HP 10391B
Inverse Assembler
Development Package
Reference Manual

for the HP 16500A Logic Analysis System,
and the HP 1650A/B and HP 1651A/B Logic Analyzers

HEWLETT PACKARD

©Copyright Hewlett-Packard Company 1990

Printing History

New editions are complete revisions of the manual. Update packages, which are issued between editions, contain additional and replacement pages to be merged into the manual by the customer. The dates on the title page change only when a new edition is published.

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

ii

List of Effective Pages

The List of Effective Pages gives the date of the current edition and of any pages changed in updates to that edition. Within the manual, any page changed since the last edition is indicated by printing the date the changes were made on the bottom of the page. If an update is incorporated when a new edition of the manual is printed, the change dates are removed from the bottom of the pages and the new edition date is listed in the Printing History and on the title page.

Pages   **Effective Date**

iv

Limitation of Warranty

The foregoing warranty shall not apply to defects resulting from improper or inadequate maintenance by the Buyer, Buyer-supplied software or interfacing, unauthorized modification or misuse, operation outside of the environmental specifications for the product, or improper site preparation or maintenance.

NO OTHER WARRANTY IS EXPRESSED OR IMPLIED. HEWLETT-PACKARD 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. HEWLETT-PACKARD SHALL NOT BE LIABLE FOR ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, WHETHER BASED ON CONTRACT, TORT, OR ANY OTHER LEGAL THEORY.

Assistance

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

For assistance, contact your nearest Hewlett-Packard Sales and Service Office. Addresses are provided at the back of this operating manual.

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.

Safety

This product has been designed and tested according to International Safety Requirements. To ensure safe operation and to keep the product safe, the information, cautions, and warnings in this operating manual must be heeded.

Contents

Chapter 1: **General Information**

Chapter 2: **Inverse Assembler Operation**

Chapter 3:   **Writing Inverse Assembler Code**

Chapter 4:   **Inverse Assembler Instruction Set**

Appendix A: **8085 Inverse Assembler**

Appendix B: **Microprocessors with Incomplete Status**

Appendix C: **68010 Inverse Assembler**

Appendix D: **Assembler Error Messages**

Index

1

General Information

Introduction

The HP 10391B Inverse Assembler Development Package allows you to design a custom inverse assembler for the HP 1650A/B, HP 1651A/B, HP 16510A/B, or HP 16511B Logic Analyzers. The inverse assembler runs in the logic analyzer, and converts the "ones and zeros" captured by the analyzer into mnemonics you're familiar with.

The inverse assembler routines are written on an HP Vectra, IBM-PC, or PC compatible using Hewlett-Packard's Inverse Assembly Language. This code is assembled on the PC, then downloaded over RS-232C to the disk in the logic analyzer. The inverse assembler file can then be loaded into a state analyzer to disassemble captured data.

To be successful with this software package, you should be familiar with general programming concepts and simple microprocessor operation. In addition, a basic understanding of state analysis with Hewlett-Packard logic analyzers is assumed in this manual.

This manual is organized as follows:

- Chapter 1 lists the equipment required by this software package, and describes how to install the software on the PC. This chapter also gives a step-by-step guide to using the programs provided in this package. This chapter should be read by all users.

- Chapters 2 and 3 are a tutorial on writing an inverse assembler. These chapters can be skipped if you've used Hewlett-Packard's Inverse Assembly Language (IAL) on the HP 64000 Development System.

- Chapter 4 is the language reference for the Inverse Assembler Language (IAL).

- The appendices hold listings of sample source code, and a tutorial for advanced topics.

Equipment Required

The following equipment in needed to use the HP Inverse Assembler Development Package:

1. HP Vectra, IBM-PC, or PC compatible with a minimum of 256 Kbytes of memory and MS-DOS® 2.1 or above.

2. One flexible disk drive with an internal hard disk (recommended configuration) for the PC, or two flexible disk drives.

3. RS-232C port configured as COM 1 or 2 on the PC. Recommended card:

   - HP 24540A Serial/Parallel Card, or
   - HP 24541A Dual Serial Card.

4. RS-232C printer cable. Recommended cable:

   - For 25-pin ports: HP p/n 13242-60010 or equivalent.
   - For 9-pin ports: HP 24542G or equivalent.

5. HP 1650A/B or HP 1651A/B Logic Analyzer, or HP 16500A Logic Analysis System with an HP 16510A/B or HP 16511B State/Timing Card installed.

The Inverse Assembler Development Package will require approximately 220 Kbytes of disk space.

Installing the Software

The Inverse Assembler Development Package includes one 5.25 inch flexible disk which contains all of the software for this package. This section describes how to install this software on a hard disk or another flexible disk.

What's On This Disk

The following files are included in the Inverse Assembler Development Package:

Filename	Description
INSTALL.BAT	Batch file for installing the Inverse Assembler Development Package.
ASM.EXE	Assembler for HP's Inverse Assembly Language.
AIAL	A look-up table used by the IAL assembler.
IALDOWN.EXE	Download program to put the inverse assembler on the logic analyzer disk.
I8085.S	Source code for Intel 8085 inverse assembler.
8085.BAT	Batch file for automating the assembly and download process.
8085.CMD	Input file for use with 8085.BAT
I68010.S	Source code for Motorola 68010 inverse assembler.
68010.BAT	Batch file for automating the assembly and download process.
68010.CMD	Input file for use with 68010.BAT.

Installing the Software on a Hard Disk

Use the following steps to install the software on a hard disk:

1. Insert the 5.25 inch flexible disk containing the HP 10391B software into Drive A: of the PC.

2. At the DOS prompt, change the current drive to the hard disk. For instance, if your hard disk is Drive C:, type

   C:

   at the DOS prompt.

3. At the DOS prompt, make the subdirectory where you want the Inverse Assembler Development Package installed. For instance, if you want to put the software in the subdirectory \IAL on the hard disk, type

   MKDIR IAL

   at the DOS prompt. If you are installing the software in the root directory or into a subdirectory that already exists, this step is not needed.

4. Change the current DOS directory to the directory where you want the software installed. For instance, if you want to install the software in subdirectory \IAL, type

   CD\IAL

   at the DOS prompt.

5. Type

   A:\INSTALL

   at the DOS prompt. This will begin the installation process. When the installation is completed, you will see the message

   Installation Complete!

   on the PC screen and you will be returned to a DOS prompt in the root directory of the current drive.

What the Installation Does

Running the INSTALL.BAT batch file does the following:

1. It copies these files from Drive A: to the current directory on the hard disk:

   ASM.EXE
   IALDOWN.EXE

   I8085.S
   8085.BAT
   8085.CMD

   I68010.S
   68010.BAT
   68010.CMD

2. It creates the subdirectory

   \ HP64700\ TABLES

   on your hard disk.

3. It copies the file

   AIAL

   into the subdirectory \ HP64700\ TABLES on your hard disk. This file is a table used by the ASM.EXE program. It MUST be in this subdirectory for the ASM.EXE program to execute properly.

Installing the Software on a Flexible Disk

Use the following steps to install the software on a flexible disk:

1. Insert the 5.25 inch flexible disk containing the HP 10391B into Drive A: of the PC.

2. Put the destination flexible disk into Drive B:. If the flexible disk in Drive B: is not formatted, format it at this time.

3. At the DOS prompt, change the current drive to Drive B: by typing

B:

at the DOS prompt.

4. At the DOS prompt, make the subdirectory where you want the Inverse Assembler Development Package installed. For instance, if you want to put the software in the subdirectory \ IAL on the flexible disk, type

MKDIR IAL

at the DOS prompt. If you are installing the software in the root directory or into a subdirectory that already exists, this step is not needed.

5. Change the current DOS directory to the directory where you want the software installed. For instance, if you want to install the software in subdirectory \ IAL, type

CD\ IAL

at the DOS prompt.

6. Type

A:\ INSTALL

at the DOS prompt. This will begin the installation process. When the installation is completed, you will see the message

Installation Complete!

on the PC screen and you will be returned to a DOS prompt in the root directory of Drive B:.

What the Installation Does

Running the INSTALL.BAT file does the following:

1. Copies these files from Drive A: to the current directory of the flexible disk in Drive B:

   ASM.EXE
   IALDOWN.EXE

   I8085.S
   8085.BAT
   8085.CMD

   I68010.S
   68010.BAT
   68010.CMD

2. Creates the subdirectory

   \ HP64700\ TABLES

   on the flexible disk in Drive B:.

3. Copies the file

   AIAL

   into the subdirectory B:\ HP64700\ TABLES. This file contains a table used by the ASM.EXE program. It MUST be in this subdirectory for the ASM.EXE program to execute properly.

Advanced Installation Topics

The installation instructions just presented assume you will be executing the ASM.EXE and IALDOWN.EXE files while in the subdirectory where these files are stored. If you want to execute these files while in a different subdirectory, or from a different drive, the following steps will be necessary:

1. Add the PATH statement to your AUTOEXEC.BAT file that points to the subdirectory where the Inverse Assembler Development Package is located. For example, if the software was installed in the subdirectory C:\IAL, you must add the following statement to your AUTOEXEC.BAT file:

   PATH= C:\IAL

2. Add the SET statement to your AUTOEXEC.BAT file that points to the drive where subdirectory \HP64700 was created by the INSTALL.BAT file. For example, if the subdirectory \HP64700 was created on the C: drive, the following line should be added to your AUTOEXEC.BAT file:

   SET HPTABLES= C:\HP64700\TABLES

After adding these statements to your AUTOEXEC.BAT file, reboot your PC.

These statements will allow you to execute the ASM.EXE and IALDOWN.EXE programs from any subdirectory in your PC.

Setting Up the Hardware

These steps must be performed to properly set up the hardware for the Inverse Assembler Development Package:

1. Connect the logic analyzer to the COM 1 or 2 port of the PC using the RS-232C cable specified earlier in this chapter.

2. Turn on and boot up the logic analyzer.

3. Check the RS-232C configuration.

    a. If you are using an HP 1650A or HP 1651A, press the I/O key, and select the "RS-232-C Configuration" field.

    b. If you are using an HP 1650B or HP 1651B, press the I/O key, and select "Controller connected to RS-232C."

    c. If you are using an HP 16500A, select the "RS-232C" field in the System Configuration Menu. Change the pop-up to read "RS-232C Connected to: Controller".

    d. The required RS-232C configuration for all of the logic analyzers is:

        Protocol: XON/XOFF
        Data Bits: 8
        Stop Bits: 1
        Parity: None
        Baud rate: 9600

    e. Select the "Done" field on the display when the configuration is set up correctly.

4. Set up the logic analyzer disk drive.

    a. Put a blank, unformatted flexible disk into the front disk drive of the logic analyzer.

    b. If you are using an HP 1650A/B or HP 1651A/B, press the I/O key, and select the Disk Operations field.

    c. If you are using an HP 16500A, select the "Configuration" field in the System Configuration Menu. Select the "Front Disk" field to go to the System Front Disk Menu.

    d. Format the blank disk by changing the "Load" field to "Format Disk," then selecting "Execute."

The hardware is now set up to download an inverse assembler from the PC.

Building a Custom Inverse Assembler

As an overview, here are the steps needed to build an inverse assembler for an HP 1650A/B, HP 1651A/B, HP 16510A/B, or HP 16511B logic analyzer:

1. Write the inverse assembly algorithm.

   The procedures to disassemble the information captured by the logic analyzer are written using HP's Inverse Assembly Language (IAL). The environment, syntax, and constraints of this language are presented in chapters 2 through 4, and appendices A through C of this manual.

   The software received with this product includes the source code for the Intel 8085 and Motorola 68010 inverse assemblers. These files may be helpful when first learning the Inverse Assembly Language. Listings of these files are included in appendices A and C.

   This source code can be written with almost any PC text editor. For a discussion of the requirements of the text editor, see "Choosing a Text Editor" in chapter 4.

2. Assemble the source code.

   The ASM.EXE program included on the software disk is used to convert the ASCII source code written in step 1 into a relocatable file that the logic analyzer can understand.

   For a complete discussion of the assembler syntax and options, see "Assembling the Source Code" later in this chapter.

3. Download the relocatable file to the logic analyzer.

   The IALDOWN.EXE program copies the relocatable file generated in step 2 to a file on the flexible disk of the logic analyzer. This transfer goes through the RS-232C port configured as COM 1 or 2 on the PC.

   IALDOWN.EXE prompts you for the source and destination filenames, the file description, and for the "Invasm" option required. If desired, this process can be automated using a batch file. For all the details on IALDOWN.EXE, see "Downloading the Relocatable File" later in this chapter.

4. Build the logic analyzer configuration file.

   The logic analyzer must now be told how to capture information from the target system. This configuration is entered from the analyzer's front panel and must specify the following:

   a. What channels of the logic analyzer are monitoring the address, data, and status information of the target system.

   b. Which clocks are used to latch data from the target system, and which clock edges are used.

   If desired, the configuration file can also set up symbol tables and special trace specifications.

   Once the logic analyzer configuration is completed, the inverse assembler can be loaded from the logic analyzer disk to disassemble captured states.

   For more detail on this step, see "Building the Configuration File" later in this chapter.

The rest of this chapter discusses the details of steps 2, 3, and 4. Writing the source code for an inverse assembler is the topic of chapters 2 through 4, and the appendices of this manual.

Assembling the Source Code

The source code for an inverse assembler must be converted into a format that the logic analyzer understands. The ASM.EXE program in this software package is an assembler that does this conversion.

To assemble source code, simply type

ASM < filename>

at the DOS prompt, where < filename> is the name of the file in which the source code is stored.

For example, to assemble the source code for the 8085 inverse assembler that was included with this software, type

ASM I8085.S

at the DOS prompt.

Here is what happens when ASM.EXE is executed:

1. The assembler reads the source code from the file you specified. If it cannot find the file you specified, it will generate the message:

   asm: Termination, Input source file not found.

2. The assembler then gets a special look-up table from the subdirectory \ HP64700\ TABLES. If the assembler cannot find this file or its subdirectory, it will generate the message:

   asm: Termination, Unimplemented or invalid processor name

   This message indicates that the Inverse Assembler Development package was not installed correctly, or that the line:

   SET HPTABLES= < disk> \ HP64700\ TABLES

   should be added to the PC's AUTOEXEC.BAT file. See the installation section of this chapter for details.

3.   The assembler then makes several passes through the source code to convert the source code into a format the logic analyzer can understand. The output of this step is called the relocatable object code, or relocatable code for short.

   If the assembler encounters any errors in the source code, it will stop the assembly process and generate an error message on the PC screen. A complete list of the assembler error messages is provided in appendix D.

4.   Finally, the assembler writes the relocatable code to a DOS file. The DOS filename is the original filename with ".R" appended to it.

   For example, if you typed

   ASM I8085.S

   to assemble the 8085 source code, the relocatable code will be placed in the file I8085.R.

For a complete list of the assembler error messages, see appendix D.

Assembler Options

Options can be specified when calling the assembler to make debugging the source code easier. These options are added to the ASM command when starting the assembly process.

The full syntax of the ASM program is:

ASM [-o] [-l] [-n] [-e] [-x] [-t] < file> [> list_file]

Items in the square brackets ([..]) are optional parameters for this program.

< file> is the name of the source file to be assembled.

Option Definitions   ASM recognizes the following options which must be preceded by a dash (-). Options can be concatenated after a single dash (for example, -ox)

-o   Generate a listing of the assembler results. This listing includes the following:

   a. Source statements with the associated object code,

   b. Error messages, with a pointer to the error, and

   c. A summary of errors with a descriptive list.

-l   Same as -o

-n   No listing of assembler results, except for errors (default).

-e   Not implemented for the IAL

-x   Adds a cross-reference to the listing generated by -o or -l. This cross-reference shows all line numbers that have text in the label field, including variable and constant declarations. It also shows which lines reference these labels.

-t   Causes assembly with no object code generation or relocatable file creation.

For example, ASM -ox I8085.S would assemble the I8085.S file and produce a listing and a cross-reference, as well as the relocatable file I8085.R. The cross-reference is appended to the end of the listing and would begin with:

FILE: I8085.S   CROSS REFERENCE TABLE

LINE	SYMBOL	TYPE	REFERENCES
231	LOD_STO	D	180
40	LO_INPUT_STATUS	P	644,657,672
100	ILLEGAL_OPCODE	D	196,199,387,417,447,492
340	MOVES	D	336
675	LO_WAS_OPCODE	D	672

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
     M = Multiple Defined
     P = Program (PROG)
     R = Predefined Register
     S = Special
     U = Undefined

The default output location for listings and the cross-references is the PC screen. The listing output can be rerouted to a file using the optional [> list_file]. For example, the command

ASM -ox I8085.S > I8085.L

would do the following:

1. Assemble the source code in I8085.S,

2. Create a listing of the assembler results, with a cross-reference,

3. Put the relocatable code in file I8085.R, and

4. Put the listing in file I8085.L.

To dump the listing directly to the PC's printer, use

ASM -ox I8085.S > PRN

Downloading the Relocatable File

Once the relocatable file has been constructed with the ASM.EXE program, the relocatable file should be downloaded to the disk in the logic analyzer. The download process is handled by the IALDOWN.EXE program.

To start the download, type IALDOWN at the DOS prompt. IALDOWN will do the following:

1. Ask you for the filename to store the relocatable file under on the logic analyzer's disk. You should type in the filename after the prompt, then press < Enter> .

2. Ask you for the file description that will be displayed in the logic analyzer disk menu. You should type in the file description after the prompt, then press < Enter> .

3. Ask you for the name of the relocatable file on the PC. You should type in the name of the relocatable file, including the .R extension, then press < Enter> .

4. Ask you which COM Port (1 or 2) you are using. You should type in 1 or 2, depending on which COM Port you are using to download to the HP 1650A/B, 1651A/B, or HP 16500A.

5. Ask you for the "Invasm" Field Option. Select the letter for the "Invasm" option that is appropriate for your inverse assembler, then press < Enter> .

Note 👆 The "Invasm" field is used for microprocessors with limited status information. For more information on using this field, see appendix B.

6. When the last question is answered, the program will read the relocatable file from the PC disk and download it through COM 1 or 2 to the front disk of the logic analyzer. It also downloads the filename and file description information. During the download, the logic analyzer will display:

STORING FILE TO DISK

7. When the download is complete, the PC will return you to the DOS prompt.

Here are the steps for downloading the 8085 inverse assembler included with this software package. This example assumes the relocatable file is in the same DOS subdirectory as the IALDOWN.EXE program.

```
C> IALDOWN

Logic Analyzer Filename =  I8085

Logic Analyzer File Description  _____
(must be 32 characters or less) =  INTEL 8085 INVERSE ASSEMBLER

Relocatable File on the PC =  I8085.R

COM Port to use (1 or 2)

"Invasm" Field Options:

  A =  No "Invasm" Field
  B =  "Invasm" Field with no pop-up
  C =  "Invasm" Field with pop-up.  2 choices in pop-up.
  D =  "Invasm" Field with pop-up.  8 choices in pop-up.

Select the appropriate letter (A, B, C or D): A
```

Some notes on this process:

The logic analyzer filename can be no more than 10 characters long. Valid characters for the filename are A-Z, 0-9, and _ (underscore). The first character of the filename must be an uppercase alpha character.

The logic analyzer file description must be 32 characters or fewer. Notice that there is a line on the PC screen directly above the file description. This line is 32 characters long and can be used as a "ruler" for your file description.

The relocatable filename must include the .R extension.

Shortcuts When Using IALDOWN

To save time when using IALDOWN, you may enter the answers to the IALDOWN prompts when starting the program. When answering the IALDOWN questions on the DOS command line, the following rules must be followed:

1. Separate each answer by a space,

2. Enclose the file description in quotes (".."),

3. The logic analyzer filename, the length of the file description, and the relocatable filename extension must follow the guidelines shown in the step-by-step example.

For example, typing

IALDOWN I8085 'INTEL 8085 INVERSE ASSEMBLER"I8085.R 1 A

at the DOS prompt would download the 8085 inverse assembler through COM 1 to the logic analyzer disk just like the previous step-by-step example.

Batch Files   The ASM and IALDOWN programs can be included in batch files to automate the assembly and download process. The files 8085.BAT and 68010.BAT on the software disk are simple examples of batch files for both the Intel 8085 and the Motorola 68010. The 8085.BAT file contains two lines:

ASM I8085.S
IALDOWN < 8085.CMD

The first line tells the ASM.EXE program to assemble the file I8085.S. The relocatable code will be placed in file I8085.R.

The second line of the batch file starts the IALDOWN program, with the answers to the program prompts coming from file 8085.CMD. 8085.CMD has the following five lines:

I8085
INTEL 8085 INVERSE ASSEMBLER
I8085.R
1
A

These are the answers to IALDOWN prompts.

To execute this batch file, type

8085

at the DOS prompt. The batch file will then assemble the I8085.S source code and download it to the logic analyzer disk, just like the previous step-by-step example.

Note ☝   If the answers to IALDOWN are placed in a separate file, as shown in this example, each line of the "answer" file must be terminated with a < CR> < LF> (carriage return, line feed).

Building the Configuration File

The logic analyzer's configuration file stores the set-up needed by the instrument to capture the states from the target system. A few rules must be followed when configuring the logic analyzer for an inverse assembler to work properly.

- The inverse assembler can only be used in State analysis.

- The following labels MUST be defined in the Format Menu:

Label	Description
ADDR	Label for the logic analyzer channels connected to the target system Address Bus.
DATA	Label for the logic analyzer channels connected to the target system Data Bus.
STAT	Label for the logic analyzer channels connected to the target system Status Lines. These lines should indicate what kind of bus cycle was captured by the logic analyzer.

- For the HP 16511B, the following labels must also be defined in the Format menu:

Label	Description
ADDR_B	Label for the logic analyzer channels connected to the target system auxiliary address bus for the HP 16511B.
DATA_B	Label for the logic analyzer channels connected to the target system auxiliary data bus for the HP 16511B.

The ADDR, DATA, ADDR_B, DATA_B, and STAT labels are used with specific communication variables in the Inverse Assembly Language. The relationship between the ADDR, DATA, ADDR_B, DATA_B, and STAT labels and the communication variables is shown in the table below:

Label	Is Linked with the Communication Variable
ADDR	INITIAL_ADDRESS and INPUT_ADDRESS
DATA	INITIAL_DATA and INPUT_DATA
STAT	INPUT_STATUS
ADDR_B	INPUT_ADDR_B
DATA_B	INPUT_DATA_B

Note 👉 The ADDR_B and DATA_B variables are NOT used in the HP 1650A/B, HP 1651A/B, and HP 16510A/B Logic Analyzers.

Because of the link between the labels and the communication variables, the ADDR, DATA, and STAT labels must be defined in the Format menu before the inverse assembler is loaded from the logic analyzer disk.

The following steps may be used as a checklist when building the logic analyzer configuration file:

1. In the logic analyzer Configuration menu, set up one of the analyzers to be a State Analyzer.

2. Assign the pods to the State Analyzer that will capture the signals in the target system. Each pod has 16 channels; assign the number of pods necessary to acquire all of the needed signals.

3. In the State Format menu, enter the labels ADDR, DATA, and STAT, and labels ADDR_B and DATA_B for the HP 16511B. Assign the appropriate channels to each label.

4. Set up the clock specification to properly capture states from the target system. The signals captured on the clock edge must be set up 10 ns before the clock edge, and must hold typically 0 ns after the clock edge. Refer to the reference manual for your logic analyzer for exact specifications. Verify the timing of the state clock using timing diagrams for the target system.

5. In the logic analyzer Format menu, set the Clock Period field to correspond to how fast you are clocking in data. For more information on the Clock Period field, refer to the reference manual for your logic analyzer.

6. If desired, enter additional labels, define Symbol Tables, or set up a Trace Specification for the logic analyzer.

Linking the Inverse Assembler and the Configuration File

A link can be established between the inverse assembler and the configuration file. With a link established, loading the configuration file from the disk will automatically load the inverse assembler at the same time. Here is how to link the configuration file with the inverse assembler:

1. Set up the logic analyzer to capture states from the target system. This can be done manually by entering the configuration from the front panel, or by loading a configuration file from the disk. The configuration must follow the rules listed on page 1-23.

2. Insert the inverse assembler disk into the disk drive of the logic analyzer. Load the inverse assembler file from this disk into the analyzer you have configured for state analysis.

3. Go to the State Listing.

4. Go to the field under the DATA label that contains "Hex" and select this field. A pop-up menu will appear on the logic analyzer screen.

5. The pop-up will have seven fields. Select the field labeled "Invasm" and the inverse assembler will display mnemonics in place of hex data.

6. Store the configuration file to the inverse assembler disk. This establishes a link between the configuration file and the inverse assembler. When the configuration file is loaded it will automatically load the inverse assembler.

Putting It All Together

This chapter has discussed the steps needed to convert IAL source code to an inverse assembler used with a logic analyzer. This final example will put all the steps together.

In this section, you will actually assemble source code that was provided with the software package and put it on the logic analyzer disk. You will also create the configuration file for this inverse assembler and link the configuration file to inverse assembler. The example used is the inverse assembler for the Motorola 68010 microprocessor.

1. If you have not already done so, install the Inverse Assembler Development Package on your PC.

2. Connect the hardware as described in this chapter. When finished, you should have a blank, formatted disk in the logic analyzer disk drive.

3. Assemble the 68010 source code that was provided with the software package. Type

   ASM I68010.S

   at the DOS prompt. This will generate a relocatable file called I68010.R

4. Download the relocatable code to the logic analyzer. Type

IALDOWN

at the DOS prompt. When prompted by the IALDOWN program, answer the questions as follows:

Logic Analyzer Filename = I68010

Logic Analyzer File Description _____
(must be 32 characters or less) = MOTOROLA 68010 INVERSE ASSEMBLER

Relocatable File on the PC = I68010.R

COM Port to use (1 or 2)

"Invasm" Field Options:

  A = No "Invasm" Field
  B = "Invasm" Field with no pop-up
  C = "Invasm" Field with pop-up. 2 choices in pop-up.
  D = "Invasm" Field with pop-up. 8 choices in pop-up.

Select the appropriate letter (A, B, C or D): B

This will download the inverse assembler to the logic analyzer disk. When completed, you will be returned to the DOS prompt on the PC.

5. Manually set up the logic analyzer Configuration and Format menus as shown below.

```
┌─────────────────────────────────────────────────────────────────┐
│  ( State/Timing E )  (Configuration)              (Print)  ( Run )│
│   ┌─ Analyzer 1 ──────┐    ┌─ Analyzer 2 ──────┐                 │
│   │ Name: [  68010  ] │    │                   │                 │
│   │                   │    │ Type: [   Off   ] │                 │
│   │ Type: [  State  ] │    │                   │    ─────── ─    │
│   │                   │    │                   │ Unassigned Pods │
│   │    ┌─ Pod 1 ──┐   │    └───────────────────┘   ┌─ Pod 4 ──┐  │
│   │    │ ──────── │   │                             │ ──────── │  │
│   │    ┌─ Pod 2 ──┐   │                             ┌─ Pod 5 ──┐  │
│   │    │ ──────── │   │                             │ ──────── │  │
│   │    ┌─ Pod 3 ──┐   │                                          │
│   │    │ ──────── │   │                                          │
│   └───────────────────┘                                          │
│                                                                  │
└─────────────────────────────────────────────────────────────────┘
```

```
┌─────────────────────────────────────────────────────────────────┐
│  ( State/Timing E )  ( Format 1 )                 (Print)  ( Run )│
│  (Clock Period)(      Clock      )                    (Symbols)   │
│  (  > 60 ns   )( J↑             )                                 │
│              Pod E3          Pod E2          Pod E1              │
│          ┌─────────┐     ┌─────────┐     ┌─────────┐            │
│          │   TTL   │     │   TTL   │     │   TTL   │            │
│          ┌─────────┐     ┌─────────┐     ┌─────────┐            │
│          │  Clock  │     │  Clock  │     │  Clock  │            │
│  (Label) Pol  ──────────   ──────────   ──────────             │
│              15 ... 87 .... 0  15 ... 87 .... 0  15 ... 87 .... 0│
│  ┌────┐ ┌─┐                                                     │
│  │ADDR│ │+│ ********,.......  ****************  ................ │
│  ┌────┐ ┌─┐                                                     │
│  │DATA│ │+│ ................  ................  ****************│
│  ┌────┐ ┌─┐                                                     │
│  │STAT│ │+│ ........,********  ................  ................│
│  ┌────┐                                                         │
│  │ Off│                                                         │
│  ┌────┐                                                         │
│  │ Off│                                                         │
│  ┌────┐                                                         │
│  │ Off│                                                         │
│  ┌────┐                                                         │
│  │ Off│                                                         │
│  ┌────┐                                                         │
│  │ Off│                                                         │
│  └────┘                                                         │
└─────────────────────────────────────────────────────────────────┘
```

6. Go to the Front Disk Drive menu and load the inverse assembler into the State analyzer that you just set up.

7. Go to the State Listing menu and change "Hex" under the DATA label to "Invasm." You should see a screen that looks like this:

```
┌──────────────────────────────────────────────────────────────┐
│ ( State/Timing E )  ( Listing 1 )   ( Invasm )  ( Print ) ( Run )│
│ ┌──────────┐                                                   │
│ │ Markers  │                                                   │
│ │   Off    │                                                   │
│ └──────────┘                                                   │
│                                                                │
│  ┌───────┐  ┌──────┐ ┌──────────────────────┐ ┌──────┐        │
│  │ Label>│  │ ADDR │ │    68010 Mnemonic    │ │ STAT │        │
│  └───────┘  └──────┘ └──────────────────────┘ └──────┘        │
│  ┌───────┐  ┌──────┐ ┌──────────────────────┐ ┌──────┐        │
│  │ Base> │  │ Hex  │ │         hex          │ │ Hex  │        │
│  └───────┘  └──────┘ └──────────────────────┘ └──────┘        │
│                                                                │
│                                                                │
│  ┌───────────┐                                                 │
│  │           │                                                 │
│  └───────────┘                                                 │
│                                                                │
└──────────────────────────────────────────────────────────────┘
```

8. Store the configuration to the logic analyzer disk. Use the filename

C68010

and the description

MOTOROLA 68010 CONFIGURATION

If you want to test this inverse assembler, connect the logic analyzer to a 68000 or 68010 target system using the general purpose probes and grabbers. Connect the signals as follows:

Signal Name	Logic Analyzer Pod:Bit	Signal Name	Logic Analyzer Pod:Bit
A0	Pod 2:Bit 0	D0	Pod 1:Bit 0
A1	Pod 2:Bit 1	D1	Pod 1:Bit 1
A2	Pod 2:Bit 2	D2	Pod 1:Bit 2
A3	Pod 2:Bit 3	D3	Pod 1:Bit 3
A4	Pod 2:Bit 4	D4	Pod 1:Bit 4
A5	Pod 2:Bit 5	D5	Pod 1:Bit 5
A6	Pod 2:Bit 6	D6	Pod 1:Bit 6
A7	Pod 2:Bit 7	D7	Pod 1:Bit 7
A8	Pod 2:Bit 8	D8	Pod 1:Bit 8
A9	Pod 2:Bit 9	D9	Pod 1:Bit 9
A10	Pod 2:Bit 10	D10	Pod 1:Bit 10
A11	Pod 2:Bit 11	D11	Pod 1:Bit 11
A12	Pod 2:Bit 12	D12	Pod 1:Bit 12
A13	Pod 2:Bit 13	D13	Pod 1:Bit 13
A14	Pod 2:Bit 14	D14	Pod 1:Bit 14
A15	Pod 2:Bit 15	D15	Pod 1:Bit 15
A16	Pod 3:Bit 8	R/W	Pod 3:Bit 0
A17	Pod 3:Bit 9	LDS	Pod 3:Bit 1
A18	Pod 3:Bit 10	UDS	Pod 3:Bit 2
A19	Pod 3:Bit 11	VMA	Pod 3:Bit 3
A20	Pod 3:Bit 12	FC0	Pod 3:Bit 4
A21	Pod 3:Bit 13	FC1	Pod 3:Bit 5
A22	Pod 3:Bit 14	FC2	Pod 3:Bit 6
A23	Pod 3:Bit 15	BGACK	Pod 3:Bit 7
GND	Pod 2:Gnd	AS	Pod 1:J Clk
GND	Pod 3:Gnd	GND	Pod 1:Gnd

Press "RUN" on the logic analyzer to capture bus activity. Under the inverse assembler field, you will see the value of the data bus, plus the kind of bus cycle captured by the logic analyzer.

To see the 68010 mnemonics, do the following:

1. Identify a state that you know contains the first state of an opcode fetch.

2. Scroll this state to the top line of the screen.

3. Select the "Invasm" field at the top of the display.

The inverse assembler will display 68010 mnemonics for the entire screen.

® MS-DOS is a registered trademark of Microsoft Corporation.

2

Inverse Assembler Operation

Introduction

This chapter provides an overview of the inverse assembly process. It explains how the logic analyzer captures data and builds a display, and shows how an inverse assembler fits into this process. It also walks you through the steps needed to properly inverse assemble a microprocessor instruction.

This overview discusses concepts of inverse assembly without going into detail about the actual Inverse Assembly Language syntax. Syntax details are introduced in chapter 3, and a complete language reference is provided in chapter 4.

Inverse Assembler Operation

The main function of the inverse assembler is to decode data captured by the logic analyzer. This is done as follows:

1. Using the logic analyzer system software, you specify what the logic analyzer should trigger on and what states should be stored. This information is called the "Trace Specification."

2. When the "RUN" key is pressed, the system software loads the acquisition hardware with the trace specification, and starts the acquisition. The trace will continue until the trace specification is satisfied or until you stop the acquisition.

3. Once the acquisition has stopped, the Address, Data, and Status information captured by the logic analyzer hardware is passed to the system software routines that are used to build the logic analyzer display.

4. For each line on the screen, the system software does the following:

   a. Puts the Address information into the Display buffer.

   b. Calls the inverse assembler to convert the captured binary Data information into text. Then it places this text into the Display buffer.

   c. Puts additional labels, including the Status information and the Time count, into the Display buffer.

5. The system software displays the contents of the Display buffer and returns control of the instrument to the front panel.

This process is shown in Figure 2-1.
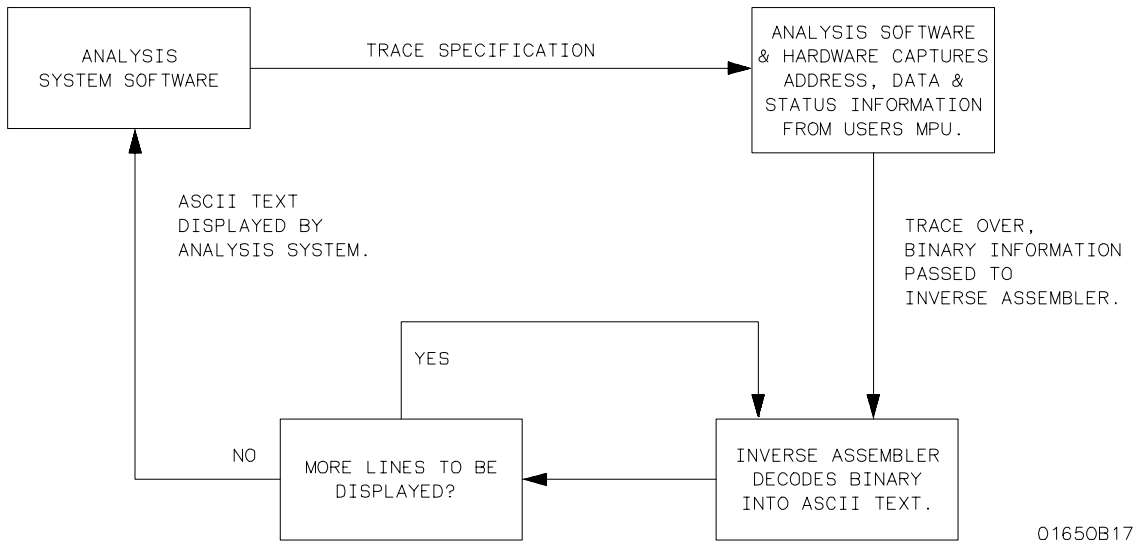
```
┌─────────────────────┐                               ┌─────────────────────┐
│                     │    TRACE SPECIFICATION        │  ANALYSIS SOFTWARE  │
│      ANALYSIS        ├──────────────────────────────►│ & HARDWARE CAPTURES │
│  SYSTEM SOFTWARE     │                               │   ADDRESS, DATA &   │
│                     │                               │ STATUS INFORMATION  │
└─────────────────────┘                               │  FROM USERS MPU.    │
         ▲                                             └─────────────────────┘
         │                                                        │
    ASCII TEXT                                       TRACE OVER,  │
    DISPLAYED BY                                     BINARY INFORMATION
    ANALYSIS SYSTEM.                                 PASSED TO
                                                     INVERSE ASSEMBLER.
         │           ┌──────────────────────┐                     │
         │      YES  │                      │                     │
         │           │                      ▼                     ▼
         │  NO    ┌──┴────────────────┐   ┌─────────────────────┐
         └────────┤  MORE LINES TO BE  │◄──┤  INVERSE ASSEMBLER  │
                  │     DISPLAYED?     │   │   DECODES BINARY    │
                  └────────────────────┘   │   INTO ASCII TEXT.  │
                                           └─────────────────────┘
                                                          O165OB17
```

Figure 2-1. How an Inverse Assembler is Used

Inverse Assembly Process

To show the inverse assembly process, we will use a very simple program for the 8085 microprocessor, shown in example 1.

Example 1:

Location	Object Code	Line	Source Line
		1 "8085"	
		2	ORG 0100H
0100	32 0601	3	STA 0106H
0103	C3 0001	4	JMP 0100H
0106	00	5	NOP

During execution, the STA instruction on line 3 stores the contents of the accumulator into the memory location at address 106H. In the next instruction, program control jumps back to STA and the process repeats in a never ending loop.

If the bus cycles generated by this program were captured by a logic analyzer, the display shown in Example 2 would be expected:

Example 2:

Line #	ADDR Hex	DATA Hex	STAT Bin	Time Rel
0000	0100	32	0011	
0001	0101	06	0010	2.00 uS
0002	0102	01	0010	1.52 uS
0003	0106	00	0001	1.48 uS
0004	0103	C3	0011	1.52 uS
0005	0104	00	0010	2.00 uS
0006	0105	01	0010	1.48 uS
0007	0100	32	0011	1.52 uS
0008	0101	06	0010	2.00 uS
0009	0102	01	0010	1.52 uS
0010	0106	00	0001	1.48 uS

etc.

If an 8085 Inverse Assembler was used to interpret the captured information, the following screen would be seen:

Example 3:

Line #	ADDR Hex	8085 Mnemonic Hex	STAT Bin	Time Rel
0000	0100	STA 0106	0011	
0001	0101	06 memory read	0010	2.00 uS
0002	0102	01 memory read	0010	1.52 uS
0003	0106	00 memory write	0001	1.48 uS
0004	0103	JMP 0100	0011	1.52 uS
0005	0104	00 memory read	0010	2.00 uS
0006	0105	01 memory read	0010	1.48 uS
0007	0100	STA 0106	0011	1.52 uS
0008	0101	06 memory read	0010	2.00 uS
0009	0102	01 memory read	0010	1.52 uS
0010	0106	00 memory write	0001	1.48 uS

etc.

The "8085 Mnemonic" field in Example 3 replaced the "DATA" field in Example 2.

To build the display in Example 3, the system software formats the screen and displays the "Line # " and "ADDR" information. Then the inverse assembler is called to fill in the "8085 Mnemonic" section. Next, the "STAT" and "Time" information is filled in by the system software. This process is repeated for every line on the display.

For the purposes of discussion, we will just look at the STA instruction found in line 3 of Example 1. The 8085 is an 8-bit microprocessor with a 16-bit addressing range. The STA instruction stores the value of the 8-bit accumulator into the memory location specified in the operand field. This instruction gets broken down into the following object code:

8-bit opcode	16-bit address
XX	XXXX

The analysis hardware would have captured the following information when tracing the execution of this instruction:

State	ADDR
0	Address of opcode
1	Address of low order address byte
2	Address of high order address byte
3	Address of store (states 2+ 3)
4	Address of next instruction

State	DATA	STAT
0	Opcode value	Opcode fetch
1	Low order address byte	Operand memory read
2	High order address byte	Operand memory read
3	Accumulator data	Memory write
4	Next opcode value	Opcode fetch

In our example, the actual data captured by the analysis hardware follows:

State	ADDR	DATA	STAT
0	0100	32	Opcode fetch
1	0101	06	Operand memory read
2	0102	01	Operand memory read
3	0106	00	Memory write
4	0103	C3	Opcode fetch

The first step for the inverse assembler is to check the status of the state being disassembled. If the state is an opcode fetch, the inverse assembler should decode it into the appropriate mnemonic. If the state is not an opcode fetch, it should display the captured data and the type of cycle captured.

In this example, the first state captured is an opcode fetch with 32H on the data bus. The inverse assembler should branch to an area of the code that will determine the mnemonic for this opcode. For the 8085, 32H is the object code for the STA instruction, so the inverse assembler will place the following in the output display buffer:

STA

The 8085 STA instruction is a multi-byte instruction. The two bytes following the opcode indicate the 16-bit address where the contents of the accumulator are to be stored in memory. The inverse assembler must be written to look forward to the next two states to properly decode the destination address of the STA instruction.

The first state following the STA instruction is the lower byte of the destination address; the second state is the upper byte. The inverse assembler will read the data in these two states, combine the results into a single 16-bit quantity and display this address after STA in the display buffer. After doing this, the output display buffer will contain:

STA 0106

The inverse assembler has now decoded the instruction back into its mnemonic form. Next, it returns to the system software to display the assembly instructions, as follows:

Line #	ADDR Hex	8085 Mnemonic Hex	STAT Bin	Time Rel
0000	0100	STA 0106	0011	

The inverse assembler will then be called for the next three analysis states that have been used to complete the STA instruction. It checks to see if these states are opcode fetches, and since they are not, the inverse assembler will simply display the status of these states. The first two states after the opcode represent the address of the operand. They are labeled as "read" states. The third state is the result of the STA instruction and is labeled as a "write" operation. Note that the address and mnemonic information show the actual data that was written and the location of the write operation.

Line #	ADDR Hex	8085 Mnemonic Hex	STAT Bin	Time Rel
0000	0100	STA 0106	0011	
0001	0101	06 memory read	0010	2.00 uS
0002	0102	01 memory read	0010	1.52 uS
0003	0106	00 memory write	0001	1.48 uS

This completes the task the inverse assembler would normally be expected to perform in displaying the results of the STA instruction execution. The inverse assembler would be called again for the fifth analysis state. In this example, it is the jump (JMP) instruction.

Summary

The following points are the key points of this chapter:

- An inverse assembler is used to convert the data captured by the logic analyzer into text on the logic analyzer display. Often, this text is microprocessor mnemonics.

- The inverse assembler is called once for each state to be displayed on the screen.

- The first thing an inverse assembler should do when called is check the status information for that state. If the status indicates an opcode fetch, the inverse assembler should decode the state into the proper mnemonic. If the state is not an opcode fetch, the inverse assembler should display the captured data and the cycle type.

- When decoding opcode fetches, it may be necessary to look forward in the acquired data to completely decode a multi-byte instruction.

3

Writing Inverse Assembler Code

Introduction	The previous chapter presented on overview of what an inverse assembler is, and what kind of tasks it needs to perform. This chapter will detail how to implement these tasks with Hewlett-Packard's Inverse Assembly Language (IAL).

Inverse assemblers written with the IAL will execute on HP 1650A/B, HP 1651A/B, HP 16510A/B, and HP 16511B Logic Analyzers. The first part of this chapter describes the environment inside the logic analyzer where an inverse assembler executes. The second half of this chapter will discuss some of the IAL instructions that can be used to perform specific inverse assembly tasks.

A complete description of IAL instructions and syntax is provided in chapter 4. |

IAL Environment	Hewlett-Packard's Inverse Assembly Language defines a simple environment where an inverse assembler is executed. The major elements of this environment are:

1. The logic analyzer acquisition memory,

2. The ACCUMULATOR,

3. Pre-defined communication variables which give the ACCUMULATOR access to the acquisition memory,

4. User-defined variables for temporary storage during inverse assembly, and

5. The output display buffer.

The logic analyzer acquisition memory is part of the logic analyzer hardware. All other parts of the IAL environment are part of the system software that build up this virtual machine. A block diagram of the environment is shown in figure 3-1 on page 3-5.

The IAL environment can be thought of as a simple pseudo-processor with a single accumulator. The inverse assembly routine runs on this pseudo-processor. The inverse assembler reads captured information from the acquisition memory, decodes this information into text (such as a microprocessor's mnemonics) and places the text into the output display buffer. As mentioned in the previous chapter, the inverse assembler will be called once for each state to be displayed.

Instructions coded in IAL are executed much like those in assembly language. The instructions are executed one at a time and in sequential order. The GOTO and CALL instructions are used to branch to another area of the inverse assembler. IF and CASE statements can be used for conditional testing.

Similar to simple microprocessor systems, the inverse assembler code is stored in a separate code space. A program counter points to the current instruction being executed by the IAL interpreter. A stack pointer and stack are used in conjunction with subroutine calls. The code space, program counter, stack pointer, and stack cannot be directly accessed by the IAL instructions.

The Logic Analyzer Acquisition Memory

The logic analyzer's memory is an array of bits that is up to 80 channels wide (160 channels wide on the HP 16511B) and up to 1024 states deep. The channels are grouped under specific labels in the logic analyzer's Format menu: ADDR, DATA, ADDR_B, DATA_B, and STAT. These labels are used to identify to the inverse assembler which channels were used for capturing the Address, Data, and Status information for each state captured.

The information captured by the logic analyzer is treated as Read Only Memory by the inverse assembler.

The Accumulator

The heart of the IAL environment is the ACCUMULATOR. The ACCUMULATOR is a single 32-bit register that can be used for the following:

- It can access the logic analyzer acquisition memory through pre-defined communication variables.

- It can read and write variables defined by the user.

- Variables read into the ACCUMULATOR can be written to the output display buffer.

- It can be operated on by arithmetic and logical operations.

- It can be tested using IF and CASE instructions.

Memory-to-memory operations are not supported by the inverse assembly language. To move contents from one variable to another, the contents must first be loaded into the ACCUMULATOR from the source location, then stored to the destination location.

IAL Variables

There are two types of variables used in the inverse assembly language:

1. Communication variables provided by the IAL interpreter. Through these variables, the inverse assembler can read the address, data, and status information captured by the logic analyzer. These variables can also be used to tag states that have already been used for inverse assembly, and can also indicate which captured states contain the first byte of an instruction fetch. Table 3-3 at the end of this chapter lists the communication variables.

2. Variables declared by an inverse assembler using the "VAR" pseudo. These can be used for holding temporary values or setting flags for internal use.

All variables and the ACCUMULATOR are treated as 32-bit integers, as shown in the following diagram:

MSB   LSB

Bit 31   Bit 0

The ADD and SUBTRACT instructions perform two's complement arithmetic on all variables, which means values can be added and subtracted to get a correct positive or negative result. The highest order bit (31) will be 0 for positive and 1 for negative results. This bit can be tested to see if the value is positive or negative.

All compares done with the "IF" instruction are unsigned. Take care when comparing variables. Since negative numbers will have the most significant bit set in two's complement, the "IF" statement will treat negative numbers as greater than positive numbers. For example, -1 (0FFFFFFFFH in two's complement) will be treated as larger than 1 (00000001H in two's complement).

The Output Display Buffer

The output display buffer is a write-only memory where ASCII text and numbers are written for each state of the inverse assembly display. Each state can print up to four lines with up to 64 characters per line.

Developing an Inverse Assembler

The development process for an inverse assembler uses the following steps:

1. Write the inverse assembler source code.

2. Assemble the source code into relocatable code.

3. Download the inverse assembler to a logic analyzer disk.

4. Load the inverse assembler into the logic analyzer.

5. Test and verify the inverse assembler's operation.

This section describes how to write the inverse assembler source code (Step 1 above) that can be properly assembled by the assembler. Steps 2 through 5 are discussed in other sections of this manual. The examples given in this chapter are good examples of the IAL syntax. Refer to chapter 4 for a complete definition of the IAL syntax and instructions.
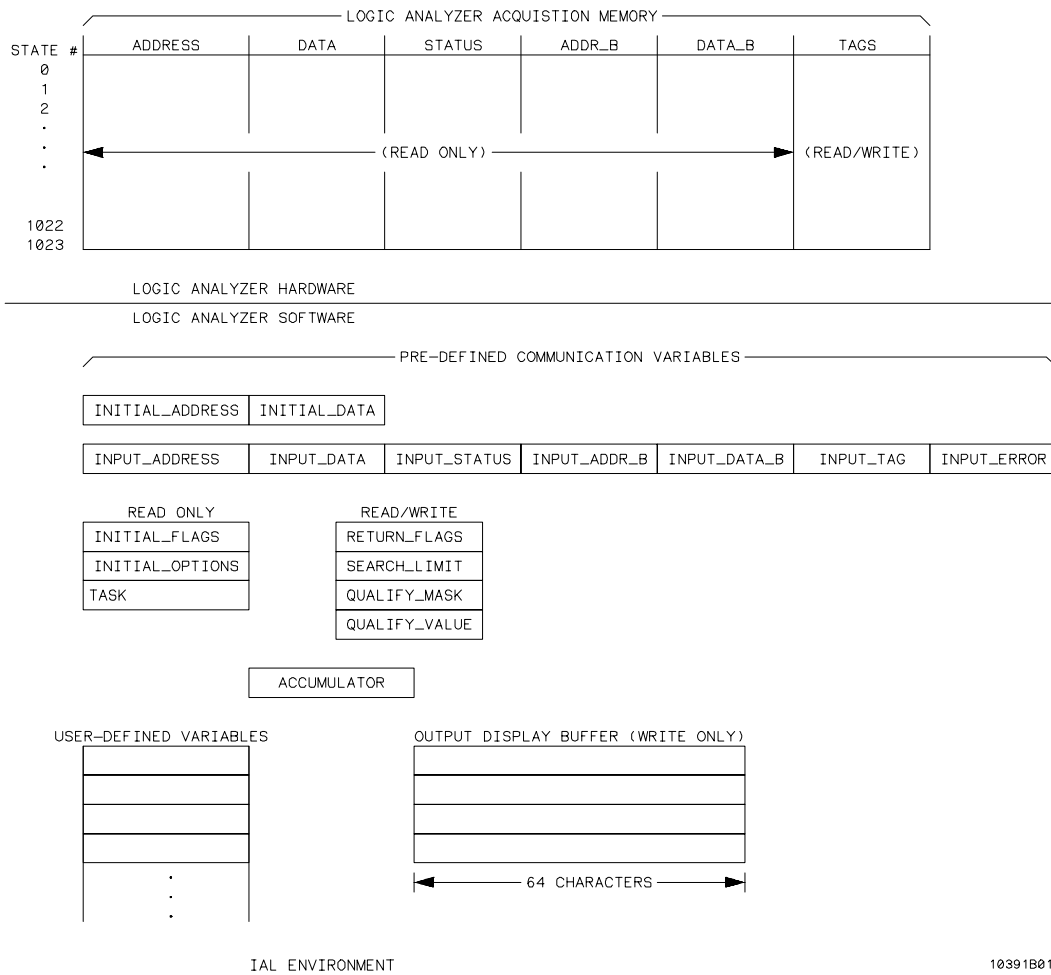
Figure 3-1. The IAL Environment

A Simple Inverse Assembler

The source code for a simple, yet complete, inverse assembler is shown below:

```
1  "IAL"
2
3        OUTPUT "Inverse assembler not present"
4        RETURN
```

If this code were assembled, downloaded to a logic analyzer disk, then loaded into the logic analyzer, typical results would look like:

Label	>	ADDR	DATA	STAT	Time
Base	>	Hex	Invasm	Bin	Rel
+ 0000		0396	Inverse assembler not present	0110	
+ 0001		0397	Inverse assembler not present	0110	680 nS
+ 0002		6106	Inverse assembler not present	1100	920 nS
+ 0003		03C5	Inverse assembler not present	0110	7.24 uS
+ 0004		03C6	Inverse assembler not present	0110	680 nS
+ 0005		6008	Inverse assembler not present	1101	920 nS
+ 0006		0396	Inverse assembler not present	0110	496.6 uS
+ 0007		0397	Inverse assembler not present	0110	680 nS
+ 0008		6006	Inverse assembler not present	1100	920 nS
+ 0009		03C5	Inverse assembler not present	0110	7.24 uS
+ 0010		03C6	Inverse assembler not present	0110	640 nS
+ 0011		6008	Inverse assembler not present	1101	920 nS
+ 0012		0396	Inverse assembler not present	0110	524.8 uS
+ 0013		0397	Inverse assembler not present	0110	640 nS
+ 0014		6006	Inverse assembler not present	1100	920 nS
+ 0015		03C5	Inverse assembler not present	0110	7.28 uS

This screen was generated, starting with line number 0000, as follows:

1. The logic analyzer system software displayed the line number of the state and the information captured under the ADDR label.

2. The system software called the inverse assembler. In this case, the instruction in line 3 of the source code puts the "Inverse assembler not present" message in the output display buffer.

3. When the RETURN instruction was executed in line 4, control was passed back to the system software.

4. The system software displayed the information captured under the STAT and Time labels.

Steps 1 through 4 were repeated 16 times until the logic analyzer screen was filled.

This example illustrates some important principles:

1. The inverse assembler will be called once for every state to be displayed on the screen.

2. Executing a RETURN instruction when the inverse assembler is not in a subroutine will end the inverse assembly for the current state, and will return control back to the logic analyzer system software.

Most inverse assemblers are more complex than the previous example. The following sections will discuss how to implement additional inverse assembly tasks. To examine a complete inverse assembler, see appendix A for an Intel 8085 inverse assembler, and appendix C for a Motorola 68010 inverse assembler.

Reading Acquisition Memory

An inverse assembler is used to convert the data captured by the logic analyzer into a display that is meaningful to the user. Usually, microprocessor mnemonics are displayed.

The first step, then, of an inverse assembler is to read the information captured by the logic analyzer. The inverse assembler routines can then decode this information into the appropriate text.

The following section demonstrates how an inverse assembler can read the logic analyzer's acquisition memory. As an illustration, the STA instruction for the Intel 8085 microprocessor will be decoded. An overview of the decoding process for this instruction was presented in chapter 2. If necessary, review chapter 2 at this time.

Decoding the STA Instruction

When the Intel 8085 executes the STA instruction, a logic analyzer captures the following raw data:

Line #	ADDR Hex	DATA Hex	STAT Bin	Time Rel
0000	0100	32	0011	
0001	0101	06	0010	2.00 uS
0002	0102	01	0010	1.52 uS
0003	0106	00	0001	1.48 uS
0004		.		
		.		
		.		

ADDR is the label assigned to the group of logic analyzer channels probing the 8085 address bus, DATA is the label for the channels on the 8085 data bus, and STAT is the label for the logic analyzer channels probing the 8085 status and control lines. STAT will be used by the inverse assembler to determine what kind of 8085 bus cycle was captured by the logic analyzer. These labels were set up in the logic analyzer's Format menu.

Line 0000 of this acquisition contains the object code for the STA instruction. Line 0001 has the lower byte of the STA destination address. Line 0002 has the upper byte of the destination address and line 0003 holds the actual write operation of the STA instruction.

If the logic analyzer were to inverse assemble this captured information, here is what would happen:

As the logic analyzer builds the display, it will print 0000 under the "Line # " column and 0100 under the ADDR label. At this point, the system software calls the inverse assembler.

When an inverse assembler is called, the system software initializes pre-defined variables called "Communication Variables." These variables allow the inverse assembler to read the acquisition memory. The following communication variables get initialized by the system software when an inverse assembler is called:

      INITIAL_ADDRESS
      INITIAL_DATA
      INPUT_ADDRESS
      INPUT_DATA
      INPUT_ADDR_B
      INPUT_DATA_B
      INPUT_STATUS
      INPUT_ERROR
      INPUT_TAG

Refer to table 3-3 on page 3-30 for a description of these communication variables.

Note 👉 INPUT_ADDR_B and INPUT_DATA_B are only used with the HP 16511B Logic Analyzer.

When decoding line 0000, the communication variables will be initialized as follows:

Communication Variable	Initialized to:
INITIAL_ADDRESS	0100H
INITIAL_DATA	32H
INPUT_ADDRESS	0100H
INPUT_DATA	32H
INPUT_STATUS	0011B
INPUT_ERROR	0
INPUT_TAG	0

Using the instruction

LOAD INPUT_STATUS

will bring the STAT value for line 0000 (0011B) into the ACCUMULATOR. The inverse assembler can then test the ACCUMULATOR to see if this state is an opcode fetch. In this inverse assembler, 0011B is the status for an opcode fetch. The inverse assembler can now begin decoding the opcode.

The instruction

LOAD INPUT_DATA

will bring the DATA value for line 0000 (32H) into the ACCUMULATOR. The inverse assembler can decode this into the STA instruction using a lookup table. When the opcode has been properly identified in the lookup table, the instruction

OUTPUT 'STA"

will place the string "STA" into the output display buffer.

Decoding the Destination Address of the STA Instruction

As discussed in chapter 2, STA is a multi-byte instruction. The destination address for this instruction is held in the next two memory locations after the STA object code. The 8085 will fetch the object code for the STA instruction, then fetch the contents of the next two memory locations to determine the destination address. To properly inverse assemble the STA instruction, the inverse assembler must duplicate the microprocessor's operation. It will have to look ahead of the current state being inverse assembled to the states that contain the STA instruction's destination address.

The Inverse Assembly Language provides the INPUT command to allow an inverse assembler to read information from any state in the acquisition memory. The INPUT command is used to "point" to the desired state. When that state is located in the acquisition memory, the following communication variables get initialized with the contents of that state:

    INPUT_ADDRESS
    INPUT_DATA
    INPUT_ADDR_B
    INPUT_DATA_B
    INPUT_STATUS
    INPUT_ERROR
    INPUT_TAG

Refer to table 3-3 on page 3-30 for a description of these communication variables.

The INPUT instruction can search in one of two modes, absolute or relative. In the absolute mode, an address is passed to the INPUT instruction. The INPUT instruction then searches forward through the acquisition memory until it finds a state whose ADDR sample matches the address passed to the instruction.

In the relative mode, the number of states is passed to the INPUT instruction. The INPUT instruction then goes forward or backwards through acquisition memory the specified number of states. In both cases, the INPUT search begins at the original state being disassembled. When the desired state is found, the communication variables listed on the previous page are loaded with the values in that state.

Here is how the INPUT instruction is used by the Intel 8085 inverse assembler to decode the destination address of the STA instruction:

1. The destination address of the STA instruction is in the two memory locations following the STA object code. The first step, then, is to store the address of the STA object code in a variable that can be modified later by the inverse assembler.

   Since the inverse assembler is currently decoding the state with the STA object code, the following instructions will put the address of the STA object code in a variable for later use:

LOAD INPUT_ADDRESS   ;Load the ACCUMULATOR with the
                            ;address of the STA object code

STORE NEW_ADDRESS   ;Store the contents of the
                            ;ACCUMULATOR into the variable
                            ;NEW_ADDRESS

   In this example, the STA instruction was fetched from address 0100H. These instructions will put the value 0100H into NEW_ADDRESS.

2. To calculate where the low byte of the destination address is located, add one to the address of the STA object code:

INCREMENT NEW_ADDRESS   ;Adds one to NEW_ADDRESS

NEW_ADDRESS now holds the value 0101H.

3. The address calculated in step 2 can be used with the INPUT instruction to point to the state that has the low byte of the destination address, as follows:

INPUT ABS,NEW_ADDRESS   ;Point to the state with the value of
                             ;NEW_ADDRESS in the ADDR column.

When the correct state is found, the INPUT instruction will initialize the following communication variables:

Communication Variable	Initialized to:
INPUT_ADDRESS	0101H
INPUT_DATA	06H
INPUT_STATUS	0010B
INPUT_ERROR	0
INPUT_TAG	0

4. The value of DATA for this state should be saved to calculate the destination address.

LOAD INPUT_DATA   ;Load the ACCUMULATOR with the low
                   ;byte of the destination address

STORE LOW_BYTE   ;Store the contents of the
                   ;ACCUMULATOR into the variable
                   ;LOW_BYTE

For this example, these instructions will put the value 06H into LOW_BYTE.

5. To calculate where the high byte of the destination address is located, add one more to the address of the STA object code:

INCREMENT NEW_ADDRESS   ;Adds one to NEW_ADDRESS

   NEW_ADDRESS now holds the value 0102H.

6. The INPUT instruction is used again. This time to point to the state that has the high byte of the destination address:

INPUT ABS,NEW_ADDRESS   ;Point to the state with the value of
                           ;NEW_ADDRESS in the ADDR column.

   When this state is found, the communication variables will be initialized as follows:

Communication Variable	Initialized to:
INPUT_ADDRESS	0102H
INPUT_DATA	01H
INPUT_STATUS	0010B
INPUT_ERROR	0
INPUT_TAG	0

7. The value of DATA for this state should also be saved for calculating the destination address.

LOAD INPUT_DATA   ;Load the ACCUMULATOR with the high
                  ;byte of the destination address

STORE HIGH_BYTE   ;Store the contents of the
                  ;ACCUMULATOR into the variable
                  ;HIGH_BYTE

   These instructions put the value 01H into HIGH_BYTE.

8. The inverse assembler now has all the information needed to calculate the STA instruction's destination address. After this calculation, the output display buffer should contain:

STA 0106

This completes the inverse assembly process for line 0000 in this example. The inverse assembler will now return control to the logic analyzer's system software. The system software will print the values under the STAT and Time label, then begin this process all over again for line 0001 on the display.

Note 👉 A complete listing of the Intel 8085 inverse assembler is given in appendix A. The basic steps presented above are the steps used by the inverse assembler in decoding the STA instruction.

The inverse assembler adds several routines to check for errors during the inverse assembly process. These routines were omitted here to add clarity.

Additional Capabilities of the Input Instruction

The INPUT instruction can also include patterns in the STAT column when searching for a specific state. Two other communication variables are used for this task. The QUALIFY_VALUE pseudo instruction defines the pattern in the STAT column that must also be present in order for a search to succeed. For example, the QUALIFY_VALUE could have been set to 0010B in the previous example to help insure that the correct state had been found. In the case of step 3, the INPUT instruction would have searched for a state in the acquisition memory with 0101H in the ADDR column and 0010B in the STAT column.

The QUALIFY_MASK pseudo instruction further modifies the QUALIFY_VALUE. This communication variable defines which bits in QUALIFY_VALUE should be ignored during the INPUT search. The QUALIFY_MASK operand is considered to be a 32-bit mask where a "0" represents a "don't care" state and a "1" represents a "care" state. For example, if QUALIFY_MASK were set to 0001B, with a QUALIFY_VALUE of 0010B, the INPUT instruction would only look for a 0 in the LSB.

QUALIFY_VALUE and QUALIFY_MASK are enabled by adding the QUALIFIED option to the INPUT instruction operand.

To speed up the operation of the inverse assembler, the SEARCH_LIMIT command can be used to specify the number of states that will be searched to find the required data. This limit optimizes the processing by not allowing the entire acquisition memory to be searched for every INPUT instruction. If the QUALIFIED option is selected for the INPUT instruction, the SEARCH_LIMIT only applies to the number of QUALIFIED states.

Note ☞ SEARCH_LIMIT may not provide an effective limit when the INPUT,QUALIFIED instruction is executed. In this case, SEARCH_LIMIT counts the number of states where the value of STAT meets the QUALIFIED condition set up using QUALIFY_VALUE and QUALIFY_MASK. If the logic analyzer did not capture any states meeting the QUALIFIED condition, the inverse assembler will still search the entire acquisition memory.

If an INPUT instruction is terminated because the SEARCH_LIMIT is encountered, the communication variable INPUT_ERROR will be set to a non-zero value.

Putting Text into the Output Display Buffer

Each state can write up to four lines into the output display buffer, with up to 64 characters per line. The OUTPUT instruction is used to place text and numbers into the buffer.

The output display buffer does not automatically wrap text to the next line on the display when the end of a line is reached. The inverse assembler must check the position in the display buffer and generate a new line to avoid writing text off the end of a line.

The FETCH_POSITION instruction allows the inverse assembler to read where the last character was printed in the display buffer, and NEW_LINE allows the inverse assembler to generate a new line in the display buffer. When NEW_LINE is executed, the next OUTPUT instruction will begin printing in column one of the new line. No commands are provided to allow the OUTPUT instruction to print to a previous line.

Other instructions associated with an OUTPUT instruction include:

DEFAULT_WIDTH	Defines the width of the output display buffer.
POSITION	Allows the inverse assembler to move to a specific column in the display buffer, or relative to where the last character was written.
FORMAT	Defines the format for displaying the contents of the ACCUMULATOR.

Generating Symbolic Addresses

In the 8085 example, the destination address for the STA instruction was displayed in hexadecimal. Normally, assembly language programs are written with symbolic references to addresses. The programmer uses an assembler and linker to convert these symbolic references to absolute addresses.

An inverse assembly display with absolute addresses may be difficult to interpret if the original assembly language program was written using symbolic references.

The HP 1650A/B, HP 1651A/B, HP 16510A/B, and HP 16511B logic analyzers include symbol tables for converting the raw captured data under a label into text. If the "base" of a label is set to "symbol," the value of a state will be replaced by the symbol defined in the logic analyzer symbol tables. For example, if the ADDR symbol table defines the pattern 0C000H as the text "KBD", then KBD will be displayed in place of 0C000H under the ADDR label. (See the logic analyzer's operating manual for more detail on symbol tables).

The ADDR symbol table can also be used by the inverse assembler when the ADDR base is set to symbol. Using the ADDR symbol table with an inverse assembler will provide a display that resembles the assembly language source code.

The instruction to generate symbolic addresses using the ADDR symbol table is:

IF_NOT_MAPPED THEN < result>

This instruction will check the ADDR symbol table for a symbol with a value or range that corresponds to the value in the ACCUMULATOR. It would normally be used as follows:

IF_NOT_MAPPED THEN OUTPUT ACCUMULATOR,FORMAT

When this instruction is executed, the following three cases are possible:

Case 1: The value in the ACCUMULATOR matches a pattern in the ADDR symbol table.

In this case, the symbol in the ADDR symbol table will be written to the output display buffer. The OUTPUT ACCUMULATOR, FORMAT portion of this instruction will not be executed.

Note This case applies only to symbols that are defined using "Pattern" in the logic analyzer symbol table.

Example:

JMP PORT_ADDRESS

Case 2: The value in the ACCUMULATOR falls within a range defined in the ADDR symbol table.

In this case, the symbol associated with the range will be written to the output display buffer. The ACCUMULATOR will be set to the offset from the beginning of the range. The < result> part of the instruction will then be executed to display this offset.

Note This case applies only to symbols that are defined using "Range" in the logic analyzer symbol table.

Example:

JMP SUBROUTINE+ 23H

Case 3: The value in the ACCUMULATOR does not match a pattern in the ADDR symbol table and does not fall within a range defined in the symbol table. (That is, both Case 1 and Case 2 above are false).

In this case, the value of the ACCUMULATOR will be displayed using the < result> part of the instruction.

Example:

JMP 0FFFFH

Note 👆 The IF_NOT_MAPPED instruction can only access the ADDR symbol table if the base for ADDR is set to symbol.

Hints on Parsing an Opcode

The inverse assembler can be written with the CASE_OF or IF instructions to identify opcodes and status information for your microprocessor. Tables or other methods can also be used for parsing opcodes.

The speed and size of the inverse assembler is dependent on the structure of the inverse assembler code. The inverse assembler instruction set is designed to produce efficient code; however, there are techniques that will increase this efficiency.

First, study the instruction set for the target microprocessor thoroughly before designing the inverse assembler code. This study should reveal the natural breaks in the opcode values. A common approach is to divide the instruction set roughly in half by using the most significant bit as the breakpoint. Other approaches are to parse the opcode based on the number of operands in the instruction or to parse it based on the addressing modes. A microprocessor with multiple accumulators tends to have the same operations performed on each accumulator. This is a convenient way to further parse the opcode. These are some of the possibilities in parsing an opcode. You should study an instruction set and decide on the best approach for your needs.

Using INPUT_TAG to Mark States

While disassembling a state, it may be useful to mark states pointed to by the INPUT instruction. The mark can be used during subsequent inverse assembler calls to disassemble this state with a special routine.

For example, many microprocessors do not differentiate externally between memory reads that fetch data from memory and memory reads that fetch operands for a multi-byte instruction. The inverse assembler can mark a state as an operand when decoding the multi-byte instruction. When this marked state is called for inverse assembly, the inverse assembler can read the tag and display "operand" instead of "memory read."

To tag states:

Each state of the logic analyzer's acquisition memory has a TAG that can be written to and read by the inverse assembler. The lower 16 bits of the TAG are user-tags and can be directly accessed by the inverse assembler.

The inverse assembler writes to the TAG using the TAG_WITH instruction. The TAG is read using the INPUT_TAG communication variable. INPUT_TAG is read the same way as the other INPUT communication variables discussed previously.

Bits 16 and 17 of the TAG are system tags. These tags are used to inverse assemble microprocessors with incomplete status. For a thorough discussion of the use of bits 16 and 17, see appendix B.

Other Communication Variables

This chapter has discussed the following communication variables:

INPUT_ADDRESS	INITIAL_ADDRESS
INPUT_ADDR_B	INPUT_DATA_B
INPUT_DATA	INITIAL_DATA
INPUT_STATUS	SEARCH_LIMIT
INPUT_ERROR	QUALIFY_VALUE
INPUT_TAG	QUALIFY_MASK

The remaining communication variables in the IAL environment are:

INITIAL_FLAGS	INITIAL_OPTIONS
RETURN_FLAGS	TASK

INITIAL_FLAGS and INITIAL_OPTIONS are used when the microprocessor does not provide enough status information to uniquely determine the first byte of an opcode fetch. Their use is detailed in appendix B. The RETURN_FLAGS and TASK variables are explained in the following sections.

RETURN_FLAGS

RETURN_FLAGS indicate the result of the IF_NOT_MAPPED instruction. Two bits (Bit 16 and 17) are interpreted as follows:

Bit 16:   0 = no mapping was done
                              1 = mapping was successful

*Bit 17:   0 = mapped to a range
                              1 = mapped to a pattern

    *Bit 17 is valid only if Bit 16 = 1

Bit 0 of RETURN_FLAGS is also used to inverse assemble microprocessors with incomplete status. See appendix B for a complete description of using Bit 0.

TASK

All of HP's current logic analyzers and emulators can inverse assemble captured data. To avoid re-writing an inverse assembler for each different instrument, Hewlett-Packard has defined an Inverse Assembly Language that is compatible among all of the following instruments at the source code level:

    HP 64XXX Emulation
    HP 64620 State Analysis
    HP 1630A/D/G Logic Analyzer
    HP 1631A/D Logic Analyzer
    HP 1650A/B and HP 1651A/B Logic Analyzer
    HP 16510A/B and HP 16511B Logic Analyzer

Inverse assembler source code written to work on one of the machines listed above can be used in another machine, often with only minor changes.

The TASK communication variable is used to identify what kind of machine the inverse assembler is operating in. By using the TASK variable to identify which machine is being used, it may be possible to write a single inverse assembler that works properly in all of the above machines.

TASK will have the following values, depending on which HP instrument the inverse assembler is running in:

Value	Environment
0	Emulation--display memory mnemonic
1	Emulation--display trace mnemonic
2	Emulation--display trace status mnemonic
3	HP 64620 State Analysis
4	HP 1630A/D and HP 1631A/D Logic Analyzers
5	HP 1630G, HP 1650A/B, HP 1651A/B, HP 16510A/B and HP 16511B Logic Analyzers

For the HP 10391B IAL Development Package, TASK will always be 5 or greater.

Table 3-1. Inverse Assembler Executable Instruction Set

Arithmetic/Logical Instructions

Instruction	Operand	Destination	Description
ADD	Memory/immediate	Accumulator	Accumulator + operand
SUBTRACT	Memory/immediate	Accumulator	Accumulator - operand
DECREMENT	Memory	Memory	Memory - 1
INCREMENT	Memory	Memory	Memory + 1
AND	Memory/immediate	Accumulator	Accumulator AND operand
EXCLUSIVE_OR	Memory/immediate	Accumulator	Accumulator XOR operand
INCLUSIVE_OR	Memory/immediate	Accumulator	Accumulator OR operand

ACCUMULATOR Instructions

Instruction	Operand	Description
LOAD	Memory/immediate	Copy Memory/immediate to Accumulator.
STORE	Memory	Copy Accumulator to Memory.
EXTRACT_BIT	Immediate	Extract Bit # (0= LSB, 31= MSB) and put in Accumulator.
ROTATE RIGHT/LEFT,	Immediate	Rotate Accumulator.
COMPLEMENT		Ones complement Accumulator.
TWOS_COMPLEMENT		Twos complement Accumulator.

Memory Instructions

Instruction	Operand	Destination	Description
SET	Memory,immediate	Memory	Memory = Immediate (-8 to + 7)

Table 3-1. Inverse Assembler Executable Instruction Set (continued)

Conditional Instructions

Instruction	Operand	Description
CASE_OF	Memory	Branch to one of the following statements, depending on the value of Memory.
CASE_OF	MSB,LSB	Branch to one of the following statements, depending on the value of the bit range in the Accumulator.
CASE_END		End of CASE statement.

Instruction	Description
IF Memory rel Memory THEN result	Test Memory vs. Memory.
IF Memory rel Immediate THEN result	Test Memory vs. Immediate value.
IF MSB,LSB rel Memory THEN result	Test the value of the bit range in the Accumulator vs. Memory.
IF MSB,LSB rel Immediate THEN result	Test the value of the bit range in the Accumulator vs. Immediate.

Where "rel" is

$=$ equal
$< >$ not equal
$< =$ less than or equal
$> =$ greater than or equal
$<$ less than
$>$ greater than

Table 3-1. Inverse Assembler Executable Instruction Set (continued)

Program Control Instructions

Instruction	Operand	Description
CALL	Label	Branch to Label, stack return address.
GOTO	Label	Branch to Label.
RETURN		Return from CALL, or leave inverse assembler.
ABORT		Leave inverse assembler.

Output Display Buffer Instructions

Instruction	Operand	Description
OUTPUT	String constant	Write string constant to output display buffer.
OUTPUT	string	Write string to output display buffer.
OUTPUT	ACCUMULATOR,FORMAT	Write value of Accumulator to output display buffer, using the defined FORMAT.
POSITION	ABS/REL,immediate	Move in the output display buffer to an absolute column number or a column relative to the current position.
DEFAULT_WIDTH	immediate	Defines the maximum width of the Output Display Buffer

Table 3-1. Inverse Assembler Executable Instruction Set (continued)

Output Display Buffer Instructions (continued)

Instruction	Description
FETCH_POSITION	Load Accumulator with the column number where the last character was printed.
NEW_LINE	Generate a new line in the output display buffer for the current state.
IF_NOT_MAPPED THEN result	OUTPUT the symbol in the ADDR symbol for the contents of the Accumulator. If no symbol is present, execute the result instruction.

Miscellaneous Instructions

Instruction	Operand	Description
INPUT REL/ABS,	Memory/Immediate [,QUALIFIED]	Point to the desired state in the acquisition memory and initialize communication variables.
TAG_WITH	Memory/Immediate	Tag the state in acquisition memory for future reference.
NOP		No action.

Table 3-2. Pseudo Instructions

Symbolic Operand Definition

Label	Instruction	Operand	Description
LABEL	ASCII/ASC	string	Defines string for OUTPUT instructions.
LABEL	CONSTANT/CONST	immediate	Defines a constant of the name LABEL and assigns the immediate value to it.
LABEL	VARIABLE/VAR	[immediate]	Declares a variable of the name LABEL and initializes it to the immediate value.
LABEL	FORMAT a,b,c[,DISPLAY_BASE]		Defines a format for the OUTPUT instruction.

Note 👉 Labels appearing on lines without instructions are assumed to be code labels for use with the GOTO and CALL instructions. Labels for ASCII, CONSTANT, VARIABLE, and FORMAT pseudo instructions must be on the same line as the rest of the pseudo to be accepted as the label for that pseudo.

Inverse Assembler Titles

Instruction	Operand	Description
LABEL_TITLE	string	Replaces the label DATA with a string.
BASE_TITLE	string	Replaces the base title Hex with a string.

Table 3-2. Pseudo Instructions (continued)

Pseudo instructions used with the INPUT instruction

Instruction	Operand	Description
QUALIFY_VALUE	immediate	Defines the value in INPUT_STATUS to search for.
QUALIFY_MASK	immediate	Defines which bits of QUALIFY_VALUE to use or ignore.
SEARCH_LIMIT	immediate	Defines the maximum number of states searched by the INPUT instruction. When the SEARCH_LIMIT is reached, the INPUT instruction stops with INPUT_ERROR initialized to a non-zero value.

Debugging Aides

Instruction	Operand	Description
MAX_INSTRUCTION	Immediate	Limits number of instructions executed.

Table 3-3. Communication Variables

Variable	Description
INITIAL_ADDRESS	Contains the value of acquisition memory for the ADDR label when the logic analyzer system software calls the inverse assembler to decode a state.
INITIAL_DATA	Contains the value of acquisition memory for the DATA label when the logic analyzer system software calls the inverse assembler to decode a state.
INPUT_ADDRESS	Contains the value of acquisition memory for the ADDR label for a specific state. INPUT_ADDRESS is initialized in two cases: 1. When the logic analyzer system software calls the inverse assembler to decode a state, or 2. When the INPUT instruction is executed. In this case, INPUT_ADDRESS contains the value in the ADDR label for the state pointed to by the INPUT instruction.
INPUT_DATA	Contains the value of acquisition memory for the DATA label for a specific state. INPUT_DATA is initialized in the same cases as INPUT_ADDRESS.
INPUT_STATUS	Contains the value of acquisition memory for the STAT label for a specific state. INPUT_STATUS is initialized in the same cases as INPUT_ADDRESS.
INPUT_ADDR_B	Contains the value of acquisition memory for the ADDR_B label for a specific state. INPUT_ADDR_B is initialized in the same cases as INPUT_ADDRESS.
INPUT_DATA_B	Contains the value of acquisition memory for the DATA_B label for a specific state. INPUT_DATA_B is initialized in the same cases as INPUT_ADDRESS.
INPUT_TAG	Contains the value of the tag for this state. Tags can be written by the inverse assembler to mark a state for future reference. They are also used when decoding a microprocessor that generates insufficient status (See appendix B).

Table 3-3. Communication Variables (continued)

Variable	Description
INPUT_ERROR	Set to zero if the communication variables were successfully initialized. Set to non-zero if the initialization failed.
QUALIFY_VALUE	Defines the pattern in the STAT label that must be present during an INPUT REL/ABS,QUALIFIED instruction.
QUALIFY_MASK	Defines which bits of QUALIFY_VALUE are used in a qualified INPUT search.
SEARCH_LIMIT	Limits the number of states searched by the INPUT instruction.
RETURN_FLAGS	Indicate the results of an IF_NOT_MAPPED instruction. Also used when disassembling microprocessors with incomplete status.
INITIAL_FLAGS	Used when disassembling microprocessors with incomplete status.
INITIAL_OPTIONS	Used when disassembling microprocessors with incomplete status.
TASK	Identifies which HP machine the inverse assembler is executing in.

4

Inverse Assembler Instruction Set

Introduction   This final chapter defines the syntax, explains the function of each executable and pseudo instruction, and illustrates how the instructions are used with one or more examples. For quick reference, the instructions are arranged alphabetically.

Choosing a Text Editor

The inverse assembler source code can be written using any text editor on your PC that can output the file in printable U.S. ASCII characters. Source code that contains non-printable, U.S. ASCII characters may not assemble properly.

Many text editors add non-printable U.S. ASCII characters to files to represent text enhancements, such as underlining or italics. These editors may also include special instructions to allow a file to be stored as a U.S. ASCII file only. These instructions are included to allow a text file to be used by other PC programs.

If the manual for your text editor is unclear about the characters stored in files, the following test may be useful:

1. Write a few paragraphs of text using your text editor. Do not include text enhancements, such as underlining or italics.

2. Store the text to disk. Use the special storage instructions for storing ASCII text, or the editor instructions that allow a file to be used by another PC program.

3. Exit the text editor. At the DOS prompt, type:

type < filename>

where < filename> is the name of the file you stored from your text editor.

4. When you select the "Enter" key after typing the above command, DOS will copy your file to the PC screen. If the screen shows characters you did not type into the file, your text editor is storing non-printable ASCII characters and cannot be used with the IAL assembler. If the screen looks exactly like the text you entered in your editor, this file can probably be used with the IAL assembler.

(Non-printable U.S. ASCII characters include special national characters and characters outside the code range of 32 through 127.)

Entering Inverse Assembler Source Code

The following syntax rules must be followed to allow IAL source code to be properly assembled by the assembler.

The First Line

The first line of inverse assembler source code MUST be the following:

"IAL"

This string MUST begin in the first column of the line and should be the only text on that line. "IAL" tells the assembler what kind of source code follows.

Line Format Rules

Each line is divided into four fields:

1. The Label Field
2. The Operation Field
3. The Operand Field
4. The Comment Field

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

Label	Operation	Operand	Comment
NEW_ADDR	VAR	0	;Defines the variable NEW_ADDR ;and initializes it to 0

Note 👆 It is recommended that each field in the source statement start at a fixed position (column) in the line. Presenting each line in a fixed format improves readability.

b. One or more spaces MUST separate the fields in a line.

c. A label field, if used, MUST begin in column one.

d. The operation and operand fields MUST not begin in column one. Column one is reserved for the start of the label field or for the delimiter of the comment field.

Additional rules and conventions governing the fields in a line are given in the following paragraphs.

Length of Lines

Lines may contain up to 110 characters (including spaces) and are terminated by a carriage return < CR> . A line containing more than 110 characters will be truncated to 110 characters.

Blank lines are ignored by the assembler and can be used to improve the readability of the source code.

Label Field   The label field is used for the following:

1. To declare variables (VAR).
2. To define constants (CONST) or string constants (ASCII).
3. To define formats (FORMAT) for the OUTPUT instructions.
4. To define the destination of GOTO or CALL instructions.

The label field is required in the ASCII, VAR, CONST, and FORMAT pseudo instructions. It is optional for all other instructions and pseudo instructions. A line with only a label is a valid line.

Every label MUST be unique. The instructions, pseudo instructions, and communication variables listed in Tables 3-1, 3-2, and 3-3 are reserved words and cannot be used for label names.

The label field starts in column one of the line and MUST be terminated by a space or a colon (:).

A label can consist of the letters A through Z (upper and lower case), the numeric digits 0 through 9, and the underline symbol (_). The label may contain any number of characters, as long as the first character is a letter.

Valid Labels:

       ab_cd
       AB_CD
       A5rHi

Invalid Labels:

       ab.cd?
       $BCDEF
       4UVWXY

If more than 15 characters are entered in the label field, the assembler will print all characters in the output listing; however, it will only use the first 15 characters for label identification. The assembler will recognize:

STATEMENT_LABEL1

and

STATEMENT_LABEL2

as identical labels and will issue a "DUPLICATE SYMBOL" error message. The assembler does check the case of characters in a label. It will treat:

Name

and

name

as two separate labels.

Operation Field

The operation field contains an IAL instruction or pseudo instruction. The operation field follows the optional label field and is separated from it by at least one space, a tab, or a colon (:). If there is no label, the instruction may begin in any column position after column one.

An instruction in the operation field MUST be all upper-case letters, exactly as listed in the "Language Reference" section of this manual.

An operation field that requires an operand is terminated by one or more spaces or by a tab. If no operand is required, the operation field can also be terminated by a carriage return, or by a semicolon (;). Termination of this type of operation field indicates the start of the comment field.

Operand Field The operand field specifies values or variables required by the IAL instruction. The operand field, if present, follows the operation field and MUST be separated from it by at least one space or tab. The following table shows the different types of operands that may be expected by an IAL instruction:

Operand	Description
memory	Name of a variable defined using the VAR instruction.
immediate	An immediate numeric value, or the name of a constant defined using the CONST declaration.
string	An ASCII string, enclosed in quotation marks (").
LABEL	The line label to branch to in a CALL or GOTO instruction.
MSB,LSB	A subrange of the ACCUMULATOR used in IF and CASE instructions. MSB specifies the most significant bit of the ACCUMULATOR to be used in the test; LSB specifies the least significant bit.
result	The IAL instruction to be executed if an IF or an IF_NOT_MAPPED instruction tests true.

Comment Field The optional comment field may contain any information that you feel is 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 line will start the comment field (except when used in an ASCII string).

In addition, an asterisk (*) in column one of a line indicates that the entire line is a comment.

Delimiters   Certain characters are used to indicate the end of fields or labels, and the beginning of others. These characters, referred to as delimiters, should not be used as ordinary characters. For example, a space cannot be used as part of a label name. A list of delimiters is shown in the following table:

Delimiter	Use
Space	Separates fields or operands; ends a label
Tab	Separates fields; ends a label
Semicolon (;)	Indicates the start of a comment field
Asterisk (*)	When used in column one, indicates the start of a comment field.
Colon (:)	Indicates the end of a label field
Apostrophes ('..')	Indicates a character string
Quotation Marks ("..")	Indicates a character string
Carets (^ ..^)	Indicates a character string

Numeric Terms   A numeric term 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, 26Q). A hexadecimal term MUST have the suffix "H" (for example: 0BBH, 2CDH, 36H). When no suffix is assigned, the decimal value is always assumed.

Note   You must start a hexadecimal term with a decimal digit. The assembler will identify a term that starts with an alphabetic character as a label or an expression.

String Constants

String constants are defined by the ASC or ASCII pseudo instruction to be used by the OUTPUT instruction. String constants are enclosed by apostrophes ('..'), quotation marks ("..''), or carets (^ ..^).

String constants do not have a numerical value and cannot be operated on by IF or CASE statements.

Language Reference

The following pages list the executable and pseudo instructions for Hewlett-Packard's Inverse Assembler Language. For quick reference, these instructions are arranged alphabetically.

ABORT   **Leave Inverse Assembler**

SYNTAX

    Label   Operation   Operand   Comment

                  ABORT

The ABORT instruction will pass control back to the logic analyzer system software even if the program is currently in a subroutine.

Example:

```
ABORT                 ;Return to
                      ;System Software.
```

ADD   **Add to Accumulator**

SYNTAX

    Label   Operation   Operand   Comment

                ADD   memory/immediate

The contents of the operand field are added to the value in the accumulator. The operand can be either a memory reference or an immediate value. The value of immediate data can range from 0 to 0FFFFFFFFH (32 bit value).

Example:

ADD   1   ;Increment
                                                       ;accumulator.

ADD   NAME   ;Add variable to
                                                       ;accumulator.

AND   **Logical AND with Accumulator**

SYNTAX

Label	Operation	Operand	Comment
	AND	memory/immediate	

This instruction performs a logical "AND" of the value of the operand and the value in the accumulator. The operand can be either a memory reference or an immediate value. The value of immediate data can range from 0 to 0FFFFFFFFH (32 bit value).

Examples:

AND   1   ;AND lower bit.

AND   MASK   ;AND with MASK
                                        ;variable.

Pseudo ASCII/ASC   **Define ASCII String**

```
SYNTAX

   Label        Operation          Operand            Comment

   LABEL        ASCII/ASC          string
```

The ASCII pseudo instruction is used to define an ASCII string to be used with an OUTPUT instruction. This is recommended for strings that are used more than once to minimize the size of the inverse assembler.

Examples:

```
LOAD_STG      ASCII     "LOAD"        ;Define text for
                                      ;OUTPUT instruction.

STORE_STG     ASC       "STORE"
```

Pseudo BASE_TITLE   **Define BASE Title**

SYNTAX

Label   Operation   Operand   Comment

               BASE_TITLE   string

The BASE_TITLE pseudo instruction is used to place text in the
BASE portion of the inverse assembler field. The string defined in
BASE_TITLE will replace the default "Hex."

Examples:

BASE_TITLE   "Mnemonic"   ;Places the text Mnemonic
                                             ;in the base of the
                                             ;inverse assembler field.

Note 👆   The BASE_TITLE text does not effect the way data is processed by
                  the logic analyzer.

CALL   **Transfer Program Control to Label**

SYNTAX

    Label   Operation   Operand   Comment

               CALL   LABEL

The CALL instruction transfers program control to the label specified. A RETURN instruction will transfer control to the statement following the CALL. The maximum subroutine nest level is 16.

Example:

```
CALL            SUBROUTIN        ;Control is
                                 ;transferred to
                                 ;SUBROUTIN.
```

CASE_OF   **Conditional Testing of Variable or Accumulator**

SYNTAX			
Label	Operation	Operand	Comment
	CASE_OF	memory/MSB,LSB	

The CASE_OF instruction allows conditional testing of either a user variable or the accumulator. Program control will branch to one of the instructions following the case depending on the value of the variable or accumulator. If the operand is a memory reference, then the value of the memory location becomes an index added to the current program counter to fetch the next instruction. Otherwise, the bit range specified is the index.

CASE_OF rules:

1. The CASE_OF statement must be followed by a CASE_END statement.

2. If the value of the operand falls into the range of the CASE, the corresponding instruction will be executed. Otherwise, the statement following CASE_END will be executed.

3. Any instruction may occur in the CASE except:

   a. IF,

   b. IF_NOT_MAPPED, or

   c. another CASE.

4. If a CALL is executed, the return from the subroutine will execute the instruction following the CASE_END.

5. GOTO and RETURN instructions are unconditional branches out of the CASE.

Example 1:

```
CASE_OF            NAME            ;Test memory NAME.
  OUTPUT           "LDA"           ;Execute this if NAME =  0.
  CALL             SUB             ;Execute this if NAME =  1.
CASE_END
< next instruction>                ;Execute next instruction
                                   ;if NAME < >  0,1.  Execution
                                   ;will also take place
                                   ;after either statement
                                   ;in the body of CASE
                                   ;is executed.
```

Example 2:

```
CASE_OF            28,27           ;Decode add/subtract group.
  OUTPUT           "addu"          ;Execute if accumulator
                                   ;bit 28,27 =  0
  OUTPUT           "subu"          ;Execute if accumulator
                                   ;bit 28,27 =  1
  OUTPUT           "adds"          ;Execute if accumulator
                                   ;bit 28,27 =  2
  OUTPUT           "subs"          ;Execute if accumulator
                                   ;bit 28,27 =  3
CASE_END
```

COMPLEMENT   **One's Complement on Accumulator**

SYNTAX			
Label	Operation	Operand	Comment
	COMPLEMENT		

A one's complement is performed on the contents of the accumulator. Bits that are 1 change to 0 and those that are 0 change to 1.

Example:

COMPLEMENT   ;One's complement
                                              ;on accumulator.

CONSTANT/ CONST   **Define Constant**
Pseudo

SYNTAX			
Label	Operation	Operand	Comment
NAME	CONST	constant	

This instruction allows commonly used constants to be defined and referenced by the label specified. Normally, immediate constants are handled automatically by the assembler, but the assembler will not optimize the use of constants that are identical. To avoid wasting data space, commonly used constants can be defined and referenced by symbolic names. All constants are 32-bit quantities.

Example:

ML24	CONST	0FFFFFFH	;Mask lower 24-bit
			;constant.

DECREMENT

DECREMENT   **Decrement Memory Location**

```
SYNTAX

    Label        Operation        Operand        Comment

                 DECREMENT        memory
```

This instruction decrements the memory location specified by the
operand by one. The operand must be defined using the VARIABLE
pseudo instruction.

Example:

DECREMENT   SAM   ;Decrement the
                                           ;variable SAM.

**Pseudo
DEFAULT_WIDTH**
                      Default width of display field

SYNTAX			
Label	Operation	Operand	Comment
	DEFAULT_WIDTH	immediate	

This instruction defines the maximum DEFAULT_WIDTH of the display field. Operands can range from 1 to 64. The logic analyzer can display a maximum of 59 characters. When a DEFAULT_WIDTH larger than 59 is specified, the extra characters can be scrolled onto the screen.

If no default width is specified, the default width is 32.

Example:

```
DEFAULT_WIDTH     40                          ;The default
                                              ;width is 40.
```

EXCLUSIVE_OR   **Exclusive OR with Accumulator**

SYNTAX

Label	Operation	Operand	Comment
	EXCLUSIVE_OR	memory/immediate	

This instruction performs a logical "EXCLUSIVE OR" of the operand value and the accumulator value. The operand can be either a memory reference or an immediate value. The value of immediate data can range from 0 to 0FFFFFFFFH (32-bit value).

Example:

EXCLUSIVE_OR   1   ;Toggle lower bit

EXCLUSIVE_OR   MASK   ;OR variable MASK.

EXTRACT_BIT   Extract from Accumulator

```
SYNTAX

   Label        Operation           Operand          Comment

                EXTRACT_BIT         immediate
```

The operand for this instruction is a bit number in the accumulator with a range from 0 to 31. This bit will be extracted and the value of the accumulator set to its value (either 0 or 1).

Example:

EXTRACT_BIT   10   ;Set accumulator
                                         ;to value of bit
                                         ;10 in accumulator.

FETCH_POSITION   Determine Current Position in the Output Buffer

SYNTAX

Label	Operation	Operand	Comment
	FETCH_POSITION		

The FETCH_POSITION instruction sets the accumulator to the
column number of the display output buffer where the next OUTPUT
instruction will write characters. The instruction is particularly useful
in determining whether or not an output string will fit on the current
output line.

Example:

```
OUTPUT STRING1                              ;output a string

FETCH_POSITION                              ;determine next column
                                            ;number to write

;
;the accumulator now contains the next column
;number.  If the second string to be output is 25
;characters, adding 25-1 to the accumulator will
;give the column number of the last character
;in STRING2
;

ADD 24                                      ;calculate column number of last

;
;Now see if the last column number to be used is
;greater than the maximum line length.  If it is,
;issue a NEW_LINE instruction, then output the
;second string.
;

IF 31,0> 64 THEN NEW_LINE                   ;start a new line if > 64

OUTPUT STRING2                              ;output the second string
```

Pseudo FORMAT   **Format Accumulator**

```
SYNTAX

   Label        Operation           Operand              Comment

   NAME         FORMAT              a,b,c [,DISPLAY_BASE]
```

FORMAT is used to define how the accumulator should be converted when used in conjunction with the OUTPUT instruction. The operands are defined as follows:

Operand	Description
a	Defines how many bits of the accumulator will be converted. It can range from 1 to 32; if a subrange is specified, then the most significant bits will be ANDed with 0.
b	Specifies the base of the conversion. It can be BIN, OCT, DEC, or HEX
c	Specifies the number of characters to be displayed by the OUTPUT instruction. Leading zeros will be supplied if the number is smaller than the converted field. If c= LEFT_JUSTIFIED, a zero suppressed number will be generated, displaying as many digits as necessary.
DISPLAY_BASE	This optional operand can be used to append a B, O, D, or H on the end of the constant converted, depending on the numeric base of the number.

All operands in the accumulator can be displayed as one ASCII character by using the keyword ASCII instead of the number-of-bits operand.

Examples:

HEX_FMT   FORMAT   16,HEX,4,DISPLAY_BASE   ;Four digit
                                                   ;hex format.

REG_FMT   FORMAT   3,DEC,LEFT_JUSTIFIED   ;Left-justified
                                                   ;decimal.

ASC_FMT   FORMAT   ASCII   ;ASCII format.

GOTO   **Transfer Program Control**

```
SYNTAX

    Label        Operation          Operand          Comment

                 GOTO               LABEL
```

The GOTO instruction transfers program control to the label specified.

Example:

GOTO   OPCODE_STATUS   ;Branch to
                                             ;status routine.

IF   **Compare Operands**

SYNTAX

    Label   Operation   Operand   Comment

               IF   memory rel memory/
                                immediate THEN result

               or

               IF   MSB,LSB rel memory/
                                immediate THEN result

    Where rel is

         = equal
         < > not equal
         < = less or equal
         > = greater or equal
         < less than
         > greater than

IF

The IF instruction allows operands to be compared and decisions made based on the results of the comparison. The first form of the IF instruction allows memory to be compared to other memory locations or to immediate data. Immediate data can range from 0 to 0FFFFFFFFH (32 bits). The operand to the THEN part can be any instruction except another IF, a CASE, or IF_NOT_MAPPED.

The second form of the IF instruction allows a bit range of the accumulator to be tested. Here the first operand specifies the most significant bit (MSB) and the second operand specifies the least significant bit (LSB). This allows all or part of the accumulator to be tested against an immediate value or memory.

Examples:

IF NAME <   101 THEN GOTO LABEL   ;Test value of
                                               ;NAME

IF 6,3 =   1001B THEN CALL SUBROUTIN   ;Test accumulator
                                               ;bit range 6-3
                                               ;inclusive.

IF_NOT_MAPPED   Check for Symbol in ADDR Symbol Table

SYNTAX

Label	Operation	Operand	Comment
	IF_NOT_MAPPED THEN result		

The IF_NOT_MAPPED instruction is used to put symbols from the ADDR symbol table into the inverse assembled listing. A listing that uses the ADDR symbol table may be easier to interpret because the logic analyzer's display will more closely resemble the original microprocessor source code.

The most common usage of the IF_NOT_MAPPED instruction is:

IF_NOT_MAPPED THEN OUTPUT ACCUMULATOR,FORMAT

When the IF_NOT_MAPPED instruction is executed, the contents of the ACCUMULATOR are compared to the ADDR symbol table. Three conditions are possible:

1. If the address in the ACCUMULATOR matches a symbol defined as a pattern, i.e., not included in a range, the symbol associated with the address is displayed. The "result" part of the instruction, in this case the "OUTPUT" instruction, is not executed. This means the value passed corresponds exactly with a particular symbol.

Example:

JMP PORT_ADDRESS

2. If the address is not found as a single valued address, but is part of a range, the symbol associated with the range will be displayed. The ACCUMULATOR will be set to the offset from the beginning of the range. The "result" part of the instruction will be executed to display this offset.

Example:

JMP SUBROUTIN+ 023H

3. If the address in the ACCUMULATOR is not found as a single valued address or as part of a range, i.e., not in the symbol table, no symbolic information will be displayed. Here, the value in the accumulator will contain the absolute address and will be displayed using the "result" part of the function.

Example:

JMP 0FFFFH

The RETURN_FLAGS variable has two flags in the upper 16 bits. These flags indicate the result of the IF_NOT_MAPPED instruction. They are interpreted as follows:

Bit 16:	0 =	no mapping was done
	1 =	mapping was successful
*Bit 17:	0 =	mapped to a range
	1 =	mapped to a single value symbol

* Bit 17 is valid only if bit 16 = 1.

INCLUSIVE_OR   **Logical OR with Accumulator**

SYNTAX

    Label   Operation   Operand   Comment

             INCLUSIVE_OR   memory/immediate

This instruction performs a "logical or" of the operand value and the accumulator value. The operand can be either a memory reference or an immediate value. The value of immediate data can range from 0 to 0FFFFFFFFH (32-bit value).

Example:

INCLUSIVE_OR 1   ;Set bit 1.

INCLUSIVE_OR MASK   ;OR with MASK.

INCREMENT   **Increment Memory Location**

SYNTAX

Label	Operation	Operand	Comment
	INCREMENT	memory	

This instruction increments the memory location specified by the operand by one. The operand must be defined by the VARIABLE pseudo.

Example:

INCREMENT   NAME   ;Increment
                                           ;variable NAME.

INPUT   **Input Data**

```
SYNTAX

    Label       Operation            Operand            Comment

                INPUT                ABS/REL,memory/immediate
                                     [,QUALIFIED]
```

The INPUT instruction to allows an inverse assembler to read
information from any state in the acquisition memory. This may be
necessary in order to completely decode a multibyte instruction, or to
display the results of an executed instruction.

The INPUT instruction is used to "point" to the desired state. When
that state is located in the acquisition memory, the following
communication variables get initialized with the contents of that state:

INPUT_ADDRESS   Contains the value of acquisition memory in the ADDR column of
                  the state being pointed to.

INPUT_DATA   Contains the value of acquisition memory in the DATA column of
                  the state being pointed to.

INPUT_ADDR_B   Contains the value of acquisition memory in the ADDR_B column of
                  the state being pointed to.

INPUT_DATA_B   Contains the value of acquisition memory in the DATA_B column of
                  the state being pointed to.

INPUT_STATUS   Contains the value of acquisition memory in the STAT column of the
                  state being pointed to.

INPUT_ERROR   Set to zero if the specified state was found and the communication
                  variables were successfully initialized. Set to non-zero if the state
                  was not found or if the variable could not be initialized.

INPUT

INPUT_TAG Contains the value of the tag for this state. Tags can be written by the inverse assembler to mark a state for future reference. They are also used when decoding a microprocessor that generates insufficient status (see appendix B).

INPUT can point relative to the current state being processed by the inverse assembler, or the acquisition memory can be searched to find a state containing a specific value in the ADDR column.

If the QUALIFIED option is specified, the INPUT instruction will also look for a specific pattern in the STAT column. This qualifier is set by the values in the QUALIFY_VALUE and QUALIFY_MASK pseudo instructions.

INPUT REL,operand:

This option points to a state before or after the current state being processed by the inverse assembler. The operand indicates how far from the current state to point and which direction (before or after the current state). If the operand is a positive number, the INPUT instruction is pointing to a state after the current state. A negative operand points to a state before the current state.

Note INPUT REL, operand does not change the current state.

If the inverse assembler is currently decoding line 1 on the logic analyzer display, the statement

INPUT REL,2

would cause the inverse assembler to skip over line 2 on the display and point to the acquired data in line 3.

The operand can be either an immediate value or a user-defined variable. To read data after the current line, use a user-defined variable or an immediate value that is positive. To read data before the current line, use an immediate value that is negative.

INPUT ABS, operand:

This option will search forward through the acquisition memory to find a state with a specific value in the ADDR column, and will point to the acquired data in the line containing that address.

The operand specifies the absolute address. The operand must be a user-defined variable or a communications variable (see table 3-3 in chapter 3).

The SEARCH_LIMIT variable limits the number of states to be searched by the INPUT routine. If QUALIFIED is specified, then the search count applies to the number of states that are satisfied by the status qualification check.

Examples:

INPUT	ABS,DATA_ADDRESS,QUALIFIED	;Qualified on ;status.
INPUT	ABS,DATA_ADDRESS	;Not qualified.
INPUT	REL,2	;Point two states ;ahead.
INPUT	REL,COUNT	;Relative with ;count.

Pseudo   **Define Inverse Assembler Title**
LABEL_TITLE

SYNTAX			
Label	Operation	Operand	Comment
	LABEL_TITLE	string	

The LABEL_TITLE pseudo instruction is used to define the title of the inverse assembler field. The string defined in LABEL_TITLE will replace the default "DATA."

Examples:

LABEL_TITLE   ^ 8085 Mnemonic^

LOAD   **Load Accumulator**

SYNTAX

    Label   Operation   Operand   Comment

               LOAD   memory/immediate

This instruction loads the accumulator with the value specified by the operand field. The operand can be either a memory reference or an immediate value. The value of immediate data can range from 0 to 0FFFFFFFFH (32-bit value).

Examples:

LOAD   1   ;Set accumulator
                                          ;to 1.

LOAD   SAM   ;Load value of SAM.

MAX_INSTRUCTION Pseudo

Limit Number of Instructions Executed

SYNTAX

Label	Operation	Operand	Comment
	MAX_INSTRUCTION	immediate	

Since the programmer has the ability to control program flow, it is possible to program an infinite loop that will never return to the calling program. To avoid this problem, a maximum number of instructions variable is used to limit the number of instructions that can be executed each time the inverse assembler is called. This number is initialized to a large number and should never interfere with the inverse assembler. However, this number can be used to set a low limit on the instruction limit to see which calls to the inverse assembler take the most time. This can be used to optimize sections or stop near-infinite loops. The value of MAX_INSTRUCTION, initialized by this pseudo, cannot be changed during the inverse assembly. If the instruction count exceeds this value, the inverse assembler is aborted. In addition, an instruction overflow message is placed in the output buffer and displayed.

The default value for MAX_INSTRUCTION is 10000.

Example:

MAX_INSTRUCTION 50   ;Check for execution
                                      ;of more than 50
                                      ;instructions.

MAX_INSTRUCTION 10000   ;This sets a large
                                      ;limit so only infinite
                                      ;loops will abort the IAL.

NEW_LINE   **Begin Generating a New Output Line**

```
SYNTAX

    Label        Operation              Operand             Comment

                 NEW_LINE
```

The NEW_LINE instruction is used when more than one line of information is to be output by the inverse assembler for a single captured analysis state. Following the execution of the NEW_LINE instruction, subsequent OUTPUT and POSITION instructions refer to the new line of inverse assembler output. The NEW_LINE instruction can be used to generate up to four inverse assembler output lines. Exceeding this limit will cause the inverse assembler to abort.

Example:

```
OUTPUT "This is line 1"
NEW_LINE
OUTPUT "****This is line 2"
```

This series of instructions will produce the output shown below:

```
This is line 1
****This is line 2
```

NOP   **No Operation**

SYNTAX

Label	Operation	Operand	Comment
	NOP		

The NOP instruction has no effect on the execution of the inverse assembler. The instruction following NOP will be executed next.

Example:

          CASE_OF   0,0

            OUTPUT   "F"

              NOP

          CASE_END

OUTPUT   **Output to Output Buffer**

```
SYNTAX

  Label      Operation           Operand           Comment

             OUTPUT              string/ACCUMULATOR
                                 ,FORMAT
```

The OUTPUT instruction expects an operand defined by the ASCII
pseudo, an immediate string, or the key word ACCUMULATOR
followed by a conversion format defined by the FORMAT pseudo.
The first two operands will copy ASCII text to the output buffer. The
third operand will convert the accumulator using the specified format
to a number in the output buffer.

Example:

```
LOAD_STG    ASCII       "LOAD"

            OUTPUT      LOAD_STG                ;Output LOAD
                                                ;text.

            OUTPUT      "LOAD"                  ;Output immediate
                                                ;text.

            OUTPUT      ACCUMULATOR,HEX_FMT     ;Convert
                                                ;accumulator
                                                ;to hex.
```

POSITION   Position Column Pointer

```
SYNTAX

    Label        Operation            Operand            Comment

                 POSITION             ABS/REL,column number
```

The POSITION instruction allows the current column pointer to be moved to an absolute or relative position in the output buffer. The column number can range from 1 to 64 for absolute positioning or -32 to 31 for relative positioning. In the relative mode, negative numbers move the column position to the left of the current location and positive numbers move it to the right.

Example:

```
POSITION            ABS,10              ;Move to column
                                        ;10.

POSITION            REL,-2              ;Move to the left
                                        ;2 columns.
```

QUALIFY_MASK &   **Set Qualify Specifications**
QUALIFY_VALUE
Pseudos

SYNTAX			
Label	Operation	Operand	Comment
	QUALIFY_MASK	immediate	
	or		
	QUALIFY_VALUE	immediate	

QUALIFY_MASK and QUALIFY_VALUE are used to set qualify specifications for the INPUT instruction. When INPUT ABS,operand,QUALIFIED or INPUT REL,operand,QUALIFIED is executed, both the address and status must be satisfied before data is returned. The mask operand is considered to be a 32-bit mask where a 0 represents a "don't care" state and a 1 represents a "care" state. The status in the analysis buffer is first masked (ANDed) with the value of QUALIFIED_MASK to obtain the value of "care" bits. Then this value is compared to QUALIFY_VALUE to see if the status is satisfied.

Note 👆 QUALIFY_MASK and QUALIFY_VALUE are also the names of communication variables and should be treated as communication variables if the mask or the value needs to vary dynamically. Use the psuedo-ops if qualify specifications are constants that do not vary.

Examples:

QUALIFY_MASK	00101B	;Only care about ;bits 0 and 2
QUALIFY_VALUE	00001B	;Value must ;be 001B.

RETURN **Return**

SYNTAX			
Label	Operation	Operand	Comment
	RETURN		

The RETURN instruction can be used to return to the instruction following a CALL or to leave the inverse assembler if a RETURN is executed without any subroutine nesting.

Example:

RETURN   ;Return to calling routine
                          ;or leave inverse assembler
                          ;if not in a subroutine.

ROTATE   **Rotate Accumulator Contents**

```
SYNTAX

    Label         Operation              Operand              Comment

                  ROTATE                 RIGHT,immediate
                                         or
                                         LEFT,immediate
```

This instruction rotates the accumulator contents either right or left the number of bits specified. The operand can range from 1 to 32. Bits that are shifted off the left side (on left shifts) are rotated back on the right side, and vice versa (circular shift).

Examples:

ROTATE   LEFT,10   ;Shift left 10
                                       ;bits and rotate
                                       ;in on right.

ROTATE   RIGHT,20   ;Shift right 20
                                       ;bits and rotate
                                       ;in on left.

SEARCH_LIMIT Pseudo

Limit Analysis Search

SYNTAX		
Operation	Operand	Comment
SEARCH_LIMIT	immediate	

The SEARCH_LIMIT instruction applies to the INPUT instruction and to reading data from the analysis buffer after a trace. The operand specifies how many analysis states should be searched in order to find the required data. The limit optimizes processing by not allowing the entire buffer to be searched each time.

The search limit should be set to the maximum number of memory or I/O references made between opcode fetches. For example, if the inverse assembler was searching for a memory read state and that state was not captured by the analysis hardware, SEARCH_LIMIT would be used to limit the number of states scanned. The variables QUALIFY_MASK and QUALIFY_VALUE may be used to qualify the search. Every time the condition is satisfied, the search count is incremented. QUALIFY_MASK and QUALIFY_VALUE can be changed to search for other conditions and SEARCH_LIMIT can be defined to reflect the number of states that are expected to be found.

The default for SEARCH_LIMIT is 16.

Examples:

```
SEARCH_LIMIT      7                    ;Search limited to 7
                                       ;analysis states.
```

SET   **Set Memory**

SYNTAX

   Label   Operation   Operand   Comment

                   SET   memory,immediate

The memory location specified is set to the value in the immediate
operand. The operand can range from -8 to + 7.

Example:

SET   NAME,2   ;Set value of
                                                 ;NAME to 2.

STORE   **Store Value in Accumulator**

```
SYNTAX

  Label          Operation              Operand              Comment

                 STORE                  memory
```

This instruction stores the value in the accumulator in the location defined by the operand. The operand label must be defined by the VAR pseudo, or be one of the communication variables (see table 3-3 in chapter 3).

Example:

STORE   NAME   ;Store accumulator value
                                         ;in variable NAME.

SUBTRACT   **Subtract from Accumulator**

SYNTAX			
Label	Operation	Operand	Comment
	SUBTRACT	memory/immediate	

The value specified by the operand is subtracted from the value in the
accumulator. The operand can be either a memory reference or an
immediate value. The value of immediate data can range from 0 to
0FFFFFFFFH (32-bit value). Negative numbers are expressed in two's
complement form.

Example:

SUBTRACT   1   ;Decrement
                                               ;accumulator.

SUBTRACT   NAME   ;Subtract value of
                                               ;NAME from
                                               ;accumulator.

SYNTAX

   Label   Operation   Operand   Comment

               TAG_WITH   memory/immediate

The TAG_WITH instruction provides a convenient method to "mark" a state. This mark can then be used during subsequent inverse assembly calls.

TAG_WITH is used most often when disassembling a multibyte instruction. When the first state of a multibyte instruction is disassembled, the inverse assembler will look ahead with the INPUT instruction to get all of the bytes of the instruction. The TAG_WITH instruction can be used to mark the bytes as "already decoded." When this "tagged" line is disassembled later, the inverse assembler can display only the status of the line, instead of attempting to decode it.

The tag operand is the value of the tag to be associated with this "tagged" state. The operand may be an immediate value, or a user-defined variable. The tag has a 16-bit value; the non-tagged value is zero. When the INPUT instruction is executed, the variable INPUT_TAG will be initialized with the tag value of that state. For more details on using TAG_WITH, refer to appendix B.

Example:

  TAG_WITH   TAG_VALUE   ;Tag current
                                    ;analysis state.

TWOS_COMPLEMENT   **Two's Complement on Accumulator**

SYNTAX

Label   Operation   Comment

TWOS_COMPLEMENT

A two's complement is performed on the accumulator, changing all 1 bits to 0 and 0 bits to 1, then adding 1 to the result.

Example:

TWOS_COMPLEMENT   ;Negate accumulator.

Pseudo
VARIABLE/VAR

Initialize and Reserve Memory

SYNTAX			
Label	Operation	Operand	Comment
NAME	VARIABLE	immediate	

The VARIABLE instruction defines a storage location that can be used on arithmetic and conditional statements. It can be initialized to a specific value with the optional operand field.

Example:

```
NAME            VARIABLE        0FFH    ;Define variable
                                        ;storage and assign
                                        ;an initial value
                                        ;of 0FFH.
```

8085
Inverse Assembler

```
^IAL^

***********************************************************************
*
*   INVERSE ASSEMBLER FOR THE 8085 MICROPROCESSOR
*
*   This source code can be used with the HP 64620S, HP 1630A/D/G
*   HP 1631A/D, HP 1650A, HP 1651A, or HP 16510A logic analyzers.
*
*   INPUT_STATUS consists of the following 8085 signals:
*                          _
*          BIT 3          IO/M
*          BIT 2          HLDA
*          BIT 1          S1
*          BIT 0          S0
*
*   These signals are clocked in at the end of the bus cycle by the
*   OR'd combination of RD, WR and INTA.
*
***********************************************************************
*
*                    INITIALIZE
*
***********************************************************************
        SEARCH_LIMIT      5
        DEFAULT_WIDTH    20
        MAPPED_WIDTH     20
        LABEL_TITLE      ^8085 Mnemonic^
        BASE_TITLE       ^    hex^
*
*       Variables used by the inverse assembler
*
```

```
NEW_ADDRESS      VARIABLE    0   ; ADDRESS TO FETCH
LOW_BYTE         VARIABLE
HIGH_BYTE        VARIABLE
LO_INPUT_ERROR   VARIABLE
LO_INPUT_STATUS  VARIABLE
MAP_FLAG         VARIABLE
ADDRESS          VARIABLE
INPUT_MODE       VARIABLE
OPCODE_TEMP      VARIABLE
*
*         Constants used by the inverse assembler
*
ML16             CONSTANT 00000FFFFH

INPUT_ABS        CONSTANT 0
INPUT_REL        CONSTANT 1

STATUS_MASK      CONSTANT 1011B
OPCODE_STATUS    CONSTANT 0011B

UNKNOWN          ASCII "unknown"
MEM_WRITE        ASCII "memory write"
MEM_READ         ASCII "memory read"
IO_WRITE         ASCII "i/o write"
IO_READ          ASCII "i/o read"
INT_ACK          ASCII "interrupt ack"
HALT             ASCII "halt"
*
* DISPLAY FORMATS
*
HEX4_FMT         FORMAT 16,HEX,4      16 BITS, IN HEX, DISPLAY 4 DIGITS
HEX2_FMT         FORMAT 8,HEX,2       8 BITS, IN HEX, DISPLAY 2 DIGITS
```

```
***********************************************************************
*
*  Entry Point of Inverse Assembler
*
***********************************************************************
        SET INPUT_MODE,INPUT_ABS              NORMALLY, ABSOLUTE READ
        SET RETURN_FLAGS,0          INITIALIZE FLAG TO NOT INSTR. LINE
        LOAD INITIAL_ADDRESS              GET ADDRESS TO DISSASSEMBLE
        STORE NEW_ADDRESS
        IF INPUT_ERROR <> 0 THEN GOTO DATA_ERROR          BRANCH IF ERROR
        IF TASK = 3 THEN GOTO ANALYSIS          HP 64620S
        IF TASK = 4 THEN GOTO ANALYSIS          HP 1630A/D & 1631A/D
        IF TASK = 5 THEN GOTO ANALYSIS          HP 1630G, 1650A/B, 1651A/B, 16510A/B, & 16511B
ILLG_TASK
        OUTPUT "Illegal Task Request"
        ABORT
DATA_ERROR
        OUTPUT "Data error"
        ABORT
ILLEGAL_OPCODE
        OUTPUT "Illegal Opcode"
        ABORT
ANALYSIS
        LOAD INPUT_STATUS          GET STATUS OF WHAT WAS READ
        AND STATUS_MASK          MASK OUT HLDA
        IF 3,0 = OPCODE_STATUS THEN GOTO OPCODE_DECODE
*
*  IF ITS NOT AN OPCODE, JUST SHOW THE DATA, FOLLOWED BY THE STATUS
*
        POSITION REL,2
        LOAD INPUT_DATA
        OUTPUT ACCUMULATOR,HEX2_FMT
```

```
MNE_STATUS
*
*  DISPLAY THE STATUS OF THE MNEMONIC
*
*
*  NOTE, THE STATUS TABLE BELOW IS CONSTRUCTED WITH THE S2 BIT (HLDA)
*        AS A DONT CARE.  ACTUALLY, BECAUSE OF THE STATE CLOCKING
*        ARRANGEMENT (INTA or RD or WR), WE WILL NEVER SEE THE CONDITION
*        WHERE HLDA IS HIGH.
      POSITION REL,1
      LOAD INPUT_STATUS
      AND STATUS_MASK              MASK OUT HLDA
      CASE_OF 3,0
         OUTPUT UNKNOWN            STATUS = 0000B
         OUTPUT MEM_WRITE          STATUS = 0001B
         OUTPUT MEM_READ           STATUS = 0010B
         OUTPUT UNKNOWN            STATUS = 0011B
         OUTPUT UNKNOWN            STATUS = 0100B
         OUTPUT MEM_WRITE          STATUS = 0101B
         OUTPUT MEM_READ           STATUS = 0110B
         OUTPUT UNKNOWN            STATUS = 0111B
         OUTPUT HALT               STATUS = 1000B
         OUTPUT IO_WRITE           STATUS = 1001B
         OUTPUT IO_READ            STATUS = 1010B
         OUTPUT INT_ACK            STATUS = 1011B
         OUTPUT HALT               STATUS = 1100B
         OUTPUT IO_WRITE           STATUS = 1101B
         OUTPUT IO_READ            STATUS = 1110B
         OUTPUT INT_ACK            STATUS = 1111B
      CASE_END
      RETURN
OPCODE_DECODE
      SET RETURN_FLAGS,1     FLAG AN INSTRUCTION LINE (AS OPPOSED TO DATA)
      LOAD INPUT_DATA
      CASE_OF 7,6
         GOTO GROUP_1        ;B7-B6=0
         GOTO GROUP_2                1
         GOTO GROUP_3                2
         GOTO GROUP_4                3
      CASE_END
```

```
****************************************************************
*
* Group 1
*     MVI r,exp
*     LXI dr,exp    DAD dr
*     STAX LDAX STA  LDA  SHLD LHLD
*     INR  DCR  INX  DCX
*     RLC  RRC  RAL  RAR  CMA  STC  CMC  DAA
*     NOP  RIM  SIM
*
****************************************************************
GROUP_1
      CASE_OF 2,0
        GOTO SPECIAL_1           00XX X000
        GOTO DAD_LXI             00XX X001
        GOTO LOD_STO             00XX X010
        GOTO INX_DCX             00XX X011
        GOTO INR                 00XX X100
        GOTO DCR                 00XX X101
        GOTO MVI                 00XX X110
        GOTO LOGICAL_1           00XX X111
      CASE_END
****************************************************************
*
* SPECIAL_1 - SPECIAL SYMBOLS
*
****************************************************************
SPECIAL_1
      IF 3,3 = 1 THEN GOTO ILLEGAL_OPCODE     00XX 1000
      CASE_OF 5,4
        OUTPUT "NOP"                 0000 0000
        GOTO ILLEGAL_OPCODE          0001 0000
        OUTPUT "RIM"                 0010 0000
        OUTPUT "SIM"                 0011 0000
      CASE_END
      RETURN
```

```
*****************************************************************
*
* DAD_LXI
*
*****************************************************************
DAD_LXI
      IF 3,3 = 0 THEN GOTO LXI                  00XX 0001
      OUTPUT "DAD  "                            00XX 1001
      CALL LONG_REG
      RETURN
LXI
      OUTPUT "LXI  "
      CALL LONG_REG
      OUTPUT ","
      CALL LEXPR_NO_MAP       DONT MAP 16 BIT DATA QUANTITY
      RETURN
*****************************************************************
*
* LOD_STO
*
*****************************************************************
LOD_STO
      IF 5,5 = 0 THEN GOTO LDAX_STAX            000X X010
      CASE_OF 4,3
        OUTPUT "SHLD"                           0010 0010
        OUTPUT "LHLD"                           0010 1010
        OUTPUT "STA "                           0011 0010
        OUTPUT "LDA "                           0011 1010
      CASE_END
      STORE OPCODE_TEMP       SAVE THE OPCODE ACROSS LEXPR
      POSITION REL,1
      CALL LEXPR
      RETURN
```

```
LDAX_STAX
        CASE_OF 3,3
          OUTPUT "STAX"                             OOOX 0010
          OUTPUT "LDAX"                             OOOX 1010
        CASE_END
        POSITION REL,1
        CALL LONG_REG
        RETURN
*************************************************************
*
* INX_DCX
*
*************************************************************
INX_DCX
        CASE_OF 3,3
          OUTPUT "INX"                              OOXX 0011
          OUTPUT "DCX"                              OOXX 1011
        CASE_END
        POSITION REL,2
        CALL LONG_REG
        RETURN
*************************************************************
*
* INR
*
*************************************************************
INR
        OUTPUT "INR  "
        CALL DREG_NAME
        RETURN
```

```
*****************************************************************
*
* DCR
*
*****************************************************************
DCR
      OUTPUT "DCR  "
      CALL DREG_NAME
      RETURN
*****************************************************************
*
* MVI
*
*****************************************************************
MVI
      OUTPUT "MVI  "
      CALL DREG_NAME
      OUTPUT ","
      CALL EXPR
      RETURN
*****************************************************************
*
* LOGICAL_1
*
*****************************************************************
LOGICAL_1
      CASE_OF 5,3
        OUTPUT "RLC"                         0000 0111
        OUTPUT "RRC"                         0000 1111
        OUTPUT "RAL"                         0001 0111
        OUTPUT "RAR"                         0001 1111
        OUTPUT "DAA"                         0010 0111
        OUTPUT "CMA"                         0010 1111
        OUTPUT "STC"                         0011 0111
        OUTPUT "CMC"                         0011 1111
      CASE_END
      POSITION REL,2
      RETURN
```

```
**************************************************************
*
* Group 2
*
**************************************************************
GROUP_2
        IF 7,0 <> 01110110B THEN GOTO MOVES     BRANCH IF NOT HALT INSTR
           OUTPUT "HLT"
           RETURN
MOVES
        OUTPUT "MOV  "
        CALL DREG_NAME
        OUTPUT ","
        CALL SREG_NAME
        RETURN
**************************************************************
*
* Group 3
*
**************************************************************
GROUP_3
        CASE_OF 5,3
          OUTPUT "ADD"                 1000 0XXX
          OUTPUT "ADC"                 1000 1XXX
          OUTPUT "SUB"                 1001 0XXX
          OUTPUT "SBB"                 1001 1XXX
          OUTPUT "ANA"                 1010 0XXX
          OUTPUT "XRA"                 1010 1XXX
          OUTPUT "ORA"                 1011 0XXX
          OUTPUT "CMP"                 1011 1XXX
        CASE_END
        POSITION REL,2
```

```
          CALL SREG_NAME
          RETURN
****************************************************************
*
* Group 4
*
****************************************************************
GROUP_4
        CASE_OF 2,0
             GOTO Rcc                      11XX X000
             GOTO POP_RET                  11XX X001
             GOTO Jcc                      11XX X010
             GOTO JMP_IO                   11XX X011
             GOTO Ccc                      11XX X100
             GOTO PUSH_CALL                11XX X101
             GOTO Immediate                11XX X110
             GOTO RST                      11XX X111
        CASE_END
        GOTO ILLEGAL_OPCODE
**************************************************************
*
* Rcc
*
**************************************************************
Rcc
        OUTPUT "R"
        CALL COND_CODES
        RETURN
**************************************************************
*
* POP_RET Etc.
*
**************************************************************
```

```
POP_RET
        IF 3,3 = 1 THEN GOTO RET_PCHL_SPHL      11XX 1001
        OUTPUT "POP  "                          11XX 0001
        CALL LONG_REG
        RETURN
RET_PCHL_SPHL
        CASE_OF 5,4
            OUTPUT "RET  "                      1100 1001
            GOTO   ILLEGAL_OPCODE               1101 1001
            OUTPUT "PCHL "                      1110 1001
            OUTPUT "SPHL "                      1111 1001
        CASE_END
        RETURN
********************************************************
*
* Jcc
*
********************************************************
Jcc
        OUTPUT "J"
        CALL COND_CODES
        POSITION REL,2
        CALL LEXPR              GET 2 BYTES
        RETURN
********************************************************
*
* JMP_IO Etc.
*
********************************************************
JMP_IO
        CASE_OF 5,3
            GOTO JMP                            1100 0011
            GOTO ILLEGAL_OPCODE                1100 1011
            GOTO OUT_IN                         1101 0011
            GOTO OUT_IN                         1101 1011
```

```
            OUTPUT "XTHL "                       1110 0011
            OUTPUT "XCHG "                       1110 1011
            OUTPUT "DI   "                       1111 0011
            OUTPUT "EI   "                       1111 1011
      CASE_END
      RETURN
JMP    OUTPUT "JMP  "
      CALL LEXPR
      RETURN
OUT_IN
      CASE_OF 3,3
            OUTPUT "OUT  "                       1101 0011
            OUTPUT "IN   "                       1101 1011
      CASE_END
      CALL EXPR
      RETURN
***********************************************************
*
* Ccc
*
***********************************************************
Ccc
      OUTPUT "C"
      CALL COND_CODES
      POSITION REL,2
      CALL LEXPR
      RETURN
***********************************************************
*
* PUSH_CALL
*
***********************************************************
PUSH_CALL
      IF 3,3 = 0 THEN GOTO PUSH              11XX 0101
```

```
            IF 5,3 <> 001 THEN GOTO ILLEGAL_OPCODE
            OUTPUT "CALL "                      1100 1101
            CALL LEXPR
            RETURN
PUSH    OUTPUT "PUSH "
            CALL LONG_REG
            RETURN
**********************************************************
*
* Immediate
*
**********************************************************
Immediate
            CASE_OF 5,3
                OUTPUT "AD"                     1100 0110
                OUTPUT "AC"                     1100 1110
                OUTPUT "SU"                     1101 0110
                OUTPUT "SB"                     1101 1110
                OUTPUT "AN"                     1110 0110
                OUTPUT "XR"                     1110 1110
                OUTPUT "OR"                     1111 0110
                OUTPUT "CP"                     1111 1110
            CASE_END
            OUTPUT "I   "
            CALL EXPR
            RETURN
**********************************************************
*
* RST
*
**********************************************************
RST
            OUTPUT "RST   "
            CASE_OF 5,3
                OUTPUT "0"                       1100 0111
                OUTPUT "1"                       1100 1111
```

```
              OUTPUT "2"                          1101 0111
              OUTPUT "3"                          1101 1111
              OUTPUT "4"                          1110 0111
              OUTPUT "5"                          1110 1111
              OUTPUT "6"                          1111 0111
              OUTPUT "7"                          1111 1111
         CASE_END
         RETURN
***********************************************************
*
* COND_CODES - OUTPUT CONDITION CODES
*
***********************************************************
COND_CODES
         CASE_OF 5,3
              OUTPUT "NZ"                    XX00 0XXX
              OUTPUT "Z "                    XX00 1XXX
              OUTPUT "NC"                    XX01 0XXX
              OUTPUT "C "                    XX01 1XXX
              OUTPUT "PO"                    XX10 0XXX
              OUTPUT "PE"                    XX10 1XXX
              OUTPUT "P "                    XX11 0XXX
              OUTPUT "M "                    XX11 1XXX
         CASE_END
         RETURN
***********************************************************
*
*  DREG_NAME - OUTPUT DESTINATION REGISTER NAME
*
***********************************************************
DREG_NAME
         CASE_OF 5,3
              OUTPUT "B"                     XX00 0XXX
              OUTPUT "C"                     XX00 1XXX
              OUTPUT "D"                     XX01 0XXX
              OUTPUT "E"                     XX01 1XXX
              OUTPUT "H"                     XX10 0XXX
```

```
            OUTPUT "L"                          XX10 1XXX
            OUTPUT "M"                          XX11 0XXX
            OUTPUT "A"                          XX11 1XXX
        CASE_END
        RETURN
***********************************************************
*
*  SREG_NAME - OUTPUT SOURCE REGISTER NAME
*
***********************************************************
SREG_NAME
        CASE_OF 2,0
            OUTPUT "B"                          XXXX X000
            OUTPUT "C"                          XXXX X001
            OUTPUT "D"                          XXXX X010
            OUTPUT "E"                          XXXX X011
            OUTPUT "H"                          XXXX X100
            OUTPUT "L"                          XXXX X101
            OUTPUT "M"                          XXXX X110
            OUTPUT "A"                          XXXX X111
        CASE_END
        RETURN
***********************************************************
*
* LONG_REG OUTPUT LONG REGISTER NAME
*
***********************************************************
LONG_REG
        CASE_OF 5,4
            OUTPUT "B"                          XX00 XXXX
            OUTPUT "D"                          XX01 XXXX
            OUTPUT "H"                          XX10 XXXX
            OUTPUT "SP"        POSSIBLY OVERWRITTEN BY PSW CONDITION
        CASE_END
        IF 7,4 <> 0FH THEN RETURN               1111 XXXX
        POSITION REL,-2
        OUTPUT "PSW"
        RETURN
```

```
************************************************************
*
*  LEXPR  OUTPUT 16-BIT HEX VALUE IN NEXT TWO BYTES
*
************************************************************
LEXPR
        SET MAP_FLAG,0          PRESET TO DO ADDRESS MAPPING
        GOTO MAP_SET

LEXPR_NO_MAP
        SET MAP_FLAG,1          PRESET FOR NO MAPPING
MAP_SET
        CALL NEXT_BYTE          GET LOW BYTE OF DATA
        STORE LOW_BYTE          SAVE THE LOW ORDER BYTE
        LOAD INPUT_ERROR        GET ERROR FLAG
        STORE LO_INPUT_ERROR    SAVE ERROR FLAG IN TEMPORARY
        LOAD INPUT_STATUS
        AND STATUS_MASK         MASK OUT HLDA
        STORE LO_INPUT_STATUS

        CALL NEXT_BYTE          GET HIGH BYTE OF DATA
        STORE HIGH_BYTE         SAVE THE HIGH ORDER BYTE

        ROTATE LEFT,8           MOVE HIGH BYTE TO UPPER 8 BITS
        INCLUSIVE_OR LOW_BYTE   PUT THE 2 BYTES TOGETHER
        AND ML16                ONLY LOWER 16 BITS ARE VALID ADDRESS
        STORE ADDRESS           SAVE THE 16 BIT ADDRESS

        LOAD INPUT_STATUS
        AND STATUS_MASK         MASK OUT HLDA
        IF 3,0 = OPCODE_STATUS THEN GOTO HI_WAS_OPCODE
        IF LO_INPUT_STATUS = OPCODE_STATUS THEN GOTO NOT_ASSOCIATED
        IF INPUT_ERROR = 0 THEN GOTO SHOW_CHECK     BRANCH IF NO DATA READ ERROR
HI_WAS_OPCODE
NOT_ASSOCIATED
        OUTPUT "**"             NO HIGH BYTE FOUND
        SET MAP_FLAG,1          SET TO SKIP ADDRESS MAPPING
        GOTO CHECK_LOW          CHECK FOR THE LOW ORDER BYTE
```

```
SHOW_CHECK
        IF MAP_FLAG = 0 THEN GOTO CHECK_LOW        BRANCH IF STILL MAPPING
            LOAD HIGH_BYTE                GET UPPER BYTE
            OUTPUT ACCUMULATOR,HEX2_FMT   DISPLAY THE HIGH BYTE

CHECK_LOW
        IF LO_INPUT_STATUS = OPCODE_STATUS THEN GOTO LO_WAS_OPCODE
        IF LO_INPUT_ERROR = 0 THEN GOTO SHOW_LOW     BRIF NO DATA READ ERROR

LO_WAS_OPCODE
            OUTPUT "**"          NO LOW BYTE FOUND
            RETURN               NO MAPPING, WE ARE FINISHED

SHOW_LOW
        IF MAP_FLAG = 0 THEN GOTO MAPPER        BRANCH IF MAPPING ALLOWED
            LOAD LOW_BYTE                       GET THE LOW ORDER BYTE
            OUTPUT ACCUMULATOR,HEX2_FMT         DISPLAY THE LOWER BYTE
            RETURN                              LEAVE

MAPPER
        LOAD ADDRESS          GET THE 16 BIT ADDRESS
        IF_NOT_MAPPED THEN OUTPUT ACCUMULATOR,HEX4_FMT
        RETURN

***********************************************************

*

*  EXPR  OUTPUT 8-BIT HEX VALUE IN NEXT BYTE

*

***********************************************************

EXPR

        CALL NEXT_BYTE        GET THE BYTE AFTER THE OPCODE
        LOAD INPUT_STATUS     destroys data in accumulator from NEXT_BYTE
        AND STATUS_MASK       MASK OUT HLDA
        IF 3,0 <> OPCODE_STATUS THEN GOTO EXPR_HEX2   BRANCH IF DATA FOUND
            OUTPUT "**"       FLAG THE BYTE AS NOT FOUND
            RETURN

EXPR_HEX2
        LOAD INPUT_DATA       RELOAD DATA
```

```
        OUTPUT ACCUMULATOR,HEX2_FMT
        RETURN
************************************************************
*
* GET NEXT BYTE FROM ANALYSIS DATA
*  INCREMENTS INPUT_ADDRESS, NEW_ADDRESS AND RETURN_COUNT
*
************************************************************
NEXT_BYTE
        INCREMENT NEW_ADDRESS        MOVE AHEAD TO NEXT ADDRESS
GET_BYTE
*
*   ENTRY POINT TO READ THE DESIRED BYTE WITHOUT INCREMENTING NEW_ADDRESS
*
        LOAD INPUT_MODE        SEE WHICH INPUT MODE WE'RE IN
        CASE_OF 0,0
          INPUT ABS,NEW_ADDRESS            READ THE DATA
          INPUT REL,NEW_ADDRESS
        CASE_END
        LOAD INPUT_DATA              SET ACCUMULATOR TO DATA READ
        RETURN
```

Microprocessors with Incomplete Status

Introduction

The Intel 8085 instructions decoded in chapters 2 and 3 were easy to disassemble because of the status information provided by the microprocessor. In the 8085, the value 0011B under the STAT label indicates that this state is the first state of an opcode fetch. Any other kind of bus cycle has a different status.

When disassembling 8085 operations, the inverse assembler goes through the following steps:

1. Load INPUT_STATUS for the state to be disassembled.

2. If the value of INPUT_STATUS represents an opcode fetch, go to the routine for decoding opcode fetches.

3. If the value of INPUT_STATUS indicates any other kind of bus cycle, display only the value in INPUT_DATA and the cycle type.

Not all microprocessors provide a unique value under the STAT label for the first byte of an opcode fetch. This appendix explains how to deal with this problem.

Note 👆 This section uses examples based on the Motorola 68010. A complete listing of the 68010 inverse assembler is provided in appendix C.

Using INPUT_TAG to Mark States

The Motorola 68010 microprocessor does not uniquely identify which bus cycle contains the first word of an instruction fetch. For example, the 68010 instruction:

JMP 00255A

would be captured as follows by the logic analyzer:

Label	>	ADDR	DATA	68010 Mnemonic	STAT
Base	>	Hex	Hex	Hex	Symbol
+ 0000		00201E	4EF9	JMP 00255A	SUPR PRGM READ
+ 0001		002020	0000	0000 supr program read	SUPR PRGM READ
+ 0002		002022	255A	255A supr program read	SUPR PRGM READ
+ 0003		00255A		(Next 68010 instruction)	

.
.
.

State 0000 above contains the opcode for the JMP instruction under the DATA label. States 0001 and 0002 contain the destination address of the JMP opcode. Note that all three of these states have the same value under the STAT label. (In this example, the STAT value has been interpreted by the STAT symbol table to make it easier to read. Also, the raw data captured by the logic analyzer is displayed under the data label next to the mnemonic field.)

If the inverse assembler follows the same procedure as the 8085, here is what will happen, beginning with line 0000:

1. Load INPUT_STATUS for state 0000. Since this value represents a program read, go to the routine that decodes opcode fetches.

2. Decode the 4EF9H in INPUT_DATA. This will put the ASCII string

JMP

into the output display buffer.

3. Look forward into states 0001 and 0002 to get the destination address for JMP. This is done using the INPUT,ABS instruction.

4. Calculate the destination address for the JMP instruction and put it into the output display buffer. At this point, the disassembly process for state 0000 is complete. The following is in the output display buffer:

JMP   00255A

Using the same process for state 0001:

1. Load INPUT_STATUS for state 0001. Since this value represents a program read, go to the routine that decodes opcode fetches.

2. Decode the 0000H in INPUT_DATA. This will put the ASCII string

ORI.B

into the output display buffer.

Step 2 is incorrect for state 0001. State 0001 is not the ORI.B opcode. It is part of the operand for the opcode in state 0000. This confusion is a direct result of the limited status information generated by the 68010. The information in STAT must be supplemented to differentiate between a state that contains an opcode fetch and a state that contains operands.

The communication variable INPUT_TAG can be used to supplement the information provided by INPUT_STATUS. Each state has a tag associated with it. The inverse assembler can use the lower 16 bits of the tag to mark a state for later reference. In the case of the 68010, the tags are used to mark a state as an operand that has already been decoded.

To mark the tags for a state, use the TAG_WITH instruction. The tags are read through the INPUT_TAG communication variable.

Here is how the 68010 disassembly process would operate when using the tag bits to supplement the INPUT_STATUS information:

1. Load INPUT_STATUS for state 0000. Since this value represents a program read, go to the routine that decodes opcode fetches.

2. Load INPUT_TAG for state 0000. Check if this state was tagged to supplement the INPUT_STATUS information. Since this state is the first state of an opcode fetch, the tag value of bits 0 through 5 will be 0.

3. Decode the 4EF9H in INPUT_DATA. This will put the ASCII string

JMP

into the output display buffer.

4. Look forward into states 0001 and 0002 to get the destination address for JMP. This is done using the INPUT,ABS instruction. Tag states 0001 and 0002 with the TAG_WITH instruction to indicate that they have already been used as an operand for decoding state 0000.

5. Calculate the destination address for the JMP instruction and put it into the output display buffer. At this point, the disassembly process for state 0000 is complete. The following is in the output display buffer:

JMP   00255A

These same steps are used for decoding state 0001.

1. Load INPUT_STATUS for state 0001. Since this value represents a program read, go to the routine that decodes opcode fetches.

2. Load INPUT_TAG for state 0001. Check if this state was tagged to supplement the INPUT_STATUS information. In this case, state 0001 was tagged as an operand when state 0000 was disassembled. Branch to a routine that will display only the value of INPUT_DATA and the type of bus cycle.

3. Read INPUT_DATA and put its value in the output display buffer. Read INPUT_STATUS and decode its value into the cycle type. Put the cycle type into the output display buffer to finish disassembling state 0001. At this time the output display buffer will hold:

0000 supr program read

Using the tags associated with each state allows the inverse assembler to supplement the status provided by the 68010. This supplement allows the inverse assembler to correctly decode the operands of an instruction.

Software Compatibility with other Logic Analyzers

As mentioned in chapter 3, Hewlett-Packard's Inverse Assembly Language is compatible with several different logic analyzers and emulators. The TASK communication variable is used to identify which environment the inverse assembler is running in.

The HP 1630 and HP 1631 logic analyzers do not use bits 2 through 15 of the INPUT_TAG. User-tags in those instruments are limited to bits 0 and 1. If the source code will be used for more than one family of logic analyzers, the inverse assembler should only use bits 0 and 1 of the INPUT_TAG to insure compatibility.

Synchronizing the Inverse Assembler to the Captured Data

In the previous section, tag bits 0 through 5 were used to help differentiate between the first state of an opcode fetch and the operands for that fetch. Unfortunately, this solution only solves some of the problems caused by microprocessors with incomplete status.

In the previous example, line 0000 was assumed to be the first state of an opcode fetch. This was an arbitrary assumption; in general there is no way to determine which state contains the first line of an opcode fetch when inverse assembling 68010 bus cycles. As previously shown, once the inverse assembler is synchronized to the information captured by the logic analyzer, the inverse assembler can work properly, using the tag bits.

The problem, then, is to find some way of synchronizing the inverse assembler to captured data. Once synchronized , the tag bits can be used to supplement the status provided by the microprocessor to insure correct inverse assembly.

The "Invasm" Field

The HP 1650A/B, HP 1651A/B, HP 16510A/B, and HP 16511B Logic Analyzers provide a field to allow you to manually synchronize the inverse assembler to the captured data. This field is called "Invasm" and is located in the middle of the first line of the Display menu.

Note 👆

The "Invasm" field is not normally displayed in the Display menu. When an inverse assembler is downloaded to the logic analyzer disk, you can specify if this field is needed. If the field is needed by the inverse assembler, it will be turned on when the inverse assembler is loaded into the logic analyzer.

The following steps explain how to use the "Invasm" field to synchronize the inverse assembler to the captured data:

1. Identify a state on the logic analyzer screen that is the first state of an instruction fetch. This can often be determined from the address and status captured in the state.

2. Roll this state up to the top line on the state listing.

3. Select the "Invasm" field. This tells the inverse assembler that the top line on the state listing is the first state of an opcode fetch. The inverse assembler is now synchronized to the captured data.

The "Invasm" field is an inverse assembler control feature used to synchronize the inverse assembler to the captured data. It uses INPUT_TAG bits 16 and 17 to communicate with the inverse assembler.

Note ☝

INPUT_TAG bits 0 through 15 are user tags. As previously discussed, the inverse assembler can read and write these tags to mark states. INPUT_TAG bits 16 and 17 are system tags and are completely under the control of the logic analyzer. The inverse assembler has READ ONLY access to bits 16 and 17 of INPUT_TAG.

Inverse assemblers that do not require the "Invasm" field to synchronize to captured data can completely ignore bits 16 and 17.

INPUT_TAG Values and How They Change

INPUT_TAG bits 16 and 17 are interpreted as shown in the following table:

Bit 17	Bit 16	Description
0	0	Not disassembled and is ILLEGAL for the inverse assembler to do so. The "Invasm" field has not been selected.
0	1	Not disassembled and is legal for the inverse assembler to do so. The "Invasm" field has been selected.
1	0	Disassembled and is not the first state of an opcode fetch.
1	1	Disassembled and is the first state of an opcode fetch.

After the logic analyzer has made a measurement, INPUT_TAG bits 16 and 17 start with a value of 0. At this point, the "Invasm" field has not been selected to synchronize the inverse assembler. None of the lines on the screen have been disassembled. In fact, it is illegal for the inverse assembler to disassemble a state when bits 16 and 17 are 0. Instead, the inverse assembler should be written to display only the value in INPUT_DATA and the kind of bus cycle indicated by the value in INPUT_STATUS.

For the 68010 example, here is what should be on the logic analyzer display before the "Invasm" field is selected:

Label	>	ADDR	DATA	68010 Mnemonic	STAT
Base	>	Hex	Hex	Hex	Symbol
+ 0000		00201E	4EF9	4EF9 supr program read	SUPR PRGM READ
+ 0001		002020	0000	0000 supr program read	SUPR PRGM READ
+ 0002		002022	255A	255A supr program read	SUPR PRGM READ
+ 0003		00255A		(Next 68010 instruction)	

.
.
.

When the "Invasm" field is selected, the analyzer sets INPUT_TAG bit 17 to 0 and bit 16 to 1 for the first state on the display. This indicates to the inverse assembler that it can disassemble this state. After doing so, the inverse assembler returns control to the analyzer. At this time, the analyzer tags the NEXT state with bit 17 equal to 0 and bit 16 equal to 1 (legal to disassemble). With this method, the remainder of the trace list can be disassembled using a single selection of the "Invasm" field. Rolling forward will disassemble the last line on the screen as it scrolls onto the display.

Note 👆

The logic analyzer only disassembles states on the instrument screen. Jumping to a different area of the trace memory will cause the inverse assembler to loose synchronization with the captured data. Also, rolling the screen backwards to display lines before the first state that was disassembled will cause previous lines to not be disassembled.

In both of these cases, you must re-synchronize the inverse assembler by repeating the process shown on page B-7.

In the 68010 inverse assembler, the first communication variable checked is INPUT_TAG bits 16 and 17. If the value in bits 16 and 17 is 0, the inverse assembler will only display the value of INPUT_DATA and the cycle type. For any other value, the inverse assembler will try to disassemble the state.

Here are the steps executed on state 0000 in the previous example if the "Invasm" field has not been selected:

1. Load INPUT_TAG for state 0000. Check the value in bits 16 and 17. Since "Invasm" has not been selected, the value here will be 0 (Not safe to inverse assemble). Branch to a routine that will display only the value of INPUT_DATA and the type of bus cycle.

2. Load INPUT_DATA and put its value in the output display buffer. Read INPUT_STATUS and decode its value into the cycle type. Put the cycle type into the output display buffer to finish disassembling state 0000. At this time the output display buffer will hold:

4EF9 supr program read

To synchronize the inverse assembler to the acquired information, you should scroll state 0000 to the top of the listing. When "Invasm" is selected, the inverse assembler does the following for this state:

1. Load INPUT_TAG for state 0000. Check the value in bits 16 and 17. Since "Invasm" has been selected, the value here will be 01B (Safe to inverse assemble). Branch to the routine that decodes opcodes.

2. Check bits 0 through 15 of INPUT_TAG to see if this state was tagged to supplement the INPUT_STATUS information. Since this state is the first state of an opcode fetch, the tag value of bits 0 through 15 will be 0.

3. Load INPUT_STATUS for state 0000. Since this value represents a program read, continue to decode this state as an opcode fetch.

4. Decode the 4EF9H in INPUT_DATA. This will put the ASCII string

JMP

into the output display buffer.

5. Look forward into states 0001 and 0002 to get the destination address for JMP. This is done using the INPUT,ABS instruction. Tag states 0001 and 0002 with the TAG_WITH instruction to indicate that they have already been used as an operand for decoding state 0000.

6. Calculate the destination address for the JMP instruction and put it into the output display buffer. At this point, the disassembly process for state 0000 is complete. The following is in the output display buffer:

JMP   00255A

7. When the inverse assembler returns control to the logic analyzer, tag bits 16 and 17 of the next state (state 0001) will be set to 01B.

The logic analyzer now calls the inverse assembler to decode the next state. The disassembler will go through the following steps for state 0001:

1. Load INPUT_TAG for state 0001. Check the value in bits 16 and 17. Since "Invasm" has been selected, the value here will be 01B (Safe to inverse assemble). Branch to the routine that decodes opcodes.

2. Load INPUT_TAG for state 0001. Check if this state was tagged to supplement the INPUT_STATUS information. In this case, state 0001 was tagged as an operand when state 0000 was disassembled. Branch to a routine that will display only the value of INPUT_DATA and the type of bus cycle.

3. Read INPUT_DATA and put its value in the output display buffer. Read INPUT_STATUS and decode its value into the cycle type. Put the cycle type into the output display buffer to finish disassembling state 0001. At this time the output display buffer will hold:

0000 supr program read

4. When the inverse assembler returns control to the logic analyzer, tag bits 16 and 17 of state 0002 will be set to 01B.

Using RETURN_FLAGS

In chapter 3, bits 16 and 17 of the RETURN_FLAGS communication variable were used to indicate the results of an IF_NOT_MAPPED instruction. RETURN_FLAGS can also be used to mark states that are the first state in an opcode fetch.

Executing the

SET RETURN_FLAGS,0

instruction in the inverse assembler indicates to the logic analyzer that the current state being decoded DOES NOT CONTAIN the first state of an instruction fetch. Executing

SET RETURN_FLAGS,1

indicates that the current state DOES CONTAIN the first state of an instruction fetch.

Note 👈

The commands above are setting RETURN_FLAGS, bit 0 to a 0 or a 1.

Setting bit 0 of RETURN_FLAGS also affects bits 16 and 17 of INPUT_TAG. When bit 0 of RETURN_FLAGS is set by using the set RETURN_FLAGS statement, bit 17 of INPUT_TAG is set to 1. Bit 16 of INPUT_TAG is set to the value of RETURN_FLAGS bit 0.

When bits 16 and 17 of INPUT_TAG are set to a 10B or 11B, it indicates the state has already been disassembled. In most cases, the inverse assembler should not try to inverse assemble a line that has already been decoded.

To complete the 68010 example, then, steps must be added to mark bit 0 of RETURN_FLAGS to insure a line will not be inverse assembled twice. For state 0000, the following steps would be performed:

1. Load INPUT_TAG for state 0000. Check the value in bits 16 and 17. Since "Invasm" has been selected, the value here will be 01B (Safe to inverse assemble). Branch to the routine that decodes opcodes.

2. Check bits 0 through 15 of INPUT_TAG to see if this state was tagged to supplement the INPUT_STATUS information. Since this state is the first state of an opcode fetch, the tag value of bits 0 through 15 will be 0.

3. Load INPUT_STATUS for state 0000. Since this value represents a program read, continue to decode this state as an opcode fetch.

4. Set RETURN_FLAGS bit 0 to 1, to indicate that this state is the first state of an instruction fetch.

5. Decode the 4EF9H in INPUT_DATA. This will put the ASCII string

JMP

into the output display buffer.

6. Look forward into states 0001 and 0002 to get the destination address for JMP. This is done using the INPUT,ABS instruction. If bits 16 and 17 of INPUT_TAG for states 0001 and 0002 are marked as "already disassembled", do not complete the inverse assembly for state 0000. In this case, these tag bits are still set to 0.

7. Tag states 0001 and 0002 with the TAG_WITH instruction to indicate that they have already been used as an operand for decoding state 0000. This statement is operating on bits 0 through 15 of the INPUT_TAG.

8. Calculate the destination address for the JMP instruction and put it into the output display buffer. At this point, the disassembly process for state 0000 is complete. The following is in the output display buffer:

JMP   00255A

9. When the inverse assembler returns control to the logic analyzer, tag bits 16 and 17 of the next state (state 0001) will be set to 01B. Tag bits 16 and 17 of this state (state 0000) are set to 11B.

The steps for state 0001 are similar:

1. Load INPUT_TAG for state 0001. Check if this state was tagged to supplement the INPUT_STATUS information. In this case, state 0001 was tagged as an operand when state 0000 was disassembled. Branch to a routine that will display only the value of INPUT_DATA and the type of bus cycle.

2. Set RETURN_FLAGS bit 0 to 0, to indicate that this state is not the first state of an instruction fetch.

3. Read INPUT_DATA and put its value in the output display buffer. Read INPUT_STATUS and decode its value into the cycle type. Put the cycle type into the output display buffer to finish disassembling state 0001. At this time the output display buffer will hold:

0000 supr program read

4. When the inverse assembler returns control to the logic analyzer, tag bits 16 and 17 of state 0002 will be set to 01B. Tag bits 16 and 17 of this state (state 0001) are set to 10B.

Summary of INPUT_TAGS Bits 16 and 17

In summary, INPUT_TAG bits 16 and 17 are used with the "Invasm" field in the logic analyzer Display menu to allow you to "point" to the line that contains the first state of an opcode fetch. This allows the inverse assembler to synchronize to the captured data on microprocessors that provide incomplete status.

Bits 16 and 17 are interpreted as follows:

Bit 17	Bit 16	Description
0	0	Not disassembled and is ILLEGAL for the inverse assembler to do so. The "Invasm" field has not been selected.
0	1	Not disassembled and is legal for the inverse assembler to do so. The "Invasm" field has been selected.
1	0	Disassembled and is not the first state of an opcode fetch. This value was generated when the SET RETURN_FLAGS,0 instruction was executed.
1	1	Disassembled and is the first state of an opcode fetch. This value was generated when the SET RETURN_FLAGS,1 instruction was executed.

The "Invasm" field with no options, as discussed here, is enabled by selecting "B" for the final input of the IALDOWN program. See chapter 1 for additional information.

States Containing Multiple Opcodes

In the 68010 example, the start of an opcode fetch could be identified by pointing to a specific state. For some microprocessors, simply pointing to a state may not be sufficient to uniquely identify where an opcode starts.

For example, the Motorola 68020 is a 32-bit microprocessor. Opcodes can start in the high or low 16-bit word contained in a 68020 32-bit fetch. In this case you need to point to the high or low word of a specific state.

As another example, the Intel 80386 is a 32-bit microprocessor where an opcode can start in any of the four byte positions of the 32-bit fetch. In addition, the 80386 can run two kinds of object code: object code originally designed for Intel's 16-bit microprocessors (such as the 8086) or object code designed specifically for the 80386 32-bit instruction set. So, when pointing to a state that holds the start of an instruction fetch, you must point to one of four bytes in the 32-bit word. Also, you must indicate if the instructions are from Intel's 16- or 32-bit instruction set.

The following section will describe how to handle both of the situations described above.

The "Invasm" Field Revisited

When discussing the 68010 inverse assembler, the "Invasm" field was used to synchronize the inverse assembler to the captured information. In the case discussed, selecting the "Invasm" field set INPUT_TAGS bits 16 and 17 to 01B.

As previously noted, the "Invasm" field is normally not displayed in the Display Menu. When an inverse assembler is downloaded to the logic analyzer disk, you can specify if this field is needed. If the field is needed by the inverse assembler, it will be turned on when the inverse assembler is loaded into the logic analyzer.

The download program, IALDOWN, provides four options related to the "Invasm" field. When downloading the inverse assembler, IALDOWN will ask which option is required for the "Invasm" field. The options available are:

Option	Description
A	No "Invasm" field needed. Do not display. This was the option selected for the 8085 inverse assembler.
B	"Invasm" field needed. No other options required. This was the option selected for the 68010 inverse assembler.
C	"Invasm" field needed. When "Invasm" is selected, provide a pop-up on the logic analyzer screen that provides the following two choices: High Low
D	"Invasm" field needed. When "Invasm" is selected, provide a pop-up on the logic analyzer screen that provides the following eight choices: Size 16, Byte 0 Size 16, Byte 1 Size 16, Byte 2 Size 16, Byte 3 Size 32, Byte 0 Size 32, Byte 1 Size 32, Byte 2 Size 32, Byte 3

For options B, C, and D, bits 16 and 17 of INPUT_TAG are interpreted as follows:

Bit 17	Bit 16	Description
0	0	Not disassembled and is ILLEGAL for the inverse assembler to do so. The "Invasm" field has not been selected.
0	1	Not disassembled and is legal for the inverse assembler to do so. The "Invasm" field has been selected.
1	0	Disassembled and is not the first state of an opcode fetch. This value was generated when the SET RETURN_FLAGS,0 instruction was executed.
1	1	Disassembled and is the first state of an opcode fetch. This value was generated when the SET RETURN_FLAGS,1 instruction was executed.

For options C and D, the communication variables INITIAL_FLAGS and INITIAL_OPTIONS are used to tell the inverse assembler which field was selected in the pop-up.

For both options C and D, bit 1 of INITIAL_FLAGS is set to 1 if a field has been selected.

For option C, INITIAL_OPTIONS bit 0 is set as follows:

Bit 0	Field selected in pop-up
0	High
1	Low

For option D, INITIAL_OPTIONS bits 0 through 3 are set as follows:

Bit 3	Bit 2	Bit 1	Bit 0	Field selected in pop-up
0	1	0	0	Size 16, Byte 0
0	1	0	1	Size 16, Byte 1
0	1	1	0	Size 16, Byte 2
0	1	1	1	Size 16, Byte 3
1	0	0	0	Size 32, Byte 0
1	0	0	1	Size 32, Byte 1
1	0	1	0	Size 32, Byte 2
1	0	1	1	Size 32, Byte 3

All other combinations of bits 0 through 3 in INITIAL_OPTIONS are unimplemented.

The inverse assembler can use this additional information when parsing through an opcode. For instance, when disassembling the Motorola 68020, option C is used. If the Low field is selected in the "Invasm" pop-up, the inverse assembler will ignore the high word of INPUT_DATA and begin inverse assembly with the low word of INPUT_DATA.

Figure B-1 is an overview of the "Invasm" field.

Code Synchronization with the HP 1630/31 Logic Analyzers

The HP 1630 family of logic analyzers do not support options C and D of the download program IALDOWN. Only the simple synchronization using option B is available with these instruments. If the source code will also be used for the HP 1630A/D/G or HP 1631A/D, do not use the INITIAL_FLAGS or INITIAL_OPTIONS communication variables.
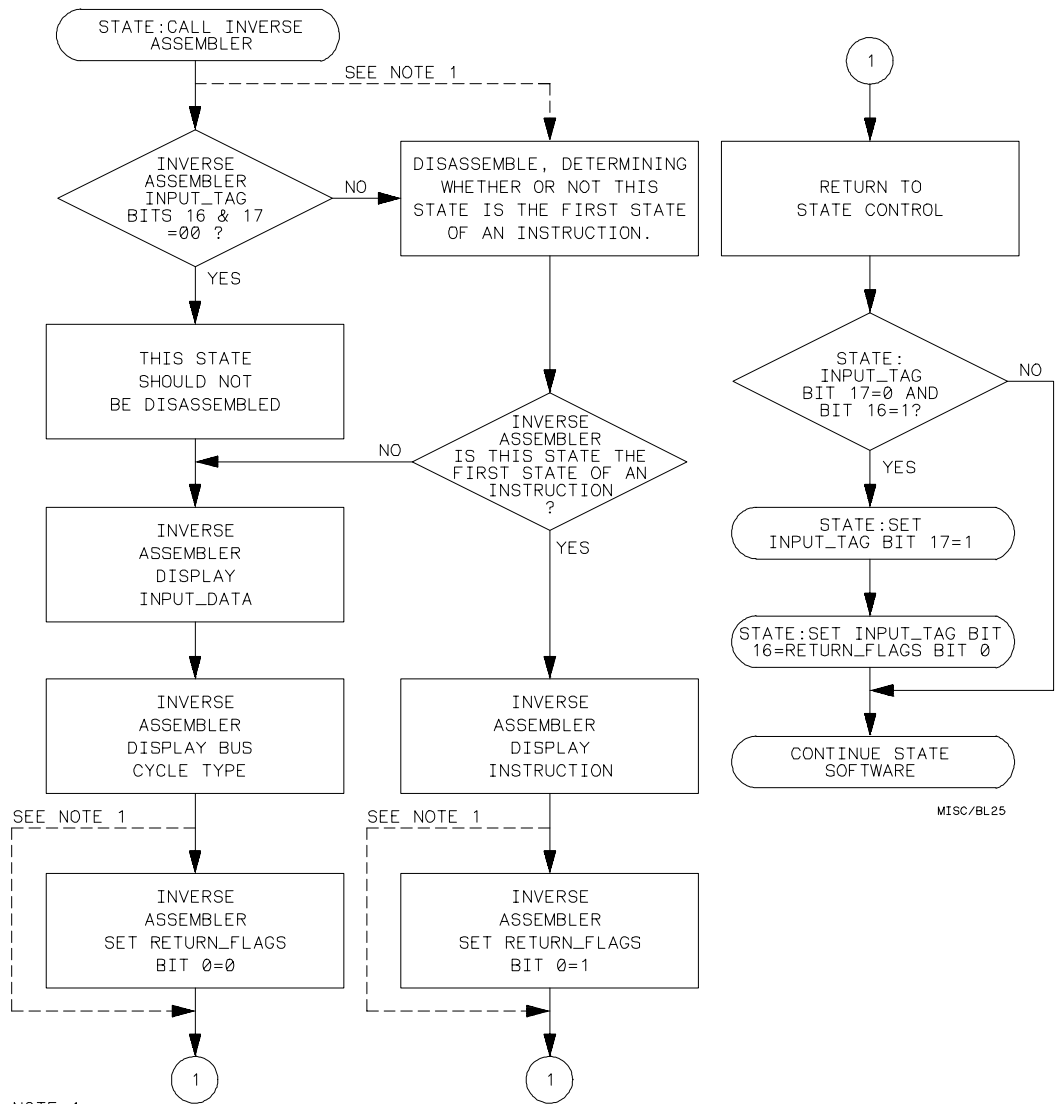
Figure B-1. Invasm Field Mechanism

68010
Inverse Assembler

```
"IAL"


**********************************************************************
*
*   INVERSE ASSEMBLER FOR THE 68010 MICROPROCESSOR
*
**********************************************************************
*
*   THE LOGIC ANALYZER CAPTURES 24 ADDRESS LINES, 16 DATA LINES AND
*   8 STATUS LINES ON THE RISING EDGE OF LAS.
*
*   THE 8 STATUS LINES FOR THIS INVERSE ASSEMBLER ARE:
*
*           BIT 0  ---  R/LW  (CPU PIN 9)
*           BIT 1  ---  LLDS  (CPU PIN 8)
*           BIT 2  ---  LUDS  (CPU PIN 7)
*           BIT 3  ---  LVMA  (CPU PIN 19)
*           BIT 4  ---  FC0  (CPU PIN 28)
*           BIT 5  ---  FC1  (CPU PIN 27)
*           BIT 6  ---  FC2  (CPU PIN 26)
*           BIT 7  ---  LBGACK  (CPU PIN 12)
*
**********************************************************************
EA_TYPE         VARIABLE       0   ; TYPE OF EA REQUESTED
REG_FIELD       VARIABLE       0   ; VALUE OF THE REGISTER FIELD
MODE_FIELD      VARIABLE       0   ; VALUE OF THE MODE FIELD
SIZE_FIELD      VARIABLE       0   ; VALUE OF THE SIZE FIELD

DISASSEMBLY_BLOCK  VARIABLE    0   ; TRUE/FALSE - IS BLOCK ENCOUNTERED?
```

```
RD_STATUS       VARIABLE        0    ; HIGH/LOW BYTE READ FAILURES
HIGH_BYTE       VARIABLE        0    ; HIGH ORDER 32 BIT ADDRESS BYTE
LOW_BYTE        VARIABLE        0    ; LOW ORDER 32 BIT ADDRESS BYTE
INT             VARIABLE        0    ; TRUE = INTERRUPT ACK
SR_ACCESS       VARIABLE        0    ; TRUE = SR ACCESS OK, FALSE OTHERWISE

TEMP1           VARIABLE        0    ; GENERAL PURPOSE TEMPORARY

REG_MASK        VARIABLE        0    ; MOVEM REGISTER MASK

PREFETCH_LOC    VARIABLE        0    ; TELLS WHERE PREFETCH IS EXPECTED

REQUESTED_ADDRESS  VARIABLE     0    ; ADDRESS WE ARE LOOKING FOR

TEMP_DATA       VARIABLE        0    ; STORES UPPER BYTE OF 68008 INSTRUCTION
ABSOLUTE        VARIABLE        0    ; VARIABLE THAT TELLS IF THE ADDRESS IS
                                     ;  OFFSET OR ABSOLUTE DUE TO OVERFLOW
INT_VAL         VARIABLE     OFFFFFH ; VARIABLE FOR INT VS CPU SPACE OPEREATIONS

*************************************************************************

ML2             CONSTANT        03H            2 LSB MASK
ML3             CONSTANT        07H            3 LSB MASK
ML4             CONSTANT        0FH            4 LSB MASK
ML8             CONSTANT        0FFH           8 LSB MASK
ML20            CONSTANT        00FFFFFH       20 LSB MASK
ML24            CONSTANT        0FFFFFFH       24 LSB MASK

INSTR_LINE      CONSTANT        1              FLAG FOR INSTR LINE
NOT_INSTR_LINE  CONSTANT        0              FLAG FOR NON-INSTR LINE

UNUSED_PREFETCH CONSTANT        1              TAG FOR PREFETCHED INSTR
OPERAND_USED    CONSTANT        2              TAG FOR USED OPERANDS

FALSE           CONSTANT        0
TRUE            CONSTANT        1

BYTE            CONSTANT        00B            BYTE TRANSFER INDICATOR
WORD            CONSTANT        01B            WORD TRANSFER INDICATOR
LONG            CONSTANT        10B            LONG TRANSFER INDICATOR

TAG_COLUMN      CONSTANT        9              COLUMN FOR TAG DISPLAY
```

```
STATUS_MASK      CONSTANT        10111111B       MASK 68010
STATUS_VALUE     CONSTANT        10101001B       VALUE 68010

SUPR_DATA        ASCII           "supr data"
SUPR_PROG        ASCII           "supr program"
USER_DATA        ASCII           "user data"
USER_PROG        ASCII           "user program"
READ             ASCII           "read"
WRITE            ASCII           "write"
TYPE_6800        ASCII           " (6800)"
DMA              ASCII           "DMA"
FOUR_STAR        ASCII           "****"
COMMA            ASCII           ","
D                ASCII           "D"
A                ASCII           "A"
B                ASCII           "B"
W                ASCII           "W"
L                ASCII           "L"
NUM_SIGN         ASCII           "#"
OR_OP            ASCII           "OR"
AND_OP           ASCII           "AND"
SUB_OP           ASCII           "SUB"
EOR_OP           ASCII           "EOR"
ADD_OP           ASCII           "ADD"
CLR_OP           ASCII           "CLR"
MOVE_OP          ASCII           "MOVE"
SR               ASCII           "SR"
CCR              ASCII           "CCR"
USP              ASCII           "USP"
TWO_STAR         ASCII           "**"
S                ASCII           "S"
RIGHT_BRACKET    ASCII           "]"
R                ASCII           "R"
PERIOD           ASCII           "."
ILLEGAL          ASCII           "Illegal"

                 SEARCH_LIMIT    34

**************************************************************************

BIN3_FMT         FORMAT          3,BIN,3         3 BITS, BIN, 3 DIGITS
DEC1_FMT_3       FORMAT          3,DEC,1         3 BITS, DEC, 1 DIGIT
HEX1_FMT         FORMAT          4,HEX,1         4 BITS, HEX, 1 DIGIT
```

```
HEX1_FMT_3      FORMAT          3,HEX,1        3 BITS, HEX, 1 DIGIT
HEX2_FMT        FORMAT          8,HEX,2        8 BITS, HEX, 2 DIGITS
HEX2_FMT_5      FORMAT          5,HEX,2        5 BITS, HEX, 2 DIGITS
HEX4_FMT        FORMAT         16,HEX,4       16 BITS, HEX, 4 DIGITS
HEX6_FMT        FORMAT         24,HEX,6       24 BITS, HEX, 6 DIGITS
HEX8_FMT        FORMAT         32,HEX,8       32 BITS, HEX, 8 DIGITS

*************************************************************************

   LABEL_TITLE ^68010 Mnemonic^
   BASE_TITLE  ^      hex^
   DEFAULT_WIDTH 31

   LOAD ML20
   STORE INT_VAL

INITIALIZE
   SET RETURN_FLAGS,INSTR_LINE  PRESET TO INSTRUCTION LINE
   SET SR_ACCESS,FALSE          INITIAL CONDITION, DONT ALLOW SR ACCESS
   SET DISASSEMBLY_BLOCK,FALSE  INITIALLY NO INVERSE ASSEMBLY BLOCK

   LOAD INPUT_ADDRESS           GET ADDRESS OF FIRST STATE PASSED
   STORE REQUESTED_ADDRESS      SAVE THIS AS THE ADDRESS WE'RE LOOKING AT

   IF INPUT_ERROR <> 0 THEN GOTO DATA_ERROR

   LOAD  STATUS_MASK       ; GET PROGRAM READ STATUS MASK
   STORE QUALIFY_MASK            ... AND SAVE AS INPUT QUALIFIER MASK
   LOAD  STATUS_VALUE      ; GET PROGRAM READ STATUS VALUE
   STORE QUALIFY_VALUE          ... AND SAVE AS INPUT QUALIFIER VALUE

CHECK_INPUT_TAG
   LOAD INPUT_TAG                       GET THE CURRENT TAG VALUE
   IF 17,16 <> 0 THEN GOTO OPCODE_DECODE    OK TO DISASSEMBLE

*    ELSE ONLY DISPLAY STATUS INFORMATION

   LOAD INPUT_STATUS
   AND QUALIFY_MASK
   IF 7,0 = QUALIFY_VALUE THEN GOTO FLAGGED_AS_INSTR

*************************************************************************
```

```
DISPLAY_STATUS
        SET RETURN_FLAGS,NOT_INSTR_LINE          DONT FLAG THIS LINE
FLAGGED_AS_INSTR
        POSITION ABS,3
        LOAD INPUT_DATA
        OUTPUT ACCUMULATOR,HEX4_FMT              DISPLAY THE INPUT DATA

        POSITION REL,1
        LOAD INPUT_TAG
        IF 1,0 <> UNUSED_PREFETCH THEN GOTO NOT_PREFETCH
            OUTPUT "unused prefetch"
            RETURN
NOT_PREFETCH
        LOAD INPUT_STATUS
        IF 7,7 = 1 THEN GOTO NOT_DMA
            OUTPUT DMA
            RETURN
NOT_DMA
        CASE_OF 6,4
            CALL    UNKNOWN
            OUTPUT USER_DATA
            OUTPUT USER_PROG
            CALL    UNKNOWN
            CALL    UNKNOWN
            OUTPUT SUPR_DATA
            OUTPUT SUPR_PROG
            CALL    INT_ACK
        CASE_END
        IF INT = TRUE THEN GOTO NO_R_W_DISP      INTERRUPT ACK IS ALWAYS READ
        POSITION REL,1
        CASE_OF 0,0
            OUTPUT WRITE
            OUTPUT READ
        CASE_END
NO_R_W_DISP
        CASE_OF 3,3
            OUTPUT TYPE_6800      INDICATE A 6800 CYCLE
```

```
          NOP                  NOTHING FOR NORMAL 68000 CYCLE
      CASE_END
*  IF THE TRANSFER IS A BYTE TRANSFER THE UNDEFINED BYTE IS xxed OUT.
*  UPPER AND LOWER DATA STROBE TELL WHICH BYTE OR BYTES ARE VALID.
*  UPPER DATA STROBE IS INPUT ON STATUS BIT 2 AND LOWER DATA
*  STROBE IS INPUT ON STATUS BIT 1.

      CASE_OF 2,1
          RETURN               WORD XFER, NO DONT CARES
          POSITION ABS,5       POSITION TO xx THE LOWER BYTE
          POSITION ABS,3       POSITION TO xx THE UPPER BYTE
          RETURN               UNDEFINED TRANSFER
      CASE_END
      OUTPUT "xx"              OUTPUT THE DON'T CARES

      RETURN                  EXIT INVERSE ASSEMBLER


***********************************************************************


OPCODE_DECODE

   LOAD INPUT_TAG             GET THE TAG FOR THIS STATE
   IF 1,0 = OPERAND_USED THEN GOTO DISPLAY_STATUS
   IF 1,0 = UNUSED_PREFETCH THEN GOTO DISPLAY_STATUS

   LOAD INPUT_STATUS          GET THE STATUS OF THE INPUT
   AND  QUALIFY_MASK          GET ONLY THE DESIRED BITS FOR PROGRAM READ
   IF 7,0 <> QUALIFY_VALUE THEN GOTO DISPLAY_STATUS    BRIF NOT OPCODE

   LOAD INPUT_DATA            GET THE DATA THAT WAS READ
MEM_OPC_DEC
   LOAD INPUT_DATA            GET THE DATA THAT WAS READ
   CASE_OF 15,12
      GOTO Bit_MOVEP_Imm
      GOTO MOVE_B_L_W
      GOTO MOVE_B_L_W
      GOTO MOVE_B_L_W
      GOTO Misc
      GOTO ADDQ_SUBQ_Scc_DBcc
```

```
            GOTO Bcc
            GOTO MOVEQ
            GOTO OR_DIV_BCD_AND_MUL_EXG
            GOTO Math_Extended
            GOTO UNIMPLEMENTED
            GOTO CMP_EOR
            GOTO OR_DIV_BCD_AND_MUL_EXG
            GOTO Math_Extended
            GOTO Shift_Rotate
            GOTO UNIMPLEMENTED
        CASE_END

************************************************************************

Bit_MOVEP_Imm
    IF 8,8 = 1 THEN GOTO Dynamic_MOVEP
    CASE_OF 11,9
        OUTPUT OR_OP               0000 0000 XXXX XXXX
        OUTPUT AND_OP              0000 0010 XXXX XXXX
        OUTPUT SUB_OP              0000 0100 XXXX XXXX
        OUTPUT ADD_OP              0000 0110 XXXX XXXX
        GOTO Static                0000 1000 XXXX XXXX
        OUTPUT EOR_OP              0000 1010 XXXX XXXX
        OUTPUT "CMP"               0000 1100 XXXX XXXX
        GOTO MOVES                 0000 1110 XXXX XXXX
    CASE_END
    OUTPUT "I"                 ALL OF THE ABOVE ARE IMMEDIATE
    CALL SHOW_SIZE             DISPLAY THE SIZE INDICATOR
    CALL IMMEDIATE             DISPLAY THE IMMEDIATE DATA
    OUTPUT COMMA                  SEPARATE OPERAND FIELDS
* NOW SET THE VARIABLE THAT ALLOWS/DISALLOWS EFFECTIVE ADDRESS ACCESS
* TO THE STATUS REGISTER

    LOAD INITIAL_DATA          GET THE ORIGINAL OPCODE BACK
    CASE_OF 11,9
        SET SR_ACCESS,TRUE      ORI CAN ACCESS STATUS REGISTER
        SET SR_ACCESS,TRUE      ANDI CAN ACCESS STATUS REGISTER
        SET SR_ACCESS,FALSE     SUBI CANNOT ACCESS STATUS REGISTER
        SET SR_ACCESS,FALSE     ADDI CANNOT ACCESS STATUS REGISTER
        NOP                     CANT GET HERE (BIT STATIC INSTR)
        SET SR_ACCESS,TRUE      EORI CAN ACCESS STATUS REGISTER
```

```
       SET SR_ACCESS,FALSE        CMPI CANNOT ACCESS STATUS REGISTER
         NOP                      CANT GET HERE (ILLEGAL OPCODE)
    CASE_END
    CALL EA_DISP                  DISPLAY THE EFFECTIVE ADDRESS
    IF SR_ACCESS = FALSE THEN RETURN            CANNOT ACCESS SR/CCR
    LOAD INITIAL_DATA             GET THE ORIGINAL OPCODE
    IF 5,0 <> 111100B THEN RETURN    COULD, BUT DIDNT ACCESS SR/CCR
    RETURN

*************************************************************************

MOVES
    OUTPUT MOVE_OP
    OUTPUT S
    CALL SHOW_SIZE
    CALL READ_NEXT_OPERAND
    IF INPUT_ERROR <>0 THEN GOTO INCOMPLETE_OPCODE
    LOAD INPUT_DATA
    STORE TEMP_DATA
    CASE_OF 11,11
            CALL EA_DISP
            CALL REG_DISP
    CASE_END
    OUTPUT COMMA
    LOAD TEMP_DATA
    CASE_OF 11,11
            CALL REG_DISP
            CALL EA_DISP
    CASE_END
    RETURN
REG_DISP
    LOAD TEMP_DATA
    CASE_OF 15,15
            OUTPUT D
            OUTPUT A
    CASE_END
    AND 07000H
    ROTATE RIGHT,12
    OUTPUT ACCUMULATOR,DEC1_FMT_3
    RETURN

*************************************************************************
```

```
Dynamic_MOVEP
   IF 5,3 = 001B THEN GOTO MOVEP
   CALL BIT_MNEMONIC            DISPLAY THE BIT MANIPULATION INSTR
   POSITION ABS,TAG_COLUMN      MOVE OVER TO OPERAND FIELD
   CALL DATA_REG_11_9           DISPLAY THE REG NUMBER (BITS 11 - 9)
   OUTPUT COMMA
   GOTO EA_DISP                 DISPLAY THE EFFECTIVE ADDRESS
*************************************************************************

BIT_MNEMONIC

*  DISPLAY THE BIT MANIPULATION INSTRUCTION (DYNAMIC/STATIC)

   OUTPUT B
   CASE_OF 7,6
      OUTPUT "TST"
      OUTPUT "CHG"
      OUTPUT CLR_OP
      OUTPUT "SET"
   CASE_END
   OUTPUT PERIOD
   IF 5,3 = 000B THEN GOTO LONG_BIT            BRIF DYNAMIC BIT
      OUTPUT B
      SET SIZE_FIELD,BYTE      INDICATE BYTE SIZE TRANSFER
      RETURN
LONG_BIT
      OUTPUT L
      SET SIZE_FIELD,LONG      INDICATE LONG SIZE TRANSFER
      RETURN

*************************************************************************

MOVEP
   OUTPUT MOVE_OP
   OUTPUT "P."              DISPLAY THE MNEMONIC
   CASE_OF 6,6
      OUTPUT W                 SIZE = WORD
      OUTPUT L                 SIZE = LONG
   CASE_END
```

```
   CASE_OF 6,6
      SET SIZE_FIELD,WORD        SIZE = WORD
      SET SIZE_FIELD,LONG        SIZE = LONG
   CASE_END
   SET MODE_FIELD,101B           SET TO ADDR INDIRECT + OFFSET MODE
   POSITION ABS,TAG_COLUMN       MOVE OVER TO THE OPERAND FIELD
   CASE_OF 7,7
      CALL EA_DISP_MODE_SET      DISPLAY THE MEMORY SOURCE LOCATION
      CALL DATA_REG_11_9         DISPLAY THE DATA REG SOURCE
   CASE_END
   LOAD INITIAL_DATA             GET BACK THE ORIGINAL OPCODE
   OUTPUT COMMA
   CASE_OF 7,7
      CALL DATA_REG_11_9         DISPLAY THE DATA REG DESTINATION
      CALL EA_DISP_MODE_SET      DISPLAY THE MEMORY DESTINATION
   CASE_END
   RETURN

***************************************************************************

Static
   CALL BIT_MNEMONIC             DISPLAY THE BIT MANIPULATION INSTR
   POSITION ABS,TAG_COLUMN       MOVE OVER TO THE OPERAND FIELD
   OUTPUT NUM_SIGN                        INDICATE IMMEDIATE DATA
   CALL READ_NEXT_OPERAND        GET THE BIT NUMBER OPERAND
   IF INPUT_ERROR = 0 THEN GOTO Static_OK
      LOAD SIZE_FIELD            GET THE SIZE INDICATOR
      CASE_OF 1,1
         OUTPUT "*"             0X = BYTE ie MODULO 8 BIT NUMBER
         OUTPUT TWO_STAR        1X = LONG ie MODULO 32 BIT NUMBER
      CASE_END
      GOTO SNODEA                Static Not Ok Display Effective Address
Static_OK
   LOAD INPUT_DATA               GET THE BIT NUMBER SPECIFIER
   CASE_OF SIZE_FIELD
      OUTPUT  ACCUMULATOR,HEX1_FMT_3          DISPLAY THE 3 BIT VALUE
      NOP                        SHOULDN'T EVER GET HERE
      OUTPUT ACCUMULATOR,HEX2_FMT_5           DISPLAY THE 5 BIT VALUE
      NOP
   CASE_END
SNODEA
```

```
      OUTPUT COMMA
      GOTO EA_DISP                  DISPLAY THE EFFECTIVE ADDRESS
***********************************************************************

MOVE_B_L_W
   OUTPUT MOVE_OP
   IF 8,6 = 001B THEN OUTPUT A
   OUTPUT PERIOD

*  THE CONVENTIONAL SIZE ROUTINE (SHOW_SIZE) CANNOT BE USED BECAUSE
*  THE MOVE INSTRUCTION USES A DIFFERENT SIZE ENCODING

   CASE_OF 13,12                 SIZE FIELD
      NOP                        SHOULDNT EVER GET HERE
      OUTPUT B                   BYTE TRANSFER
      OUTPUT L                   LONG TRANSFER
      OUTPUT W                   WORD TRANSFER
   CASE_END
   CASE_OF 13,12                 SIZE FIELD
      NOP                        SHOULDNT EVER GET HERE
      SET SIZE_FIELD,BYTE        BYTE TRANSFER
      SET SIZE_FIELD,LONG        LONG TRANSFER
      SET SIZE_FIELD,WORD        WORD TRANSFER
   CASE_END

   POSITION ABS,TAG_COLUMN       POSITION AT THE TAG DISPLAY POSITION
   CALL SOURCE_EA                DISPLAY THE DESTINATION EFFECTIVE ADDR
   OUTPUT COMMA
   LOAD INITIAL_DATA             GET THE OPCODE BACK AGAIN
   GOTO DESTINATION_EA           DISPLAY THE SOURCE EFFECTIVE ADDRESS

***********************************************************************

Misc
   IF 8,7 = 10B THEN GOTO ILLEGAL_OPCODE
   IF 8,8 = 1 THEN GOTO CHK_LEA
   IF 11,11 = 0 THEN GOTO NEGX_CLR_NOT_MOVE
   CASE_OF 10,9
      GOTO NBCD_AND_MIX          0100 1000 XXXX XXXX
      GOTO TST_TAS               0100 1010 XXXX XXXX
      GOTO MOVEM_TO_REG          0100 1100 XXXX XXXX
```

```
      GOTO MISC_JSR_JMP        0100 1110 XXXX XXXX
   CASE_END
   RETURN                      THIS RETURN CANT HAPPEN
NBCD_AND_MIX
   CASE_OF 7,6
      OUTPUT "NBCD.B"          0100 1000 00XX XXXX
      GOTO PEA_SWAP            0100 1000 01XX XXXX
      GOTO MOVEM_EXT           0100 1000 10XX XXXX
      GOTO MOVEM_EXT           0100 1000 11XX XXXX
   CASE_END
   POSITION ABS,TAG_COLUMN     MOVE OVER TO TAG FIELD
   GOTO EA_DISP                DISPLAY THE EFFECTIVE ADDRESS
PEA_SWAP
   SET TEMP1,0                 PRESET TO SWAP INSTRUCTION
   IF 5,3 <> 000B THEN SET TEMP1,1      SET TO PEA INSTRUCTION
   CASE_OF TEMP1
      OUTPUT "SWAP.W"
      OUTPUT "PEA.L"
   CASE_END
   POSITION ABS,TAG_COLUMN
   CASE_OF TEMP1
      CALL DATA_REG_2_0        SWAP INSTRUCTION
      CALL EA_DISP             PEA INSTRUCTION
   CASE_END
   RETURN
CHK_LEA
   CASE_OF 6,6
      OUTPUT "CHK.W"           0100 XXX1 10XX XXXX
      OUTPUT "LEA.L"           0100 XXX1 11XX XXXX
   CASE_END
   POSITION ABS,TAG_COLUMN     MOVE OVER TO OPERAND FIELD
   CALL EA_DISP                DISPLAY THE EFFECTIVE ADDRESS
   LOAD INITIAL_DATA           RECOVER THE ORIGINAL OPCODE
   OUTPUT COMMA
   CASE_OF 6,6
      CALL DATA_REG_11_9       BOUND IS IN DATA REGISTER
      CALL ADDR_REG_11_9       EFFECTIVE ADDRESS GOES TO ADDR REG
   CASE_END
   RETURN
```

```
NEGX_CLR_NOT_MOVE
    IF 7,6 = 11B THEN GOTO MOVE_SR_CCR          0100 0XX0 11XX XXXX
    CASE_OF 10,9
        OUTPUT "NEGX"
        OUTPUT CLR_OP
        OUTPUT "NEG"
        OUTPUT "NOT"
    CASE_END
    CALL SHOW_SIZE              DISPLAY THE OPERATION SIZE
    CALL EA_DISP               DISPLAY THE EFFECTIVE ADDRESS
    RETURN
MOVE_SR_CCR
    OUTPUT MOVE_OP
    OUTPUT PERIOD
    CASE_OF 10,9
        OUTPUT W
        OUTPUT W
        OUTPUT B
        OUTPUT W
    CASE_END
    CASE_OF 10,9
        SET SIZE_FIELD,WORD
        SET SIZE_FIELD,WORD        CANT GET HERE FOR 68000/08
        SET SIZE_FIELD,BYTE
        SET SIZE_FIELD,WORD
    CASE_END
    POSITION ABS,TAG_COLUMN
    CASE_OF 10,9
        OUTPUT SR
        OUTPUT CCR                ILLEGAL INSTRUCTION, CANT GET HERE FOR 68000/08
        CALL EA_DISP              DISPLAY THE SOURCE ADDRESS
        CALL EA_DISP              DISPLAY THE SOURCE ADDRESS
    CASE_END
    OUTPUT COMMA
    LOAD INITIAL_DATA             GET THE ORIGINAL OPCODE BACK
    CASE_OF 10,9
        CALL EA_DISP              DISPLAY THE DESTINATION ADDRESS
        CALL EA_DISP              ILLEGAL INSTRUCTION, CANT GET HERE FOR 68000/08
        OUTPUT CCR
        OUTPUT SR
    CASE_END
```

```
      LOAD INITIAL_DATA           GET THE ORIGINAL OPCODE BACK
      IF 10,9 < 2 THEN RETURN     NO UNUSED PREFETCH TO MARK
*
*  NOW, LOOK AT THE EFFECTIVE ADDRESS MODE FIELD TO DETERMINE THE NUMBER
*  OF PROGRAM READ STATES REQUIRED FOR THIS INSTRUCTION. THIS VALUE PLUS
*  ONE WILL THEN POINT TO THE PROPER LOCATION FOR THE UNUSED PREFETCH
*  STATE.
*
      CASE_OF 5,3
         SET PREFETCH_LOC,+1      DATA REGISTER DIRECT
         NOP                      ADDRESS REGISTER DIRECT      (**ILLEGAL**)
         SET PREFETCH_LOC,+1      ADDRESS REGISTER INDIRECT
         SET PREFETCH_LOC,+1      ADDRESS REGISTER INDIRECT POST-INCREMENT
         SET PREFETCH_LOC,+1      ADDRESS REGISTER INDIRECT PRE-DECREMENT
         SET PREFETCH_LOC,+2      ADDRESS REGISTER INDIRECT W/DISPLACEMENT
         SET PREFETCH_LOC,+2      ADDR REG IND W/DISPLACEMENT AND INDEX
         CALL LOOK_AT_REG_FIELD   FOR MODE = 7, REG FIELD MUST BE EXAMINED
      CASE_END
      GOTO MARK_PREFETCH          MARK THE PREFETCH
LOOK_AT_REG_FIELD
      CASE_OF 2,0                 LOOKING NOW AT THE REG FIELD
         SET PREFETCH_LOC,+2      ABSOLUTE ADDRESS (SHORT)
         SET PREFETCH_LOC,+3      ABSOLUTE ADDRESS (LONG)
         SET PREFETCH_LOC,+2      PC RELATIVE W/DISPLACEMENT
         SET PREFETCH_LOC,+2      PC RELATIVE W/DISPLACEMENT AND INDEX
         SET PREFETCH_LOC,+2      IMMEDIATE DATA
      CASE_END
      RETURN

MOVEM_EXT
      IF 5,3 = 000B THEN GOTO EXT
MOVEM
      OUTPUT MOVE_OP
      OUTPUT "M."
      CASE_OF 6,6
         OUTPUT W
         OUTPUT L
      CASE_END
      CALL READ_NEXT_OPERAND      READ THE MASK WORD
      STORE REG_MASK              SAVE THE MASK WORD
      IF INPUT_ERROR <> 0 THEN SET REG_MASK,0
```

```
      POSITION ABS,TAG_COLUMN
      LOAD INITIAL_DATA            GET THE ORIGINAL OPCODE BACK
      CASE_OF 10,10
         CALL MOVEM_MASK
         CALL EA_DISP
      CASE_END
      OUTPUT COMMA
      LOAD INITIAL_DATA            GET BACK THE ORIGINAL OPCODE
      CASE_OF 10,10
         CALL EA_DISP
         CALL MOVEM_MASK
      CASE_END
      RETURN
MOVEM_MASK
      OUTPUT "rm="
      LOAD REG_MASK
      IF 15,0 <> 0 THEN GOTO MASK_FOUND
         OUTPUT FOUR_STAR          IF NO MASK WAS FOUND
         RETURN
MASK_FOUND
      OUTPUT ACCUMULATOR,HEX4_FMT
      RETURN


EXT
      OUTPUT "EXT."
      CASE_OF 6,6
         OUTPUT W
         OUTPUT L
      CASE_END
      POSITION ABS,TAG_COLUMN
      GOTO DATA_REG_2_0

TST_TAS
      IF 7,6 <> 11B THEN GOTO TST
         OUTPUT "TAS.B"
         GOTO MNEMONIC_DISPLAYED
TST
      OUTPUT "TST"
      CALL SHOW_SIZE
```

```
MNEMONIC_DISPLAYED
   POSITION ABS,TAG_COLUMN       MOVE OVER TO THE OPERAND FIELD
   GOTO EA_DISP                  DISPLAY THE EFFECTIVE ADDRESS
MOVEM_TO_REG
   IF 7,7 = 0 THEN GOTO ILLEGAL_OPCODE
   GOTO MOVEM
MISC_JSR_JMP
   CASE_OF 8,6
      GOTO ILLEGAL_OPCODE
      GOTO TRAP_MIX
      OUTPUT "JSR"
      OUTPUT "JMP"
   CASE_END
   POSITION ABS,TAG_COLUMN
   GOTO EA_DISP
*  CALL MARK_PREFETCH           DUMP THE UNUSED PREFETCH
TRAP_MIX
   CASE_OF 5,4
      GOTO TRAP
      GOTO LINK_UNLK
      GOTO MOVE_USP
      NOP
   CASE_END
   IF 5,3 = 111B THEN GOTO MOVEC
   CASE_OF 2,0
      OUTPUT "RESET"
      OUTPUT "NOP"
      OUTPUT "STOP"
      OUTPUT "RTE"
      GOTO  RTD
      OUTPUT "RTS"
      OUTPUT "TRAPV"
      OUTPUT "RTR"
   CASE_END
   IF 2,0 <> 2 THEN GOTO NO_OPERAND
      POSITION ABS,TAG_COLUMN
      OUTPUT NUM_SIGN
      CALL READ_NEXT_OPERAND
      IF INPUT_ERROR <>0 THEN GOTO WORD_ERROR
```

```
STOP_OK
    OUTPUT ACCUMULATOR,HEX4_FMT
    RETURN
RTD
    OUTPUT "RTD"
    POSITION ABS,TAG_COLUMN
    OUTPUT NUM_SIGN
    CALL READ_NEXT_OPERAND
    IF INPUT_ERROR <>0 THEN GOTO WORD_ERROR
    LOAD INPUT_DATA
    OUTPUT ACCUMULATOR,HEX4_FMT
    RETURN
MOVEC
    OUTPUT MOVE_OP
    OUTPUT "C.L"
    POSITION ABS,TAG_COLUMN
    CALL READ_NEXT_OPERAND
    IF INPUT_ERROR <>0 THEN GOTO INCOMPLETE_OPCODE
    LOAD INPUT_DATA
    STORE TEMP_DATA
    LOAD INITIAL_DATA
    CASE_OF 0,0
            CALL DISP_CONTROL_REG
            CALL REG_DISP
    CASE_END
    OUTPUT COMMA
    LOAD INITIAL_DATA
    CASE_OF 0,0
            CALL REG_DISP
            CALL DISP_CONTROL_REG
    CASE_END
    RETURN
DISP_CONTROL_REG
    LOAD TEMP_DATA
    IF 11,0=0 THEN OUTPUT "SFC"
    IF 11,0=1 THEN OUTPUT "DFC"
    IF 11,0=800H THEN OUTPUT USP
    IF 11,0=801H THEN OUTPUT "VBR"
    IF 10,1<> 0 THEN GOTO ILLEGAL_OPERAND
    RETURN
```

```
NO_OPERAND
   IF 2,0 < 3 THEN RETURN        NO PREFETCH TO BE MARKED
   IF 2,0 = 6 THEN RETURN        CANT MARK PREFETCH FOR TRAPV
   SET PREFETCH_LOC,+1
   GOTO MARK_PREFETCH

TRAP
   OUTPUT "TRAP"
   POSITION ABS,TAG_COLUMN
   OUTPUT NUM_SIGN
   AND ML4                       VECTOR NUMBER IN LOWER 4 BITS ONLY
   OUTPUT ACCUMULATOR,HEX1_FMT
   SET PREFETCH_LOC,+1
   GOTO MARK_PREFETCH            DUMP UNUSED PREFETCH

LINK_UNLK
   CASE_OF 3,3
      OUTPUT "LINK"
      OUTPUT "UNLK"
   CASE_END
   POSITION ABS,TAG_COLUMN
   CALL ADDR_REG_2_0
   IF 3,3 = 1 THEN RETURN
   OUTPUT ",#"
   CALL READ_NEXT_OPERAND
   IF INPUT_ERROR <> 0 THEN GOTO WORD_ERROR
LINK_UNLK_OK
   OUTPUT ACCUMULATOR,HEX4_FMT  DISPLAY THE 16 BIT DISPLACEMENT
   RETURN

MOVE_USP
   OUTPUT MOVE_OP
   OUTPUT ".L"
   POSITION ABS,TAG_COLUMN
   CASE_OF 3,3
      CALL ADDR_REG_2_0
      OUTPUT USP
   CASE_END
   OUTPUT COMMA
```

```
     CASE_OF 3,3
        OUTPUT USP
        CALL ADDR_REG_2_0
     CASE_END
     RETURN


*************************************************************************


ADDQ_SUBQ_Scc_DBcc
   IF 7,6 <> 11B THEN GOTO ADDQ_SUBQ          ADDQ & SUBQ INSTRUCTIONS
   IF 5,3 = 001B THEN GOTO DBcc
      OUTPUT S
      GOTO Scc_DBcc

DBcc

      OUTPUT "DB"

Scc_DBcc
   CALL CONDITION_CODE          DISPLAY THE CONDITION CODE
   IF 5,3 = 001B THEN GOTO DBcc_2
      OUTPUT ".B"
      POSITION ABS,TAG_COLUMN      MOVE OVER TO THE TAG LOCATION
      GOTO EA_DISP

DBcc_2
   POSITION ABS,TAG_COLUMN      MOVE OVER TO THE TAG LOCATION
   OUTPUT D                     A DATA REGISTER IS BEING USED
   OUTPUT ACCUMULATOR,DEC1_FMT_3  DISPLAY THE REGISTER NUMBER
   OUTPUT COMMA
   CALL READ_NEXT_OPERAND
   IF INPUT_ERROR <> 0 THEN GOTO WORD_ERROR
DBcc_DISP_OK
   IF 15,15 = 1 THEN INCLUSIVE_OR 0FF0000H     SIGN EXTENSION
   ADD INPUT_ADDRESS            ADD PC OF DISPLACEMENT WORD TO DISPLACEMENT
ADDRESS_OUTPUT
   AND ML24                     ONLY INTERESTED IN LOWER 24 BITS
   IF_NOT_MAPPED THEN OUTPUT ACCUMULATOR,HEX6_FMT
   RETURN


*************************************************************************


ADDQ_SUBQ
   CASE_OF 8,8
```

```
        OUTPUT "ADDQ"
        OUTPUT "SUBQ"
      CASE_END
      CALL SHOW_SIZE              DISPLAY THE SIZE OF THE OPERATION
      CALL IMM_DATA_11_9          DISPLAY THE 3 BIT IMMEDIATE DATA
      OUTPUT COMMA
      GOTO EA_DISP                DISPLAY THE EFFECTIVE ADDRESS
**************************************************************************


Bcc
   OUTPUT B
   CALL CONDITION_CODE           DISPLAY THE APPLICABLE CONDITION

   IF 7,0 <> 0 THEN GOTO EIGHT_BIT    BRIF AN 8 BIT DISPLACEMENT VALUE
      OUTPUT ".W"                DISPLACEMENT WAS 16 BIT WORD
      SET SIZE_FIELD,WORD        OPERATION SIZE = WORD
      CALL READ_NEXT_OPERAND           GET THE 16 BIT DISPLACEMENT VALUE
      IF 15,15 = 1 THEN INCLUSIVE_OR 0FF0000H      SIGN EXTENSION
      IF INPUT_ERROR = 0 THEN GOTO DISP_OK
         OUTPUT "********"
         RETURN
EIGHT_BIT
   OUTPUT ".B"                   DISPLACEMENT WAS 8 BIT BYTE
   SET SIZE_FIELD,BYTE           OPERATION SIZE = BYTE
   AND ML8                       INTERESTED IN LOWER 8 BITS ONLY
   IF 7,7 = 1 THEN INCLUSIVE_OR 0FFFF00H     EXTEND THE SIGN
DISP_OK
   POSITION ABS,TAG_COLUMN       MOVE OVER TO THE OPERAND FIELD
   ADD INPUT_ADDRESS             ADD ADDRESS OF DISPLACEMENT WORD
   AND ML24                      ONLY INTERESTED IN LOWER 24 BITS
   IF SIZE_FIELD = BYTE THEN ADD 2     PC POINTING AHEAD 2 LOCATIONS
   CALL ADDRESS_OUTPUT
   LOAD INITIAL_DATA             RELOAD THE INITIAL OPCODE
   IF 15,9 <> 0110000B THEN RETURN     BRIF NOT BRA OR BSR
   IF 7,0 = 0 THEN RETURN        NO PREFETCH IF BSR.W OR BRA.W
   SET PREFETCH_LOC,+1
   GOTO MARK_PREFETCH


**************************************************************************
```

```
MOVEQ
   OUTPUT MOVE_OP
   OUTPUT "Q.L"
   POSITION ABS,TAG_COLUMN      MOVE OVER TO TAG COLUMN
   AND ML8                      GET THE 8 BIT DATA FIELD
   IF 7,7 = 1 THEN INCLUSIVE_OR 0FFFFFF00H      SIGN EXTEND TO 32 BITS
   OUTPUT NUM_SIGN                     IMMEDIATE DATA INDICATOR
   OUTPUT ACCUMULATOR,HEX8_FMT  DISPLAY THE 32 BIT IMMEDIATE VALUE
   OUTPUT COMMA
   GOTO DATA_REG_11_9           DISPLAY THE DATA REGISTER IN 11 THRU 9


***********************************************************************


Math_Extended
   CASE_OF 14,14
      OUTPUT SUB_OP
      OUTPUT ADD_OP
   CASE_END
   IF 7,6 = 11B THEN GOTO SUBA_ADDA
   IF 8,8 <> 1 THEN GOTO NOT_EXTENDED
      IF 5,4 = 00B THEN GOTO EXTENDED_MATH
NOT_EXTENDED
   CALL SHOW_SIZE               DISPLAY THE SIZE OF THE OPERATION
   POSITION ABS,TAG_COLUMN      MOVE OVER TO THE OPERAND FIELD
   CASE_OF 8,8
      CALL EA_DISP              DISPLAY THE SOURCE EFFECTIVE ADDRESS
      CALL DATA_REG_11_9        DISPLAY THE SOURCE DATA REGISTER
   CASE_END
   OUTPUT COMMA
   LOAD INITIAL_DATA            GET BACK THE ORIGINAL OPCODE
   CASE_OF 8,8
      CALL DATA_REG_11_9        DISPLAY THE DESTINATION DATA REGISTER
      CALL EA_DISP              DISPLAY THE DESTINATION EFFECTIVE ADDR
   CASE_END
   RETURN
SUBA_ADDA
   OUTPUT "A."
A_ADDRESSING
   CASE_OF 8,8
      OUTPUT W
```

```
              OUTPUT L
          CASE_END
          CASE_OF 8,8
              SET SIZE_FIELD,WORD       WORD OPERAND
              SET SIZE_FIELD,LONG       LONG OPERAND
          CASE_END
          POSITION ABS,TAG_COLUMN       MOVE OVER TO THE OPERAND FIELD
          CALL EA_DISP                  DISPLAY THE SOURCE ADDRESS
          OUTPUT COMMA
          LOAD INITIAL_DATA             GET THE ORIGINAL OPCODE
          GOTO ADDR_REG_11_9            DISPLAY THE ADDRESS DESTINATION ADDR


      ***********************************************************************


      CMP_EOR
          IF 7,6 = 11B THEN GOTO CMPA
          IF 8,8 = 1 THEN GOTO CMPM_EOR
              OUTPUT "CMP"
              CALL SHOW_SIZE            DISPLAY THE OPERATION SIZE
              CALL EA_DISP             DISPLAY THE EA SOURCE LOCATION
              OUTPUT COMMA
              GOTO DATA_REG_11_9       DISPLAY THE DATA REGISTER NAME
      CMPA
          OUTPUT "CMPA."
          GOTO A_ADDRESSING

      CMPM_EOR
          IF 5,3 <> 001B THEN GOTO EOR
              OUTPUT "CMPM"
              CALL SHOW_SIZE            DISPLAY THE OPERATION SIZE
              POSITION ABS,TAG_COLUMN   MOVE OVER TO THE OPERAND DISPLAY FIELD
              OUTPUT "[xx]+,[xx]+"      DISPLAY OPERAND (xx WILL BE BACKFILLED)
              POSITION REL,-10          MOVE BACK OVER TO SRC REG # FIELD
              CALL ADDR_REG_2_0         DISPLAY THE 3 BIT REGISTER NUMBER
              POSITION REL,+4           MOVE OVER TO THE DEST REG FIELD
              CALL ADDR_REG_11_9        DISPLAY THE 3 BIT ADDRESS REGISTER NAME
              POSITION REL,+2           MOVE TO THE END OF THE OPERAND
              RETURN

      EOR
          OUTPUT EOR_OP
          CALL SHOW_SIZE               DISPLAY THE OPERATION SIZE
```

```
      CALL DATA_REG_11_9          DISPLAY THE SOURCE DATA REGISTER
      OUTPUT COMMA
      GOTO EA_DISP                DISPLAY THE DESTINATION EFFECTIVE ADDR


************************************************************************


OR_DIV_BCD_AND_MUL_EXG

   IF 7,6 = 11B THEN GOTO DIV_MUL_U_S
   IF 14,14 = 0 THEN GOTO NOT_EXG
      IF 8,4 = 10100B THEN GOTO EXG
      IF 8,4 = 11000B THEN GOTO EXG
NOT_EXG
   IF 8,4 = 10000B THEN GOTO SBCD_ABCD
   CASE_OF 14,14
      OUTPUT OR_OP
      OUTPUT AND_OP
   CASE_END
   GOTO NOT_EXTENDED
DIV_MUL_U_S
   CASE_OF 14,14
      OUTPUT "DIV"
      OUTPUT "MUL"
   CASE_END
   CASE_OF 8,8
      OUTPUT "U"               UNSIGNED DIVIDE
      OUTPUT S                SIGNED DIVIDE
   CASE_END
   OUTPUT ".W"
   POSITION ABS,TAG_COLUMN     MOVE OVER TO TAG FIELD
   CALL EA_DISP                DISPLAY THE EFFECTIVE ADDRESS
   OUTPUT COMMA
   GOTO DATA_REG_11_9          DISPLAY THE DATA REGISTER IN BITS 11-9
SBCD_ABCD
   CASE_OF 14,14
      OUTPUT S
      OUTPUT A
   CASE_END
   OUTPUT "BCD.B"
   POSITION ABS,TAG_COLUMN     MOVE OVER TO THE OPERAND FIELD
```

```
        IF 3,3 = 0 THEN GOTO REG_TO_REG
            OUTPUT "-["
            CALL ADDR_REG_2_0        DISPLAY THE SOURCE ADDR REGISTER
            OUTPUT "],-["
            CALL ADDR_REG_11_9       DISPLAY THE DESTINATION ADDR REGISTER
            OUTPUT RIGHT_BRACKET
            RETURN
REG_TO_REG
        CALL DATA_REG_2_0            DISPLAY THE SOURCE DATA REGISTER
        OUTPUT COMMA
        GOTO DATA_REG_11_9           DISPLAY THE DESTINATION REGISTER
EXG
        OUTPUT "EXG.L"
        POSITION ABS,TAG_COLUMN
        IF 7,3 = 01000B THEN SET TEMP1,0            DATA REGISTER EXG
        IF 7,3 = 01001B THEN SET TEMP1,1            ADDR REGISTER EXG
        IF 7,3 = 10001B THEN SET TEMP1,2            ADDR/DATA REGISTER EXG
        CASE_OF TEMP1
            CALL DATA_REG_11_9       DISPLAY THE FIRST DATA REGISTER
            CALL ADDR_REG_11_9       DISPLAY THE FIRST ADDR REGISTER
            CALL DATA_REG_11_9       DISPLAY THE DATA REGISTER
            NOP                      CANT GET HERE
        CASE_END
        OUTPUT COMMA
        LOAD INITIAL_DATA            GET THE ORIGINAL OPCODE BACK
        CASE_OF TEMP1
            CALL DATA_REG_2_0        DISPLAY THE SECOND DATA REGISTER
            CALL ADDR_REG_2_0        DISPLAY THE SECOND ADDR REGISTER
            CALL ADDR_REG_2_0        DISPLAY THE ADDRESS REGISTER
            NOP                      CANT GET HERE
        CASE_END
        RETURN


*************************************************************************

Shift_Rotate
        IF 7,6 = 11B THEN GOTO Memory_Shift
        CASE_OF 4,3                  DISPLAY THE TYPE OF SHIFT
            OUTPUT "AS"
            OUTPUT "LS"
```

```
        OUTPUT "ROX"
        OUTPUT "RO"
    CASE_END
    CASE_OF 8,8                  DISPLAY THE DIRECTION OF THE SHIFT
        OUTPUT R
        OUTPUT L
    CASE_END
    CALL SHOW_SIZE               DISPLAY THE OPERATION SIZE (B,L,W)

    IF 5,5 = 1 THEN GOTO REG_COUNT
        CALL IMM_DATA_11_9       DISPLAY THE IMMEDIATE DATA
        GOTO COUNT_DISPLAYED
REG_COUNT
    CALL DATA_REG_11_9           DISPLAY THE DATA REG IN BITS 11-9
COUNT_DISPLAYED
    OUTPUT COMMA
    GOTO DATA_REG_2_0            DISPLAY THE SELECTED DATA REGISTER


**********************************************************************


Memory_Shift
    CASE_OF 10,9                 DISPLAY THE TYPE OF SHIFT
        OUTPUT "AS"
        OUTPUT "LS"
        OUTPUT "ROX"
        OUTPUT "RO"
    CASE_END
    CASE_OF 8,8                  DISPLAY THE DIRECTION OF THE SHIFT
        OUTPUT R
        OUTPUT L
    CASE_END
    OUTPUT ".W"                  WORD OPERATIONS ONLY
    SET SIZE_FIELD,WORD          SET OPERATION SIZE TO WORD
    POSITION ABS,TAG_COLUMN      MOVE OVER TO THE TAG DISPLAY FIELD
    GOTO EA_DISP                 DISPLAY THE EFFECTIVE ADDRESS


**********************************************************************


UNIMPLEMENTED
    OUTPUT "Unimplemented Instruction:"
```

```
        OUTPUT ACCUMULATOR,HEX4_FMT
        SET PREFETCH_LOC,+1          RELATIVE POSITION OF EXPECTED PREFETCH
        GOTO MARK_PREFETCH           DUMP THE UNUSED PREFETCH


*************************************************************************
*  68000 INVERSE ASSEMBLER UTILITY ROUTINES
*************************************************************************


EA_DISP

*  DISPLAY AN EFFECTIVE ADDRESS
SOURCE_EA
        SET EA_TYPE,1               INDICATE A "NORMAL" LOCATION EA
        GOTO EA_TYPE_SET
DESTINATION_EA
        SET EA_TYPE,0              INDICATE A "SPECIAL" LOCATION EA

EA_TYPE_SET
        LOAD INITIAL_DATA         GET THE ORIGINAL OPCODE
        CASE_OF EA_TYPE
           ROTATE RIGHT,6         MOVE MODE FIELD TO LSB (SPECIAL)
           ROTATE RIGHT,3         MOVE MODE FIELD TO LSB (NORMAL)
        CASE_END
        AND ML3                   GET THE  LOWER THREE BITS
        STORE MODE_FIELD          SAVE THE MODE FIELD

        LOAD INITIAL_DATA         GET THE ORIGINALLY INPUT OPCODE
        CASE_OF EA_TYPE
           ROTATE RIGHT,9         MOVE REGISTER FIELD TO LSB (SPECIAL)
           NOP                    NO MOVE REQUIRED (NORMAL)
        CASE_END

EA_DISP_MODE_SET

*  ENTER HERE IF THE MODE HAS BEEN PRESET.  THIS ASSUMES THE REGISTER
*  NUMBER TO BE IN THE LOWER 3 BITS.
```

```
   AND ML3                    GET THE LOWER 3 BITS
   STORE REG_FIELD            SAVE THE REGISTER FIELD
 CASE_OF MODE_FIELD
    OUTPUT D                 DATA REGISTER DIRECT
    OUTPUT A                 ADDR REGISTER DIRECT
    OUTPUT "[A"               ADDR REGISTER INDIRECT
    OUTPUT "[A"               ADDR REGISTER INDIRECT W/POSTINCREMENT
    OUTPUT "-[A"              ADDR REGISTER INDIRECT W/PREDECREMENT
    GOTO DISP16              ADDR REGISTER INDIRECT W/DISPLACEMENT
    GOTO DISP8_INDEX         ADDR REG INDIRECT W/DISPLACEMENT&INDEX
    GOTO SPECIAL             SPECIAL ADDRESSING MODES
 CASE_END

 LOAD REG_FIELD             GET THE REGISTER FIELD
 OUTPUT ACCUMULATOR,DEC1_FMT_3  DISPLAY THE SELECTED REGISTER NUMBER

 LOAD INITIAL_DATA          GET THE ORIGINAL OPCODE BACK
 CASE_OF MODE_FIELD
    RETURN                   DATA REGISTER DIRECT
    RETURN                   ADDRESS REGISTER DIRECT
    OUTPUT RIGHT_BRACKET              ADDRESS REGISTER INDIRECT
    OUTPUT "]+"              ADDR REGISTER INDIRECT W/POSTINCREMENT
    OUTPUT RIGHT_BRACKET              ADDR REGISTER INDIRECT W/PREDECREMENT
    NOP                     SHOULDNT EVER GET HERE
    NOP                     SHOULDNT EVER GET HERE
    NOP                     SHOULDNT EVER GET HERE
 CASE_END
 RETURN


***************************************************************************


DISP16

*   16 BIT DISPLACEMENT MODE

   CALL DISP_NEXT_WORD        DISPLAY THE 16 BIT OFFSET VALUE
   OUTPUT "[A"
   LOAD REG_FIELD             GET THE REGISTER INDICATED
   OUTPUT ACCUMULATOR,DEC1_FMT_3  DISPLAY THE SELECTED REGISTER
   OUTPUT RIGHT_BRACKET
```

```
        LOAD INITIAL_DATA          GET THE ORIGINAL OPCODE BACK
        RETURN


***********************************************************************


DISP8_INDEX

*  8 BIT DISPLACEMENT AND INDEX MODE
   CALL READ_NEXT_OPERAND        GET THE EXTENSION WORD

   IF INPUT_ERROR = 0 THEN GOTO DISP8_OK
      OUTPUT "**[A"              ADDRESS REG WILL BE BACKFILLED
      LOAD REG_FIELD            GET THE ADDRESS REGISTER NUMBER
      OUTPUT ACCUMULATOR,DEC1_FMT_3              DISPLAY REG #
      OUTPUT ",**.*]"
      RETURN
DISP8_OK
   AND ML8                        GET THE 8 BIT DISPLACEMENT
   OUTPUT ACCUMULATOR,HEX2_FMT  DISPLAY THE 8 BIT DISPLACEMENT
   OUTPUT "["
   LOAD INPUT_DATA              GET THE ORIGINAL INPUT DATA BACK
   OUTPUT A
   LOAD REG_FIELD              GET THE ADDRESS REGISTER NUMBER
   OUTPUT ACCUMULATOR,DEC1_FMT_3  DISPLAY THE ADDRESS REGISTER NUMBER
   OUTPUT COMMA
   CASE_OF 15,15
      OUTPUT D               DATA REGISTER IS INDEX
      OUTPUT A               ADDRESS REGISTER IS INDEX
   CASE_END
   LOAD INPUT_DATA              GET THE EXTENSION WORD
   ROTATE RIGHT,12             MOVE REGISTER NUMBER TO LSB
   OUTPUT ACCUMULATOR,DEC1_FMT_3  DISPLAY THE REGISTER NUMBER
   OUTPUT PERIOD
   LOAD INPUT_DATA              GET THE EXTENSION WORD
   CASE_OF 11,11
      OUTPUT W
      OUTPUT L
   CASE_END
   OUTPUT RIGHT_BRACKET
```

```
        LOAD INITIAL_DATA              RESTORE THE ORIGINAL OPCODE
        RETURN


    ***********************************************************************


    SPECIAL
        CASE_OF REG_FIELD
            GOTO ABS_SHORT              ABSOLUTE SHORT ADDRESS
            GOTO ABS_LONG               ABSOLUTE LONG ADDRESS
            GOTO PC_DISPLACEMENT        PC WITH DISPLACEMENT
            GOTO PC_INDEX               PC WITH DISPLACEMENT & INDEX
            GOTO IMMEDIATE_SR           IMMEDIATE/ SR CCR MODIFY
        CASE_END
        GOTO ILLEGAL_OPERAND           ILLEGAL ADDRESSING MODES


    ***********************************************************************


    ABS_SHORT
        CALL READ_NEXT_OPERAND         GET THE EXTENSION WORD
        IF INPUT_ERROR = 0 THEN GOTO ABS_SHORT_OK
            OUTPUT "******"
            RETURN
    ABS_SHORT_OK
        IF 15,15 = 1 THEN INCLUSIVE_OR 0FF0000H
        GOTO ADDRESS_OUTPUT


    ***********************************************************************


    ABS_LONG
        CALL READ_NEXT_OPERAND         GET THE HIGH ORDER BYTE
        STORE HIGH_BYTE                ... AND SAVE IT
        SET RD_STATUS,00B              INITIAL STATUS...NO ERRORS
        IF INPUT_ERROR <> 0 THEN SET RD_STATUS,01B   HIGH BYTE FAILURE
        CALL READ_NEXT_OPERAND         GET THE LOW ORDER BYTE
        STORE LOW_BYTE                 ... AND SAVE IT
        LOAD RD_STATUS                 GET THE CURRENT READ STATUS
        IF INPUT_ERROR <> 0 THEN INCLUSIVE_OR 10B    LOW BYTE FAILURE
        STORE RD_STATUS                SAVE THE READ STATUS

        IF RD_STATUS = 00B THEN GOTO MAP_32         BRIF MAPPING IS OK
            LOAD HIGH_BYTE             GET THE UPPER 16 BIT BYTE
            CASE_OF RD_STATUS
```

```
      NOP                    CANT GET HERE
      OUTPUT FOUR_STAR                      01 = HIGH BYTE FAILURE
      OUTPUT ACCUMULATOR,HEX4_FMT           10 = HIGH BYTE OK
      OUTPUT FOUR_STAR                      11 = BOTH BYTE FAILURE
    CASE_END
    LOAD LOW_BYTE           GET THE LOWER 16 BIT BYTE
    CASE_OF RD_STATUS
      NOP                    CANT GET HERE
      OUTPUT ACCUMULATOR,HEX4_FMT           01 = LOW BYTE OK
      OUTPUT FOUR_STAR                      10 = LOW BYTE FAILURE
      OUTPUT FOUR_STAR                      11 = BOTH BYTE FAILURE
    CASE_END
    RETURN
MAP_32
  LOAD HIGH_BYTE           GET THE UPPER 16 BITS
  AND 0FFH                 MASK TO LOWER 2 HIGH BYTE DIGITS
  ROTATE LEFT,16           MOVE TO THE UPPER 8 BITS
  INCLUSIVE_OR LOW_BYTE    OR IN THE LOWER 16 BITS
  GOTO ADDRESS_OUTPUT


*************************************************************************


PC_DISPLACEMENT
  CALL READ_NEXT_OPERAND        GET THE 16 BIT DISPLACEMENT
  IF INPUT_ERROR <> 0 THEN GOTO WORD_ERROR
PC_DISP_OK
  IF 15,15 = 1 THEN INCLUSIVE_OR 0FF0000H    SIGN EXTEND DISPLACEMENT
  ADD INPUT_ADDRESS             ADD DISPLACEMENT TO THE CURRENT PC
  CALL ADDRESS_OUTPUT
  OUTPUT "[PC]"
  RETURN


*************************************************************************


PC_INDEX
  CALL READ_NEXT_OPERAND      GET THE EXTENSION WORD
  IF INPUT_ERROR = 0 THEN GOTO PC_INDEX_OK
    OUTPUT "**[PC,**.*]"
    RETURN
PC_INDEX_OK
  AND 0FFH                                   IGNORE UPPER BYTE
```

```
   IF 7,7 = 1 THEN INCLUSIVE_OR 0FFFF00H     SIGN EXTEND DISPLACEMENT
   ADD INPUT_ADDRESS              ADD DISPLACEMENT TO THE CURRENT PC
   CALL ADDRESS_OUTPUT
   LOAD INPUT_DATA
   OUTPUT "[PC,"
   CASE_OF 15,15
      OUTPUT D
      OUTPUT A
   CASE_END
   ROTATE RIGHT,12               MOVE INDEX REGISTER TO LSB
   OUTPUT ACCUMULATOR,DEC1_FMT_3                DISPLAY REG NUMBER
   OUTPUT PERIOD
   LOAD INPUT_DATA               GET THE EXTENSION WORD BACK
   CASE_OF 11,11                 DISPLAY THE SIZE OF THE INDEX
      OUTPUT W
      OUTPUT L
   CASE_END
   OUTPUT RIGHT_BRACKET
   RETURN


***************************************************************************


IMMEDIATE_SR
   IF SR_ACCESS = FALSE THEN GOTO IMMEDIATE
      CASE_OF SIZE_FIELD
         OUTPUT CCR          BYTE TRANSFER TO CCR
         OUTPUT SR           WORD TRANSFER EFFECTES ENTIRE SR
         GOTO ILLEGAL_OPERAND
         GOTO ILLEGAL_OPERAND
      CASE_END
      RETURN
IMMEDIATE
   OUTPUT NUM_SIGN
   CALL READ_NEXT_OPERAND            GET THE NEXT OPERAND WORD
   IF INPUT_ERROR = 0 THEN GOTO EXT1_OK
      CASE_OF SIZE_FIELD
         OUTPUT TWO_STAR        NO BYTE FOUND
         OUTPUT FOUR_STAR       NO WORD FOUND
         OUTPUT FOUR_STAR       HIGH ORDER WORD NOT FOUND
         GOTO ILLEGAL_OPCODE    ILLEGAL SIZE SPECIFICATION
      CASE_END
      GOTO CHECK_FOR_LONG
```

```
EXT1_OK
   CASE_OF SIZE_FIELD
      OUTPUT ACCUMULATOR,HEX2_FMT              DISPLAY 8 BIT DATA
      OUTPUT ACCUMULATOR,HEX4_FMT              DISPLAY 16 BIT DATA
      OUTPUT ACCUMULATOR,HEX4_FMT              DISP HIGH ORDER 16 BITS
      GOTO ILLEGAL_OPCODE                      SHOULDNT EVER GET HERE
   CASE_END
CHECK_FOR_LONG
*
*  NOW, SEE IF WE ARE DEALING WITH LONG IMMEDIATE VALUES.  IF NOT,
*  THERE IS NO NEED TO READ A SECOND EXTENSION WORD.
*
   IF SIZE_FIELD <> 10B THEN RETURN            BRIF NOT LONG DATA
*
*  AT THIS POINT, WE ARE LOOKING AT "LONG" IMMEDIATE VALUES ONLY
*
   CALL READ_NEXT_OPERAND          GET THE 2ND EXTENSION WORD
   IF INPUT_ERROR <> 0 THEN GOTO WORD_ERROR
EXT2_OK
   OUTPUT ACCUMULATOR,HEX4_FMT
   RETURN


*************************************************************************


DISP_NEXT_WORD

*  DISPLAY THE NEXT WORD (16 BIT QUANTITY) FROM THE ANALYZER MEMORY

   CALL READ_NEXT_OPERAND          GET THE NEXT STATE

   IF INPUT_ERROR <> 0 THEN GOTO WORD_ERROR
WORD_READ_OK
   OUTPUT ACCUMULATOR,HEX4_FMT
   RETURN


*************************************************************************


READ_NEXT_OPERAND

   INCREMENT REQUESTED_ADDRESS
   INCREMENT REQUESTED_ADDRESS
   CALL EXMODE_PW
```

```
    CALL CHECK_FOR_BLOCK          DONT ALLOW IT TO CROSS BLOCK BOUNDARIES

    IF INPUT_ERROR = 0 THEN TAG_WITH OPERAND_USED     MARK THIS AS A USED STATE
SKIP_RET
    LOAD INPUT_DATA
    RETURN


EXMODE_PW

BCMODE_PW
      INPUT ABS,REQUESTED_ADDRESS,QUALIFIED      READ THE NEXT STATE
      RETURN

READ_MEMORY_OPERAND


*************************************************************************


CHECK_FOR_BLOCK

    IF TASK = 3 THEN RETURN     DISABLE FOR STATE (TEMPORARY)

*  THIS ROUTINE CHECKS TO BE SURE THAT AN INVERSE ASSEMBLY BLOCK WILL
*  NOT BE CROSSED INTO BY AN INSTRUCTION BEING DECODED IN A PREVIOUS
*  BLOCK.  FOR MORE INFORMATION ON THIS PROCEDURE, SEE STEVE WILLIAMS.
*
*  THE OPERATION OF THIS ROUTINE:
*
*     1.  IF A DISASSEMBLY BLOCK HAS BEEN PREVIOUSLY ENCOUNTERED
*         DURING THE DECODING OF THE CURRENT INSTRUCTION, THIS
*         ROUTINE ASSUMES THAT WE ARE STILL IN THAT BLOCK, AND
*         SETS "INPUT_ERROR" TO MAKE THE INVERSE ASSEMBLER THINK
*         THAT A READ ERROR DID OCCUR.
*
*     2.  IF THERE WAS A NORMAL ERROR IN READING THIS STATE, NO
*         DISASSEMBLY BLOCK IS ASSUMED, BUT "INPUT_ERROR" DOES
*         REFLECT THAT AN ERROR OCCCURRED.
*
*     3.  IF THE CONTROL TAG (BITS 17 AND 16 OF INPUT_TAG) INDICATES
*         THIS STATE TO HAVE BEEN PREVIOUSLY USED AS A FIRST INSTRUCTION,
*         THEN A DISASSEMBLY BLOCK BOUNDARY IS SAID TO HAVE
```

```
*          ENCOUNTERED.  THE DISASSEMBLY_BLOCK FLAG IS SET, AND
*          AN INPUT ERROR IS FLAGGED SO THAT THE INVERSE ASSEMBLER
*          CAN DO THE PROPER THINGS WITH THE ERROR FLAG.
   IF DISASSEMBLY_BLOCK = TRUE THEN SET INPUT_ERROR,1
   IF INPUT_ERROR <> 0 THEN RETURN              LEAVE IF READ ERROR
   LOAD INPUT_TAG          GET THE TAG ASSOCIATED WITH THE NEW STATE
   IF 17,16 <> 11B THEN RETURN   11B CONTROL TAG IS ONLY TAG NOT OK TO USE.
   SET DISASSEMBLY_BLOCK,TRUE          A DISASSEMBLY BLOCK IS PRESENT
   SET INPUT_ERROR,1     MAKE IT LOOK LIKE THE READ FAILED
   RETURN


************************************************************************


SHOW_SIZE

* DISPLAY THE SIZE OF THE OPERATION (BYTE, WORD, LONG)
   OUTPUT PERIOD
   CASE_OF 7,6
      OUTPUT B
      OUTPUT W
      OUTPUT L
      GOTO ILLEGAL_OPERAND
   CASE_END
   ROTATE RIGHT,6              MOVE SIZE TO LSB
   AND ML2                     ONLY LOWER 2 BITS ARE THE SIZE FIELD
   STORE SIZE_FIELD            SAVE THE SIZE FIELD
   LOAD INITIAL_DATA           RESTORE THE ORIGINAL OPCODE
   POSITION ABS,TAG_COLUMN     MOVE OVER TO OPERAND FIELD
   RETURN


************************************************************************


IMM_DATA_11_9

* DISPLAY THE 3 BIT IMMEDIATE DATA
* VALUES 1-7 ARE INTERPRETED AS 1-7, VALUE 0 IS INTERPRETED AS 8
```

```
     OUTPUT NUM_SIGN                    IMMEDIATE DATA
   IF 11,9 <> 0 THEN GOTO IMM_NOT_0
      OUTPUT "8"
      RETURN
IMM_NOT_0
      ROTATE RIGHT,9          MOVE DATA TO LOWER 3 BITS
      OUTPUT ACCUMULATOR,DEC1_FMT_3              DISPLAY THE 3 BIT #
      LOAD INITIAL_DATA        RESTORE THE INITIAL OPCODE
      RETURN


************************************************************************


CONDITION_CODE

*  DISPLAY THE APPLICABLE CONDITION CODES FOR Scc, DBcc AND Bcc INSTR

   CASE_OF 11,8
      GOTO COND_T_RA           RA FOR Bcc, T FOR Scc AND DBcc
      GOTO COND_F_BSR          BSR, F FOR Scc AND DBcc
      OUTPUT "HI"
      OUTPUT "LS"
      OUTPUT "CC"
      OUTPUT "CS"
      OUTPUT "NE"
      OUTPUT "EQ"
      OUTPUT "VC"
      OUTPUT "VS"
      OUTPUT "PL"
      OUTPUT "MI"
      OUTPUT "GE"
      OUTPUT "LT"
      OUTPUT "GT"
      OUTPUT "LE"
   CASE_END
   RETURN

COND_T_RA

*  DISPLAY A "T" (TRUE) FOR Scc AND DBcc. "RA" FOR Bcc INSTR

   CASE_OF 12,12
      OUTPUT "RA"              BRA INSTRUCTION
```

```
         OUTPUT "T"                ST AND DBT INSTRUCTIONS
    CASE_END
    RETURN
COND_F_BSR

*  DISPLAY AN "F" (FALSE) FOR Scc AND DBcc, OR BSR

    CASE_OF 12,12
       OUTPUT SR               BSR INSTRUCTION
       OUTPUT "F"                SF AND DBF INSTRUCTIONS
    CASE_END
    RETURN


*************************************************************************


ILLEGAL_OPERAND
    POSITION ABS,TAG_COLUMN
    OUTPUT ILLEGAL
    OUTPUT " Operand    "
    ABORT


*************************************************************************


DATA_ERROR
    OUTPUT "Data Error"
    ABORT


*************************************************************************


ILLEGAL_OPCODE
    OUTPUT ILLEGAL
    OUTPUT " Opcode"
    ABORT


*************************************************************************


WORD_ERROR
    OUTPUT FOUR_STAR
    RETURN
*************************************************************************
```

```
INCOMPLETE_OPCODE
    OUTPUT "incomplete opcode"
    ABORT


*************************************************************************


ADDR_REG_11_9

*  DISPLAY THE ADDRESS REGISTER FOUND IN BITS 11 THRU 9

    LOAD INITIAL_DATA          BE SURE THE INITIAL OPCODE IS LOADED
    ROTATE RIGHT,9             MOVE THE REGISTER FIELD TO 3 LSB

ADDR_REG_2_0
    OUTPUT A
    GOTO REG_TYPE_SHOWN

DATA_REG_11_9

*  DISPLAY THE DATA REGISTER FOUND IN BITS 11 THRU 9

    LOAD INITIAL_DATA          BE SURE THE INITIAL OPCODE IS LOADED
    ROTATE RIGHT,9             MOVE THE REGISTER FIELD TO 3 LSB

DATA_REG_2_0
    OUTPUT D

REG_TYPE_SHOWN

*  DISPLAY THE REGISTER NUMBER FOUND IN THE REG FIELD (BITS 11 THRU 9)

    OUTPUT ACCUMULATOR,DEC1_FMT_3  DISPLAY THE REGISTER NUMBER
    LOAD INITIAL_DATA             RESTORE THE ORIGINAL OPCODE
    RETURN


*************************************************************************


EXTENDED_MATH

*  DISPLAY "X", SIZE AND OPERANDS FOR ADDX AND SUBX INSTRUCTIONS
```

```
        OUTPUT "X"
        CALL SHOW_SIZE                  DISPLAY THE OPERATION SIZE
        CASE_OF 3,3
           SET MODE_FIELD,000B          SRC & DEST ARE DATA REG DIRECT MODE
           SET MODE_FIELD,100B          SRC & DEST ARE PREDEC ADDR IND MODE
        CASE_END
        CALL EA_DISP_MODE_SET           DISPLAY THE SOURCE OPERAND
        ROTATE RIGHT,9                  MOVE DESTINATION TO LOWER 3 BITS
        OUTPUT COMMA
        GOTO EA_DISP_MODE_SET           DISPLAY THE DESTINATION OPERAND


************************************************************************


MARK_PREFETCH
*
*  FOR INSTRUCTIONS THAT HAVE KNOWN, UNUSED PREFETCH STATES, FIND THE
*  NEXT OPCODE STATE, THEN SEE IF ITS ADDRESS IS THE ADDRESS IMMEDIATELY
*  FOLLOWING THE CURRENT INSTRUCTION.  IF SO, THEN ITS UNUSED PREFETCH.
*
     INPUT REL,PREFETCH_LOC,QUALIFIED  LOOK FOR THE NEXT PROGRAM READ STATE
     CALL CHECK_FOR_BLOCK           DONT CROSS INVERSE ASSEMBLY BLOCK
     IF INPUT_ERROR <> 0 THEN RETURN   NO MORE PROG READ STATUS STATES
     LOAD REQUESTED_ADDRESS         ADDR OF STATE SUSPECTED TO HAVE UNUSED PREFETCH
     ADD 2                          MOVE AHEAD TO POINT AT UNUSED PREFETCH
     SUBTRACT INPUT_ADDRESS         SUBTRACT OFF THE CURRENT ADDRESS
     IF 31,0 <> 0 THEN RETURN
     TAG_WITH UNUSED_PREFETCH
     RETURN


************************************************************************


UNKNOWN
*
*  THIS IS TO DISPLAY THE FUNCTION CODES OF UNKNOWN MEMORY CYCLES
*
     OUTPUT "unknown ("
     AND  01110000B
     ROTATE RIGHT,4
```

```
      OUTPUT ACCUMULATOR,BIN3_FMT
      OUTPUT ")"
      LOAD INPUT_STATUS
      RETURN

************************************************************************

INT_ACK

*

* THIS ROUTINE CHECKS FOR INT_ACK OR CPU SPACE OPERATION

*

      LOAD INPUT_ADDRESS
      SET INT,TRUE
      IF 23,4 <> INT_VAL THEN GOTO CPU
      OUTPUT "int_ack"
      LOAD INPUT_STATUS
      RETURN

CPU
      SET INT,FALSE
      OUTPUT "cpu space"
      LOAD INPUT_STATUS
      RETURN


************************************************************************
```

D

Assembler Error Messages

Detection and Listing

The assembler (ASM.EXE) detects and lists all errors noted in the source program. The 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, generally only the first error noted will be reported.

Each error message contains an error code, a cursor (^ ^) that points to the error location in the source statement, and a statement that indicates the line number of the previous source statement that was in error to facilitate error tracing.

A summary of the number of errors within the program, along with a brief description of all error codes noted, is given at the end of the program listing.

The error message format is as follows:

```
            ERROR-(code), see line XX   ^



      Error        Line Numbers      Cursor Pointing
      Code         of Previous      To Source Statement
                     Error               Error
```
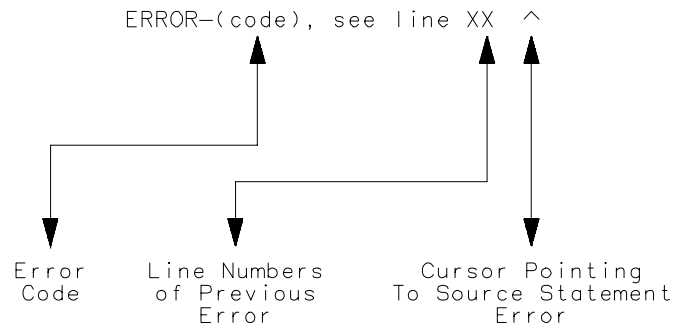
10391M01

Error Codes

A list of the error codes (in alphabetical order) along with a description of their meaning is shown below:

Code	Error Definition
AS	ASCII STRING - The ASCII string was terminated improperly.
DE	DEFINITION ERROR - Indicated symbol must be defined prior to it being referenced. Symbol may be defined later in the program sequence.
DS	DUPLICATE SYMBOL - Indicates that the defined symbol noted has been previously defined in the program assembly sequence.
ET	EXPRESSION TYPE - The resulting type of expression is invalid. Absolute expression was expected and not found.
IC	ILLEGAL CONSTANT - Indicates that the assembler encountered a constant that is not valid. For example: 109B (9 is invalid)
IE	ILLEGAL EXPRESSION - Specified expression is either incomplete or an invalid term was found within the expression.
IO	INVALID OPERAND - Specified operand is either incomplete and inaccurately used for this operation. This occurs when an unexpected operand is encountered or the operand is missing. If the required operand is an expression, the error indicates that the first item in the operand field is illegal.

Code	Error Definition
IS	ILLEGAL SYMBOL - Syntax expected an identifier and encountered an illegal character or token.
MO	MISSING OPERATOR - An arithmetic operator was expected, but was not found.
MP	MISMATCHED PARENTHESIS - Missing right or left parenthesis.
SE	STACK ERROR - Indicates that a statement or expression does not conform to the required syntax.
TR	TEXT REPLACEMENT - Indicates that the specified text replacement string is invalid.
UC	UNDEFINED CONDITIONAL - Conditional operation code is invalid.
UO	UNDEFINED OPERATION CODE - Operation code encountered is not defined or the assembler does not allow 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.
US	UNDEFINED SYMBOL - The indicated symbol is not defined as a label.

Index

16511B, 1-23, 3-2
68010 connections, 1-31
68010 inverse assembler, C-1
68010 mnemonics, 1-32
68010.BAT, 1-3, 1-22
68010.CMD, 1-3
8085, 2-6
8085 inverse assembler, A-1
8085 inverse assembly, 2-4
8085.BAT, 1-3, 1-22
8085.CMD, 1-3, 1-22
\ HP64700, 1-8
\ HP64700\ TABLES, 1-5, 1-7, 1-13

A

ABORT, 4-9
Absolute addresses, 3-18
Absolute mode, 3-12
Absolute positioning, 4-44
ACCUMULATOR, 3-1 / 3-3, 3-17, 4-6, 4-43
   convert, 4-43
ACCUMULATOR instructions, 3-24
Acquisition memory, 3-1 / 3-2, 4-34
   Reading, 3-8
ADD, 3-4, 3-24, 4-10
Additional labels, 1-25
ADDR, 1-23 / 1-24, 3-8
ADDR_B, 1-23 / 1-24
AIAL, 1-3, 1-5, 1-7
AND, 4-11
Apostrophes, 4-7

Arithmetic, 3-4, 4-55
Arithmetic instructions, 3-24
ASCII, 3-28, 4-1, 4-4, 4-26
ASCII source code, 1-11
ASCII STRING, D-2
ASCII/ASC, 4-12
ASM, 1-22
ASM.EXE, 1-3, 1-5, 1-7 / 1-8, 1-11, 1-13, D-1
Assembler options, 1-15
Assembling source code, 1-13
Asterisk, 4-6 / 4-7
AUTOEXEC.BAT, 1-8

B

BASE_TITLE, 4-13
Batch files, 1-22
Baud rate, 1-9
Binary, 4-7
Blank lines, 4-3
Branching, 3-2
Building a configuration file, 1-23
Building a custom inverse assembler, 1-11

C

CALL, 3-2, 3-28, 4-14, 4-16, 4-47
Carets, 4-7
CASE, 3-2 / 3-3, 4-8
CASE_END, 4-15
CASE_OF, 3-20, 4-15 / 4-16

I

I68010.R, 1-27
I68010.S, 1-3
I8085.R, 1-14
I8085.s, 1-3
IAL, 3-1, 4-2
IAL environment, 3-1 / 3-2
IAL variables, 3-3
IALDOWN, 1-21 / 1-22, B-17, B-19
   prompts, 1-21
IALDOWN.EXE, 1-3, 1-8, 1-12, 1-19
IF, 3-2 / 3-4, 3-20, 4-8, 4-15, 4-28 / 4-29
IF_NOT_MAPPED, 3-18, 3-20, 3-22, 4-15, 4-30 / 4-31
ILLEGAL CONSTANT, D-2
ILLEGAL EXPRESSION, D-2
ILLEGAL SYMBOL, D-3
Immediate, 4-6
INCLUSIVE_OR, 4-32
Incomplete status, 3-21 / 3-22, B-1, B-6, B-15
INCREMENT, 4-33
Infinite loop, 4-39
INITIAL_ADDRESS, 3-30
INITIAL_DATA, 3-30
INITIAL_FLAGS, 3-22, 3-31, B-18 / B-19
INITIAL_OPTIONS, 3-22, 3-31, B-18 / B-19
INPUT, 3-11 / 3-12, 3-15 / 3-16, 3-29, 4-34 / 4-36, 4-45, 4-49
INPUT ABS, 4-36
INPUT REL, 4-35
INPUT,ABS, B-3 / B-4, B-11
INPUT_ADDR_B, 3-30
INPUT_ADDRESS, 3-30, 4-34
INPUT_DATA, 3-30, 4-34, B-1, B-10
INPUT_DATA_B, 3-30
INPUT_ERROR, 3-16, 3-31, 4-34
INPUT_STATUS, 3-30, 4-34, B-1, B-3, B-10

INPUT_TAG, 3-21, 3-30, 4-35, B-2 / B-3, B-5, B-7, B-9 / B-13, B-18
   values, B-8
INPUT_TAGS, B-15
INSTALL.BAT, 1-3, 1-5, 1-7 / 1-8
Instruction, 4-5
   upper-case, 4-5
Instructions, 3-27
INVALID OPERAND, D-2
Invasm field, 1-19, B-6 / B-9, B-15 / B-17
   options, B-17
Inverse assembler, 2-5, 2-9
   development, 3-4
   example, 1-27, 3-6
   illegal disassembly, B-8
   operation, 2-2, 3-7
   size, 3-20
   speed, 3-20
   synchronizing, B-6 / B-7
   titles, 3-28
Inverse assembly algorithm, 1-11
Inverse assembly process, 2-4

L

Label, 4-6
   invalid, 4-4
   length, 4-5
   upper/lower-case, 4-5
   valid, 4-4
Label field, 4-4
Label names, 4-4
LABEL_TITLE, 4-37
Labels, 1-25, 3-28
Length of lines, 4-3
Line format rules, 4-3
Linking
   Configuration file, 1-26
   Inverse assembler, 1-26
LOAD, 4-38

Read operation, 2-8
Relative mode, 3-12
Relative positioning, 4-44
Relocatable code, 1-14
Relocatable file, 1-11, 1-19, 1-21
    downloading, 1-12
Relocatable object code, 1-14
Result, 4-6
RETURN, 3-7, 4-14, 4-16, 4-47
RETURN_FLAGS, 3-22, 3-31, 4-31, B-12 / B-14
Rolling backwards, B-9
ROTATE, 4-48
RS-232C configuration, 1-9
RUN key, 1-32, 2-2

Stop Bits, 1-9
STORE, 4-51
String, 4-6
String constants, 4-8
SUBTRACT, 3-4, 4-52
Symbol table, 3-18 / 3-19, 4-30
Symbolic addresses, 3-18
Symbolic operand
    definition, 3-28
Symbols, 4-30
Synchronizing the inverse assembler, B-6
Syntax, 4-1
Syntax rules, 4-2
System software, 2-2
System tags, 3-21, B-7

S

SEARCH_LIMIT, 3-16, 3-31, 4-36, 4-49
Semicolon, 4-6 / 4-7
SET, 4-50
SET statement, 1-8
Sign bit, 3-4
Software compatibility, B-5, B-19
Software files, 1-3
Software installation, 1-3, 1-5, 1-7
    on a flexible disk, 1-6
    on a hard disk, 1-4
Source code
    assembling, 1-13
    assembly, 1-11
Space, 4-7
STA, 3-11
STA instruction, 2-4, 2-6 / 2-7
    decoding, 3-8
Stack, 3-2
STACK ERROR, D-3
STAT, 1-23 / 1-24, 3-8, 3-15, 4-35, B-1, B-3
State analysis, 1-23
State Listing menu, 1-30
Status, 2-7

T

Tab, 4-7
TAG_WITH, 3-21, 4-53, B-3 / B-4, B-14
Tags, 3-21, 3-30, 4-35, 4-53, B-3
TASK, 3-23, 3-31, B-5
Termination, 4-5
Termination of lines, 4-3
Text editor, 4-1 / 4-2
TEXT REPLACEMENT, D-3
THEN, 4-29
Trace specification, 2-2
Two's complement, 3-4
TWOS_COMPLEMENT, 4-54
TYPE, 1-17

U

UNDEFINED CONDITIONAL, D-3
UNDEFINED OPERATION CODE, D-3
UNDEFINED SYMBOL, D-3

US ASCII, 4-1
   non-printable, 4-1 / 4-2
User-tags, 3-21, B-7

V

Valid file description, 1-21
Valid filenames, 1-21
VAR, 3-3
VARIABLE, 3-28
VARIABLE/VAR, 4-55

W

Write operation, 2-8