Load + 1.0 onto top of stack	10011011	11011001	11101000
FLDCW Load control word	10011011	11011001	mod101r/m
FLDENV Load 8087	10011011	11011001	m,0d100r/m
FLD2E Load $\log_2 10$ onto top of stack	10011011	11011001	11101010

Table D-3. 8087 Instruction Set Summary (Cont'd)

Mnemonic	Byte 1	Byte 2	Byte 3
FLDLG2 Load $\log_{10}2$ onto top of stack	10011011	11011001	11101001
FLDLN2 Load $\log_e2$ onto top of stack	10011011	11011001	11101101
FLDPI load pi onto top of stack	10011011	11011001	11101011
FLDZ Load + 0.0 onto top of stack	10011011	11011001	11101110
FMUL/FMULP Multiply real or multiply real and pop stack			
Stack top and stack element	10011011	11011d00	‘‘00‘[i]
Stack top and memory operand	10011011	11011m00	mod001r/m
Pop stack	10011011	11011110	mod001r/m
FNOP No operation	10011011	11011001	11010000
FSTCW/FNSTCW Store control word	1001101	11011001	mod111r/
FSTENV/FNSTENV Store 8087 enviornment	‘00‘‘0‘‘	11011001	mod110r/m
FSTENV/FNSTENV Store 8087 environment	10011011	11011001	mod110r/m
FSTSW/FNSTSW Store 8087 status word	10011011	11011101	mod111r/m

Table D-3. 8087 Instruction Set Summary (Cont'd)

Mnemonic	Byte 1	Byte 2	Byte 3
FPATAN Partial arctangent function	10011011	11011001	11110011
FPREM Partial remainder	0011011	11011001	11111000
FPTAN Partial tangent function	10011011	11011001	11110010
FRNDINT Round to integer	10011011	11011001	11111100
FRSTOR Restore state	10011011	11011101	mod100r/
FSCALE Scale	10011011	11011001	1111110
FSQRT Square root	10011011	11011001	11111010
FST Store real			
Stack top and stack element	10011011	11011101	11010[i]
Stack top to real memory operand	10011011	11011m01	mod010r/m
FSTP Store real and pop stack			
Store top of stack into stack element and pop stack	10011011	1011101	11011[i]
Store top of stack into short or long real memory and pop stack	10011011	11011m01	mod011r/m

Table D-3. 8087 Instruction Set Summary (Cont'd)

Mnemonic	Byte 1	Byte 2	Byte 3
Store top of stack into temporary real operand and pop stack	10011011	1011011	mod111r/m
FSUB/FSUBP Subtract real or subtract real and pop stack			
Stack top and stack element	10011011	11011d00	11100[i]
Stack top and memory operand	10011011	11011m00	mod100r/m
Pop stack	10011011	11011110	11100[i]
FSUBR/FSUBRP Reversed real subtraction or reversed subtraction and pop stack			
Stack top and stack element	10011011	11011d00	11101[i]
Stack top and memory operand	10011011	11011m00	mod101r/m
Pop stack	10011011	11011110	11101[i]
FTST Test top of stack	10011011	11011001	11100100
FWAIT CPU wait	10011011		
FXAM Examine top of stack element	10011011	11011001	11100101

Table D-3. 8087 Instruction Set Summary (Cont'd)

Mnemonic	Byte 1	Byte 2	Byte 3
FXCH Exchange contents of stack element with stack top	10011011	11011001	11001[i]
FXTRACT Extract exponent and significand from number in top of stack	10011011	11011001	11110100
FYL2X Calculate $Y^* \log_2 X$	10011011	11011001	11110001
FYL2XP1 Calculate $Y^* \log_2(X+1)$	10011011	11011001	11111001

E

8089 Programming and Instruction Set Summary

The 8089 microprocessor independently manages and maintains I/O operations. This lifts the I/O burden from the host CPU, significantly improving system throughput.

There are two system configurations: LOCAL and REMOTE. In LOCAL configuration the 8089 shares the system bus with the host processor. In REMOTE the 8089 shares the system bus and has a remote bus not accessible to the host processor. In LOCAL, the 8089 and the host processor have a common bus controlled by request/grant (RQ/GT) circuitry. The system bus shared by the processors can be 8 or 16 bits. The 8089 can address a gigabyte of memory and 64k of I/O addresses. In REMOTE, the 8089 can address memory up to 64k over the remote bus and one gigabyte over the system bus.

Local 64K address space refers to addresses on the remote bus in the REMOTE configuration. In LOCAL, this 64k address space is used for I/O addressing. System space addresses in the LOCAL configuration access memory. In REMOTE, system addresses access the shared system bus.

Note

Use the processor number, "8089_86" or "8089_88", for the assembler directive.

8089 Architecture

There are two independent I/O channels on the 8089. Each channel operates simultaneously. Each has a separate set of registers. Each channel also has separate external interrupt, DMA request, and external terminate pins.

Registers

8089 registers are used in assembly language task block programs and in DMA transfer operations. Registers are identical for both channels. 8089 register organization is shown in figure E-1.

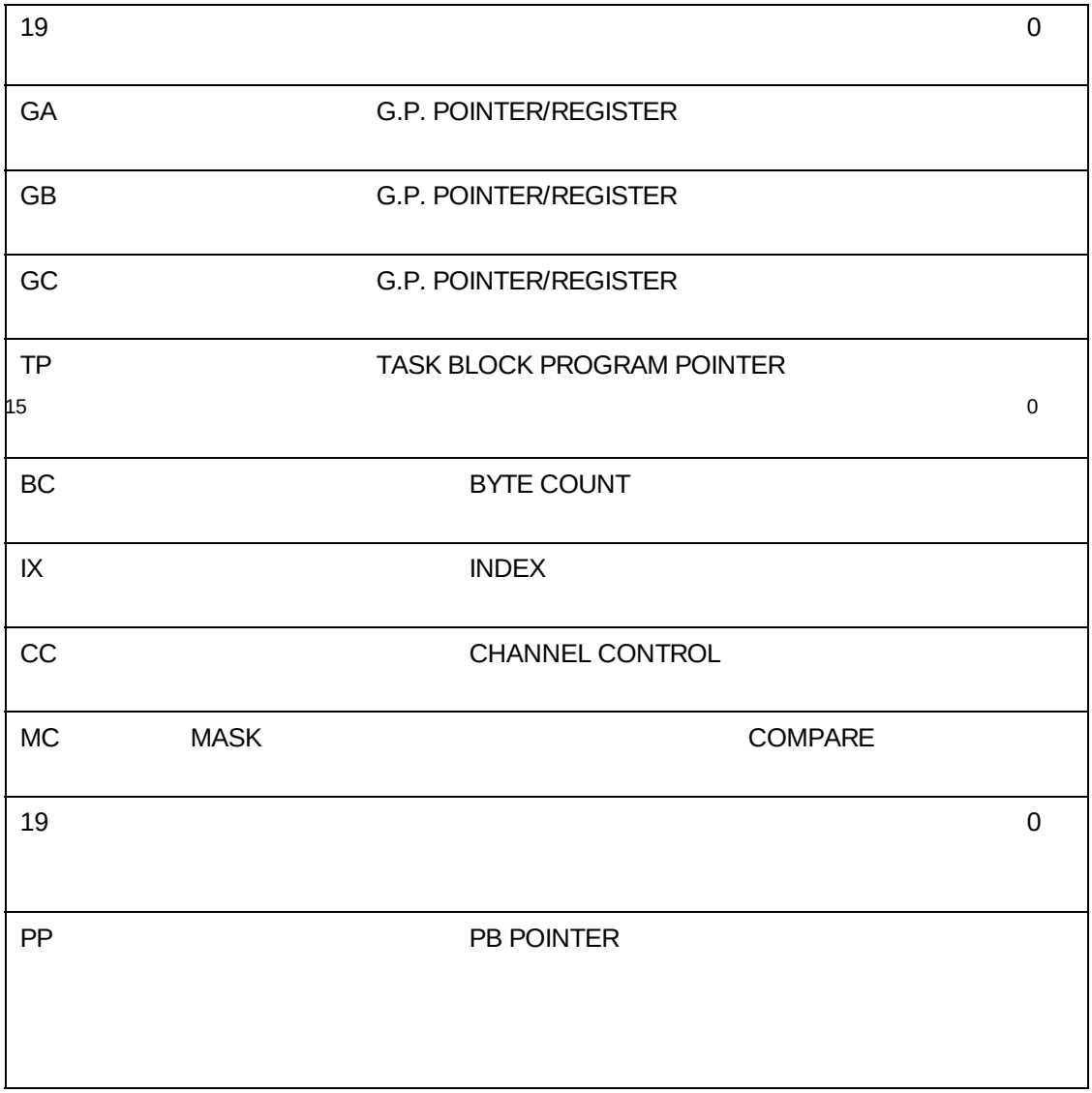

Figure E-1. 8089 Registers

Each register in Figure E-1 has a tag bit associated with it. The tag bit is primarily used in data addressing. A "1" indicates a 16-bit local space address (I/O). A "0" indicates a 20-bit system space address (memory).

Registers GA and GB are 20-bit pointer/registers, plus a tag bit. These registers are used to point to data in task block programs. They provide the source and destination addresses in DMA transfers as controlled by register CC parameters. GA and GB can also be used as 16-bit general purpose registers in task block programs.

Register GC is a 20-bit pointer/register, plus a tag bit. GC points to data in task block programs. During DMA transfers in the translate mode, GC contains the base address of a 256-byte translation table. GC can also be used as a 16-bit general purpose register in task block programs.

Register TP is a 20-bit pointer/register with a tag bit. It points to the address of the next instruction to be executed. TP is loaded from the command parameter block (PB) when task block program execution is started or resumed.

Register BC is a 16-bit general purpose register used as a byte counter during DMA transfers. With an 8-bit source, BC is decremented by one after each transfer (by two after each transfer from a 16-bit source).

Register IX is a 16-bit general purpose register. The contents of IX is added to a base pointer/register to access data in some memory addressing modes.

Register CC is a 16-bit register that controls DMA transfers and chained task block program instruction execution.

Register MC is a 16-bit general purpose register that supplies mask and compare bytes for instructions JMCE and JMCNE. It is also used in DMA transfer mask/compare operations.

The last register, **PP,** cannot be programmed by the user. This 20-bit register is automatically loaded with the channel command parameter block address when a channel is

started. PP always points to system space (memory). In accessing the user defined part of PB, PP is used as a base address.

Operands

The 8089 has six types of operands: register, pointer/register, immediate data, program location, data memory, and data memory bit. The following paragraphs explain these operands.

Register Operands

The register operand symbols were shown in figure E-1. These symbols identify the registers for the assembler. **They cannot be redefined by the programmer**.

Example

```
MOVI    GA,0F00H        ;Move immediate value 0F00H to register GA.
```

Pointer/Register Operands

The pointer/registers are: GA, GB, GC, and TP. They are 20-bit registers with associated tag bits. These registers point to data memory and I/O space in the CPU system. These registers are included here because they can be used as 16-bit general purpose registers.

Example

```
LPDI    C,TABLE         ;Load register GC with 16 bits of
                        ;immediate data represented by TABLE.
```

Immediate Data Operands

Immediate data operands can be a data memory location, a program location, or an 8- or 16-bit value.

Examples

```
ARRAY   DS  56      ;Reserve 56 bytes of data memory.
                    ;The first byte is labeled ARRAY.

LPDI    TP,LABEL1   ;Load register TP with the address
                    ;of program location LABEL1.

ORI     GA,0D6BH    ;OR contents of GA with 16-bit
                    ;immediate value of 0D6BH.
```

Program Location Operands

A program location operand is used in conditional and unconditional control transfer instructions to specify the jump location. In most cases this is a label representing the jump location in the program.

Example

```
JMP     LAST1       ;Jump unconditionally to instruction
                    ;LAST1.
```

Data Memory Operands

Data memory is always addressed indirectly through one of the pointer registers: GA, GB, GC, or PP (represented by reg). The 20-bit system space (memory) and 16-bit local space (I/O) can be accessed. There are four forms of data memory operands.

1. Base address only [reg]; reg contains the data memory address.

Example

```
MOV     CC,[GA]      ;Starting at the address in GA, move
                     ;16 bits of data memory to register CC.
```

2. Base address plus unsigned 8-bit offset [reg].d (d is expression evaluated modulo 256 forming an 8-bit offset).

Example

```
OR      MC,[GC].6    ;OR register MC with word of data
                     ;memory starting at location GC+ 6
                     ;(low byte).
```

3. Base address plus index register [reg+ IX] forms data memory address. No change occurs in base address or index register.

Example

```
ADD     [GB+ IX],BC  ;Add contents of BC to data memory
                     ;starting at address GB+ IX (low byte).
```

4. Base address plus index register [reg+ IX+]. Index register is post incremented by byte (1) or word (2). Data memory address is sum of base address and index register. After instruction execution, index register is automatically incremented by size of operand. No change in base address occurs.

Example:

```
DEC     [GA+ IX+ ]      ;Decrement data memory word starting
                        ;at GA+ IX. After execution, IX is
                        ;incremented by 2 (word).
```

Data Memory Bit Operands

Instructions that operate on bits of a data memory byte need operands specifying the bit. Bits are numbered as follows:

			BITS				
7	6	5	4	3	2	1	0
MSB							LSB

Example:

```
CLR     [GB],4          ;Clear bit four of data memory byte at GB.
```

Special 8089 Pseudo Instructions

This information supplements chapter 6 of this manual, the PSEUDO INSTRUCTION SUMMARY. Only instructions applicable to the 8089 processor are included here.

Pseudos DB, DD, and DW are the same as the 8086. 8089 pseudo DS is the same as the 8086 DBS. These are explained in Chapter 7 of this manual. For the 8089 pseudo PUBLIC, use the 64000 GLOBAL. EVEN is explained on the following page.

EVEN

Set Program Counter To Even Address
(Special 8089 Pseudo)

Syntax

Label	Operation
[Name]	EVEN

Description

The EVEN pseudo instruction will increment the current program counter by one if it is odd. If it is even the pseudo is ignored.

If a label name is present, it is assigned the starting address of the program counter.

Example

```
FILE: EVEN:USERID                          HEWLETT-PACKARD: 808986 Assembler
LOCATION     OBJECT CODE          LINE          SOURCE LINE

                                  1 "8089_86"
0000         003F55               2            DB          0,3FH,55H
                                  3 LBEL       EVEN
ERROR-UO                                       ^
0003         4000                 4 L1         DB          40H,0
Errors=  1, previous error at line 3
UO - Unidentified Opcode, Opcode encountered is not defined for this micro-processor
```

8089 Instruction Set Summary

The instruction set is summarized in table E-1 by type of instruction. Refer to the manufacturer's user manual for details. An explanation of the symbols used in the table follows.

b:	Data memory bit symbol.
IM8:	8-bit immediate value.
IM16:	16-bit immediate value.
L:	Expression specifying program location.
DM8:	8 bits data memory.
DM16:	16 bits data memory.
DM24:	3 bytes data memory.
DM32:	4 bytes data memory.
Reg8:	The least significant byte in a 16-bit register. If it is the destination of a data transfer, the data is sign extended (bit 7) to 16 bits. If the register is a 20-bit register, data is sign extended to 20 bits and the tag bit is set to 1. In a 20-bit pointer register, the four MSB are undefined after all arithmetic and logical operations (except addition). Addition to a pointer register can result in a carry into the four MSB. All data is sign extended to 16 bits with arithmetic and logical operations.
Reg16:	All of the 16-bit register is used in an operation. If a 20-bit pointer register is the destination of a data transfer, the data is sign extended (bit 15) to 20 bits and the tag bit is set to logical 1. Also, the upper four bits (16 to 19) are undefined after arithmetic and logical operations. Addition to a pointer register can result in a carry into the four MSB.

Table E-1. 8089 Instruction Set Summary

Arithmetic and Logical		
ADD		
	Reg16,DM16 DM16,Reg16	Add register and 16-bit data memory
ADDB		
	Reg8,DM8 DM8,Reg8	Add register and 8-bit data memory
DDBI		
	Reg8,IM8 DM8,IM8	Add register or 8-bit data memory and 8-bit immediate value
ADDI		
	Reg16,IM16 DM16,IM16	Add register or 16-bit data memory and 16-bit immediate value
AND		
	Reg16,DM16 DM16,Reg16	And register with 16-bit data memory
ANDB		
	Reg8,DM8	And register with 8-bit data memory
ANDBI		
	Reg8,IM8 DM8,IM8	And register or 8-bit data memory with 8-bit immediate value

Table E-1. 8089 Instruction Set Summary (Cont'd)

Arithmetic and Logical (Cont'd)		
ANDI		
	Reg16,IM16 DM16,IM16	And register or 16-bit data memory with 16-bit immediate value
DEC		
	Reg16 DM16	Decrement register or 16-bit data memory
DECB		
	DM8	Decrement 8-bit data memory
INC		
	Reg16 DM16	Increment register or 16-bit data memory
INCB		
	DM8	Increment 8-bit data memory
OR		
	Reg16,DM16 DM16,Reg16	Or register and 16-bit data memory
ORB		
	Reg8,DM8 DM8,Reg8	Or register and 8-bit data memory

Table E-1. 8089 Instruction Set Summary (Cont'd)

Arithmetic and Logical (Cont'd)		
ORBI		
	Reg8,IM8 DM8,IM8	OR register or 8-bit data memory with 8-bit immediate value
ORI		
	Reg16,IM16 DM16,IM16	Or register or 16-bit data memory with 16-bit immediate value
NOT		
	Reg16 DM16 Reg16,DM16	Complement register or 16-bit data memory optional: put complemented data in register
NOTB		
	DM8 Reg8,DM8	Complement 8-bit data memory; optional: put complemented data in register
Bit Manipulation and Test		
SETB		
	DM8,b	Set selected data memory bit to one
CLR		
	DM8,b	Clear selected data memory bit to zero
JBT/LJBT		
	DM8,b,L	Jump on data memory bit true (1)

Table E-1. 8089 Instruction Set Summary (Cont'd)

Bit Manipulation and Test (Cont'd)		
JNBT/LJNBT		
	DM8,b,L	Jump on data memory bit not true (1)
Control Transfer - Unconditional		
CALL/LCALL		
	DM24,L	Store TP pointer/register and tag bit; jump
JMP/LJMP		
	L	Jump
Control Transfer - Conditional		
JMCE/LJMCE		
	DM8,L	Jump on mask/compare equal
JMCNE/LJMCNE		
	DM8,L	Jump on mask/compare not equal
JNZ/LJNZ		
	Reg16,L DM16,L	Jump on nonzero register or data memory word
JNZB/LJNZB		
	DM8,L	Jump on nonzero data memory byte

Table E-1. 8089 Instruction Set Summary (Cont'd)

Control Transfer - Conditional (Cont'd)		
JZ/LJZ		
	Reg16,L DM16,L	Jump on zero register or data memory
JZB/LJZB		
	DM8,L	Jump on zero data memory byte
Data Transfer		
LPD		
	P,DM32	Load 20-bit pointer/register from data memory
LPDI		
	P,IM16	Load 20-bit pointer/register from immediate value
MOVP		
	DM24,P P,DM24	Move 20-bit pointer/register to (store) or from (re-store) memory
MOV		
	Reg16,DM16 DM16,Reg16 DM16,DM16	Move 16 bits of data memory to/from data memory or register

Table E-1. 8089 Instruction Set Summary (Cont'd)

Data Transfer (Cont'd)		
MOVB		
	Reg8,DM8 DM8,Reg8 DM8,DM8	Move 8 bits of data memory to/from data memory or register
MOVI		
	Reg16,IM16 DM16,IM16	Move 16 bits of immediate value to data memory or register
MOVIB		
	Reg8,IM8 DM8,IM8	Move 8 bits of immediate value to data memory or register
Miscellaneous		
HLT		
		Halt task block program execution; channel BUSY flag byte in the CB Cleared to 00H
NOP		
		No operation
SINTR		
		Set interrupt service flip flop
TSL		
	DM8,IM8,L	Test and set data memory byte with system bus locked

Table E-1. 8089 Instruction Set Summary (Cont'd)

Miscellaneous (Cont'd)	
WID	
S,D	Set DMA source and destination logical widths
XFER	
	Begin DMA transfer following execution of next instruction

F

70320/70330 Programming And Instruction Set Summary

This appendix contains general information. Architecture, operands, and condition flags are briefly discussed. The instructions of 70320/70330 microprocessors are upward compatible with those of 70108/70116 microprocessors. Only instruction specific to the 70320/70330 microprocessors are described in this appendix. For common instructions, refer to the "70108/70116 Programming And Instruction Set Summary" chapter in this manual. For detailed descriptions of the microprocessors, refer to the manufacturers users manual.

Note

See the "EXT" pseudo op in the chapter titled "Pseudo Instruction Summary" about "EXT" conflicts with NEC processors.

Programming Considerations

Sixteen-bit operands may be assigned to even or odd address locations. For data and address operands, the least significant byte of the word will be stored in next higher address. The 70330 microprocessor automatically performs the required number of memory accesses, one if the word operand begins on an even byte address, and two if it begins on an odd byte address. The 70320 always performs two memory accesses for each 16-bit operand.

As the assembler directive, use "70320" or "70330".

Addressing Capabilities

In general, memory operands may be addressed directly, using a 16-bit offset, or indirectly, using a base and/or index register added to an optional 8- or 16-bit displacement value.

Instruction Set Summary

The instruction set of the 70320/330 is upwardly compatible with that of the 70108/70116 in the native mode.

The mnemonic instructions which are specific (added) to the 70320/70330 are summarized in table F-1. The instruction set is arranged in alphabetical order. For detailed information refer to the manufacturer's users guide.

Figure F-1 shows the typical machine instruction format. The location of an operand in a register or memory will be specified by up to three fields in instruction format. These fields are the

mode field (mod) , the register field (reg), and the register/memory field (r/m). When used, they occupy the second byte of the instruction format. The mode field occupies the two most significant bits of the byte and specifies how the r/m fields will be used in locating the operand. The reg field occupies the next three bits following the mode field and specifies either an 8-bit register or a 16-bit register where an operand will be located.

Note, bytes three through six of an instruction are optional fields that usually contain the displacement (DISP) value of a memory operand and/or the actual value of an immediate constant operand. The effective address (EA) of the memory operand will be computed according to the mode and r/m fields as follows:

Mode **R/M Fields**

*if mod= 00 then DISP= 0, disp-lo and disp-hi are absent.
if mod= 01 then DISP= disp-lo sign-extended to 16-bits, and disp-hi is absent.
if mod= 10 then DISP= disp-hi:disp-lo.
if mod= 11 then r/m is treated as a reg field.
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

*except if mod= 00 and r/m= 110 then EA= disp-hi:disp-lo.

Register operands may be indicated within the instruction format by the reg field which will represent the selected register, or by an encoded field, in which case EA will represent the register selected by the r/m field. Instructions without a "W" bit in their format refer to either 8- or 16-bit registers according to the following reg field assignments:

	16-bit (W= 1)	8-bit (W= 0)
reg field:	000 = reg AW	000 = reg AL
	001 = reg CW	001 = reg CL
	010 = reg DW	010 = reg DL
	011 = reg BW	011 = reg BL
	100 = reg SP	100 = reg AH
	101 = reg BP	101 = reg CH
	110 = reg IX	110 = reg DH
	111 = reg IY	111 = reg BH

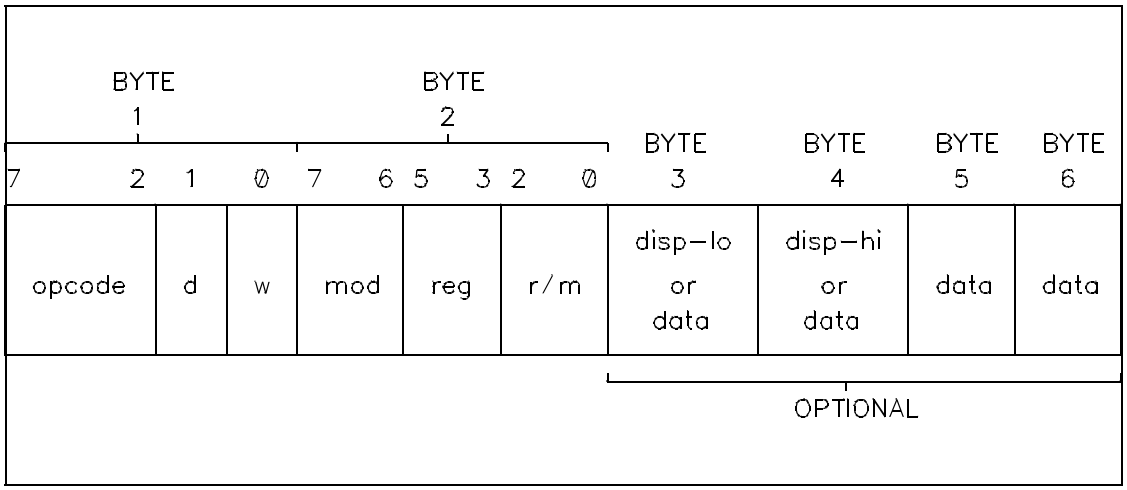

Figure F-1. Typical Instruction Format

d:	direction to (1) or from (0) register
w:	byte (0) or word (1) operation
mod and r/m:	addressing mode - register or memory
reg:	register select

The SEGMENT OVERRIDE PREFIX takes the form of:001reg110 in which the register is assigned in the following manner:

reg	Segment register
00	DS1
01	PS
10	SS
11	DS0

70320/70330 Register Names

The following symbols are reserved. They have special meaning to the assembler and cannot appear as user-defined symbols.

SYMBOL	DESCRIPTION
AH	High-order byte of register A
AL	Low-order byte of register A
AW	16-bit register A
BH	High-order byte of register B
BL	Low-order byte of register B
BP	Base Pointer
BW	16-bit register B
CH	High-order byte of register C
CL	Low-order byte of register C
CW	16-bit register C
DH	High-order byte of register D
DL	Low-order byte of register D
DS0	Data segment 0 register
DS1	Data segment 1 register
DW	16-bit register D
IX	Source index register
IY	Destination index register
PC	Program counter
PS	Program segment register
SP	Stack pointer
SS	Stack segment register

SYMBOL	DESCRIPTION
BRG0	Baud rate generator register 0
BRG1	Baud rate generator register 1
DIC0	DMA interrupt request control register 0
DIC1	DMA interrupt request control register 1
DMAC0	DMA control register 0
DMAC1	DMA control register 1
DMAM0	DMA mode register 0
DMAM1	DMA mode register 1
EMS0	External interrupt macro service control register 0
EMS1	External interrupt macro service control register 1
EMS2	External interrupt macro service control register 2
EXIC0	External interrupt request control register 0
EXIC1	External interrupt request control register 1
EXIC2	External interrupt request control register 2
FLAG	User flag register
IDB	Internal data area base register
INTM	External interrupt mode register
MD0	Modulo/Timer register 0
MD1	Modulo/Timer register 1

SYMBOL	DESCRIPTION
P0	Port 0
P1	Port 1
P2	Port 2
PM0	Port 0 mode register
PM1	Port 1 mode register
PM2	Port 2 mode register
PMC0	Port 0 mode control register
PMC1	Port 1 mode control register
PMC2	Port 2 mode control register
PMT	Port T mode register
PRC	Processor control register
PT	Port T
RFM	Refresh mode register
RxB0	Receive buffer register 0
RxB1	Receive buffer register 1
SCC0	Serial control register 0
SCC1	Serial control register 1
SCE0	Serial error register 0
SCE1	Serial error register 1
SCM0	Serial mode register 0
SCM1	Serial mode register 1
SEIC0	Serial interrupt request control register 0
SEIC1	Serial interrupt request control register 1
SRIC0	Serial receive interrupt request control register 0
SRIC1	Serial receive interrupt request control register 1
SRMS0	Serial receive macro service control register 0
SRMS1	Serial receive macro service control register 1
STBC	Standby control register

SYMBOL	DESCRIPTION
STIC0	Serial transmit interrupt request control register 0
STIC1	Serial transmit interrupt request control register 1
STMS0	Serial transmit macro service control register 0
STMS1	Serial transmit macro service control register 1
TBIC	Time base interrupt request control register
TM0	Timer register 0
TM1	Timer register 1
TMC0	Timer control register 0
TMC1	Timer control register 1
TMIC0	Timer unit interrupt request control register 0
TMIC1	Timer unit interrupt request control register 1
TMIC2	Timer unit interrupt request control register 2
TMMS0	Timer unit macro service control register 0
TMMS1	Timer unit macro service control register 1
TMMS2	Timer unit macro service control register 2
TxB0	Transmit buffer register 0
TxB1	Transmit buffer register 1
WTC	Wait control register

Instruction Set Symbols

The symbols used in table F-1, Instruction Set Summary, are as follows:

SYMBOL	DESCRIPTION
addr	address (16 bits)
addr-hi	Most significant byte of address
addr-lo	Least significant byte of address
d	One-bit field identifying direction to (1) or from (0) register
data	Immediate operand (8- or 16-bit)
disp	8- or 16-bit displacement from end of current instruction
disp-hi	Most significant byte in 16-bit offset displacement
disp-lo	Least significant byte of 16-bit offset displacement
imm	3, 4 or 8-bit immediate operand
mod	Two-bit field defining addressing mode
offset-hi	Most significant byte in 16-bit offset destination address of target instruction
offset-lo	Least significant byte in 16-bit offset destination address of target instruction
reg	Field that defines the defines the register used
r/m	Three-bit field, in conjunction with the mod and "W" fields defines EA
seg	Segment register
seg-hi	Most significant byte in 16-bit segment destination address of target instruction
seg-lo	Least significant byte in 16-bit segment destination address of target instruction
sfr	An 8-bit variable which specifies an 8-bit special function register
port	Number of I/O port
s:w	Sign-extended byte indicator

SYMBOL	DESCRIPTION
v	Interrupt: defines variable type (v= 1), or type 3 (v= 0) Shift or Rotate; variable number of bits to shift or rotate (v= 1), or one bit (v= 0)
w	One-bit field identifying byte (0) or word (1) instruction
z	Instruction being repeated terminated when zero flag is equal to z

Table F-1. 70320/70330 Specific Inst. Set Summary

Mnemonic	Byte 1	Byte 2	Byte 3	Byte 4
BTCLR Branch if True and Clear	00001111	10011100	sfr	imm Byte 5 disp
RETRBI Return from Register Bank Switching Interrupt	00001111	10010001		
BRKCS Break Context Switch	00001111	00101101	11000 reg	
FINT Finish Interrupt	00001111	10010010		
MOVSPA Move Stack Pointer After context switch	00001111	00100101		
MOVSPB Move Stack Pointer Before context switch	00001111	10010101	11111 reg	
STOP Stop	00001111	10011110		
TSKSW Task switch	00001111	10010100	11111 reg	

Assembler Error Messages

Detection and Listing

The assembler detects and lists all errors noted in a source program module. Program errors are indicated in the source program listing by a two-letter code following each source statement that contains an error.

Note 

If multiple errors occur in the same source statement, only the first error noted will be reported (in most cases).

Each error message contains an error code. The error message contains a cursor (^) that points to the error location in the source statement. The error message also contains a statement that indicates the line number of the previous source statement that was in error. Line number indicators facilitate error tracing.

At the end of the program listing is a summary of the number of errors within the program. A brief description of all error codes is also noted at the end of the program listing.

The error message format is as follows:

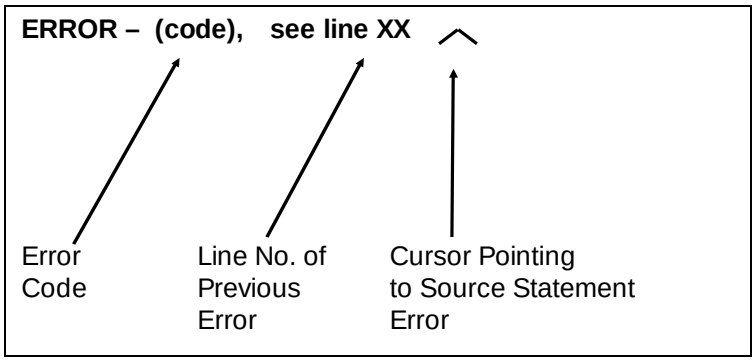

Figure G-1. Error Message Format

G-2 Assembler Error Messages

Assembler Error Codes

The list of error codes (in alphabetical order) along with a description of their meaning is as follows:

AS **ASCII STRING** - The length of ASCII string was not valid or the string was terminated improperly.

CL **CONDITIONAL LABEL** - Syntax of a conditional macro source statement requires a conditional label that was missing.

DE **DEFINITION ERROR** - Indicated symbol must be defined prior to it being referenced. (Symbol may be defined later in program sequence).

DS **DUPLICATE SYMBOL** - Indicates that the defined symbol noted has been previously defined in the program assembly sequence. (This occurs when the same symbol is equated to two values (using EQU directive) or when the same symbol labels two instructions).

DZ **DIVISION BY ZERO** - Invalid mathematical operation found resulting in the assembler trying to divide by zero.

EG **EXTERNAL GLOBAL** - Externals cannot be defined as globals.

EO **EXTERNAL OVERFLOW** - Program module found to have too many external declarations.

ES **EXPANDED SOURCE** - Indicates insufficient input buffer area designated to perform macro expansion. (This could be the re-

sult of too many arguments being specified for a parameter substitution, or too many symbols being entered in the macro definition).

ET **EXPRESSION TYPE** - The resulting type of expression was found to be invalid. An absolute expression was expected and not found or expression contains an illegal combination of relocatable types (refer to chapter 4 for rules and conventions).

IC **ILLEGAL CONSTANT** - Indicates that the assembler encountered a invalid constant. For example: 109B (9 is invalid)

IE **LLEGAL EXPRESSION** - Specified expression found was either incomplete or an invalid term was within the expression.

IO **INVALID OPERAND** - Specified operand was either incomplete or inaccurately used for this operation. (This occurs when an unexpected operand was encountered or the operand was missing. If the required operand is an expression, the error indicates that the first item in the operand field was illegal).

IP **ILLEGAL PARAMETER** - Illegal parameters were found in macro header.

IS **ILLEGAL SYMBOL** - Syntax expected an identifier and instead encountered an illegal character or token.

LR **LEGAL RANGE** - Address or displacement caused the location counter to exceed the maximum memory location of the instruction's addressing capability.

MC **MACRO CONDITION** - Relational (conditional) operator in macro was found to be invalid.

MD **MACRO DEFINITION** - Macro was called before being defined in the source file. (Macro definition must precede call).

ML **MACRO LABEL** - Label was not found within the macro body. (Macros cannot contain labels.)

MM **MISSING MEND** - A macro definition with a missing MEND directive was included in the program.

MO **MISSING OPERATOR** - An arithmetic operator was expected but not found.

MP **MISMATCHED PARENTHESES** - Right or left parenthesis were not found.

MS **MACRO SYMBOL** - A local symbol within a macro body is required but was not found.

NI **NESTED INCLUDE** - The INCLUDE pseudo instruction cannot be nested.

PE **PARAMETER ERROR** - An error detected in the macro parameter was listed in the source statement.

PH **PHASE ERROR** - More code was generated during pass 2 than during pass 1. This error will be caused by an illegal use of a forward reference to a variable.

RC **REPEAT CALL** - Repeat cannot precede a macro call.

RM **REPEAT MACRO** - Repeat pseudo-operation code cannot precede a macro definition.

SE **STACK ERROR** - A statement or expression does not conform to the required syntax.

TR **TEXT REPLACEMENT** - The specified text replacement string was found to be invalid.

UC **UNDEFINED CONDITIONAL** - Conditional operation code was found to be invalid.

UO **UNDEFINED OPERATION CODE** - Operation code encountered was not defined for the microprocessor, or the assembler disallowed the operation to be processed in its current context. (This occurs when the operation code is misspelled or an invalid delimiter follows the label field.)

UP **UNDEFINED PARAMETER** - The parameter found in a macro body was not included in the macro header.

US **UNDEFINED SYMBOL** - The indicated symbol was not defined as a label or declared as external.

Notes

H

Linker Error Messages

Error Messages

When an error is detected during the link process, the linker will determine if the error is fatal or nonfatal. If the error is classified as fatal, the linker will abort the linking process. If the error is nonfatal the linker will continue the linking process, but will generate error messages that will be listed in the output listing. A description of each error message is give in the following paragraphs.

Fatal Error Messages

Upon encountering a fatal error the linker will display one of the following messages on the terminal. The linker will abort the link process and return control of the system to the monitor.

Target Processors Disagree

The linker will issue this message if the relocatable modules to be linked are designed for different processors. Ensure that all relocatable modules assigned for linking are written for the same type microprocessor.

Checksum Error

The linker will issue this message if it is unable to read a relocatable file due to a checksum error or other irregularities in the

file. To correct this situation, reassemble the relocatable file; then, relink.

File Not Found

The linker will issue this message if it is unable to locate a file during a link operation.

File Extension and File Type Disagree

The linker will issue this message if the extension assigned to a file does not agree with its type.

Linker Command File Not Found

The linker will issue this message if a link is requested using an invalid command file name.

Nonfatal Error Messages

Upon encountering nonfatal errors, the linker will continue the link operation and print the error messages (except initialization errors) in the output listing. An error message that is listed will contain a description of the error and the name of the file where the error occurred. If the null list is in effect, the linker will direct the error messages to the system CRT.

Illegal entry: reenter

During initialization the linker will indicate on the terminal that the user has made an illegal response to an interrogation. To correct this situation, reenter the proper response.

Duplicate symbol

During pass 1 of the link process, the linker detects that the same symbol has been declared global by more than one relocatable module. The first definition holds true. The relocatable module that first defines the symbol may be found in the cross-

reference table. To correct this error, remove the extra global declarations.

Load address out of range.

The linker has tried to relocate code beyond the addressing range of the specified microprocessor. To correct this situation, reassign the relocatable addresses.

Multiple transfer address

During pass 1, the linker finds that the transfer address has been defined by more than one relocatable module. The first definition holds true. The relocatable module that first defined the transfer address will be given at the conclusion of the linking. To correct this situation, remove the extra transfer address. Reassemble the amended relocatable module; then, relink. If a xfer address is defined by both a nonload program and a load program, no error will be given. The load program xfer address takes precedence.

Undefined symbol

During pass 2, undefined symbol error occurs when the linker finds that a symbol has been declared external but not defined by a global definition. To correct this situation, define the symbol.

Out of memory in xref

Unlike the fatal error (Out of Memory in Xref), this error occurs when memory space is available for a complete symbol table but only a portion of the cross-reference table. The linker will complete the xref operation, listing only that portion of the cross-reference table for which memory space was available. To correct this situation, reduce the number of files, global symbols, and/or external symbols used during the current link.

Memory overlap

This error indicates that relocatable program areas have been overlapped in memory. The error message will list the program names and the overlapping areas.

Max addr or seg boundary exceeded

This error occurs when the linker has attempted to locate code outside the valid addressing range of the processor or the current segment.

ASCII Conversion Table

General

To produce the ASCII characters in column 1 in the ASCII table, hold down the control (CTRL) key on the keyboard and then press the corresponding character key listed in column # 3. For example, CTRL–H produces a BS or backspace (ASCII = 08H) and CTRL–[produces an ESC or escape (ASCII = 1BH).

Also, deciphering the hexadecimal value of a character is accomplished by adding the plus value (+ 0, + 20, + 40, + 60) of the column in which the character appears to the "N" column value directly across from the character. For example, the hexadecimal value of "a" is 1 + 60, or 61H. The hexadecimal value of "Q" is 11 + 40, or 51H. Similarly, the hexadecimal value of ":" is 1A + 20, or 3AH.

Conversely, subtracting the highest possible plus value from the hexadecimal value will yield the "N" column value. Directly across from the "N" column value, in the appropriate plus value column, will be the desired character. For example, subtracting 60 from 61H yields 1. The "N" column value of 1 is directly across from "a" in the + 60 column. In a like manner, subtracting 20 from 3A yields 1A. The "N" column value of 1A is directly across from ":" in the + 20 column.

American Standard Code for Information Interchange (ASCII)

Column Number	Column # 1 (+ 0)	Column # 2 (+ 20)	Column # 3 (+ 40)	Column # 4 (+ 60)
0	NUL	SP	@	’
1	SOH	!	A	a
2	STX	"	B	b
3	ETX	#	C	c
4	EOT	$	D	d
5	ENQ	%	E	e
6	ACK	&	F	f
7	BEL	’	G	g
8	BS	(	H	h
9	HT	)	I	i
A	LF	*	J	j
B	VT	+	K	k
C	FF	,	L	l
D	CR	-	M	m
E	SO	.	N	n
F	SI	/	O	o
10	DLE	0	P	p
11	DC1(Xon)	1	Q	q
12	DC2(tape)	2	R	r
13	DC3(Xoff)	3	S	s
14	DC4	4	T	t
15	NAK	5	U	u
16	SYN	6	V	v
17	ETB	7	W	w
18	CAN	8	X	x
19	EM	9	Y	y
1A	SUB	:	Z	z
1B	ESC	;	[	{
1C	FS	<	\	|
1D	GS	=	]	}
1E	RS	>	^	~
1F	US	?	_	DEL

I-2 ASCII Conversion Table

Index