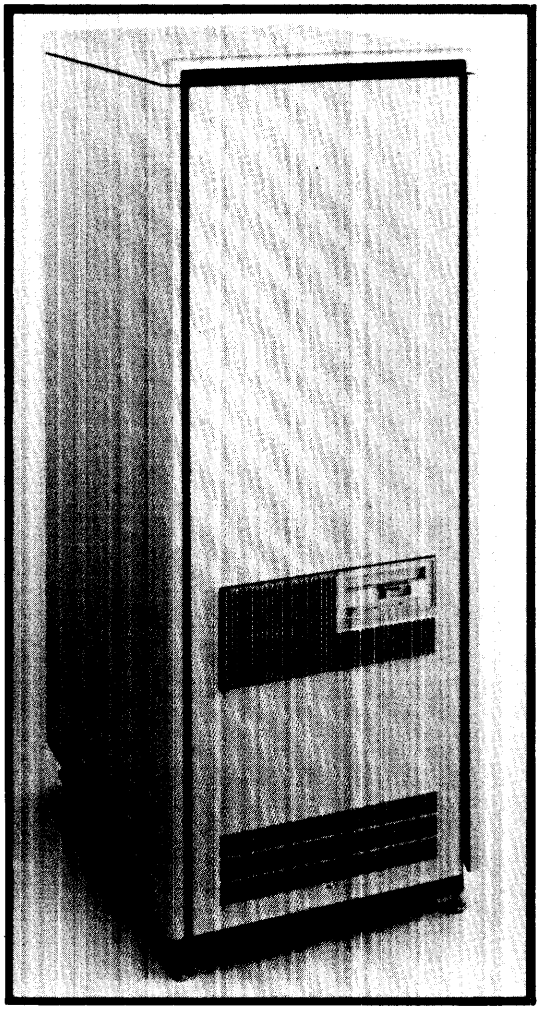
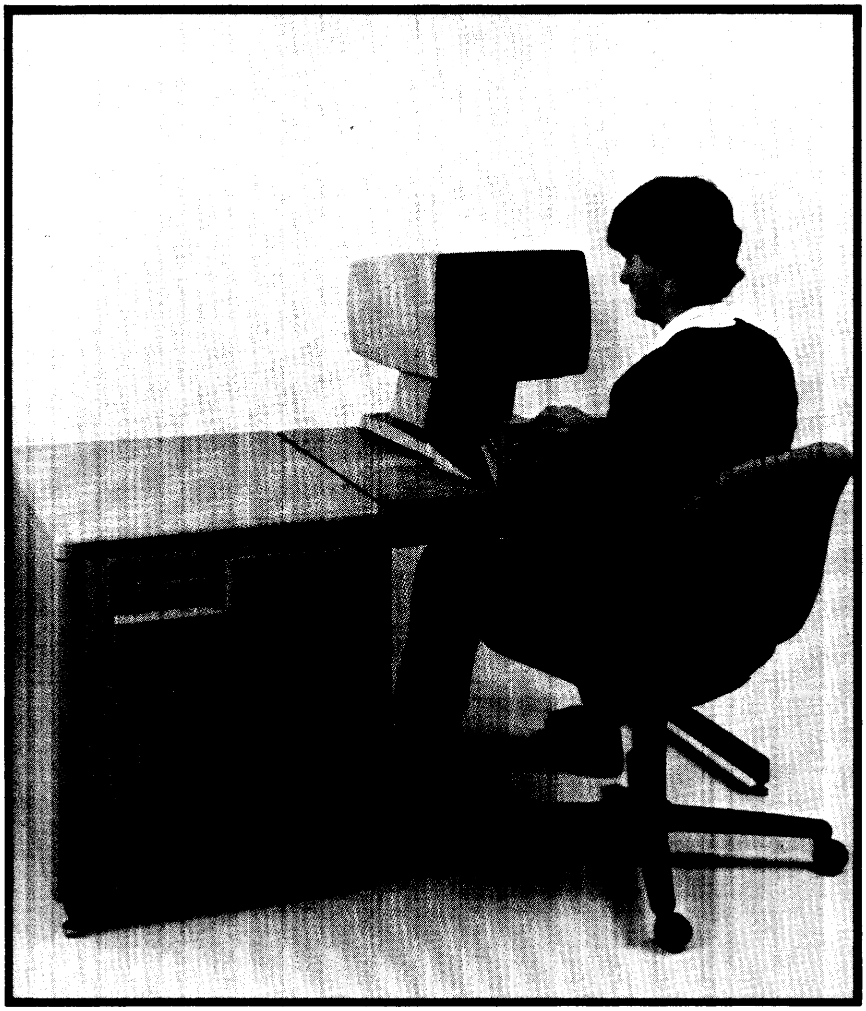
HEWLETT
PACKARD

# HP 92045A Microprogramming Package

## Reference Manual

# HP 1000 A-Series





# HP 92045A Microprogramming Package

## Reference Manual

Includes:
Paraphraser Programming
WLOAD WCS Loader and
PROM Burn Program

HEWLETT PACKARD

HEWLETT-PACKARD COMPANY
Data Systems Division
11000 Wolfe Road
Cupertino, California 95014

MANUAL PART NO. 92045-90001
Printed in U.S.A. February 1982

# PRINTING HISTORY

The Printing History below identifies the Edition of this Manual and any Updates that are included. Periodically, Update packages are distributed which contain replacement pages to be merged into the manual, including an updated copy of this Printing History page. Also, the update may contain write-in instructions.

Each reprinting of this manual will incorporate all past Updates, however, no new information will be added. Thus, the reprinted copy will be identical in content to prior printings of the same edition with its user-inserted update information. New editions of this manual will contain new information, as well as all Updates.

To determine what software manual edition and update is compatible with your current software revision code, refer to the appropriate Software Numbering Catalog, Software Product Catalog, or Diagnostic Configurator Manual.

First Edition ................................ Feb 1982

## NOTICE

The information contained in this document is subject to change without notice.

HEWLETT-PACKARD MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Hewlett-Packard shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance or use of this material.

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

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

# PREFACE

The purpose of this manual is to provide the user with the information needed to develop and write microprograms which will run on the A700 processor for HP 1000 systems. The scope of this manual is complete in that no other manuals are required as references for writing microprograms. It covers the following major subjects:

**Part I - Why Microprogramming**

- Microprogramming Concepts
- Microprogram Controllable Functions

**Part II - Microprogramming Methods**

- Microprogramming Preparation
- Word Types
- Microorders
- Writing Microinstructions

**Part III - Microprogramming Support Software and Hardware**

- Using the Paraphraser Microassembler
- Writable Control Store Support Software
- PROM Control Store Support Software
- HP 12156A Floating-Point Card Microprogramming

**Part IV - Microprogramming Examples**

- Example Microprograms

**Appendixes**

- Includes paraphraser information including summaries of microorders, microorder phrases, and a summary of floating-point microinstructions
- Includes the base set listing, and information on debugging microcodes
- There is also a fold-out functional block diagram.

# PREFACE (Continued)

To gain a better understanding of the processor hardware, refer to the *HP 1000 A700 Computer Reference Manual,* part no. 02137-90001. For information on the Control Store Cards used for microprogram storage in this computer, refer to the *HP 1000 A700 User Control Store Installation and Reference Manual,* part no. 02137-90003. Installation of the HP 12156A Floating-Point Card is covered in the *HP 12156A Floating-Point Processor Kit Installation and Reference Manual,* part no. 12156-90001. The Writeable Control Store driver ID.41 is described in its own manual, *RTE Driver ID.41 For 12153A WCS Cards Reference Manual,* part no. 92045-90002.

# CONTENTS

## PART I - WHY MICROPROGRAMMING?

Section 1 Page

### MICROPROGRAMMING CONCEPT

Microprogramming Overview ...................... 1-2
Program Analysis Methods ........................ 1-3
The Microprogramming Process .................... 1-4
Executing Your Microprogram ..................... 1-6
Summary ......................................... 1-6

Section 2 Page

### MICROMACHINE DESCRIPTIONS

Microprogram Controllable Computer Functions .... 2-1
Overall Circuit Description ........................ 2-5
Microprogram Control ............................... 2-5
Arithmetic/Logic Unit .......................... 2-5
Input/Output (Backplane) Section ............... 2-5
Processor Registers ............................ 2-6
Memory ......................................... 2-7
Memory Controller .......................... 2-6
Memory Array ............................... 2-6
Dynamic Mapping System ..................... 2-6
Boot Memory ................................ 2-7
Virtual Control Panel (VCP) ................ 2-7
A Closer Look At The Functions ..................... 2-7
Some Definitions And Timing Points ........... 2-15
How These Functions Interrelate .............. 2-15
Control Store ...................................... 2-17
Description Of The Base Set ........................ 2-18
Operational Overview ............................... 2-19
Instruction Decoding Loops .................... 2-19
Macroinstruction Fetching ..................... 2-20
Macroinstruction Execution .................... 2-21

## PART II - MICROPROGRAMMING METHODS

Section 3 Page

### MICROPROGRAMMING PREPARATION STEPS

Microprogramming Hardware ..................... 3-1
Microprogramming Support Software .............. 3-2
The Paraphraser Microcode Microassembler ...... 3-3
Driver ID.41 ................................ 3-3
WLOAD ....................................... 3-3
Loading The Microprogramming Support Software ..................................... 3-3
PROM Code Generator ......................... 3-4
Preparatory Steps ................................. 3-4
Debugging Microcode ............................ 3-4

Section 4 Page

### MICROINSTRUCTION FORMATS

Microinstruction Binary Format .................. 4-1
Definition Of Word Types ........................ 4-1
Word Type 1 .................................. 4-3
Word Type 2 .................................. 4-4
Word Type 3 .................................. 4-5
Word Type 4 .................................. 4-5
Word Type 5 .................................. 4-6
Word Type 6 .................................. 4-6
Arithmetic Data Paths ........................... 4-7
ALU Functions ................................... 4-7
Standard ALU Functions ....................... 4-8
Special ALU Functions ........................ 4-8
Shift Functions ................................. 4-22
Microorder Definitions .......................... 4-23
Microcode Restrictions and Considerations ........ 4-27
Memory And I/O Microorders ................... 4-27
Fetch Microorders ............................ 4-27
BFB And IFCH Microorders ..................... 4-27
RDIO And WRIO Microorders .................... 4-27
Memory Reads ................................. 4-27
Map References ............................... 4-27
Base Register and Data and Code Maps ........ 4-28

Section 5 Page

### TIMING CONSIDERATIONS

Computer Timing ................................. 5-1
Memory and I/O Access ........................... 5-1
Processor Clock Freeze .......................... 5-2
Data Transfers .................................. 5-3

Section 6 Page

### MAPPING TO USER'S MICROPROGRAMMING AREA

Control Store Mapping Method .................... 6-1
Software Entry Points ........................ 6-2
User Instruction Group ....................... 6-2
UIG Opcode Blocks ........................ 6-4
User Area UIG Blocks ..................... 6-4
HP Reserved UIG Blocks ....................... 6-5
User's Area Mapping Example .................. 6-5
Branching To Your Microprogram ........... 6-6
Returning To The Base Set ................ 6-7
Calling Microprograms ........................ 6-8
Handling Interrupts ............................. 6-9
Example Of Checking Interrupt Condition ....... 6-9
Microcode Time-Out ........................... 6-10
Registers Reserved for the User ................. 6-10
Microprogramming Considerations ................. 6-10

Section 7 Page

### WRITING MICROPROGRAMS

Planning and Preparation ........................ 7-1
The Paraphraser ................................. 7-1

# CONTENTS (Continued)

Paraphraser Rules .......... 7-2
  Control Statement .......... 7-3
  Comments .......... 7-3
  Numbers .......... 7-3
Microinstruction Sentence .......... 7-4
  Labels .......... 7-4
  Directives .......... 7-5
    Origin Directives .......... 7-5
    Align Directives .......... 7-6
    Define Directive .......... 7-6
  Microinstruction Specifications .......... 7-7
Writing Microinstructions .......... 7-7
  Sentences .......... 7-7
  Phrases .......... 7-9
  Writing Phrases .......... 7-9
  Defaulted Fields .......... 7-9
    Field Conflicts .......... 7-10
    Word Type Conflicts .......... 7-10
    Unrecognizable Phrase .......... 7-10
    Bad Field Default .......... 7-10
Description Of Phrases .......... 7-11
  Branching Phrases .......... 7-11
  Arithmetic Phrases .......... 7-13
    Basic Arithmetic Phrases .......... 7-14
    Arithmetic Phrases With Shift or Rotate .......... 7-16
    Arithmetic Phrases With Carry-in Modifier .......... 7-17
    Arithmetic Phrases With ALU Special .......... 7-19
    Arithmetic Phrases With Immediate Data .......... 7-21
  Conditional Phrases .......... 7-23
    Conditional Branching Phrases .......... 7-23
    Conditional Return Phrases .......... 7-23
    Conditional SP0 Phrases .......... 7-24
  Special Phrases .......... 7-24
  Field Forcing Phrases .......... 7-24

**PART III - MICROPROGRAMMING SUPPORT SOFTWARE AND HARDWARE**

Section 8 Page

**USING THE PARAPHRASER**

Loading MPARA .......... 8-1
Using The Paraphraser Microassembler .......... 8-1
Execution Command .......... 8-2
The Paraphraser Output .......... 8-3
  Binary Object Code .......... 8-3
  Paraphraser Output Listings .......... 8-3
Error Messages .......... 8-5
  Label Errors .......... 8-5
  Translation Errors .......... 8-6

Section 9 Page

**WRITABLE CONTROL STORE (WCS) SUPPORT SOFTWARE**

WCS Mapping .......... 9-1
Using WLOAD .......... 9-2
WLOAD Commands .......... 9-2
  ON Command .......... 9-3
  OF Command .......... 9-3
  Equate Command .......... 9-3
  Remove Command .......... 9-3
  Status Command .......... 9-3
  Load ASCII Command .......... 9-4
  Display Mapping Command .......... 9-4
  LU Command .......... 9-4
  Read Command .......... 9-5
  Initialize Command .......... 9-5
  Background Command .......... 9-5
  Write Data Command .......... 9-5
  Load Binary Command .......... 9-6
  Test Command .......... 9-6
  Exit Command .......... 9-6
  Transfer File .......... 9-7
  Return From Transfer File .......... 9-7
  Help File .......... 9-7
  Comment .......... 9-7

Section 10 Page

**WLOAD PROM BURN TAPE FUNCTION**

PCS PROM Specificatons .......... 10-1
Using WLOAD for Burn Tapes .......... 10-1
Burn Tape Command .......... 10-2
Error Reporting .......... 10-3
Output Format .......... 10-3
PROM Burn Output .......... 10-5

Section 11 Page

**FLOATING POINT PROCESSOR**

General Description .......... 11-1
Basic Capabilities .......... 11-1
General Operation .......... 11-3
  Interface to the Micromachine .......... 11-3
  Sequence of Operation .......... 11-3
  Transfer of Control Word .......... 11-4
  Transfer of Input Operands .......... 11-4
  Transfer of Results .......... 11-5
  Transfer of Error Conditions .......... 11-5
Writing Microcode For The FPP .......... 11-6
  General Floating-Point Microcode .......... 11-6
  Floating Point Control Word Fields .......... 11-6
  Paraphraser Floating Point Sentence .......... 11-8
  Floating Point Division .......... 11-9
    Paraphraser Division Sentence .......... 11-10
  Control Store Starting Address .......... 11-10

# CONTENTS (Continued)

PART IV - MICROPROGRAMMING EXAMPLES

Section 12 Page
MICROPROGRAMS
Branching To The Example Programs ............ 12-1
Buffer Initialization Example ..................... 12-3
Shell Sort Example .............................. 12-6
Privileged Driver Example ...................... 12-11
Structure of Example Program .................. 12-12

Appendix A Page
SUMMARY OF WORD TYPES ................. A-1

Appendix B Page
SUMMARY OF MICROORDERS ............... B-1

Appendix C Page
SUMMARY OF MICROORDER PHRASES .................................... C-1

Appendix D Page
FLOATING POINT MICRO-INSTRUCTIONS ................................ D-1

Appendix E Page
BASE SET LISTING ............................ E-1

Appendix F Page
FUNCTIONAL BLOCK DIAGRAM.............. F-1

Appendix G Page
DEBUGGING MICROCODE .................... G-1
Diagnostic Connections ............................ G-1
Programming the Logic Analyzer ................. G-2
Diagnostic Window Microcode ..................... G-3

Index ........................................... I-1

# ILLUSTRATIONS

Title Page

Overview of Microprogramming Steps .............. 1-2
Major Sections of the Computer .................... 2-3
Simplified Microprogram Control Section .......... 2-14
Control Store Map ................................ 2-17
Microinstruction Word-Type Binary-Format Summary ...................................... 4-2
Single Word, Single Bit Shifts ..................... 4-24
Double-Word, Single-Bit Left Shifts ............... 4-25
Double-Word, Single-Bit Right Shifts ............. 4-26
Example of Microprogram Source File ............. 7-8

Title Page

Example of a Paraphraser Label Listing ........... 8-4
Example of a Paraphraser Floating Field Listing ........................................ 8-4
Address Label Errors ............................. 8-5
Translation Errors ................................ 8-6
Translation Format of PROM Burn Code .......... 10-3
Example PROM Binary Code Printout ............ 10-4
Floating-Point Processor Data Paths .............. 11-2
Jump Table to Example Microprograms ........... 12-2
Flowchart of Shell Sort Program ................. 12-7

# TABLES

Title Page

Computer Functions .............................. 2-8
Preparatory Steps ................................ 3-5
Manual and Software References .................. 3-6
Summary of A700 Computer Word Types .......... 4-7
Microorder Definitions ........................... 4-9
Control Store UIG Software Entry Point Assignments .................................. 6-3

Title Page

Microprogrammable Floating-Point Operations ...................................... 11-2
Microprogrammable Floating-Point Functions ..... 11-6
Connector J4 Signal Identification ................. G-2
Connector J5 Signal Identification ................. G-2
Functional Block Diagram ........................ F-3

## *PART I – Why Microprogramming?*

Section 1 Microprogramming Concept

Section 2 Micromachine Descriptions

## *PART II – Microprogramming Methods*

Section 3 Microprogramming Preparation Steps

Section 4 Microinstruction Formats

Section 5 Timing Considerations

Section 6 Mapping to User's Microprogramming Area

Section 7 Writing Microprograms

## *PART III – Microprogramming Support Software and Hardware*

Section 8 Using the Paraphraser

Section 9 Writable Control Store (WCS) Support Software

Section 10 WLOAD PROM Burn Tape Function

Section 11 Floating Point Processor

## *PART IV – Microprogramming Examples*

Section 12 Microprograms

## *APPENDIXES*

Appendix A Summary of Word Types

Appendix B Summary of Microorders

Appendix C Summary of Microorders Phrases

Appendix D Floating Point Microinstructions

Appendix E Base Set Listing

Appendix F Functional Block Diagram

Appendix G Debugging Microcode

Index

# SECTION 1
# MICROPROGRAMMING CONCEPT

# PART I
# Why Microprogramming?

# MICROPROGRAMMING CONCEPT

SECTION 1

The HP A700 processor for HP 1000 systems has a microprogrammed architecture to provide flexibility of micromachine operation. It allows future firmware enhancements of the instruction set and provides you with significant performance increases for your application.

Microprograms in computers offer the following advantages:

- Reduction of program execution time. By developing microprograms for often-used techniques, program execution time is significantly decreased. Execution time is reduced because:
  - Many instruction fetches are eliminated.
  - Microinstructions typically execute three to ten times faster than Assembler instructions.
  - Multiple operations can occur during a single microinstruction.
  - The microinstruction word width (32 bits in the A700 processor) provides a larger instruction repertoire than available with the Assembler word width (16 bits).
  - Many more registers and computer instructions are available to the microprogrammer than are available to the high-level language programmer.
- Implementation of customized computer instructions. Customized instructions (i.e., microprograms) can provide facilities not otherwise available. Examples are:
  - Post indexing and/or preindexing macroinstructions
  - Stack macroinstructions.
  - Special arithmetic macroinstructions (double integer, decimal, etc.)

Types of applications that can be programmed:

- Sort routines (e.g., bubble, shell, radix-exchange, and quicksort).
- Arithmetic or Floating Point Calculations that can take advantage of the HP 12156A Floating Point Processor.
- Transcendental Functions (e.g., sine, square root, and logarithms).
- Fast Fourier Transform (FFT).

You may also create microprograms to control your own customized hardware. Some microprogram examples are given in Part IV.

Microprogramming has disadvantages in some areas when compared to high-level language programming. For example:

- Microprogramming of all or almost all routines for an application program can be cumbersome and unprofitable. An analysis should be made first to determine those areas that can benefit most from microprogramming.
- Microprograms are not relocatable in control store.
- Microprograms written for the A700 processor cannot be used on other HP processors without being rewritten.

Although additional effort is required to become familiar with the processor in order to write a microprogram, the results are usually well worth the effort. The use of the paraphaser microassembler facilitates programming in such a way that the wide (32 bit) microword is easily coded. The following paragraphs outline the considerations involved when you decide to microprogram.

## 1-1. MICROPROGRAMMING OVERVIEW

The first consideration for an application program, or perhaps a library routine running in an RTE environment, is the execution time. Does it or any parts of it have to be faster? This may or may not be obvious in external operation (i.e., waiting time is too long for a line printer output when a certain calculation is performed, terminal response is too slow, etc.).

Some method of analyzing the programming environment must be used to identify the areas which consume excessive time. Consider the basic methods described below:

- Employ programming analysis devices (e.g., the HP 1610 Logic Analyzer) which attaches to the computer. This is the most accurate but most expensive method.
- Perform a programmatical analysis. This is a compromise over the above method, it is less costly but less accurate.

In summary, the first step is to find out what would be advantageous to microprogram. An analysis of the programming environment may reveal, for example, that setting up a microprogram for a seldom-used library routine would not give a cost effective return in overall software efficiency.

## 1-2. PROGRAM ANALYSIS METHODS

The first analysis method (use a programming analysis device) described above under Microprogramming Overview is beyond the scope of this manual.

The second analysis method described above requires a special program that monitors executing programs. It should record just where most of a program's execution time is spent, and it should point out to the user which sections of code can be optimized or microprogrammed to speed up program execution and throughput.

For example, this monitor program could insert counters in the program being tested to determine how many times an instruction or routine is executed.

There are other ways of obtaining useful timing information. For example, you can use the interrupt method as follows:

- Use a time-scheduled program to monitor the desired program.
- Reserve a "word block counter," for example, at every 500 words or so of main memory.

In the time-scheduled program of the interrupt method, each time the device interrupts, the P-register could be sampled and the count incremented for the associated "word block counter." That is, a record is generated for the program location counter at periodic intervals. This could be done several hundred thousand times and, at the end of the sample period, a percentage of time spent in each area of memory can be obtained. Then . . .

- The load map of the program being analyzed can be examined to determine which parts of the program could possibly be microprogrammed to decrease execution time.
- The resolution for your analysis program could be changed, as could other parameters in the program to obtain the desired profile.

This is the general idea of how an activity profile generation program could be used. Also you can refer to the *Contributed Library Catalog,* part no. 22999-90040, for programs you may be able to use.

Once the activity profile generation program output is analyzed for any excessive computer time, you are ready to concentrate on a particular area for microprogramming. However, keep in mind the following:

- The maximum benefit of microprogramming will not be realized by simply imitating Assembly language instructions in microroutines.
- In order to determine specifically what to microprogram, the computer functions and program intent should be studied before you begin to write your microprogram. The final result will be a microprogrammed solution that executes in much less time and is totally or at least partially transparent.

An overview of the steps to take in order to get your microprogram into operation is covered in the following paragraph.

## 1-3. THE MICROPROGRAMMING PROCESS

Figure 1-1 provides an overview of the steps involved in microprogramming the A700 processor. The figure illustrates the following:

- After a program analysis, the entry point (microaddress) for the control store module that you will be using must be determined.
- The microprogram is then written as a source file for the Paraphraser microassembly language according to the information given in Part II of this manual. Using Edit/1000 it is corrected and stored in a disc file.
- The paraphraser is executed (run MPARA). The microprogram source file is translated by MPARA and microassembled to generate object code. It will also provide a source listing, and an error list (if any). Optionally, according to the source program command statement, it will provide a floating field listing and a label listing.
- Subsequent editing on the source program using Edit/1000 can correct any errors.
- The object code microprogram usually will be loaded into Writable Control Store (WCS) using the WLOAD Utility and tested for fault free operation. Bugs in the microprogram are eliminated by again editing the source program, translating it with MPARA, storing it again in WCS and testing its operation until operation is free of errors.

NOTE

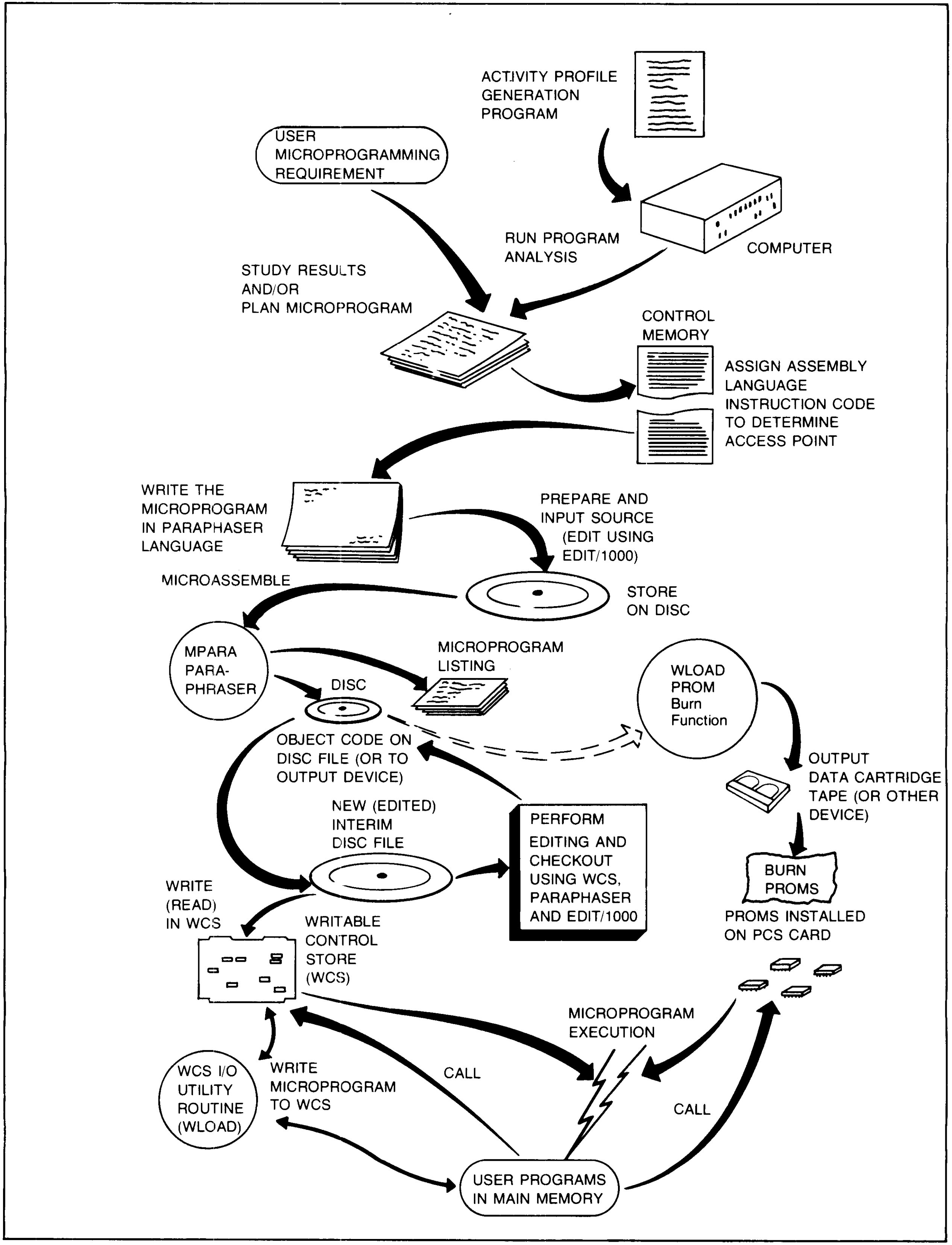
> The HP 12153A Writable Control Store Kit is an important part of developing microprogramming. The developing of programs to store in Writable Control Store (WCS) or PROM Control Store (PCS) is the primary purpose of this manual (described in sections 9 and 10). Information on WCS and PCS cards will be found in the HP 1000 A700 User Control Store Installation and Reference Manual, part no. 02137-90003.

The ready-to-run microprogram can be stored in two ways:

- It can be left in WCS.
- You can create a permanent microprogram through the use of the Control Store PROM Burn Program of WLOAD. This software, in turn, can be used to generate the mask tapes or data files which are used to have Programmable Read Only Memory (PROMs) fused or "burned." The PROMs can then be installed on the HP 12155A PROM Control Store (PCS) Card.

The advantages of executing microprograms from WCS are:

- WCS can be reused for many microprograms.
- WCS can be used to swap microprograms in and out of the system to suit a variety of users.


ACTIVITY PROFILE GENERATION PROGRAM
USER MICROPROGRAMMING REQUIREMENT
COMPUTER
RUN PROGRAM ANALYSIS
STUDY RESULTS AND/OR PLAN MICROPROGRAM
CONTROL MEMORY
ASSIGN ASSEMBLY LANGUAGE INSTRUCTION CODE TO DETERMINE ACCESS POINT
WRITE THE MICROPROGRAM IN PARAPHASER LANGUAGE
PREPARE AND INPUT SOURCE (EDIT USING EDIT/1000)
MICROASSEMBLE
STORE ON DISC
MPARA PARA-PHRASER
MICROPROGRAM LISTING
DISC
WLOAD PROM Burn Function
OBJECT CODE ON DISC FILE (OR TO OUTPUT DEVICE)
OUTPUT DATA CARTRIDGE TAPE (OR OTHER DEVICE)
NEW (EDITED) INTERIM DISC FILE
PERFORM EDITING AND CHECKOUT USING WCS, PARAPHASER AND EDIT/1000
BURN PROMS
PROMS INSTALLED ON PCS CARD
WRITE (READ) IN WCS
WRITABLE CONTROL STORE (WCS)
MICROPROGRAM EXECUTION
WCS I/O UTILITY ROUTINE (WLOAD)
WRITE MICROPROGRAM TO WCS
CALL
CALL
USER PROGRAMS IN MAIN MEMORY

8200-8

Figure 1-1. Overview of Microprogramming Steps

The disadvantages are:

- Microprograms in WCS can be destroyed by an errant user of the system.
- When computer power is removed, your microprogram is lost and must be reloaded.
- Each WCS card requires an I/O slot in the computer; although a PCS card also requires a slot it stores twice as much microcode.

The advantages of fusing (or burning) the microprogram into PROMs are:

- The program is permanently stored on the PCS, and when the power is removed from the computer it does not have to be reloaded.
- One PCS card contains twice as much control store as does a WCS card; therefore, potentially fewer card slots are used if large storage is needed.

The disadvantage is:

- There is much more involved in storing and changing the microprogram with PROMs than there is with WCS such that "bugs" in the microprogram will be harder to correct.

## 1-4. EXECUTING YOUR MICROPROGRAM

If your microprogram is stored in PROMs, it can be executed immediately through User Instruction Group (UIG) instructions. Whether your microprogram is contained in WCS or PCS, it can link Assembly language routines to microprograms. The hardware and firmware map each UIG instruction to a unique control memory destination. UIG instructions are covered in Section 6.

Microprograms that reside in WCS execute at the same speed as do those residing in PCS. Both WCS and PCS resident microprograms can be used along with the base set in control store. The base set is defined as the computer's standard instruction set of microprograms.

## 1-5. SUMMARY

To effectively create a microprogram, the programmer must have the following:

- An understanding of what to microprogram.
- An understanding of the HP A700 processor operation and its architecture.
- Knowledge of the methods used to map to and access control store.
- Knowledge of the appropriate microprogramming hardware and sofware products.

One way to obtain this knowledge is to attend the Hewlett-Packard Computer Microprogramming course. The above subjects are all covered in the remaining portions of this manual but remember that most important first step, find out *what you should microprogram.*

# SECTION 2
## MICROMACHINE DESCRIPTIONS

# MICROMACHINE DESCRIPTIONS

SECTION 2

This section covers detailed information that you should know about HP 1000 A700 computer operation. You should study it before attempting to write a microprogram for it. The following paragraphs describe:

- The hardware functions controlled by microinstruction.
- Aspects of the base set microprogrammed operation that will be important to your microprogramming.

To implement your own microprograms you will not need to know the computer design at the logic circuit level. The information in this book should be entirely sufficient for your needs. The base set discussion will help you to become aware of the existing microprogram's operation. Below is a look at the overall computer followed by details on registers and other functions.

## 2-1. MICROPROGRAM CONTROLLABLE COMPUTER FUNCTIONS

Figure 2-1 illustrates the four major sections of the computer that control computer functions. In order of importance they are the following:

- Microprogram Control Logic Section
- Arithmetic/Logic Unit (ALU) Section.
- Memory Controller Section.
- Input/Output (Logic) Section.

The other sections shown are memory array, memory maps, boot memory, processor registers, and control store. The base set on the lower processor connects by control store bus to the optional Writable Control Store (WCS), PROM Control Store (PCS), and Floating Point Processor (FPP).

Accessories shown in the overall block diagram that are directly associated with microprogramming are the following:

- HP 12153A Writable Control Store (WCS) Kit.
- HP 12155A PROM Control Store (PCS) Kit.
- HP 12156A Floating Point Processor (FPP) Kit.

Important information about accessories related to microprogramming is covered in other sections of this manual, and a general description of the controllable computer functions is contained in the following paragraphs.

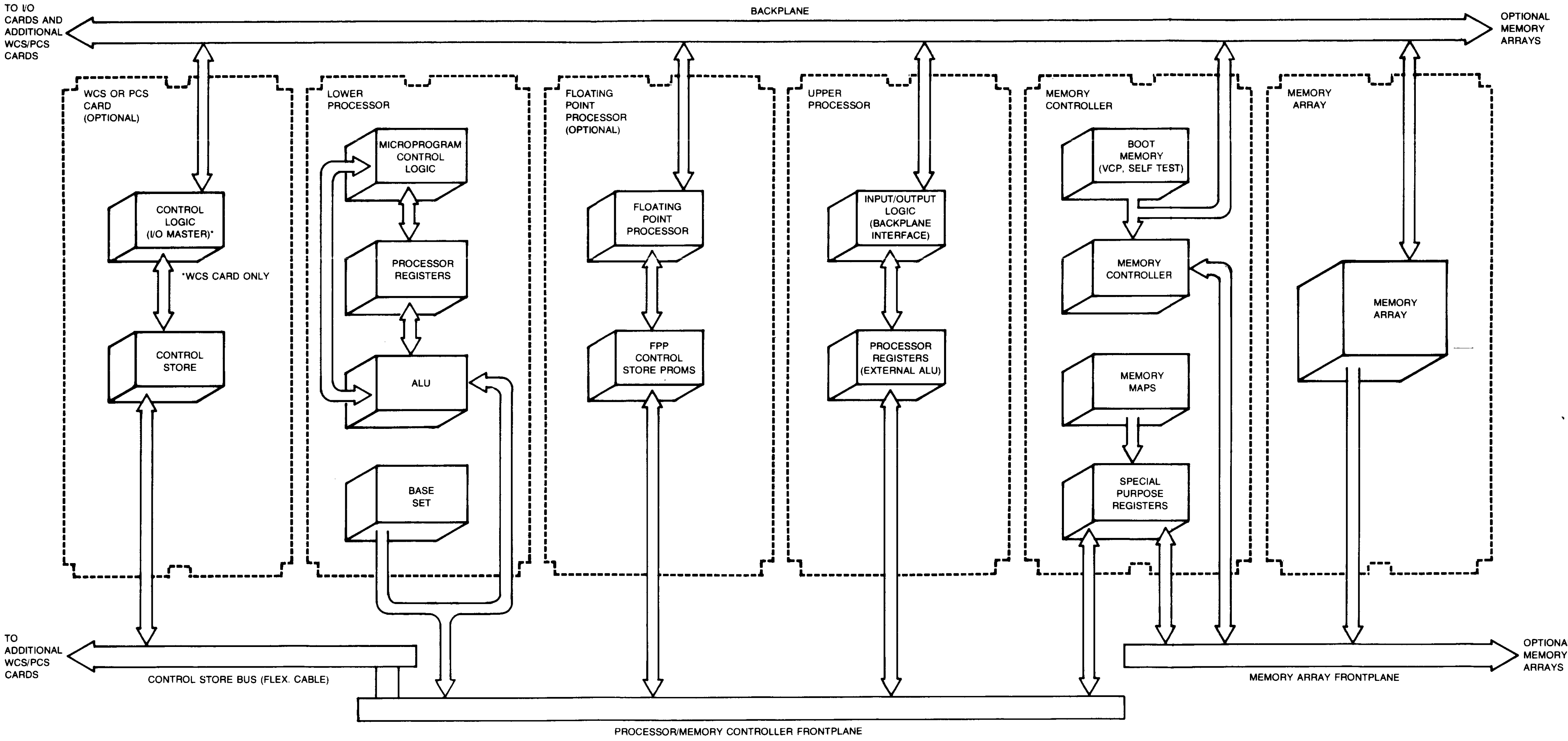

TO I/O CARDS AND ADDITIONAL WCS/PCS CARDS
BACKPLANE
OPTIONAL MEMORY ARRAYS
WCS OR PCS CARD (OPTIONAL)
CONTROL LOGIC (I/O MASTER)*
*WCS CARD ONLY
CONTROL STORE
LOWER PROCESSOR
MICROPROGRAM CONTROL LOGIC
PROCESSOR REGISTERS
ALU
BASE SET
FLOATING POINT PROCESSOR (OPTIONAL)
FLOATING POINT PROCESSOR
FPP CONTROL STORE PROMS
UPPER PROCESSOR
INPUT/OUTPUT LOGIC (BACKPLANE INTERFACE)
PROCESSOR REGISTERS (EXTERNAL ALU)
MEMORY CONTROLLER
BOOT MEMORY (VCP, SELF TEST)
MEMORY CONTROLLER
MEMORY MAPS
SPECIAL PURPOSE REGISTERS
MEMORY ARRAY
MEMORY ARRAY
TO ADDITIONAL WCS/PCS CARDS
CONTROL STORE BUS (FLEX. CABLE)
PROCESSOR/MEMORY CONTROLLER FRONTPLANE
MEMORY ARRAY FRONTPLANE
OPTIONAL MEMORY ARRAYS

Figure 2-1. Major Sections of the Computer

## 2-2. OVERALL CIRCUIT DESCRIPTION

The HP 1000 A700 Computer consists of two processor cards, memory controller and memory array. The upper processor card contains the Input/Output or Backplane Interface section and some of the processor registers. The lower processor card contains the Arithmetic Logic Unit (ALU), the microprogram control section, some processor registers, and the base set firmware. The processor cards communicate with the memory controller card over the Processor/Memory Controller Frontplane. Additional details of these sections are described below.

## 2-3. MICROPROGRAM CONTROL

The Microprogram Control Logic includes a "look-up" table, a micromachine sequencer, control store (memory) for the card, a four-deep microprogram subroutine stack, a microinstruction register, and decoders. The "look-up" table has an entry for each macroinstruction that selects the appropriate base set microinstruction address.

The base set, WCS, PCS, and FPP receive addresses and send data, respectively, to and from the Microprogram Control over the Control Store Bus.

The base set (standard instruction microprogram) is stored in 2k-microwords of ROM and is part of the "basic" computer. Extensions of the control store which you can use for your microprogramming are the 4k-microword WCS, the 8k-microword PCS, and the FPP card with either 2k- or 4k-microwords. (A microword is 32 bits wide and contains one microinstruction.) No more than four control store cards are allowed which can be any combination of WCS, PCS cards, and one FPP card. However, the maximum number of microinstructions words that can be addressed is 16k-words.

WCS communicates with the I/O section to allow microprograms to be written to and read from the Memory Array. They connect to each other through the backplane, through which some signals for the control and loading of WCS are passed from the I/O section.

The WCS, PCS,and FPP are connected through the frontplane to the micromachine control store buses in a parallel fashion. Therefore, when executing microcode there is no difference in addressing the base set or the microinstructions you have added. The microinstruction output is the same. No matter how the microprogram control is physically implemented, together they appear as one large microprogram facility.

## 2-4. ARITHMETIC/LOGIC UNIT

The Arithmetic/Logic Unit (ALU) section of the computer includes most of the hardware required to actually carry out commands of the microinstructions. It provides the logic to perform arithmetic and logical operations on the data.

## 2-5. INPUT/OUTPUT (BACKPLANE) SECTION

The Input/Output (I/O) Section serves as the backplane interface between the computer and external devices. The I/O hardware responds either to Microprogram Control stimuli (for computer-initiated data or control operations) or to device stimuli (for device-signaling attention requests), and hence becomes the active communication link between the computer and peripheral devices.

## 2-6. PROCESSOR REGISTERS

The directly-addressable working registers of the processor include an instruction register, a program counter register, an accumulator, a memory return register, and temporary storage (scratch) registers for use in processor operations.

There are also indirectly-addressed register files including general-purpose, privileged, and special-purpose external registers. Special-purpose external registers are located on the memory controller card and FPP, and communicate with the processor over the processor/memory controller frontplane. Some of the privileged registers are reserved for use by the base set, and the special-purpose registers are used for map addressing, parity errors, interrupt updates, and reserved for future requirements.

## 2-7. MEMORY

**2-8. MEMORY CONTROLLER.** The Memory Controller controls the main memory of the computer system. It contains 32 Memory Maps with 32 registers each for dynamic mapping of the memory array. The Memory Maps store information used to generate the physical address of data accessed during a memory cycle (see Dynamic Mapping below).

Memory Protect is part of memory control. Memory protect can interrupt and report the logical address of any instruction that attempts to read or write into protected pages of memory, or execute certain instructions flagged by the Dynamic Mapping System. The Read and Write protect bits are stored in the Memory Maps. However, an I/O device using Direct Memory Access (DMA) can access protected memory for both reads and writes; however, this occurrence is prevented by the RTE operating system.

**2-9. MEMORY ARRAY.** All programs and data reside in the Memory Array section. The Assembly language macroinstructions stored in main memory are decoded by the Microprogram Control Section of the processor.

**2-10. DYNAMIC MAPPING SYSTEM.** The 32k words logical address space of the HP 1000 A700 architecture is expanded to 16 megawords of physical memory through a process called "mapping." The Dynamic Mapping system uses Map RAMs located on the Memory Controller to store information for generating the physical address of the data accessed in a memory cycle. The map RAMs extend the 15 logical address bits to the equivalent of 24 physical address bits using dynamic memory mapping.

Since the maps involved can be dynamically reloaded, accessibility to the entire physical memory is accomplished. When the base register is enabled, two maps are used together to extend the addresses. These are called the Data and Code Maps. (Note: The base register is not supported by the RTE-A.1 operating system; however, an update at some future time may include this support.)

**2-11. BOOT MEMORY.** Boot memory including ROM and RAM is stored on the memory controller card, and it is used each time the computer is powered up. The ROM includes a self test program which is a short processor checkout program.

**2-12. VIRTUAL CONTROL PANEL (VCP).** The VCP program is an interactive program stored in the boot memory ROM that is written in HP 1000 Assembly code. The VCP enables an optional external device (usually a terminal) to control the processor in a manner similar to a conventional computer control panel. Using the VCP, an operator can access various registers (A, B, P, etc.), examine or change memory, and control execution of a program or load and initiate execution of the operating system or diagnostic.

## 2-13. A CLOSER LOOK AT THE FUNCTIONS

In Table 2-1 the microprogrammable functions of the major computer sections (shown in Figure 2-1) and the registers are described at a level which is consistent with microprogramming requirements. Wherever appropriate, the associated microorders and microinstruction fields are mentioned. Microorders and Microinstruction Fields are covered in detail in Section 4 of this manual. Table 2-1 also has a section which briefly describes the bus system.

Refer to the functional block diagram in Appendix E when reviewing the table. Once you understand the computer's architecture and the effect of microorders, microorder phrases, and the paraphraser microassembler, you will need only the detailed block diagram, and microorder charts to write microprograms.

Figure 2-2 is a simplified block diagram of the Microprogram Control Section of the processor. In a "conventional" computer control section, specific hardware is dedicated to each function performed by the instruction set. The major advantage of the "conventional" approach is higher speed to process the instruction set. The major disadvantage is inflexibility for special applications or for enhancements.

In the microprogrammed computer the logical functions are defined by a series of microinstructions contained in microroutines (subroutines of microcode). The microroutines are contained in the processor memory called "control store." The basic instructions for processor operation contained in permanent control store is called the "base set." In this way the microprogrammed computer is more flexible than the "hardwired" computer since microprograms can be modified to reprogram the hardware to perform different functions.

The Microprogram Controller executes microinstructions at a very high rate which is fast enough to keep the main memory busy almost all the time. Thus, the speed penalty for using the microprogrammed architecture is essentially not a factor, especially when processing the base set of instructions.

Since this computer is completely microprogrammable, user programs can be made to execute much faster with the application of user microprogramming.

Table 2-1. Computer Functions

## MICROPROGRAM CONTROL SECTION

**Entry Look-Up Table:**
Entered after microorder JTAB. Has a microroutine entry point for each macroinstruction in basic instruction set. 16-bit instruction decoded to 8-bit address vectored to microaddress space from hexadecimal 100 to 1FF.

**Micromachine Sequencer:**
Increments microroutine under microprogram control. Next address comes from one of the following address locations:

1. Current address plus one;
2. Branch address in the current 64-word block or anywhere in the 16k (3FFF hex) word address field;
3. Branch address supplied by look-up table;
4. Return address from microsubroutine stack.

Subroutine branches will increment the stack pointer and push the current microaddress plus one onto the microsubroutine stack. A return from subroutine will supply the next microaddress from the top of the stack and decrement the stack pointer. On power-up, the micromachine sequencer starts execution at location 0000 (hex).

**Control Store:**
The control store on the processor contains the base set of microinstructions and microcode diagnostics. Control store is extended over the frontplane bus to PCS, WCS and FPP cards. The processor can address 16k- words of which 2k-words are the base set.

**Decoder:**
The Decoder receives the microinstruction word that is output from the control store and decodes it. The decoded outputs are the processor control lines which cause the microinstruction to be implemented.

## ARITHMETIC LOGIC UNIT

**Arithmetic Data Paths:**
Microinstructions can specify the following data paths:

1. A-bus operand which can come from either a register-file register or Immediate Data from the microinstruction;
2. B-bus operand which can come from a register-file register or any other processor register;
3. ALU output data to be stored in a processor register or written to main memory.

**ALU Functions:**
There are two categories of ALU functions as follows:

1. Standard operations (coded in the microinstruction *ALU field*)
2. Special operation (coded in the microinstruction when SPEC is in the *ALU field* and the special operation to be performed is specified in the *ALUS field*).

Note (parenthetic mnemonics refer to processor status bits): For all arithmetic operations, Carry Flag (CF) and ALU Overflow (ALOV) will be updated with the ALU results. Microorder ZERO forces the ALU output (F-Bus) to all 0s, and disables the update of the following conditions in the condition register during the current cycle: CF, ALOV, SF (Shift Flag), YZ (Y-bus all Zeroes), Y15 (Y-bus bit 15 set), and B15 (B-bus bit 15).

A. **Standard ALU Functions:**
Arithmetic operations (true 2s complement add or subtract), are always performed with either a carry or a borrow. For add, the carry-in normally defaults to 0 but can be forced to 1 by microorder FCIN. For subtract the borrow is the complement of the carry-in, and the carry-in defaults to 1 but can be forced to 0 by microorder FCIN. The standard functions include logical operations which always result in the "clear flag" and "ALU overflow" being cleared.

Table 2-1. Computer Functions (Continued)

## ARITHMETIC LOGIC UNIT (Continued)

B. **Special ALU Functions:**
When (SPEC) is in the microinstruction *ALU field* with the operation coded in the *ALUS field* the following functions can be performed:

1. Byte swapping and masking, four-bit left rotate, and bit manipulation (microorders ASG and SRG). These operations are performed by the external ALU and operate only on the B-Bus. CF and ALOV processor condition bits are cleared. The four-bit left rotate (RL4) will not affect the processor Shift Flag (SF) bit.

2. Arithmetic operations which may include a shift having multiply, divide, and floating point algorithms. Processor bits CF, ALOV, YZ, and SF represent different conditions used for some of these operations. (See Table 4-1).

**Shift Functions:**
There are three categories of shifts that can be executed in microcode:

1. Single-word, single bit shifts;
2. Double-word, single bit shifts, enabled by (DW) double-word bit;
3. Special function shifts (SPEC in the microinstruction ALU field ).

The processor SF register holds the shifted out bit, and SF will be updated only for shift functions. It is not updated for "four-bit left-rotate" (RL4) and when the microinstruction ALU field contains the ZERO microorder. Details of shift functions are covered under SP0 and SP1 Field in Table 4-1.

## MEMORY SECTION

**Memory Capacity:**
16,384k words physical address space, addressable over 24 memory address lines. Memory array cards may have 64k-words, 128k-words, 256k-words or 512k-words per card, up to a total of four cards per system as long as the maximum capacity is not exceeded. The memory array cards automatically configure themselves in ascending address order as they are installed in the backplane.

**Map RAMs:**
32k-words of memory can be addressed without mapping. Dynamic mapping is achieved through Map RAMs on the memory controller. The Map RAMs convert the 15-bit logical address received into a 24-bit physical address. Map RAM information can be changed by the processor. Memory read or write accesses are initiated by the processor or by an I/O device using DMA. The mapping system is used to separate code and data when the base register is enabled. When the base register is disabled, there are 32 maps each with 32 registers which do address translation. When the base register is enabled, there are 16 pairs of maps. The lower map of a pair is the data map that translates data references, the upper map of a pair is the code map which translates instruction fetches.

**Memory Protect:**
Memory is protected by bits in Map RAM for read and for write. If the processor attempts a read or write to protected memory, a memory protect interrupt is generated but the backplane handshake is allowed to complete. Writes can occur to read protected memory and reads can occur from write protected memory. An I/O device using DMA can read from and write to protected memory.

**Data Format:**
16-bit words and one parity check bit. Each data transfer moves 17 bits. Parity errors generate a parity error interrupt signal.

**Timing:**
A complete memory access to main memory occurs within two clock cycles where the clock cycle is 250 ns. The fastest data transfer rate is 2.0 Mword/sec or 4.0 Mbyte/sec.

Table 2-1. Computer Functions (Continued)

## INPUT/OUTPUT SECTION

**I/O Control and Select Logic:**
I/O timing, signal generation, and I/O address selection take place from this function. The interface control signals are generated as a result of the Microprogram Controller executing I/O microorders.

**Interrupt Control:**
Interrupt sources are latched and prioritized by hardware, and the interrupt source location is stored in a memory trap cell. A microcode interrupt service routine reads the Interrupt Status Register (IST) to determine the interrupt of highest priority and take appropriate action. A read of IST into CT followed by a CT30 will vector to a unique point for each interrupt. The interrupt service routine executes a trap cell as described in the A700 Computer Reference Manual.

**Central Interrupt Latch (CIL):**
The CIL, one of the Special External Registers referenced by microorder SRIN, supplies an updated I/O select code address (refer to Base Set description).

## BUS SYSTEM

The processor/memory-controller frontplane (microprogram control and data bus) connects the memory controller card and the two processor cards for the processor to access memory maps and external registers residing on the memory controller card, and to allow communication between the two cards.

The frontplane bus connects the base set and the control-store accessories with the processor's microprogram control section over which the microinstructions are transferred.

The backplane transfers data and addresses to memory from the processor. Each word of data into the processor is stored in the T-register. All data transfers to and from an I/O card go over the backplane.

## DIRECTLY-ACCESSED REGISTERS

**Directly-Addressable Register Files:**
Register files R00 through R17 (octal) are directly accessed on either the A or B Bus or in the STOR field. Their functional names are:

| Reg. No. (Octal) | Function Name | Reg. No. (Octal) | Function Name | Reg. No. (Octal) | Function Name |
|---|---|---|---|---|---|
| R00 | A | R05 | HP1* | R12 | S2 |
| R01 | B | R06 | HP2* | R13 | S3 |
| R02 | X | R07 | USR* | R14 | S4 |
| R03 | Y | R10 | S0 | R15 | S5 |
| R04 | ACC | R11 | S1 | R16 | S6 |
| | | | | R17 | S7 |

*Reserved registers for HP (HP1 and HP2) and for the user (USR).

**Instruction Register (CT):**
The Instruction Register (designated the CT register since it serves as a general-purpose counter for shift and rotate operations) will be decremented by the SP2 microorder, or it will decrement automatically whenever microorder CTZ or CTZ4 is specified in the microinstruction condition field. This register is loaded with the macroinstructions from memory which are input to the micromachine microprogram "look up" table. The data or instruction is returned on a fetch (FCHB or FCHP) microorder (macro instruction fetch). The counter register must not be overwritten while the instruction is still needed.

Table 2-1. Computer Functions (Continued)

**DIRECTLY-ACCESSED REGISTERS (Continued)**

**IST Register:**
The Interrupt Status Register contains information pertaining to the interrupt system including mask and enable bits. It sets up a priority list of interrupts pending, and allows interrupt bits to be set and and cleared.

**LR Register:**
The Light Register drives 16 LEDs and will display the results of the Self-Test Program.

**MAP Register:**
The register referenced by microorder MAP in the microinstruction *B or STOR field* is a map register located in the memory controller. This register is addressed through MPAR (an indirectly-addressed special external register in the memory controller).

**MEMR Register:**
The MEMR Register controls memory access. MEMR is the lower eight bits of 16-bits of storage and the upper eight bits contain status information. A store to MEMR will not change the status bits. The bits are as follows:

0 - 4 MEMR Register map for main memory accesses. Contains number of Data Map if base register is enabled (Bit 0 = 0).
5 MEMR A/B Addressability bit (bit is 0 if memory locations 0 and 1 address A or B registers, respectively).
6 MEMDIS bit of MEMR. Bit 6 = 1 enables boot memory.
7 MEMR memory system enable bit. Bit 7 = 0 enables memory access.
8 Always zero.
9 Slave-. Logic zero if Slave- signal is asserted on backplane.
10 Parity Error. Logic one if parity error interrupt is pending.
11 A/B Fetch-. Logic zero if last fetch was from A or B register.
12 MTO. Logic one if microcode time out occurred.
13 Mlost. Logic one if memory lost on last power down. Valid for 10 msec only after power is up.
14 Power Fail Warning. Logic one if power going down within 5 msec.
15 TDI. Logic one if interrupts are temporarily disabled.

**BASE Register:**
BASE is the Base Register. When the base register is enabled, all memory references (except FCHP, FCHB, and RDPC) use the Data Map. FCHP, FCHB, and RDPC use the Code Map unless the address is in the Base Register. If the address is in the Base Register, use the Data Map. The Base Register bits are as follows:

0 - 14 Base register value. Value is added to any memory address on base page other than 0 or 1 (A and B regs.) when base register is enabled.

15 If 1 the base register is enabled and if 0 it is disabled.

Note: The base register is not supported by the RTE-A.1 operating system. If the base register is turned on in a microprogram, it must be turned off before completion.

**N-Register:**
The N (Index) Register is a 4-bit register used for indirect addressing of the privileged and non-privileged processor registers and the special external registers of the memory controller. N is decremented by a special microorder DN, and incremented by a special microorder (IN). When N is referenced the upper 12 bits contain status information. A store to N will not change them. The bits are as follows:

0 - 3 N-Register for indirect and special register addresses.

4 - 7 Always *zero* (Not Used).

8 - 15 Status bits same as upper 8-bits of MEMR Register.

Table 2-1. Computer Functions (Continued)

### DIRECTLY-ACCESSED REGISTERS (Continued)

**P-Register:**
The P (Program Counter) Register generally holds the macro-program counter and is used to fetch the next instruction or to get the 2nd or additional operands of multiple word instructions. It can also be used for general purpose reads and writes but the program counter must be saved and restored. P can be incremented by a special microorder.

**Q-Register:**
The Q-Register is available as a microinstruction *B-field* operand. The ALU output is loaded into Q by an SP0 or SP1 microorder (LDQ). Q is used as the least-significant word in double-word shifts and for some special ALU operations. When (Q) is specified in the microinstruction *B field* it is multiplexed with the B bus data to become the ALU input; i.e., it is not actually enabled to the B-bus.

**SR-Register:**
The Switch Register stores the settings of the Frontplane digit switches. The bits are designated as follows:

| | |
|---|---|
| 0 - 7 | Reserved for use by the VCP for start-up option |
| 8, 9 | Available for the user |
| 10 | Reserved for HP microprogram |
| 11 - 15 | Reserved for use by self test firmware |

**T-Register:**
Receives returned data from memory or I/O. T is loaded only from a memory read or fetch or from an I/O read and it can not be loaded directly by the microcode.

### INDIRECTLY-ACCESSED REGISTERS

**GRIN General-Purpose Non-privileged Register File:**
There are 16 indirectly-accessed general-purpose registers which are accessed through the N-register. They are referenced by microorder GRIN in either the microinstruction *B field or STOR field* . They should only be used as "scratch" registers in base-set and user microcodes.

**PRIN Privileged Register File:**
There are 16 indirectly-accessed registers of which several are dedicated for use by the base set. These registers are accessed through the N-register and referenced by microorder PRIN in either the B or STOR field of the microinstruction. Refer to the paragraphs on the Base Set for information on using these registers.

**SRIN Special-Purpose External Registers:**
These registers are located in the memory controller and communicate with the processor over the frontplane bus. They are indirectly accessed by microoder SRIN in the microinstruction *B field or the STOR field* . The register selected is determined by the four low-order bits of the index register (N). The registers are defined as follows:

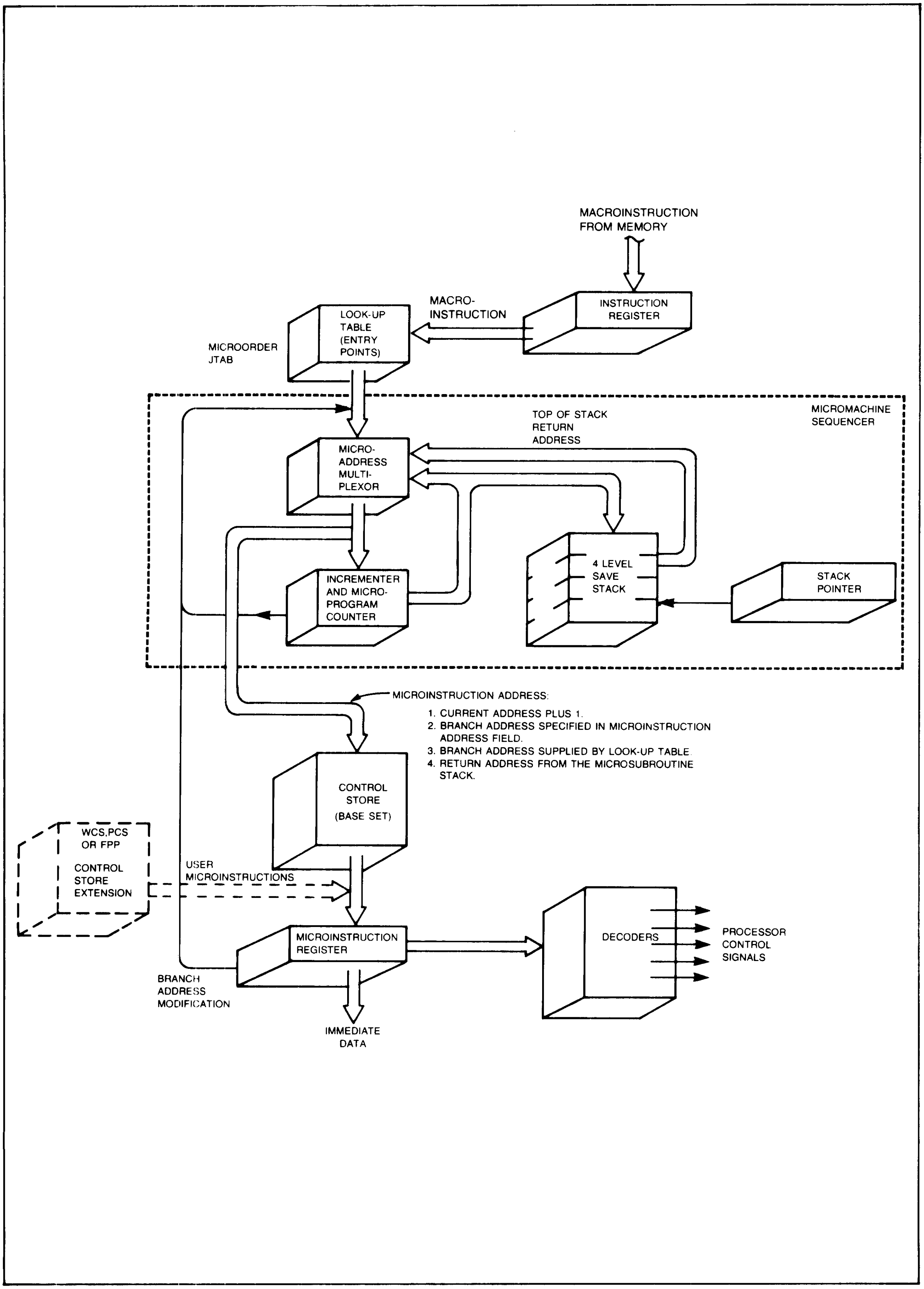
0 MPAR: Map address register which can be written to and read. It contains the 10-bit address presented to the map RAMs for processor access to the map registers. Any read or write to the maps increments MPAR. (Bits 10 - 15 are always zero.)

1 PEL1: Parity Error Latch is a 16-bit read-only register containing the low 16 bits of physical address where the last parity error occurred. It is updated even if parity interrupts are disabled. Addresses are latched for both DMA and processor errors.

2 PEL2: Parity Error Latch 16-Bit read-only register containing the high 8-bits of physical address of the last parity error. This address is stored in the low eight bits of the register, and the remaining 8 bits are always 0.

Table 2-1. Computer Functions (Continued)

| INDIRECTLY-ACCESSED REGISTERS (Continued) | |
|---|---|
| 3 | CIL: Central Interrupt Latch read-only register containing the trap cell address of the last I/O interrupt. The microcode uses this address to update the central interrupt register located in the register file (controlled by base set microcode). |
| 4 - B | HP Reserved. |
| C - F | Reserved. Access to the FPP (see Section 11). |



8200-5

Figure 2-2. Simplified Microprogram Control Section

## 2-14. SOME DEFINITIONS AND TIMING POINTS

Some definitions about control and timing will be clarified next followed by a description of the computer's interrelated functions and its operation.

- A micromachine is hardware that executes microinstructions.

- The microprogram controller always executes "microcoded" microinstructions during "microcycles."

- One microcycle is the time interval required to completely execute a microinstruction.

- A microinstruction is a 32-bit coded word (code definition is called the microcode) that defines specific hardware operations to be performed by the computer.

- Each microinstruction is composed of at least one, and up to seven microorder fields. Each microorder defines a specific operation to be performed in the computer. Some microorders accomplish multiple operations by themselves.

- A field is a contiguous section of bits of the microinstruction that are decoded into microrders; e.g., the ALU field.

- A word-type is a list of fields that comprise the microinstruction. In this computer there are eleven word-types defining the microinstruction formats.

- Microinstructions physically reside in Control Store and are the basic building blocks of microprograms.

- Segments of microprograms may be called microroutines.

- A portion of microcode called from a microroutine will be referred to as a microsubroutine.

Part II of this manual provides specific information on timing that you will need for microprogramming.

## 2-15. HOW THESE FUNCTIONS INTERRELATE

All the functions described in the preceding paragraphs are interrelated in an operational sense through the microprogrammed operation of the computer. Here are a few points to remember:

- The computer is always under microprogram control and executing microinstructions when power is applied.

- A microroutine in the base-set fetches Macroinstructions (Assembly language instructions of the HP 1000 operation codes set) stored in main memory. Each macroinstruction is interpreted as a "pointer" (address) to a microroutine, resident in Control Store, which implements the instruction by executing a sequence of microinstructions.

- The selected microinstructions are loaded into the Microinstruction Register, and data is directed to the appropriate destination by the microprogram invoked.

A few other points should be considered before examining what Control Store (microprogram memory) can accomplish:

- The Microprogram Control Section decodes each microinstruction into fields, then executes the indicated microorders in the proper sequence.

- Each microorder performs a distinct operation and the microorders are not necessarily related to each other in each microinstruction.

Keep the above points in mind as you read through the following steps of how the Microprogram Control Section might operate in a microroutine:

- The microinstruction in the microinstruction register typically calls for the contents of some register to be enabled onto a data/address bus. Then certain "and/or" and "rotate/shift" operations of the ALU take place during the microcycle and, at the end of the microcycle, a specified destination register is clocked to receive the prevailing data from its input lines.

- While a jump-to-subroutine microinstruction presently in the microinstruction register is being executed, the Stack Pointer is incremented to supply the current address plus one to the microaddress stack and following a return from subroutine, this new address will be used to load the microinstruction register in the next cycle.

- Several "branch-on-test" microsubroutines are available (e.g., conditions of carry, the sign, a zero result, presence of a particular bit, etc.) that provide branches to microroutines designed to react to the condition.

- Just prior to microprogram completion, fetching of the next instruction is begun from the currently executing microprogram. There is usually a return to the return address of the microsubroutine stack as specified by the microroutine. Fetching of the next macroinstruction is completed after the address return.

Do not be concerned if the details of microprogram control are not clear to you at present. You will gain more knowledge and understanding of computer operation as you learn the microprogramming language by reading through this manual and writing microprograms. Some further points:

- If the microprogram execution time exceeds the interval between pending interrupts allowed by your particular system application, the interrupts can be lost. Your microprogram must be written to test for pending interrupts if it takes a large amount of time.

- When a pending I/O interrupt is detected, the hardware latches and prioritizes this interrupt with other interrupt sources. Other interrupts are internal to processor operation and take higher priority positions. The microcode interrupt service routine is invoked which services the pending interrupts in prioritized order.

## 2-16. CONTROL STORE

In a general way, you can look at control store as being devoted to serving three areas:

- The computer base set.
- HP microprogrammed accessories (WCS, PCS, and FPP).
- Future HP enhancements (the user can use this area with the reservation that HP may reclaim part or all of it for future firmware packages that may be released).
- The user microprogramming area.

All 16,384 addressable (32-bit) words of control store are logically partitioned into 1k-word modules. Figure 2-3 shows the control store map (represented in 1k word separations) and it identifies the areas of usage listed above. Notice that the 0k- and 1k-word modules are dedicated to the standard base set. The 4k- and 5k-word modules are used for present HP accessories, and 8k- through 11k-word modules are for future HP enhancement firmware but may be used by the user with this reservation. The remaining control store modules of 12k- through 15k-words are reserved for additional microprograms written by you.

| CONTROL MEMORY ALLOCATION | MODULE | ADDRESS (HEXADECIMAL) | DECIMAL | SOFTWARE ENTRY POINT | NUMBER OF USER POINTS |
|---|---|---|---|---|---|
| HP BASE SET | 0k | 0 - 3FF | 00000-01023 | YES* | — |
| | 1k | 400 - 7FF | 01024-02047 | YES* | — |
| HP RESERVED | 2k | 800 - BFF | 02048-03071 | YES* | — |
| | 3k | C00 - FFF | 03072-4095 | YES* | — |
| HP RESERVED, SIS,VIS,FPP | 4k | 1000 - 13FF | 04096-05119 | YES* | — |
| | 5k | 1400 - 17FF | 05120-06143 | YES* | — |
| HP RESERVED | 6k | 1800 - 1BFF | 06144-07167 | NO | — |
| | 7k | 1C00 - 1FFF | 07168-08191 | NO | — |
| HP RESERVED/USER | 8k | 2000 - 23FF | 08192-09215 | YES** | 16** |
| | 9k | 2400 - 27FF | 09216-10239 | NO | — |
| | 10k | 2800 - 2BFF | 10240-11263 | YES** | 16** |
| | 11k | 2C00 - 2FFF | 11264-12287 | NO | |
| RESERVED FOR USER | 12k | 3000 - 33FF | 12288-13311 | YES | 32 |
| | 13k | 3400 - 37FF | 13312-14335 | YES | 32 |
| | 14k | 3800 - 3BFF | 14336-15359 | NO | |
| | 15k | 3C00 - 3FFF | 15360-16383 | NO | |

* HP use only.
** May be used for HP future firmware packages.

Figure 2-3. Control Store Map

## 2-17. DESCRIPTION OF THE BASE SET

A listing of the complete base set, including the JTAB microorder Jump Table, is provided in Appendix E. An overall description of the base set is given below.

The base set microroutines are good examples of microprogramming techniques you may use as a guide for writing your microprograms. Also, you may want to use some of these microroutines in your microprograms as utility microroutines. HP recommends that the user places a copy of any base set subroutines to be used in the user's control store space. DO NOT JUMP INTO THE HP BASE SET. This is because HP reserves the right to modify their base set routines.

The base set microroutines provide you with the capability to execute all the base set instructions described in the *HP 1000 A700 Computer Reference Manual*, part no. 02137-90001. In the base set are:

- Microroutines to execute instructions in the following groups:
  - Memory Reference
  - Alter-Skip
  - Shift-Rotate
  - Input/Output
  - Extended Arithmetic
  - Floating Point (Without FPP)
  - Dynamic Mapping System
  - Double Integer
  - Language Instruction
  - Operating System
- Microroutines that
  - Execute the built-in firmware diagnostics
  - Initiate the macrocoded self test
  - Handle interrupts.
  - Fetch indirect operands.
- Some typical operations performed by the base set microprogram include:
  - A power-up sequence.
  - A short diagnostic check of the processor and memory controller
  - A read/fetch operation to execute an instruction, then fetch the data to perform an ALU operation, and finally storing the data in a register.
  - A write operation (e.g., writing the incremented value in an ISZ instruction).
  - I/O operating routines; e.g., processor-initiated transfers or device initiated transfers of data to perform an ALU operation, and finally storing this data into a register.

The timing relationships involved in operations such as the above mentioned typical operations are covered in Section 5 of this manual.

## 2-18. OPERATIONAL OVERVIEW

The following paragraphs provide an overview of how the Microprogram Control Section performs several operations in parallel in the base set. The references to example addresses are given in hexadecimal. The microroutines for the HP 1000 Assembler instruction codes (macroinstructions) illustrate several techniques that you should be aware of to effectively execute your own microprograms. You may find it helpful to refer to the functional block diagram in Appendix F for assistance in understanding these operations.

## 2-19. INSTRUCTION DECODING LOOPS

Most of the time the processor is executing in a microcoded loop that decodes macorinstructions. This loop (called the JTAB loop) is entered through microinstructions that have the JTAB microorder in the op code (operation code) field. There are two JTAB loops in the &CONTROL section of the base set firmware as follows:

a. Normal macroinstruction decoding that is entered when bit 11 of the switch register (SR register) is closed (logic "0"). There are two microinstructions in this loop: the first with the JTAB microorder, and the second for a branch return if there are no interrupts pending.

b. Diagnostic JTAB loop that is entered when bit 11 of the switch register is open (logic "1"). It contains the same microinstructions as in the normal JTAB loop but information is placed on the Y-BUS during additional microinstructions so that the information can be input to frontplane pinouts.

## 2-20. MACROINSTRUCTION FETCHING

Macroinstruction fetching is the operation which obtains the "next" instruction to be executed from main memory. In this computer, a "look ahead" technique is used for this process. That is, fetching is begun while simultaneously completing the execution of the "current" instruction; and fetching is completed while preparing for execution of this "next" instruction. This is accomplished by starting the fetch operation using the FCHB or FCHP microorder just prior to termination of the "currently" executing instruction microroutine. When the fetched instruction is returned from main memory to the T-register, it will also be stored into CT.

For illustrative purposes, suppose that the "currently" executing microroutine is for an XOR instruction (that had been obtained from main memory location octal 2000). The P-register has already been incremented so that as the microroutine for XOR is completing its execution, the fetch (FCHP) is initiated for main memory location 2001. (Assume that with the completion of the XOR execution, an augend is left in the A-register and that at main memory location 2001 there is an ADA macroinstruction.)

Upon termination of this "current" macroinstruction's routine, control passes to a microroutine in the control firmware of the base set.

The microroutine of the control firmware checks for an interrupt condition, and then branches to the microinstruction containing the JTAB microorder. ("JTAB" means "jump table", and is the microorder that begins instruction decoding.)

The "JTAB" microinstruction increments the program counter, clears a general-purpose flag, and begins a memory access of an MRG address if the instruction loaded in the CT is a memory reference group instruction. In this manner of "look ahead" fetching, the overhead required for instruction fetching is minimized. User microprograms must be designed to terminate in a similar manner.

The JTAB microorder causes the current microaddress plus one to be saved on the microstack. The last result of this multi-functioned microinstruction is that the next sequential microaddress executed will come from locations 100-1FF (hex), and this look-up table entry point is a function of the 16 bits of the macroinstruction.

In the example being used, an ADA instruction from main memory location 2001 has been stored in the instruction register CT and an operand address (assume the address is 300) has been formed in the memory generation logic. The read operation, initiated at the beginning of the JTAB microsubroutine, obtains the operand (the addend) for the ADA instruction from main memory location 300 but the information has yet to arrive in the T-register.

Note that after the program counter is incremented in the JTAB microsubroutine, the program counter points to the location after the opcode. If the instruction is a one-word instruction (such as the CAX instruction), then the the microroutine for that instruction can simply do a "FCHP, RTN" to complete. If additional words or addresses are needed by the instruction (such as the DEF after the LDX instruction), the microroutine for that instruction can begin a memory read using the "RDPC" microorder.

You can see in these examples that it is the microprogrammer's responsibility to complete the instruction with the program counter pointing to the next opcode, begin the fetch, and return to the control firmware.

In the example being used, the operand address (300) formed in the address generation logic is used to read the operand (addend) for the ADA instruction. The ADA microroutine adds the addend to the augend in the A-register, and in the same cycle begins the next instruction fetch.

## 2-21. MACROINSTRUCTION EXECUTION

Execution of the macroinstructions (assembly language instructions) is carried out by the specific microoders contained in the individual microinstructions of the appropriate microroutines as they are decoded from the MIR.

In the example being used, recall that before the operand address (octal 300) was formed in the Address Generation Logic it contained address 2001 (the address of the ADA instruction) and the P register contained 2002 if the rules stated above are followed. Now the content of P is incremented by one due to the JTAB microinstruction line (contains microorder IP). Thus P is adjusted to 2003 in preparation for the fetch (FCHP) operation that will be initiated as the microroutine for the ADA instruction (from main memory location 2001) is being executed.

Again, using the ADA instruction as an example, the microinstruction for the ADA immediately begins a fetch operation from the main memory address (2002) in the program counter (in the "look-ahead" manner previously described) to obtain the next macroinstruction.

The operand is moved from main memory to the A-register in the following way: Recall that the microsubroutine called by JTAB has already begun a read operation if the instruction was an MRG instruction. This read operation gets the ADA operand from main memory (via the T-register), places it on the B-bus, and the ALU adds the contents of the T-register to the A-register (which is specified in the A-bus field) and stores the result in the A-register. If a carry results, the E-bit is set; if two's complement arithmetic overflow results, an O-bit is set. The setting of the E and O bits are enabled using the "ENOE" microorder in the SP0 field.

The last result of this microinstruction for the ADA macroinstruction is to return to the control firmware from the address saved on the microstack by the JTAB microsubroutine, using a "RTN" microorder.

To summarize, the main points you should remember from the above operation description are the following:

- A fetch operation begins in a "look-ahead" manner while the execution of the previous instruction is carried out. Once a branch to your microprogram is made, it is possible for you to stay in the user microprogramming area until it is desired to return to the fetch microroutine. Before returning, however, you should terminate your microprogram properly.

- In regard to the length of time your microprogram executes, it should be written so that interrupts cannot be lost and computer operation will not be suspended. The processor contains a "watchdog" timer that will abort your microroutine if an interrupt is not serviced within 10 milliseconds. Therefore, your microprogram should not be allowed to run more than 10 milliseconds.

Interrupts examples were not included in the operational overview of this section since they are covered in Part II of this manual.

# SECTION 3
## MICROPROGRAMMING PREPARATION STEPS

# PART II
# Microprogramming Methods

# MICROPROGRAMMING PREPARATION STEPS

SECTION 3

When you are ready to begin microprogramming, there are certain initial steps which are necessary to prepare your RTE operating system so that it will accept the microprogramming environment. These steps have to do with the available hardware and software in your computer which includes the following:

- Installation of additional control store memory "hardware" for the storage of your microprograms. This would be either WCS, PCS, FPP, or a combination of these.
- Installation of microprogramming support software for microprogram development, HP 92045A Microprogramming Package. (This software is not needed for running the microprograms.) The software package includes the MPARA paraphraser microprogram assembler, and the WLOAD WCS card loader and PROM burn program.
- The ID.41 driver (also part of the HP 92045A microprogramming support package) is needed for the WCS. ID.41 is in the set of drivers that is loaded at system generation time.

The RTE Microprogramming Support Software package operates in the RTE-A.1 operating system environment. Microprograms may also be developed in the RTE-6/VM environment using MPARA.

## 3-1. MICROPROGRAMMING HARDWARE

The HP 12153A Writable Control Store (WCS) Kit is the recommended hardware for microprogram development and it, of course, can be used for normal execution of your microprograms in your application environment. Each WCS card contains 4k-words of control store.

The other available cards to extend the computer's control-store capacity are the HP 12155A PROM Control Store (PCS) card and the HP 12156A Floating Point Processor (FPP) card. These cards cannot be used for development since the microprograms must be "burned" into PROMs. Therefore, the usual practice is to have a WCS card for development and, after the microprogram has been debugged, PROMs are "burned" and then installed on the cards. The PCS card can have up to 8k-words of control store and the FPP card can have up to 4k-words of control store.

In this computer the WCS and PCS cards are installed in contiguous backplane slots below the lower processor card. Up to four cards total of WCS and PCS or other firmware accessory (such as an HP 12156A floating point processor) in any combination can be installed. The maximum address is 16k-words; therefore, it is possible to install more cards than can be addressed. (The FPP card is installed between the processor cards.)

The WCS card after microprogram development could be replaced with a PCS card for permanent programs. The WCS card is both an I/O card communicating over the backplane for user access to read and write microcode and a processor writable control-store card communicating over the frontplane. The PCS card communicates with the processor only over the frontplane (its backplane connection provides only power and the address priority chain).

WCS cards can be enabled and disabled over the backplane by an OTA 32 instruction (sign bit of A=1 for ON and A=0 for OFF). When it is enabled or ON, the WCS card functions as an extension of the processor's control store and the user cannot access data over the backplane. When it is disabled or OFF, the control store addresses are ignored and the user can access the card through the backplane.

NOTE

> Any PCS card can contain optional HP supplied firmware sets, as well as user-generated PROM firmware.

When WCS cards are in operation the processor continually loads the next microaddress onto the control store address bus. The card addresses are in 1k-word by 32-bit modules. The WCS and PCS cards are prioritized so that each card recognizes only its block of control-store addresses, and a priority chain assures that only one card is driving the control-store data bus at any one time. The processor control store and any PCS cards are at the bottom of the priority chain and will drive the control-store bus only if no other device is driving it.

Each PCS card provides eight 1k-word modules of "read only memory." The address block of each module is set by switches on the card. Each address block also has a switch to disable the block.

Since WCS cards can be turned off and on for backplane or frontplane operation, respectively, a WCS card and PCS card can have the identical address blocks.

Typically, a WCS card will be placed in the backplane in a lower priority slot than a PCS card so that it will have a higher frontplane priority (backplane and frontplane priorities are opposite in priority sequence). The higher priority is necessary if you are going to overlap the addresses. In this case, when the WCS card is on (frontplane enabled), its higher priority will accept the address and disable the PCS card. If the WCS card is off, the PCS card will accept the address. If the PCS card has the higher priority, the opposite operation occurs.

The operational states, hardware supplied, PROM installation, and installation guide lines for WCS and PCS cards are contained in the *HP 1000 A700 Computer User Control Store Installation and Reference Manual,* part no. 02137-90003.

## 3-2. MICROPROGRAMMING SUPPORT SOFTWARE

In order to develop and run microprograms in a dynamic manner in the RTE operating environment you will need the HP 92045A RTE-A Microprogramming Support Software Package which includes the following:

- MPARA microassembler program
- WCS I/O and PROM Burn Utility Routine WLOAD
- ID.41 WCS driver

These programs and the WCS driver are described below.

## 3-3. THE PARAPHRASER MICROCODE MICROASSEMBLER

The paraphaser microcoding microassembler language converts a source microprogram into binary object code which may be directed to an output device and/or stored in a disc file. The paraphraser is a necessary tool for preparing microprograms since the microinstruction word length is 32 bits which makes other coding methods difficult.

The source may be input from an input device or disc file. The disc file is easiest since this file can be the same file developed when writing and editing the program with the HP 1000 Editor. The object code will be in the standard microinstruction format which is recognized by the WLOAD utility routine. The program can supply a source listing, a floating field listing of the microinstructions, a label listing, and a list of any errors.

The paraphraser program name is MPARA. MPARA can run with or without the File Manager, and it requires a minimum of 28k words of memory. All information on preparation of microprograms with the paraphraser and output of the microprograms is contained in Sections 7 and 8 of this manual.

## 3-4. DRIVER ID.41

Driver ID.41 must be configured into the RTE system during system generation to provide software linking between MPARA, WLOAD, and the WCS card.

NOTE

> The microprogramming support software can be included either during system generation or loaded into the system when required.

Driver ID.41 drives HP 12153A WCS cards for reads and writes (from and to main memory) and allows control of WCS board functions. The driver utilizes DMA which provides fast data transfer.

When configured in the RTE system, all WCS cards should have a select code of octal 20 or higher. In the system, the driver can be called directly with an EXEC call, or through the WLOAD program (refer to the *RTE Driver ID.41 For HP 12153A WCS Cards Reference Manual*).

## 3-5. WLOAD

The WCS I/O Utility program WLOAD uses driver ID.41 and transfers microprogram object code into WCS when run by the user. Section 9 in this manual contains information on WLOAD used as an I/O utility. WLOAD also includes a PROM "burn tape" function (see paragraph 3-7).

## 3-6. LOADING THE MICROPROGRAMMING SUPPORT SOFTWARE

The microprogramming support software can be loaded during system generation or on line, using RTE LINK. The exception to this is the driver ID.41 which can be loaded only at system generation time. (Refer to *RTE Driver ID.41 For HP 12153A WCS Card Reference Manual,* part no. 92045-90002.)

## 3-7. PROM CODE GENERATOR

The process of loading the microcode into the PROMs (Programmable Read Only Memory) is accomplished for fusing ("burning") the binary bits into the PROM chip. The binary code for the PROMs is generated by the PROM "burn tape" function of WLOAD that uses the final binary object code of the microprogram as input. The program should be tested and debugged by running the program from a WCS card before making expensive PROMs. For additional information on PROM burning, refer to Section 10 of this manual.

## 3-8. PREPARATORY STEPS

Condensed information on the required preparatory steps for microprogramming appear in Table 3-1 along with references to the sections of this manual (or to applicable documents). The letters in the referenced column are keyed to entries in Table 3-2, and the numerals refer to sections in this manual.

Table 3-2 is a list of HP 92045A Microprogramming Software and HP manuals used by the microprogrammer for the HP 1000 A700 computer systems. Section 12 provides examples of the procedures you may want.

In preparation for microprogramming, the WCS cards to be used must be initialized before they can be used.

## 3-9. DEBUGGING MICROCODE

After you have written your source microcode and fixed any errors found by MPARA, load the object code into WCS and try running it. If its performance is not to your satisfaction you will want to "debug" it. Microcode debugging on the A700 processor is most efficiently accomplished through the use of a logic analyzer. Hewlett-Packard logic analyzers are recommended since they were used throughout the development of the base set and floating point microcode and provided the desired results.

A logic analyzer allows the actual micromachine execution to be followed, and it can be programmed to trace the micromachine execution upon detection of certain conditions. Details on connecting a logic analyzer and information on its use are given in Appendix G (Debugging Microcode).

Table 3-1. Preparatory Steps

| STEP | TASKS | REFERENCE (Table 3-2 or manual sects.) |
|---|---|---|
| 1 | Establish your microprogramming goal. Develop your own microprogram or try one of the examples first. For example, run a short microprogram from start to finish by referring to Section 12. | 1, 12, F |
| 2 | Become familiar with the computer and steps to microprogramming (hardware, control memory mapping). | 2, 3, 5, 6, A |
| 3 | Establish control memory module and mapping scheme. | 2, 6, F |
| 4 | Plan, develop, and write first-pass microprogram (or if desired, a simple sample program). | 4, 7, 8, 12, M |
| 5 | Plan, develop, and write main memory linking method. | 6, 12, L |
| 6 | Place RTE system off-line and power down if not already in this state. | A |
| 7 | Install the optional control store cards: 12153A WCS, 12155A PCS, and 12156A FPP. | A, B, D, E |
| 8 | Generate and configure the RTE system. The WCS driver ID.41 should be found on your primary system disc. | H |
| 9 | Load the necessary microprogramming support software from the following list into your disc files from the primary system disc.<br>— WLOAD<br>— MPARA Paraphraser Microassember | 3, C, D, I, J, K |
| 10 | Microassemble your source with the paraphraser. | 8, J, F |
| 11 | If necessary, correct errors at the source using WCS, HP Edit/1000 and microassemble again. | 1, 8, 9, C, M |
| 12 | Load main memory program that links to microprogram. | C |
| 13 | Execute microprogram from main memory. | 8, 9, C |
| 14 | Correct any logical errors found during microprogram execution. (Fix the source using Edit/1000.) | 8, 9, C, M |
| 15 | If you are planning to "burn-in" PROMs, you must do so from a corrected microassembled object program. Correct source and microassemble until the final object code is obtained. Go to step 16.<br>OR<br>If going to use dynamic microprogramming and your microprogram executes properly, it can be used through WCS. Development is complete at this point unless this was an example program. | 8, 9, C, M |
| 16 | To prepare mask code, run WLOAD PROM code program. | 10, K |
| 17 | Burn PROMs from binary code cartridge tape. | |
| 18 | Mount PROMs on HP 12155A PCS card, test microprogram. | C, L |

Table 3-2. Manual and Software References

| REFERENCE (from Table 3-1) | MANUAL OR SOFTWARE |
|---|---|
| A | HP 1000 A700 Computer Reference Manual, part no. 02137-90001. |
| B | HP 1000 A700 Computer Installation and Service Manual, part no. 02137-90002. |
| C | Your System Programmer's Reference Manual, part no. 92077-90007. |
| D | HP 1000 A700 User Control Store Installation and Reference Manual, part no. 02137-90003. |
| E | HP 12156A Floating Point Processor Kit Installation and Reference Manual, part no. 12156-90001. |
| F | HP 92045A Microprogramming Package Reference Manual, part no. 92045-90001. |
| G | RTE Driver ID.41 for HP 12153A WCS Card Reference Manual, part no. 92045-90002. |
| H | RTE-A.1 General Information (gives guide to system generation) part no. 92077-90006. |
| I | WLOAD WCS I/O Utility Routine (on primary disc). |
| J | MPARA Paraphraser (on primary disc). |
| K | PROM Code Generator function in WLOAD (on primary disc). |
| L | MACRO/1000 Reference Manual, part no. 92059-90001. |
| M | Edit/1000 Users Guide, part no. 92074-90001. |

# SECTION 4
## MICROINSTRUCTION FORMATS

# MICROINSTRUCTION FORMATS

SECTION 4

Since microprogramming involves the micromachine's interpretation of the microinstruction word, it is essential that the microprogrammer understand the binary structure of the microinstruction and how the paraphraser is used to automatically microassemble the microinstruction into the proper format.

In this section you will find the following information:

- The microinstruction word types.
- The 32-bit microinstruction field divisions of each word type.
- The definitions and uses for all microorders.

Additional information that you will need to know is covered in Section 7 and Section 8.

## 4-1. MICROINSTRUCTION BINARY FORMAT

The HP 1000 A700 computer microinstruction word is made up of fields where each field has a particular definition. A field may contain no more than one microorder. A microorder causes the micromachine to carry out one or several machine operations. The definition of the number of bits in each field and the names of the fields is called the microinstruction format. Figure 4-1 shows the binary structure of the microinstruction word types.

The binary format gives the order of bits of the microinstruction after it has been assembled by the paraphraser microassembler language. The microprogrammer does not have to know how many bits are in each field but he does have to know the definitions and the operations of the microorders contained in the fields. This is because the paraphraser microassembler correctly formats the microinstructions from your source program which can be written in a "format free" style.

## 4-2. DEFINITION OF WORD TYPES

The HP 1000 A700 computer has a microword of 32 bits and six different Word Types. These word types have subsets (or special word types) that impose certain restrictions on the microorders used in them. The Word Types are distinguished from each other by the Operation (OP) Field; i.e., the micromachine will decode the microinstruction according to the contents of the OP Field.

In addition, word types 1 through 5 are interpreted as special word types 1S through 5S if the ALU field is coded with SPEC (Special). The special word types have certain subroutine operations coded in the field used for SP0/SP1 in word types 1 through 4. For the special microorders this field is labeled ALUS (ALU Special); e.g., as shown in Figure 4-1.

<table>
<tr><th>BIT</th><th>31</th><th>30</th><th>29</th><th>28</th><th>27</th><th>26</th><th>25</th><th>24</th><th>23</th><th>22</th><th>21</th><th>20</th><th>19</th><th>18</th><th>17</th><th>16</th><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>WORD<br>TYPE<br>1</td><td colspan="5">OP1</td><td colspan="4">ABUS</td><td colspan="5">SP0</td><td colspan="4">SP2</td><td colspan="4">ALU</td><td colspan="5">BBUS</td><td colspan="5">STOR</td></tr>
<tr><td>WORD<br>TYPE<br>2</td><td colspan="5">OP2</td><td colspan="4">ABUS</td><td colspan="5">SP0</td><td colspan="4">CNDX</td><td colspan="4">ALU</td><td colspan="5">BBUS</td><td colspan="5">STOR</td></tr>
<tr><td>WORD<br>TYPE<br>3</td><td colspan="4">OP3</td><td colspan="6">ADRS</td><td colspan="4">SP1</td><td colspan="4">CNDX</td><td colspan="4">ALU</td><td colspan="5">BBUS</td><td colspan="5">STOR</td></tr>
<tr><td>WORD<br>TYPE<br>4</td><td colspan="4">OP4</td><td colspan="6">ADRS</td><td colspan="4">SP1</td><td colspan="4">SP2</td><td colspan="4">ALU</td><td colspan="5">BBUS</td><td colspan="5">STOR</td></tr>
<tr><td>WORD<br>TYPE<br>5</td><td colspan="4">OP5</td><td colspan="14">ADRL (LONG BRANCH ADDRESS)</td><td colspan="4">ALU</td><td colspan="5">BBUS</td><td colspan="5">STOR</td></tr>
<tr><td>WORD<br>TYPE<br>6</td><td colspan="2">OP6</td><td colspan="16">DAT (IMMEDIATE DATA)</td><td colspan="4">ALU</td><td colspan="5">BBUS</td><td colspan="5">STOR</td></tr>
<tr><td>WORD<br>TYPE<br>1S</td><td colspan="5">OP1</td><td colspan="4">ABUS</td><td colspan="5">ALUS*</td><td colspan="4">SP2</td><td colspan="4">SPEC</td><td colspan="5">BBUS</td><td colspan="5">STOR</td></tr>
<tr><td>WORD<br>TYPE<br>2S</td><td colspan="5">OP2</td><td colspan="4">ABUS</td><td colspan="5">ALUS*</td><td colspan="4">CNDX</td><td colspan="4">SPEC</td><td colspan="5">BBUS</td><td colspan="5">STOR</td></tr>
<tr><td>WORD<br>TYPE<br>3S</td><td colspan="4">OP3</td><td colspan="6">ADRS</td><td colspan="4">ALUS*</td><td colspan="4">CNDX</td><td colspan="4">SPEC</td><td colspan="5">BBUS</td><td colspan="5">STOR</td></tr>
<tr><td>WORD<br>TYPE<br>4S</td><td colspan="4">OP4</td><td colspan="6">ADRS</td><td colspan="4">ALUS*</td><td colspan="4">SP2</td><td colspan="4">SPEC</td><td colspan="5">BBUS</td><td colspan="5">STOR</td></tr>
<tr><td>WORD<br>TYPE<br>5S</td><td colspan="4">OP5</td><td colspan="14">ADRL (LONG JUMP TABLE ADDRESS)</td><td colspan="4">SPEC</td><td colspan="5">BBUS</td><td colspan="5">STOR</td></tr>
</table>

*Special microorder in ALUS field when ALU field is coded SPEC.

8200-4

Figure 4-1. Microinstruction Word-Type Binary-Format Summary

## 4-3. WORD TYPE 1

Word Type 1 allows ALU functions to be performed with full capability in A, B, STOR and Special Fields. The A-Bus and B-Bus fields specify the registers enabled onto the corresponding bus. These registers are to be operated on by the ALU as specified in the ALU field. The resultant data is stored in the register specified in the STOR field. Special Fields zero and two (SP0 and SP2) can be used to perform additional operations.

The OP Field of this word type may contain the JTAB (Jump Table) microoder which is the instruction to jump to the entry point "look up" table. The "look up" table provides the destination address where the subroutine to be executed begins. The subroutine is specified in the ALU field.

The OP Field may contain microorder RTN (Return) which is the unconditional return from a subroutine to an address on top of the microinstruction stack. A NOP (No Operation) in the OP Field will cause the next sequential microorder to be executed.

Word Type 1 microorders are the following:

| **FIELD** | **MICROORDERS** |
|---|---|
| OP | NOP, JTAB, RTN |
| A-BUS | Any A-Bus source. |
| SP0 | Any SP0 microorder (except STOR). |
| SP2 | Any SP2 microorder (jump modifiers, CT30 and CT74, cannot be used). |
| ALU | Any ALU microorder. |
| B-BUS | Any B-Bus source. |
| STOR | Any STOR destination. |

## 4-4. WORD TYPE 2

Word Type 2 is equivalent to Word Type 1 except that the SP2 field is replaced by the CONDITION CODE (CNDX) Field. The OP field determines the conditional operation to be performed if the condition is true or false as specified.

The OP Field microorder selects the address for continuation after the specified condition is checked: RTNT (Return True) and RTNF (Return False) cause a return to the address at the top of the microroutine stack if the condition in the CNDX field is true or false, respectively. SP0T (SP0 True) and SP0F (SP0 False) causes the microorder in the SP0 field to be executed only when the condition specified is true or false, respectively. Following this operation, the next sequential microorder will be executed.

Word Type 2 microorders are the following:

| **FIELD** | **MICROORDERS** |
|---|---|
| OP | RTNT, RTNF, SP0T, SP0F |
| A-BUS | Any A-Bus source. |
| SP0 | Any SP0 microorder (except shifts and LDQ cannot be used with SP0T or SP0F). |
| CNDX | Any CNDX field condition. |
| ALU | Any ALU function (Special ALU operations can not be used with SP0T or SP0F). |
| B-BUS | Any B-BUS source. |
| STOR | Any STOR destination. |

## 4-5. WORD TYPE 3

Word Type 3 is similar to Word Type 2 but it is used for conditional branching where the destination address (given in the Address Field) is within the current 64 word block of micromemory. If the current microinstruction is in the last location of a 64 word block (ends in 3F hex), then the branch, if it is to occur, will be to the next 64 word block instead of the current one. Since the A-BUS Field is not available in Word Type 3, the A-BUS defaults to the accumulator (ACC or R4 of the register file. The SP1 Field is the only special field available which is a subset of the SP0 Field.

In the OP Field, JMPT (Jump True), JMPF (Jump False), JSBT (Jump Sub True), and JSBF (Jump Sub False) jump to the destination address if the condition in the CNDX field is true or false, as appropriate. For the "Jump Sub" microorders, the jump is to a subroutine at the destination address and the current address +1 is pushed onto the microsubroutine stack and the stack pointer is incremented.

Word Type 3 microorders are the following:

| **FIELD** | **MICROORDER** |
|---|---|
| OP | JMPT, JMPF, JSBT, JSBF |
| ADDRESS | Destination address in current 64 word block. |
| SP1 | Any SP1 microorder. |
| CNDX | Any CNDX Field condition. |
| ALU | Any ALU function. |
| B-BUS | Any B-BUS source. |
| STOR | Any STOR destination. |

## 4-6. WORD TYPE 4

Word Type 4 is similar to Word Type 3 except that the branch is made unconditionally. This allows the use of the SP2 Field along with the SP1 Field. As in Word Type 3, if the current microinstruction is in the last location of a 64 word block, then the branch will be to the next 64 word block. The A-BUS defaults to the accumulator (ACC or R4 of the register file).

The OP Field may contain either JMP (Jump) or JSB (Jump Sub) for jumping to the destination address or to a subroutine address at the destination address, respectively.

Word Type 4 microorders are the following:

| **FIELD** | **MICROORDERS** |
|---|---|
| OP | JMP, JSB |
| ADDRESS | Destination address in current 64 word block. |
| SP2 | Any SP2 microorder. |
| SP1 | Any SP1 microorder. |
| ALU | Any ALU function. |
| B-BUS | Any B-BUS source. |
| STOR | Any STOR destination. |

## 4-7. WORD TYPE 5

Word Type 5 is used to perform an unconditional branch to any location in the 16k-word control store. The A-BUS Field defaults to the accumulator (ACC or R4 of the register file).

The OP Field microoders may be either JMPL (Jump Long) and JSBL (Jump Sub Long) where JMPL is a jump to the destination address, and JSBL is to the subroutine at the destination address. In both, a jump modify can be specified by coding SPEC in the ALU field which will cause the lower four bits of the destination address to be replaced by the lower four bits of the Instruction Register (Register CT).

Word Type 5 microorders are the following:

| **FIELD** | **MICROORDERS** |
|---|---|
| OP | JMPL, JSBL |
| ADDRESS | Destination address anywhere in the 16k-word control store. |
| ALU | Any ALU function (SPEC will cause a jump modify operation equivalent to CT30). |
| B-BUS | Any B-BUS source. |
| STOR | Any STOR destination. |

## 4-8. WORD TYPE 6

Word Type 6 is used for enabling immediate data onto the A-BUS. The OP Field for this word is IMM (Immediate Data).

Its microorders are the following:

| **FIELD** | **MICROODERS** |
|---|---|
| OP | IMM |
| IMM | Immediate data to be placed on the A-BUS. |
| ALU | Any ALU function except SPEC. |
| B-BUS | Any B-BUS source. |
| STOR | Any STOR destination. |

The Word Types and their fields are summarized in Table 4-1.

Table 4-1. Summary of HP 1000 A700 Computer Word Types

| TYPE | FIELD1 | FIELD2 | FIELD3 | FIELD4 | FIELD5 | FIELD6 | FIELD7 |
|---|---|---|---|---|---|---|---|
| 1 | OP1 | ABUS | SP0 | SP2 | ALU | BBUS | STOR |
| 2 | OP2 | ABUS | SP0 | CNDX | ALU | BBUS | STOR |
| 3 | OP3 | ADRS | SP1 | CNDX | ALU | BBUS | STOR |
| 4 | OP4 | ADRS | SP1 | SP2 | ALU | BBUS | STOR |
| 5 | OP5 | ADRL | | | ALU | BBUS | STOR |
| 6 | OP6 | DAT | | | ALU | BBUS | STOR |
| 1S | OP1 | ABUS | ALUS | SP2 | SPEC | BBUS | STOR |
| 2S | OP2 | ABUS | ALUS | CNDX | SPEC | BBUS | STOR |
| 3S | OP3 | ADRS | ALUS | CNDX | SPEC | BBUS | STOR |
| 4S | OP4 | ADRS | ALUS | SP2 | SPEC | BBUS | STOR |
| 5S | OP5 | ADRL* | | | SPEC* | BBUS | STOR |

*Go to microinstruction table for microorder long branch jump (lower four bits of destination address replaced by bits 3-0 of CT).

## 4-9. ARITHMETIC DATA PATHS

The micromachine is based on a three-address architecture. The microinstruction can specify an A-Bus operand and a B-Bus operand. The resultant Y-Bus data is stored into a location specified in the STOR field. The A-BUS operand can be either a register in the register file specified in the A-Bus Field or Immediate Data from the microinstruction. The B-Bus operand is specified in the B-Bus Field and can come from the register file or from other dedicated registers in the processor. The A and B Buses are operated on by the ALU. ALU output data (F-Bus) is passed to the Y-Bus, with or without shifting, and stored into a processor register or written to main memory as specified in the STOR Field. A data path external to the ALU is used for byte manipulation and ASG or SRG instructions.

## 4-10. ALU FUNCTIONS

The ALU functions are divided into two categories:

1. Standard ALU operations which are coded in the ALU field and can be combined with a shift in the SP0 or SP1 field.

2. Special ALU operations performed when SPEC is in the ALU field. The SP0 or SP1 field becomes the ALUS field that is used to indicate which Special operation to perform.

## 4-11. STANDARD ALU FUNCTIONS

The standard ALU functions can be divided into two types: Arithmetic and Logical.

The arithmetic operations are true two's complement add or subtract functions. They are always performed with either a carry or borrow. For add operations the carry-in normally defaults to 0, but can be forced to a 1 with a special microorder (FCIN). For subtract operations, the borrow is the complement of the "carry-in." "Carry-in" normally defaults to 1 for subtract, but can be forced to 0 with a special microorder (FCIN). For all arithmetic operations CF (Carry Flag) and ALOV (ALU Overflow) will be updated with the ALU results.

The logical operations are performed as a bit-by-bit logical function on the A-Bus and B-Bus. Since logical operations will not generate a carry or overflow, CF and ALOV are always cleared at the end of the microcycle except for the microorder ZERO. ZERO, which forces the output of the ALU to all zeros, will disable the update of the following conditions during the current cycle: CF, ALOV, SF (Shift Flag), YZ (Y-Bus zeros), and Y15 (Y-Bus bit 15), and B15 (B-Bus bit 15).

## 4-12. SPECIAL ALU FUNCTIONS

When SPEC is in the ALU field, two types of ALU operations can be performed: Internal ALU Specials (arithmetic) and External ALU Specials (logical). The Special ALU operation is coded in the ALUS (ALU Special) field.

The Internal ALU Specials are a group of operations provided by the ALU which can be used for multiply, divide, and floating point algorithms. Each function may include an arithmetic operation and a shift. The conditions CF, ALOV, YZ, and SF may be used to represent different conditions for ALU Specials than for Standard arithmetic functions.

The External ALU Specials are performed external to the standard ALU. (The standard ALU executes all the ALU operations described in paragraph 4-11.) The external ALU Specials include byte swapping and masking, a four-bit left rotate and bit manipulation used for ASG and SRG emulation. All these operations are logical operations, so CF and ALOV are cleared. The four-bit left rotate (RL4) will not affect the Shift Flag (SF).

For a detailed description of these functions, see the microorder definitions for ALUS in Table 4-2.

Table 4-2. Microorder Definitions

**MICRO-ORDER** — **DEFINITION**

**WORD TYPE 1: OP FIELD**

JTAB

Meaning: A Jump to subroutine will occur. The Entry Point Look-Up Tables provide the lower 8 bits of the destination address which is vectored into the microaddress space between 100-1FF (hexadecimal). The current address +1 is pushed onto the stack and the stack pointer is incremented. Normally, JTAB will only be used to begin execution of a microroutine after a fetch microorder has been executed so that the instruction will be loaded into T and CT registers. Since JTAB will also initiate the execution of MRG instructions, it should be coded as follows for correct operation:

OP1/JTAB SP0/NOP SP2/IP ABUS/B ALU/ADAC BBUS/T STOR/CWRB

(The CWRB in the STOR Field may not actually occur). JTAB will always cause the B-Bus to be read as if it is an MRG instruction; i.e., it will resolve the MRG address. JTAB will always clear the double word bit (DW) and the temporary interrupt disable (TDI). Since the ALU function is ADAC and there is no carry-in, CF and ALOV will be cleared at the end of the cycle. The MA register is loaded (with the resolved MRG address) on every JTAB.

JTAB forces the SP0, A and STORE fields to what would otherwise be coded on the first microinstruction of an MRG instruction:

| INSTRUCTION | CT15-11 | SP0 | A | STOR |
|---|---|---|---|---|
| AD* | X100X | RDB | CAB | NOP |
| AND | X0010 | RDB | CAB | NOP |
| CP* | X101X | RDB | CAB | NOP |
| IOR | X0110 | RDB | CAB | NOP |
| ISZ | X0111 | RDB | CAB | NOP |
| JMP,I | 10101 | RDB | CAB | NOP |
| JMP,D | 00101 | FCHB | CAB | NOP |
| JSB,I | 10011 | RDB | CAB | NOP |
| JSB,D | 00011 | NOP | CAB | NOP |
| LD* | X110X | RDB | CAB | NOP |
| ST* | X111X | NOP | CAB | CWRB |
| XOR | X0100 | RDB | CAB | NOP |
| Non-MRG | X000X | NOP | CAB | NOP |

Because these microorders may be forced, microorders FCHB (SP0) and RDB (SP0) can not be coded in the JTAB microinstruction. A jump modifier is not allowed. The RDB and CWRB forced by JTAB use the code map unless the address is on the base page.

NOP

Meaning: No operation affecting the microcode flow will take place; i.e., the next sequential microorder will be executed.

Usage: This is a default microoder where the OP field is blank.

RTN

Meaning: An unconditional return from subroutine will occur. The next microaddress will come from the top of the microsubroutine stack and the stack pointer will be decremented. The return can be to any address in micromemory. No jump modifier is allowed.

Table 4-2. Microorder Definitions (Continued)

| MICRO-ORDER | DEFINITION |
|---|---|
| **WORD TYPE 2: OP FIELD** | |
| RTNT | If the condition specified in the CNDX field is True, then this is the same as the RTN OP. Otherwise, the next sequential operation is executed. |
| RTNF | If the condition specified in the CNDX Field is False then this is the same as the RTN OP. Otherwise, the next sequential instruction is executed. |
| SP0F | The microorder in the SP0 field will be executed only if CNDX is False. The next sequential microorder will be executed. SPEC cannot be coded in the ALU Field. The SP0 Field must not contain a shift function or LDQ |
| SP0T | The microorder in the SP0 field is executed only if CNDX is True. The next sequential microorder will be executed. SPEC cannot be coded in the ALU field. The SP0 Field must not contain a shift function or LDQ. |
| **WORD TYPE 3: OP FIELD** | |
| JMPT | Jump to target address within current 64 word block only if CNDX is True. Otherwise, the next sequential microorder is executed. If the current address is in the last location of a 64 word block, then the jump will be to the next 64 word block. |
| JMPF | Same as JMPT except that the jump will occur only if CNDX is False. |
| JSBT | Jump to subroutine at target address within current 64 word block only if CNDX is True. If CNDX is True, the current address + 1 is pushed onto the microsubroutine stack and the stack pointer is incremented. If the current address is the last location of a 64 word block, then the jump will be to the next 64 word block. If CNDX is not True, then the next sequential microinstruction will be executed. |
| JSBF | Same as JSBT except that the jump to subroutine occurs only if CNDX is False and the next sequential microorder is executed if CNDX is True. |
| **WORD TYPE 4: OP FIELD** | |
| JMP | Word Type 4. Unconditional Jump to target address within the current 64 word block. If the current address is the last location of a 64 word block then the jump will be to the next 64 word block. Jump modifiers may be used. |
| JSB | Unconditional jump to subroutine at the target address within the current 64 word block. The current address + 1 is pushed onto the microsubroutine stack and the stack pointer will be incremented. If the current address is the last location of a 64 word block, then the jump will be to the next 64 word block. Jump modifiers may be used. |
| **WORD TYPE 5: OP FIELD** | |
| JMPL | Unconditional jump to target address anywhere in the microaddress space. Coding SPEC in the ALU field performs a modified jump. This causes the lower 4 bits of the target address to be replaced with bits 3-0 of CT and all ones to be put on the Y bus. |
| JSBL | Unconditional jump to subroutine at target address anywhere in the microaddress space. The current address +1 is pushed onto the microsubroutine stack and the stack pointer is incremented. A jump modify may be performed by coding SPEC in the ALU field. This causes the lower 4 bits of the target address to be replaced with bits 3-0 of the CT register and all ones to be put on the Y-bus. |

Table 4-2. Microorder Definitions (Continued)

| MICRO-ORDER | DEFINITION |
|---|---|
| **WORD TYPE 6: OP FIELD** | |
| IMM | Immediate data used as the A-Bus operand. The next sequential instruction is executed. |
| **WORD TYPES 1 AND 2: A-BUS FIELD** | |
| The A Field specifies the A-Bus operand and is available in Word Types 1 and 2. For Word Types 3, 4 and 5 the A-Bus defaults to the accumulator (R04). | |
| (R00) A | Macro A-Register, R00 of the register file. |
| (R01) B | Macro B-Register, R01 of the register file. |
| X (R02) | Macro X-Register, R02 of the register file. |
| Y (R03) | Macro Y-Register, R03 of the register file. |
| ACC (R04) | Accumulator, R04 of the register file. A-Bus defaults to ACC in WORD Types 3, 4 and 5. |
| HP1 (R05) | R05 of the register file. Reserved for use as the Return register. |
| HP2 (R06) | R06 of register file. Reserved (do not use). |
| USR (R07) | R07 of the register file. Reserved for the user. That is, no HP microcode will use this register. |
| S0 - S7<br>($R10_8$ - $R17_8$) | General-purpose registers in the register file. |
| **WORD TYPES 1, 2, 3, AND 4: SP0 FIELD AND SP1 FIELD** | |
| The SP1 field contains a subset of the SP0 Field. The SP0 Field is available in Word Types 1 and 2, and the SP1 Field is available in Word Types 3 and 4. The microorders below are available in both the SP1 Field and the SP0 Field. | |
| ACF | Perform the ALU operation specified using the carry flag as the ALU carry in. This microorder has no effect if the ALU field contains a logical operation. |
| AL1 | Arithmetic Left Shift. If the double-word bit (DW) is set, then a double-word shift will be performed with the ALU output as the most significant word and the Q-register as the least significant word. This special cannot be used with the SP0T or SF0F microorders in the OP field. |
| AR1 | This special cannot be used with SP0T or SP0F in the OP Field. Same as LL1 except that an arithmetic Right Shift is performed. |
| CLE | Clear the macro Extend register at the end of the microcycle. |
| FCIN | Force the ALU carry-in to 1 for add operations (ADDC, ADBC, CMBC, ADAC, CMAC). Force the ALU carry-in (BORROW-) to 0 for subtract operations (SBAC and SBBC). This microorder has no effect if the ALU field contains a logical operation. |
| IP | Increment the P-register at the end of the microcycle. |

Table 4-2. Microorder Definitions (Continued)

| MICRO-ORDER | DEFINITION |
|---|---|
| **WORD TYPES 1, 2, 3, AND 4: SP0 FIELD AND SP1 FIELD (Continued)** | |
| IN | Increment the N (Index) register at the end of the microcycle. |
| LDQ | Store the ALU output (preshifter) in the Q-register. LDQ used with SP0T or SP0F in the OP field. |
| LL1 | Logical shift of the ALU output Left 1 position. If the double-word bit (DW) is set, then a double-word shift will be performed with the ALU output as the most significant word and the Q-register as the least significant word. This special can not be used with the SP0T or SP0F microorders in the OP field. |
| LR1 | Same as LL1 except that a logical Right Shift is performed. This special cannot be used with SP0T or SP0F in the OP Field. |
| NOP | No operation. |
| RDB | Perform a memory read using the B-Bus as the memory address. The data read is returned to the T-register. Uses Data Map if base register is enabled. |
| RL1 | Same as LL1 except that a Left Rotate is performed. This special cannot be used with SP0T or SP0F in the OP Field. |
| RR1 | Same as LL1 except that a Right Rotate is performed. This special cannot be used with SP0T or SP0F in the OP Field. |
| RDP | Perform a memory read using the address in the P-register. The data read is returned to the T-register. Uses Data Map if base register is enabled. |
| STE | Set the macro Extend register at the end of the microcycle. |
| **WORD TYPES 1 AND 2: SP0 FIELD** | |
| BFB | Same as a RDB except that a line called "RNI" is asserted on the backplane which goes to external devices to indicate that an instruction is on the backplane (typically an I/O instruction). If the base register is enabled, BFB uses the Code Map. |
| CK2 | Enables the optional floating point processor to clock at twice its normal rate. |
| CLO | Clear the integer overflow register at the end of the microcycle. |
| ENOE | Enable the macro register E, and the integer overflow register to be set from the results of the current microcycle.<br>Add operations: (ADDC, ADAC, ADBC, CMAC, CMBC), E is set if the carry out of the ALU is set.<br>Subtract operations: (SBBC, SBAC), E is set if the carry out of the ALU is clear; that is, E is set if there is a borrow.<br>Logical operations: the setting of E is undefined. The integer overflow is set if the overflow of the ALU is set. Otherwise, E and the integer overflow register are unchanged. The setting of E and integer overflow occurs at the end of the microcycle. |

Table 4-2. Microorder Definitions (Continued)

**MICRO-ORDER** — **DEFINITION**

**WORD TYPES 1 AND 2: SPO FIELD (Continued)**

FCHB — Read with address on the B-Bus to perform an instruction fetch. Data returned on FCHB will be loaded into both the T-register and the CT-register (Instruction Register). A fetch microorder must be executed before returning to the JTAB subroutine to perform all the necessary "housekeeping" functions in order to execute the next assembly instruction. If an interrupt is pending, FCHB will be inhibited and INTP set. There can only be one fetch (FCHB or FCHP) per macroinstruction. If the base register is enabled, FCHB uses the Code Map.

IFCH — Performs an interrupt instruction fetch. IAK (interrupt acknowledge) will be asserted on the backplane. A memory read will occur where the address and map selection are driven by the highest priority interrupting device. P-register is not altered. The data returned will be loaded into the T-register. This microorder is normally executed only in the interrupt service microroutine. This microoder will not work properly if the value of the B bus is 0 or 1. The central interrupt latch (CIL) will be loaded with the address. (CIL is an external register referenced by SRIN.) If the base register is enabled, IFCH uses the Data Map.

LDBR — Load the Base Register with the data on the Y-Bus at the end of the microcycle.

STOR — Perform the STOR operation specifed only if the condition specified (CNDX) is met. STOR can only be used with SPOT or SPOF in the OP field.

STO — Set the integer overflow register at the end of the microcycle.

**WORD TYPES 1, 2, 3, AND 4: ALUS FIELD WHERE ALU FIELD CONTAINS SPEC**

The following specials are available in the ALUS field (Word Types 1,2,3, and 4) when the ALU field contains SPEC. F is used in the microorder descriptions to indicate the ALU output or F-Bus. Q should not be used in the B-field along with the ALU Specials. In this case, Q is an implied operand.

ASG — Performs a transformation of the B-bus operand and passes it to the Y-bus. This instruction is normally used to change the ASG instruction in the instruction register CT into a more usable form.

The ASG data transformation is the following:

| Y-Bus gets | B-Bus | Y-Bus gets | B-Bus |
|---|---|---|---|
| 0 | 8 | 8 | X |
| 1 | SKP | 9 | X |
| 2 | SE | 10 | X |
| 3 | 9 | 11 | 11 |
| 4 | 0 | 12 | X |
| 5 | 1 | 13 | X |
| 6 | 2 | 14 | X |
| 7 | 3 | 15 | X |

NOTES: In the table above, X is undefined, SE indicates the state the E bit should be at the end of the instruction (excluding the effects of INA). SKP is a logical "1" if the ASG instruction should skip (again excluding the effects of INA).

Table 4-2. Microorder Definitions (Continued)

| MICRO-<br>ORDER | DEFINITION |
|---|---|
| **WORD TYPES 1, 2, 3, AND 4: ALUS FIELD WHERE ALU FIELD CONTAINS SPEC (Continued)** | |
| DIV | Two's complement divide step:<br>F = B+A+Cin, if SCFF = 0;<br>F = B−A−1+Cin, if SCFF = 1;<br>where SCFF = "sign compare flip flop", Cin=SCFF, and F = F-Bus.<br>The sign compare flip flop is internal to the ALU and is driven onto the YZ output line during the DIV operation. The sign compare flip flop is updated only when DNRM, DIV, or DIV1 is performed. When one of these microorders is specified, the SCFF will be updated at the end of the cycle with "A15 exclusive NOR F15"; i.e., SCFF (next cycle) = 1 if the signs of A and the F-bus are the same, SCFF (next cycle) = 0 if the signs of A and the F-bus are different.<br>A double word, logical left shift is performed on the F-Bus and Q-register with "A15 exclusive NOR F15" shifted into the least significant bit position.<br>The processor conditions are the following:<br>CF = carry out;<br>ALOV = overflow;<br>YZ = sign compare flip flop;<br>SF = A15 exclusive NOR F15. |
| DIV1 | First two's complement divide step. This is the same as DNRM with the exception that the value shifted into the least significant bit position is "A15 exclusive OR F15." |
| DNRM | Double-Length Normalize Step: F-Bus = B+Cin, where Cin=0<br>A double word, logical left shift is performed on the F-bus and Q where a zero is shifted into the least significant bit position.<br>Processor conditions are the following:<br>CF = F15 exclusive OR F14;<br>ALOV = F14 exclusive OR F13;<br>YZ = Q register and F-Bus (before shifting) are all zeros;<br>SF = A15 exclusive OR F15 (F-bus bit 15).<br>DNRM will cause the sign compare flip flop (SCFF) to be loaded at the end of the cycle with "A15 exclusive NOR F15." |
| RL4 | Rotate the B-Bus operand left 4 places and pass to the Y-Bus. This microorder is used in emulation of SRG instructions. CF and ALOV are cleared. The shift flag is not updated by this microorder. If the SP2 field contains LWE or LWF, they are ignored. |

Table 4-2. Microorder Definitions (Continued)

| MICRO-ORDER | DEFINITION |
|---|---|
| **WORD TYPES 1, 2, 3, AND 4: ALUS FIELD WHERE ALU FIELD CONTAINS SPEC (Continued)** | |
| SM2C | Sign magnitude to 2's complement conversion:<br>F = B+Cin, if B15 = 0;<br>F = (−B)+Cin, if B15 = 1;<br>where Cin=B15, and F = F-bus.<br>There is no shift but bit 15 of the Y-Bus is forced as follows:<br>Y15= B15 exclusive OR F15.<br>The processor conditions are the following:<br>CF = carry out;<br>ALOV = overflow;<br>YZ = B15;<br>SF = not updated. |
| SNRM | Single length normalize step: F = B+Cin, where Cin=1, F = F-bus.<br>F is not shifted, but a logical left shift is performed on Q with a zero shifted into the least significant bit position.<br>The processor conditions are the following:<br>CF = Q15 exclusive OR Q14 (before shifting);<br>ALOV = Q14 exclusive OR Q13 (before shifting);<br>YZ = Q register all zeros before shifting;<br>SF = Q15 (before shifting). |
| SRG | Performs a transformation of the B-Bus operand and passes it to the Y-Bus. This microorder is normally used to change the SRG instruction in the CT-register (instruction register) a more usable form. |

| **The Transformation If B-Bus Bit 10 = 0** | | | | **The Transformation If B-Bus Bit 10 = 1** | | | |
|---|---|---|---|---|---|---|---|
| **Y-Bus** | **B-Bus** | **Y-Bus** | **B-Bus** | **Y-Bus** | **B-Bus** | **Y-Bus** | **B-Bus** |
| 0 | 8 | 8=0 | — | 0 | 0 | 8=0 | — |
| 1 | 9 | 9=0 | — | 1 | 1 | 9=0 | — |
| 2 | 6 | 10=0 | — | 2 | 2 | 10=0 | — |
| 3 | 7 | 11 | 11 | 3 | 3 | 11 | 11 |
| 4 | 0 | 12=0 | — | 4 | 8 | 12=0 | — |
| 5 | 1 | 13=0 | — | 5 | 6 and (8 or 9) | 13=0 | — |
| 6 | 2 | 14 | 5 | 6 | 7 | 14 | 6 |
| 7 | 4 | 15 | 3 | 7=0 | — | 15 | 9 |

NOTES: CF and ALOV are cleared.

| MICRO-ORDER | DEFINITION |
|---|---|
| SWAP | Swap bytes of B-bus field operand and pass to Y-bus. CF and ALOV are cleared. |
| SWZU | Swap bytes of B-bus field operand, zero the upper byte and pass to Y-bus. CF and ALOV are cleared. |
| SWZY | Swap bytes of B-bus field operand, zero the lower byte and pass to Y-bus. CF and ALOV are cleared. |

Table 4-2. Microorder Definitions (Continued)

| MICRO-ORDER | DEFINITION |
|---|---|
| **WORD TYPES 1, 2, 3, AND 4: ALUS FIELD WHERE ALU FIELD CONTAINS SPEC (Continued)** | |
| TMPY | Two's complement multiply step. This is the same as UMPY with one exception: the value shifted into the most significant bit position is:<br>F15 exclusive OR OVR,<br>where F15 is bit 15 of F-Bus and OVR=ALU overflow from the current cycle. |
| TMLC | Last cycle of two's complement multiply:<br>F = B+Cin, if Q0=0;<br>F = B−A−1+Cin, if Q0=1<br>where Cin=Q0, F = F-bus.<br>A double word, logical right shift is performed on F-bus and Q-register with the following shifted into the most significant bit position:<br>F15 exclusive OR OVR,<br>where F15 = bit 15 of F-bus, and OVR=ALU overflow of the current cycle.<br>The processor conditions are the following:<br>CF = carry out;<br>ALOV= overflow;<br>YZ= Q0;<br>SF = Q0. |
| UMPY | Unsigned multiply step:<br>F = B+Cin, if Q0=0;<br>F = A+B+Cin, if Q0=1;<br>where Cin=0, F = B-bus (ALU output).<br>A double-word logical right shift is performed with F and Q, with the carry flag from the current cycle being shifted into the most significant bit position. The processor conditions are updated as follows:<br>CF = carry out;<br>ALOV = overflow;<br>YZ = Q0;<br>SF = Q0. |
| ZLY | Zero the lower byte of the B-bus field operand and pass to Y-bus. CF and ALOV are cleared. |
| ZUY | Zero the upper byte of the B-bus field operand and pass to Y-bus. CF and ALOV are cleared. |
| **WORD TYPES 1 AND 4: SP2 FIELD** | |
| This field specifies a number of special functions in the processor. | |
| CLF | Clear general-purpose flag at the end of current microcycle. |
| CMDW | Complement the Double Word bit at the end of the current microcycle. For standard ALU functions the double-word bit indicates that a shift specified will be a double-word shift. |
| CMID | Complement the sense of the "temporary interrupt system disable" at the end of the current microcycle. |

Table 4-2. Microorder Definitions (Continued)

| MICRO-ORDER | DEFINITION |
|---|---|
| **WORD TYPES 1 AND 4: SP2 FIELD (Continued)** | |
| CT30 | Replace bits 3-0 of jump address with bits 3-0 of the CT-register. This is a jump modify microorder and can be used only in Word Type 4. |
| CT74 | Replace bits 3-0 of jump address with bits 7-4 of the CT-register. This is a jump modify microorder and can be used only in Word Type 4. |
| DCT | Decrement the general-purpose counter at the end of the current microcycle. |
| DN | Decrement the N register at the end of the cycle. If DN and IN are both coded, the N register will decrement. |
| FCHP | Read with address in P to perform an instruction fetch. Data returned on FCHP will be loaded into both T and CT (Instruction Register). A fetch microorder must be executed before returning to the JTAB microinstruction to perform all necessary "housekeeping" functions in order to execute the next assembly instruction. If an interrupt is pending, FCHP will be inhibited and INTF will be set. If the base register is enabled, FCHP uses the Code Map. |
| IP | Increment the P-register at the end of the microcycle. If IP is coded in both the SP2 and SP0 fields the P register will only increment once. |
| LWE | Link with E; that is, E is shifted into the data word and the bit shifted out is loaded into E. LWE can only be coded with a shift coded in the SP0 field. |
| LWF | Link with F. F (the general purpose flag) is shifted into the data word and the bit shifted out is loaded into F. LWF can only be used with a shift in the SP0 or SP1 field. |
| NOP | No special operation is performed. |
| RDIO | Perform an I/O handshake on the backplane where an I/O card will will send an operand to the CPU. The returned operand is loaded into register T. Before executing this microorder, the microcode must check that IORQ is asserted on backplane (see microorder IORQ). |
| RDPC | Perform a memory read using the address in the P-register. If the base register is enabled, RDPC uses the Code Map. |
| STF | Set general-purpose flag at the end of the current microcycle. |
| WRIO | Perform an I/O handshake on the backplane where the processor card will supply data to be received by the I/O card from the Y-Bus. For this microorder to function correctly, the microcode must check that IORQ is asserted. |
| **WORD TYPES 2 AND 3: CONDITION (CNDX) FIELD** | |
| The conditions are tested by the conditional operations. They are updated at the end of the microcycle, such that they refer to conditions generated during the previous microcycle or registers loaded at the end of the previous microcycle. | |
| B15 | Bit 15 of the B-Bus was a logic "1" during the previous micro-cycle. This condition is not valid if Q was the B-Bus operand during the previous cycle. The B15 condition is not updated when zero is in the ALU field. |

Table 4-2. Microorder Definitions (Continued)

| MICRO-ORDER | DEFINITION |
|---|---|
| **WORD TYPES 2 AND 3: CONDITION (CNDX) FIELD (Continued)** | |
| ALOV | The overflow output of the ALU set at the end of the previous microcycle. For standard ALU functions, ALOV is a true two's complement overflow but it may indicate a different condition for SPEC functions. The ALOV condition is not updated when ZERO is in the ALU field. |
| CF | The carry output set at the end of the previous microcycle. It is a true two's complement carry for add operations, and borrow− (complement of borrow) for subtract operations. CF may indicate different conditions for SPEC functions. The CF condition is not updated when the ALU field contains ZERO. |
| CTZ | All bits of the CT-register are zero. This condition is not valid during the cycle immediately following a load of the CT register. CT will be decremented at the end of the microcycle when CTZ is in the CNDX field. |
| CTZ4 | All the lower 4 bits of the CT-register are zero. The CTZ4 condition will not be valid during the cycle immediately following a load of the CT-register. CT will be decremented at the end of the microcycle when CTZ4 is in the CNDX field. |
| E | The macro E (Extend) register is set. |
| F | General-purpose flag is set. Note that the state of F may be altered by JTAB in the OP field or STF, CLF, LWF in the SP2 field. |
| INTP | A processor interrupt is pending. This condition lags the interrupt condition by one cycle. Example:<br>**CONDITION** — **RESULT**<br>Parity Error Occurs; — *Set Parity Error Flag<br>If INTP Go To 1 Cycle; — *INTP Not True Yet<br>If INTP Go To 2 Cycles; — *Now INTP Is True |
| INTF | The Interrupt Flip-Flop is set meaning that the last fetch was ignored because a qualified interrupt request was pending. This indicates that the interrupt service routine must be called before the next JTAB microorder is executed. INTF is cleared in the interrupt service routine by any STOR to the interrupt status register (IST). |
| IORQ | IORQ (I/O request) was asserted on the backplane at the the previous microcycle. |
| MPEN | Memory protect was enabled during the previous cycle. |
| O | The integer overflow register is set. |
| PON | This microorder allows the firmware to distinguish a "force to zero" on powerup from a "jump to nonexistent micromemory" or a "microcode time out." It indicates that the processor power-on signal was asserted during the previous cycle. |
| SF | Shift flag is set. When a shift operation is executed, the shift flag is loaded with the bit shifted out. For further explanation see the paragraph in this section on Shifting Functions, and the ALU Special microorder definitions (ALUS Field). The SF condition is not updated when ZERO is in the ALU field. |
| Y15 | Bit 15 of the Y-Bus was a logic "1" at the end of the previous microcycle. |
| YZ | The Y-Bus was all zeros at the end of the previous cycle. When an ALU special microorder is executed, YZ may indicate a different condition. (See description of SPEC microorders in the ALUS field.) The YZ condition is not updated when the ALU field contains ZERO. |

Table 4-2. Microorder Definitions (Continued)

| MICRO-ORDER | DEFINITION |
|---|---|
| **WORD TYPES 1 - 6: ALU FIELD** | |
| This field describes the function to be performed on the A-Bus operand and the B-Bus operand. | |
| AND | Logically "AND" A with B. CF and ALOV are cleared. |
| ADAC | Add (two's complement) the A-Bus operand to Cin: A+Cin; where Cin defaults to 0. |
| ADBC | Add (two's complement) Cin to B-Bus operand: B+Cin; where Cin defaults to 0. |
| ADDC | Add (two's complement) A-Bus operand and B-Bus operand: A+B+Cin; where Cin defaults to 0. |
| CAND | Logically "AND" B with the one's complement of A. CF and ALOV are cleared. |
| CMAC | Add the one's complement of the A-Bus operand to Cin: (NOT A)+Cin; where Cin defaults to 0. |
| CMBC | Add the one's complement of B to Cin: (NOT B)+Cin; where Cin defaults to 0. |
| IOR | Logically "inclusive OR" A with B. CF and ALOV are cleared. |
| NAND | Logically "NAND" A with B. CF and ALOV are cleared. |
| INOR | Logically "inclusive NOR" A with B. CF and ALOV are cleared. |
| SBAC | Subtract (two's complement) A-Bus operand from B-Bus operand: B−A−1+Cin; where Cin= −borrow and Cin defaults to 1. |
| SBBC | Subtract (two's complement) B-Bus operand from A-Bus operand: A−B−1+Cin; where Cin= −borrow and Cin to 1. |
| SPEC | Perform special operation specified in ALUS field. SPEC cannot be used with SP0T, SP0F or IMM in the OP field, and cannot use Q in the B Field. With Word Type 5, SPEC will cause a jump modify operation equivalent to CT30, and all ones will be put on the Y-Bus. |
| XNOR | Logically "exclusive NOR" A with B. CF and ALOV are cleared. |
| XOR | Logically "exclusive OR" A with B. CF and ALOV are cleared. |
| ZERO | Forces the ALU output to all zeros. The following conditions are not updated: Y15, YZ, CF, ALOV, SF, and B15. |
| **WORD TYPES 1 - 6: B-BUS FIELD** | |
| The B-Bus Field is used to specify the B-Bus operand and is in all Word Types. | |
| CAB | Conditionally enable to the B-Bus either the macro A- or B-register:<br>A-register if bit 11 of the CT register = 0; B-Register if bit 11 of the CT register = 1. |
| CT | Counter, also used as an instruction register. After a fetch, the instruction is returned to both T and CT. |
| CXY | Conditionally enable to the B-Bus either the macro X- or Y-register:<br>X-Register if bit 3 of the CT-register = 0; Y-Register if bit 3 of the CT-register = 1. |

Table 4-2. Microorder Definitions (Continued)

| MICRO-ORDER | DEFINITION |
|---|---|
| **WORD TYPES 1 - 6: B-BUS FIELD (Continued)** | |
| FA | The Fetch Address register which generally holds the address of the last macroinstruction fetch. |
| GRIN | Enable to the B-Bus the register in the general register file indexed by N. |
| IST | Interrupt status register. |
| N | The Index Register (bits 0-3) for indirect addressing of the privileged and general register files and the external registers (bits 4-7, not used; bits 9-15, status information). |
| P | The P-register which generally holds the macro program counter. |
| PRIN | Register in the privileged register file indexed by N. |
| Q | Q-register (internal to the ALU). The Q-register becomes the B-Bus operand for ALU operations, although it is not actually enabled onto the B-Bus. Thus, Q cannot be used in the B-Field for External ALU Specials or memory operations that use the B-Bus as the address (BFB, RDB, FCHB). Also, specifying Q in the B-Field will cause the B15 condition to be undefined during the following cycle. |
| MA | The last memory address register which generally contains the address of the most recent access to main memory. MA is loaded by any memory reference microorder and also when JTAB is asserted. |
| MAP | Enable Map Register addressed by MPAR to the B-Bus. MPAR is an external register. Bit 15 is read protect, bit 14 is write protect, and bits 0 to 9 have the physical page number. |
| MEMR | Enable the Memory Control Register onto the B-Bus. |
| A (R00) | Macro A-register (R00). |
| B (R01) | Macro B-Register (R01). |
| X (R02) | Macro X-Register (R02). |
| Y (R03) | Macro Y-Register (R03). |
| ACC (R04) | Register file accumulator (R04). |
| HP1 (R05) | R05 of the register file. Reserved for use by HP as the Return register. |
| HP2 (R06) | R06 of the register file. Reserved for use by HP. |
| USR (R07) | R07 of the register file. Reserved for the user. That is, no HP microcode will use this register. |
| S0 - S7<br>($R10_8$ - $R17_8$) | General-Purpose Registers |
| SR | Enable the processor switch register to the B-Bus. An open switch is read as a "0" and a closed switch as a "1". |
| SRIN | Enable to the B-Bus the Special register indexed by N, bits 0-3.<br>Registers: 0=MPAR; 1=PEL1; 3=PEL2; 3=CIL; 4-15, HP reserved. |

Table 4-2. Microorder Definitions (Continued)

| MICRO-ORDER | DEFINITION |
|---|---|
| **WORD TYPES 1 - 6: B-BUS FIELD (Continued)** | |
| T | The T-register to which data from the backplane is stored after memory and I/O reads. If data from a previous memory or I/O read has not yet been returned to the T-register, then the processor will freeze. If the previous read was A/B addressed, the A- or B-Register will be enabled onto the B-Bus. |
| **WORD TYPES 1 - 6: STOR FIELD** | |
| The STOR Field is used to specify the destination register (or main memory) into which the Y-Bus will be loaded at the end of the microcycle. Register updating occurs only at the end of the cycle. | |
| CAB | Store the Y-Bus conditionally in either the macro A- or B-register:<br>A-register if bit 11 of the CT-register = 0; B-Register if bit 11 of the CT-register = 1. |
| CT | Counter Register (instruction register). |
| CXY | Store the Y-Bus conditionally in either the macro X- or Y-register:<br>X-Register if bit 3 of the CT-register = 0; Y-Register if bit 3 of the CT-register = 1 |
| CWRB | Conditional write (used for indirect storing). Perform a memory write if B15 = 0; otherwise perform a memory read. The read or write will be to the main memory location pointed to by the address on the B-Bus. Data for write is from the Y-Bus. |
| GRIN | The register in the general register file indexed by N. |
| IST | Store the Y-Bus to the Interrupt Status Register. |
| LR | Enable and latch the Y-Bus into the processor status LEDs. Storing a "0" in LR will light the corresponding LED, and storing a "1" will turn off the LED. |
| N | The Index Register which is used for indirect addressing of the GRIN and PRIN register files and the external SRIN registers. |
| NOP | No store is performed. |
| MAP | Store Y-Bus into MAP Register addressed by MPAR. Bit 15: read protect. Bit 14: execute protect. Bits 0-13: physical page no. |
| MEMR | Store the Y-Bus into the Memory Contol Register. |
| P | The P-register which generally holds the macro program counter. |
| PRIN | The register in the privileged register file indexed by N. |
| A (R00) | Macro A-Register |
| B (R01) | Macro B-Register |
| X (R02) | Macro X-Register |
| Y (R03) | Macro Y-Register |
| ACC (R04) | Register file accumulator. |

Table 4-2. Microorder Definitions (Continued)

| MICRO-ORDER | DEFINITION |
|---|---|
| **WORD TYPES 1 - 6: STOR FIELD (Continued)** | |
| HP1 (R05) | R5 of the register file. Reserved for use as the Return register. |
| HP2 (R06) | R6 of the register file. Reserved for use by HP. |
| USR (R07 | R7 of the register file. Reserved for user. That is, no HP microcode will use this register. |
| S0 - S7<br>($R10_8$ - $R17_8$) | General-purpose registers. |
| SRIN | Store Y-Bus into the special register indexed by N, bits 0-3.<br>Registers: 0 = MPAR; 1 = PEL1; 2 = PEL2; 3 = CIL; 4-15, HP reserved. |
| WRP | Write to memory at the address specified by P. Data comes from the Y-Bus. |
| WRB | Write to main memory at the address on the B-Bus. Data to write comes from the Y-Bus |

## 4-13. SHIFT FUNCTIONS

There are three categories of shifts which can be executed in the microcode:

1. Single Word, single bit shifts;
2. Double Word, single bit shifts;
3. Special function shifts (SPEC in the ALU field).

The double Word bit (DW) is used to differentiate between single and double word shifts. The Shift Flag (SF) is used to hold the bit which is shifted out. Note that the Shift Flag will be updated ONLY for shift functions. The Shift Flag is normally updated for all shift functions except RL4 (4 bit left rotate). The Shift Flag is not updated when the ALU field contains ZERO.

Single word, single-bit shifts are indicated by microorders in the SP0 or SP1 field when the double-word bit is cleared and the ALU field does not contain SPEC. The shifts are arithmetic, logical or rotational either right or left. The output of the ALU (F-Bus) is shifted and then enabled to the Y-Bus. SF will be updated as follows:

SF = F-Bus bit 0 for all right shifts
SF = F-Bus bit 15 for left rotate and logical shift
SF = F-Bus bit 14 for left arithmetic shifts

A link with either E (Extend register) or F (general-purpose flag) can be specified in the SP0 field. In this case, E or F will be shifted into the F-Bus data word and the bit shifted out (same as the bit loaded into SF) will be loaded into either E or F (see Figure 4-2 for Single-Word Single-Bit Shifts).

Double-word single-bit shifts are indicated by microorders in the SP0 or SP1 field when the double-word bit is set and the ALU field does not contain SPEC. The shifts are arithmetic, logical, or rotational — either right or left. The output of the ALU (F-Bus) is shifted together with the Q register where the F-Bus is the most significant word and Q is the least significant word. The F-Bus, after shifting will be enabled to the Y-Bus and the shifted data from Q will be loaded back into Q. SF will be updated as follows:

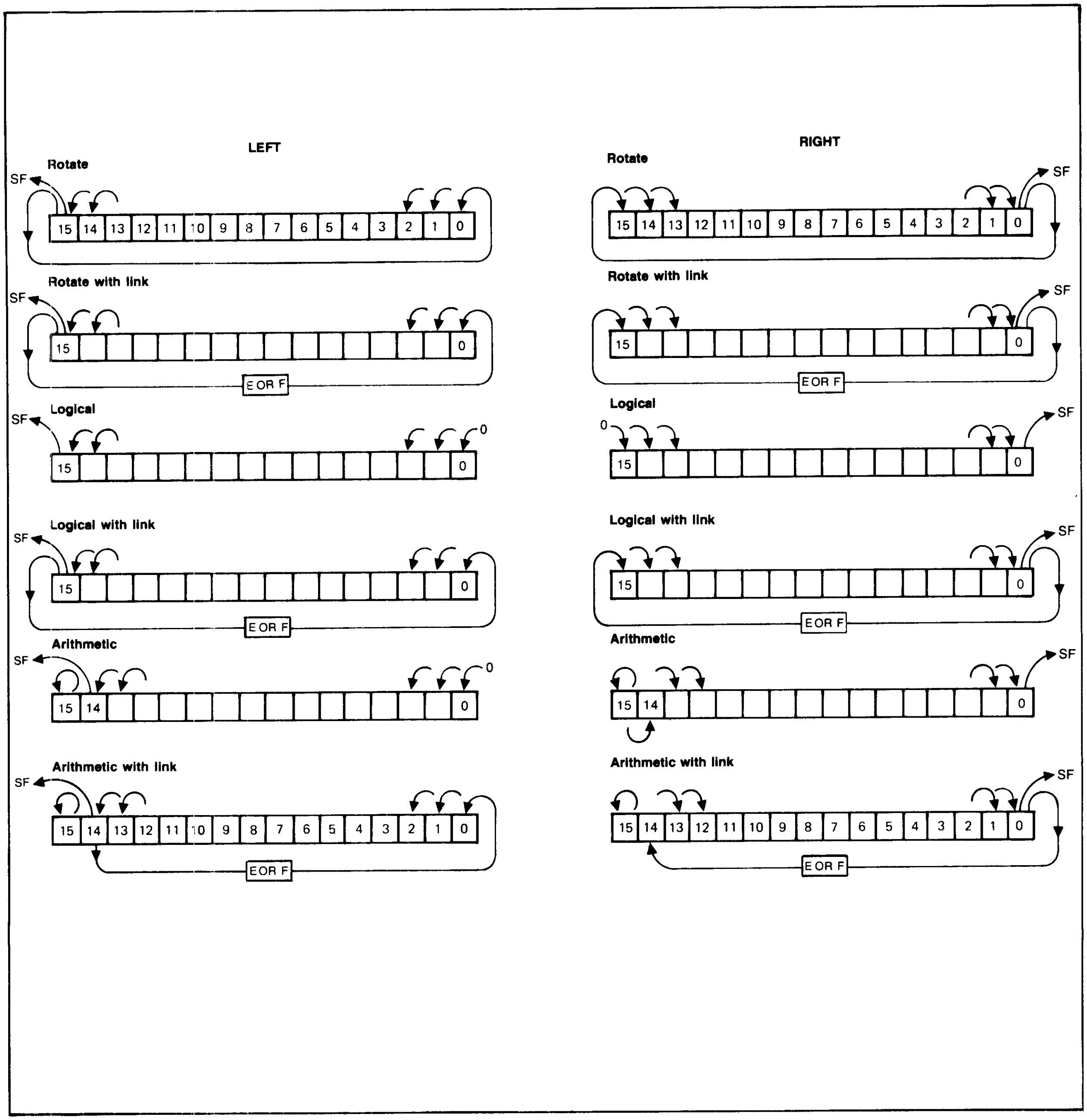
SF = Q bit 0 for all right shifts
SF = F-Bus bit 15 for left rotate and logical shifts
SF = F-Bus bit 14 for left arithmetic shifts

A link with either E (Extend register) or F (general-purpose flag) can be specified in the S0 field. In this case, E or F will be shifted into the F-Bus/Q data word and the bit shifted out (same as the bit loaded into SF) will be loaded into either E or F (see Figure 4-3 for Double-Word Single-Bit Left Shifts, and Figure 4-4 for Double-Word Single-Bit Right Shifts).

There are seven Internal ALU Special functions and one External ALU Special shift function which perform shifts. These shifts may be either single- or double-word shifts by definition and their function is not affected by the state of the double-word bit. For the ALU Specials, if a link with E or F is specified, then the bit shifted out (same as the bit loaded into SF) will be loaded into E or F, but E and F will NOT be shifted in; that is, the normal shifting operation will occur. For the External ALU Special shift (RL4) any link specified will be ignored. These shift operations and the Shift Flag updating are explained in detail under the SP0 and SP1 fields in the microorder definitions. (The SP1 field is a subset of the SP0 field so they are given under "Word Types 1 - 4 for the SP0 Field and SP1 Field.")

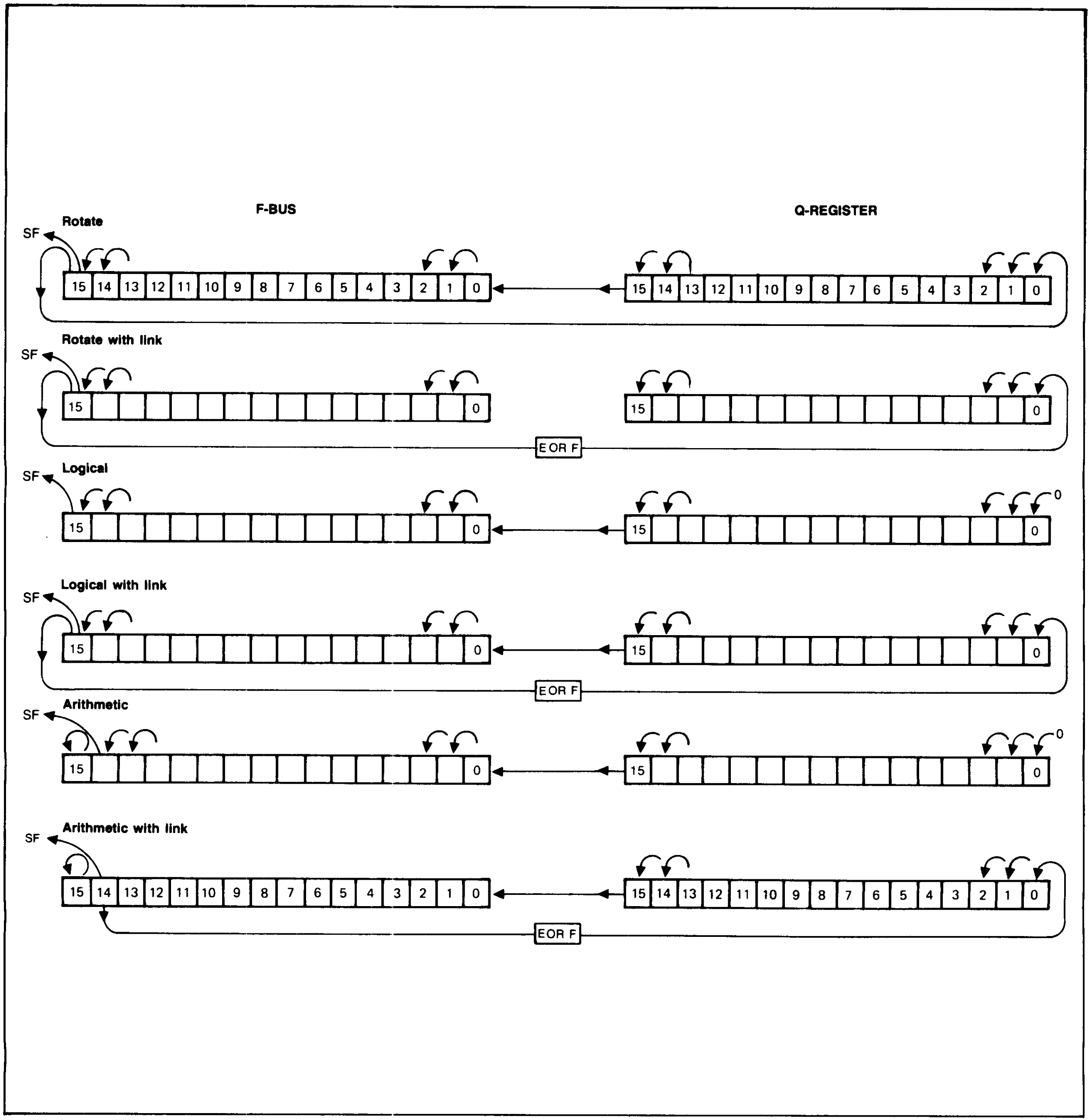
## 4-14. MICROORDER DEFINITIONS

Microorder Definitions are given in Table 4-2, and a Summary of Microorders which shows in tabular form the microorders contained in each field is provided in Appendix B.



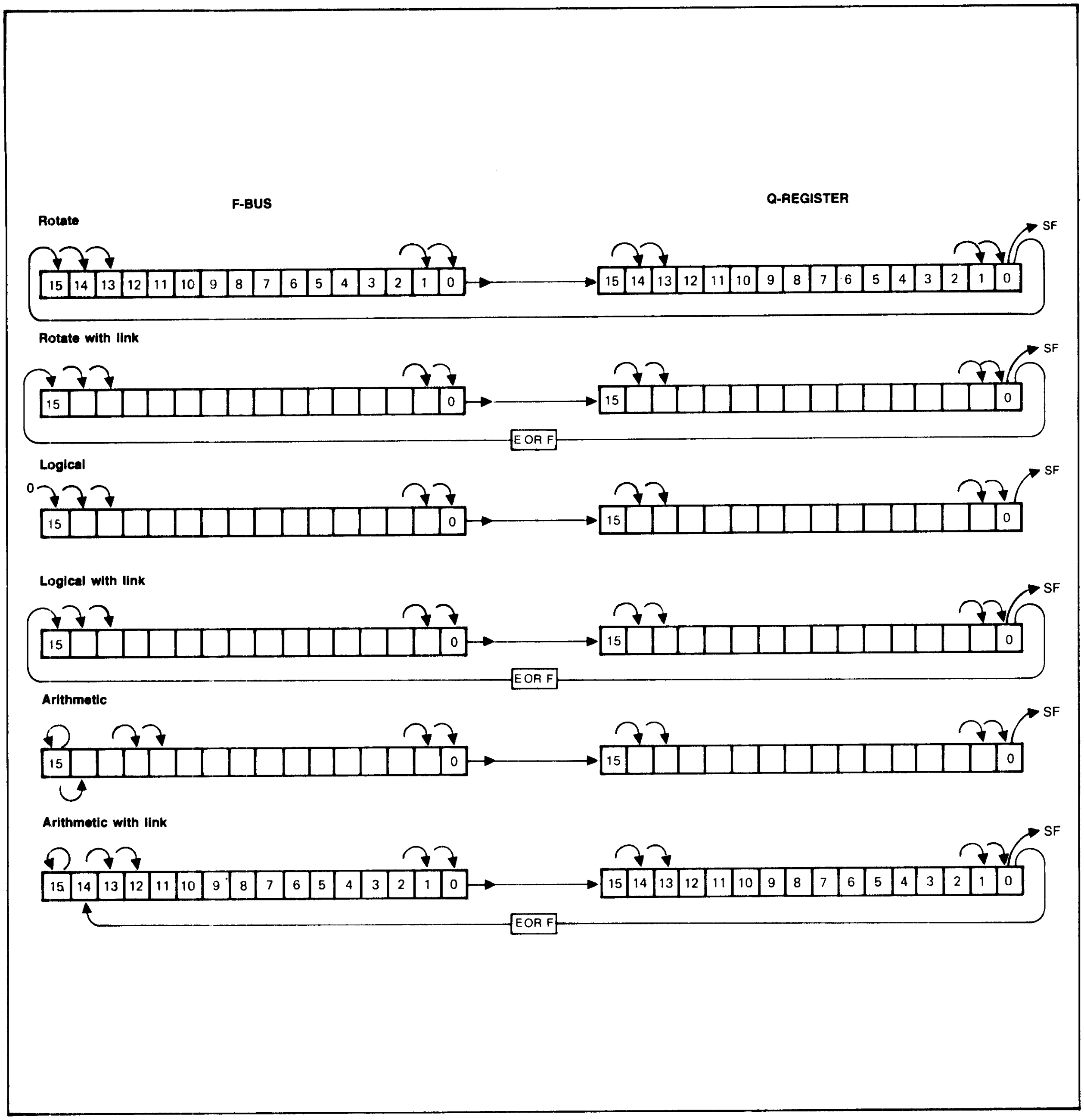
8200-9

Figure 4-2. Single Word, Single Bit Shifts



8200-11

Figure 4-3. Double-Word, Single-Bit Left Shifts



8200-10

Figure 4-4. Double-Word, Single-Bit Right Shifts

# 4-15. MICROCODE RESTRICTIONS AND CONSIDERATIONS

## 4-16. MEMORY AND I/O MICROORDERS

Do not code more than one memory or I/O microorder in the same microinstruction. These microorders are: RDP, RDPC, RDB, FCHP, FCHB, IFCH, BFB, WRB, WRP, CWRB, RDIO, and WRIO.

## 4-17. FETCH MICROORDERS

The fetch microorders are FCHB and FCHP. Every microroutine to emulate an instruction must contain exactly one fetch microorder. A store to the interrupt status register is *not* allowed in the microinstruction in which the fetch occurs. A memory or I/O microorder, as defined above, or any modification of CT or IST are not allowed between the fetch microinstruction and the return JTAB loop. The fetch microorder will not cause CT to be loaded at the end of the current microcycle but may cause it to be loaded at the end of the following microcycle. Therefore, the microinstruction following the fetch may reference CT if it does not contain the T microorder. No following microorder should reference CT.

## 4-18. BFB and IFCH MICROORDERS

BFB and IFCH must not reference the A- or B-Registers since they will not function in this manner. This means that either A/B addressability must be off or that the B-Bus must have a value greater than 1 during the cycle when BFB or IFCH is executed. Therefore, the user microcode must resolve this problem.

## 4-19. RDIO AND WRIO MICROORDERS

Microorder T should not be coded in the B field of a microinstruction for operation in the same microroutine as an RDIO or WRIO microorder.

## 4-20. MEMORY READS

All memory reads must be terminated with T in the B field. Note that T is contained in the B field of the JTAB microinstruction to terminate the FCHB or FCHP at the end of the JTAB microroutine.

## 4-21. MAP REFERENCES

Do not code the MAP microorder in the STOR field.

## 4-22. BASE REGISTER AND DATA AND CODE MAPS

When the base register is disabled, all memory references use the map specified in bits 0-4 of MEMR, which may contain any map number. When the base register is enabled, it is assumed that bits 0-4 of MEMR contain an even map number which is used as the Data Map. The corresponding Code Map will be the Data Map with bit 0 forced to a "1". The following memory references use the Data Map: RDP, RDB, WRB, WRP, and CWRB. The following memory references use the Code Map, except when they reference the base page in which case they will use the Data Map: FCHB, FCHP, RDPC, and BFB. When the base register is enabled, any memory reference to the base page will have the base register added to its logical address. This will not affect addressing the A or B registers.

# SECTION 5
## TIMING CONSIDERATIONS

# TIMING CONSIDERATIONS

SECTION 5

Certain details about computer timing should be considered for microprogram applications so that you can do the following:

- Intelligently and effectively make use of computer time when you execute your microprograms.
- Synchronize microinstructions properly for the operations that you wish to perform with your microprograms.

The information you need to know about the computer's timing to effectively microprogram can be separated into three categories:

- A basic definition of the processor cycle time period.
- Conditions that can vary the speed of execution of your microprograms.
- How you estimate the time it takes for a microprogram to execute.

In the HP A700 processor, the timing as related to microprogramming is very simple since almost all microinstruction operations take place within a single clock period. The microinstruction time periods are described in this section.

## 5-1. COMPUTER TIMING

As defined in Section 2 of this manual, microinstructions are executed in the micromachine during "microcycles." One microcycle is the time interval required to completely execute one normal microinstruction. The length of one microcycle is 250 nanoseconds.

A normal microinstruction is defined here as one that does not result in a processor clock "freeze."

## 5-2. MEMORY and I/O ACCESS

The processor has the capability of buffering (or latching) the data and address of just one memory or I/O access while a previous *memory access* is being executed. While an *I/O access* is being executed there is no buffering (or latching) of data and address. Different lengths of time are required for these operations as follows:

a. Memory accesses typically take two processor cycles (microcycles). If memory refresh occurs at the same time, the access can extend to four microcycles.

b. I/O accesses take three microcycles each.

Any DMA activity from I/O cards has priority over the processor operations; thus, any memory or I/O access can be delayed by the DMA time.

## 5-3. PROCESSOR CLOCK FREEZE

A freeze of the processor clock is a waiting condition where the current microinstruction will not be executed until the freeze is terminated. There are four conditions associated with memory or I/O access that will cause a processor clock freeze. They are the following:

a. If the microorder T is in the B-Bus Field of the current microinstruction while the T-register has not yet been loaded with data from the previous memory or I/O read.

b. If a memory or I/O access has been requested by a microorder, and the memory and I/O access logic is busy (that is, it is still holding an address or data from a previous request which has not yet gone out on the backplane).

c. The MAP microorder in either the B-Bus or STOR field is executed while the maps are in use by the memory controller for a memory cycle. Since the map registers are required by the processor for the new microorder, a freeze will occur. This freeze lasts a maximum of one clock cycle.

d. The FCHP and FCHB microorders cause a freeze if any read or write is in progress. This ensures that any memory protect violation is detected at the end of the instruction that caused it.

A processor clock freeze is transparent to the microcode. It will delay the transition of the processor clock at the middle of the clock cycle for an integral number of cycles. When a freeze occurs, none of the microorders in the current (frozen) microcycle will be executed until the freeze condition is no longer in effect. A processor clock freeze does not effect the system clock or any other cards.

Maximum performance is obtained by minimizing the number of times freezes occur in any microprogram since each freeze adds one or more clock periods of 250 nanoseconds to the total run time. In microprogramming consider the effect of memory read operations and T register usage.

All memory read operations use the T-register and take at least two processor clock cycles. Thus, the T-register is not available in the cycle after a memory read. Freezes can be avoided by not programming T-register reference microorders immediately following memory read microorders. An example of microinstruction execution times of memory accesses through the T-register comparing the time with a freeze and without a freeze follows:

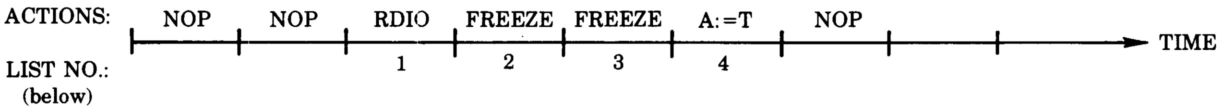
```
S0:=55;         *
S1:=a,RDP;      *start memory read
S2:=t;          *freeze time = 1.0 usec

S1:=a,:RDP;     *start memory read
S0:=55          *wait for memory read
S2:=t           *no freeze, time = 0.750 usec
```

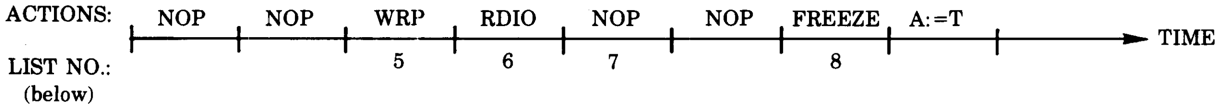
Avoidance of a freeze is beneficial if you can effectively use the cycle occurring in between the memory reference microorder and the T microorder.

Microinstruction execution including I/O accesses, and reading and writing to memory is illustrated in the following examples where the explanation and backplane action is given in a list below the examples:

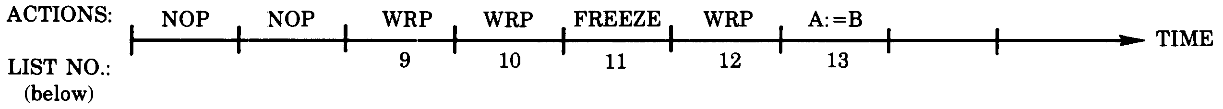
PROGRAM: NOP; NOP; RDIO; A:=T; NOP



PROGRAM: NOP; NOP; WRP; RDIO; NOP; NOP; A:=T



PROGRAM: NOP; NOP; WRP; WRP; WRP; A:=B



ACTIONS LIST:

**First Example**

1. I/O Read Started.
2. Freeze, T not ready.
3. Freeze.
4. Executed

**Second Example**

5. Write Started.
6. I/O Read Buffered.
7. I/O Read Started.
8. Freeze Since I/O Read Not Done.

**Third Example**

9. Write Started.
10. Write Buffered.
11. Freeze: Buffer In use
 2nd Write Started.
12. No Freeze, 3rd Write Buffered.
13. No Freeze, 3rd Write Started.

## 5-4. DATA TRANSFERS

A complete memory access to main memory occurs within two 250 nanosecond processor cycles. Thus, the fastest data transfer rate, taking into account memory refresh cycles, is 2.0 Mwords per second or 4.0 Mbytes per second. An example of a data transfer program is the cross-move-words macroinstruction (e.g., dynamic mapping macroinstruction MW01) which moves words at memory speed.

# SECTION 6
# MAPPING TO USER'S MICROPROGRAMMING AREA

# MAPPING TO USER'S MICROPROGRAMMING AREA

SECTION 6

In order to have operational flexibility when using your microprogramming facilities, you must have an understanding of the methods used to branch from main memory to control store and then back to your program in main memory when your microprogram is completed. This section provides information that will enable you to do the following:

- Understand the control store mapping scheme.
- Link to the user's microprogramming area from your Assembly Language, FORTRAN, or PASCAL program.
- Pass parameters to your microprogram.
- Understand control store branch address modification (using some of the available microorders).
- Return from control store (making a "normal" exit).
- Pass parameters from your microprogram back to your main memory program.

For this discussion on mapping it will be assumed that your microprograms have already been prepared (using the Paraphraser) and placed in some facility of control store (e.g., WCS or PCS). Section 7 describes how to assign starting addresses to your microprograms.

Part III (Microprogramming Support Software and Hardware) of this manual describes methods used to get microprograms into control store. One method is the creating and installing of permanent microprograms. Another method is to use WCS and the WCS related microprogramming support software (ID.41,WLOAD) to load microprograms into control store and swap (or overlay) them with other microprograms.

## 6-1. CONTROL STORE MAPPING METHOD

The Microprogram Control Section is always in control of the computer, and the base-set microroutines carry out the steps for reading the instructions (and data) from main memory. In this operation, all instructions are placed in the instruction register and decoded. (The JTAB macroinstructions decoding loop is described in Section 2 of this manual.

Data can be considered as "parameters" which can be loaded into the desired and appropriate registers by your microprogram to later perform certain operations. Parameter passing will be described later in this section.

The process of decoding the macroinstruction bits determines which control store address (i.e., which microprogram) is called by the macroinstruction (Assembly language instruction) fetched from main memory.

The decoding process, or mapping method, described below is at the level you will need for normal user microprogramming. Also the instructions for mapping to particular control store entry points are defined.

## 6-2. SOFTWARE ENTRY POINTS

Recall that the control store map in Figure 2-3 shows all 16k words of control store, the boundary addresses of each memory area, and whether or not the memory area has "software entry points" available to the user. The software entry points are opcodes reserved for the user that branch to microaddresses where your microprograms begin.

The hardware/firmware combination in the Microprogram Control Section is the facility that defines the control store software entry points. As described in Section 2, certain areas of control store may be used for HP microprograms and/or microprogrammed computer enhancements. Thus, you should know the contents of any area of control store before you put your microprograms there.

You should restrict your use of software entry point instruction codes to those set aside for entrance into the user's microprogramming area. The instruction codes for most sofware entry points (excluding the base set and HP reserved areas of control store) are defined in the following paragraph, and the instruction codes for entrance into the user's area (the primary subject of this section) are identified.

Once the user instruction has entered control store, the user's microprogram may branch to any control store location. Again, the use of discretion is implied since the areas shown in Figure 2-3 which are reserved for HP microprograms and/or microprogrammed accessories may be filled with microprograms.

NOTE

> The contents and placement of microroutines of the A700 base set are reserved by HP and they are subject to change without notice. Therefore, branching into the HP base set to use existing microroutines (such as INDREAD) is not recommended. However, you may copy these routines into your microprogramming areas for your own use.

## 6-3. USER INSTRUCTION GROUP (UIG)

For the purposes of mapping to the "user" areas, the processor base set has a reserved block of binary codes called the User Instruction Group (UIG). The user's areas that have these codes are designated as UIG Software Entry Points. Entry to other control store areas requires an extra branch after reaching control store.

All opcode blocks which are accessible through the UIG instructions are shown in Table 6-1. This table is arranged in UIG instruction (binary code) order.

Table 6-1. Control Store UIG Software Entry Point Assignments

| RANGE OF UIG INSTRUCTION MAIN MEMORY VALUES USED (OCTAL) | CONTROL MEMORY ENTRY POINT RANGE (HEX) | USE |
|---|---|---|
| 105000 – 105137 | — | HP Reserved (Base Set,FPP) |
| 105140 – 105177 | 0900 – 090F | HP Reserved (Base Set) |
| 105200 – 105237 | — | HP Reserved (Base Set,FPP) |
| 105240 – 105257 | — | HP Reserved (VMA) |
| 105260 – 105277 | 2800 – 280F | HP Reserved |
| 105300 – 105317 | — | HP Reserved (Base Set) |
| 105320 – 105337 | — | HP Reserved (SIS) |
| 105340 – 105357 | 0D00 – 0D0F | HP Reserved |
| 105360 – 105377 | 0F00 – 0F0F | HP Reserved |
| 101 (or 105) 400 – 417 | 0800 – 080F | HP Reserved |
| 101 (or 105) 420 – 437 | 0A00 – 0A0F | HP Reserved |
| 101 (or 105) 440 – 457 | 0C00 – 0C0F | HP Reserved |
| 101 (or 105) 460 – 477 | 0E00 – 0E0F | HP Reserved |
| 101 (or 105) 500 – 517 | 3000 – 300F | User Reserved |
| 101 (or 105) 520 – 537 | 3200 – 320F | User Reserved |
| 101 (or 105) 540 – 557 | 3400 – 340F | User Reserved |
| 101 (or 105) 560 – 577 | 3600 – 360F | User Reserved |
| 101 (or 105) 600 – 617 | 2000 – 200F | HP/User Reserved |
| 101 (or 105) 620 – 637 | 2200 – 220F | HP/User Reserved |
| 101 (or 105) 640 – 657 | 2400 – 240F | HP/User Reserved |
| 101 (or 105) 660 – 677 | 2600 – 260F | HP/User Reserved |
| 101 (or 105) 700 – 737 | — | HP Reserved (DMS) |
| 101 (or 105) 740 – 777 | — | HP Reserved (EIG) |

NOTES:

1. HP Base Set is the HP A700 firmware.
2. Floating Point firmware is located in the HP 12156A FPP control store.
3. VMA = Virtual Memory Access; SIS = Scientific Instruction Set; DMS = Dynamic Mapping System; EIG = Extended Instruction Set.
4. HP Reserved areas should not be entered by the user, HP/User Reserved areas can contain user microcode but HP may use this control store area for future firmware packages; and User Reserved areas will never be used by HP.

The UIG instructions permit you to link Assembly language routines to your microprograms. The key to UIG is the upper byte (most significant bits) of the calling code which must have the following format:

octal 105xxx (bit 11 of the CT=1)

or:

octal 101xxx (bit 11 of the CT=0)

where xxx equals values to be defined in the following paragraphs.

**6-4. UIG OPCODE BLOCKS.** The UIG instructions are decoded in blocks of 16 opcodes that must be further decoded by the microcode.

The control store opcode block selection is determined by the value of bits 8 through 4 in the CT Register (still part of the coded UIG instruction). In general, a secondary index (composed of bits 3 through 0) directly determines which address in the first 16 locations of the selected opcode block will be used for entry.

Bit 11 in the third octal digit (105xxx or 101xxx) of the UIG instruction in CT can be used as an indicator in your microprogram by the CAB microorder. For example, the CAB microorder in the STORE and BBUS fields will test bit 11 of the CT register for a 0 or 1 to select either the A- or B-register, respectively.

The value of bits 8 through 4 of the UIG instruction in the CT register is not directly translatable into the complete address of the control store opcode block but these bits determine the address of branches in the control store base set JTAB Jump Table, which in turn direct a branch to the opcode block.

**6-5. USER AREA UIG BLOCKS.** The control store hexadecimal addresses ranging from 3000 through 3FFF are directly accessible and comprise the primary user's microprogramming area. The 1k-word modules of control store from 2000 (hex) through 2FFF (hex) may be used by user microcode but these modules may be claimed for future HP firmware enhancement packages. The available user control store is shown in the control store map of Figure 2-3.

The blocks of opcodes 101500-101577 (octal) or 105500-105577 (octal) are dedicated to the user. The 101600-101677 (octal) or 105600-105677 (octal) opcodes may be used by the user, with the reservation that future HP firmware packages might reclaim these opcodes. For the same reasons, the 1k control store modules from 2000 (hex) to 0x2FFF (hex) may be used by the user with the reservation that they may be used by HP for optional enhancements to the A700 instruction set.

Each opcode block has 16 possible control store software entry points provided by the UIG instruction secondary index (UIG instruction bit 3 through 0 combination). The secondary index directly determines which control store address (of the first 16 locations in the selected module) will be loaded into the CT Register. The range of values for UIG instructions you should use to access the respective control store addresses are summarized below. Since each opcode block can be entered at 16 different locations, there are 128 direct entry points into the recommended user's microprogramming area.

The UIG instruction (binary codes) blocks you can use are the following:

101500 through 101577 or 105500 through 105577, and

101600 through 101677 or 105600 through 105677 which may be used by HP for future A700 optional instruction set enhancements.

## 6-6. HP RESERVED UIG BLOCKS

The opcode blocks of control store have software entry points that are not available to the user. These include the software entry points for the base set, and for HP enhancements such as the Floating Point Processor and Scientic Instruction Set microcode areas, etc.

Some of the opcodes for HP enhancements are reserved for future firmware packages so until these packages are available they may be used with discretion by the user. The opcodes that are currently defined instructions are in the base set for each instruction group. Those that are HP- or user-reserved are decoded in the &USER section of the base set. Refer to Appendix E if you require more information about the base set.

To avoid access to the HP reserved area do not use the following opcode blocks of UIG instruction (binary codes) for main memory to control store linking:

105000 through 105337, and

101400 through 101477 or 105400 through 105477, and

101700 through 101777 or 105700 through 105777

The EIG (Extended Instruction Group) and DMS (Dynamic Mapping System) macroinstructions are in the UIG. The base set listing (Appendix E) shows the DMS microprogram has two "table goto" lines to fully decode the 64 separate opcodes that are included in this group. (All 64 of these opcodes are "used" by this group, although some of the opcodes map onto other opcodes or some are unused.)

The two "table goto" lines are included here for reference:

```
$origin 0x18b$                *DMS jtab entrypoint
  gototbl DMS_TBL1, stor/n;   *opcodes 10(x01)700-10(x01)717
$origin 0x18c$                *DMS jtabl entrypoint
  gototbl DMS_TBL2, stor/n;   *opcodes 10(x01)720-10(x01)737
```

These microinstructions branch outside of the JTAB jumptable, and the actual next address is the target address with bits 3-0 of the macroinstruction inserted in bits 3-0 of control store address.

## 6-7. USER'S AREA MAPPING EXAMPLE

A typical example of mapping to the user's microprogramming area through the base set using a recommended UIG instruction is discussed below. Information about the proper procedure to use in linking to main memory and for returning to main memory is also included.

Suppose that your main memory program has a UIG instruction 105602 (octal) written into a particular location designated "I." The UIG instruction can have address pointers and/or operands in main memory locations I+1, I+2, etc.

For example:

**MAIN MEMORY**

| Location | Contents |
|---|---|
| I | 105602 |
| I+1 | o |
| I+2 | o |
| o | o |
| o | o |
| o | o |

## 6-8. BRANCHING TO YOUR MICROPROGRAM

During execution, UIG instruction 105602 maps to micromemory address 3002 (hexadecimal) as follows. The previous microprogram (presumably an HP macroinstruction) performs a fetch of your UIG opcode and returns to the line after the JTAB of the JTAB loop being used. The JTAB loop is listed below for reference.

```
jtab:
{
  Normal jtab location.
  Loop until interrupt.
}
  jtab,                       *Subroutine call to macroinstruction
                              *  emulation routine.
    clf,                      *  put the flag into known state
    ip,                       *  Inc P to address after opcode.
    cwrb:=b, bbus/t;          *  Force orders that are required for
                              *    the MRG decode to begin memory
                              *    references in this cycle.
  if not intf goto jtab,      *Loop until interrupt causes
    acc:=ones;                *  a fetching to be ignored.
interrupt:                    *INTERRUPT:
  goto int_vector,            *  branch to interrupt handler
    ct:=ist;                  *  load interrupt priority code into ist
```

The previous microprogram would have returned to the "if not intf" line which checks for interrupts. If an interrupt occurred and the fetch was held off, the next microinstruction to be executed would be the "interrupt:" line and the base set would have executed the interrupt. If no interrupt occurred, then the next microinstruction to be executed would be the "jtab:" line, which begins your microprogram.

The JTAB microinstruction, in conjunction with the look-up table, would produce 1AA (hexadecimal) for the next microaddress. At this location in micromemory, the base set has a microinstruction that branches to your microprogram, as shown here:

```
$origin 0x1AA$               *opcodes 101 (or 105) 600-637
  gototbl 0x3000,            *branch to the user opcode
    stor/n;                  *initialize the n register with 0xF
```

The "gototbl 0x3000" function, which is the paraphraser representation for "op5/jmp, adrl/ 0x3000,alu/spec", branches to one of 16 microaddresses from 0x3000 to 0x300F, depending on the contents of the CT register. The lower four bits of the CT register are substituted for the lower four bits of the microaddress branched to after the GOTOTBL (CT contains the user instruction). Because the lower four bits of CT contain the value 2, the next microaddress to be executed is 0x3002.

Upon reaching the user microprogramming area (at address 0x3002), the following situation exists:

| CONDITION | REASON |
|---|---|
| CT = 105602 (octal) | Loaded by the previous fetch. |
| acc = FFFF (hexadecimal) | Loaded by the "if not intf" line. |
| p = I + 1 | Due to the "ip" in the JTAB microinstruction. |
| F = 0 | Cleared in the JTAB microinstruction. |
| n = (hexadecimal) | Due to "gototbl" line. |

At location 3002 (hex), your microprogram begins. Typically, the first 16 locations in a user module are set up with unconditional branches to the actual user microroutine but you may use the remaining fields to perform important functions such as reading the next memory address.

```
location  microinstruction          comments
0x3000      goto user0;             *entry point 0
0x3001      goto user1;             *entry point 1
0x3002      goto user2,             *my instruction!
              rdp, ip,              * begin read of next location
              a:=a-acc;             * increment the A register
0x3003      goto user3;             *
  ..          ..
  ..          ..
0x300F      goto userF;             *

0x3010     user2:                   *continue my instruction!
             call INDREAD;          *read from the memory address
             ...                    *etc.
```

From location 3010 (hex), your microprogram can continue execution.

## 6-9. RETURNING TO THE BASE SET

Once you have completed your microprogram, you must return to the base set to have it continue execution of the micromachine operations. (Remember, when the micromachine is executing your microprogram, you are in complete control.) It is your responsibility to fetch the next macroinstruction, and for this purpose, you should be sure to set the program counter to the next macroinstruction to be executed after your user-group macroinstruction.

In this example, the program counter was already set to the next macroinstruction address, so the last microinstruction to be executed could be simply the following:

```
fchp, rtn;          *return to the base set
```

## 6-10. CALLING MICROPROGRAMS

Procedures for invoking your microprograms from assembly language and high level languages are described below. The basic concepts of invoking microprograms and passing parameters should also be evident from this information.

In the RTE-A.1 environment, the best way to assign a name to a user opcode is through the RPL mechanism of Macro/1000. With this feature, your application programs can call your microprograms or software-equivalents for your microprograms, and the choice is made at the time the program is linked. Create a file with the names of your opcodes RPL'ed to the user opcodes you have chosen. The following is an example:

```
MACRO,L
      NAM MYRPL,7
      ENT MYOP,USER1,USER2,USER3
;THIS FILE CREATES THE EXTERNAL RPL'S WHICH ARE LOADED
;  WITH THE PROGRAM TO DEFINE MY USER OPCODES
;
MYOP  RPL 105500B
USER1 RPL 105501B
USER2 RPL 105502B
USER3 RPL 105503B
      END
```

After you have created a file with opcode names, you can call your microprogram from assembly language as follows:

```
MACRO,L
      NAM TEST,7
      ENT TEST
      EXT MYOP,ISC,NMBR,IBUF
TEST  NOP
      JSB MYOP      ;EXECUTE MY OPCODE
      DEF *+4       ; WHICH
      DEF ISC       ;   TAKES
      DEF NMBR      ;     THESE
      DEF IBUF      ;       PARAMETERS
      JMP TEST,I    ;
      END TEST
```

This microprogram accesses its parameters from memory locations pointed to by succeeding DEFs, and the location directly after the opcode points to the next instruction to be executed after MYOP (the .ENTR calling sequence). (See RTE-DOS relocatable library for complete details.)

If your microprogram complies with the .ENTR calling sequence as this microprogram does, you can call your microprograms directly from FORTRAN/1000 or Pascal/1000.

In FORTRAN:

```
C  Execute my microprogram
   CALL MYOP(ISC,NMBR,IBUF)
```

In Pascal:

```
   MYOP(ISC,NMBR,IBUF);   (execute my microprogram)
```

If your microprogram accepts parameters in the A, B, X, Y, E, and O registers, or does not comply with the .ENTR calling sequence, you must access your microprogram directly from assembly language. Otherwise, you must provide an assembly language interface routine to access your microprogram from high level languages. Following the .ENTR calling sequence is preferable. Good examples of HP microprograms that honor the .ENTR calling sequence are those of the Vector Instruction Set. (Note: VIS information will be furnished as an update to this manual since it is not available at the time of the first printing.)

In any case, while linking your application program, relocate the RPL file after you relocate your application program to define your microprogram entry points. Make sure your microprogram has been correctly loaded with WLOAD, and then your application program is ready to run with microcoded enhancements.

## 6-11. HANDLING INTERRUPTS

If your microprogram executes for a period longer than 25 microseconds, you should make your microprogram interruptable so that interrupt response time on the processor can be acceptable. This requires that you be able to either save the current state of your microprogram in memory locations or in registers, or simply restart your microprogram from scratch. Either way, you must periodically check the "intp" condition, and if it is true, perform a fetch ("fchp" or "fchb") of the fetch address of your user-group opcode. (The fetch will be held off by the interrupt condition, and the base set will handle the interrupt condition after you do a return back to the JTAB loop.)

## 6-12. EXAMPLE OF CHECKING INTERRUPT CONDITION

As an example, at a convenient point in your microprogram you might include the following microinstructions:

```
if intp goto quit;        *better check for interrupts now!
 ....                     *
quit:                     *have to quit now due to interrupts
  p:=fa,                  * reload the program counter
    fchb, rtn;            * fetch and return to the base set
```

This procedure will work if you can return directly to the base set using the return address on the subroutine stack. However, if you have to check interrupts within a subroutine of your own, you can jump to the location specified in the base set as INST_RESTART (at location 0xD0) and the base set will handle the interrupt. For example:

```
if intp goto quit;        *check for interrupts
  ...                     *
quit:                     *
  goto INST_RESTART p:=fa;*let the base set handle the fetch.
```

## 6-13. MICROCODE TIME-OUT

You should be aware that if your microprograms execute for longer than 10 milliseconds without servicing interrupts, your microprogram may be subjected to a microcode time-out. When this occurs, your processor will perform an Unimplemented Instruction Trap (UIT) interrupt (on YOUR user-group opcode). If you are operating within the RTE-A environment, RTE-A will abort execution of the program which called your microprogram.

## 6-14. REGISTERS RESERVED FOR THE USER

Your microprograms will probably be similar to some of the already-existing HP1000 instructions, such as FAD, or .ENTR, so be sure to look over the microprograms for these instructions. These microprograms use the registers of the machine in exactly the same way that you may use them to hold temporary values. If you need to define registers for user-defined functions, the HP base set has allocated registers for you that it will not alter. The directly accessible register USR (R07) may be used as a user-defined register, as can registers 4 through 7 of the PRIN register file.

## 6-15. MICROPROGRAMMING CONSIDERATIONS

Because the base set manages all of the functions of the processor, you must perform certain functions (and refrain from performing others) to enable the base set to continue execution after your microprogram has finished. Normally, problems with your microcode will not stop micromachine operation so that other users in an RTE environment cannot continue using the system while you are debugging your microcode. However, the following rules should be followed for allowing the base set to continue doing its job.

After the fetch and before your return observe the following:

1. Do not store to or otherwise alter the CT or IST registers.
2. Do not initiate any memory or I/O read or write.
3. Do not alter the privileged registers in the PRIN register that are used by the base set; i.e., alter only PRIN registers 4 through 7.
4. Do not initiate I/O requests without having previously established that IORQ is present. (You should generally not be executing I/O requests unless you have special hardware with which to interact or a microcoded driver.)
5. Do not execute the JTAB microorder.
6. Do not alter the HP-reserved directly accessible registers HP1,HP2.
7. Do not alter the MEMR register unless you replace it before you fetch and return.
8. If your microprogram is long, be prepared to honor interrupts.
9. Do not alter the page mapping registers (through "map" in the STOR field) unless that is a function of the microprogram.

# SECTION 7
## WRITING MICROPROGRAMS

# WRITING MICROPROGRAMS

SECTION 7

With the information in this final section of Part II you will be able to write your microprograms so that they will be accepted by the Paraphaser. If properly prepared, your microprogram will be processed (using information in Section 8) to generate micro-object code which is ready to load in WCS for execution in the computer. This section provides:

- A suggested method for preparing your microprograms.
- A description of the paraphraser labels, directives, fields, and other rules for preparation.
- Paraphraser control methods.
- Methods of making microprogram starting address assignments and making other modifications using directives.

The information in this section requires as a prerequisite, a study of the preceding sections (particularly Section 4 and 6).

## 7-1. PLANNING AND PREPARATION

Plan your microprogram essentially the same way as you would plan an Assembly language program but base the objective on the concepts described in Section 1 for microprogramming. Steps that must be taken to achieve the objective should be clear and organized. The logical sequence for the microprogram can be prepared in flowchart form for easier programming.

When preparing your microprogram, take full advantage of your system's EDIT/1000 capability. The editor provides the tools for generating the source code and storing it in a disc file. The files can be accessed later for editing and microassembling. Complete instructions for using the editor are contained in the HP EDIT/1000 Reference Manual, part no. 92074-90001.

The paraphraser program MPARA will accept RTE text files (type 4) as source files. Its output consists of two files as follows: a listing file (type 4) and a microcode file (type 5).

You can include along with the microinstructions in your program as many comments as you feel are appropriate. Comments are to help you correct the program for errors or to help you explain your program so that others can understand it at some later time. Paragraph 7-3 covers the details you will need on these subjects.

## 7-2. THE PARAPHRASER

The Paraphraser is the microcoding language of the A700 processor. It is an RTE microassembler for translating your microprogram "free format" source file into binary object code. After running the paraphraser with the source microcode program, the object microcode is stored in a file, the number of errors are printed on the output device, and optionally a listing of address labels are output.

When writing your microprogram in the free-format manner allowed by the paraphraser, it is not necessary to set up Tab spacing for fields or write a microorder for each field since the paraphraser will automatically set up and fill in the fields as required for each word type.

The program Control Statement at the beginning of the microprogram provides you with the options of producing an address label listing, and a floating field listing. These listings are described in Section 8.

The Paraphraser program is named MPARA. MPARA is written in Pascal with assembler enhancements, and it will run on an HP 1000 system in a 32k byte partition. MPARA must be loaded into a disc file of your system. Loading and using the MPARA is covered in Section 8 of this manual. MPARA allows your program to contain definitions of a combined total of about 600 address labels, data labels, and microorders.

The resulting binary object code is in a format that is recognized by the WLOAD utility program used to load the HP 12153A WCS card and to generate a PROM format for firmware to be installed on the HP 12155A PCS card.

The rules for preparing a program for the paraphraser are described in this section. The hardware/ software environment for the paraphraser is described in Section 3.

## 7-3. PARAPHRASER RULES

The Paraphraser accepts 80-character variable-field-length source records. The 80-character line is wide enough to contain the longest microorder phrase and a comment on the same line.

The "free-format" construction of microinstructions means that you can write them in sentence-like groups of expressions. These expressions include microorders, which are translated by the paraphraser into microcode. They do not have to be organized rigidly into fields of a certain number of bits, or in a certain order as required by conventional microassemblers.

All the fields do not have to be specified in the microinstruction sentences since the paraphraser will automatically put "default" microorders in the proper microinstruction fields. Operation Code (OP Code) microorders determine the microinstruction word type and the field requirements.

Source programs contain the following elements:

- Control statement.
- Comments.
- Sentences which are composed of microinstruction specifications including field/microorder expressions or phrases, numbers, labels, and directives.

## 7-4. CONTROL STATEMENT

Every microprogram must start with a control statement in the first line. This statement must start in column one with the syntax description file mnemonic MPARA. The MPARA mnemonic is followed by one or more command options spaced by commas. The options are as follows:

- Option L for a label listing.
- Option F for a floating field listing.
- An optional name and comment field that is displayed by WLOAD when reading microcode files. Format: ‘ NAM, comment’;.

Examples of control statements are the following:

```
MPARA,L,F;   *Control statement for label and floating field listings
MPARA,F,     'NIS, Vector instruction set microcode';
MPARA,       'SORT, integer sort microcode',L;
```

## 7-5. COMMENTS

A comment either follows an asterisk ( * ) or it is enclosed in a set of brackets ( { . . . } ). Note that these are the upper case brackets (equivalent to capital letters) on Hewlett-Packard terminals. Comments are ignored by the paraphraser and passed on to the list device. Asterisk noted comments are useful for general commenting. Bracketed comments are useful for numbering the entries in a microcode jump table, for comments requiring several lines, and for nulling out sections of code as you go through the debugging process.

Rules for comments are the following:

1. A comment can start in any column on any line.

2. When a comment is defined by an asterisk ( * ) the remainder of the line is ignored. The asterisk must be repeated on each comment line.

3. When a comment is defined by an opening bracket ( { ) it must be completed with a closing bracket ( } ) which need not be on the same line. A microinstruction can be included on the same line following a closing bracket. If there is a opening bracket with no closing bracket, an error message will be produced. An example of a comment follows where GOTO 0 is a microinstruction:

```
{THIS IS A COMMENT} GOTO 0; *THIS IS ALSO A COMMENT

{THIS IS A COMMENT INCLUDING AN ASTERISK *} GOTO 0;

GOTO 0; *{ }THIS IS A COMMENT DUE TO THE ASTERISK
```

## 7-6. NUMBERS

MPARA lets you specify values that are decimal, hexadecimal or octal. The base of the number is determined as follows:

a. If the number begins with "1-9", it is interpreted as decimal number; for example, 2048 or 8196.

b. If the number begins with "0x" or "0X", the rest of the number is interpreted as hexadecimal; for example, 0xFFFF or 0X1AF.

c. If the number begins with "0" (and not "0x" or "0X"), the rest of the number is interpreted as octal; for example, 0777 or 0127772.

## 7-7. MICROINSTRUCTION SENTENCE

A microinstruction sentence is specified as a group of labels, directives, and microinstruction specifications which is ended by a semicolon ( ; ).

A microinstruction sentence can be on more than one line, and separate phrases in the same sentence are separated by a comma ( , ). An asterisk noted comment can be included on any line following a microinstruction sentence, phrase, or directive. Sentences, phrases, and paraphraser errors are explained in detail below under Writing Microinstructions.

## 7-8. LABELS

MPARA allows you to give symbolic names to microaddresses. These names are called "labels." A label consists of a set of consecutive characters followed by a colon ( : ) and a blank. The accepted set of characters in the label symbol are described by the following:

- First Character: A-Z, a-z, period (.), underscore (__), "at" symbol (@).
- Characters After First: A-Z, a-z, 0-9, period (.), underscore (__), "at" symbol (@).
- Termination: Label continues until a colon (:) is found.
- Length Allowed: Any length, up to a full line.
- Uniqueness: Labels must be unique in the first eight characters. Lower case letters are equivalent to upper case letters (A-Z = a-z).

MPARA allows you to specify more than one label per sentence and to include multiple directives per sentence.

The following is a simple example:

```
LABEL1:                    *both labels have the same value
LABEL2:                    *
  GOTO LABEL3;             *
```

The following is a more complex example:

```
LABEL1: $origin 50$        *both labels will actually have the
LABEL2:                    *value decimal 50, even though LABEL1
  GOTO LABEL3;             *occurs before origin directive
```

The following is an very complex example:

```
LABEL1: $origin 50$        *both labels will actually have the
LABEL2: $origin 100$       *value decimal 200!
  GOTO LABEL3              *Note that the last origin directive
$ORIGIN 200$ ;             * specified is the one that counts
```

The address of the last origin directive seen by MPARA takes precedence over any other origin directive addresses.

The following lines contain examples of both good and bad labels:

```
this_is_a_good_label:    *good
hello:                   *good
this_/is_a bad_label:    *it has the unacceptable character (/)
8this_is_bad_also:       *it begins with a number
hello_world:             *these two labels are not unique in their
hello_worlds:            *first eight characters.
woops :                  *bad..the colon must follow the symbol
```

For good labels, the label takes on the value of the following microinstruction's microaddress. For bad or duplicated labels, an error message is produced and the rest of the microinstruction is skipped.

## 7-9. DIRECTIVES

Directives are commands you give to MPARA along with your microinstruction specifications. A directive is used for one of the following: setting the current microaddress to a value, aligning the microaddress forward to a 16 or 64 microword boundary, or defining new names for microorders.

When MPARA detects any bad directive, it will output an appropriate error message and skip the current microinstruction.

The directives of MPARA begin and end with a dollar sign ($).

**7-10. ORIGIN DIRECTIVES.** Directives set the address where MPARA places an associated microinstruction. The following examples give good and bad origin directives:

```
$origin 77$              *set origin to decimal 77
$origin 0x1a$            *set origin to hexadecimal 1A
$origin 017$             *set origin to octal 17
$origin 01A$             *bad origin value
$orign 0$                *bad directive
$origin 0 go_to_begin$   *bad directive (too many parameters)
```

Normally when MPARA begins, the current microaddress is automatically set to zero and incremented for each successive microinstruction. However, with the origin directive you can set the address to some other value.

The following specification for two simple microinstructions illustrates the way that the origin directive interacts with labels and other directives:

```
$origin 50$              *set the microaddress to 50 decimal,
OTHER_LABEL:             *define a label
  GOTO THIS_IS_A_LABEL;  *this microinstruction is a jump
$origin 100$             *set the microaddress to 100 decimal
 THIS_IS_A_LABEL:        *define a label here
   GOTO OTHER_LABEL;     *this microinstruction is a jump
```

These two microinstructions perform a jump from microaddress 50 to microaddress 100 and back.

Note that the semicolon ( ; ) is a special character. (However, any semicolons inside of comments are ignored...see the comments section.) The origin directive has only one parameter: the value to which the current microaddress should be set.

**7-11. ALIGN DIRECTIVE.** Sometimes it is convenient to be able to align the current microaddress so that bits 0,1,2,3,4 or more of the microaddress are zero. With this processor, you will most often align to a 64-word block. You might do this to group sections of code to best take advantage of short jumps.

Also, you will align to 16-word blocks to set up jump tables for use with the CT30 and CT74 microorders. The format of the align command is as follows:

```
$align num$
```

where num is the microaddress block to which to align.

Example:

```
ENTER_FOR_JUMPTABLE:          *this microinstruction
  $origin 20$                 *  does a
  GOTO JTABLE, CT30;          *  computed jump

JTABLE: $align 16$            *align this jumptable to location 32.
{00} GOTO LABEL1;             *
{01} GOTO LABEL2;             *  (refer to Table 4-2 for
{02} GOTO LABEL3;             *  a description of CT30)
{etc                                        }
```

**7-12. DEFINE DIRECTIVE.** The define directive allows the user to specify the values to be assigned to microorders that are not already defined in MPARA. The format of the define directive is

```
$DEFINE fld/ord value$
```

where fld is the name of the field in which you want to define the microorder, ord is the name of the microorder you want to define, and value is the numerical value you want to assign to the microorder.

The symbol you specify for the microorder must follow the same character selection rules as for a label. This directive is useful for defining the names of bit masks in the immediate data field, defining the values of labels that exist outside of your source file, or for giving new and meaningful names to the register file registers.

Some examples follow:

```
$DEFINE ADRL/LOOP 0x640$    *this is a label I'll use later
$DEFINE DAT/BITMASK 0x5555$ *this is an immediate data symbolic label
$DEFINE ABUS/F1L 016$       *define a name for
$DEFINE BBUS/F1L 016$       *  a scratch file register to be used
$DEFINE STOR/F1L 016$       *  to hold a floating point operand.
CALL LOOP;                  *use the label I defined.
F1L:=F1L AND BITMASK;       *use the scratch file register and
                            *the immediate data label I defined.
```

## 7-13. MICROINSTRUCTION SPECIFICATIONS

Microinstruction specifications can be freely combined with labels and directives in the same microinstruction sentence as required for the most efficient micromachine operation.

The specifications of the microinstruction sentence consist of the microorders defined in Section 4 along with any acceptable numerical parameters added by the microprogrammer. They can be written either as field/microorder expressions which force the microorder into the specified field unless it produces an unacceptable microinstruction, or simply as phrases acceptable to the paraphraser for interpretation into field/microorder expressions as defined below under Description of Phrases.

## 7-14. WRITING MICROINSTRUCTIONS

Microinstructions are written in free-form sentences that are to be translated into binary code by the paraphraser. It is not necessary to specify all fields of the microinstructions since the specified field or fields will cause the paraphraser to produce defaulted microorders in the remaining fields as required. The defaulted microorder requirements are determined by the particular word type of the microinstruction; and the particular word type is determined by the op code (operation code) field. Sometimes the word type is specified in your program but usually it is deduced by the paraphraser from the phrase you have written for some other field. When a microorder is specified for a particular field (called a "forced field phrase"), the microorder is forced into that field as long as it is acceptable to the paraphraser.

When your microprogram is processed by the paraphraser, it interpretes the phrases, directives, and labels, adds defaulted microorders according to word type, sorts them into field/microorder format, and then translates the field/microorders into microcode to form complete microinstructions. An example microprogram source file is shown in Figure 7-1.

## 7-15. SENTENCES

The microinstruction sentences of your source file are a group of characters followed by a semicolon ( ; ). MPARA translates your sentences into microinstructions. MPARA does not limit your options in expressing microinstructions, it increases it.

You can represent any valid microinstruction by an MPARA sentence. Also, it is extremely versatile; e.g., MPARA will let you insert mnemonics in the fields, and even will let you use phrases that look like PASCAL code such as (a:=b+100;). The free-format syntax of MPARA gives you lots of room for adding comments (a highly recommended practice). See the sample microprograms in Section 12 for examples of microcode written for MPARA.

Note: Numbers on the left are line numbers in the source file.

```
01  MPARA,L; *memory reference utilities <811110.1357>
02
03  $define adrl/TDI_DISABLE 0x7d0$
04
05  INDREAD: $origin 0x7c0$       *INDIRECT RESOLUTION UTILITY
06    nop:=t, rdb;                *1st level: start new read
07    if not b15 then rtn,        *  if address was direct then return
08      s6:=ma+one;               *  save address+1 of read in s6
09    nop:=t, rdb;                *2nd level: start new read
10    if not b15 then rtn,        *  if address was direct then return
11      s6:=ma+one;               *  save address+1 of read in s6
12    indrloop: nop:=t, rdb;      *3rd level and beyond: start new read
13    if not b15 then rtn,        *  if address was direct then return
14      s6:=ma+one;               *  save address+1 of read in s6
15  call TDI_DISABLE, nop:=memr;  *assure that interrupts are enabled
16  goto indrloop;                *loop until indirect is resolved
```

Line 01: Control Statement. MPARA required for the paraphraser to read its syntax description file when executing, L is an option that produces a label listing, and anything after the asterisk is a comment (*memory.....).

Line 03: Directive to MPARA defining a symbolic label for the subroutine TDI__DISABLE which is presumably in another file or in the A700 base set ROMs.

Line 05: Directive to MPARA that specifies that the current microinstruction will be placed at microaddress 7C0 (hexadecimal). It defines the symbolic label for the microaddress as INDREAD in the ADRL field.

Line 06: Two microinstruction phrases. The (nop:=t) phrase means that MPARA will set the STOR field to the NOP microorder, the ALU field to the ADBC microorder (pass from BBUS to YBUS), and the BBUS field to the T microorder. The (rdb) phrase means that the SP0 field will contain the RDB microorder. Note that these phrases could have been reversed, or on different lines since the MPARA language is free-format.

Lines 07 and 08: Two phrases of one complete microinstruction sentence. The (if not b15 then rtn) is a conditional phrase that says that MPARA will set the OP2 field to the RTNF microorder, and the CNDX field to the B15 microorder. The (s6:=ma+one) is an arithmetic phrase which says that MPARA will set the STOR field to the S6 microorder (a scratch file register), the ALU field to the ADBC microorder, the BBUS field to the MA microorder, and the SP0 field to FCIN.

Lines 09 through line 14: Contains microinstructions identical to those explained previously.

Line 12: Defines symbolic label. Creates microorder INDRLOOP in the ADRL field with a value of 7C4 (hexadecimal).

Line 15: Contains a branch phrase that loads the OP5 field with JSBL, and the ADRL field will contain the value of the TDI__DISABLE microorder defined in line 03. The (nop:=memr) is an arithmetic phrase similar to line 06, except that the BBUS field will contain the MEMR microorder.

Line 16: Contains a branch phrase that causes the microprogram to loop. Note this microinstruction specifies the contents of the OP5 and ADRL fields but does not specify the BBUS, ALU or STOR fields. These unspecified fields default to the ACC, ZERO and NOP fields, respectively.

Figure 7-1. Example of Microprogram Source File

## 7-16. PHRASES

Sentences in MPARA, as in the natural language analogy, can be subdivided into phrases. A phrase in MPARA is a group of characters that specifies one or more microorders. The statement "one or more" is important because some of the microorders of the micromachine are interrelated; e.g., the source and destination of an ALU operation along with the ALU operation itself.

Phrases are a "short cut" to writing microcode. For example, MPARA allows you to write the phrase "A :=B" to specify that the STOR field gets the A microorder, the BBUS field gets the B microorder, and the ALU field gets the ADBC microorder. Written out completely in microorders this is "STOR/A, BBUS/B, ALU/ADBC", where a "microorder" inserted in a "field" is represented as "field/microorder."

However, you must have an understanding of the micromachine (covered in Section 2). You should also have a working knowledge of the microorders, the fields, and the microinstruction Word Type formats (covered in Section 4). You should know what fields you are using when you specify a phrase. Because you are microcoding, you probably want to optimize the given task. Also, the amount of micromemory available is limited so you'll want to do as much operation in one microcycle as possible. This will cause your microprograms to run faster.

Phrases recognized by the paraphraser are described in detail in this section of the manual under Description of Phrases.

## 7-17. WRITING PHRASES

Phrases in MPARA are separated by commas, and the last phrase of a microinstruction sentence ends with a semicolon. You can chain together phrases until MPARA accepts your sentence or finds an error in it. If MPARA accepts your sentence, then it will produce a microinstruction in the microcode file. If MPARA does finds an error, it will put an error message in your listing file and skip to the next microinstruction sentence.

As with any MPARA sentence, you need not put the whole phrase on one line of your source file. Here are some examples:

```
IF NOT YZ                    *If the YBUS was not zero in the
  GOTO 63;                   *preceeding cycle, then branch.
IF INTP CALL SUBR01;         *If interrupts are present then
                             *call a subroutine
```

Note that some phrases have more than one meaning. For instance, the GOTO 0 phrase can mean OP4/JMP,ADRS/0 for a short jump or OP5/JMPL,ADRL/0 for a long jump. In any case, MPARA will attempt to chose the appropriate meanings so that a valid microinstruction will be constructed.

## 7-18. DEFAULTED FIELDS

When your microinstruction sentence does not fill every field in the word type you are using, MPARA will attempt to default the remaining fields. However, not all fields can be reasonably defaulted (refer to Bad Field Default below).

One example of field defaulting is the following sentence:

```
NOP;
```

in which the OP1 field gets the no-operation (NOP) microorder, the SP2 field defaults to NOP, the SP0 field defaults to NOP, the ABUS defaults to ACC, the BBUS defaults to ACC, the STOR field defaults to NOP, and the ALU field defaults to ZERO.

An even briefer way to specify a NOP microinstruction is to write semicolon ( ; ) as the sentence. In this case all of the fields default to the same microorders as when "NOP;" is the sentence and the OP1 field defaults to NOP.

The default values are noted in the summary of microorders in Appendix B.

The following paragraphs describe some of the errors that MPARA looks for in your microinstruction sentences. The Section 8 on Using the Paraphraser covers the listing file and has a complete listing of errors and examples.

**7-19. FIELD CONFLICTS.** If the sentence that you have specified is written such that it tries to fill a field more than once, it will produce an error. As an example, if your sentence was the following:

```
A:=B, B:=A;
```

the paraphraser would be required to fill three fields of the microinstruction twice (which it cannot do).

**7-20. WORD TYPE CONFLICT.** Your sentence selects microorders that exist only in fields of different word types. For example, an immediate data (IMM) of operation 6 (OP6) cannot occur in the same microcycle as a jump (JMP) of operation 4 (OP4); thus an error is produced if an attempt was made to program these in the same microinstruction.

**7-21. UNRECOGNIZABLE PHRASE.** If MPARA cannot match one or more of your phrases to the acceptable phrases, it will produce an error. For example, if you made a typing mistake and specified "GOTTO 0;" as a microinstruction instead of "GOTO 0", MPARA would not recognize the phrase.

**7-22. BAD FIELD DEFAULT.** Not all of the fields have default values. For example, if you specified a phrase that only filled the ADRL (Long Branch Address) field, the paraphraser would not know whether you wanted the OP5 field to default to JMPL or JSBL. Also, the Condition Field (CNDX) and ALU Special (ALUS) fields do not have default values. Bad Field Defaults will produce errors and the error messages will be stored in the list file.

## 7-23. DESCRIPTION OF PHRASES

Phrases accepted by the paraphraser are described below under several categories as follows:

a. Branching Phrases

b. Arithmetic Phrases

c. Conditional Phrases

d. Special Phrases

e. Field-Forcing Phrases

The phrases you can use are listed below along with the resulting fields for each, and the meaning of the phrase.

A summary of phrases is provided in Appendix C.

## 7-24. BRANCHING PHRASES

Branching phrases are used for either short jumps to the current 64-word memory block or long jumps to anywhere in the 16k-word control memory. Branching word types are either Word Type 4 for GOTO jumps to address; or Word Type 5 for CALL jumps to subroutine; or Word Type 1 miscellaneous branches for return from subroutine, no operation, or to decode a macroinstruction. Branching phrases are described below.

### JUMP TO ADDRESS BRANCHING PHRASES

GOTO adr

Fields: OP4/JMP ADRS/adr.

Meaning: Short jump to address (adr).

GOTO adr

Fields: OP5/JMPL ADRL/adr.

Meaning: Long jump to address (adr).

LGOTO adr

Fields: OP5/JMPL ADRL/adr.

Meaning: Long jump to address (adr).

SGOTO adr

Fields: OP4/JMP ADRS/adr.

Meaning: Short jump to address (adr).

## JUMP TO SUBROUTINE BRANCHING PHRASES

CALL adr

Fields: OP5/JSBL ADRL/adr.

Meaning: Long jump to subroutine (adr).

CALL adr

Fields: OP4/JSB ADRS/adr.

Meaning: Short jump to subroutine (adr).

LCALL adr

Fields: OP5/JSBL ADRL/adr.

Meaning: Long jump to subroutine (adr).

SCALL adr

Fields: OP4/JSB ADRS/adr.

Meaning: Short jump to subroutine (adr).

## JUMP TO TABLE BRANCHING PHRASES

GOTOTBL adr

Fields: OP5/JMPL ADRL/adr ALU/SPEC.

Meaning: See Note below on the decoding of macroinstructions below.

CALLTBL adr

Fields: OP5/JSBL ADRL/adr ALU/SPEC

Meaning: See Note below on the decoding of macroinstructions.

NOTE

GOTOTBL and CALLTBL are the phrases generally used for decoding macroinstructions. After the processor executes the JTAB microorder in the control firmware, all macroinstructions in the 101XXX and 105XXX range are decoded to blocks of 16 consecutive opcodes. Once in the jump table, the firmware does a GOTOTBL to a consecutive block of 16 microaddresses, based on the low four bits of the macroinstruction in CT (instruction register).

For example, if the macroinstruction in CT (instruction register) is 105005 the GOTOTBL phrase is "goto 0x1000;", the firmware will branch to location 1005 (hexadecimal). You should place jumps to your code in these locations, or jumps to the unimplemented opcode handler in the control firmware. CALLTBL is the same as GOTOTBL except that it results in a "jump to subroutine" (the current microaddress plus one is pushed onto the subroutine stack).

The miscellaneous branches are the following:

NOP

Field: OP1/NOP

Meaning: A NOP (no operation) will not perform a branch or return; instead, the current address is incremented to the next microinstruction.

JTAB

Field: OP1/JTAB.

Meaning: The JTAB microorder is a special microoder for decoding macroinstructions. Do not use this microorder unless you are changing the method of decoding macroinstructions and handling interrupts. JTAB performs a "jump to macroinstruction subroutine" to a microaddress in the range of hexadecimal 100-1FF (refer to the base set subroutine for FPLA).

RTN

Field: OP1/RTN

Meaning: Return to the microaddress on the micromachine stack, and decrement the stack pointer. Used for returning from subroutines that write, or to return from your microroutine to the control firmware.

## 7-25. ARITHMETIC PHRASES

Arithmetic phrases allow the user to specify the microorders for arithmetic operations. The A700 computer has a three address architecture, meaning that the inputs to the ALU come from two sources that you specify and the output goes to a register that you specify.

The value on the A-Bus (as specified by the ABUS or DAT fields) can be one of the internal-register file registers or the immediate data value specified in the microinstruction. The value on the B-Bus (as specified by the BBUS field) can be any of the readable registers on the machine. The destination of the Y-Bus (as specified by the STOR field) can be any of the writable registers on the machine, or to the bit bucket ( NOP ).

ALU operations and shift operations in phrases are signified by " := " where the register to be written in the STOR field (Y-Bus) is on the left of the expression and the source register and input to the ALU (abus or bbus) is on the right of the expression.

NOTE

In the arithmetic phrases given below, the lower case (non-capital) letters represent registers specified by the programmer. Where a register is represented by a bus, (e.g., abus, bbus) the bus indicated is the input to the ALU that will be used to transfer the contents of named register (internal or any readable register for the abus or bbus, respectively).

The capital letter mnemonics on the left of the resulting field expressions are the names of the microinstruction fields.

Also, shift and rotate modifiers can be added to the arithmetic phrases, as can microorders that modify the carry-in of the ALU.

Arithmetic phrases are divided into the following categories:

a. Basic arithmetic phrases;

b. Arithmetic phrases with shift or rotate;

c. Arithmetic phrases modifying carry-in;

d. Arithmetic phrases using alu special field;

e. Arithmetic phrases with immediate data.

**7-26. BASIC ARITHMETIC PHRASES.** The basic arithmetic phrases allow you to specify ALU operations involving the ALU field, BBUS field, STOR field, and ABUS field. This includes operations such as passing the value of a register through the ALU to another register, or adding the contents of two registers and storing the results in another register.

The basic arithmetic phrases are described below showing the resulting fields and the meaning of the phrase:

### BASIC ARITHMETIC PHRASES

`stor := bbus`

Fields: ALU/adbc STOR/stor BBUS

Meaning: The value of the register specified as "bbus" is passed through the ALU and stored in the register specified as "stor."

`stor := abus`

Fields: ALU/adac STOR/stor ABUS/abus

Meaning: The value of the register specified as "abus" is passed through the ALU and stored in the register specified as "stor."

`stor := NOT bbus`

Fields: ALU/cmbc STOR/stor BBUS/bbus

Meaning: The one's complement of the value of the register specified as "bbus" is stored in the register specified as "stor."

`stor := NOT abus`

Fields: ALU/cmac STOR/stor ABUS/abus

Meaning: The one's complement of the value of the register specified as "abus" is stored in the register specified as "stor."

`stor := ONES`

Fields: ALU/xnor STOR/stor BBUS/ACC ABUS/ACC

Meaning: The 16-bit quantity which is all ones is stored into the register specified as "stor." Note that "all ones" is the two's complement representation of number "−1."

stor := ZEROS

Fields: ALU/xor STOR/stor BBUS/ACC ABUS/ACC

Meaning: The 16-bit quantity which is all zeros is stored into the register specified as "stor."

stor := ZERO

Fields: ALU/zero STOR/stor

Meaning: The 16-bit quantity which is all zeros is stored into the register specified as "stor." Note that this operation will not update the CF, Y15, B15, SF and ALOV conditions.

stor := NOT abus AND bbus
stor := bbus AND NOT abus

Fields: ALU/cand STOR/stor BBUS/bbus ABUS/abus

Meaning: The one's complement of the register specified as "abus" is logically ANDed with the content of the register specified as "bbus",and stored into the register specified as "stor."

stor := abus - bbus

Fields: ALU/sbbc STOR/stor BBUS/bbus ABUS/abus

Meaning: The result of subtracting the content of the "bbus" register from the content of the "abus" register is stored in the "stor" register.

stor := bbus - abus

Fields: ALU/sbac STOR/stor BBUS/bbus ABUS/abus

Meaning: The result of subtracting the content of the "abus" register from the content of the "bbus" register is stored in the "stor" register.

stor := abus + bbus
stor := bbus + abus

Fields: ALU/addc STOR/stor BBUS/bbus ABUS/abus

Meaning: The result of summing the content of the "abus" register with the content of the "bbus" register is stored in the "stor" register.

stor := abus XNOR bbus
stor := bbus XNOR abus

Fields: ALU/xnor STOR/stor BBUS/bbus ABUS/abus

Meaning: The result of exclusive NORing the content of the "abus" register with the content of the "bbus" register is stored in the "stor" register.

stor := abus XOR bbus
stor := bbus XOR abus

Fields: ALU/xor STOR/stor BBUS/bbus ABUS/abus

Meaning: The result of exclusive ORing the content of the "abus" register with the content of the "bbus" register is stored in the "stor" register.

```
stor := abus AND bbus
stor := bbus NAND abus
```

Fields: ALU/and STOR/stor BBUS/bbus ABUS/abus

Meaning: The result of ANDing the content of the "abus" register with the content of the "bbus" register is stored in the "stor" register.

```
stor := abus NAND bbus
stor := bbus NAND abus
```

Fields: ALU/nand STOR/stor BBUS/bbus ABUS/abus

Meaning: The one's complement of the logical AND of the contents of the "abus" register and the "bbus" register is stored into the "stor" register.

```
stor := abus IOR bbus
stor := bbus IOR abus
```

Fields: ALU/ior STOR/stor BBUS/bbus ABUS/abus

Meaning: The inclusive OR of the contents of the "abus" register and the "bbus" register is stored in the "stor" register.

```
stor := abus INOR bbus
stor := bbus INOR abus
```

Fields: ALU/inor STOR/stor BBUS/bbus ABUS/abus

Meaning: The inclusive NOR of the contents of the "abus" register and the "bbus" register is stored in the "stor" register.

**7-27. ARITHMETIC PHRASES WITH SHIFT OR ROTATE.** The arithmetic phrases with shift or rotate allow single- or double-word shifts of values that can be specified in basic arithmetic phrases.

### ARITHMETIC PHRASES WITH SHIFT OR ROTATE

`stor := LL1` (right side of basic arithmetic phrase)

Field: SP0/LL1 or SP1/LL1

Meaning: The result of the basic arithmetic phrase is logically left-shifted and stored in the "stor" register.

`stor := LR1` (right side of basic arithmetic phrase)

Field: SP0/LR1 or SP1/LR1

Meaning: The result of when the basic arithmetic phrase is logically right-shifted and stored in the "stor" register.

`stor := RL1` (right side of basic arithmetic phrase)

Field: SP0/RL1 or SP1/RL1

Meaning: The result of the basic arithmetic phrase is left-rotated and stored in the "stor" register.

stor := RR1 (right side of basic arithmetic phrase)

Field: SP0/RR1 or SP1/RR1

Meaning: The result of the basic arithmetic phrase is right-rotated and stored in the "stor" register.

stor := AL1 (right side of basic arithmetic phrase)

Field SP0/AL1 or SP1/AL1

Meaning: The result of when the basic arithmetic phrase is left-shifted arithmetically and stored in the "stor" register.

stor := AR1 (right side of basic arithmetic phrase)

Field: SP0/AR1 or SP1/AR1

Meaning: The result of the basic arithmetic phrase is right-shifted arithmetically and stored in the "stor" register.

**7-28. ARITHMETIC PHRASES WITH CARRY-IN MODIFIER.** The arithmetic phrases with carry-in modifier allow the user to specify microorders in the ALU, ABUS, BBUS, and STOR fields, and the FCIN and ACF microorders in the SP0 and SP1 fields.

### ARITHMETIC PHRASES WITH CARRY-IN MODIFIER

stor := bbus + ONE

Fields: ALU/adbc STOR/stor BBUS/bbus SP0/FCIN or SP1/FCIN

Meaning: The content of the "bbus" register plus one is stored in the "stor" register.

stor := abus + ONE

Fields: ALU/adac STOR/stor ABUS/abus SP0/FCIN or SP1/FCIN

Meaning: The content of the "abus" register plus one is stored in the "stor" register.

stor := - bbus
stor := NOT bbus + ONE

Fields: ALU/cmbc STOR/stor BBUS/bbus SP0/FCIN or SP1/FCIN

Meaning: The two's complement of the "bbus" register is stored in the "stor" register.

stor := - abus
stor := NOT abus + ONE

Fields: ALU/cmac STOR/stor ABUS/abus SP0/FCIN or SP1/FCIN

Meaning: The two's complememt of the "abus" register is stored in the "stor" register.

stor := abus - bbus - ONE

Fields: ALU/sbbc STOR/stor BBUS/bbus ABUS/abus SP0/FCIN or SP1/FCIN

Meaning: The result of subtracting the content of the "bbus" register and the value one from the "abus" register is stored in the "stor" register.

`stor := bbus - abus - ONE`

Fields: ALU/sbac STOR/stor BBUS/bbus ABUS/abus SP0/FCIN or SP1/FCIN

Meaning: The result of subtracting the content of the "abus" register and the value one from the "bbus" register is stored in the "stor" register.

`stor := bbus + abus + ONE`
`stor := abus + bbus + ONE`

Fields: ALU/addc STOR/stor BBUS/bbus ABUS/abus SP0/FCIN or SP1/FCIN

Meaning: The sum of the "abus" register plus the "bbus" register plus one is stored in the "stor" register

`stor := bbus + CF`

Fields: ALU/adbc STOR/stor BBUS/bbus SP0/ACF or SP1/ACF

Meaning: The sum of the "bbus" register plus the value of the carry flag is stored in the "stor" register.

`stor := abus + CF`

Fields: ALU/adac STOR/stor ABUS/abus SP0/ACF or SP1/ACF

Meaning: The sum of the "abus" register plus the value of the carry flag is stored in the "stor" register.

`stor := NOT bbus + CF`

Fields: ALU/cmbc STOR/stor BBUS/bbus SP0/ACF or SP1/ACF

Meaning: The sum of the one's complement of the "bbus" register and the carry flag is stored in the "stor" register.

`stor := NOT abus + CF`

Fields: ALU/cmac STOR/stor ABUS/abus SP0/ACF or SP1/ACF

Meaning: The sum of the one's complement of the "abus" register and the carry flag is stored in the "stor" register.

`stor := abus - bbus - BR`

Fields: ALU/sbbc STOR/stor BBUS/bbus ABUS/abus SP0/ACF or SP1/ACF

Meaning: The result of subtracting the content of the "bbus" register and content of the carry flag from the "abus" register is stored in the "stor" register. The content of the carry flag is the borrow (BR).

`stor := bbus - abus - BR`

Fields: ALU/sbac STOR/stor BBUS/bbus ABUS/abus SP0/ACF or SP1/ACF

Meaning: The result of subtracting the content of the "abus" register and the content of the carry flag from the "bbus" register is stored in the "stor" register. The content of the carry flag is the borrow (BR).

```
stor := abus + bbus + CF
stor := bbus + abus + CF
```

Fields: ALU/addc STOR/stor BBUS/bbus ABUS/abus SP0/ACF or SP1/ACF

Meaning: The sum of the "abus" register, the "bbus" register, and the content of the carry flag is stored in the "stor" field.

**7-29. ARITHMETIC PHRASES WITH ALU SPECIAL.** Microinstruction sentences which have microorder SPEC (Special) in the ALU field take on a special meaning. The third field that contains SP0 and SP1 in word types 1, 2, 3, and 4 becomes the ALUS field and the word types are defined as 1S, 2S, 3S, and 4S. The ALUS field must contain the ALUS microorders. The phrases for these microorders are listed below.

## ARITHMETIC PHRASES WITH ALU SPECIAL

```
stor := SWAP ( bbus )
```

Fields: ALU/SPEC STOR/stor BBUS/bbus ALUS/SWAP

Meaning: The upper byte of the "bbus" register is stored in the lower byte of the "stor" register, and the lower byte of the "bbus" register is stored in the upper byte of the "stor" register.

```
stor := SWZU ( bbus )
```

Fields: ALU/SPEC STOR/stor BBUS/bbus ALUS/SWZU

Meaning: The upper byte of the "bbus" register is stored in the lower byte of the "stor" register, and the upper byte of the "stor" register is set to zero.

```
stor := SWZL ( bbus )
```

Fields: ALU/SPEC STOR/stor BBUS/bbus ALUS/SWZL

Meaning: The lower byte of the "bbus" register is stored in the upper byte of the "stor" register, and the lower byte of the "stor" register is set to zero.

```
stor := ZUY (bbus)
```

Fields: ALU/SPEC STOR/stor BBUS/bbus ALUS/ZUY

Meaning: The lower byte of the "bbus" register is stored in the lower byte of the "stor" register, and the upper byte of the "stor" register is set to zero.

```
stor := ZLY ( bbus )
```

Fields: ALU/SPEC STOR/stor BBUS/bbus ALUS/ZLY

Meaning: The upper byte of the "bbus" register is stored in the upper byte of the "stor" register, and the lower byte of the "stor" register is set to zero.

```
stor := SRG (bbus)
```

Fields: ALU/SPEC STOR/stor BBUS/bbus ALUS/SRG

Meaning: The content of the "bbus" register is operated on by the SRG function, and the result is stored in the "stor" register.

```
stor := RL4 (bbus)
```

Fields: ALU/SPEC STOR/stor BBUS/bbus ALUS/RL4

Meaning: The content of the "bbus" register is rotated left four bits and the result is stored in the "stor" register.

```
stor := ASG ( bbus )
```

Fields: ALU/SPEC STOR/stor BBUS/bbus ALUS/ASG

Meaning: The content of the "bbus" register is operated on by the ASG function, and the result is stored in the "stor" register.

```
stor := UMPY (bbus ,abus)
stor := UMPY (bbus)
```

Fields: ALU/SPEC STOR/stor BBUS/bbus ABUS/abus ALUS/UMPY

Meaning: Perform the unsigned multiply step on the registers specified as "bbus" and "abus" and store the result into the "stor" register.

```
stor := TMPY (bbus ,abus)
stor := TMPY (bbus)
```

Fields: ALU/SPEC STOR/stor BBUS/bbus ABUS/abus ALUS/TMPY

Meaning: Perform the two's-complement multiply step on the registers specified as "bbus" and "abus" and store the result into the "stor" register.

```
stor := SM2C ( bbus )
```

Fields: ALU/SPEC STOR/stor BBUS/bbus ABUS/abus ALUS/SM2C

Meaning: Perform the signed-magnitude to two's-complement conversion on the "bbus" register and store the result into the "stor" register.

```
stor := TMLC (bbus ,abus)
stor := TMLC (bbus)
```

Fields: ALU/SPEC STOR/stor BBUS/bbus ABUS/abus ALUS/TMLC

Meaning: Perform the last cycle of the two's complement multiply on the "bbus" register and store the result into the "stor" register.

```
stor := DNRM (bbus, abus)
stor := DNRM ( bbus )
```

Fields: ALU/SPEC STOR/stor BBUS/bbus ABUS/abus ALUS/DNR

Meaning: Perform the double-normalize step on the contents of the "bbus" register, and store the result into the "stor" register.

```
stor := SNRM ( bbus )
```

Fields: ALU/SPEC STOR/stor BBUS/bbus ABUS/abus ALUS/SNR

Meaning: Perform the single-normalize step on the "bbus" register contents and store the result into the "stor" register.

```
stor := DIV (bbus,abus)
stor := DIV (bbus)
```

Fields: ALU/SPEC STOR/stor BBUS/bbus ABUS/abus ALUS/DIV

Meaning: Perform the two's-complement divide step on the contents of the "bbus" and "abus" registers and store the result into the "stor" register.

```
stor := DIV1 (bbus,abus)
stor := DIV1 (bbus)
```

Fields: ALU/SPEC STOR/stor BBUS/bbus ABUS/abus ALUS/DIV1

Meaning: Perform the two's-complement first divide step on the contents of the "bbus" and "abus" registers and store the result into the "stor" register.

**7-30. ARITHMETIC PHRASES WITH IMMEDIATE DATA.** Microinstructions with the arithmetic phrase Immediate Data will have microorder IMM (Immediate) in the first field (op code field) and the data in the DAT (Data) field. These microinstructions are word type 6. The phrases for word type 6 are listed below with their resulting fields and meaning.

### ARITHMETIC PHRASES WITH IMMEDIATE DATA

```
stor := data
```

Fields: ALU/adac STOR/stor DAT/data OP6/IMM

Meaning: The value of "data" is stored into the "stor" register.

```
stor := NOT data
```

Fields: ALU/cmac STOR/stor DAT/data OP6/IMM

Meaning: The one's complement of "data" is stored into the "stor" register.

```
stor := NOT data AND bbus
stor := bbus AND NOT data
```

Fields: ALU/cand STOR/stor BBUS/bbus DAT/data OP6/IMM

Meaning: The one's complement of "data" is logically ANDed with the contents of the "bbus" register and the result is stored into the "stor" register.

```
stor := data - bbus
```

Fields: ALU/sbbc STOR/stor BBUS/bbus DAT/data OP6/IMM

Meaning: The content of the "bbus" register is subtracted from "data" and the result is stored in the "stor" register.

```
stor := bbus - data
```

Fields: ALU/sbac STOR/stor BBUS/bbus DAT/data OP6/IMM

Meaning: The value of "data" is subtracted from the content of the "bbus" register and the result is stored in the "stor" register.

```
stor := bbus + data
```

Fields: ALU/addc STOR/stor BBUS/bbus DAT/data OP6/IMM

Meaning: The value of "data is added to the content of the "bbus" register and the result is stored in the "stor" register.

```
stor := data XNOR bbus
stor := bbus XNOR data
```

Fields: ALU/xnor STOR/stor BBUS/bbus DAT/data OP6/IMM

Meaning: The content of the "bbus" register and "data" are exclusive-NORed and the result is stored in the "stor" register.

```
stor := data XOR bbus
stor := bbus XOR data
```

Fields: ALU/xor STOR/stor BBUS/bbus DAT/data OP6/IMM

Meaning: The content of the "bbus" register and "data" are exclusive-ORed and the result is stored in the "stor" register.

```
stor := data AND bbus
stor := bbus AND data
```

Fields: ALU/and STOR/stor BBUS/bbus DAT/data OP6/IMM

Meaning: The content of the "bbus" register and "data" are logically ANDed and the result is stored in the "stor" register.

```
stor := data NAND bbus
stor := bbus NAND data
```

Fields: ALU/nand STOR/stor BBUS/bbus DAT/data OP6/IMM

Meaning: The content of the "bbus" register and "data" are logically NANDed and the result is stored in the "stor" register.

```
stor := data IOR bbus
stor := bbus IOR data
```

Fields: ALU/ior STOR/stor BBUS/bbus DAT/data OP6/IMM

Meaning: The contents of the "bbus" register and "data" are logically inclusive-ORed and the result is stored in the "stor" register.

```
stor := data INOR bbus
stor := bbus INOR data
```

Fields: ALU/inor STOR/stor BBUS/bbus DAT/data OP6/IMM

Meaning: The contents of the "bbus" register and "data" are logically inclusive-NORed and the result is stored in the "stor" register.

## 7-31. CONDITIONAL PHRASES

Conditional phrases provide the means to specify conditional branching, conditional return, and conditional execution of the SP0 field. These phrases all start with "IF", and allow the specification of the OP2, OP3, and CNDX (condition) fields.

### 7-32. CONDITIONAL BRANCHING PHRASES.

The conditional branching phrases given below result in a jump to a microaddress that is within the same 64-word block as the current address plus one. The jump is either to the address or subroutine that is specified by "adr" in the ADRS field.

The following phrases are recognized for conditional branching:

**CONDITIONAL BRANCHING PHRASES**

IF cndx GOTO adr

Fields: OP3/JMPT CNDX/cndx ADRS/adr

Meaning: If condition is true, then jump to the address specified in ADRS.

IF NOT cndx GOTO adr

Fields: OP3/JMPF CNDX/cndx ADRS/adr

Meaning: If condition is not true, then jump to the address specified in ADRS.

IF cndx CALL adr

Fields: OP3/JSBT CNDX/cndx ADRS/adr

Meaning: If condition is true, then jump to the subroutine at the address specified in ADRS.

IF NOT cndx CALL adr

Fields: OP3/JSBF CNDX/cndx ADRS/adr

Meaning: If condition is not true, then jump to the subroutine at the address specified in ADRS.

### 7-33. CONDITIONAL RETURN PHRASES.

The return from a conditional return phrases is to the microaddress on the micromachine stack, and the stack pointer is decremented. Conditional return phrases are the following:

**CONDITIONAL RETURN PHRASES**

IF cndx THEN RTN

Fields: OP2/RTNT CNDX/cndx

Meaning: Return if condition is true.

IF NOT cndx THEN RTN

Fields: OP2/RTNF CNDX/cndx

Meaning: Return if condition is false.

**7-34. CONDITIONAL SP0 PHRASES.** There are some hardware limitations to conditional SP0 phrases as follows: Conditional shifts and rotates are not allowed (this includes the ALUS field), and the LDQ special can not be performed conditionally. (MPARA does not check for this limitation.) The conditional SP0 phrases are the following:

**CONDITIONAL SP0 PHRASES**

`IF cndx THEN sp0`

| | | | |
|---|---|---|---|
| Fields: | OP2/SP0T | CNDX/cndx | SP0/sp0 |
| Meaning: | Execute microorder in SP0 field if condition is true. | | |

`IF NOT cndx THEN sp0`

| | | | |
|---|---|---|---|
| Fields: | OP2/SP0F | CNDX/cndx | SP0/sp0 |
| Meaning: | Execute microorder in SP0 field if condition is false. | | |

## 7-35. SPECIAL PHRASES

The microorders in the SP0, SP1, and SP2 can be specified simply as the microorder mnenonic. Note that some of the microorders are associated with ALU operations, and the ALU phrases specify some specials, such as FCIN. Some examples are the following:

**SPECIAL PHRASES**

| Special Phrase | Resulting Field | Meaning |
|---|---|---|
| sp0 | SP0/sp0 | sp0 microorder placed in SP0 field |
| sp1 | SP1/sp1 | sp1 microorder placed in SP1 field |
| sp2 | SP2/sp2 | sp2 microorder placed in SP2 field |

## 7-36. FIELD FORCING PHRASES

A field forced phrase is one in which the microorder is given along with the field in which it is to be placed. Several examples of field forced phrase formats are as follows:

| | |
|---|---|
| `OP1/op1` | where an op field 1 microorder (e.g., JTAB) is specified for field OP1; |
| `CNDX/cndx` | where a condition field microorder (e.g., CF) is specified for field CNDX; |
| `DAT/dat` | where data is specified for immediate data for field DAT. |

If a microorder is written in "forced field" form which does not belong in the specified field, an error message will be generated during the microassembling process.

# SECTION 8
## USING THE PARAPHRASER

# PART III
# Microprogramming Support Software and Hardware

# USING THE PARAPHRASER

SECTION 8

This section provides instructions for actually microassembling your microprogram after you have prepared the microprogram using information from Part II of this manual. The MPARA paraphraser microassembler program must be installed in the RTE operating system of your computer. Refer to Section 3 of this manual for guidelines on preparing for microprogramming.

This section provides the information on executing the paraphraser microassembler and information on its output such as the following:

- Binary object code (microcode) file
- Address Label listing
- Floating field listing of the microorders
- Error messages output to list device

The microcode file can be downloaded into the WCS using the WLOAD utility (and the ID.41 WCS driver). WLOAD also generates binary code formatted for burning PROMs.

## 8-1. LOADING MPARA

The paraphraser microassembler program MPARA requires a 32k word memory partition, and it has five segments. MPARA is loaded as follows using the relocating linking loader program LINK:

1. Call LINK from the file manager

```
FMGR: RU,LINK
```

2. Load MPARA using LINK command files as follows:

```
LI,$PLIB
SZ,32 (32k partition SiZe)
RE,%MPARA (RElocate file)
EN (ENd and Exit)
```

## 8-2. USING THE PARAPHRASER MICROASSEMBLER

As described in Section 7, the paraphraser microassembler accepts "free format" microprogram sentences and translates them to produce the binary object code of the microprogram. The program control statement at the beginning of the program determines if a label listing and a floating field listing will be output on the list device. Error messages, if any, cannot be suppressed and will always be output to the list device.

Refer to Sections 4 and 7 for descriptions of the microinstructions and how to write them for the paraphraser. The following paragraphs provide a procedure for microassembling a microprogram.

## 8-3. EXECUTION COMMAND

The paraphraser can be scheduled to run on the HP 1000, A700 computer system with this command:

:RU,MPARA,*source input,list output,binary output*

*source input* Name of FMGR file containing the Paraphraser source code; this entry must conform to the format required by the FMGR namr parameter.

*list output* Choose one of the following:

– (minus sign)
FMGR file name
logical unit number

If the minus sign is specified, and the source file name begins with an ampersand (&), the ampersand is replaced with a apostrophe and the remaining source file name characters are used for the list file name. The list file is forced to reside on the same cartridge (character reference code) as the source file. For example:

| | |
|---|---|
| &LIST | source file |
| 'LIST | list file name |

If an FMGR file name is specified, it must conform to the format required by the FMGR *namr* parameter. The list file is created if it does not exist. If the file does exist, the first character in the file name must be an apostrophe; otherwise an error results.

If a logical unit number is specified, the listed output is directed to that logical device.

*binary output* Choose one of the following:

– (minus sign)
FMGR file name
logical unit number

If the minus sign is specified, and the source file name begins with an ampersand (&), the ampersand is replaced with a percent symbol and the remaining source file name characters are used for the binary file name. The binary file is forced to reside on the same cartridge (character reference code) as the source file. For example:

| | |
|---|---|
| &MCODE | source file |
| %MCODE | binary file name |

If an FMGR file name is specified, it must conform to the format required by the FMGR *namr* parameter. The binary file is created if it does not exist. If the file does exist it is necessary that

a. the first character of the file's name be a percent sign (%).

b. the existing file be the type specified in the *namr* parameter (if the file type is not declared in namr, the file's type must be Type 5, relocatable binary).

If the above conditions are not met, a paraphraser error will result.

If a logical unit number is specified, the binary output is directed to that logical device.

If this parameter is omitted, no binary object code is generated. Examples;

```
:RU,MPARA,&SOURCE,'LIST,%MCODE
```

Schedules MPARA to microassemble source file &SOURCE. Listed output is directed to list file 'LIST, and binary object code is directed to binary file %MCODE.

When MPARA is finished translating your microprogram, it will print the following on your terminal:

```
/MPARA: total errors: n
```

where n is the number of mistakes you have made or if n=0 there are no mistakes.

## 8-4. THE PARAPHRASER OUTPUT

The following paragraphs describe the various forms of output available from the paraphraser microassembler; namely, the microassembled binary object code, the source listing, the optional label listing, the optional floating-field listing, and error messages.

## 8-5. BINARY OBJECT CODE

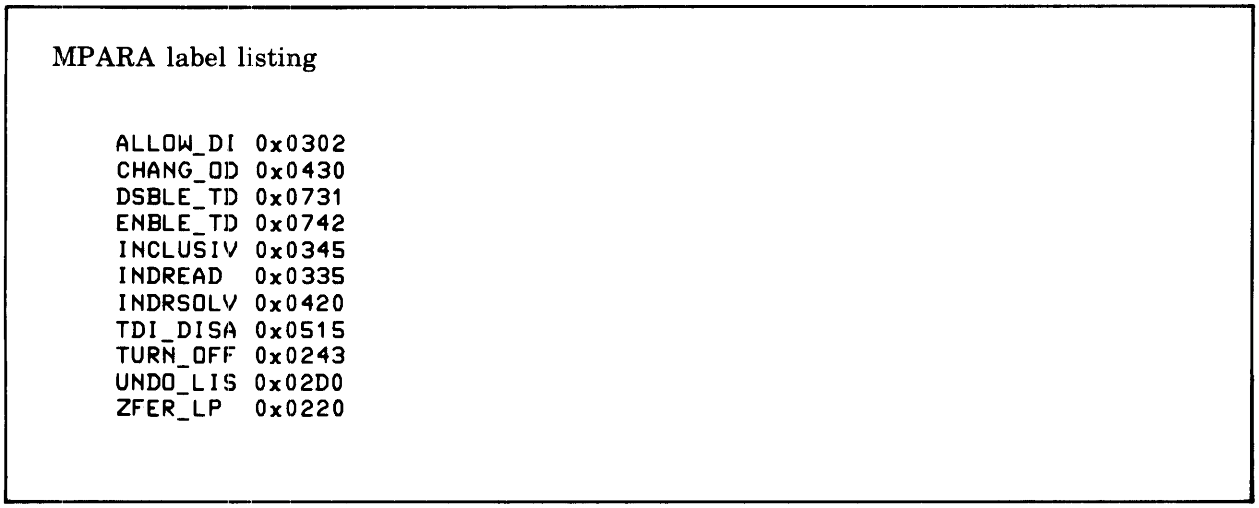
The standard object code output of the paraphraser microsassembler to a disc file or some other output device must consist of one or more microinstruction records including a NAM (name) record.

The standard object format is acceptable by all programs that accept standard relocatable format (RTE type 5 files). Therefore the object code can be stored from an input device into a disc file as a binary relocatable by the FMGR STore command. If the paraphraser FMGR run string specifies an output file or LU, the paraphraser automatically stores the object code into the specified file or LU.

## 8-6. PARAPHRASER OUTPUT LISTINGS

The paraphraser prints the microprogram source program with line numbers in hexadecimal on the specified list device or disc file. The optional label list and floating field list are output to the same device or file if they are specified in the microprogram command statement. An error listing is also provided if there are any mistakes in the program.

For examples of source listings, refer to the sample program in Figure 7-1 or the sample programs in Section 12. The label listing provides each label in alphabetical order with the line number shown in hexadecimal. An example label listing is provided in Figure 8-1.

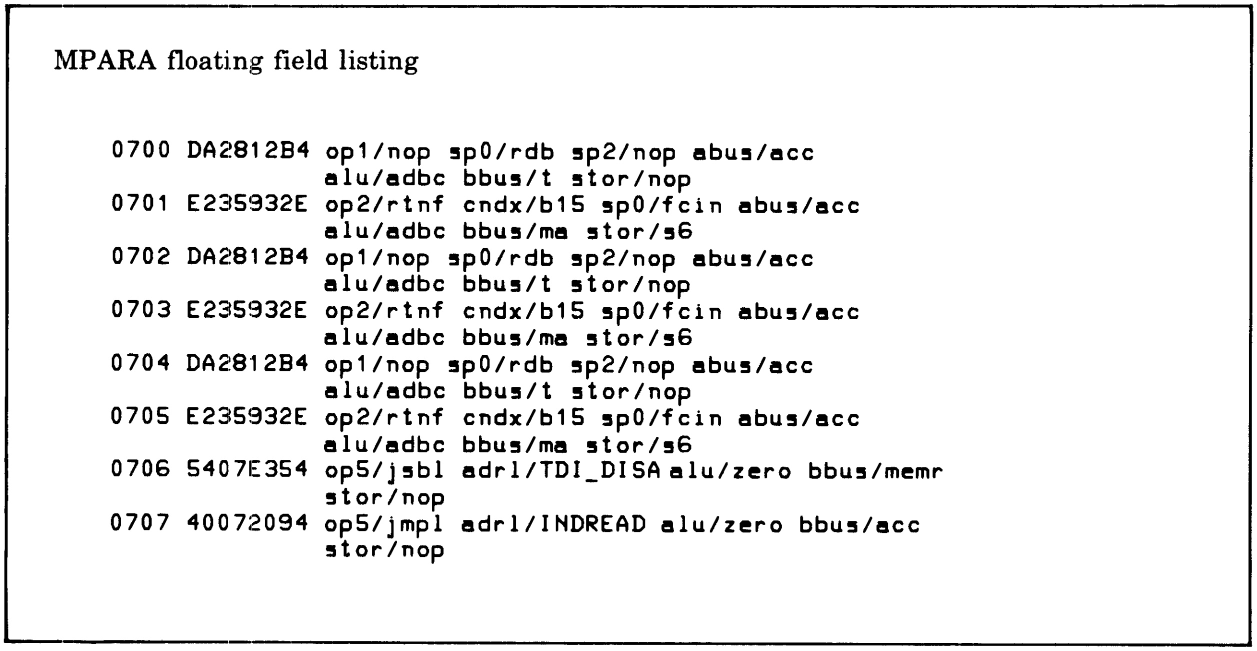


MPARA label listing

```
ALLOW_DI 0x0302
CHANG_OD 0x0430
DSBLE_TD 0x0731
ENBLE_TD 0x0742
INCLUSIV 0x0345
INDREAD  0x0335
INDRSOLV 0x0420
TDI_DISA 0x0515
TURN_OFF 0x0243
UNDO_LIS 0x02D0
ZFER_LP  0x0220
```

Figure 8-1. Example of a Paraphraser Label Listing

The floating field listing shows a microinstruction per line with the line number in hexadecimal and the 32-bit microcode given in hexadecimal. The op (operation) field is given first along with the word type which is determined by the microorder in the op field. Each field with its microorder is given together; for example, "op6/imm" for word type 6 and the microorder Immediate data in the op field. An example of a floating field listing is given in Figure 8-2.



MPARA floating field listing

```
0700 DA2812B4 op1/nop sp0/rdb sp2/nop abus/acc
              alu/adbc bbus/t stor/nop
0701 E235932E op2/rtnf cndx/b15 sp0/fcin abus/acc
              alu/adbc bbus/ma stor/s6
0702 DA2812B4 op1/nop sp0/rdb sp2/nop abus/acc
              alu/adbc bbus/t stor/nop
0703 E235932E op2/rtnf cndx/b15 sp0/fcin abus/acc
              alu/adbc bbus/ma stor/s6
0704 DA2812B4 op1/nop sp0/rdb sp2/nop abus/acc
              alu/adbc bbus/t stor/nop
0705 E235932E op2/rtnf cndx/b15 sp0/fcin abus/acc
              alu/adbc bbus/ma stor/s6
0706 5407E354 op5/jsbl adrl/TDI_DISA alu/zero bbus/memr
              stor/nop
0707 40072094 op5/jmpl adrl/INDREAD alu/zero bbus/acc
              stor/nop
```

Figure 8-2. Example of a Paraphraser Floating Field Listing

## 8-7. ERROR MESSAGES

Wherever your source file contains errors, MPARA will give a descriptive error message. There are two types of errors that MPARA detects:

1. Label errors are errors that MPARA detects while passing through the source file to determine the values of address labels that you want to define.

2. Translation errors are errors that MPARA detects while translating your sentences into microinstructions.

## 8-8. LABEL ERRORS

The address label pass errors are located in the listing file right after the control statement. The address label errors are described in Figure 8-3.

**BAD CONTROL STATEMENT**

**Message:** *** label pass error: bad control statement

**Reason:** First line in the source file did not start with "MPARA".

**BAD OPTION IN CONTROL STATEMENT**

**Message:** *** label pass error: bad option "Q"

**Reason:** User specified an invalid option in the control statement.

**DUPLICATE ADDRESS LABEL**

**Message:** *** label pass error: redefined label name: "DUPLICAT"

**Reason:** The user had defined the address label "DUPLICATE" twice. Labels may only be defined once.

**BAD DIRECTIVE**

**Message:** *** label pass error: missing $ in directive

**Reason:** MPARA expects all directives to begin and end with a dollar sign, and the label pass checks your directives. However, a more descriptive error is inserted in the listing file after the bad directive as a result of the translation.

**BAD COMMENT**

**Message:** *** label pass error: missing ending }

**Reason:** A bracketed comment was begun, but not ended before MPARA reached end of the file.

Figure 8-3. Address Label Errors

## 8-9. TRANSLATION ERRORS

Whenever MPARA detects a bad sentence during its translation of the input expressions, an error message is inserted right after the offending sentence and the rest of the sentence is skipped. MPARA translates your sentences in a left-to-right manner, so the left-most phrases, directives, etc will be checked first. Figure 8-4 describes the translation errors.

**BAD PHRASES**

**MISSPELLED WORD**

**Bad Phrase:** iff y15 goto there;

**Message:** *** error: unrecognized sentence due to "iff"

**Reason:** The "iff" in the source file has no meaning to MPARA. The programmer probably incorrectly typed "if."

**UNRECOGNIZED PHRASE**

**Bad Phrase:** A:=NON_EXISTENT_REGISTER;

**Message:** *** error: unrecognizable arithmetic phrase due to "NON_EXIS"

**Reason:** MPARA recognized the phrase as an arithmetic phrase, but NON_EXISTENT_REGISTER is something that MPARA does not understand. If the user had defined that name in the BBUS field using the "define" directive, then the sentence would have been recognized. In this case, MPARA found something it did not understand while processing an arithmetic phrase. Similar messages may be produced for errors in other types of phrases.

**MISSPELLED WORD**

**Bad Phrase:** if y15 gotto there;

**Message:** *** error: unrecognizable conditional phrase due to "gotto"

**Reason:** The "gotto" is "goto" miss typed.

**UNDEFINED ADDRESS LABEL**

**Bad Phrase:** goto nowhere;

**Message:** *** error: unrecognizable branch phrase due to "NOWHERE"

**Reason:** The address label "NOWHERE" was not defined in the source file. Address labels must be defined in the address field using the usual "NOWHERE:" method or using the define directive.

**TWO MICROORDERS IN ONE FIELD**

**Bad Phrase:** cmid, cmid;

**Message:** *** error: conflict in sp2 field due to "cmid"

**Reason:** The program has tried to fill the sp2 field more than once. Conflict may occur in any field if more than one microorder is specified for that field. Note that in cases where microorders are duplicated in separate fields, MPARA will choose the fields so that a conflict does not occur. An example is the increment p-register microorder (IP), which exists in the SP0, SP1 and SP2 fields. The sentence "cmid,ip;" will be interpreted as SP2/CMID and SP0/IP.

Figure 8-4. Translation Errors

### CONFLICTING WORD TYPES

**Bad Phrase:** if y15 goto 100, sp2/ip;

**Message:** *** error: word type conflict

**Reason:** The sentence specifies fields from different word types. Valid MPARA sentences must select microorders in fields within one word type. Note that if “if y15 goto 100, ip” had been specified, MPARA would have selected SP1/IP, and word type conflict would not have occurred.

### NON-VALID CHARACTER

**Bad Phrase:** if y15 goto there%;

**Message:** *** error: bad character or number due to “there%”

**Reason:** The percent character is not a valid MPARA character.

### NON-VALID NUMBER

**Bad Phrase:** s0:=100a;

**Message:** *** error: bad character or number due to “100a”;

**Reason:** The “100a” is not a valid number. This type of error is inserted in your list file whenever MPARA is scanning your sentences and finds bad characters or bad numbers. The following are some examples of bad numbers in sentences:

```
s0:=0x10G;  *bad hexadecimal number
s0:=0100B;  *bad octal number
s0:=100Z;   *bad decimal number
```

## BAD DIRECTIVES

### UNRECOGNIZED DIRECTIVE

**Directive:** $orgin 100$

**Message:** *** error: bad directive

**Reason:** The programmer probably meant to use the origin directive.

### BAD NUMBER

**Directive:** $origin 0AA$

**Message:** *** error: bad number in directive

**Reason:** The number inside of the directive is not represented correctly.

### INCORRECTLY TYPED NAME FIELD

**Directive:** $define adddr/test 100$

**Message:** *** error: field name is not defined

**Reason:** The user is trying to define microorder “test” in the “adrl” field but incorrectly typed the field name in the directive.

### MISSING $ SIGN

**Directive:** $origin 100 ;

**Message:** *** error: missing ending $

**Reason:** All directives begin and end with a dollar sign, and this directive is missing the ending dollar sign.

Figure 8-4. Translation Errors (Continued)

### MISCELLANEOUS

#### SENTENCE TOO LONG

**Message:** *** error: exceeded maximum sentence length

**Reason:** (No example given here.) MPARA's internal sentence holding buffer been exceeded. Note that all valid MPARA sentences will not exceed this buffer.

#### BRACKETED COMMENT NOT COMPLETE

**Message:** *** error: missing ending }

**Reason:** (No example given here) A bracket comment was begun, but a closing bracket was not encountered before the end of the file.

Figure 8-4. Translation Errors (Continued)

# SECTION 9

## WRITABLE CONTROL STORE (WCS) SUPPORT SOFTWARE

# WRITABLE CONTROL STORE (WCS) SUPPORT SOFTWARE

SECTION 9

The previous section (Section 8) describes a method of preparing a microprogram and storing this source program in a system file. The source program, prepared "off line" or on some other system, could have been stored in a system file by loading it through a system input device. The source program is then translated by the paraphraser (MPARA) microassembler program and filed as binary object code (or microcode) in another system file. This later file is the ready-to-use microinstructions of your program. In order to make use of this microcode it must be moved into the Control Store (micromachine memory) of the computer.

The computer's extended Control Store for user programs is provided by Writable Control Store (WCS) and PROM Control Store (PCS) cards. Normally, the microprogram is initially loaded into a WCS card so that test runs of the program can demonstrate that it has no "bugs" before burning PROMs to install on a PCS card.

The WCS cards are loaded by using a program called "WLOAD." WLOAD is a utility program which loads WCS and generates PROM "burn tape" code under the RTE operating system. An understanding of WCS memory mapping is essential for loading microprograms into it. This subject is summarized below. For additional information on the WCS card, which can be useful to the user for a better understanding of how to load it, refer to the HP 1000 A700 User Control Store Installation and Reference Manual, part no. 02137-90003. The WLOAD PROM "burn tape" function is covered in Section 10.

## 9-1. WCS MAPPING

The micromachine of the HP A700 computer has a microcode address space of 16k words of which the user may use 8k words. The 16k words are conceptually divided into 16 logical 1k modules numbered from 0 through 15. Each WCS card contains four banks of RAMs (Random Access Memory) for a total of 4k-words per card. The banks are numbered 0, 1, 2, and 3.

A mapping RAM on each card maps the logical modules to the physical banks. The map RAM has 16 locations each of which corresponds to a logical module. On each card, the logical module may be assigned a physical block that will be enabled when addressed through mapping, or it may be unmapped. If a logical module is mapped on more than one card at a time, the card which is higher priority in the control store chain will be enabled and will disable the other cards (including the processor control store).

## 9-2. USING WLOAD

To load a WCS card using WLOAD, the user assigns an LU to the card to identify its I/O location for program interaction. Next, the appropriate logical to physical mapping is set up, and then a file or files are usually specified from which to download data.

An example of running WLOAD to load a WCS card follows:

```
RU,WLOAD
```

NOTE

> The WLOAD prompt "xx>" will appear on your terminal where xx is the WCS LU which is currently specified. In this example, the WCS LU is specified by the user as 63 is the first step under WLOAD.

Continue with this procedure while running under WLOAD execution:

| **PROCEDURE** | **COMMENTS** |
|---|---|
| 0>LU,63 | WLOAD starts up with LU=0; User enters LU of WCS. |
| 63>IN | Initialize. Turn off WCS and unmap all logical modules. |
| 63>EQ,4,0 | User equates the logical module 4 address (1000-13FF hex) to physical bank 0. |
| 63>LB,%EXMPL | User loads microcode from the binary format file %EXMPL (example). |
| 63>ON | User turns WCS on. |
| 63>EX | Exit program. |

## 9-3. WLOAD COMMANDS

WLOAD commands are two characters. Some of the commands require parameters which may be included on the command line separated by commas. If required commands are not included on the command line, WLOAD will prompt for the parameters.

Commands which read from or write to either the data RAMs or the map RAMs require that WCS be turned off. If WCS was on when such a command is executed, WCS will be automatically turned off, and it will be turned back on after execution of the command, unless a WCS I/O error occurs.

Before turning WCS off or executing the command, the input parameters are checked for validity. The following checks are performed as applicable:

1. If the logical module is between 0 and 15.
2. If the physical bank is between 0 and 3.
3. If the WCS address or data is in hex format and the address <16k (4000).
4. If the input file (or LU) can be opened.
5. If the output file (or LU) can be opened or created.

## 9-4. ON COMMAND

This command turns WCS on. A check is done to make sure WCS actually turned on. The command format is:

```
ON
```

## 9-5. OF COMMAND

This command turns WCS off. A check is done to make sure WCS actually turned off. The command format is:

```
OF
```

## 9-6. EQUATE COMMAND

This command stands for "Equate (map) a logical module to a physical bank." The input parameters are checked for validity. The command format is:

EQ,*logical*,*physical*

where:

*logical* is a logical module number between 0 and 15;

*physical* is a physical bank number between 0 and 3.

All logical modules containing addresses referenced in the microcode of the input file must be mapped before loading.

## 9-7. REMOVE COMMAND

This command removes (unmaps) a logical module. The input parameter is checked for validity. The command format is:

RE,*logical*

where:

*logical* is the logical module between 0 and 15

## 9-8. STATUS COMMAND

This command displays current status (on/off) of WCS without altering it. The status is obtained from the WCS card. The format of the command is:

```
ST
```

## 9-9. LOAD ASCII COMMAND

This command loads WCS with ASCII-format data from a file or LU. The programmer must have previously mapped (using the EQ command) all logical modules containing addresses referenced in the microcode being loaded from the input file. The loading of WCS is actually an overlay of current data, so that any address locations not specified in the input file are not altered. This command is useful for overlaying "patches" onto WCS in order to change a few lines of microcode. The format of the command is:

LA,*input file* or *lu*

where:

*input file* is the file name where the microcode resides;

*lu* is the logical unit number of an input device if the microcode is to be input through that device.

The input file must contain one microinstruction per line consisting of the hexadecimal address followed by the hexidecimal data. This is similar to the floating field listing generated by the paraphraser.

A validity check is done on the input file or LU. In reading data from the input file, the other errors which may occur are:

a. Input file data is incomplete or incorrect format.

b. Address too high. (Address in input file exceeds the 16k logical address space).

c. Logical module not mapped. (An address was read for which the corresponding logical module was not mapped on the WCS card).

If any of these errors occur, WLOAD will stop execution of the command and the data in WCS will not be altered.

## 9-10. DISPLAY MAPPING COMMAND

This command displays the contents of the mapping RAM. The format of this command is:

DM

## 9-11. LU COMMAND

This command assigns the LU for interaction with WLOAD. A check is done to make sure that the LU specified is the correct interface type (41) and that the LU can be locked. The LU will remain locked until a new LU is specified or the program is exited. (Note that lu 0 is always a valid lu to specify). The format of this command is:

LU,*lu*

## 9-12. READ COMMAND

This command reads WCS data from the address range specified and outputs it to the file or logical unit (lu) specified. Default is the user's terminal. The start and end addresses are specified in hexadecimal and an error is reported if they are greater than 16k or if the end address is less then the start address. The output file will be created if it does not already exist. If an address in the range given is not mapped on the board, an error will be reported and WLOAD will stop executing the command. The format of this command is:

RD,*start address,* [*end address,output file or lu*]

where:

*start address* is a hexadecimal number of the lowest address in the range;

*end address* is a hexadecimal number of the highest address in the range (defaults to start address).

## 9-13. INITIALIZE COMMAND

This command initializes WCS. This turns WCS off and unmaps all logical modules. The format of this command is:

IN

## 9-14. BACKGROUND COMMAND

This command loads one microword of background data into every location in the specified logical module. A validity check is done on the input parameters, and an error is reported if the logical module specified is not mapped. The format of this command is:

BG,*logical module,hex data*

where:

*logical module* is a module number between 0 and 15 (see EQ command);

*hex data* is a 32-bit hexadecimal number for the background data.

## 9-15. WRITE DATA COMMAND

This command loads one location of WCS with the data specified. The input parameters are checked for validity. If the address specified is not logically mapped on the card, an error is reported. The format of this command is:

WD,*hex address,hex data*

where:

*hex address* is a hexadecimal number from 0 to 4000 (16k)

*hex data* is the data specified in hexadecimal

## 9-16. LOAD BINARY COMMAND

This command (Load Binary) loads WCS with binary format data from a file or lu. This is the standard object code output of the paraphraser. The programmer must have previously mapped all logical modules which contain addresses referenced in the input file. The loading of WCS is actually an overlay of current data, so that any address locations not specified in the input file are not altered. The format of this command is:

LB,*input file* or *lu*

where:

*input file* is the file name containing your microcode or lu is the input device (e.g., cartridge tape) containing the microcode.

A validity check is done on the input file or LU. In reading data from the input file, the other errors which may occur are:

a. Input file data is incomplete or incorrect format.

b. Address too high. (Address in input file exceeds the 16k logical address space).

c. Logical module not mapped. (An address was read for which the corresponding logical module was not mapped on th WCS card).

If any of these errors occur, WLOAD will stop execution of the command and will not alter WCS data.

## 9-17. TEST COMMAND

Performs a destructive test of all four physical banks of the WCS RAMs. This test writes patterns of alternating ones and zeros into the RAMs and then reads the data back out of the RAMs. The test is run first with a pattern starting with "0", then run again with a pattern starting with "1." All errors are listed in the output file. The total number of errors found in each physical bank is reported to the user's terminal. The format of this command is:

TE,*output file*

where:

output file is the namr of the output file used for listing any errors found during the test. If the error listing is not desired, then type "carriage return" in response to the prompt for output.

## 9-18. EXIT COMMAND

This command exits the WLOAD program. The format of this command is:

EX

## 9-19. TRANSFER FILE

This command transfers control to a file that contains the commands required for loading WCS with your microcode. Using a transfer file can save you time when microcode must be loaded more than once.

Commands will be read from the file and echoed to the user's terminal. If any error occurs while WLOAD is executing from a transfer file, control will be transferred back to the user's terminal and the transfer file will be closed. The transfer file command can not be used in another transfer file. The form of this command is:

*:namr*

where:

*namr* is the file name or lu

Transfer of control to a file can also be done by specifying the file as the first parameter in the run string:

RU,WLOAD,*namr*

## 9-20. RETURN FROM TRANSFER FILE

The command to transfers control from a transfer file back to the user's terminal is a double colon. The format of this command is:

::

The EX command can be used to simultaneously exit the WLOAD program and the transfer file. Any lines following an EX or :: in the transfer file will be ignored.

## 9-21. BT COMMAND

This is the Burn Tape command for the generation of PROM "burn tape" microcode. Use of this command is covered in Section 10.

## 9-22. HELP FILE

To display a list on your terminal of the WLOAD commands, call the HELP file. The format of the command to call this file is:

HE or ??

## 9-23. COMMENT

Any line that begins with an asterisk (*) is treated as a comment line and will be ignored by WLOAD:

*comment (anything that starts with *)

# SECTION 10
## WLOAD PROM BURN TAPE FUNCTION

# WLOAD PROM BURN TAPE FUNCTION

SECTION 10

The WLOAD PROM burn tape function translates your microprogram into binary code that is formatted for ROM firmware. The PROM burn binary code is generally stored in a computer file and then dumped onto HP cartridge tape (burn tape) for reading in an HP 264XX terminal which controls the PROM burning equipment. The fabricated ROMs are for installation on the HP 12155A PROM Control Store card or the HP 12156A Floating Point Processor. (Note: The PROM "burn tape" code can also be used for the A700 base set.)

Typical PROM burning equipment used for burning A700 processor PROMs are the DATA I/O System 19, the PRO-LOG PROM Burner or equivalent.

Before making PROM burn tapes, the microprogram should be completely tested and debugged using a WCS card for microprogram storage. The source should be corrected and microassembled using the paraphraser to provide MPARA output code from which the final PROM burn object code files are generated.

## 10-1. PCS PROM SPECIFICATIONS

The format of the microinstructions stored in the ROMs used on the PCS card is 32-bits divided by 8-bits. This format requires four ROMs per set. The PROMs can be the Signetics 825181, Harris HM 7681-5 or equivalent. The required PROM characteristics are as follows:

**PROM Characteristics**

| | |
|---|---|
| Size: | 1k x 8 bits |
| Address Access Time: | 70 nsec max, |
| Chip Enable Access Time: | 40 nsec max |
| Power Supply Current: | 175 mA max |

## 10-2. USING WLOAD FOR BURN TAPES

The WLOAD commands used to interact with WCS are covered in Section 9.

Specifying the LU of the WCS as 0 is a special case which the user may employ to generate PROM "burn tape" code, whether or not there is a WCS card in the system. When LU=0, any commands which involve reading or loading WCS (data or map RAMS) will be executed within program memory. In other words, LU 0 will look like a WCS card including maps and 4k of microcode space. The user can employ any WLOAD command to load or read the "WCS" of LU 0. (The ON Command will have no effect.)

An example of the procedure for running WLOAD to use program memory for burn tape microcode is the following:

| **PROCEDURE** | **COMMENTS** |
|---|---|
| 0> | User leaves LU=0 and loads his microcode into program memory. |
| 0>EQ,4,0 | User equates the logical module 4 address (1000-1400 hex) to physical bank 0. |
| 0>BG,4,00000000 | The user puts a background of all zeros into logical module 4. This is the unburned state of the user's PROMs. |
| 0>LB,%EX1 | User loads microcode from two different MARA object code files. |
| 0>LB,%EX2 | These microprograms reside at different places in the logical module 4. |
| 0>BT,P,1,4,FBURN | Generate the burn tape file FBURN for a 1k x 8 PCS PROM starting at logical module 4. |

## 10-3. BURN TAPE COMMAND

PROM burn tape code is generated by the WLOAD program when the Burn Tape Command is executed. This command can be used in two ways as follows:

a. When the LU specified in WLOAD is zero (0) the PROM burn code tape is generated from the microcode stored in program memory.

b. When the LU specified in WLOAD is an existing WCS card, the PROM burn code tape will be generated from the WCS card contents.

In either of the above cases the same sequence of commands are used before executing the BT command including setting up the map, writing background data, and loading microprogram files.

The PROM burn tape function of WLOAD generates code for the entire PROM so that all logical modules which fall into the address space of the PROM must be mapped. The user can optionally load either all zeros (0s) or all ones (1s) into unused areas of the PROM with Background (BG) command.

The format of the Burn Tape (BT) command is:

BT,*prom type,prom size,starting logical module,output file*

where:

*prom type* is either B = Base Set to reside on the lower processor card or,
P = PCS or FPP.

*prom size* is either 1 = 1k words
2 = 2k words
4 = 4k words

starting logical module = the logical module number between 0 and 15 at which the PROM is to start

*output file* = the file that the burn code will be stored into. If the input file does not exist it will be created.

## 10-4. ERROR REPORTING

The WLOAD program will report an error for any of the following conditions:

a. Output file cannot be opened or created.

b. Invalid input parameter for PROM type or size.

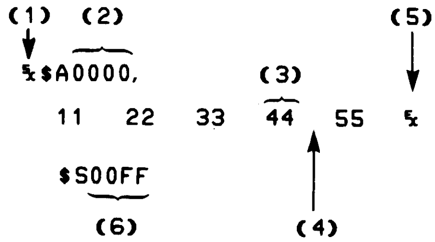
c. Starting logical module not valid or logical module plus PROM size exceeds 16k address space.

d. A logical module within the PROM address space is not mapped.

## 10-5. OUTPUT FORMAT

The translation format of the PROM burn code for five bytes of data output is shown in Figure 10-1. This is an ASCII-Hex(Space) format. The figure shows the data characters and the control characters annotated.

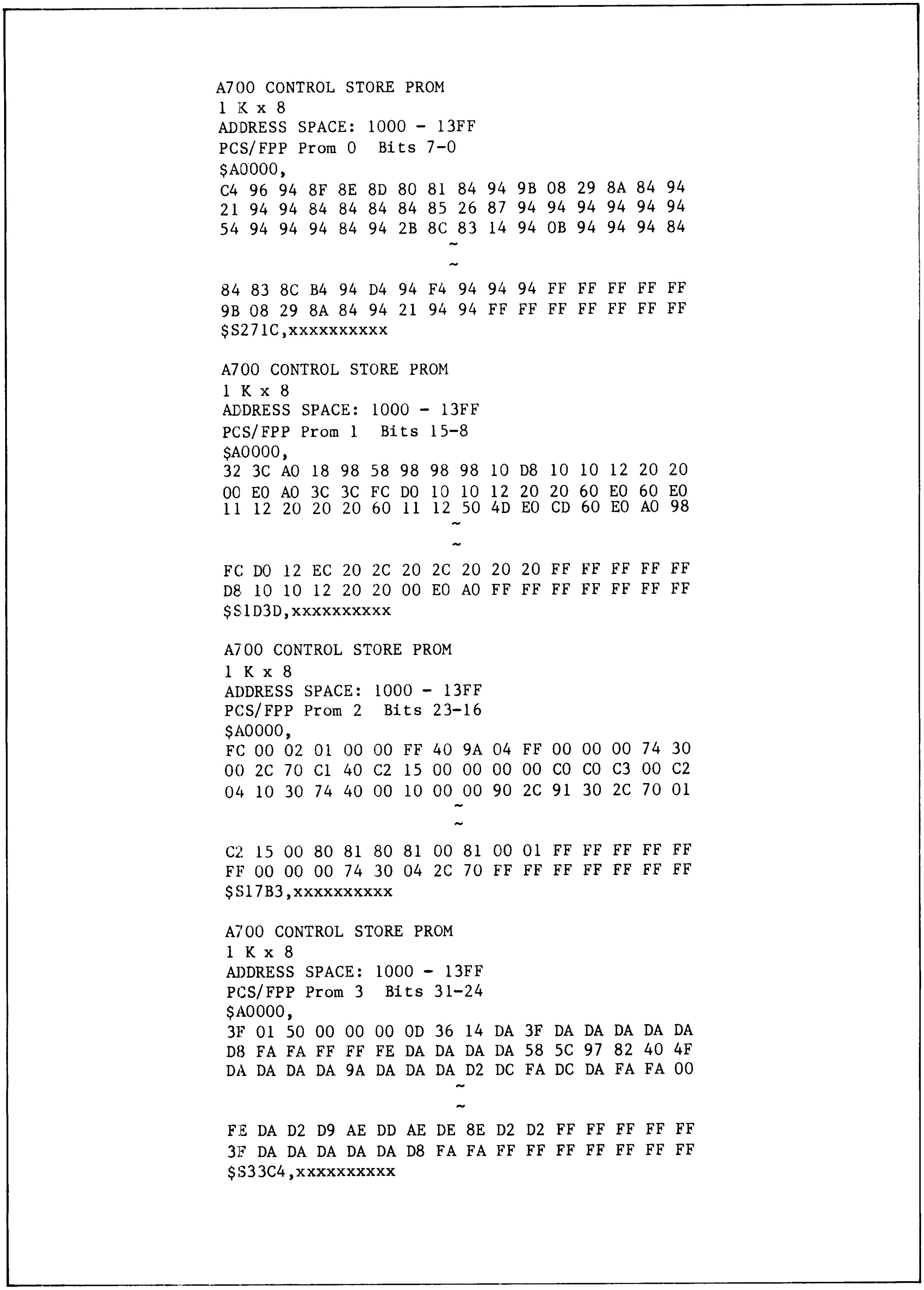
An example of a printout of the binary code output from a WLOAD PROM burn tape function pass is shown in Figure 10-2.



FORMAT IDENTIFICATION

1. Start Code ( % )
2. Address Field
3. Data Byte
4. Execute Code (a space)
5. End Code ( % )
6. Sum-Check Field

Figure 10-1. Translation Format of PROM Code



```
A700 CONTROL STORE PROM
1 K x 8
ADDRESS SPACE: 1000 - 13FF
PCS/FPP Prom 0  Bits 7-0
$A0000,
C4 96 94 8F 8E 8D 80 81 84 94 9B 08 29 8A 84 94
21 94 94 84 84 84 84 85 26 87 94 94 94 94 94 94
54 94 94 94 84 94 2B 8C 83 14 94 0B 94 94 94 84
                        ~
                        ~

84 83 8C B4 94 D4 94 F4 94 94 94 FF FF FF FF FF
9B 08 29 8A 84 94 21 94 94 FF FF FF FF FF FF FF
$S271C,xxxxxxxxxx

A700 CONTROL STORE PROM
1 K x 8
ADDRESS SPACE: 1000 - 13FF
PCS/FPP Prom 1  Bits 15-8
$A0000,
32 3C A0 18 98 58 98 98 98 10 D8 10 10 12 20 20
00 E0 A0 3C 3C FC D0 10 10 12 20 20 60 E0 60 E0
11 12 20 20 20 60 11 12 50 4D E0 CD 60 E0 A0 98
                        ~
                        ~

FC D0 12 EC 20 2C 20 2C 20 20 20 FF FF FF FF FF
D8 10 10 12 20 20 00 E0 A0 FF FF FF FF FF FF FF
$S1D3D,xxxxxxxxxx

A700 CONTROL STORE PROM
1 K x 8
ADDRESS SPACE: 1000 - 13FF
PCS/FPP Prom 2  Bits 23-16
$A0000,
FC 00 02 01 00 00 FF 40 9A 04 FF 00 00 00 74 30
00 2C 70 C1 40 C2 15 00 00 00 00 C0 C0 C3 00 C2
04 10 30 74 40 00 10 00 00 90 2C 91 30 2C 70 01
                        ~
                        ~

C2 15 00 80 81 80 81 00 81 00 01 FF FF FF FF FF
FF 00 00 00 74 30 04 2C 70 FF FF FF FF FF FF FF
$S17B3,xxxxxxxxxx

A700 CONTROL STORE PROM
1 K x 8
ADDRESS SPACE: 1000 - 13FF
PCS/FPP Prom 3  Bits 31-24
$A0000,
3F 01 50 00 00 00 0D 36 14 DA 3F DA DA DA DA DA
D8 FA FA FF FF FE DA DA DA DA 58 5C 97 82 40 4F
DA DA DA DA 9A DA DA DA D2 DC FA DC DA FA FA 00
                        ~
                        ~

FE DA D2 D9 AE DD AE DE 8E D2 D2 FF FF FF FF FF
3F DA DA DA DA DA D8 FA FA FF FF FF FF FF FF FF
$S33C4,xxxxxxxxxx
```

Figure 10-2. Example PROM Binary Code Printout

The first four lines of output are comment lines. These lines are not used by the PROM programmer. The comment lines appear as a header on each to the four subsections of binary code but the header will not output on the burn tape since there is no Start of Transmission ( ␂ ) symbol until the beginning of the PROM code in front of the address field (0000 in the example printout).

These lines provide the following information:

a. The size and organization of the PROM as specified by function parameters.

b. The address space in hexadecimal.

c. Whether the code is for a PCS, FPP, or Base Set PROM, which one it is of the set of four PROMs, and which bits of the 32 bits are contained in it.

## 10-6. PROM BURN OUTPUT

Each of the subfiles, marked by a zero-length record, can be output to cartridge tape as multiple files by using the file manager STore command and the SAve record format. This saves the files with the embedded EOF (End Of File) marks.

The store string is the following:

ST,*output file*,*tape lu*, SA

where

*output file* is the name of the disc file containing the binary code, and

*tape lu* is the logical unit number of the cartridge tape drive in the HP 264X terminal.

# SECTION 11
## FLOATING POINT PROCESSOR

# FLOATING POINT PROCESSOR

SECTION 11

This section contains the information required for writing microroutines to obtain high-speed floating point operations using the HP 12156A Floating Point Processor. It also provides a simplified operating description of the card, and microcode examples utilizing floating point operation.

For physical characteristics of the card and installation information, refer to HP 12156A Floating Point Processor Kit Installation and Reference Manual, part no. 12156-90001.

## 11-1. GENERAL DESCRIPTION

The HP 12156A Floating Point Processor (FPP) contains hardware to accelerate the execution of floating-point dependent macroinstructions and provide floating-point microprogramming capability. These macroinstructions include the basic single- and double-precision floating-point instructions (add, subtract multiply, divide, etc.), the Scientific Instructon Set (SIS), and the Vector Instruction Set (VIS). The FPP card contains four accumulator locations, ROM for storage of 512 arithmetic constants, 4k-words of microcode address space, and logic to interface to the processor. The FPP card plugs into the backplane between the upper processor and lower processor cards.

## 11-2. BASIC CAPABILITIES

The FPP interfaces to the computer processor over the frontplane. In operation, it accepts control words and operands, it performs operations on the operands, and returns the results to the processor. The operations performed by the card that can be microprogrammed are listed in Table 11-1.

Figure 11-1 shows a block diagram of the internal data paths on the FPP. The arithmetic functions are carried out by the Floating-Point Arithmetic Logic Unit (FPALU) which has A and B input ports and a D output port. The D output is buffered onto the B-Bus. Control signals come off the Y-bus.

Four accumulators are available as scratch memory. Each accumulator feeds either A or B input ports to the FPALU. Results from the D port can be transferred into any accumulator. Each accumulator can contain either a single or double precision number.

The Arithmetic Constant ROM (ACR) contains the constants required in floating point arithmetic sequences. The first constant in a sequence is selected by an address pointer, and the pointer is automatically incremented as the sequence progresses. The selected constants are loaded into the A-operand port of the FPALU.

Table 11-1. Microprogrammable Floating-Point Operations

| OPERATION | DESCRIPTION |
|---|---|
| add | single, double floating point |
| subtract | single, double floating point |
| multiply | single, double floating point,<br>double integer, 32 bit logical (unsigned) |
| divide | single, double floating point<br>double integer, 32 bit logical (unsigned) |
| fix | double floating point to single integer,<br>double floating point to double integer |
| float | single integer to double floating point,<br>double integer to double floating point,<br>double integer to single floating point,<br>double floating point to single floating point |
| convert | double floating point to single floating point |
| shift left | 0 to 63 bit left shifts |
| shift right | 0 to 63 bit right shifts |

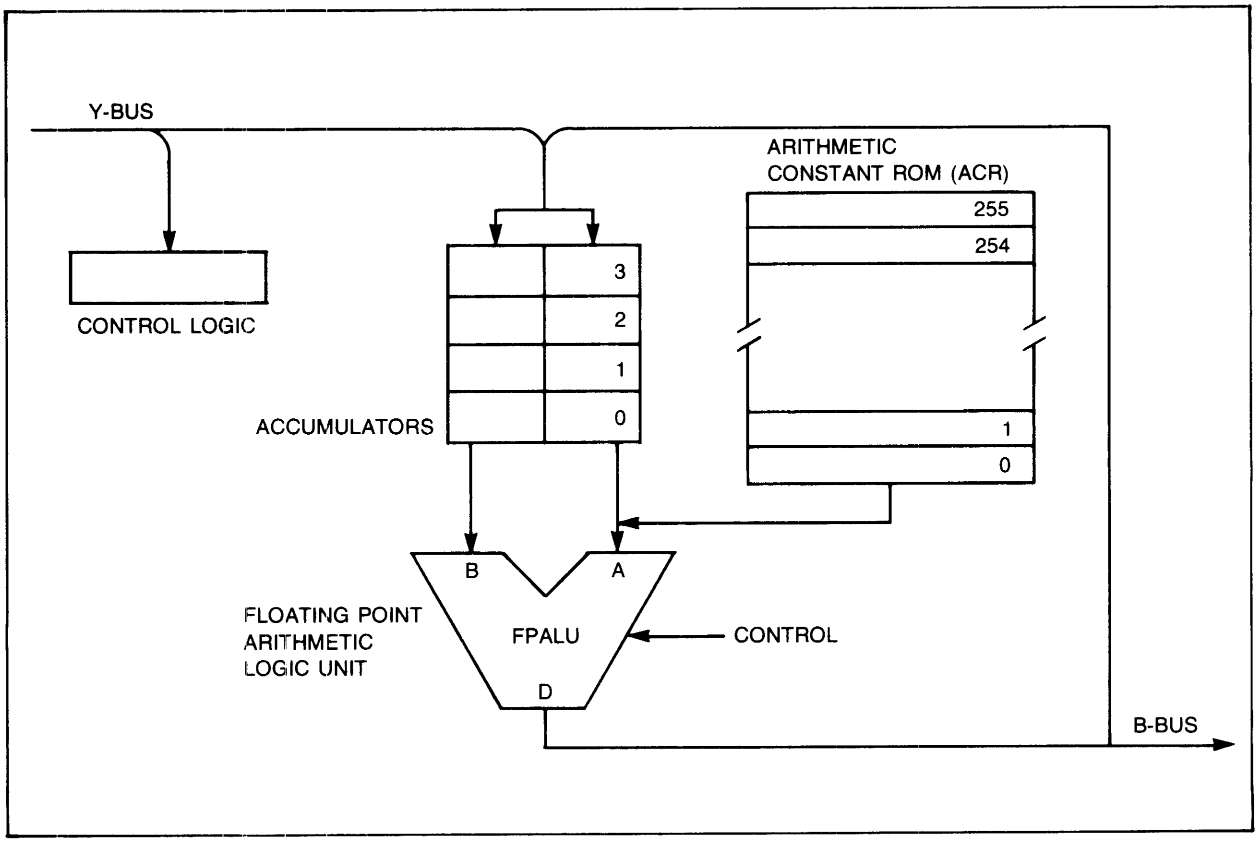


Figure 11-1. Floating-Point Processor Data Paths

## 11-3. GENERAL OPERATION

In operation, a control word is first passed to the FPP. This is done by storing to a reserved location in the special register file (SRIN). The card then interprets the information stored at this location as its control information and responds accordingly.

Once a control word is passed, the FPP accepts the necessary operands, and performs the requested function. After a required propagation delay, the result of the operation is either stored into one of the accumulators or returned to the processor over the B-Bus or both. An error condition is available to the processor which indicates whether an overflow or underflow has occurred during the operation.

## 11-4. INTERFACE TO THE MICROMACHINE

Communication between the micromachine and the FPP occurs through the upper four of the 16 Special-Purpose External Registers referenced by microorder SRIN. These are registers C through F (hexadecimal). The registers are selected indirectly through the N register (refer to Section 2 of this manual).

When SRIN is in the STORE field of a microinstruction and N is set to C (hex) through F (hex), the data present on the Y-Bus will be accepted by the FPP. Similarly when SRIN is in the B-Bus field and N is set to C (hex) through F (hex), the FPP will drive the B-Bus with the required information. The table below shows the assignment for the four SRIN locations:

| REGISTER | WRITE | READ |
|---|---|---|
| SRIN-F | DIVIDE CONTROL WORD | RESULT |
| SRIN-E | CONTROL WORD | RESULT, ALSO SAVED IN ACCUMULATOR |
| SRIN-D | A-SIDE OPERAND | ERROR CONDITION |
| SRIN-C | B-SIDE OPERAND | HP RESERVED |

## 11-5. SEQUENCE OF OPERATION

The FPP will operate in one of two general sequences. The first sequence is used for all floating point operations except division, and the second sequence is used for division.

Sequence for add, sub, mpy, fix, flt, shl, shr, cv:

1. Transfer a control word to the FPP.
2. Transfer the input operands to the FPP.
3. Wait three microcycles.
4. Transfer results to the destination.
5. Check the error condition if necessary.

Sequence of operation for division:

1. Transfer a control word to the FPP.
2. Transfer the input operand to the FPP.
3. Transfer the required divide control words.
4. Wait two microcycles.
5. Transfer the results to the destination.
6. Check the error condition if necessary.

## 11-6. TRANSFER OF CONTROL WORD

To start any operation, a control word must first be passed to the FPP. To make the transfer, the N register must first be set to E and SRIN must be in the STORE field of the microword. Data on the Y-Bus is then transferred to the FPP control logic. This control word contains the following information:

1. The operation to be performed by the card,
2. The source for the A-side operand,
3. The source for the B-side operand,
4. The destination for the result.

After the control word is transferred, the FPP will accept the operands from the designated source.

## 11-7. TRANSFER OF INPUT OPERANDS

After a control word has been passed to the FPP, the control logic determines which operands to accept from the Y-Bus and which operands are to come from an accumulator or ACR.

When both operands come from the micromachine over the Y-Bus, the A-side operand must be stored at location SRIN-D, and the B-side operand must be stored at location SRIN-C.

When only one operand is required from the micromachine, (the other operand coming from an accumulator or ACR) the operand must be passed through SRIN-D regardless of whether the operand is an A- or B-operand.

NOTE

> When operands are passed to the FPP they are always stored in an accumulator location. The location used to store the operands is determined by two fields in the control word. All operands are handled with the most significant word first.

When no operands are required from the micromachine, a special in the SP0 field is used to transfer the operands to the FPALU. One of the SP0 fields is dedicated for use in controlling the FPP card. The SP0 microorder CK2 is used to transfer data from the accumulators and/or ACR to the FPALU at two words (32-bits) per microcycle. This special is used only when no operands are required from the micromachine. A line of microcode containing CK2 in the SP0 field will transfer 2 words to both A and B ports of the FPALU. For a double precision operation, CK2 must be coded in two lines of microcode to transfer the four words of data.

The A-side operand can come from either the Y-Bus, one of four accumulator locations or from the ACR. The B-side operand can come only from the Y-Bus or one of the four accumulator locations.

## 11-8. TRANSFER OF RESULTS

When the FPP has completed the required operation the result is available to be accessed from SRIN-E or SRIN-F. The result is retrieved by having SRIN in the B-field of the microword and N set to E or F. The destination can be any accumulator location, register file location, or main memory location.

When the destination for the result is only the B-Bus (not stored in an accumulator) the result must be read from SRIN-F. When the destination is both the B-Bus and one of the four accumulators, the result must be read from SRIN-E. At the time of the read, the data is written into an accumulator and is available on the B-Bus.

Data can be transferred to an accumulator at double speed by also asserting the CK2 special in the SP0 field with the read from SRIN-E. During this double transfer only the first word is available on the B-Bus.

The following table shows all possible combinations for the destination of the result.

| | DESTINATION | | SP0 | OPERAND |
|---|---|---|---|---|
| READ FROM | ACCUMULATOR | B-BUS | FIELD | TRANSFER RATE |
| SRIN-F | | X | — | 1 Word per Microcycle |
| SRIN-E | X | X | — | 1 Word per Microcycle |
| SRIN-E | X | (X) | CK2 | 2 Words per Microcycle |

## 11-9. TRANSFER OF ERROR CONDITIONS

An error condition status word can be accessed after the required delay time either before or after the results have been retrieved. This error condition word is read at SRIN-D. The most significant bit of this word is read as a logic one if an overflow or an underflow has occurred during the last operation. The bit is cleared if no error has occurred.

# 11-10. WRITING MICROCODE FOR THE FPP

## 11-11. GENERAL FLOATING-POINT MICROCODE

For all floating point operations, the control word passed to the floating point card at SRIN-E contains the information in the fields shown in the diagram below.

CONTROL WORD AT SRIN-E

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| FUNCTION | | | | | | | A | | B | D-ADDR | | B-ADDR | | A-ADDR | |

## 11-12. FLOATING-POINT CONTROL-WORD FIELDS

The FUNCTION field translates to the seven-bit function opcode which is used by the FPP logic circuitry to decide what operation to perform. These functions are written as phrases in the paraphraser language. These phrases and the operations they perform are given in Table 11-2.

Table 11-2. Microprogrammable Floating-Point Functions

| PHRASE | OPERATION PERFORMED | ARITHMETIC EXPRESSION |
|---|---|---|
| add.f2 | single precision addition | D:=A+B |
| sub.f2 | single precision subtraction | D:=A-B |
| add.f4 | double precision addition | D:=A+B |
| sub.f4 | double precision subtraction | D:=A-B |
| shr.i4 | 0 to 63 right shift | D:=A shifted right B |
| shl.i4 | 0 to 63 left shift | D:=A shifted left B |
| ft.i1.f4 | float single integer to double precision floating | D:=A |
| ft.i2.f2 | float double integer to single precision floating | D:=A |
| ft.i2.f4 | float double integer to double precision floating | D:=A |
| cv.f4.f2 | convert double precision to single precision floating | D:=A |
| fx.f4.i1 | fix double precision floating to single integer | D:=A |
| fx.f4.i2 | fix double precision floating to double integer | D:=A |
| mul.i2 | double integer multiply | D:=A*B |
| mul.l2 | double logical multiply (unsigned) | D:=A*B |
| mul.f2 | single precision multiply | D:=A*B |
| mul.f4 | double precision multiply | D:=A*B |
| div.i2 | double integer divide | D:=A/B |
| div.l2 | double logical divide (unsigned) | D:=A/B |
| div.f2 | single precision divide | D:=A/B |
| div.f4 | double precision divide | D:=A/B |
| clear | clear opcode (default) | |

The A field of the control word is used to select the source for the A-side operand. This field encodes the three possible combinations of the source for the A-operand into a 2-bit field.

A-operand source:

| PHRASE | OPERATION |
|---|---|
| a__bus | A-side operand from Y-Bus (default) |
| a__acc | A-side operand from an accumulator |
| a__rom | A-side operand from ROM |

The B field is used to select the source for the B-side operand.

B-operand source:

| PHRASE | OPERATION |
|---|---|
| b__bus | B-side operand from Y-Bus (default) |
| b__acc | B-side operand from an accumulator |

The A-ADDR field is the address of the accumulator where the A-side operand is presently located, or where the A-side operand is to be stored when being passed over the Y-Bus to the FPP. If the A-side operand comes from ROM the contents of this field have no effect.

A-ADDR field:

| PHRASE | OPERATION |
|---|---|
| a0 | accumulator 0 (default) |
| a1 | accumulator 1 |
| a2 | accumulator 2 |
| a3 | accumulator 3 |

The B-ADDR field is the address of the accumulator where the B-side operand is presently located, or where the B-side operand is to be stored when passed to the FPP.

B-ADDR field:

| PHRASE | OPERATION |
|---|---|
| b0 | accumulator 0 (default) |
| b1 | accumulator 1 |
| b2 | accumulator 2 |
| b3 | accumulator 3 |

The D-ADDR field is the address of the accumulator where the result is to be stored.

D-ADDR field:

| PHRASE | OPERATION |
|---|---|
| d0 | accumulator 0 (default) |
| d1 | accumulator 1 |
| d2 | accumulator 2 |
| d3 | accumulator 3 |

## 11-13. PARAPHRASER FLOATING-POINT SENTENCE

The required information for an FPP control word is specified in the paraphraser by a floating point sentence initiated by "fp" followed by function phrases. The "fp" microinstruction specification is a symbol for the paraphraser that causes the specified information to be encoded in the 16-bit data field of a Word Type 6.

The general form for the paraphraser sentence is the following:

```
stor:=fp( function,            *what operation to perform
          a-operand source,    *source for the a-side operand
          b-operand source,    *source for the b-side operand
          a-operand address,   *accumulator address for a-side operand
          b-operand address,   *accumulator address for b-side operand
          d-result address);   *accumulator address for result
```

An example of writing floating-point microinstructions in the paraphraser language is shown below:

```
n:=0xE;                     *set n to 0xE
srin:=fp(add.f2,a_bus,      *add the operand coming over the y-bus
   b_acc,a0,b1,d2);         *to the contents of accumulator 1
dn;                         *set n to 0xD
srin:=a;                    *pass operand (a,b) to the FPP
srin:=b;
nop;                        *wait three cycles
nop;
  in;                     *set n to 0xE
x:=srin;                    *store the result in (x,y)
y:=srin;                    *result also is stored in accumulator 2
```

NOTE

> Each expression ending in a comma is a phrase, and each paraphraser sentence ends in a semicolon.

The example floating point sentence says: perform a single-precision floating-point addition (add.f2) on a number to be passed to the FPP over the y-bus (a__bus) and a number in an accumulator (b__acc). The number being passed over the y-bus is stored in accumulator zero (a0), and the number in the accumulator is at location one (b1). The result, if read from SRIN-E, will be stored in accumulator two (d2).

The paraphraser phrase "fp(. . .)" generates 16 bits of immediate data which can be stored to any register. This 16 bits of data must eventually be stored to SRIN-E to initiate an FPP operation. The list of fields within the parenthesis can be in any order or can be defaulted.

## 11-14. FLOATING POINT DIVISION

In floating point division the control word at SRIN-F has two primary uses. The first use is to provide the required control signal to the FPP during divide operations. The second use is to load the Arithmetic Constant ROM address pointer with the address of the next constant to be used.

To transfer the divide control word to the FPP, SRIN must be in the STOR field of the microword, and N must be set to F. The word stored to SRIN contains the information below.

**CONTROL WORD AT SRIN-F**

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| DIVIDE OPERATON | | | | | | | | ROM ADDRESS | | | | | | | |

The DIVIDE OPERATION field contains one of three function opcodes that is used by the floating point hardware during division.

Divide operation codes are the following:

| **PHRASE** | **OPERATION** |
|---|---|
| divsetup | prepare for division sequence |
| qbit3 | generate three bits of quotient |
| qbit2 | generate two bits of quotient |

During any division, a specific sequence of divide control words must be passed to the FPP to control the operation. These divide control words are passed to the FPP after the main control word and all operands have been transferred to the FPALU by methods previously described. After the last operand has been transferred to the FPALU, the following sequence of divide control words must be stored to SRIN-F:

| **PHRASE** | **OPERATION** |
|---|---|
| div.i2 | 1 divsetup, 11 qbit3, 1 qbit2 |
| div.l2 | 1 divsetup, 11 qbit3, 1 qbit2 |
| div.f2 | 1 divsetup, 9 qbit3 |
| div.f4 | 1 divsetup, 19 qbit3 |

After the last divide control word has been stored to SRIN-F, two wait states are required before the results are available.

The ROM ADDRESS field contains the data to be stored into the ROM address pointer register. This address pointer is always automatically incremented after each access allowing the next constant to be accessed. The ACR is used by the microcode which executes the SIS macroinstructions and is not intended for general use.

**11-15. PARAPHRASER DIVISION SENTENCE.** Floating point division information is specified in the paraphraser by a floating point sentence. The general form of the division sentence is the following:

```
stor:=fp(divide operation);       *divide operation to perform

        or

stor:=fp(rom_addr/label);         *address of ACR pointer
```

An example of a floating point divide operation is given below:

```
n:=0xE;                           *set n to 0xE
srin:=fp(div.f2,a_acc,            *divide the contents of accumulator
  b_acc,a0,b1,d1);                *zero by accumulator one
ck2;                              *clock in operands
in;                               *set n to 0xF
srin:=fp(divsetup);               *prepare to do divide sequence
srin:=fp(qbit3);                  *transfer nine qbit3 sequences
srin:=fp(qbit3);
srin:=fp(qbit3);
srin:=fp(qbit3);
srin:=fp(qbit3);
srin:=fp(qbit3);
srin:=fp(qbit3);
srin:=fp(qbit3);
srin:=fp(qbit3);
nop;                              *wait two cycles
nop, dn;                          *set n to 0xE
a:=srin;                          *place result in (a,b)
b:=srin;
```

## 11-16. CONTROL STORE STARTING ADDRESS

The capabilities of the FPP are available identically in all microcode to the 16k-words of control store of the computer.

The starting microaddress of the first block of control store located on the FPP is 0x1000 (hex). The length of the block is determined by the jumpers associated with that block, and it is either 2k or 4k words long. The starting address for the second block of control store on the FPP is switch selectable.

Each block also has one switch that will enable or disable the associated block of control store. When enabled, the control store of that block will respond to microaddresses within its range. When disabled, the control store will not respond.

All control store on the FPP has higher priority in the control store chain than the control store on the processor card, but has lower priority than any control store on the WCS or PCS cards.

The block of control store from 0x1180 to 0x11C0 x1000) has logic associated with it for overlaying a portion of the control store on the processor containing the section of the jump table which decodes the floating-point dependent macroinstructions that are executed from the FPP.

This feature is enabled and disabled by a switch on the FPP card. When disabled, this block of microcode will be executed from the processor control store; i.e., normal operation. When enabled, this block will be executed from the control store an the FPP, which disables the processor control store.

For information on setting the FPP switches and jumper installation refer to the HP 1000 A700 User Control Store Installation and Reference Manual, part no. 02137-90003.

# SECTION 12
## MICROPROGRAMS

# PART IV
# Microprogramming Examples

# MICROPROGRAMS

SECTION 12

The microprogram examples in this section illustrate the microprogramming concepts presented throughout the remainder of this manual. Each microprogram is complete in itself and can be used directly in the A700 processor or used as an example for creating your own microprogram. The following assumptions are made for the use of the material in this section.

- The program is to be run on an HP 1000 A700 computer system in which the software of the HP 92045A Microprogramming Package has been loaded into the RTE operating system.
- RTE system interface and device table entries (SC-LU relationship) must have been made.

The examples use the paraphraser microprogram language to prepare the source microprograms and generate object code. The source microprograms can be edited with the Edit/1000 editor. The object code should be tested using the HP 12153A WCS card in the control store of the processor

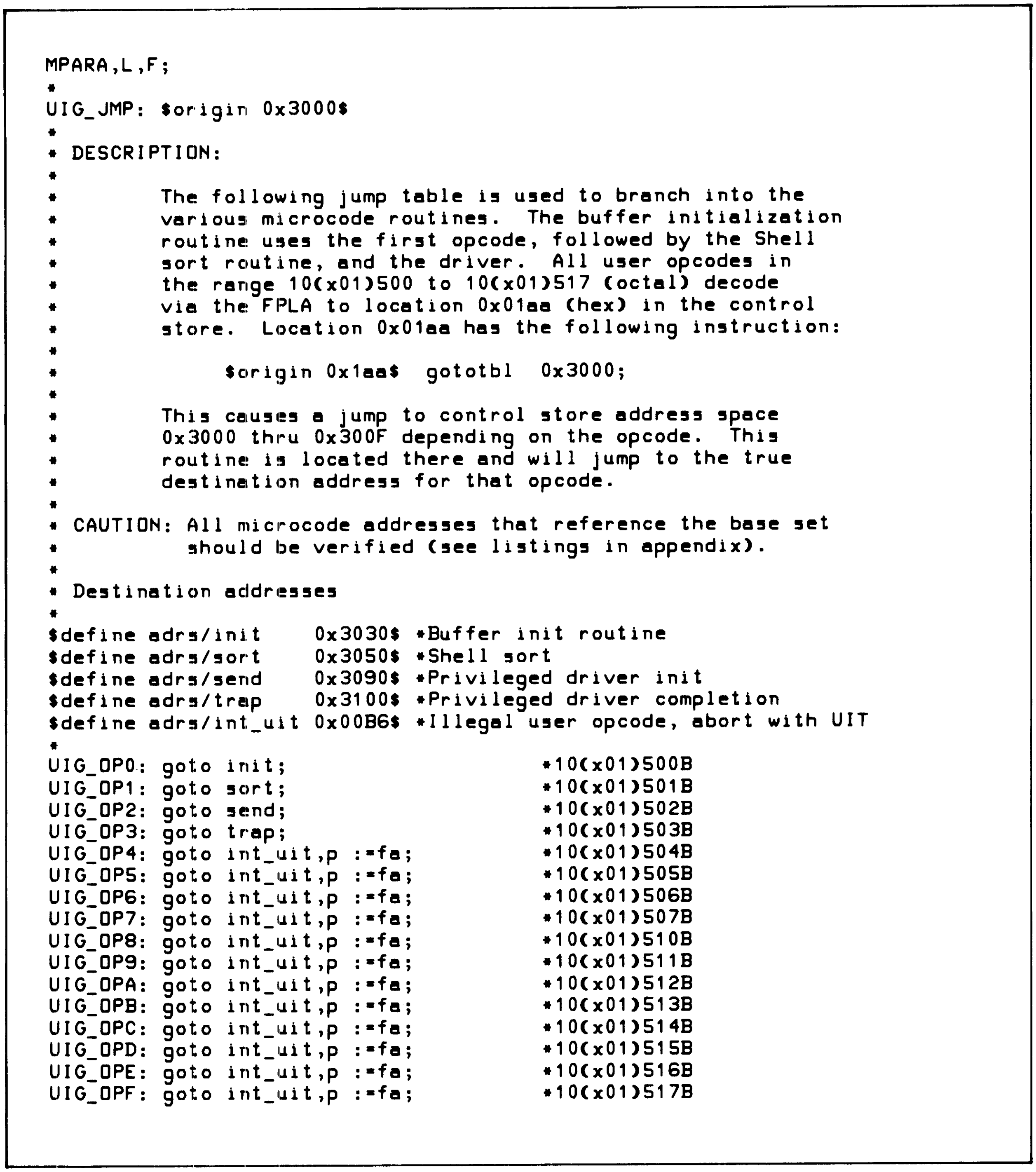
When you are ready to run the paraphraser from your disc source file, refer to the procedures in Section 8 for Using the Paraphraser. This section tells you the control commands to use and describes the output listing and error messages you may obtain from paraphraser execution. When you are ready to load your program into WCS, refer to Section 9 on Writeable Control Store Support Software.

There are three microprogram examples included in this section. They are a buffer initialization routine, a Shell sort, and a privileged driver. The order of complexity increases with each example. The privileged driver is very complex and should not be attempted before the microprogrammer has attended the Hewlett-Packard RTE microprogramming class for HP 1000 systems (product number 22964B). The microprogrammer should also have a good working knowlege of RTE-A.1 or RTE-XL operating system internals and HP 1000 driver writing.

All three of the example microcode routines were debugged using the HP 1610B Logic State Analyzer.

## 12-1. BRANCHING TO THE EXAMPLE PROGRAMS

A jump table (as described in Section 6) is used to branch into the three microroutines. The jump table is shown in Figure 12-1.



```
MPARA,L,F;
*
UIG_JMP: $origin 0x3000$
*
* DESCRIPTION:
*
*         The following jump table is used to branch into the
*         various microcode routines.  The buffer initialization
*         routine uses the first opcode, followed by the Shell
*         sort routine, and the driver.  All user opcodes in
*         the range 10(x01)500 to 10(x01)517 (octal) decode
*         via the FPLA to location 0x01aa (hex) in the control
*         store.  Location 0x01aa has the following instruction:
*
*              $origin 0x1aa$  gototbl  0x3000;
*
*         This causes a jump to control store address space
*         0x3000 thru 0x300F depending on the opcode.  This
*         routine is located there and will jump to the true
*         destination address for that opcode.
*
* CAUTION: All microcode addresses that reference the base set
*            should be verified (see listings in appendix).
*
* Destination addresses
*
$define adrs/init     0x3030$ *Buffer init routine
$define adrs/sort     0x3050$ *Shell sort
$define adrs/send     0x3090$ *Privileged driver init
$define adrs/trap     0x3100$ *Privileged driver completion
$define adrs/int_uit  0x00B6$ *Illegal user opcode, abort with UIT
*
UIG_OP0: goto init;                    *10(x01)500B
UIG_OP1: goto sort;                    *10(x01)501B
UIG_OP2: goto send;                    *10(x01)502B
UIG_OP3: goto trap;                    *10(x01)503B
UIG_OP4: goto int_uit,p :=fa;          *10(x01)504B
UIG_OP5: goto int_uit,p :=fa;          *10(x01)505B
UIG_OP6: goto int_uit,p :=fa;          *10(x01)506B
UIG_OP7: goto int_uit,p :=fa;          *10(x01)507B
UIG_OP8: goto int_uit,p :=fa;          *10(x01)510B
UIG_OP9: goto int_uit,p :=fa;          *10(x01)511B
UIG_OPA: goto int_uit,p :=fa;          *10(x01)512B
UIG_OPB: goto int_uit,p :=fa;          *10(x01)513B
UIG_OPC: goto int_uit,p :=fa;          *10(x01)514B
UIG_OPD: goto int_uit,p :=fa;          *10(x01)515B
UIG_OPE: goto int_uit,p :=fa;          *10(x01)516B
UIG_OPF: goto int_uit,p :=fa;          *10(x01)517B
```

Figure 12-1. Jump Table to Example Microprograms

## 12-2. BUFFER INITIALIZATION EXAMPLE

EXAMPLE 1: INITIALIZE BUFFER, FORTRAN PROGRAM

```
FTN7X,L,I,Y
      PROGRAM JOKE3
C
C MAIN PROGRAM:
C
C   Calls assembly routine 'INIT' to initialize a
C   user buffer.  'INIT' invokes the microcode.
C
C RUN STRING:  RUN,JOKE3,START,INC,NUMBR
C
C              START:  Starting value of buffer
C              INC:    Increment between values
C              NUMBR:  Total number of elements
C
      IMPLICIT INTEGER (A-Z)
      DIMENSION BUFF(10000)
      DIMENSION PARMS(5)
      EQUIVALENCE (PARMS(1),START), (PARMS(2),INC)
      EQUIVALENCE (PARMS(3),NUMBR)
C
C Get starting value, increment, and number of elements
C
      CALL RMPAR(PARMS)
      IF (IABS(NUMBR) .GT. 10000) GOTO 999
C
C Initialize the buffer
C
      CALL INIT(BUFF,START,INC,NUMBR)
C
C Print the buffer to scheduling lu
C
      SESN = -1
      LU = LOGLU(SESN)
      WRITE(LU,10)(BUFF(J),J=1,NUMBR)
10    FORMAT(8(2X,I6))
999   END
```

EXAMPLE 1: INITIALIZE BUFFER, ASSEMBLER INTERFACE

```
MACRO,L
      NAM INIT,7
*
* Calls the microcode to initialize the buffer passed
* by the calling program.
*
* CALLING SEQUENCE:
*
*         CALL INIT(BUFF,START,INC,NUMBR)
*
      ENT INIT
      EXT .ENTR,.INIT
*
* .INIT must be declared as an entry point in a seperate
* assembly module as follows:  .INIT  RPL  105500B
*
BUFF  BSS 1          BUFFER ADDRESS
START BSS 1          STARTING VALUE
INC   BSS 1          INCREMENTS
NUMBR BSS 1          NUMBER OF ELEMENTS
*
```

```
*
INIT  NOP
      JSB .ENTR                GET PARAMETERS
      DEF BUFF
*
* Branch to control Store Address (0x3000)
*
      JSB .INIT                USER OPCODE
      DEF @BUFF                BUFFER ADDRESS
      DEF @START               STARTING VALUE
      DEF @INC                 INCREMENT
      DEF @NUMBR               NUMBER OF ELEMENTS
*
      JMP @INIT                RETURN
      END INIT
```

## EXAMPLE 1: INITIALIZE BUFFER, MICROPROGRAM

```
MPARA,L,F;
*
UG_INIT: $origin 0x3030$
*
* DESCRIPTION:
*
*        Instruction 'INBUF' initializes a buffer in the user
*        program as specified by the calling program.  The
*        instruction is non-interruptable and does not check
*        for interrupts after every write to user memory.
*        The memory protect logic is enabled so that any memory
*        violation will be detected before the next instruction
*        is executed (ie. before the next JTAB).  Therefore,
*        memory is protected throughout the entire instruction.
*
* CALLING SEQ:
*
*        JSB .INIT
*        DEF BUFF  (,I)
*        DEF START (,I)
*        DEF INC   (,I)
*        DEF NUMBR (,i)
*
* Where: BUFF    is the user buffer
*        START   is the starting value to initialize
*                the buffer with (ie. buff(1)=start)
*        INC     the increment to the next buffer value
*                (ie. buff(2) = start+inc)
*        NUMBR   the number of words to initialize
*                (if numbr <0 use abs(numbr))
*
*
EXTRNL:  $define adrs inst_restart 0x00D0$
*
*
INT_BUF: rdp, ip;                             *read def buff
         call RSV_IND;                        *resolve indirects
*
* must complete read started by RSV_IND
* s7 has the direct buffer address on return
```

```
*
          nop := t,                       *dummy read
             ip, rdp;                     *read START
          call RSV_IND,                   *resolve indirects
             s6 := s7;                    *save def buff
          acc := t,                       *save START value
             ip, rdp;                     *read INC
          call RSV_IND;                   *resolve indirects
          s1 := t,                        *save INC value
             ip, rdp;                     *read NUMBR
          call RSV_IND;                   *resolve indirects
          s2 := p;                        *save return pc
          ct := t;                        *load ct w/ NUMBR
CONT:     if yz goto DONE;                *NUMBR=0?
          if not y15 goto POS_OK;         *NUMBR>0?
          ct := -t;                       *make pos
POS_OK:   p := s6,                        *p=def buff
             dct;                         *ajust count
*
* init buffer
*
NEXT:     wrp := acc, ip;                 *write to buffer
          acc := acc + s1,                *next data element
             if not ctz goto NEXT;        *
DONE:     p := s2;                        *restore pc
          fchp, rtn;                      *return
*
****************************************************
*                                                  *
*         Subroutine RSV_IND                       *
*                                                  *
****************************************************
*
*      RSV_IND used to resolve indirect
*      references by caller.  After 3 levels
*      of indirect, interrupts are checked.
*      Control is returned to the base set
*      if there is a pending interrupt.
*
* Calling parameters:
*
* On entry: Unresolved address must be in "T" register.
*
* On exit:  Data is returned in the "T" register.
*           User must pick it up!
*           The direct address is returned in the
*           scratch register s7.
*
*                                         *freeze ?
RSV_IND: s7 := t;                         *def to y bus
         if not y15 then rtn,             *1st level
           rdb, bbus/t;                   *data or def read
         s7 := t;                         *def to y bus
         if not y15 then rtn,             *2nd level
           rdb, bbus/t;                   *data or def read
         s7 := t;                         *def to y bus
         if y15 then goto CK_INT;         *allow interrupts
         rdb, bbus/t, rtn;                *otherwise read data
*                                         *and return
* Check for interrupt conditions
*
CK_INT:  if intp goto SER_INT;            *you lose
         rdb, bbus/t, goto RSV_IND;       *keep looking
```

```
*
* Jump back to Base Set to service interrupt
*
* Note that since we are checking interrupts within
* a subroutine, the code cannot just return to the
* base set, but MUST do a goto.
*
*
SER_INT: goto inst_restart,           *service interrupt
         p := fa;                     *original pc
```

## 12-3. SHELL SORT EXAMPLE

This example performs a sort of numeric data to illustrate the benefits of microprogramming a typical program that may be used repeatedly in a particular application. It includes three parts:

a. A FORTRAN program to generate an unsorted buffer, print the unsorted buffer, call a sort program, and print the sorted buffer.

b. An Assembly language program to interface to a microprogram which performs the actual sort.

c. The sort microprogram that uses a diminishing increment sorting algorithm to sort an array of integers into ascending order. The routine does check for interrupts and will return to the base set if necessary. This method is called a "Shell sort."

The calling sequence is as follows:

```
LDA NUMBR
LDB BUFF
CLE
JSB SORT
```

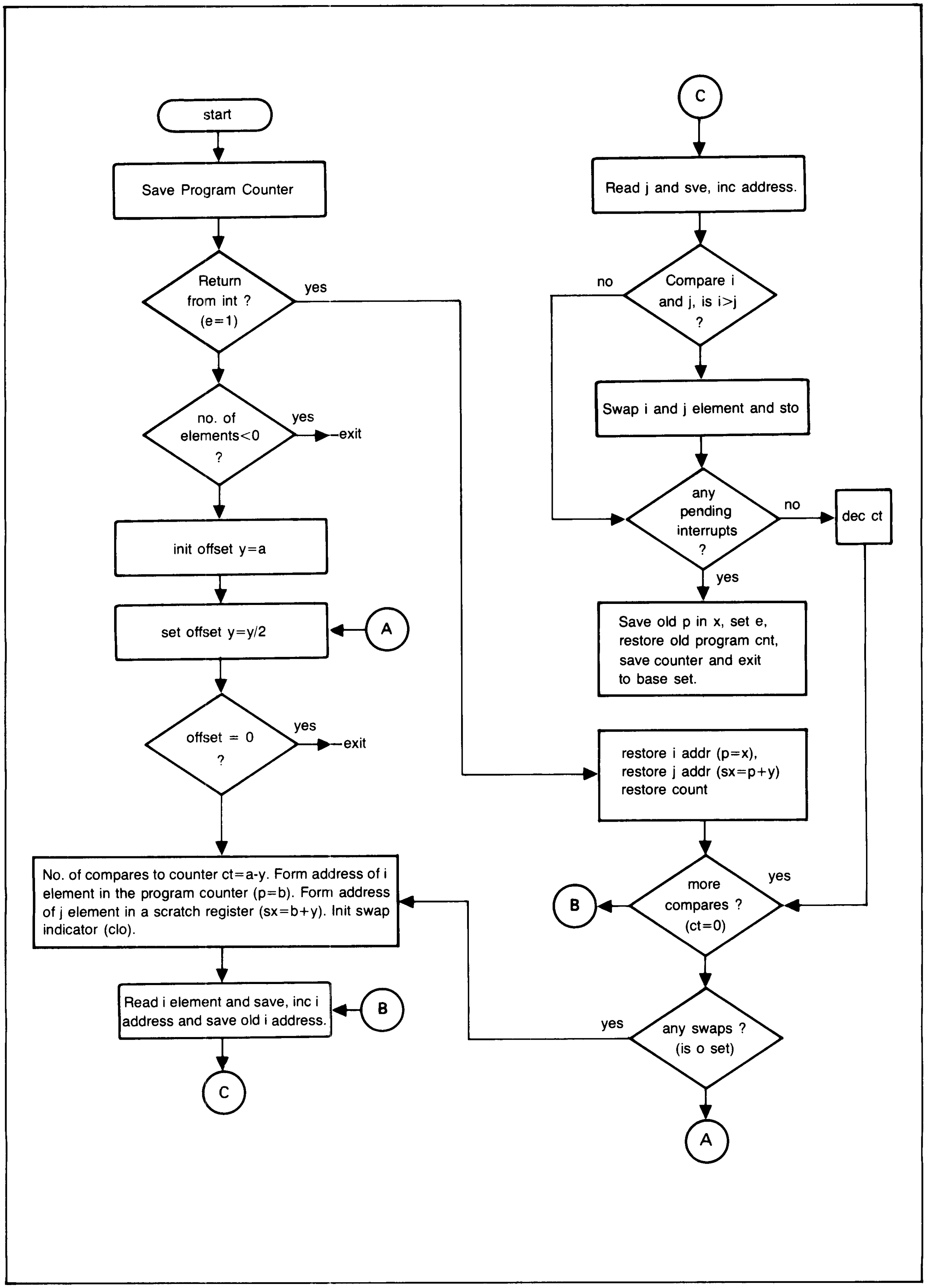
Where:

| | |
|---|---|
| NUMBR | is the number of array elements to sort |
| BUFF | is the starting address of the buffer |
| E reg | indicates first entry into SORT |

Register usage:

| | |
|---|---|
| A | number of elements (not modified) |
| B | address of buffer (not modified) |
| X | used to save interrupted address |
| Y | used to hold current partition increment |
| O | used as a swap flag |
| E | used as a reenterant flag |

The flowchart of Figure 12-2 is provided to help you follow the program code in the shell sort example.


start
Save Program Counter
Return from int ? (e=1)
yes
no. of elements<0 ?
yes
exit
init offset y=a
set offset y=y/2
A
offset = 0 ?
yes
exit
No. of compares to counter ct=a-y. Form address of i element in the program counter (p=b). Form address of j element in a scratch register (sx=b+y). Init swap indicator (clo).
Read i element and save, inc i address and save old i address.
B
C
C
Read j and sve, inc address.
no
Compare i and j, is i>j ?
Swap i and j element and sto
any pending interrupts ?
no
dec ct
yes
Save old p in x, set e, restore old program cnt, save counter and exit to base set.
restore i addr (p=x), restore j addr (sx=p+y) restore count
B
more compares ? (ct=0)
yes
yes
any swaps ? (is o set)
A

8200-2

Figure 12-2. Flowchart of Shell Sort Program

## EXAMPLE 2: SHELL SORT, FORTRAN TEST PROGRAM

```
FTN7X,L,I,Y
      PROGRAM JOKE4
C
C MAIN PROGRAM:
C
C   Calls Subroutine 'INIT' to initialize a user
C   buffer (ie. unsorted buffer).
C
C   Then subroutine 'SORT' is called to sort the
C   data buffer.  'SORT' calls the microcode.
C
C RUN STRING:  RU,JOKE4,START,INC,NUMBR
C
C              START:  STARTING VALUE
C              INC:    INCREMENT BETWEEN VALUES
C              NUMBR:  TOTAL NUMBER OF ELEMENTS
C
      IMPLICIT INTEGER (A-Z)
      DIMENSION BUFF(10000)
      DIMENSION PARMS(5)
      EQUIVALENCE (PARMS(1),START), (PARMS(2),INC)
      EQUIVALENCE (PARMS(3),NUMBR)
C
C Get starting value, increment, and number of elements
C
      CALL RMPAR(PARMS)
      IF (IABS(NUMBR) .GT. 10000) GOTO 999
C
C Initialize the buffer per run string parameters
C
      CALL INIT(BUFF,START,INC,NUMBR)
C
C Print the unsorted buffer to scheduling lu
C
      SESN = -1
      LU = LOGLU(SESN)
      WRITE(LU,1)
1     FORMAT(/20X,"UNSORTED BUFFER"/)
      WRITE(LU,8)(BUFF(J),J=1,NUMBR)
8     FORMAT(8(2X,I6))
C
C Sort the buffer
C
      CALL SORT(BUFF,NUMBR)
C
C Print the sorted buffer
C
      WRITE(LU,15)
15    FORMAT(//20X,"SORTED BUFFER"/)
      WRITE(LU,18)(BUFF(J),J=1,NUMBR)
18    FORMAT(8(2X,I6))
999   END
```

## EXAMPLE 2: SHELL SORT: TEST ASSEMBLER INTERFACE

```
MACRO,L
      NAM SORT,7
*
* Calls the microcode to sort a buffer passed
* by the calling program.
*
* CALLING SEQUENCE:
*
*       CALL SORT(BUFF,NUMBR)
*
      ENT SORT
      EXT .ENTR,.SORT
*
* .SORT must be declared as an entry point in another
* assembly module as follows:   .SORT   RPL   105501B
*
BUFF  BSS 1                   BUFFER ADDRESS
NUMBR BSS 1                   NUMBER OF ELEMENTS
*
*
SORT  NOP
      JSB .ENTR               GET PARAMETERS
      DEF BUFF
*
* Branch to control store address (0x3001)
*
      LDA @NUMBR
      LDB BUFF
      CLE
*
* B=DEF BUFF    A=#ELEMENTS   E=0
*
      JSB .SORT
      JMP @SORT
      END SORT
```

## EXAMPLE 2: SHELL SORT, MICROPROGRAM

```
MPARA,L;
*
UG_SORT: $origin 0x3050$
*
* DESCRIPTION:
*
*        Instruction 'SORT' uses a diminishing increment sorting
*        algorthim to sort an array of integers into ascending
*        order.  'SORT' is interruptable and therefore does
*        check for memory protect on every memory read and write.
*        Memory protect checks are implicit on the A700 CPU
*        if the logic is enabled.  (This is different from the
*        microcode in Hewlett-Packard M/E/F Series computers.)
*
* CALLING SEQ:
*
*        LDA NUMBR
*        LDB BUFF
*        CLE
*        JSB .SORT
*
* Where: NUMBR   is the number of array elements to sort
*        BUFF    is the starting address of the buffer
*        E reg   indicates first entry into SORT
*
```

EXAMPLE 2: SHELL SORT, MICROPROGRAM (Continued)

```
*
*  Register usage:     A   number of elements (not modified)
*                      B   address of buffer (not modified)
*                      X   used to save interrupted address
*                      Y   used to hold current partition
*                          increment
*                      O   used as a swap flag
*                      E   used as a reenterant flag
*
SORT:     s7 := p;                          *save program counter
             if e then goto INT_RTN;        *re-entering?
*
* Initial program entry
*
          y := a;                           *save no. elements
          if y15 then goto EXIT,            *neg no. ?, if so goodbye
             acc := ones;                   *set acc = -1
*
* Calculate current partition
*
SETY:     y := lr1(y);                      *y=y/2
          if yz then goto EXIT;             *all done?
*
* Start sort with current partition offset
*
ST_PASS: ct := a-y;                         *calc loop count
          p := b,                           *p=i element addr
             dct;                           *dec counter once
          s6 := b+y,                        *s6=j element addr
             clo;                           *init swap flag
*
* Compare elements i and j
*
CMPAR:    rdp,ip,                           *read i element
             s5 := p;                       *save old i addr
          s4 := t;                          *save i value
          rdb,                              *read j element
             s6 := s6 - acc;                *inc j addr
          s3 := t;                          *save j value
          s2 := s4 xor s3;                  *like signs ?
          if not y15 then goto SUB_UM,      *yes, subtract um
             s4 := s4;                      *test neg i
          if y15 then goto CK_INT;          *if so, i<j
          goto SWAP;                        *j>i
SUB_UM:   s1 := s3-s4;                      *j-i>0 ?
          if not y15 goto CK_INT;           *if yes, no swap
*
* Swap the elements
*
SWAP:     wrb := s4, bbus/ma,               *old j adr = i value
             sto;                           *indicate swap
          wrb := s3, bbus/s5;               *old i adr = j value
*
* Check for interrupts (ie. MP, Parity, TBG, etc.)
*
CK_INT:   if not intp goto END_CK,          *any interrupts?
*
* Interrupts pending so exit as follows:
*
*     Save the next i address (p) in the x register
*     Save the counter in the reserved user register
*     Restore the original program counter
*     Set e to indicate interrupt entry when we return
*     Return to base set
```

```
*
INT_EX:  x := p,                        *save i addr
            ste;                        *set int flag
         usr := ct;                     *save count
         p := fa, fchb,                 *restore old pc
            rtn;                        *and return
*
* Return from interrupt section:
*
*      Restore i address to p
*      Restore j address to s6
*      Accumulator to -1
*      Restore count
*
INT_RTN: p:=x;                          *current i addr
         ct := usr;                     *restore count
         s6 := p+y;                     *current j addr
         acc := ones;                   *acc = -1
*
* Check for last compare in current partition
*
END_CK:  if not ctz goto CMPAR;         *more compares ?
         if o goto ST_PASS;             *any swaps ?
         goto SETY;                     *get next increment
*
* Start Instruction Fetch, Exit
*
EXIT:    p := s7, fchb,                 *restore pc
           rtn;                         *return
```

## 12-4. PRIVILEGED DRIVER EXAMPLE

An I/O driver has an initiation section and a continuation/completion section. For any given I/O request, the operating system is involved with the driver in the operations of both these sections.

The initial EXEC I/O request call forces an entry into the operating system. The operating system verifies the EXEC request parameters and builds an I/O request block from the user's parameters and performs other system checks. The I/O request block is passed onto the driver to carry out the I/O operation. The operating system performs these steps in about 1 or 2 milliseconds.

There are some high speed, real time applications that require the user program to immediately service an I/O device without waiting for the operating system initial functions. Therefore, the programmer needs a way to "bypass" the operating system set-up and error checking. This can be accomplished by having the user program perform the driver initiation functions in a "privileged" routine. The required time can be further reduced by microcoding the initiation routine. A privileged microcoded initiation section driver is more than ten times faster than the normal driver that uses the operating system; however, the operating system protection is sacrificed.

After the I/O card has either completed the data transfer or has completed a portion of the data transfer, the I/O card can be programmed to generate an interrupt. The interrupt starts the continuation section by forcing an entry into the operating system. The operating system must save the current machine state, determine the cause of the interrupt, and enter the completion section of the appropriate driver.

The operating system overhead in the continuation/completion portion of the driver can also be eliminated by writing a privileged driver which can be further improved by microcoding. The privileged driver continuation routine is entered directly from the trap cell when the card interrupt occurs (i,e., bypassing the normal entry into the operating system). Therefore, the driver is responsible for saving the interrupted machine state on entry, performing the necessary I/O, and restoring the interrupted machine state upon exit.

## 12-5. STRUCTURE OF EXAMPLE PROGRAM

The example privileged microprogrammed driver is divided into two parts: an initiation routine called "SEND" and a continuation/completion routine called "TRAP". The "SEND" routine is entered directly from the user program, and it is responsible for outputting a 16-bit data word and a 16-bit control word to the HP 12006A Parallel Interface Card (PIC).

The "SEND" routine outputs the data to the I/O card, programs the card to interrupt, and returns to the user program. When the card's flag is set, the interrupt occurs. The interrupt causes the execution of the trap cell associated with the select code of the PIC card (30B). Normally, the trap cell contains a JSB indirect instruction into the operating system. For the privileged driver "TRAP," the user opcode is in the trap cell. Therefore, when the interrupt occurs, the "TRAP" microcode is entered directly from the trap cell. The "TRAP" routine is responsible for saving the state of the machine, completing the I/O transfer, and restoring the state of the machine upon exit.

The calling sequence for the "SEND" routine is shown below:

Calling Sequence:

```
JSB .SEND
DEF RTN         Return address
OCT SCODE       select code of PIC card
OCT DVADR       data for R31 of PIC
OCT DATA        data for R30 of PIC
OTB 2B,C        I/O instruction to load global register
OTA 31B         I/O instruction to load R31
OTA 30B         I/O instruction to load R30
STC 30B,C       I/O instruction to send "DVCMD"
LIB 2B          I/O instruction to save global register
OTB 2B,C        I/O instruction to load global register
CLC 30B,C       I/O instruction to disable device
OTB 2B,C        I/O instruction to reset global register
```

The A700 processor I/O architecture allows the I/O cards to monitor instruction fetches and execute the I/O instructions that match the I/O card's select code. Therefore, the microcoded driver must fetch the I/O instructions from memory and broadcast the I/O instructions over the backplane for reading by the I/O cards. (This is quite different from the Hewlett-Packard M/E/F computer line, where the I/O instructions are generated internally in the microcode.) This is why the instruction ".SEND" passes the I/O instructions as parameters to the microcoded driver, so that the driver can "broadcast" these instructions over the backplane.

Once the instruction is broadcast to the I/O cards, the microcode must assist the I/O card in executing the I/O instruction if processor resources are required. The I/O card is responsible for executing the instruction and the microcode acts as a slave processor to the I/O card during the execution of the instruction.

EXAMPLE 3: PRIVILEGED DRVER, MAIN PROGRAM

```
MACRO,L
      NAM JOKE6,3
*
* MAIN PROGRAM:
*
* Calls the microcoded driver "SEND" to output a 16-bit data word
* to register 30 and a 16-bit control word to register 31 of the
* parallel interface card (12006A).
*
* "SEND" programs the card to interrupt after accepting the  data
* (ie. when the device's flag is set) and returns to the the user
* program.
*
* On interrupt, the base set microcode executes the trap cell. The
* trap cell contains a user opcode that branches to the microcoded
* completion driver called "TRAP".
*
* CAUTION:  This is the ONLY program that can access the PIC
*           card because the operating system's protection
*           has been bypassed.
*
      ENT JOKE6
      EXT EXEC,$LIBR,$LIBX,.XSA1,.XLA1,$PIMK,.SEND
*
*  .SEND must be declared as an external in another assembly module
*  as follows:  .SEND  RPL  105502B
*
*  Disable memory protect by going privileged so that the trap cell
*  for the PIC card can be modified without causing a memory protect
*  violation.
*
*
JOKE6 JSB $LIBR      *GO PRIVILEGED
      NOP            *
      LDA OPCOD      *PATCH IN TRAP CELL FOR
      JSB .XSA1      *MICROCODED DRIVER "TRAP"
      @SCODE         *
*
*  The PIC card is set up to be a privileged interface. This means
*  that an interrupt from the PIC card could interrupt either a
*  user program or the operating system. Non-privileged I/O cards
*  can only interrupt a user program or the idle loop of the
*  operating system.
*
*  Note: In Hewlett-Packard M/E/F Series computers, a card is
*  privileged if it is physically  placed beneath a privileged fence
*  card and the generation specifies that the system is privileged.
*
*  In HP A700-Series computers, a privileged mask register is
*  maintained on every I/O card that informs the I/O card
*  whether or not it can interrupt.  The operating
*  system maintains the value of the privileged mask (the default
*  value is determined at generation time).  The value is output
*  to the I/O cards with an OTA 0B command.
```

EXAMPLE 3: PRIVILEGED DRIVER, MAIN PROGRAM (Continued)

```
*
*  The format of the privilege mask is shown below:
*
*        ----------------------------
*                                st
*        ----------------------------
*        15                         0
*
*          CARD INTERRUPT DISABLED
*  BIT       WHEN BIT IS SET
*
*   15        scodes 77-74
*   14        scodes 73-70
*   13        scodes 67-64
*   12        scodes 63-60
*   11        scodes 57-54
*   10        scodes 53-50
*    9        scodes 47-44
*    8        scodes 43-40
*    7        scodes 37-34
*    6        scodes 33-30
*    5        scodes 27-24
*    4        scodes 23-20
*    3        scodes 17-14  reserved for CPU
*    2        scodes 13-10  reserved for CPU
*    1        tbg
*    0        status bit
*
      JSB .XLA1      *GET DEFAULT PRIVILEGE MASK
      DEF $PIMK      *
      AND =B177677   *CLEAR BIT FOR SCODE 30-33
      JSB .XSA1      *RESTORE IT
      DEF $PIMK
      JSB $LIBX      *TURN MEMORY PROTECT
      DEF *+1        *BACK ON
      DEF *+1        *
*
*  The initiation of the I/O is started by the "SEND" microcode
*  and is completed by the "TRAP" microcode. The "TRAP" microcode
*  communicates with this program, so we must make sure that this
*  program stays in memory in the same partition.  The best way
*  to do this is for the System Manager to "assign" this program
*  to a fixed partition at boot-up time. Also, the program should
*  lock itself into memory with an exec 22 call. These precautions
*  are needed because we are bypassing the operating system's
*  internal protection and set up.
      JSB EXEC       *LOCK IN PARTITION
      DEF *+3
      DEF D22
      DEF LOCK
*
*  Output the data to the PIC card
*
*  Transfer times:
*
*  Start of "SEND" to completion of "SEND" - 76.8 us
*  Base set code to handle interrupt       - 11.1 us
*  Start of "TRAP" to completion of "TRAP" - 16.0 us
*
*  Total transfer time about 105 us.
*
*  NOTE: These times were done on pre-released hardware and base
*        set.  Actual times will be about 8 percent faster.
*        The PIC card was installed with a loop-back hood.
```

## EXAMPLE 3: PRIVILEGED DRIVER, MAIN PROGRAM (Continued)

```
*
      CLE           *INDICATE FIRST ENTRY
      JSB .SEND     *ENTER LAB6 MICROCODE
      DEF NEXT      *RETURN
SCODE OCT 30        *PIC SELECT CODE
R31   OCT 45        *R31 DATA
R30   OCT 155       *R30 DATA
      OTB 2B,C      *SET UP GLOBAL
      OTA 31B       *DATA TO R31
      OTA 30B       *DATA TO R30
      STC 30B,C     *START TRANSFER
*
* The following instructions are used by the
* trap cell driver code.
*
      LIB 2B        *SAVE GLOBAL CAUSE PRIVILEGED
      OTB 2B,C      *SET GLOBAL FOR PIC
      CLC 30B,C     *DISABLE PIC
      OTB 2B,C      *RESTORE GLOBAL
*
* Do another transfer, this time indicate that the set up
* code has already been completed.
*
*  Transfer times:
*
*  Start of "SEND" to completion of "SEND" - 22.4 us
*  Base set code to handle interrupt       - 11.1 us
*  Start of "TRAP" to completion of "TRAP" - 16.0 us
*
*  Total transfer time approx. 50 us
*  NOTE: These times were done on pre-released hardware and
*  base set. Actual times will be about 8 percent faster.
*  was installed with a loop-back hood.
*
NEXT  CCE           *NOT FIRST ENTRY
      JSB .SEND     *ENTER LAB6 MICROCODE
      DEF EXIT      *RETURN
      OCT 30        *PIC SELECT CODE
      OCT 47        *R31 DATA
      OCT 144       *R30 DATA
      OTB 2B,C      *SET UP GLOBAL
      OTA 31B       *DATA TO R31
      OTA 30B       *DATA TO R30
      STC 30B,C     *START TRANSFER
*
* The following instructions are used by the trap cell code.
*
      LIB 2B        *SAVE GLOBAL CAUSE PRIVILIGED
      OTB 2B,C      *SET GLOBAL FOR PIC
      CLC 30B,C     *DISABLE PIC
      OTB 2B,C      *RESTORE GLOBAL
*
* Unlock the partition and go home
*
EXIT  JSB EXEC      *UNLOCK PARTITION
      DEF *+3
      DEF D22
      DEF UNLCK
*
      JSB EXEC
      DEF *+2
      DEF SIX
LOCK  DEC 1
UNLCK DEC 0
D22   DEC 22
SIX   DEC 6
OPCOD OCT 105503
      END JOKE6
```

## EXAMPLE 3: PRIVILEGED DRIVER, MICROPROGRAM "SEND"

```
MPARA,L;
*
UG_SEND:
*
* DESCRIPTION:
*
*   The instruction "SEND" is a microcoded driver used to output a
*   16-bit data word to register 30 and a 16-bit control word to
*   register 31 of the parallel interface card (12006A).  The card
*   is programmed to interrupt upon completion of the data transfer.
*   A microcoded completion routine "TRAP" is used to complete the
*   data transfer. The routine "TRAP" is entered directly from the
*   trap cell upon interrupt.
*
*
* CALLING SEQ:    CLE             Indicates first entry
*                 CCE             All subseqent entries
*
*                 JSB .SEND        User opcode
*                 DEF RTN         Return Address
*                 OCT SCODE       Select code of PIC card
*                 OCT CNTL        Data for R31 of PIC
*                 OCT DATA        Data for R30 of PIC
*                 OTB 2B,C        Set up Global Register
*                 OTA 31B         Output to R31 of PIC
*                 OTA 30B         Output to R30 of PIC
*                 STC 30B,C       Enable transfer and interrupt
* (used by trap)  LIB 2B          Save Global Register
*       .         OTB 2B,C        Output to Global Register
*       .         CLC 30B,C       Disable PIC card
*       .         OTB 2B,C        Restore Global
*
*******************************************************************
*                                                                 *
*              Start of Main Program                              *
*                                                                 *
*******************************************************************
*
* Environment:  The instruction "SEND" is entered directly from the
*               user program.  Therefore, memory protect is enabled
*               and the mapping system is set up for the currnet
*               executing program (ie. the program that executed
*               the instruction "SEND"). The global register is
*               available and does not have to be saved.
*
* Register usuage:
*
*          s0  -- Contains the return address of the next
*                 user instruction
*          s5  -- Scratch register
*          s6  -- Scratch register
*          s7  -- Scratch register
*         usr  -- bits 0-5  PIC card's select code
*                 bit 15  1=card busy  0=card available
*     prin(6) -- Contains the map set number associated with the
*                 PIC card's select code.  This map set (ie. port
*                 map) will contain a copy of the current user map.
*     prin(7) -- Contains the 15 bit logical memory address that
*                 points to the LIB 2B instruction in the user map.
*                 This instruction will be used by the "TRAP"
*                 microcoded driver.
*
*  Note:
*
* Usr, prin(6), and prin(7) are used to pass information to the
* trap cell microcode completion routine.
*
```

## EXAMPLE 3: PRIVILEGED DRIVER, MICROPROGRAM "SEND" (Continued)

```
*
*
SEND:    $origin 0x3090$
*
         rdp, ip;                      *read return address
         s0 := t;                      *save it
*
*  All I/O instructions must be executed with memory protect
*  disabled in order to function correctly. Otherwise, the iorq
*  (input/output request) signal will not be recognized by this
*  microcoded driver.
*
         call MP_DIS;                  *disable memory protect
         rdp, ip;                      *read select code
*
*  Go set up the port map associated with the select code for the
*  PIC card.  The port map information will be used by the trap
*  cell microcoded driver that is entered after the I/O card
*  interrupts. The select code is returned in the user reserved
*  register "USR".
*
         call MAP_SAV;                 *go set up port map
*
*  Pull in the parameters passed by the calling program.
*
         b := usr,                     *b=select code
            rdp, ip;                   *read R31 data
         a := t,                       *a=R31 data
            rdp, ip;                   *get R30 data
         s5 := t,                      *temp save of R30 data
            stf;                       *indicate no iohs
*
*  Broadcast the I/O instruction "OTB 2B,C" over the backplane for
*  the I/O cards to recognize.  This I/O instruction loads the
*  global registers (located on every I/O card) with the select
*  code in the B register.  Since the PIC's select code is in the
*  B register, we have "enabled" the PIC card to execute all further
*  I/O instructions.  Only the I/O card whose select code matches
*  the current contents of the global register will execute the I/O
*  instruction.
*
         call IO_CMD;                  *broadcast OTB 2B,C
         call OUT_GLB;                 *execute OTB 2B,C
*
         ip, clf;                      *do IOHS from now on
*
*  Output the 16 bit control word to Register 31 of the PIC card
*
         call IO_CMD;                  *do OTA 31B
*
*  Output the 16 bit data word to Register 30 or the PIC card
*
         a := s5,                      *a=R30 data
            ip;                        *
         call IO_CMD;                  *do OTA 30B
         ip;                           *
*
*  Send the Device Command signal to the device and clear the
*  flag so that the device flag will generate an interrupt.
*
         call IO_CMD;                  *do STC 30B,C
*
* Indicate that the I/O operation is in progress. The "USR" register
* register is used to communicate information to the trap cell
* microcode.
```

EXAMPLE 3: PRIVILEGED DRIVER, MICROPROGRAM "SEND" (Continued)

```
*
         usr := 0x8000 ior usr;        *set busy bit
         ip;                           *P=pointer to LIA 2B
         n := 0x0007;                  *set index for PRIN
         prin := p;                    *save for trap cell code
         call MP_ENB;                  *turn MP back on
         p := s0;                      *restore  user pc
         fchp, rtn;                    *return to user
********************************************************************
*                                                                  *
*            Subroutine MP_DIS                                     *
*                                                                  *
********************************************************************
*
*  Disables the memory protect logic by setting bit bit 12 in the
*  CPU's interrupt status register (ist). The subroutine also
*  clears any outstanding memory protect violations by writing a
*  value of 2 into the lower 4 bits of the ist register.
*
MP_DIS:  s6 := 0xfff0 and ist;         *set MP bit in ist
         ist := s6 ior 0x1000;         *
         s6 := 0xfff0 and ist;         *clear any generated
         ist := 0x0002 ior s6;         *MP violations
         rtn;                          *
*
********************************************************************
*                                                                  *
*            Subroutine MP_ENB                                     *
*                                                                  *
********************************************************************
*
*    Enables the memory protect logic by clearing bit 12
*    in the CPU's interrupt status register (ist).
*
*
MP_ENB:  s6 := 0xfff0 and ist;         *clear MP bit in ist
         ist := s6 and not 0x1000;     *
         rtn;                          *
*
*
********************************************************************
*                                                                  *
*                 Subroutine OUT_GLB                               *
*                                                                  *
********************************************************************
*
*   Executes a 'OTB 2B,C' I/O instruction to load the global
*   registers on every I/O card.  The clear flag option is  used
*   to enable the global register.  Prior to calling this routine,
*   the I/O instruction must be broadcast over the I/O backplane
*   in order to work correctly.  A command word of 12 decimal is
*   output over the backplane to the I/O cards.  Then, the select
*   code is output.  Each of the I/O cards will save the value of
*   the global register internally. The global register was
*   enabled by the clear flag option in the I/O opcode and its state
*   is recorded in a privileged CPU status register (prin(e)).
*
OUT_GLB: s7 := 12;                     *command word for IOP
CK_IORQ: if not iorq goto CK_IORQ;     *wait for iorq
         nop := s7, wrio;              *send command to IOP
         nop := b, wrio;               *send select code
         n := 0x000e;                  *update CPU status for CLF
         prin := prin ior 1;           *bit 0=1 for GR enabled
         rtn;
```

## EXAMPLE 3: PRIVILEGED DRIVER, MICROPROGRAM "SEND" (Continued)

```
*
*
*******************************************************************
*                                                                 *
*                 Subroutine MAP_SAV                              *
*                                                                 *
*******************************************************************
*
*
*   Copies the current user map into the port map reserved for the
*   select code of the PIC card.  This is usually done by the
*   operating system prior to entry into a driver.  Since our
*   microcoded driver is bypassing the operating system, the driver
*   has to perform the set up. The port map is used by the trap cell
*   microcode in order to access the user area.  This driver was
*   written such that any user program could be executing when the
*   interrupt occurs. That is why the current user map must be saved.
*
*   On entry:  e = 0   indicates first entry into the routine
*              e = 1   indicates set up has already been done
*
*              A read of the PIC card's select code has already
*              been started by the calling routine.
*
*   On exit:   The PIC card's select code is loaded into the
*              user reserved register "USR", and the port
*              map is set to the current user map.
*
*              The user reserved privileged register (prin(6))
*              is loaded with the port map number. This value
*              is used by the trap cell microcode to enable the
*              port map when accessing the user space.
*
MAP_SAV: if e then rtn,                 *1st entry?
            usr := t;                   *save select code!
         n := 0x0006;                   *index to priv register
         prin := usr - 0x0008;          *save map set no. for PIC
*
*  In an A600/A700/RTE-A.1 computer environment there are 32 sets
*  of 32 maps used to access greater than 32K of memory.  The
*  map set allocation is as follows:
*  Program allocation:
*
*   map set #0  for the operating system
*   map set #3  for the current user map
*
*  I/O allocation:
*
*   map set #8  for select code 20B
*   map set #9  for select code 21B
*
*              .
*              .
*
*   map set #31 for select code 47B
*
*   A map address register is used to access the individual
*   map registers (0-1023).  Therefore, the map address register
*   (mpar) contains 0-31 to access the operating system's map and
*   992-1023 for select code's 47B port map.
*
*   The lower 5 bits of the "memr" register contains the active
*   map set number (0-31) that is currently being used for memory
*   accesses (ie. the execute map).  Since we are executing in
*   the user space, this number better be 3!
*
```

EXAMPLE 3: PRIVILEGED DRIVER, MICROPROGRAM "SEND" (Continued)

```
*
         s6 := memr and 0x001f;       *calc source mpar
         s6 := rl4(s6);               *mpar=3*32
         s6 := rl1(s6);               *
         s7 := rl4(prin);             *calc dest mpar
         s7 := rl1(s7);               *mpar=map set*32
         ct := 31;                    *loop count
*
* srin(0) = map address register (mpar)
*
         n := zero;                   *index for mpar
*
*   transfer 32 user map registers to port map for the PIC
*
TR_MAP:  srin := s6;                  *source mpar
         s6 := s6 + one;              *
         s5 := map;                   *save map value
         srin := s7;                  *destination mpar
         s7 := s7 + one;              *
         map := s5,                   *load map
            if not ctz goto TR_MAP;   *done ?
         rtn;                         *
*****************************************************************
*                                                               *
*                Subroutine IO_CMD                              *
*                                                               *
*****************************************************************
*
*  This routine is used to broadcast an I/O instruction  over the
*  backplane.  The I/O cards monitor the instructions on the
*  backplane and execute the I/O instructions when appropriate.
*
* Entry: The program counter points to the I/O instruction in
*        memory that is to be broadcast.  In this example, the I/O
*        instructions are passed by the calling program as
*        parameters in the user space.  Make sure that the "memr"
*        register is set to the correct map when accessing memory!
*
*        The general purpose CPU flag is set if no I/O handshake
*        is required by the I/O instruction.  The flag is clear if
*        the I/O instruction requires a handshake from the microcode
*        in order to execute the I/O instruction. All I/O
*        instructions with their lower 6 bits >17B require an I/O
*        handshake (iohs)!
*
*      Note:  Refer to the base set routines BCST and IOHS for
*             a more general purpose routine.
*
*
IO_CMD:  bfb, bbus/p;             *broadcast the I/O instruction
         ct := t;                 *I/O opcode to ct and freeze!
         if f then rtn;           *flag set for no handshake
*
*  CAUTION:  We must wait a certain time period after the I/O
*            instruction has been broadcast over the backplane
*            before checking for the input/output request (iorq)
*            signal generated by the I/O cards.  The current base
*            set waits two cycles after the freeze (ct := t)
*            instruction before checking for the iorq signal.
*
         nop;                         *timing
         if not iorq then rtn;        *does IOP need our help?
```

EXAMPLE 3: PRIVILEGED DRIVER, MICROPROGRAM "SEND" (Continued)

```
*
*  The CPU microcode must act as a slave processor to the I/O
*  processor chip (IOP) if the I/O card needs CPU resources in order
*  to execute the I/O instruction (ie. access to the A register,
*  etc.).  The IOP asserts the iorq to signal that it needs the
*  slave processor's help.  The CPU microcode must read a control
*  word from the IOP that tells the CPU what to do.  The control
*  word is defined as follows:
*
*                        Data bus bit
*                      ----------------
*                        8  7  6  5  4
*
*   Nop                  X  0  0  0  0
*   Load program cnter   X  0  0  0  1
*   Load A register      X  0  0  1  0
*   Load B register      X  0  0  1  1
*   Clear O register     X  0  1  0  0
*   Set O register       X  0  1  0  1
*   Merge into A/B       X  0  1  1  0
*   Inc program cnter    X  0  1  1  1
*   Undefined            X  1  0  0  0
*   Enable boot memory   X  1  0  0  1
*   Read A register      X  1  0  1  0
*   Read B register      X  1  0  1  1
*   Clear E register     X  1  1  0  0
*   Set E register       X  1  1  0  1
*   Read P register      X  1  1  1  0
*   Read and inc P       X  1  1  1  1
*
*   X=0 if last handshake
*   X=1 if more handshakes are required
*       (ie. continue looping)
*
*
IOHS_DO: rdio,                          *read IOP cntl word (CW)
            s7 := ct;                   *save instruction opcode
         nop := 0x0100 and t;           *check continue bit
         if yz then goto IOHS_QT,       *last one?
            ct := t;                    *put CW into counter
         call IOHS_TB, ct74,            *index into table via CW
            ct := t;                    *load opcode into counter
*
IORQ_WT: if iorq goto IOHS_DO;          *wait for handshake
         goto IORQ_WT;                  *try again
*
IOHS_QT: goto IOHS_TB, ct74,            *last time
            ct := s7;                   *load opcode into counter
*
*  The command table is indexed by the IOP control word bits 4-7.
*  The table must begin on an even 16 word boundry.  Refer to the
*  base set's iohs_tbl.
*
```

## EXAMPLE 3: PRIVILEGED DRIVER, MICROPROGRAM "SEND" (Continued)

```
*
IOHS_TB: $align 16$
{00}     rtn;                           *nop
{01}     rdio, goto IOHS_LP;            *load p from IOP
{02}     rdio, goto IOHS_LA;            *load a from IOP
{03}     rdio, goto IOHS_LB;            *load b from IOP
{04}     sto, rtn;                      *set o
{05}     clo, rtn;                      *clear o
{06}     rdio, goto IOHS_MI;            *merge into a/b
{07}     ip, rtn;                       *increment p
{10}     rtn;                           *write status to IOP
{11}     goto IOHS_BT;                  *enable BOOT memory
{12}     nop := a, wrio, rtn;           *write a to IOP
{13}     nop := b, wrio, rtn;           *write b to IOP
{14}     cle, rtn;                      *clear e
{15}     ste, rtn;                      *set e
{16}     nop := p, wrio, rtn;           *write p to IOP
{17}     nop := p, wrio, ip, rtn;       *write p to IOP, inc p
*
*
* Execution of individual commands
*
*
IOHS_LP: p := t, rtn;                   *load p from IOP
*
IOHS_LA: a := t, rtn;                   *load a from IOP
*
IOHS_LB: b := t, rtn;                   *load b from IOP
*
IOHS_MI: s6 := cab;                     *load current a/b
         cab := s6 ior t, rtn;          *merge in IOP value
*
IOHS_BT: rtn;                           *sorry charlie
*
```

## EXAMPLE 3: PRIVILEGED DRIVER, MICROPROGRAM "TRAP"

```
MPARA,L;
*
UG_TRAP: $origin 0x3100$
*
* DESCRIPTION:
*
*  This microcode is entered directly from the trap cell upon an
*  interrupt from the parallel interface card. The main program
*  must write the instruction opcode into the trap cell associated
*  with the PIC card's select code. The "TRAP" microcode is
*  essentially the privileged portion of the microcoded driver.
*  The TRAP microcode must save the current machine state, perform
*  the necessary I/O to the PIC card, and restore the machine state
*  upon exit.
*
*  The initiation of the I/O on the PIC card was started by the
*  "SEND" microcode.
```

## EXAMPLE 3: PRIVILEGED DRIVER, MICROPROGRAM "TRAP" (Continued)

```
*
* CALLING SEQ: JSB .SEND
*              DEF RTN
*              OCT SCODE       Select code of PIC card
*              OCT CNTL        Data for R31 of PIC
*              OCT DATA        Data for R30 of PIC
*              OTB 2B,C        Output to Global Register
*              OTA 31B         Output to R31 of PIC
*              OTA 30B         Output to R30 of PIC
*              STC 30B,C       Enable transfer and interrupt
* prin(7) ---> LIB 2B          Save Global Register
*              OTB 2B,C        Output to Global Register
*              CLC 30B,C       Disable PIC card
*              OTB 2B,C        Restore Global Register
*
*  Subroutines located in the "SEND" microcode routine
*  that are used by "TRAP".
*
EXTRNL:  $define adrl/IO_CMD 0x30C7$   *Execute I/O command
         $define adrl/OUT_GLB 0x30AF$  *Output to global register
         $define adrl/MP_ENB  0x30AC$  *Enable memory protect
*
*
*Environment: The "TRAP" microcode is entered directly from the trap
*             cell upon interrupt.  The CPU microcode has already
*             disabled memory protect and saved the previous
*             mapping information in a CPU privileged register.
*             The CPU microcode has also modified the "memr"
*             register by setting the active map set to 0 (ie. the
*             operating system's map).  Therefore, all memory
*             references will refer to the lower 32k of memory
*             unless memr is modified.
*
*Registers:
*          usr   --  Bits 0-5 contains select code of the PIC
*                    Bit 15  busy=1  available=0
*                    Register must be set by the "SEND"
*                    microcode.
*        prin(6) --  Contains the map set number associated
*                    with the PIC card's select code.
*                    Register must be set by the "SEND"
*                    microcode.
*        prin(7) --  Contains the 15 bit logical address of the
*                    'LIB 2B' I/O instruction in the user
*                    map of the program that called the
*                    "SEND" microcode.
*            s0  --  A register save value
*            s1  --  B register save value
*            s2  --  Program counter save value
*            s3  --  Global register save value
*            s5  --  Scratch reg
*            s6  --  Scratch reg
*            s7  --  Scratch reg
*
*
*  Save the state of the machine
*
*
DVR_CON: s0 := a;                      *save a
         s1 := b;                      *save b
         s2 := p;                      *save p
         nop := usr;                   *check busy bit
         if not y15 then goto EXIT;    *not busy, ignore
*
```

## EXAMPLE 3: PRIVILEGED DRIVER, MICROPROGRAM "TRAP" (Continued)

```
*
* In order to gain access to the user area where the I/O
* instructions are located,  the memr register needs to be modified
* so that all memory references will be done in the previous user
* map that called the "SEND" microcode. The map set number was
* saved in prin(6) by the "SEND" microcode.
*
          n := 0x0006;                   *memr save index
          memr := prin,                  *memr=prin(6)
             in;                         *pc save index
*
* The program cntr is reset to point to the LIB instruction in the
* user map (this was also saved by the "SEND" ).
*
          p := prin,                     *pc=prin(7)
*
* Save the current state of the global register
*
             clf;                        *set for iohs
          call IO_CMD;                   *do LIB 2B
          s3 := b,                       *save global
             stf;                        *set for no iohs
          b := usr and 0x003f;           *b=PIC select code
*
* If the global register is still set to the PIC, then just disable
* the device and exit.
*
* Otherwise, reset the global register to the PIC's select code via
* the OTB 2B,C command
*
          nop := b xor s3,               *cmpr global to scode
             ip;                         *point to OTB 2B,C
          if yz then goto DIS_DEV;       *equal?
*
* Reset Global
*
          call IO_CMD;                   *broadcast w/o iohs
          call OUT_GLB;                  *do OTB 2B,C
*
* Disable the PIC card
*
DIS_DEV:  clf, ip;                       *set for iohs
          call IO_CMD;                   *do CLC 30B,C
*
* Restore Machine State
*
          nop := s3 xor b;               *global reset ?
          if yz goto RST_MAP;            *if not skip it
RST_GLB:  b := s3,                       *reset global
             stf,                        *no iohs
             ip;                         *point to OTB 2B,C
          call IO_CMD;                   *broadcast w/o iohs
          call OUT_GLB;                  *do OTB 2B,C
EXIT:
*
* The interrupted mapping information was saved in prin(d) by the
* base set microcode upon interrupt.  The base set maintains three
* separate maps; an execute map, data1 map, and data2 map. The
* execute map contains the map set number for the executing user
* program (which is 3) or the map set number for the O/S which is 0.
*
* Since this driver is privileged, the interrupted execution map
* could be either the user or the system (ie. we can interrupt
* either a user program or the O/S). The data1 and data2 maps
* could contain any one of the map set numbers (0-31).  Refer to
* the DMS instructions for details.
```

EXAMPLE 3: PRIVILEGED DRIVER, MICROPROGRAM "TRAP" (Continued)

```
*
* Upon interrupt, the base set saves the current values of the
* three maps and the state of memory protect (enb/dis) into a
* privileged register (prin(d)) called IMAP (interrupted maps).
* The format of this register is shown below:
*
*
*      15   14       10  9       5  4       0
*     ---------------------------------------
*       MP   data2       data1      execute
*     ---------------------------------------
*                   prin(d)
*
*     MP    =  state of memory protect at time of interrupt
*              MP=1 for enabled   MP=0 for disabled
*     data1 =  value of data1 map at time of interrupt
*     data2 =  value of date2 map at time of interrupt
*   execute =  value of execute map at time of interrupt
*
*
*   The base set maintains the CURRENT value of the execute,
*   data1, and data2 maps in two registers (prin(f) and memr).
*   The format is as follows:
*
*
*           13   12 - 8         5    4  -  0
*     ---------------------------------------
*           a/b  data2         a/b   data1
*     ---------------------------------------
*                   prin(f)
*
*      a/b = 1   a/b addressability disabled
*      a/b = 0   a/b addressability enabled
*      data1     map set number for data1 map
*      data2     map set number for data2 map
*
*   memr(0-4) = Current execute map set number.
*
*
RST_MAP: n := 0x000d;                    *index to IMAP
         s5 := prin;                     *save IMAP
         n := ones;                      *reset data2/data1
         s6 := rr1(s5);                  *
         s7 := rr1(s6);                  *
         s7 := 0x1f00 and s7;            *data2 in upper byte
         s6 := rl4(s6);                  *
         s6 := swzu(s6);                 *
         s6 := s6 and 0x001f;            *data1 in low byte
         prin := s6 ior 0x2020;          *no a/b addressability
         prin := prin ior s7;            *set unpacked data2/data1
         memr := s5 and 0x001f;          *reset exec map (s5/bbus)
         if not b15 goto DONE;           *enable MP if needed
         call MP_ENB;                    *
DONE:    p := s2;                        *restore pc
         b := s1;                        *restore b
         a := s0;                        *restore a
         fchp, rtn;                      *return to user/system
```

# APPENDIX A

## SUMMARY OF WORD TYPES

# SUMMARY OF WORD TYPES

APPENDIX A

Summary of Word Types vs. Field Contents

| WORD TYPE | FIELD1 | FIELD2 | FIELD3 | FIELD4 | FIELD5 | FIELD6 | FIELD7 |
|---|---|---|---|---|---|---|---|
| 1 | OP1 | ABUS | SP0 | SP2 | ALU | BBUS | STOR |
| 2 | OP2 | ABUS | SP0 | CNDX | ALU | BBUS | STOR |
| 3 | OP3 | ADRS | SP1 | CNDX | ALU | BBUS | STOR |
| 4 | OP4 | ADRS | SP1 | SP2 | ALU | BBUS | STOR |
| 5 | OP5 | ADRL | | | ALU | BBUS | STOR |
| 6 | OP6 | DAT | | | ALU | BBUS | STOR |
| 1S | OP1 | ABUS | ALUS | SP2 | SPEC | BBUS | STOR |
| 2S | OP2 | ABUS | ALUS | CNDX | SPEC | BBUS | STOR |
| 3S | OP3 | ADRS | ALUS | CNDX | SPEC | BBUS | STOR |
| 4S | OP4 | ADRS | ALUS | SP2 | SPEC | BBUS | STOR |
| 5S | OP5 | ADRL* | | | SPEC* | BBUS | STOR |

*Go to microinstruction table for microorder long branch jump (lower four bits of destination address replaced by bits 3-0 of CT).

# APPENDIX B
## SUMMARY OF MICROORDERS

# SUMMARY OF MICROORDERS

APPENDIX
B

Summary of Microorders by Field

| FIELD | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **CODE** | **OP** | **CNDX** | **SP0** | **SP1** | **SP2** | **ALUS** | **ALU** | **ABUS** | **BBUS** | **STOR** |
| 00000 | IMM | SF | NOP* | NOP* | NOP* | UMPY | SPEC | A | A | A |
| 00001 | IMM | F | LDQ | LDQ | CMDW | TMPY | SBAC | B | B | B |
| 00010 | IMM | ALOV | RR1 | RR1 | DCT | SM2C | SBBC | X | X | X |
| 00011 | IMM | CF | RL1 | RL1 | CLF | RMLC | ADDC | Y | Y | Y |
| 00100 | IMM | YZ | LR1 | LR1 | STF | DNRM | ADBC | ACC* | ACC* | ACC |
| 00101 | IMM | Y15 | LL1 | LL1 | IP | SNRM | CMBC | HP1** | HP1** | HP1** |
| 00110 | IMM | B15 | AR1 | AR1 | LWF | DIV | ADAC | HP2** | HP2** | HP2** |
| 00111 | IMM | INTF | AL1 | AL1 | LWE | DIV1 | CMAC | USR** | USR** | USR** |
| 01000 | JMPL | IORQ | RDP | RDP | CMID | SWAP | ZERO* | S0 | S0 | S0 |
| 01001 | JMPL | PON | IN | IN | RDP | SWZU | CAND | S1 | S1 | S1 |
| 01010 | JSBL | MPEN | RDB | RDB | WRIO | SWZY | XNOR | S2 | S2 | S2 |
| 01011 | JSBL | O | STE | STE | DN | ZUY | XOR | S3 | S3 | S3 |
| 01100 | JMP | E | CLE | CLE | FCHP | ZLY | AND | S4 | S4 | S4 |
| 01101 | JMP | INTP | FCIN | FCIN | RDIO | SRG | INOR | S5 | S5 | S5 |
| 01110 | JSB | CTZ4 | ACF | ACF | CT30 | RL4 | NAND | S6 | S6 | S6 |
| 01111 | JSB | CTZ | IP | IP | CT74 | ASG | IOR | S7 | S7 | S7 |
| 10000 | JMPF | | STOR | | | | | | GRIN | GRIN |
| 10001 | JMPF | | — | | | | | | FA | WRP |
| 10010 | JMPT | | — | | | | | | SRIN | SRIN |
| 10011 | JMPT | | — | | | | | | P | P |
| 10100 | JSBF | | — | | | | | | Q | NOP* |
| 10101 | JSBF | | — | | | | | | T | WRB |
| 10110 | JSBT | | — | | | | | | IST | IST |
| 10111 | JSBT | | — | | | | | | N | N |
| 11000 | JTAB | | IFCH | | | | | | PRIN | PRIN |
| 11001 | — | | BFB | | | | | | MA | CWRB |
| 11010 | RTN | | CK2 | | | | | | MEMR | MEMR |
| 11011 | NOP* | | ENOE | | | | | | CT | CT |
| 11100 | RTNF | | STO | | | | | | SR | LR |
| 11101 | SP0F | | CLO | | | | | | MAP | MAP |
| 11110 | RTNT | | FCHB | | | | | | CAB | CAB |
| 11111 | SP0T | | LDBR | | | | | | CXY | CXY |

OP Field Divisions:

| OP1=JTAB | OP2=SP0T | OP3=JMPF | OP4=JMP | OP5=JMPL | OP6=IMM |
|---|---|---|---|---|---|
| NOP | SP0F | JMPT | JSB | JSBL | |
| RTN | RTNT | JSBF | | | |
| | RTNF | JSBT | | | |

* Default Microorder.

**Reserved register for Hewlett-Packard (HP1 and HP2) and user (USR).

# APPENDIX C

## SUMMARY OF MICROORDER PHRASES

# SUMMARY OF MICROORDER PHRASES

APPENDIX C

Summary of Microorder Phrases

| BRANCHING PHRASES | | | |
|---|---|---|---|
| **PHRASE** | **RESULTING FIELDS** | | |
| GOTO adr | OP4/JMPL | ADRL/adr | |
| GOTO adr | OP5/JMP | ADRS/adr | |
| LGOTO adr | OP5/JMPL | ADRL/adr | |
| SGOTO adr | OP4/JMP | ADRS/adr | |
| CALL adr | OP5/JSBL | ADRL/adr | |
| CALL adr | OP4/JSB | ADRS/adr | |
| LCALL adr | OP5/JSBL | ADRL/adr | |
| SCALL adr | OP4/JSB | ADRS/adr | |
| GOTOTBL adr | OP5/JMPL | ADRL/adr | ALU/SPEC |
| CALLTBL adr | OP5/JSBL | ADRL/adr | ALU/SPEC |
| NOP | OP1/NOP | | |
| JTAB | OP1/JTAB | | |
| RTN | OP1/RTN | | |

| BASIC ARITHMETIC PHRASES | | | | |
|---|---|---|---|---|
| **PHRASE** | **RESULTING FIELDS** | | | |
| stor := bbus | ALU/adbc | STOR/stor | BBUS/bbus | |
| stor := abus | ALU/adac | STOR/stor | | ABUS/abus |
| stor := NOT bbus | ALU/cmbc | STOR/stor | BBUS/bbus | |
| stor := NOT abus | ALU/cmac | STOR/stor | | ABUS/abus |
| stor := ONES | ALU/xnor | STOR/stor | BBUS/ACC | ABUS/ACC |
| stor := ZEROS | ALU/xor | STOR/stor | BBUS/ACC | ABUS/ACC |
| stor := ZERO | ALU/zero | STOR/stor | | |
| stor := NOT abus AND bbus | ALU/cand | STOR/stor | BBUS/bbus | ABUS/abus |
| stor := bbus AND NOT abus | ALU/cand | STOR/stor | BBUS/bbus | ABUS/abus |
| stor := abus - bbus | ALU/sbbc | STOR/stor | BBUS/bbus | ABUS/abus |
| stor := bbus - abus | ALU/sbac | STOR/stor | BBUS/bbus | ABUS/abus |
| stor := abus + bbus | ALU/addc | STOR/stor | BBUS/bbus | ABUS/abus |
| stor := bbus + abus | ALU/addc | STOR/stor | BBUS/bbus | ABUS/abus |
| stor := abus XNOR bbus | ALU/xnor | STOR/stor | BBUS/bbus | ABUS/abus |
| stor := bbus XNOR abus | ALU/xnor | STOR/stor | BBUS/bbus | ABUS/abus |
| stor := abus XOR bbus | ALU/xor | STOR/stor | BBUS/bbus | ABUS/abus |
| stor := bbus XOR abus | ALU/xor | STOR/stor | BBUS/bbus | ABUS/abus |
| stor := abus AND bbus | ALU/and | STOR/stor | BBUS/bbus | ABUS/abus |
| stor := bbus AND abus | ALU/and | STOR/stor | BBUS/bbus | ABUS/abus |
| stor := abus NAND bbus | ALU/nand | STOR/stor | BBUS/bbus | ABUS/abus |
| stor := bbus NAND abus | ALU/nand | STOR/stor | BBUS/bbus | ABUS/abus |
| stor := abus IOR bbus | ALU/ior | STOR/stor | BBUS/bbus | ABUS/abus |
| stor := bbus IOR abus | ALU/ior | STOR/stor | BBUS/bbus | ABUS/abus |
| stor := abus INOR bbus | ALU/inor | STOR/stor | BBUS/bbus | ABUS/abus |
| stor := bbus INOR abus | ALU/inor | STOR/stor | BBUS/bbus | ABUS/abus |

Summary of Microorder Phrases (Continued)

## ARITHMETIC PHRASES WITH SHIFT OR ROTATE MICROORDERS

These phrases specify the shift or rotate microorders to shift or rotate the output of the ALU. The shift or rotate microorders can be placed in either the SP0 or SP1 fields which is indicated here by SP*.

The resulting fields are the shift or rotate microorders shown and the resulting fields from the basic arithmetic phrase.

| PHRASE | RESULTING FIELD |
|---|---|
| stor := LL1 (right side of basic arithmetic phrase) | SP*/LL1 |
| stor := LR1 (right side of basic arithmetic phrase) | SP*/LR1 |
| stor := RL1 (right side of basic arithmetic phrase) | SP*/RL1 |
| stor := RR1 (right side of basic arithmetic phrase) | SP*/RR1 |
| stor := AL1 (right side of basic arithmetic phrase) | SP*/AL1 |
| stor := AR1 (right side of basic arithmetic phrase) | SP*/AR1 |

## ARITHMETIC PHRASES WITH CARRY-IN MODIFYING MICROORDERS

These phrases use the FCIN or ACF microorders to modify the carry-in of the ALU. In the following phrases, FCIN or ACF will be placed in either the SP0 or SP1 fields which is indicated here as SP*.

| PHRASE | RESULTING FIELDS | | | | |
|---|---|---|---|---|---|
| stor := bbus + ONE | ALU/adbc | STOR/stor | BBUS/bbus | | SP*/FCIN |
| stor := abus + ONE | ALU/adac | STOR/stor | | ABUS/abus | SP*/FCIN |
| stor := - BBUS | ALU/cmbc | STOR/stor | BBUS/bbus | | SP*/FCIN |
| stor := NOT bbus + ONE | ALU/cmbc | STOR/stor | BBUS/bbus | | SP*/FCIN |
| stor := - ABUS | ALU/cmac | STOR/stor | | ABUS/abus | SP*/FCIN |
| stor := NOT abus + ONE | ALU/cmac | STOR/stor | | ABUS/abus | SP*/FCIN |
| stor := abus - bbus - ONE | ALU/sbbc | STOR/stor | BBUS/bbus | ABUS/abus | SP*/FCIN |
| stor := bbus - abus - ONE | ALU/sbac | STOR/stor | BBUS/bbus | ABUS/abus | SP*/FCIN |
| stor := abus + bbus + ONE | ALU/addc | STOR/stor | BBUS/bbus | ABUS/abus | SP*/FCIN |
| stor := bbus + abus + ONE | ALU/addc | STOR/stor | BBUS/bbus | ABUS/abus | SP*/FCIN |
| stor := bbus + CF | ALU/adbc | STOR/stor | BBUS/bbus | | SP*/ACF |
| stor := abus + CF | ALU/adac | STOR/stor | | ABUS/abus | SP*/ACF |
| stor := NOT bbus + CF | ALU/cmbc | STOR/stor | BBUS/bbus | | SP*/ACF |
| stor := NOT abus + CF | ALU/cmac | STOR/stor | | ABUS/abus | SP*/ACF |
| stor := abus - bbus - BR | ALU/sbbc | STOR/stor | BBUS/bbus | ABUS/abus | SP*/ACF |
| stor := bbus - abus - BR | ALU/sbac | STOR/stor | BBUS/bbus | ABUS/abus | SP*/ACF |
| stor := abus + bbus + CF | ALU/addc | STOR/stor | BBUS/bbus | ABUS/abus | SP*/ACF |
| stor := bbus + abus + CF | ALU/addc | STOR/stor | BBUS/bbus | ABUS/abus | SP*/ACF |

## SPECIAL ARITHMETIC PHRASES

| PHRASE | RESULTING FIELDS | | | | |
|---|---|---|---|---|---|
| stor := alus (bbus) | ALU/SPEC | STOR/stor | BBUS/bbus | | ALUS/alus |
| stor := alus (bbus, abus) | ALU/SPEC | STOR/stor | BBUS/bbus | ABUS/abus | ALUS/alus |

Summary of Microorder Phrases (Continued)

**ARITHMETIC PHRASES WITH IMMEDIATE DATA***

| PHRASE | RESULTING FIELDS | | | |
|---|---|---|---|---|
| stor := data | ALU/adac | STOR/stor | | DAT/data | OP6/IMM |
| stor := NOT data | ALU/cmac | STOR/stor | | DAT/data | OP6/IMM |
| stor := NOT data AND bbus | ALU/cand | STOR/stor | BBUS/bbus | DAT/data | OP6/IMM |
| stor := bbus AND NOT data | ALU/cand | STOR/stor | BBUS/bbus | DAT/data | OP6/IMM |
| stor := data - bbus | ALU/sbbc | STOR/stor | BBUS/bbus | DAT/data | OP6/IMM |
| stor := bbus - data | ALU/sbac | STOR/stor | BBUS/bbus | DAT/data | OP6/IMM |
| stor := data + bbus | ALU/addc | STOR/stor | BBUS/bbus | DAT/data | OP6/IMM |
| stor := bbus + data | ALU/addc | STOR/stor | BBUS/bbus | DAT/data | OP6/IMM |
| stor := data XNOR bbus | ALU/xnor | STOR/stor | BBUS/bbus | DAT/data | OP6/IMM |
| stor := bbus XNOR data | ALU/xnor | STOR/stor | BBUS/bbus | DAT/data | OP6/IMM |
| stor := data XOR bbus | ALU/xor | STOR/stor | BBUS/bbus | DAT/data | OP6/IMM |
| stor := bbus XOR data | ALU/xor | STOR/stor | BBUS/bbus | DAT/data | OP6/IMM |
| stor := data AND bbus | ALU/and | STOR/stor | BBUS/bbus | DAT/data | OP6/IMM |
| stor := bbus AND data | ALU/and | STOR/stor | BBUS/bbus | DAT/data | OP6/IMM |
| stor := data NAND bbus | ALU/nand | STOR/stor | BBUS/bbus | DAT/data | OP6/IMM |
| stor := bbus NAND data | ALU/nand | STOR/stor | BBUS/bbus | DAT/data | OP6/IMM |
| stor := data IOR bbus | ALU/ior | STOR/stor | BBUS/bbus | DAT/data | OP6/IMM |
| stor := bbus IOR data | ALU/ior | STOR/stor | BBUS/bbus | DAT/data | OP6/IMM |
| stor := data INOR bbus | ALU/inor | STOR/stor | BBUS/bbus | DAT/data | OP6/IMM |
| stor := bbus INOR data | ALU/inor | STOR/stor | BBUS/bbus | DAT/data | OP6/IMM |

*Data can be immediate or a floating point control word.

**CONDITIONAL PHRASES**

| PHRASE | RESULTING FIELDS | | |
|---|---|---|---|
| IF cndx GOTO adr | OP3/JMPT | CNDX/cndx | ADRS/adr |
| IF cndx CALL adr | OP3/JSBT | CNDX/cndx | ADRS/adr |
| IF cndx RTN | OP2/RTNT | CNDX/cndx | |
| IF cndx THEN RTN | OP2/RTNT | CNDX/cndx | |
| IF cndx sp0 | OP2/SP0T | CNDX/cndx | SP0/sp0 |
| IF cndx THEN sp0 | OP2/SP0T | CNDX/cndx | SP0/sp0 |
| IF NOT cndx GOTO adr | OP3/JMPF | CNDX/cndx | ADRS/adr |
| IF NOT cndx CALL adr | OP3/JSBF | CNDX/cndx | ADRS/adr |
| IF NOT cndx RTN | OP2/RTNF | CNDX/cndx | |
| IF NOT cndx THEN RTN | OP2/RTNF | CNDX/cndx | |
| IF NOT cndx sp0 | OP2/SP0F | CNDX/cndx | SP0/sp0 |
| IF NOT cndx THEN sp0 | OP2/SP0F | CNDX/cndx | SP0/sp0 |

**SPECIAL PHRASES**

| PHRASE | RESULTING FIELD |
|---|---|
| sp0 | SP0/sp0 |
| sp1 | SP1/sp1 |
| sp2 | SP2/sp2 |

Summary of Microorders Phrases (Continued)

| FIELD FORCING PHRASES | |
|---|---|
| PHRASE | RESULTING FIELD |
| OP1/op1 | OP1/op1 |
| OP2/op2 | OP2/op2 |
| OP3/op3 | OP3/op3 |
| OP4/op4 | OP4/op4 |
| OP5/op5 | OP5/op5 |
| OP6/op6 | OP6/op6 |
| SP0/sp0 | SP0/sp0 |
| SP1/sp1 | SP1/sp1 |
| SP2/sp2 | SP2/sp2 |
| CNDX/cndx | CNDX/cndx |
| ABUS/abus | ABUS/abus |
| BBUS/bbus | BBUS/bbus |
| STOR/stor | STOR/stor |
| ALU/alu | ALU/alu |
| ALUS/alus | ALUS/alus |
| DAT/dat | DAT/dat |
| ADRL/adr | ADRL/adr |
| ADRS/adr | ADRS/adr |

# APPENDIX D
## FLOATING POINT MICROINSTRUCTIONS

# FLOATING-POINT MICROINSTRUCTIONS

APPENDIX D

Summary of Floating Point Control-Word Fields

| FUNCTION DESCRIPTION | CONTROL WORD FIELDS | | | | | |
|---|---|---|---|---|---|---|
| | FUNCTION | A-SOURCE | B-SOURCE | D-ADDR | B-ADDR | A-ADDR |
| SPF Addition | add.f2 | a_bus | b_bus | d0 | b0 | a0 |
| SPF Subtraction | sab.f2 | a_acc | b_acc | d1 | b1 | a1 |
| DPF Addition | add.f4 | a_rom | | d2 | b2 | a2 |
| DPF Subtraction | add.f4 | | | d3 | b3 | a3 |
| 0-63 right shift | shr.i4 | | | | | |
| 0-63 left shift | shl.i4 | | | | | |
| Float SI to DPF | ft.i1.f4 | | | | | |
| Float DI to SPF | ft.i2.f2 | | | | | |
| Float DI to DPF | ft.i2.f4 | | | | | |
| Convert DPF to SPF | cv.f4.f2 | | | | | |
| Fix DPF to SI | fx.f4.i1 | | | | | |
| Fix DPF to DI | fx.f4.i2 | | | | | |
| DI Multiply | mul.i2 | | | | | |
| DL Multiply | mul.l2 | | | | | |
| SPF Multiply | mul.f2 | | | | | |
| DPF Multiply | mul.f4 | | | | | |
| DI Divide | div.i2 | | | | | |
| DL Divide | div.l2 | | | | | |
| SPF Divide | div.f2 | | | | | |
| DPF Divide | div.f4 | | | | | |
| Clear Opcode (default) | clear | | | | | |

*Abbreviations used:
SI = Single Integer, DI = Double Integer,
SPF = Single Precision Floating Point,
DPF = Double Precision Floating Point,
DL = Double Logical

Summary of Floating-Point Microinstructions

| ALL OPERATIONS (GENERAL FORM) | | |
|---|---|---|
| stor:=fp( | function, | *what operation to perform |
| | a-operand source, | *source for the a-side operand |
| | b-operand source, | *source for the b-side operand |
| | a-operand address | *accumulator address for a-side operand |
| | b-operand address, | *accumulator address for b-side operand |
| | d-result address); | *accumulator address for result |

## Summary of Floating-Point Divide Microinstructions

| PHRASE | DIVIDE OPERATIONS |
|---|---|
| divsetup | prepare for division sequence |
| qbit3 | generate three bits of quotient |
| qbit2 | generate two bits of quotient |

## Summary of Divide Operations vs. Sequence of Control Words

| OPERATION | SEQUENCE OF DIVIDE CONTROL WORDS |
|---|---|
| div.i2 | 1 divsetup, 11 qbit3, 1 qbit2 |
| div.l2 | 1 divsetup, 11 qbit3, 1 qbit2 |
| div.f2 | 1 divsetup, 9 qbit3 |
| div.f4 | 1 divsetup, 19 qbit3 |

## Divide Microinstruction Phrases

| | |
|---|---|
| stor:=fp(divide operation); | *divide operation to perform |
| or | |
| stor:=fp(rom__addr/label); | *address of ACR pointer |

# APPENDIX E
## BASE SET LISTING

APPENDIX

E

# BASE SET LISTING

```
:**********************************************************************
:*                                                                    *
:*  NAME:  A700 PROCESSOR BASE SET                                    *
:*  PGMR:  SRK,TMH                                                    *
:*  DATE:  <820204.1442>                                              *
:*                                                                    *
:**********************************************************************
:* (C) Copyright Hewlett Packard Company 1982. All rights reserved.   *
:* No part of this program may be photocopied, reproduced or          *
:* translated to another program language without the prior written  *
:* consent of Hewlett Packard Company.                                *
:**********************************************************************
:LL,6
:SL,6,'LIST,WRSA
:LI,*LIST
:LI,&JFPLA
:ST,'CONTROL_AND_IOG        ,6
:ST,'MRG                    ,6
:ST,'ASG                    ,6
:ST,'SRG                    ,6
:ST,'EAG                    ,6
:ST,'EIG                    ,6
:ST,'FPSG                   ,6
:ST,'DMS                    ,6
:ST,'DIS                    ,6
:ST,'LIS                    ,6
:ST,'VMA                    ,6
:ST,'OSS                    ,6
:ST,'USER                   ,6
:ST,'SELFTEST               ,6
::
```

```
 FPLA: 82S100 SRK <820204.1442>
*        PRODUCT TERM     OUTPUT
                         HHHHHHHH
*      111111
*ROW   5432109876543210  76543210
*___   ________________  ________
  0    HH--------------  ..A..... *\
  1    H-H-------------  ..A..... * |
  2    H--H------------  ..A..... * |
  3    -H--L-----------  ...A.... * |
  4    --H-L-----------  ....A... * | memory reference group
  5    ---HL-----------  .....A.. * |
  6    -H--H-----------  ...A..A. * |
  7    --H-H-----------  ....A.A. * |
  8    ---HH-----------  .....AA. */
  9    HLLL-LHH--L-----  A.A...A. *\  1000 x011 xxxx xxxx group
 10    HLLL-LHH-L------  A.A..A.. * |      (except EIG)
 11    HLLL-LHHL-------  A.A.A... */
 12    HLLL-LHH---H----  .......A *--
 13    HLLL-LHHHHH-----  .AA..... *\
 14    HLLL-LHHHHH-H---  .AAA.... * | extended instruction group
 15    HLLL-LHHHHH--H--  .AA.A... * |
 16    HLLL-LHHHHH---H-  .AA..A.. * |
 17    HLLL-LHHHHH----H  .AA...A. */
 18    HLLLHLHL--------  A....... *\
 19    HLLLHLHLH-------  A...A... * |
 20    HLLLHLHL-H------  A....A.. * | 1000 1010 xxxx xxxx group
 21    HLLLHLHL--H-----  A.....A. * |     (except fps)
 22    HLLLHLHL---H----  A......A */
 23    HLLLHLHLL---LLLL  A..A.... *-- FP single
 24    HLLLHLHLL---LLHL  A..AA... *-- FP double
 25    LLLL-H----------  AAA..... *\      - default
 26    LLLL-HLLLLLHLL--  AAA.A... * | ASG  - SSA
 27    LLLL-HLLLLL-LLH-  AAA..A.. * |      - SZA
 28    LLLL-HLLLLL-LL-H  AAA...A. * |      - RSS
 29    LLLL-H-HLLLLL-LL  AAAA..A. * |      \ CLA CMA CCA
 30    LLLL-HH-LLLLL-LL  AAAA.A.. * |      /
 31    LLLL-H--LLLLLHLL  AAAAA... */      - INA
 32    LLLL-L----------  AA...... *\      - default
 33    LLLL-LLLLL------  AA.A.... * | SRG  - first NOP
 34    LLLL-L----H-----  AA..A... * |      - CLE
 35    LLLL-L------H---  AA...A.. * |      - SL@
 36    LLLL-L-----L-LLL  AA....A. */      - second NOP
```

```
0044    37    HLLLLLLH-LLLLLLL  .A....A. *\
0045    38    HLLLLLL-HLLLLLLL  .A.....A * | DIV, MPY, DLD, DST, JLA, JLB
0046    39    HLLLHLL-HLLLLLLL  .A...A.A * |
0047    40    HLLLHLLH-LLLLLLL  .A...AA. */
0048    41    HLLLLLL-LLLLH----  .......A *-- ASL and ASR
0049    42    HLLLLLL-LLLHL----  ......A. *-- LSL and LSR
0050    43    HLLLLLL-LLHLL----  ......AA *-- RRL and RRR
0051    44    HLLL-H----------  ..A..... *\  IO group
0052    45    HLLL-H----LL----  ..A....A * | IO group (low select codes)
0053    46    HLLL-H-LLL------  ..A...A. */  IO group (hlt)
0054    47    HLLLHLHLL---HHLL  A.AA.... *-- double integer
```

```
MPARA source listing
0000 MPARA; *control routines and I/O group instructions <820204.1442>
0000  $origin 0x000$ *file = &CONTR <820204.1442>
0000  ************************************************************************
0000  * (C) Copyright Hewlett Packard Company 1982. All rights reserved.  *
0000  * No part of this program may be photocopied, reproduced or         *
0000  * translated to another program language without the prior written  *
0000  * consent of Hewlett Packard Company.                                *
0000  ************************************************************************
0000  *820119 SRK - CLC 0 now turns off TBG; PFW does a CLC 4
0000  *820120 SRK - CLC 0 now enables PFW
0000
0000  ************************************************************************
0000  *   - Power-up routine and the beginning of the selftest              *
0000  *   - VCP boot routine                                                *
0000  *   - JTAB loops for instruction decoding:                            *
0000  *     - Normal JTAB loop                                              *
0000  *     - Diagnostic JTAB loop                                          *
0000  *     - Broadcasting JTAB loop                                        *
0000  *   - Interrupt handlers:                                             *
0000  *     - Diversion Handlers:                                           *
0000  *       - A/B fetch                                                   *
0000  *       - Time Base Generator Tick                                     *
0000  *     - Macro Interrupt Handlers                                      *
0000  *       - Parity Error                                                *
0000  *       - Unimplemented Instruction Trap                              *
0000  *       - Memory Protect Violation                                    *
0000  *       - Slave Handler (Break to VCP)                                *
0000  *       - Floating Point Overflow (Not supported)                     *
0000  *       - Time Base Generator Flag                                    *
0000  *       - External Interrupts (of the IO Master Variety)              *
0000  *     - Jam Handlers:                                                 *
0000  *       - Jump to Nonexistent Micromemory                             *
0000  *       - Microcode Timeout                                           *
0000  *     - I/O Group Instructions:                                       *
0000  *       - All HLT instructions                                        *
0000  *       - I/O Group with Select code >=20B (Executed by IOM)          *
0000  *       - I/O Group with Select code <=20B (Executed by IOM and CPU)*
0000  *     - General routines for hardware manipulation                    *
0000  ************************************************************************
0000
0000  *ist asynchronous clear/set codes
0000  $define dat/I_INT_CLR 0x000c$ *ist:  clear intf condition
0000  $define dat/I_PE_CLR  0x0000$ *      clear parity interrupt
0000  $define dat/I_PE_SET  0x0001$ *      set   parity interrupt
0000  $define dat/I_MP_CLR  0x0002$ *      clear memory protect interrupt
0000  $define dat/I_MP_SET  0x0003$ *      set   memory protect interrupt
0000  $define dat/I_MTO_CLR 0x0006$ *      clear mto jam indicator
0000  $define dat/I_TBGT_CL 0x0004$ *      clear tbgtick diversion
0000  $define dat/I_TBGT_SE 0x0005$ *      set   tbgtick diversion
0000  $define dat/PE_CODE   0x0009$ *      parity error interrupt code
0000
0000  *ist enable bits and backplane bits
0000  $define dat/I_PEE     0x0010$ *      parity error enable
0000  $define dat/I_PFWE    0x0020$ *      power fail warning enable
0000  $define dat/I_INTR    0x0100$ *      external interrupt mask
0000  $define dat/I_FLTO    0x0200$ *      floating point overflow bit
0000  $define dat/I_TBGOFF  0x0400$ *      time base generator on/off
0000  $define dat/I_PSODD   0x0800$ *      parity sense (0 is even)
0000  $define dat/I_MPD     0x1000$ *      memory protect disable
0000  $define dat/I_SCHOD   0x2000$ *      backplane schod line
```

```
0000  $define dat/I_CRS     0x4000$ *      backplane crs line
0000  $define dat/I_TBGF    0x8000$ *      time base generator flag
0000  $define dat/I_RESET   0x1C00$ *      ist reset value
0000
0000  *memr map bits and diagnostic window
0000  $define dat/MEMR_BOOT 0x0040$ *      bootmemory select bit
0000  $define dat/MEMR_AB   0x0020$ *      ab addr disabling bit
0000  $define dat/MEMR_TDI  0x8000$ *      tdi indicator
0000  $define dat/MEMR_PFW  0x4000$ *      pfw indicator
0000  $define dat/MEMR_MLST 0x2000$ *      mlost indicator
0000  $define dat/MEMR_MTO  0x1000$ *      mto indicator
0000  $define dat/MEMR_ABF  0x0800$ *      abfetch indicator
0000  $define dat/MEMR_PE   0x0400$ *      parity error indicator
0000  $define dat/MEMR_SLAV 0x0200$ *      slave indicator
0000
0000  *privileged register allocation
0000  $define dat/N_VMA_START    0$ *prin: start of VMA
0000  $define dat/N_CIR         10$ *      central interrupt register
0000  $define dat/N_TBGT_C      11$ *      missed TBG tick counter
0000  $define dat/N_MP_VIOLAT   12$ *      MP violation address
0000  $define dat/N_LR           9$ *      lights register (upper byte)
0000  $define dat/N_IMAP        13$ *      packed IMAP
0000  $define dat/N_ST          14$ *      status register (ST)
0000  $define dat/N_WMAP        15$ *      unpacked WMAP (just DATA1,DATA2)
0000
0000  *special register specification
0000  $define dat/N_CIL          3$ *srin: central interrupt latch
0000  $define dat/N_PEL1         1$ *      parity error latch 1 (low 16 bits)
0000  $define dat/N_PEL2         2$ *      parity error latch 2 (high 6 bits)
0000  $define dat/N_MPAR         0$ *      map address register
0000
0000  *status register definition (microcode status)
0000  $define dat/ST_GRDI   0x1000$ *st:   global register disabled
0000  $define dat/ST_LV2    0x0100$ *      level 2 enable
0000  $define dat/ST_LV3    0x0010$ *      level 3 enable
0000  $define dat/ST_LV23   0xFEEF$ *      level 2 and 3 enabled
0000  $define dat/ST_LV23T  0xFEEE$ *      level 2 and 3 and TBG_E enabled
0000  $define dat/ST_TBG_E  0x0001$ *      time base generator interrupt enable
0000  $define dat/ST_RESET  0x1101$ *      initial value GR di LV2 en TBG_E en
0000  $define dat/ST_FWID   0x0200$ *      firmware identification bit (see JNM)
0000
0000  *trap cell definition
0000  $define dat/TRAP_PFW      04$ *trap: power fail warning trap location
0000  $define dat/TRAP_PE       05$ *      parity error trap location
0000  $define dat/TRAP_MP       07$ *      memory protect interrupt
0000  $define dat/TRAP_TBG      06$ *      TBG trap location
0000  $define dat/TRAP_UIT     010$ *      UIT trap location
0000
0000  *external addresses and links
0000  $define adrl/SELFTEST 0x007$  *link to SELFTEST in &SELFTEST module
0000
0000  LOC_ZERO:  $origin 0$
0000  {
0000    Jam entrypoint --  handle the following conditions:
0000      PON - power coming up
0000      JNM - jump to non_existent micromemory
0000      MTO - microcode timeout
0000  }
0000    if pon goto not_power_up;   *Is power on or coming up?
0001    power_up:                   *Power is coming up.
0001    FAILTRAP:                   *
0001      ist:=0x0C00;              *  turn of tbg now--prevent MTO
0002      if not pon goto failloop; *  and also check the conditional
0003      if pon goto SELFTEST,     *  branching mechanism.
0003        acc:=acc xor acc;       *  Execute SELFTEST
0004    failloop:                   *Selftest will loop here on failure
0004      goto failloop;            *Should not get here,
0005                                *  put the brakes on if possible.
0005    not_power_up:               *Is conditional branching bad?
0005      if not pon goto FAILTRAP; *  y: shut of tbg and enter trap
0006      goto int_mtojnm;          *  n: must be JNM or MTO
0007
00CF  START_UP: $origin 0x0cF$      *link from &SELFTEST module
00CF  {
00CF    Initialize the Phoenix and begin executing macroinstructions.
```

```
00CF  }
00CF    call BOOT;                  *Prepare to enter VCP.
00D0
00D0  INST_RESTART:
00D0  PROCESS_DISPATCH: $origin 0x0D0$
00D0  {
00D0    Fetch the first instruction of the instruction stream.
00D0    Do not move this location.  (Link from many routines.)
00D0  }
00D0    fchp,                       *Fetch the first instruction
00D0      acc:=ones,                *  (acc=ones is used by many routines)
00D0      goto vector;              *
00D1
00D1  vector:
00D1  {
00D1    Execute the process's instruction stream until an
00D1    interrupt or exception condition comes true.
00D1    Decide on which jtab to use.
00D1    These locations must not be moved for WCS switching to work.
00D1  }
00D1    n:=N_VMA_START;             *prepare to null VMA register
00D2    if intf goto interrupt,     *first fetch interrupted?
00D2      nop:=r14(sr);             *look for diagnostic looping
00D3    if not y15 goto jtab,       *should we use the diagnostic JTAB loop?
00D3      prin:=ones;               *set the VMA register to ones
00D4    goto jtab_diag;
00D5
00DD  jtab: $origin 0x0DD$
00DD  {
00DD    Normal jtab location.
00DD    Loop until interrupt.
00DD    Do not move this location for mind-swaps to occur without UI.
00DD  }
00DD    jtab,                       *Subroutine call to macroinstruction
00DD                                *  emulation routine.
00DD      clf,                      *  put the flag into known state
00DD      ip,                       *  Inc P to address after opcode.
00DD      cwrb:=b, bbus/t;          *  Force orders that are required for
00DE                                *    the MRG decode to begin memory
00DE                                *    references in this cycle.
00DE    if not intf goto jtab,      *Loop until interrupt causes
00DE      acc:=ones;                *  a fetching to be ignored.
00DF  interrupt:                    *INTERRUPT:
00DF    goto int_vector,            *  branch to interrupt handler
00DF      ct:=ist;                  *  load priority code into ist
00E0
00E0  jtab_diag:  $origin 0x0E0$
00E0  {
00E0    Jtab location with diagnostic window.
00E0    Loop until interrupt.
00E0  }
00E0    nop:=t;                     *viewing of instruction on YBUS.
00E1    nop:=fa;                    *viewing of fetch address.
00E2    jtab,                       *Subroutine call to macroinstruction routine
00E2      clf,                      *  put flag into known state
00E2      ip,                       *  Inc P to address after opcode.
00E2      cwrb:=b, bbus/t;          *  Force orders that are required for
00E3                                *    the MRG decode to begin memory
00E3                                *    references in this cycle.
00E3    nop:=a;                     *A reg
00E4    nop:=b;                     *B reg
00E5    nop:=ist;                   *ist reg
00E6    nop:=memr;                  *memr reg
00E7    n:=N_ST;                    *
00E8    nop:=prin;                  *microcode status register
00E9    if not intf goto            *Loop until an interrupt condition
00E9      jtab_diag,                *  caused the fetch to be ignored.
00E9      acc:=ones;                *  Set acc to -1 for decrementing.
00EA    goto int_vector,            *INTERRUPT out of JTAB loop
00EA      ct:=ist;                  *
00EB
00EF  int_vector: $origin 0x00EF$
00EF  {
00EF    BRANCH TO INTERRUPT HANDLER
```

```
00EF    Remember:
00EF      DW might be enabled here
00EF      ACC is assumed to hold "ones" for many routines
00EF      Interrupt holdoff may be enabled in certain cases.
00EF  }
00EF    goto int_table, ct30,       *do table jump into interrupt table
00EF      ist:=ist;                 *clear INT flip-flop
00F0
00F0  int_table:  $origin 0x00F0$
00F0  {
00F0    Branch to the interrupt or diversion that caused
00F0    the fetch to be ignored.
00F0    Locations 6-F are entered because of INTF handling.
00F0    UIT, MTO, JNM just branch through here for
00F0    1610 viewing.
00F0  }
00F0  {00} UIT:                     *Unimplemented instruction trap.
00F0         goto INT_UIT, p:=fa;   *
00F1  {01} JNM:                     *Jump to non-existent micromemory
00F1         goto INT_UIT, p:=fa;   *
00F2  {02} MTO:                     *Microcode timeout
00F2         goto INT_UIT, p:=fa;   *
00F3  {03} PRIV_TRAP: goto int_priv;*Privileged instruction interrupt
00F4  {04} goto INT_UIT, p:=fa;     *hardware failure
00F5  {05} goto INT_UIT, p:=fa;     *hardware failure
00F6  {06} goto INT_UIT, p:=fa;     *hardware failure
00F7  {07} ct:=t, goto VECTOR;      *ab fetch diversion
00F8  {10} goto tbg_tick;           *time base generator tick diversion
00F9  {11} goto int_pe;             *parity error macrointerrupt
00FA  {12} goto int_mp, p:=fa;      *memory protect macrointerrupt
00FB  {13} goto int_slav;           *slave macrointerrupt
00FC  {14} goto int_pfw;            *power fail warning macrointerrupt
00FD  {15} goto int_flv;            *floating point overflow macrointerrupt
00FE  {16} goto int_tbgf;           *time base generator macrointerrupt
00FF  {17} goto int_ext,            *external (I/O) macrointerrupt
00FF         p:=p+acc;              *decrement p (will be ip'd at JTAB)
0100
0100  JFPLA_UIT: $origin 0x100$     *JTABLE entrypoint if no fpla terms match
0100    goto UIT, nop:=ct;          *
0101
008A  ab_fetch: $origin 0x008a$
008A  {
008A    The fetch that occurred was from the a or b registers.
008A    Process the a or b fetch, and branch to the appropriate
008A    vector point. This is not a macrointerrupt.
008A    Was handled in INT_TABLE.
008A  }
008A
008A  tbg_tick:
008A  {
008A    Process a time base generator tick.
008A    This is not a macrointerrupt.
008A  }
008A    sl:=ist and 0xfff0;         *
008B    ist:=I_TBGT_CLR ior sl;     *Clear TBGtick
008C    n:=N_TBGT_COUNTER;          *
008D    call SET_TBG_INT,           *Enable TBG interrupt
008D      prin:=prin-acc;           *Increment the TBG tick counter
008E    goto PROCESS_DISPATCH;      *Redispatch the process.
008F
008F  int_pe:
008F  {
008F    Parity error interrupt.
008F  }
008F    sl:=ist and 0xffe0;         *
0090    ist:=sl ior I_MP_CLR;       *Null a lower priority MP interrupt
0091                                * and shut off PE int
0091    s3:=TRAP_PE;                *save the trap cell address
0092    ist:=sl ior I_PE_CLR;       *Null the PE interrupt and shut off PE int
0093    goto TRAP_CELL_FETCH,       *do trap cell fetch
0093      p:=p+acc;                 *decrement p (will be ip'd at vector)
0094
0094  int_priv:
0094  {
0094    Privileged instruction interrupt.
0094    (Just like memory protect.)
```

```
0094  }
0094    s1:=ist and 0xfff0;         *set the memory protect interrupt
0095    ist:=s1 ior I_MP_SET;       *
0096    goto PROCESS_DISPATCH;      *and refetch
0097
0097  int_mp:
0097  {
0097    Memory protect interrupt.
0097  }
0097    s3:=TRAP_MP;                *save trap cell addr
0098    s1:=ist and 0xfff0;         *
0099    ist:=I_MP_CLR ior s1;       *clear interrupt
009A    n:=N_MP_VIOLATION;          *save violation address
009B    goto TRAP_CELL_FETCH,       *perform trap cell fetch
009B      prin:=fa;                 *set IVR to fetch address
009C                                *p is set to fetch address, which will
009C                                *  be incremented in the jtab line.
009C                                *  the trap cell JSB will always write
009C                                *  FA+1 into the return address NOP.
009C
009C  int_slave:
009C  {
009C    Process the slave condition - goto virtual control panel mode
009C  }
009C    call PULSE_SCHOD;           *Assert and release SCHOD.
009D    call IOHS, bbus/t;          *Perform the IOHS.
009E    goto PROCESS_DISPATCH;      *  VCP process.
009F
009F  int_pfw:
009F  {
009F    Process the power fail warning interrupt.
009F  }
009F    s3:=TRAP_PFW;               *save trap cell address
00A0    call CLR_LV2;               *implicit CLC 4
00A1    goto TRAP_CELL_FETCH,       *do trap cell fetch
00A1      p:=p+acc;                 *decrement p (will be ip'd at JTAB)
00A2
00A2  int_flv:
00A2  {
00A2    Process the floating point overflow interrupt.
00A2  }
00A2    s3:=TRAP_UIT;               *save trap cell address
00A3    ist:=not I_FLTO and ist;    *
00A4    goto TRAP_CELL_FETCH,       *do trap cell fetch
00A4      p:=p+acc;                 *decrement p (will be ip'd at jtab)
00A5
00A5  int_tbgf:
00A5  {
00A5    Process the time base generator flag interrupt.
00A5  }
00A5    ist:=ist and not I_TBGF;    *clear the TBG flag
00A6    s3:=TRAP_TBG;               *save trap cell address
00A7    n:=N_TBGT_COUNTER;          *
00A8    call SET_TBG_INT,           *enable tbg int only if qualifiers are true
00A8      prin:=prin+acc;           *decrement TBGT counter
00A9    goto TRAP_CELL_FETCH,       *do trap cell fetch
00A9      p:=p+acc;                 *decrement P (will be ip'd at JTAB)
00AA                                *
00AA  int_ext:
00AA  {
00AA    Process an external interrupt from an io card.
00AA  }
00AA    call WMAP_PACK, n:=ones;    *s0 = current WMAP
00AB    ist:=ist ior I_MPD;         *turn off memory protect
00AC    prin:= memr ior 0x2020;     *new DATA1 = old EXECUTE
00AD    memr:=0xffc0 and memr;      *new EXECUTE = zero
00AE    ifch, acc:=ones;            *Do the interrupt fetch (no a b addr!)
00AF    n:=N_IMAP;                  *
00B0    prin:=s0;                   *load IMAP
00B1    n:=N_CIL;                   *
00B2    ct:=t, clf;                 *Load trap cell into instruction register
00B3    s0:=0x003f and srin;        *Store Central Interrupt Latch
00B4    n:=N_CIR;                   *
00B5    prin:=s0,                   *into Central Interrupt Register
00B5      goto VECTOR;              *and execute trap cell
```

```
00B6
00B6
00B6  int_uit:
00B6  {
00B6    Process the unimplemented instruction interrupt
00B6  }
00B6    nop:=memr and MEMR_PE;      *If uit was caused by a PE on the fetch,
00B7    if yz goto uit_not_pe,      *  then PE interrupt has priority.
00B7      acc:=ones;                *
00B8  uit_is_pe:                    *is PE pending?
00B8    goto PROCESS_DISPATCH;      *  y: let it happen
00B9  uit_not_pe:                   *  n: do a UIT
00B9    s3:=TRAP_UIT;               *
00BA
00BA  TRAP_CELL_FETCH:              *Perform trap cell fetch for low SC's
00BA    call WMAP_PACK, n:=ones;    *s0 = current WMAP
00BB    prin:=memr ior 0x2020;      *new DATA1 = old EXECUTE
00BC    memr:=memr and 0xFFC0;      *new EXECUTE = zero
00BD    ist:=ist ior I_MPD;         *disable memory protect
00BE    n:=N_CIR, bbus/memr;        *load CIR with code
00BF    if not b15 goto tdireset,   *(is tdi set?)
00BF      rdb, prin:=s3;            *read trap cell contents
00C0    cmid;                       *(make sure TDI is reset)
00C1  tdireset:                     *
00C1    n:=N_IMAP;                  *load IMAP with old WMAP
00C2    prin:=s0, clf;              *
00C3    ct:=t,                      *load trap cell into instruction register
00C3      goto VECTOR;              *and execute trap cell
00C4
00C4  int_mtojnm:
00C4  {
00C4    Process the microcode timeout and jump to nonexistent micromemory
00C4    interrupts.
00C4  }
00C4      nop:=memr and MEMR_MTO;   *MTO  indicates that
00C5      if yz goto jnm_assume,    *the microcode time out occurred.
00C5        acc:=ones;              *  (acc must = ones for TRAP_CELL)
00C6      s1:=0xfff0 and ist;       *MTO: clear out indicator
00C7      ist:=I_MTO_CLR ior s1;    *  and process as a UIT
00C8      n:=N_TBGT_COUNTER;        *
00C9      goto MTO,                 *  make up for the lost tbg tick
00C9        prin:=prin-acc;         *
00CA     jnm_assume:                *JNM: process as a UIT
00CA      nop:=prin and ST_FWID;    *  (If firmware identification in progress
00CB      if yz goto JNM;           *   then do not UIT!)
00CC      prin:=prin and            *firmware identification in progress!!
00CC        not ST_FWID;            *  clear bit
00CD      a:=ones, fchp, rtn;       *  a:=ones, and return
00CE
00CE  **********************************************************************
00CE  * Input/Output Group Macroinstructions
00CE  * organization:
00CE  *    -the jump table microcode for
00CE  *       - IO Group HLT instructions (all select codes)
00CE  *       - IO Group instructions with select code >= 20B
00CE  *       - IO Group instructions with select code <= 20B
00CE  *    -a decode table for select code <= 20B
00CE  *    -the microcode that executes the low select code instructions
00CE  **********************************************************************
00CE
0120  IOG_HI: $origin 0x120$        *high select code instructions <> HLT
0120    goto IOG_HIGH;              *  (SC >= 20B)
0121  IOG_LO: $origin 0x121$        *low select code instructions <> HLT
0121    gototbl SC_TABLE, stor/n;   *  (SC < 20B)
0122  HLT_HI: $origin 0x122$        *high select code HLT
0122    goto HLT;                   *  (SC >= 20B)
0123  HLT_LO: $origin 0x123$        *low select HLT
0123    goto HLT;                   *  (SC < 20B)
0124
04C0  $origin 0x04c0$
04C0  SC_TABLE:                     *THE LOW SELECT CODES ARE DEDICATED TO:
04C0  {0} goto SC_00, n:=acc+acc;   *interrupt system flag and interrupt mask
04C1  {1} goto SC_01, ct:=srg(ct),  *O register instructions
04C1        cmid;                   *and lights and switches
04C2  {2} goto SC_02, n:=acc+acc;   *global register
```

```
04C3  {3} goto SC_03, n:=acc+acc;   *virtual control panel
04C4  {4} goto SC_04, n:=acc+acc;   *power fail
04C5  {5} goto SC_05, n:=acc+acc;   *parity error
04C6  {6} goto SC_06, n:=acc+acc;   *time base generator
04C7  {7} goto SC_07, n:=acc+acc;   *memory protect
04C8  {8} goto SC_10, n:=acc+acc;   *
04C9  {9} goto SC_11, n:=acc+acc;   *
04CA  {a} goto SC_12, n:=acc+acc;   *
04CB  {b} goto SC_13, n:=acc+acc;   *
04CC  {c} goto SC_14, n:=acc+acc;   *
04CD  {d} goto SC_15, n:=acc+acc;   *
04CE  {e} goto SC_16, n:=acc+acc;   *
04CF  {f} goto SC_17, n:=acc+acc;   *
04D0
04D0  ************************************************************************
04D0  * Select Code 00 Decoding and Execution                                *
04D0  ************************************************************************
04D0  sc_00_tbl: $align 16$
04D0  {stf_00:} goto SET_LV3;       *STF - set Interrupt System Flag
04D1  {mi@_00:} goto IOHS, bbus/t;  *MI@ - merge A w/Interrupt Mask Register
04D2  {clf_00:} rtn;                *CLF - clr Interrupt System Flag
04D3  {li@_00:} goto IOHS, bbus/t;  *LI@ - load A w/Interrupt Mask Register
04D4  {sf@_00:} goto sf@00,nop:=s0; *SF? - skip if ISF
04D5  {ot@_00:} goto ot@00, bbus/t; *OT@ - output A to Interrupt Mask Register
04D6  {sf@_00:} goto sf@00,nop:=s0; *SF? - skip if ISF
04D7  {@lc_00:} if not y15 then rtn;*STC - NOP
04D8  clc00:                        *CLC - IO system reset
04D8    goto clc_00;                *
04D9  sc_00:                        *DECODE SC_00:
04D9    call BCST;                  * broadcast this instruction
04DA    cmid, ct:=srg(ct);          * set tdi, transform instruction for decoding
04DB    if mpen goto priv_00,       * check memory protect
04DB      s0:=111(ct);              * set sf on the ',C' bit
04DC    call sc_00_tbl, ct74,       * execute instruction
04DC      nop:=r14(ct);             * set y15 on the STC vs CLC bit
04DD    if sf call clf_00;          * if ',C' then execute CLF_00
04DE    fchp, rtn;                  * that's all
04DF  priv_00:                      *
04DF    goto PRIV_TRAP;             *link to PRIV_TRAP
04E0
04E0  ************************************************************************
04E0  * Select Code 01 Decoding and Execution                                *
04E0  ************************************************************************
04E0  sc_01_tbl: $align 16$
04E0  {stf_01:} rtn, sto;           *STO - set O register
04E1  {mi@_01:} goto mi@01,         *MI@ - merge from switch register
04E1              s0:=cab;          *
04E2  {clf_01:} rtn, clo;           *CLO - clear O register
04E3  {li@_01:} goto li@01,         *LI@ - lbad from switch register
04E3              s0:=zero;         *
04E4  {sf@_01:} goto sf@01,nop:=s0; *SF? - skip if o set or clear
04E5  {ot@_01:} goto ot@01,         *OT@ - store into LED register
04E5              s0:=zuy(cab);     *      zero out microcode side of switches
04E6  {sf@_01:} goto sf@01,nop:=s0; *SF? - skip if o set or clear
04E7  {@lc_01:} rtn;                *STC,CLC - NOP
04E8                                *
04E8  SC_01:                        *DECODE SC_01:
04E8    call sc_01_tbl, ct74,       *  call the select code table
04E8      s0:=111(ct);              *  set sf on the ',C' bit of instruction
04E9    if sf then clo;             *  if ',C' then clear 0
04EA    fchp, rtn;                  *  that's all
04EB  sf@01:                        *
04EB    if y15 goto sfs01;          *
04EC  sfc01:                        *SOC - skip if 0 clear
04EC    if o then rtn;              *
04ED    ip, rtn;                    *
04EE  sfs01:                        *SOS - skip if 0 set
04EE    if not o then rtn;          *
04EF    ip, rtn;                    *
04F0  ot@01:                        *OT@ - output to lights register
04F0    n:=N_LR;                    *  (The upper half of the lights register
04F1    s0:=prin ior s0;            *
04F2    lr:=not s0, rtn;            *
04F3  mi@01:                        *   The value is kept in N_LR in prin.)
04F3  li@01:                        *load from switch register
```

```
04F3   s1:=sr;                      * (must be compatible with L-series)
04F4   ct:=ct ior 0x00ff;           *use opcode as count
04F5 li@lp:                         *
04F5   s1:=lll(s1), lwf;            *reverse switches!  (L-series funny)
04F6   s2:=lrl(s2), lwf;            * low s1-s8 go in bits 15-8
04F7   if not ctz4 goto li@lp,      * do 16 times
04F7     s2:=zly(s2);               * zero out lower switches
04F8   nop:=lll(ct);                *restore SF to ,C request
04F9   s2:=s2 and not 0xC000;       *kill switches 14,15
04FA   if not b15 then rtn,         *if switch 15 was zero
04FA     cab:=s0 ior s2;            *  that’s all (do MI@ and LI@)
04FB   nop:=MEMR_MLST and memr;     *and if memory is not lost
04FC   if not yz then rtn;          *
04FD   cab:=cab ior 0x4000;         *then set bit 14 of CAB
04FE   rtn;                         *NOTE: did not do TBG_E of L-series
04FF                                *      (only used for diagnostic purposes)
04FF
04FF clf_00:                        *execute the ",C" function and CLF function
04FF   goto CLR_LV3;                *  for SC00
0500
0500 *************************************************************************
0500 * Select Code 02 Decoding and Execution                                 *
0500 *************************************************************************
0500
0500 $align 64$
0500 sc_02_tbl: $align 16$
0500 {stf_02:} goto stf02;          *STF - disable global register state
0501 {mi@_02:} goto IOHS, bbus/t;   *MI@ - merge @ w/ global register
0502 {clf_02:} rtn;                 *CLF - enable global register state
0503 {li@_02:} goto IOHS, bbus/t;   *LI@ - load @ w/ global register
0504 {sf@_02:} goto sf@02,nop:=s0;  *SF? - skip if GR
0505 {ot@_02:} goto fakeiohs,       *OT@ - output @ to global register
0505           bbus/t;              *
0506 {sf@_02:} goto sf@02,nop:=s0;  *SF? - skip if GR
0507 {@lc_02:} rtn;                 *STC,CLC - NOP
0508
0508 SC_02:                         *DECODE SC_02:
0508   call BCST;                   *  broadcast for IO master
0509   cmid, ct:=srg(ct);           *  set TDI, transform opcode for decode table
050A   if mpen goto priv_02,        *  kill ’em if memory protect is on
050A     s0:=lll(ct);               *  save ’,C’ state in SF
050B   call sc_02_tbl, ct74,        *  call decode table
050B     nop:=rl4(ct);              *  set y15 to SFC vs SFS opcode
050C   if sf call clf_02;           *  if ’,C’ then CLF_02
050D   fchp, rtn;                   *  that’s all
050E priv_02:                       *
050E priv_04:                       *
050E priv_05:                       *
050E priv_06:                       *
050E   goto PRIV_TRAP;              *link to PRIV_TRAP
050F
050F clf_02:                        *execute the ",C" function (and CLF)
050F   goto clf02;                  *
0510
0510 *************************************************************************
0510 * Select Code 03 Decoding and Execution                                 *
0510 *    (all instructions in this select code are NOPs that                *
0510 *     are not broadcast.)                                               *
0510 *************************************************************************
0510
0510 *************************************************************************
0510 * Select Code 04 Decoding and Execution                                 *
0510 *************************************************************************
0510 sc_04_tbl: $align 16$
0510 {stf_04:} rtn;                 *STF - NOP
0511 {mi@_04:} goto mi@04,s1:=cab;  *MI@ - merge A w/ Central Interrupt Register
0512 {clf_04:} rtn;                 *CLF - NOP
0513 {li@_04:} goto li@04,s1:=zero;*LI@ - load A w/ Central Interrupt Register
0514 {sf@_04:} goto sf@04,nop:=s0;  *SF? - skip if IIF
0515 {ot@_04:} goto ot@04;          *OT@ - output A to Central Interrupt Register
0516 {sf@_04:} goto sf@04,nop:=s0;  *SF? - skip if IIF
0517 {@lc_04:} if y15 goto clc_04;  *
0518 stc_04:                        *STC - clear Interrupt Inhibit Flag
0518   goto SET_LV2;                *      set level 2 interrupt enable
0519 clc_04:                        *CLC - set Interrupt Inhibit Flag
```

```
0519    goto CLR_LV2;               *      clr level 2 interrupt enable
051A  sc_04:                        *DECODE SC_04:
051A    cmid, ct:=srg(ct);          * set TDI, transform instruction
051B    if mpen goto priv_04,       * flog him if memory protect is enabled
051B      s0:=111(ct);              * set SF if ',C' is set
051C    call sc_04_tbl, ct74,       * decode instruction
051C      nop:=r14(ct);             * set Y15 on STC vs CLC bit
051D    fchp, rtn;                  * that's all folks
051E
051E  odd_ps:                       *continuation of CLF05
051E    ist:=ist ior I_PSODD;       *
051F    rtn;                        *
0520
0520  **************************************************************************
0520  * Select Code 05 Decoding and Execution                                  *
0520  **************************************************************************
0520  sc_05_tbl: $align 16$
0520  {stf_05:} ist:=ist and not    *STF - set parity sense to even (0)
0520                 I_PSODD;       *
0521  {mi@_05:} rtn;                *MI@ - merge A w/ parity violation address
0522   clf_05:  goto ODD_PS;        *CLF - set parity sense to odd (1)
0523  {li@_05:} goto li@05;         *LI@ - load A w/ parity violation address
0524  {sf@_05:} goto sf@05,nop:=s0; *SF? - skip if PS
0525  {ot@_05:} rtn;                *OT@ - output A to parity violation address
0526  {sf@_05:} goto sf@05,nop:=s0; *SF? - skip if PS
0527  {@lc_05:} if y15 goto clc05;  *
0528  stc05:                        *STC - enable parity interrupts
0528    ist:=ist ior I_PEE;
0529    rtn;                        *
052A  clc05:                        *CLC - disable parity interrupts
052A    ist:=ist and not I_PEE;
052B    rtn;                        *
052C  SC_05:                        *DECODE SC_05:
052C    cmid, ct:=srg(ct);          *  set TDI, transform instruction
052D    if mpen goto priv_05,       *  zmag him if memory protect is enabled
052D      s0:=111(ct);              *  set SF on ',C' bit
052E    call sc_05_tbl, ct74,       *  decode instruction
052E      nop:=r14(ct);             *  set Y15 on STC vs CLC bit
052F    fchp, rtn;                  *  that's all
0530
0530  **************************************************************************
0530  * Select Code 06 Decoding and Execution                                  *
0530  **************************************************************************
0530  sc_06_tbl: $align 16$
0530  {stf_06:} goto stf06;         *STF - set Time Base Generator Flag
0531  {mi@_06:} rtn;                *MI@ - NOP
0532   clf_06:  goto clf06;         *CLF - clr Time Base Generator Flag
0533  {li@_06:} rtn;                *LI@ - NOP
0534  {sf@_06:} goto sf@06,nop:=s0; *SF? - skip if TBGF
0535  {ot@_06:} rtn;                *OT@ - NOP
0536  {sf@_06:} goto sf@06,nop:=s0; *SF? - skip if TBGF
0537  {@lc_06:} if y15 goto clc_06; *
0538  stc_06:                       *STC - turn on Time Base Generator
0538    ist:=ist and not I_TBGOFF;  *
0539    rtn;                        *
053A  clc_06:                       *CLC - turn off Time Base Generator
053A    goto OF_TBG;                *
053B  sc_06:                        *DECODE SC_06:
053B    cmid, ct:=srg(ct);          * set TDI, transform instruction
053C    if mpen goto priv_06,       * plop him if memory protect is enabled
053C      s0:=111(ct);              * set SF on ',C' bit
053D    call sc_06_tbl, ct74,       * decode instruction
053D      nop:=r14(ct);             * set Y15 on STC vs CLC bit
053E    if sf call clf_06;          * if ',C' call CLF_06
053F    fchp, rtn;                  * that's all
0540
0540  **************************************************************************
0540  * Select Code 07 Decoding and Execution                                  *
0540  **************************************************************************
0540  sc_07_tbl: $align 16$
0540  {stf_07:} rtn;                *STF - NOP
0541  {mi@_07:} goto mi@07,s1:=cab; *MI@ - merge A w/ violation register
0542   clf_07:  rtn;                *CLF - NOP
0543  {li@_07:} goto li@07,s1:=zero;*LI@ - load A w/ violation register
0544  {sf@_07:} rtn;                *SF? - NOP
```

```
0545 {ot@_07:} goto ot@07;        *OT@ - output A into violation register
0546 {sf@_07:} rtn;               *SF? - NOP
0547 {@lc_07:} if y15 then rtn;   *CLC - NOP
0548 stc07:                       *STC - enable memory protect until interrupt
0548   ist:=ist and not I_MPD;    *
0549   rtn;                       *
054A sc_07:                       *DECODE SC_07:
054A   cmid, ct:=srg(ct);         * set TDI, transform instruction
054B   if mpen goto priv_07,      * sqsh her if memory protect is enabled
054B     s0:=111(ct);             * set SF to ',C' bit
054C   call sc_07_tbl, ct74,      * decode instruction
054C     nop:=r14(ct);            * set Y15 on STC vs CLC bit
054D   fchp, rtn;                 * that's all
054E priv_07:                     *
054E   goto PRIV_TRAP;            *link to PRIV_TRAP
054F
054F **********************************************************************
054F * IOHS and BCST utilities                                            *
054F **********************************************************************
054F
0550 iohs_tbl: $align 16$         *execute the control word.
0550 {00} rtn;                    *  nop
0551 {01} rdio, goto iohs_ldp;    *  load P from backplane
0552 {02} rdio, goto iohs_lda;    *  load A from backplane
0553 {03} rdio, goto iohs_ldb;    *  load B from backplane
0554 {04} sto, rtn;               *  STO
0555 {05} clo, rtn;               *  CLO
0556 {06} rdio, goto iohs_mi@;    *  merge data into *
0557 {07} ip, rtn;                *  inc(P)
0558 {10} rtn;                    *  undefined
0559 {11} goto iohs_boot;         *  enable bootstrap roms
055A {12} nop:=a, wrio, rtn;      *  put A on backplane
055B {13} nop:=b, wrio, rtn;      *  put B on backplane
055C {14} cle, rtn;               *  CLE
055D {15} ste, rtn;               *  STE
055E {16} nop:=p, wrio, rtn;      *  put P on backplane
055F {17} nop:=p, wrio, ip, rtn;  *  put P on backplane, inc(P)
0560
0560
0560 IOHS:
0560 {
0560  Perform an IO Handshake with the L-series backplane
0560    Calling sequence:
0560      call BCST;        *broadcast the instruction at FA
0560      call IOHS, bbus/t; *must freeze! do IO handshake
0560 }
0560   clf;                       *F will be set if IOHS performed
0561   ;                          *must wait for IORQ
0562   if not iorq then rtn;      *if no IORQ now then never.
0563 beginloop:                   *
0563   rdio,                      *read first command word
0563     s7:=ct;                  *  (save instruction)
0564 iohs_loop:                   *LOOP
0564   stf;                       *set flag to indicate IOHS performed
0565   nop:=0x0100 and t;         *freeze until I/O device returns CW
0566   if yz then goto iohs_quit, *branch if this is the last IOHS
0566     ct:=t;                   *  prepare for table jump
0567   call iohs_tbl, ct74,       *call IOHS table
0567     ct:=s7;                  *  restore CW
0568 iorq_wait:                   *
0568   if not iorq goto iorq_wait; *wait for IORQ (IOM has failed if forever)
0569   goto iohs_loop,            *  loop and wait for iorq.
0569     rdio, s7:=ct;            *
056A
056A iohs_quit:                   *last execution of loop
056A   goto iohs_tbl, ct74,       *goto IOHS table (last IOHS)
056A     ct:=s7;                  *  (save CW)
056B
056B iohs_ldp:                    *load P
056B   p:=t, rtn;                 *
056C iohs_lda:                    *load A
056C   a:=t, rtn;                 *
056D iohs_ldb:                    *load B
056D   b:=t, rtn;                 *
056E iohs_mi@:                    *merge *
```

```
056E   s7:=cab;                    *
056F   cab:=s7 ior t, rtn;         *
0570 iohs_boot:                    *enable boot rom.
0570   memr:=0x0040 ior memr;      *  enable rom bit
0571   call WMAP_PACK, n:=ones;    *store the WMAP in location 2 of
0572   s1:=0100;                   *bootmemory
0573   ist:=ist ior I_MPD;         *  disable MP
0574   cmid,                       *  must do tdi to prevent IO interrupt
0574     wrb:=s0, bbus/s1, rtn;    *  write packed WMAP into boot loc 100
0575
0575 BCST:
0575 {
0575   Broadcast routine
0575   BCST uses the BFB special to refetch the operand with RNI-.
0575     ABFetches are written into 0or1 of bootram and BFBed
0575     from there.
0575 }
0575   nop:=0x0800 and memr;       *if not AB fetch then
0576   if not yz then bfb,         *  then do backplane fetch of opcode
0576     bbus/fa;                  *
0577   if not yz then rtn,         *  and return
0577     s7:=ct;                   *
0578 bcst_abf:                     *else broadcast ABFetch
0578   s6:=memr;                   *save current map
0579   memr:=0x0060;               *load MEMR with bootmemory mode (no abaddr)
057A   wrb:=s7, bbus/fa;           *write instruction into bootmemory.
057B   bfb, bbus/fa;               *fetch instruction from bootmemory,
057C                               *  assert RNI- and freeze
057C   memr:=s6, bbus/t, rtn;      *  before restoring memr.
057D
057D
057D iog_mp:
057D   goto PRIV_TRAP;
057E
057E ************************************************************************
057E * Select Codes >= 20, HLT and Unused Select Code Execution             *
057E ************************************************************************
057E sc_03:                        *not implemented
057E sc_10:                        *not implemented
057E sc_11:                        *not implemented
057E sc_12:                        *not implemented
057E sc_13:                        *not implemented
057E sc_14:                        *not implemented
057E sc_15:                        *not implemented
057E sc_16:                        *not implemented
057E sc_17:                        *not implemented
057E IOG_HIGH:                     *
057E HLT:                          *HLT AND HIGH SELECT CODE IO INSTRUCTIONS
057E   if mpen goto iog_mp;        *check memory protect
057F   call BCST, acc:=zeros;      *broadcast
0580   call IOHS, nop:=t;          *
0581   cmid;                       *do interrupt holdoff
0582   fchp, rtn;                  *
0583
0583 *****************************************************************************
0583 * Continuation of Select Code <= 20 Instructions                            *
0583 *****************************************************************************
0583
0583 *                             *SELECT CODE 00:
0583 sf@00:                        *branch on sfc vs sfs
0583   if y15 goto sfs00;          *
0584 sfc00:                        *skip if isf set
0584   nop:=ST_LV3 and prin;       *
0585 sfc@@:                        *(general purpose skip if flag set)
0585   if not yz then rtn;         *
0586   ip, rtn;                    *
0587 sfs00:                        *skip if isf clear
0587   nop:=ST_LV3 and prin;       *
0588 sfs@@:                        *(general purpose skip if flag clr)
0588   if yz then rtn;             *
0589   ip, rtn;                    *
058A ot@00:                        *output to interrupt mask register
058A   prin:=not ST_TBG_E and prin;*clear out TBG enable bit
058B   nop:=cab and 0x0002;        *should I enable TBG?
058C   if not yz goto ot@00f;      *
```

```
058D    prin:=prin ior ST_TBG_E;     *yes: set my flag and
058E    call SET_TBG_INT;            *  enable it if all qualifiers are enabled
058F  ot@00f:                        *no: continue
058F  fakeiohs:                      *    perform fake IOHS (CPU drives CW)
058F    sl:=12;                      *fake out IO control word for writing
0590    nop:=0x0800 and ct;          *  A or B to backplane
0591    if yz then fcin,             *if OTA then control word = 12
0591      sl:=sl;                    *       else control word = 13
0592    nop:=sl, wrio;               *write control word to backplane
0593    nop:=cab, wrio, rtn;         *write A or B to backplane
0594
0594  * Select code 2
0594  stf02:                         *Disable global register
0594    prin:=ST_GRDI ior prin;      *   state
0595    rtn;                         *
0596  clf02:                         *Enable global register
0596    prin:=not ST_GRDI and prin;  *   state
0597    rtn;                         *
0598  sf@02:                         *
0598    if y15 goto sfs02;           *
0599  sfc02:                         *
0599    nop:=ST_GRDI and prin;       *skip if GR is enabled
059A    goto sfc@@;                  *
059B  sfs02:                         *
059B    nop:=ST_GRDI and prin;       *skip if GR is disabled
059C    goto sfs@@;                  *
059D  sf@04:                         *
059D    if y15 goto sfs04;           *
059E  sfc04:                         *skip if power going down
059E    nop:=memr and MEMR_PFW;      *
059F    goto sfc@@;                  *
05A0  sfs04:                         *skip if power coming up
05A0    nop:=memr and MEMR_PFW;      *
05A1    goto sfs@@;                  *
05A2  li@04:                         *load from central interrupt register
05A2  mi@04:                         *merge from central interrupt register
05A2    n:=N_CIR;                    *
05A3    cab:=prin ior sl, rtn;       *
05A4  ot@04:                         *output to central interrupt register
05A4    n:=N_CIR;                    *
05A5    prin:=cab and 077;           *
05A6    rtn;                         *
05A7
05A7  *SELECT CODE 5, CONTINUED
05A7  sf@05:
05A7    if not y15 goto sfc05;       *
05A8  sfs05:                         *skip if parity sense is even
05A8    sl:=I_PSODD and ist;         *
05A9    goto sfc@@;                  *
05AA  sfc05:                         *skip if parity sense is odd
05AA    sl:=I_PSODD and ist;         *
05AB    goto sfs@@;                  *
05AC  li@05:                         *load from parity error register
05AC    n:=N_PEL1;                   *LI@ = load low 16 bits of PE
05AD    if not sf then rtn, in,      *LI@,C = load high 8 bits of PE
05AD      cab:=srin;                 *
05AE    cab:=zuy(srin), rtn;         *
05AF
05AF  *SELECT CODE 6, CONTINUED
02FE  stf06: $origin 0x2FE$          *Set TBGflag
02FE    n:=N_TBGT_COUNTER;           *
02FF    goto SET_TBG_INT,            *
02FF      prin:=prin-acc;            *
0300
077D  clf06: $origin 0x77D$          *Clear TBGflag
077D    n:=N_TBGT_COUNTER;           *
077E    ist:=ist and not I_TBGF;     *
077F    prin:=zeros, rtn;            *
0780
00D8  sf@06: $origin 0xD8$           *
00D8    if y15 goto sfs06;           *
00D9  sfc06:                         *Skip if TBGflag is clear
00D9    n:=N_TBGT_COUNTER;           *
00DA    goto sfc@@, nop:=prin;       *
00DB  sfs06:                         *Skip if TBGflag is set
```

```
00DB    n:=N_TBGT_COUNTER;          *
00DC    goto sfs@@, nop:=prin;      *
02DF  li@07: $origin 0x2DF$         *Load @ from MP violation register.
02DF  mi@07:                        *Merge @ from MP violation register.
02DF    if not sf then rtn;         *LI@,H and MI@,H are NOPs
02E0    n:=N_MP_VIOLATION;          *
02E1    cab:=prin ior sl, rtn;      *
02E2
00EB  ot@07: $origin 0x00EB$        *Output @ to MP violation register.  !!
00EB    if not sf goto ot@07h,      *
00EB      sl:=cab;                  *
00EC    n:=N_MP_VIOLATION;          *
00ED    prin:=cab and 0x7fff;       *
00EE  ot@07h:
00EE    rtn;                        *
00EF
00EF  ******************************************************************
00EF  * Interrupt enable routines                                      *
00EF  *   These routines are origin'd throughout the base set to       *
00EF  *   fill in holes due to fragmentation                           *
00EF  ******************************************************************
00EF
0270  CLR_LV2: $origin 0x270$       *CLEAR THE LEVEL 2 INTERRUPT ENABLE
0270    n:=N_ST;                    * (behind SRG code)
0271    prin:=prin and not ST_LV2;  * clear level 2 enable
0272    ist:=ist and not 0x8120;    * 0x0020 disable PFW interrupts
0273    rtn;                        * 0x8000 disable TBGflag interrupts
0274                                * 0x0100 disable IO interrupts
0274
0274  SET_LV2:                      *SET THE LEVEL 2 INTERRUPT ENABLE
0274    n:=N_ST;                    * emulate its functions.
0275    prin:=prin ior ST_LV2;      * set level 2 enable
0276    ist:=ist ior 0x0020;        * enable PFW
0277    nop:=prin inor ST_LV23;     * should I/O and TBG be enabled?
0278    if not yz then rtn;         * no: then return
0279    ist:=ist ior 0x0100;        * yes: enable I/O
027A    goto SET_TBG_INT;           * yes: do the TBG flag thing
027B
027B  CLR_LV3:                      *CLEAR THE LEVEL 3 INTERRUPT ENABLE
027B    n:=N_ST;                    * emulate its functions.
027C    prin:=prin and not ST_LV3;  *
027D    ist:=ist and not 0x8100;    * 0x8000 disable TBGF
027E    rtn;                        * 0x0100 disable I/O
027F
023A  SET_LV3: $origin 0x23A$       *SET THE LEVEL 3 INTERRUPT ENABLE
023A    n:=N_ST;                    * (behind ASG code)
023B    prin:=prin ior ST_LV3;      * set level 3 enable
023C    nop:=prin inor ST_LV23;     * if level
023D    if not yz then rtn;         *   2 and 3 are enabled,
023E    ist:=ist ior 0x0100;        *   then enable I/O interrupts
023F    goto SET_TBG_INT;           *   then set a TBG interrupt
0240
0370  clc_00:    $origin 0x370$     *system reset
0370    sl:=0xfff0 and ist;         *pulse CRS
0371    ist:=sl ior I_CRS;          *
0372    goto clc0flags, ist:=sl;    *parity system on
0373                                *memory protect off
0373                                *clear pending memory protect interrupt
0373                                *enable  level 2
0373                                *disable level 3
0373                                *gr disabled
0373                                *tbg flag cleared, TBG turned off
0373
0373  PULSE_SCHOD:                  *Assert SCHOD for one cycle
0373    sl:=0xfff0 and ist;         *  to acknowledge SCHID.
0374    ist:=sl ior I_SCHOD;        *
0375    ist:=sl; rtn;               *
0377
0377  OF_TBG:                       *TURN OFF THE TIME BASE GENERATOR
0377    sl:=0xfff0 and ist;         *Clear possible tbgtick and
0378    ist:=0x0404 ior sl;         * turn it off in the same line
0379    n:=N_TBGT_COUNTER;          *
037A    ;                           *  and wait for tick to propogate.
037B    sl:=0xfff0 and ist;         *clear tbgtick that may have
037C    ist:=I_TBGT_CLR ior sl;     *  propogated because of turning off TBG
037D    rtn, prin:=zero;            *
```

```
037E
02E2  DI_PE_INT: $origin 0x2e2$     *Disable parity interrupts
02E2    s1:=0xfff0 and ist;         *  (behind EAG divide code)
02E3    ist:=s1 ior I_PE_CLR;       *
02E4    ist:=s1 and not I_PEE;      *
02E5    rtn;                        *
02E6
02E6  SET_TBG_INT:                  *Set the TBGflag.  (Called when one of the
02E6    n:=N_ST;                    * qualifiers is enabled).
02E7    nop:=prin inor ST_LV23T;    * level 2 and 3 and TBG_E must be enabled
02E8    if not yz then rtn;
02E9    n:=N_TBGT_COUNTER;          * Is TBG tick counter greater than zero?
02EA    nop:=prin;                  *
02EB    if yz then rtn;             * n: then return;
02EC    ist:=ist ior I_TBGF;        * y: then set TBGFLAG.
02ED    rtn;                        *
02EE
02EE  BOOT:                         *SET THE PROCESSOR UP TO BOOT
02EE    n:=N_MPAR;                  *  Set up maps for logical to physical
02EF    ct:=1023;                   *    mapping of pages
02F0    srin:=0;                    *    0 to 1023
02F1    set_map:                    *
02F1      if not ctz goto set_map,  *
02F1        map:=srin;              *
02F2  RESET_PU:                     *RESET UPPER PROCESSOR (used by selftest)
02F2    memr:=MEMR_BOOT;            *A700 will enter boot memory
02F3    p:=020002, bbus/memr;       *in Virtual Control Panel
02F4    if not b15 goto no_tdi,     *
02F4      n:=111(ones);             *n:=N_ST;
02F5    cmid;                       *complement the interrupt holdoff
02F6  no_tdi:                       *
02F6  clc0flags:                    *
02F6    call OF_TBG;                *TBG initialization: turn it off.
02F7    call SET_LV2;               *enable level 2 interrupts
02F8    call CLR_LV3;               *disable level 3 interrupts
02F9    ist:=0x1c30;                *reset PE int (PE enabled)
02FA    ist:=0x1c32;                *reset MP int (PFW enabled)
02FB    ist:=0x1c36, bbus/memr;     *reset MTO indicator
02FC    prin:=ST_RESET;             *reset microcode-kept status
02FD    nop:=zero, ldbr, rtn;       *initialize base regiseter
02FE
02AB  WMAP_PACK: $origin 0x2AB$     *STORE PACKED WMAP INTO S0
02AB    nop:=zero, ldq;             *DW might be enabled!
02AC    s0:=memr and 0x001f;        *get EXECUTE field
02AD    s1:=prin and 0x1f00;        *
02AE    s1:=rl1(s1),                *
02AE      if not mpen goto wmapnmp; *
02AF    s0:=s0 ior 0x8000;          *
02B0  wmapnmp:                      *
02B0    s1:=rl1(s1);                *get DATA2 field
02B1    s2:=prin and 0x001f;        *
02B2    s2:=rl1(s2);                *
02B3    s2:=rl4(s2);                *get DATA1 field
02B4    s0:=s0 ior s1;              *
02B5    s0:=s0 ior s2, rtn;         *that's all.....
02B6
02B6  WMAP_UNPACK:                  *load WMAP from s1
02B6    s2:=rr1(s1);                *(DW must be disabled!)
02B7    if not b15 goto wmap_nomp,  *
02B7      s3:=rl1(s2);              *
02B8    ist:=ist and not I_MPD;     *turn on memory protect
02B9  wmap_nomp:                    *
02B9    s3:=0x1f00 and s3;          *s3 = DATA2 field
02BA    s2:=rl4(s2);                *
02BB    s2:=swzu(s2);               *
02BC    s2:=s2 and 0x001f;          *s2 = DATA1 field
02BD    prin:=s2 ior 0x2020;        *set A/B addressibility off for DATA1, DATA2
02BE    memr:=s1 and 0x001f;        *
02BF    prin:=prin ior s3,          *
02BF      rtn;                      *
02C0
02C0
02C1
```

```
MPARA source listing
0000 MPARA; *memory reference group and memory utilities <820204.1442>
0104  $origin 0x104$ *file = &MRG <820204.1442>
0104  ********************************************************************
0104  * (C) Copyright Hewlett Packard Company 1982. All rights reserved.  *
0104  * No part of this program may be photocopied, reproduced or        *
0104  * translated to another program language without the prior written  *
0104  * consent of Hewlett Packard Company.                               *
0104  ********************************************************************
0104
0104  $define adrl/INST_RESTART 0x00D0$
0104
0104  AND_D: $origin 0x104$          *AND,D (read of data was begun by JTAB)
0104    a:=a and t,                  * logical and A with memory data.
0104      fchp, rtn;                 * fetch...
0105
0106  JSB_D: $origin 0x106$          *JSB,D (JTAB latches MRG address in MA)
0106    acc  :=p, cmid;              * store p value into return register
0107    wrb:=acc, bbus/ma,           * write p value to MRG address
0107      goto jsbx1;                *
0108
0108  XOR_D: $origin 0x108$          *XOR,D (read of data was begun by JTAB)
0108    a:=a xor t,                  * logical xor A with data
0108      fchp, rtn;                 * fetch...
0109
0109  jsbx1: p:=ma+one, goto jsbx2; * set p to MRG address plus one
010A
010A  JMP_D: $origin 0x10a$          *JMP,D (fetch was begun by JTAB)
010A    p:=ma,                       * Stuff MRG address into p
010A     rtn;                        *
010B
010C  IOR_D: $origin 0x10c$          *IOR,D (read of data was begun by JTAB)
010C    a:=a ior t,                  * logical or A with data
010C      fchp, rtn;                 * fetch...
010D
010E  ISZ_D: $origin 0x10e$          *ISZ,D (read of data was begun by JTAB)
010E    p:=ma;                       * prepare for write
010F    wrp:=t-acc, goto isz_dx;     * write incremented value
0110
0110  ADA_D: $origin 0x110$          *ADA,D (read of data was begun by JTAB)
0110    a:=a+t, enoe,                * add memory to A, set E or O
0110      fchp, rtn;                 * fetch...
0111
0112  ADB_D: $origin 0x112$          *ADB,D (read of data was begun by JTAB)
0112    b:=b+t, enoe,                * add memory to B, set E or O
0112      fchp, rtn;                 * fetch...
0113
0114  CPA_D: $origin 0x114$          *CPA,D (read of data was begun by JTAB)
0114    acc:=cab;                    *
0115    nop:=acc xor t,              * compare A and data
0115      goto cp@x;                 *
0116
0116  CPB_D: $origin 0x116$          *CPB,D (read of data was begun by JTAB)
0116    acc:=cab;                    *
0117    nop:=acc xor t,              * compare B and data
0117      goto cp@x;                 *
0118
0118  LDA_D: $origin 0x118$          *LDA,D (read of data was begun by JTAB)
0118    a:=t,                        * Load A.
0118      fchp, rtn;                 * fetch...
0119
011A  LDB_D: $origin 0x11a$          *LDB,D (read of data was begun by JTAB)
011A    b:=t,                        * Load B.
011A      fchp, rtn;                 * fetch...
011B
011B  jsbx2:                         *jsb extension 2 - just fetch and return
011C  STA_D: $origin 0x11c$          *STA,D (store of data was begun by JTAB)
011C    fchp, rtn;                   * fetch...
011D
011E  STB_D: $origin 0x11e$          *STB,D (store of data was begun by JTAB)
011E    fchp, rtn;                   * fetch...
011F
0124  AND_I: $origin 0x124$          *AND,I (read of indirect was begun by JTAB)
0124    call INDREAD;                * indirect
```

```
0125   a:=a and t,              * logical and a and data
0125     fchp, rtn;             * fetch...
0126
0126  JSB_I: $origin 0x126$     *JSB,I (read of indirect was begun by JTAB)
0126   goto INDJSB,             * write p into return register,
0126     acc  :=p;              *   write into return register
0127
0128  XOR_I: $origin 0x128$     *XOR,I (read of indirect was begun by JTAB)
0128   call INDREAD;            * indirect
0129   a:=a xor t,              * logical xor a and data
0129     fchp, rtn;             * fetch...
012A
012A  JMP_I: $origin 0x12a$     *JMP,I (read of indirect was begun by JTAB)
012A   cmid;                    * set TDI
012B   goto jmpind, nop:=t;     * indirect
012C
012C  IOR_I: $origin 0x12c$     *IOR,I (read of indirect was begun by JTAB)
012C   call INDREAD;            * indirect
012D   a:=a ior t,              * logical ior A and data
012D     fchp, rtn;             * fetch...
012E
012E  ISZ_I: $origin 0x12e$     *ISZ,I (read of indirect was begun by JTAB)
012E   call isz_ix;             * indirect
012F   wrp:=t-acc, goto isz_dx; * write incremented value back
0130
0130  ADA_I: $origin 0x130$     *ADA,I (read of indirect was begun by JTAB)
0130   call INDREAD;            * indirect
0131   a:=a+t, enoe,            * add data to A
0131     fchp, rtn;             * fetch...
0132
0132  ADB_I: $origin 0x132$     *ADB,I (read of indirect was begun by JTAB)
0132   call INDREAD;            * indirect
0133   b:=b+t, enoe,            * add data to B
0133     fchp, rtn;             * fetch...
0134
0134  CPA_I:  $origin 0x134$    *CPA,I (read of indirect was begun by JTAB)
0134   call INDREAD, acc:=cab;  * indirect
0135   nop:=acc xor t,          * compare A to data
0135     goto cp@x;             *
0136
0136  CPB_I:  $origin 0x136$    *CPB,I (read of indirect was begun by JTAB)
0136   call INDREAD, acc:=cab;  * indirect
0137   nop:=acc xor t,          * compare B to data
0137     goto cp@x;             *
0138
0138  LDA_I: $origin 0x138$     *LDA,I (read of indirect was begun by JTAB)
0138   call INDREAD;            * indirect
0139   a:=t,                    * load A
0139     fchp, rtn;             * fetch...
013A
013A  LDB_I: $origin 0x13a$     *LDB,I (read of indirect was begun by JTAB)
013A   call INDREAD;            * indirect
013B   b:=t,                    * load B
013B     fchp, rtn;             * fetch...
013C
013C  STA_I: $origin 0x13c$     *STA,I (read of indirect was begun by JTAB)
013C   call INDSTORE, acc:=a;   * indirect store
013D   fchp, rtn;               * fetch...
013E
013E  STB_I: $origin 0x13e$     *STB,I (read of indirect was begun by JTAB)
013E   call INDSTORE, acc:=b;   * indirect store
013F   fchp, rtn;               * fetch...
0140
0700  INDREAD: $origin 0x700$   *indirect read utility
0700   nop:=t, rdb;             *1st level: save t, start new read
0701   if not b15 then rtn,     *  if t was direct then return
0701     s6:=ma+one;            *  save address+1 in
0702   nop:=t, rdb;             *2nd level:
0703   if not b15 then rtn,     *
0703     s6:=ma+one;            *
0704  indrd2:                   *
0704   nop:=t, rdb;             *3rd level:
0705   if not b15 then rtn,     *
0705     s6:=ma+one;            *
0706  call tdi_disable,         *assure that interrupts are enabled
```

```
0706    nop:=memr;
0707  goto INDREAD;
0708
0708  tdi_disable:                *disable tdi and check interrupts
0708    if not b15 goto check_ints; *if tdi is set
0709    cmid;;                      *  then disable (and wait for intp)
070B  check_ints:                   *
070B    if not intp then rtn;       *if no interrupts then return
070C    goto INST_RESTART,          *restart instruction
070C      p:=fa;                    *reset program counter
070D
070D  INDSTORE:                     *indirect store utility
070D    cwrb:=acc, bbus/t;          *1st level:  if direct then store
070E    s6:=ma+one,                 *  and return, else read new memory
070E      if not b15 then rtn;      *  address.
070F    cwrb:=acc, bbus/t;          *2nd level:
0710    s6:=ma+one,                 *
0710      if not b15 then rtn;      *
0711    cwrb:=acc, bbus/t;          *3rd level:
0712    s6:=ma+one,                 *
0712      if not b15 then rtn;      *
0713  call tdi_disable,             *
0713    nop:=memr;                  *
0714  goto INDSTORE;                *
0715
0715  INDRSOLV:                     *indirect resolution utility
0715    s7:=t+s0;                   *1st level: save t
0716    if not b15 then rtn;        *  if direct then rtn,
0717    rdb, bbus/t;                *  initiate read
0718    s7:=t+s0;                   *2nd level: save t
0719    if b15 then rdb, bbus/t;    *  if indirect initiate read
071A    if not b15 then rtn;        *  if direct then return
071B    s7:=t+s0;                   *3rd level: save t
071C    if b15 then rdb, bbus/t;    *  if indirect initiate read
071D    if not b15 then rtn;        *  if direct then return
071E  call tdi_disable,             *
071E    nop:=memr;                  *
071F  goto INDRSOLV;                *
0720
0720  INDJMP:                       *indirect branching utility
0720    nop:=t;                     *level 1: freeze until data returned.
0721    if not b15 goto fetch,      *  if direct then fetch
0721      p:=t;                     *
0722    rdb, bbus/t,                *level 2:
0722      call indjshowt;           *
0723  jmpind:                       *
0723    if not b15 goto fetch,      *  if direct then fetch and return
0723      p:=t;                     *
0724    rdb, bbus/t,                *level 3:
0724      call indjshowt;           *  freeze until data returned
0725    if not b15 goto fetch,      *  if direct then fetch and return
0725      p:=t;                     *
0726  call tdi_disable,             *assure that interrupts are disabled.
0726    nop:=memr;                  *
0727  goto INDJMP;                  *
0728
0728  indjshowt:                    *
0728    nop:=t, rtn;                *
0729
0729  isz_ix:                       *continue ISZ,I
0729    nop:=t, rdb;                *freeze until address is returned
072A    if not b15 then rtn,        *if direct then return
072A      p:=ma;                    *  load P
072B    call INDREAD;               *do indirect
072C    p:=ma, rtn;                 *load P
072D
072D  INDJSB: $origin 0x72D$        *Indirect jsb routine...
072D    cwrb:=acc  , bbus/t, cmid;  * 1st level: if direct then write else read.
072E    p:=ma+one,                  *   p gets address to jump to
072E      if not b15 goto fetch;    *   if direct then done.
072F    cwrb:=acc  , bbus/t;        * 2nd level:
0730    p:=ma+one,                  *
0730      if not b15 goto fetch;    *
0731    cwrb:=acc  , bbus/t;        * 3rd level:
0732    p:=ma+one,                  *
```

```
0732      if not b15 goto fetch;     *
0733    call tdi_disable,            *
0733      nop:=memr;                 *
0734    goto INDJSB;                 *
0735
0735  INDRDBL:                       *DOUBLE READ UTILITY (intrpt after 2)
0735    nop:=t, rdb;                 *get past DEF
0736    if not b15 goto dblrdone,    *was it direct?
0736      p:=ma+one;                 *y: prepare for second read
0737    nop:=t, rdb;                 *n: start indirect read
0738    if not b15 goto dblrdone,    *was it direct?
0738      p:=ma+one;                 *y: prepare for second read
0739    if not intp goto indrdbl;    *check interrupts now
073A    goto INST_RESTART, p:=fa;    *
073B  dblrdone:                      *
073B    s7:=t, rdp, rtn;             *save high word, start second word read
073C
073C  isz_dx:                        *isz extension:
073C    if not yz goto fetch,        * optimize for non skip case
073C      p:=fa-acc;                 * p:=next opcode
073D    p:=p-acc, goto fetch;        * fetch...
073E
073E  cp@x:                          *cp@ extension:
073E    if not yz then ip;           * if not equal then skip
073F  fetch:                         *
073F    fchp, rtn;                   * fetch...
0740
0740
0741
```

```
MPARA source listing
0000 MPARA; *alter skip group macroinstructions <820204.1550>
01E0  $origin 0x1E0$  *file = &ASG <820204.1550>
01E0  ************************************************************************
01E0  * (C) Copyright Hewlett Packard Company 1982. All rights reserved.    *
01E0  * No part of this program may be photocopied, reproduced or          *
01E0  * translated to another program language without the prior written   *
01E0  * consent of Hewlett Packard Company.                                 *
01E0  ************************************************************************
01E0
01E0  ASG_ALL: $origin 0x1e0$    *ASG instructions (JTAB entry) 3 - 9 cycles
01E0
01E0  $origin 0x1e0$             *JTAB entry for non-simple ASG instructions.
01E0    n:=rll(cab);             * set up n register for asg transformation.
01E1    ct:=asg(ct),             * perform and execute complex ASG instruction.
01E1      goto asg_ext;          * (7-9)
01E2  $origin 0x1e2$             *JTAB entry for RSS  (4)
01E2  asg_skip:                  *
01E2    ip;                      *
01E3  asg_noskip:                *
01E3    fchp, rtn;               *
01E4  $origin 0x1e4$             *JTAB entry for SZ@  (4)
01E4  sz@:                       *
01E4    if yz then ip;           *
01E5    fchp, rtn;               *
01E6  $origin 0x1e6$             *JTAB entry for SZ@,RSS (4)
01E6  sz@rss:                    *
01E6    if not yz then ip;       *
01E7    fchp, rtn;               *
01E8  $origin 0x1e8$             *JTAB entry for SS@  (4)
01E8    if not y15 then ip;      *
01E9    fchp, rtn;               *
01EA  $origin 0x1ea$             *JTAB entry for SS@,RSS (4)
01EA    if y15 then ip;          *
01EB    fchp, rtn;               *
01EC  $origin 0x1ec$             *JTAB entry for SS@,SZ@  (5-6)
01EC    if not y15 goto asg_skip;*
01ED    goto sz@;                *
01EE  $origin 0x1ee$             *JTAB entry for SS@,SZ@,RSS  (5)
01EE    if y15 goto asg_skip;    *
01EF    goto sz@rss;             *
01F0  in@noste:                  *
01F0    if not alov then rtn,    *
```

```
01F0     fchb, bbus/p;          *
01F1   sto, rtn;                *
01F2  $origin 0x1f2$            *JTAB entry for CL@  (3)
01F2    cab:=zero, fchp, rtn;   *  A or B := zero
01F4  $origin 0x1f4$            *JTAB entry for CM@
01F4    cab:=not cab, fchp, rtn; *  A or B := not A or B  (3)
01F6  $origin 0x1f6$            *JTAB entry for CC@
01F6    cab:=ones, fchp, rtn;   *  A or B := ones
01F8  $origin 0x1f8$            *JTAB entry for IN@      (4)
01F8    acc:= -acc, fchp;       *  A or B := A or B plus 1
01F9    cab:=cab+acc, enoe, rtn; *  (can set E or O)
01FA  $origin 0x1fa$            *JTAB entry for CL@,IN@   (4)
01FA    cab:= -acc, fchp, rtn;  *  A or B := 1
01FC  $origin 0x1fc$            *JTAB entry for CM@,IN@   (4; 5 if setting O)
01FC    if not yz goto in@noste, *
01FC      cab:= -cab;           *
01FD    ste, fchp, rtn;         *
01FE  $origin 0x1fe$            *JTAB entry for CC@,IN@   (3)
01FE    cab:=zero, ste,         *  A or B := 0
01FE      fchp, rtn;            *  (E is set on rollover)
01FF
01FF  asg_ext:  $origin 0x1FF$  *ASG extension  (7-9)
01FF    goto asg_tbl1, ct30,    * fall into asg_tbl1,
01FF      acc:= -acc;           * put +1 in accumulator
0200
0200  asg_tbl1:
0200  {
0200    Execute the following asg instructions within the table:
0200      CL@,CM@,CC@,NOP   - through bbus,abus and alu fields
0200      CLE,CME,CCE,NOP   - through ste or cle in store field
0200      SEZ,SS@,SL@       - increment p if skip equation is true.
0200    Branch to asg_tbl2 if no skip occurred,
0200      otherwise to tbl3.
0200  }
0200  {00} nop:=cab,          ct74, cle, goto asg_tbl2;  *
0201  {01} cab:=zero,         ct74, cle, goto asg_tbl2;  *
0202  {02} nop:=cab,          ip,   cle, goto asg_iskp;  *
0203  {03} cab:=zero,         ip,   cle, goto asg_iskp;  *
0204  {04} nop:=cab,          ct74, ste, goto asg_tbl2;  *
0205  {05} cab:=zero,         ct74, ste, goto asg_tbl2;  *
0206  {06} nop:=cab,          ip,   ste, goto asg_iskp;  *
0207  {07} cab:=zero,         ip,   ste, goto asg_iskp;  *
0208  {10} cab:=not cab,      ct74, cle, goto asg_tbl2;  *
0209  {11} cab:=ones,         ct74, cle, goto asg_tbl2;  *
020A  {12} cab:=not cab,      ip,   cle, goto asg_iskp;  *
020B  {13} cab:=ones,         ip,   cle, goto asg_iskp;  *
020C  {14} cab:=not cab,      ct74, ste, goto asg_tbl2;  *
020D  {15} cab:=ones,         ct74, ste, goto asg_tbl2;  *
020E  {16} cab:=not cab,      ip,   ste, goto asg_iskp;  *
020F  {17} cab:=ones,         ip,   ste, goto asg_iskp;  *
0210
0210  asg_tbl2:
0210  {
0210    If IN@ enabled then do it.
0210    If SZ@ enabled then branch to do SZ@ (with or with RSS).
0210    YZ condition is set to cab for sz@ test.
0210  }
0210  {00}                fchp, rtn;                    *
0211  {01}                fchp, rtn;                    *        RSS
0212  {02} nop:=cab,                goto asg_sz@;       *    SZ@
0213  {03} nop:=cab,                goto asg_rsz@;      *    SZ@,RSS
0214  {04} cab:=cab+acc, enoe, fchp, rtn;               *IN@
0215  {05} cab:=cab+acc, enoe, fchp, rtn;               *IN@,    RSS
0216  {06} nop:=cab+acc,            goto asg_in@sz@;    *IN@,SZ@
0217  {07} nop:=cab+acc,            goto asg_in@rsz@;   *IN@,SZ@,RSS
0218  {10}                fchp, rtn;                    *
0219  {11}                fchp, rtn;                    *        RSS
021A  {12} nop:=cab,                goto asg_sz@;       *    SZ@
021B  {13} nop:=cab,                goto asg_rsz@;      *    SZ@,RSS
021C  {14} cab:=cab+acc, enoe, fchp, rtn;               *IN@,
021D  {15} cab:=cab+acc, enoe, fchp, rtn;               *IN@,    RSS
021E  {16} nop:=cab+acc,            goto asg_in@sz@;    *IN@,SZ@
021F  {17} nop:=cab+acc,            goto asg_in@rsz@;   *IN@,SZ@,RSS
0220
0220  asg_tbl3:
```

```
0220 {
0220   If IN@ enabled then do it.
0220   fchp and return.
0220 }
0220 {00} fchp, rtn;
0221 {01} fchp, rtn;
0222 {02} fchp, rtn;
0223 {03} fchp, rtn;
0224 {04} fchp, rtn, cab:=cab+acc, enoe;
0225 {05} fchp, rtn, cab:=cab+acc, enoe;
0226 {06} fchp, rtn, cab:=cab+acc, enoe;
0227 {07} fchp, rtn, cab:=cab+acc, enoe;
0228 {10} fchp, rtn;
0229 {11} fchp, rtn;
022A {12} fchp, rtn;
022B {13} fchp, rtn;
022C {14} fchp, rtn, cab:=cab+acc, enoe;
022D {15} fchp, rtn, cab:=cab+acc, enoe;
022E {16} fchp, rtn, cab:=cab+acc, enoe;
022F {17} fchp, rtn, cab:=cab+acc, enoe;
0230
0230 asg_iskp:
0230   goto asg_tbl3, ct74;
0231
0231 asg_in@sz@:
0231   if yz then ip;
0232 asg_in@:
0232   cab:=cab+acc, enoe, fchp, rtn;
0233
0233 asg_sz@:
0233   if yz then ip;
0234   fchp, rtn;
0235
0235 asg_in@rsz@:
0235   if not yz then ip;
0236   cab:=cab+acc, enoe, fchp, rtn;
0237
0237 asg_rsz@:
0237   if not yz then ip;
0238   fchp, rtn;
0239
0239
023A
```

```
MPARA source listing
0000 MPARA; *shift/rotate group macroinstructions <820204.1550>
01C0 $origin 0x1c0$ *file = &SRG <820204.1550>
01C0 *********************************************************************
01C0 * (C) Copyright Hewlett Packard Company 1982. All rights reserved.  *
01C0 * No part of this program may be photocopied, reproduced or         *
01C0 * translated to another program language without the prior written *
01C0 * consent of Hewlett Packard Company.                               *
01C0 *********************************************************************
01C0
01C0 SRG_ALL: $origin 0x1c0$     *JTAB entry for SRG instructions (3-11cycles)
01C0
01C0 $origin 0x1c0$              *JTAB entry for srg1,NOP,NOP,srg2
01C0   call srg1, ct:=srg(ct);   *
01C1   goto srg2;                *
01C2 $origin 0x1c2$              *JTAB entry for srg1,NOP,NOP,NOP
01C2   call srg1, ct:=srg(ct);   *
01C3   fchp, rtn;                *
01C4 $origin 0x1c4$              *JTAB entry for srg1,NOP,SL@,srg2
01C4   call srg1, ct:=srg(ct);   *
01C5   goto sl@srg2;             *
01C6 $origin 0x1c6$              *JTAB entry for srg1,NOP,SL@,NOP
01C6   call srg1, ct:=srg(ct);   *
01C7   goto sl@;                 *
01C8 $origin 0x1c8$              *JTAB entry for srg1,CLE,NOP,srg2
01C8   call srg1, ct:=srg(ct);   *
01C9   goto clesrg2;             *
01CA $origin 0x1ca$              *JTAB entry for srg1,CLE,NOP,NOP
01CA   call srg1, ct:=srg(ct);   *
```

```
01CB     cle, fchp, rtn;           *
01CC   $origin 0x1cc$              *JTAB entry for srg1,CLE,SL@,srg2
01CC     call srg1, ct:=srg(ct);   *
01CD     goto clesl@srg2;          *
01CE   $origin 0x1ce$              *JTAB entry for srg1,CLE,SL@,NOP
01CE     call srg1, ct:=srg(ct);   *
01CF     goto clesl@;              *
01D0   $origin 0x1d0$              *JTAB entry for NOP ,NOP,NOP,srg2
01D0     ct:=srg(ct);              *
01D1     goto srg2;                *
01D2   $origin 0x1d2$              *JTAB entry for NOP ,NOP,NOP,NOP
01D2     fchp, rtn;                * (very NOP)
01D4   $origin 0x1d4$              *JTAB entry for NOP ,NOP,SL@,srg2
01D4     ct:=srg(ct);              *
01D5     goto sl@srg2;             *
01D6   $origin 0x1d6$              *JTAB entry for NOP ,NOP,SL@,NOP
01D6     goto sl@;                 *
01D8   $origin 0x1d8$              *JTAB entry for NOP ,CLE,NOP,srg2
01D8     ct:=srg(ct);              *
01D9     goto clesrg2;             *
01DA   $origin 0x1da$              *JTAB entry for NOP ,CLE,NOP,NOP
01DA     cle, fchp, rtn;           * (CLE only)
01DB   srg1:                       *
01DB     gototbl srg_tbl1;         *
01DC   $origin 0x1dc$              *JTAB entry for NOP ,CLE,SL@,srg2
01DC     ct:=srg(ct);              *
01DD     goto clesl@srg2;          *
01DE   $origin 0x1de$              *JTAB entry for NOP ,CLE,SL@,NOP
01DE     goto clesl@;              *
01DF
0240   srg_tbl1: $origin 0x240$    *srg instructions, continued
0240   {
0240     Perform the 1st instruction type of the SRG instruction and return
0240   }
0240   {00}                             rtn;             *DISABLED *LS
0241   {04}                             rtn;             *DISABLED *LR
0242   {10} cab:=all(cab),              rtn;             *ENABLED  *LS
0243   {14} acc:=arl(ones),        lwf, goto srg_@1r1;   *ENABLED  *LR
0244   {01}                             rtn;             *DISABLED *RS
0245   {05} nop:=rrl(cab),         lwe, rtn;             *DISABLED ER*
0246   {11} cab:=arl(cab),              rtn;             *ENABLED  *RS
0247   {15} cab:=rrl(cab),         lwe, rtn;             *ENABLED  ER*
0248   {02}                             rtn;             *DISABLED R*L
0249   {06} nop:=rll(cab),         lwe, rtn;             *DISABLED EL*
024A   {12} cab:=rll(cab),              rtn;             *ENABLED  R*L
024B   {16} cab:=rll(cab),         lwe, rtn;             *ENABLED  EL*
024C   {03}                             rtn;             *DISABLED R*R
024D   {07}                             rtn;             *DISABLED *LF
024E   {13} cab:=rrl(cab),              rtn;             *ENABLED  R*R
024F   {17} cab:=rl4(cab),              rtn;             *ENABLED  *LF
0250
0250   srg_tbl2:
0250   {
0250     Perform the 2nd instruction type of the SRG instruction,
0250       fetch and return.
0250   }
0250   {00}                          fchp, rtn;             *DISABLED *LS
0251   {01}                          fchp, rtn;             *DISABLED *RS
0252   {02}                          fchp, rtn;             *DISABLED R*L
0253   {03}                          fchp, rtn;             *DISABLED R*R
0254   {04}                          fchp, rtn;             *DISABLED *LR
0255   {05} nop:=rrl(cab),           lwe,  goto srg_tbl2;   *DISABLED ER*
0256   {06} nop:=rll(cab),           lwe,  goto srg_tbl2;   *DISABLED EL*
0257   {07}                          fchp, rtn;             *DISABLED *LF
0258   {10} cab:=all(cab),           fchp, rtn;             *ENABLED  *LS
0259   {11} cab:=arl(cab),           fchp, rtn;             *ENABLED  *RS
025A   {12} cab:=rll(cab),           fchp, rtn;             *ENABLED  R*L
025B   {13} cab:=rrl(cab),           fchp, rtn;             *ENABLED  R*R
025C   {14} acc:=arl(ones),          lwf,  goto srg_@1r2;   *ENABLED  *LR
025D   {15} cab:=rrl(cab),           lwe,  goto srg_tbl2;   *ENABLED  ER*
025E   {16} cab:=rll(cab),           lwe,  goto srg_tbl2;   *ENABLED  EL*
025F   {17} cab:=rl4(cab),           fchp, rtn;             *ENABLED  *LF
0260
0260   srg_@1r1:                     *perform first *LR and return
0260     cab:=lll(acc and cab),      *
```

```
0260      clf, rtn;                 *(clear flag for possible second *LR)
0261  srg_@lr2:                     *perform 2nd *LR , fetch and return
0261    cab:=lll(acc and cab),      *
0261      fchp, rtn;                *
0262  srg2:                         *perform the second SRG instruction
0262    goto srg_tbl2, ct74;        *
0263  sl@:                          *perform the SL@, fetch and return
0263    nop:=lrl(cab);              *
0264    if not sf then ip;          *
0265    fchp, rtn;                  *
0266  clesl@:                       *perform the CLE and SL@, fetch and return
0266    nop:=lrl(cab);              *
0267    if not sf then ip;          *
0268    fchp, cle, rtn;             *
0269  clesl@srg2:                   *perform the CLE, SL@ and second SRG
0269    nop:=lrl(cab);              *
026A    if not sf then ip;          *
026B  clesrg2:                      *perform the CLE and second SRG
026B    goto srg_tbl2, ct74, cle;   *
026C  sl@srg2:                      *perform the SL@ and second SRG
026C    nop:=lrl(cab);              *
026D    if not sf then ip;          *
026E    goto srg_tbl2, ct74;        *
026F
026F
0270
```

```
MPARA source listing
0000 MPARA; *extended arithmetic group macroinstructions <820204.1550>
0280  $origin 0x280$ *file = &EAG <820204.1550>
0280  *********************************************************************
0280  * (C) Copyright Hewlett Packard Company 1982. All rights reserved.  *
0280  * No part of this program may be photocopied, reproduced or         *
0280  * translated to another program language without the prior written  *
0280  * consent of Hewlett Packard Company.                               *
0280  *********************************************************************
0280
0141  MPY: $origin 0x141$           *JTAB entry for MPY
0141    goto eag_mpy, s0:=zero;     *
0142  DIV: $origin 0x142$           *JTAB entry for DIV
0142    goto eag_div, ct :=acc+acc; *
0143  JLA: $origin 0x143$           *JTAB entry for JLA
0143    goto eag_jl@, rdp;          *
0145  DLD: $origin 0x145$           *JTAB entry for DLD
0145    goto eag_dld, rdp;          *
0146  DST: $origin 0x146$           *JTAB entry for DST
0146    goto eag_dst, rdp;          *
0147  JLB: $origin 0x147$           *JTAB entry for JLB
0147    goto eag_jl@, rdp;          *
0101  AS@: $origin 0x101$           *JTAB entry for ASL and ASR
0101    goto eag_as@;               *
0102  LS@: $origin 0x102$           *JTAB entry for LSL and LSR
0102    goto eag_ls@, s0:=ct;       *
0103  RR@: $origin 0x103$           *JTAB entry for RRL and RRR
0103    goto eag_rr@, s1:=a;        *
0104
0140  eag_jl@: $origin 0x140$       *JL@ extension
0140    goto INDJMP, cab:=p-acc;    * load A or B with return address
0141
0148  eag_dld: $origin 0x148$       *DLD continued
0148    call INDREAD,               *read first word
0148      p:=p-acc;                 *
0149    a:=t;                       *A := first word
014A    bbus/s6, rdb;               *read second word
014B    b:=t, fchp, rtn;            *B := second word
014C
014C  eag_dst:                      *DST continued
014C    call INDSTORE, acc:=a;      *store first word (A)
014D    wrb:=b, bbus/s6, ip;        *store second word (B)
014E    fchp, rtn;                  *fchp, rtn
014F
0280  eag_mpy: $origin 0x280$       *MPY continued
0280    rdp,                        *read DEF
```

```
0280      ct:=acc+acc;              *low 4 bits of ct := 14
0281    call INDREAD,               *indirect
0281      acc:=a;                   *
0282    nop:=t, ldq,                *Q holds the multiplier
0282      dct;                      *low 4 bits of ct := 13
0283    b:=tmpy(s0), ip;            *first multiply step
0284    mloop:                      *Repeat 14 times:
0284      if not ctz4 goto mloop,   *  B,Q := Two's complement multiply of
0284        b:=tmpy(b);             *  of ACC and Q.  (see 2903 specials)
0285    b:=tmlc(b);                 *Two's comp mult last step.
0286    a:=q,                       *Two's complement multiply never
0286      clo,                      * overflows.
0286      fchp, rtn;                *fetch...
0287
0287  eag_as@:                      *Arithmetic shift left or right
0287    nop:=0x0200 and ct;         *  decode between left and right
0288    dct;                        *synchronize count
0289    if not yz goto eag_asr,     *
0289      nop:=a, ldq;              *prepare for double word shift or normalize
028A
028A  eag_asl:                      *Arithmetic shift left
028A    aslloop:                    *  Loop until ct is zero:
028A      if ctz4 goto asldone,     *    double normalize (B,A)
028A        b:=dnrm(b);             *    and check
028B      if not cf goto aslloop,   *    for arith overflow.
028B        acc:=b;                 *    remember b sign bit.
028C    aslovlp:                    *  Overflow occurred in loop:
028C      if not ctz4 goto aslovlp, *    finish
028C        b:=dnrm(b);             *      normalizing
028D    aslovfl:                    *    set sign bit
028D      b:=b and 0x7fff;          *      correctly to opposite sign
028E      acc:=0x8000, bbus/acc;    *      of acc, which held the overnormalized
028F      if not b15 then           *      value of b.
028F        b:=b ior acc;           *
0290      fchp, a:=q, sto, rtn;     *      fetch, set o and rtn;
0291    asldone:                    *   Overflow did not occur in loop:
0291      if not cf goto aslok,     *     though it
0291        acc:=b;                 *       may have on exit.
0292      b:=b xor 0x8000;          *       correct sign bit.
0293      fchp, a:=q, sto, rtn;     *       end instructioneturn.
0294    aslok:                      *
0294      fchp, clo,                *     but it didn't.
0294        a:=q, rtn;              *
0295
0295  eag_asr:                      *Arithmetic shift right instruction
0295    cmdw, clo;                  *Set double word bit.
0296    asrloop:                    *Loop until CTZ4:
0296      if not ctz4 goto asrloop, *  decrement count,
0296        b:=arl(b);              *  arithmetic shift B and Q
0297    fchp,                       *Fetch,
0297      a:=q,                     *  replace A with Q
0297      rtn;                      *  and end instruction
0298
0298  eag_ls@:                      *Logical left or right shift instruction
0298    cmdw, ct:=ct+acc;           * set DW, synch count
0299    nop:=0x0200 and s0;         * decode between LSL and LSR
029A    if not yz goto eag_lsr,     *
029A      nop:=a, ldq;              * prepare for double word shift
029B
029B  eag_lsl:                      *LSL instruction
029B    lslloop:                    *Loop until CTZ4:
029B      if not ctz4 goto lslloop, *  decrement count,
029B        b:=lll(b);              *  logical left shift B and Q
029C    fchp,                       *Fetch,
029C      a:=q,                     *  replace A with Q
029C      rtn;                      *  and end instruction
029D
029D  eag_lsr:                      *LSR instruction
029D    lsrloop:                    *Loop until CTZ4:
029D      if not ctz4 goto lsrloop, *  decrement count,
029D        b:=lrl(b);              *  logical right shift B and Q
029E    fchp,                       *Fetch,
029E      a:=q,                     *  replace A with Q
029E      rtn;                      *  and end instruction
029F
```

```
029F  eag_rr@:                     *RRL and RRR instructions
029F    if ctz4 goto rr@_swap,     * if count is 16 (represented by 0) then
029F      s0:=ct;                  *   swap!
02A0    cmdw;                      * set double word bit
02A1    nop:=0x0200 and s0;        * decocde between RRL and RRR
02A2    if not yz goto eag_rrr,    *
02A2      nop:=a, ldq;             * prepare for double rotate
02A3
02A3  eag_rrl:
02A3    rrlloop:                   *Loop until CTZ4:
02A3      if not ctz4 goto rrlloop, * decrement count,
02A3        b:=rl1(b);             * rotate left B and Q
02A4    fchp,                      *Fetch,
02A4      a:=q,                    * replace A with Q
02A4      rtn;                     * and end instruction
02A5
02A5  eag_rrr:                     *Rotate right instruction
02A5    rrrloop:                   *Loop until CTZ4:
02A5      if not ctz4 goto rrrloop, * decrement count,
02A5        b:=rr1(b);             * rotate right B and Q
02A6    fchp,                      *Fetch,
02A6      a:=q,                    * replace A with Q
02A6      rtn;                     * and end instruction
02A7
02A7  rr@_swap:                    *special case for RRL and RRR: swap
02A7    a:=b, fchp;                *
02A8    b:=s1, rtn;                *
02A9
02C0  eag_div:  $align 64$
02C0  {
02C0    integer divide continuation
02C0    at entry:
02C0      acc    = divisor
02C0      a      = dividend lower
02C0      b      = dividend upper
02C0    during execution:
02C0      a      = partial remainder (lower word)
02C0      b      = partial remainder (upper word)
02C0      acc    = holds divisor
02C0      q      = holds partial remainder (lower word) and quotient as
02C0               developed
02C0    at exit:
02C0      acc    = divisor
02C0      a      = quotient
02C0      b      = remainder
02C0      o is set if overflow occurred
02C0    q,acc,ct are used
02C0  }
02C0    rdp, nop:=a, ldq;          *
02C1    call INDREAD,              *
02C1      ct:=ct+acc;              *low 4 bits of ct := 13
02C2    s2:=b xor t;               *s2:=expected sign
02C3    s1:=b xor b, clo, ip;      *s1:=zero, assume no overflow
02C4    if not b15 goto pr_pos,    *make the partial remainder positive
02C4      acc:=t;                  *
02C5    a:= s1-a, stf, ldq;        *negate (b,a), f shows
02C6    b:= not b + cf;            *  that two quadrant fix occurred
02C7    if alov goto div_mneg,     *if most negative number then quit
02C7      acc:=t;                  *
02C8  pr_pos:                      *
02C8    if yz goto div_zero,       *if divisor is zero then quit
02C8      s0:=div1(b);             *  generate quotient sign bit
02C9    if sf goto dvr_neg,        *if quotient sign bit is 1, then
02C9      s0:=div(s0);             *  divisor was negative
02CA
02CA  dvr_pos:
02CA  {
02CA    divisor is positive.
02CA  }
02CA    nop:=b-acc;                *compare |dnd| - |dvr| (must be <0)
02CB    if not y15 goto div_ovfl,  *if quotient to be > 2**16 then
02CB      s0:=div(s0);             *  integer overflow will occur
02CC  qp_rp_loop:                  *Repeat
02CC    if not ctz4 goto           *  division step
02CC      qp_rp_loop,              *  to form
```

```
02CC     s0:=div(s0);            *  full quotient
02CD    if sf goto qp_rp_check,  *Fix remainder after shift of
02CD      b:=lrl(s0);            *  last divide step.
02CE    b:=b ior 0x8000;         *
02CF  qp_rp_check:               *Check for remainder correction.
02CF    if not y15 goto div_done,  *
02CF      a:=q;                  *
02D0  qp_rp_rc:                  *Remainder correction
02D0    b:=b+acc,                *
02D0      goto div_done;         *
02D1
02D1  dvr_neg:
02D1  {
02D1    Divisor is negative.
02D1  }
02D1    nop:=b+acc;              *compare |dnd| - |dvr| (must be <0)
02D2    if not y15 goto div_ovfl,  *If quotient to be > 2**16
02D2      s0:=div(s0);           *  then integer overflow will occur.
02D3  qn_rp_loop:                *Repeat
02D3    if not ctz4 goto         *  division step
02D3      qn_rp_loop,            *  to form
02D3      s0:=div(s0);           *  full quotient.
02D4    if not sf goto qn_rp_sgn,  *Fix remainder after it was shifted
02D4      b:=lrl(s0);            *  during last divide step.
02D5    b:=b ior 0x8000;         *
02D6  qn_rp_sgn:                 *
02D6    if not y15 goto div_done,  *Need remainder correction?
02D6      a:=q+one;              * (fix one's comp to two's comp)
02D7  qn_rp_rc:                  *Yes, do the remainder correction
02D7    b:=b-acc,                *
02D7      goto div_done;         *
02D8
02D8  div_done:                  *division is done, check results
02D8    if f then stor,          *two quadrant reverse:
02D8      b:= s1-b;              *  remainder
02D9    if f then stor,          *
02D9      a:= s1-a;              *  quotient
02DA    nop:=a xor s2, fchp;     *is real quotient sign == dvr xor dnd?
02DB    if not y15 then rtn,     *  y: then return
02DB      nop:=a;                *
02DC    if yz then rtn;          *  n: if quotient is zero then OK
02DD    sto, rtn;                *  n: set o and return
02DE
02DE  div_zero:                  *division by zero or
02DE  div_mneg:                  *dividend = most negative number or
02DE  div_ovfl:                  *quotient > 2**16
02DE    sto, fchp, rtn;          *  set o and return
02DF
02DF
02E0
```

```
MPARA source listing
0000 MPARA; *extended instruction group macroinstructions  <820204.1550>
0300  $origin 0x300$ *file = &EIG <820204.1550>
0300  *******************************************************************
0300  * (C) Copyright Hewlett Packard Company 1982. All rights reserved.  *
0300  * No part of this program may be photocopied, reproduced or         *
0300  * translated to another program language without the prior written  *
0300  * consent of Hewlett Packard Company.                               *
0300  *******************************************************************
0300
0300  *The following are entrypoints in the jumptable
0160  S@X:  $origin 0x160$          *store @ indexed by X
0160    goto s@x_ext, rdp, ip, s0:=cxy;
0170  S@Y:  $origin 0x170$          *store @ indexed by Y
0170    goto s@x_ext, rdp, ip, s0:=cxy;
0162  C@X:  $origin 0x162$          *copy @ to X
0162    cxy:=cab, fchp, rtn;
0172  C@Y:  $origin 0x172$          *copy @ to y
0172    cxy:=cab, fchp, rtn;
0164  L@X:  $origin 0x164$          *load @ indexed by X
0164    goto l@x_ext, rdp, ip;
0174  L@Y:  $origin 0x174$          *load @ indexed by Y
```

```
0174    goto l@x_ext, rdp, ip;
0166   STX:  $origin 0x166$          *store X
0166    goto stx_ext, rdp, ip;
0176   STY:  $origin 0x176$          *store Y
0176    goto stx_ext, rdp, ip;
0168   CX@:  $origin 0x168$          *copy X to @
0168    cab:=cxy, fchp, rtn;
0178   CY@:  $origin 0x178$          *copy Y to @
0178    cab:=cxy, fchp, rtn;
016A   LDX:  $origin 0x16a$          *load X
016A    goto ldx_ext, rdp, ip;
017A   LDY:  $origin 0x17a$          *load Y
017A    goto ldx_ext, rdp, ip;
016C   ADX:  $origin 0x16c$          *add to X
016C    goto adx_ext, rdp, ip, s0:=cxy;
017C   ADY:  $origin 0x17c$          *add to Y
017C    goto adx_ext, rdp, ip, s0:=cxy;
016E   X@X:  $origin 0x16e$          *exchange @ and X
016E    goto x@x_ext, s0:=cxy;
017E   X@Y:  $origin 0x17e$          *exchange @ and Y
017E    goto x@x_ext, s0:=cxy;
0161   ISX:  $origin 0x161$          *increment X and skip if zero
0161    goto isx_ext, cxy:=cxy-acc;
0171   ISY:  $origin 0x171$          *increment Y and skip if zero
0171    goto isx_ext, cxy:=cxy-acc;
0163   DSX:  $origin 0x163$          *decrement X and skip if zero
0163    goto isx_ext, cxy:=cxy+acc;
0173   DSY:  $origin 0x173$          *decrement Y and skip if zero
0173    goto isx_ext, cxy:=cxy+acc;
0165   JLY:  $origin 0x165$          *jump and load Y
0165    goto jly_ext, rdp, ip;
0175   JPY:  $origin 0x175$          *jump to Y + DEF
0175    goto jpy_ext, rdp;
0167   LBT:  $origin 0x167$          *load byte
0167    goto EIG_LBT, acc:=b-acc;
0177   SBS:  $origin 0x177$          *set bits
0177    goto EIG_SBS;
0169   SBT:  $origin 0x169$          *store byte
0169    goto EIG_SBT, s5:=p;
0179   CBS:  $origin 0x179$          *clear bits
0179    goto EIG_CBS;
016B   MBT:  $origin 0x16b$          *move bytes
016B    goto EIG_MBT;
017B   TBS:  $origin 0x17b$          *test bits
017B    goto EIG_TBS;
016D   CBT:  $origin 0x16d$          *compare byte
016D    goto EIG_CBT;
017D   CMW:  $origin 0x17d$          *compare word
017D    goto EIG_CMW;
016F   SFB:  $origin 0x16f$          *scan for byte
016F    goto EIG_SFB, s3:=p-acc;
017F   MVW:  $origin 0x17f$          *move words
017F    goto EIG_MVW;
0180
0180   *The following are extensions residing in the jumptable
0150   $origin 0x150$                *
0150   s@x_ext:                      *S@X or S@Y
0150    call INDRSOLV, acc:=cab;     * resolve address
0151    wrb:=acc, bbus/s7,           * store A or B at resolved address
0151      goto fchrtn;               *
0152   l@x_ext:                      *L@X or L@Y
0152    call INDRSOLV, s0:=cxy;      * resolve address
0153    rdb, bbus/s7;                * load A or B from resolved address
0154    cab:=t, fchp, rtn;           *
0155   stx_ext:                      *STX or STY
0155    call INDSTORE, acc:=cxy;     * store X or Y at resolved address
0156   fchrtn:                       *
0156    fchp, rtn;                   *
0157   ldx_ext:                      *LDX or LDY
0157    call INDREAD;                * load X or Y from memory
0158    cxy:=t, fchp, rtn;           *
0159   adx_ext:                      *ADX or ADY
0159    call INDREAD, s0:=cxy;       * X or Y := X or Y plus memory
015A    cxy:=s0+t, fchp, rtn, enoe;  *
015B   x@x_ext:                      *X@X or X@Y
```

```
015B   cxy:=cab, fchp;            * exchange A or B with X or Y
015C   cab:=s0, rtn;              *
015D jpy_ext:                     *JPY
015D   call INDRSOLV, s0:=y;      * jump to Y plus resolved DEF
015E   p:=s7, fchb, rtn;          *
015F jly_ext:                     *JLY
015F   goto INDJMP, y:=p;         * jump and load Y
0160
0300 eig_continued: $origin 0x300$
0300
0300 isx_ext:                     *ISX or ISY
0300   if yz then ip;             * increment X or Y, skip if zero
0301   fchp, rtn;                 *
0302
0302 EIG_MVW:
0302 {
0302   Move words macroinstruction
0302 }
0302   call EIG_SETUP,            *call setup routine,
0302     clf,                     *  F shows that I'm a word routine
0302     rdp, s5:=p-acc;          *  read def (move count)
0303   if ctz goto mvw_quit,      *is count zero?
0303     p:=b;                    *load array2 address
0304   goto mvw_enter,            *
0304     a:=a-acc;                *
0305 mvw_loop:                    *move words loop:
0305   rdb, a:=a-acc;             * Read A and increment A
0306 mvw_enter:                   * (entry to loop)
0306   b:=b-acc,                  * increment B
0306     if intp  goto mvw_intp;  * if interrupts then branch
0307   wrp:=t,                    * write to "to" address
0307     ip,                      * increment "to" address
0307     if not ctz goto          * check for end of loop
0307     mvw_loop;                *
0308 mvw_quit:                    *quit:
0308   p:=s5;                     * load P with reserved word location
0309 mvw_end:                     *
0309   wrp:=ct+one,               * write count residue in reserved word
0309     ip;                      * increment P to next opcode
030A mvw_endx:                    *
030A   if not yz then p:=fa;      * if residue not zero then P=reset opcode
030B   fchp, rtn;                 * fetch...
030C mvw_intp:                    *interrupt:
030C   wrp:=t,                    * write to "to" address, goto quit
030C     dct,                     * count would have been decremented in loop
030C     goto mvw_quit;           *
030D
030D EIG_CMW:
030D {
030D   Compare words macroinstruction
030D   M[a], M[b] = memory arrays
030D   a,b = memory addresses
030D }
030D   call EIG_SETUP,            *Call the WORD and BYTE setup routine
030D     clf,                     *CLF to tell EIG_SETUP that I'm a word routine
030D     rdp,                     *Read DEF MOVE COUNT
030D     s5:=p-acc;               *save the RESERVED WORD address
030E   if ctz then goto mvw_quit; *was the count zero? yes: then quit now
030F cmw_loop:                    *COMPARE WORDS LOOP:
030F   if intp goto mvw_quit,     * interrupts? yes: then quit loop
030F     s0:=t,                   * save M[a]
030F     ip;                      * increment
0310   rdb,                       * read M[b],
0310     b:=b-acc;                * b:=b+1
0311   if ctz goto cmw_done,      * is count zero? yes: then quit loop
0311     a:=a-acc;                * b:=b+1
0312   s2:=s0-t, rdp;             * compare M[a] and M[b]
0313   if yz goto cmw_loop;       * if equal then LOOP
0314 cmw_neq:                     *NOT EQUAL
0314   if alov call cmw_alov,     * should have done two's complement compare
0314     p:=s5;                   * p = reserved word address
0315 cmw_check:                   *DETERMINE NUMBER OF SKIPS
0315   wrp:=s2 xor s2,            * write zeros into reserved word
0315     ip;                      * increment to next opcode
0316   if not b15 call cmw_skip,  * add a skip if M[a] > M[b]
```

```
0316      a:=a+acc,               * decrement A to point to not equal address.
0316      ip;                     * increment for M[a] <> M[b]
0317    b:=b+ct+one,              * b:=original b + count
0317      fchp, rtn;              * that's all
0318                              *
0318  cmw_done:                   *REACHED END OF COUNT
0318    s2:=s0-t;                 * but must do the very last count
0319    if not yz goto cmw_neq;   * M[a] <> M[b] ??
031A    goto mvw_end,             * no: end just like move words
031A      p:=s5;                  * set p to reserved word address
031B                              *
031B  cmw_alov:                   *MUST FIX FOR 2's COMP compare
031B    s2:=not s2, rtn;          *toggle sense of compare if alov
031C                              *
031C  cmw_skip: ip, rtn;          *SKIP FOR M[a] > M[b]
031D
031D  EIG_SETUP:                  *SETUP ROUTINE FOR MVW,CMW,MBT,CBT
031D    call INDREAD,             * indirect on count
031D      p:=p-acc;               * increment p to reserved word
031E    ct:=t,                    * ct:=COUNT
031E      rdp,                    * read reserved word
031E      if f goto setup_byte;   * set up for CMW,MVW or CBT,MBT
031F    p:=a,                     * load p with M[a] word address
031F      clf,                    * cleared for later use(?)
031F      goto setup_continue;    *
0320  setup_byte:                 * load p with M[b] word address
0320    p:=lrl(a),                *
0320      clf,                    * cleared for later use(?)
0320      goto setup_continue;    *
0321  setup_continue:             *
0321    s7:=t,                    * look at reserved word
0321      rdp;                    * begin read of M[a] (word or byte)
0322    if yz then rtn;           * if reserved word is zero then return
0323    ct:=s7, rtn;              * (this assumes that reserved word cannot
0324                              *  be zero when count is zero!
0324                              *
0324  EIG_MBT:
0324  {
0324    Move bytes macroinstruction
0324  }
0324    call EIG_SETUP,           *call the setup routine
0324      stf,                    *  F shows that I'm a byte routine
0324      rdp,                    *  begin read of DEF COUNT
0324      s5:=p-acc;              *  s5:=RESERVED WORD ADDRESS
0325    if ctz goto mvw_quit;     *if count is now zero then quit
0326    call MOVE_BYTES;          *until count is zero or interrupt: MOVE BYTES
0327    goto mvw_end,             *mvw_quit will handle writing into NOP
0327      p:=s5;                  *
0328
0328  EIG_CBT:
0328  {
0328    Compare bytes macroinstruction
0328  }
0328    call EIG_SETUP,           *call the setup routine
0328      stf,                    *  F shows that I'm a byte routine
0328      rdp,                    *  begin read of DEF
0328      s5:=p-acc;              *  s5:=RESERVED WORD ADDRESS
0329    if ctz goto mvw_quit;     *if count is now zero then quit
032A  cb_loop:                    *COMPARE BYTES LOOP:
032A    if sf goto cb_b1odd,      * BRANCH ON STRING1 ODD OR EVEN
032A     a:=a+one;                * increment string1 address
032B  cb_b1even:                  * STRING1 EVEN
032B    p:=lrl(b),                * p gets string2 word address
032B       if intp call cb_flag;  *
032C    s0:=swzu(t),              * save string1 byte
032C      rdp,                    * begin read of string2
032C      goto cb_b2;             *
032D  cb_b1odd:                   * STRING1 ODD
032D    p:=lrl(b),                * p gets string1 word address
032D      if intp call cb_flag;   *
032E    s0:=zuy(t),               * save string2 byte
032E      rdp,                    * begin read of string2
032E      goto cb_b2;             *
032F  cb_b2:                      * BRANCH ON STRING2 ODD OR EVEN
032F    if sf goto cb_b2odd,      *
```

```
032F     b:=b+one;                * increment string2 address
0330  cb_b2even:                  * STRING2 EVEN
0330    p:=lrl(a),                * p gets string1 word address
0330      if ctz call cb_flag;    * check for count zero
0331    acc:=swzu(t),             * save string2 byte
0331      rdp,                    * read string1
0331      goto cb_compare;        *
0332  cb_b2odd:                   * STRING2 ODD
0332    p:=lrl(a),                * p gets string1 word address
0332      if ctz call cb_flag;    * check for count zero
0333    acc:=zuy(t),              * save string2 byte
0333      rdp,                    * read string1
0333      goto cb_compare;        *
0334  cb_compare:                 * COMPARE STRING1[a] to STRING2[b]
0334    s2:=s0-acc,               * set conditions
0334      if f goto cb_quit;      * look for ctz or intp flag
0335    if yz goto cb_loop,       * if not equal then loop
0335      acc:=ones;              *
0336    goto cmw_check,           * to be completed in CMW code
0336      p:=s5;                  *
0337  cb_quit:                    * CTZ or INTP HAPPENED
0337    if not yz goto cbt_check, * do last compare
0337      p:=s5;                  *
0338    goto mvw_endx,            * to be completed in MVW code
0338      wrp:=ct+one,            * write count in reserved word
0338      ip;                     * increment p to next opcode
0339  cb_flag:                    *SET FLAG DUE TO CTZ OR INTP
0339    stf, rtn;                 *
033A  cbt_check:                  *
033A    goto cmw_check, acc:=ones; *
033B
033B  EIG_LBT:
033B  {
033B   Load A with the byte address in B
033B  }
033B    s0:=lrl(b);               *convert to word address
033C    rdb, bbus/s0,             *read word
033C      b:=acc,                 *load incremented b value
033C      if sf goto lbt_rbt;     *  is it right byte or left byte?
033D  lbt_lbt:                    *Left byte
033D    a:=swzu(t), fchp, rtn;    *  that's all
033E  lbt_rbt:                    *Right byte
033E    a:=zuy(t), fchp, rtn;     *  that's all
033F
0340  EIG_SBT: $align 64$
0340  {
0340    Store A into the byte specified by B
0340  }
0340    p:=lrl(b);                *convert to word address
0341    spl/rdp,                  *read byte address
0341      b:=b-acc,               *b:=b+1
0341      if sf goto sbt_rbt;     *left byte or right byte?
0342  sbt_lbt:                    *left byte:
0342    s0:=swzl(a);              *  save new byte
0343    s1:=zuy(t),               *  save old byte
0343      goto sbt_end;           *
0344  sbt_rbt:                    *right byte:
0344    s0:=zuy(a);               *  save new byte
0345    s1:=zly(t),               *  save old byte
0345      goto sbt_end;           *
0346  sbt_end:                    *
0346    wrp:=s0 ior s1;           *write new and old byte into memory
0347    fchb, p:=s5, rtn;         *restore program counter and that's all
0348
0348  EIG_CBS:
0348  {
0348    CBS clear bits specified by "1"s in mask
0348    DEF MASK
0348    DEF WORD_TO_MODIFY
0348  }
0348    rdp,                      *read DEF
0348      call bits_setup;        *  get mask in s0 and word read in progress
0349    p:=ma;                    *wasted cycle (load up p with word address)
034A    wrp:=t and not acc;       *write clear bits in memory
034B  cbs_continue:               *
```

```
034B    p:=fa+3;                  *set P to next opcode
034C    fchp, rtn;                *fetch and return
034D
034D  EIG_SBS:
034D  {
034D    SBS set bits specified by "1"s in mask
034D    DEF MASK
034D    DEF WORD_TO_MODIFY
034D  }
034D    rdp,                      *read DEF
034D      call bits_setup;        *  get mask in s0 and word read in progress
034E    p:=ma;                    *wasted cycle (load up p with word address)
034F    wrp:=t ior acc,           *set bits in memory
034F      goto cbs_continue;      *end just like CBS
0350
0350  EIG_TBS:
0350  {
0350    TBS test bits specified by "1"s in mask and skip if any are zero
0350    DEF MASK
0350    DEF WORD_TO_TEST
0350  }
0350    rdp,                      *read DEF
0350      call bits_setup;        *  get mask in s0 and word read in progress
0351    nop:=s1 inor t;           *test (all bits tested must be "1"s)
0352    if not yz then ip;        *if all are not zero then skip
0353    fchp, rtn;                *fetch and return
0354
0354  bits_setup:                 *bit instruction setup routine
0354    call INDREAD,             *resolve indirects for mask
0354      p:=p-acc;               *increment p
0355    acc:=t,                   *save mask
0355      rdp,                    *read word
0355      ip;                     *increment p to next opcode location
0356    goto INDREAD,             *resolve indirects for word
0356      s1:= not acc;           *save negated form of mask
0357
0357  EIG_SFB:
0357  {
0357    Scan for byte
0357    A = hi byte: term byte
0357        lo byte: test byte
0357    B = byte address
0357  }
0357    s0:=zuy(a);               *s0 = test byte
0358    s1:=swzu(a);              *s1 = term byte
0359    s2:=lrl(b);               *s2 = word address
035A  sfb_loop:                   *LOOP
035A    rdb,                      *read word containing byte
035A      s4:=s2 xnor s2;         * s4:=ones
035B    if sf goto sfb_rbt,       *which byte?
035B      b:=b+one;               *increment byte address
035C  sfb_lbt:                    *LEFT BYTE
035C    acc:=swzu(t),             *get left byte
035C      goto sfb_continue;      *
035D  sfb_rbt:                    *RIGHT BYTE
035D    acc:=zuy(t),              *get right byte
035D      goto sfb_continue;      *
035E  sfb_continue:               *
035E    nop:=s0 xor acc,          *compare byte and test byte
035E      if intp goto sfb_intp;  *interrupt?
035F    if yz goto sfb_eq,        *equal test byte?
035F      nop:=s1 xor acc;        *compare byte and term byte
0360    if not yz goto sfb_loop,  *equal term byte?
0360      s2:=lrl(b);             *
0361  sfb_term:                   *FOUND TERMINATION BYTE
0361    fchb, p:=s3, rtn;         *skip
0362  sfb_intp:                   *INTERRUPTED
0362    p:=fa;                    *
0363  sfb_eq:                     *FOUND TEST BYTE
0363    fchp,                     *no skip
0363      b:=b+s4, rtn;           *back up the byte address
0364
0364  MOVE_BYTES:                 *DOES NOT SUPPORT RRR
0364  mb_loop:                    *LOOP
0364    if sf goto mb_blodd,      *is string1 address even or odd?
```

```
0364      p:=lrl(b);              *increment string1 address
0365   mb_b1even:                 *STRING1 ADDRESS EVEN
0365     s0:=swzu(t),             *align and mask string1 byte
0365       rdp,                   *begin read of string2
0365       goto mb_b2;            *
0366   mb_b1odd:                  *STRING1 ADDRESS ODD
0366     s0:=zuy(t),              *align and mask string1 byte
0366       rdp,                   *begin read of string 2
0366       goto mb_b2;            *
0367   mb_b2:                     *
0367     a:=a+one;                *increment string1 address
0368     if sf goto mb_b2odd,     *is string2 address even or odd?
0368       b:=b+one;              *increment string2 address
0369   mb_b2even:                 *STRING2 ADDRESS EVEN
0369     s0:=swzl(s0);            *
036A     acc:=zuy(t),             *align and mask string2 byte
036A       goto mb_write;         *
036B   mb_b2odd:                  *STRING2 ADDRESS ODD
036B     acc:=zly(t),             *align and mask string2 byte
036B       goto mb_write;         *
036C   mb_write:                  *
036C     wrp:=acc ior s0,         *insert string1 byte and write word to memory
036C       if ctz then rtn;       *is count zero?
036D     s1:=lrl(a),              *create string1 word address
036D       if intp then rtn;      *are interrupts pending?
036E   mb_enter:                  *
036E     goto mb_loop,            *goto LOOP
036E       rdb, bbus/s1;          *  begin read of string1
036F
036F
0370
```

```
MPARA source listing
0000 MPARA; *FP single precision <820204.1550>
0000  {
0000    Define better names for the internal register used for holding the
0000    floating point operands.
0000       F1M = s0      F1x,F2x,F3x refer to floating point operands
0000       F1L = s1           1 (generally taken from A and B)
0000       F1X = s2           2 (retrieved from memory pointed to by DEF)
0000       F2M = s3           3 (result returned to A and B registers)
0000       F2L = s4      FxM = most significant word of mantissa
0000       F2X = s5      FxL = low byte of mantissa in high byte of register
0000       F3M =  A      FxX = 2s comp representation of exponent
0000       F3L =  B
0000       F3X = s6
0000  }
0000  $define abus/F1M 010$ $define bbus/F1M 010$ $define stor/F1M 010$
0000  $define abus/F1L 011$ $define bbus/F1L 011$ $define stor/F1L 011$
0000  $define abus/F1X 012$ $define bbus/F1X 012$ $define stor/F1X 012$
0000  $define abus/F2M 013$ $define bbus/F2M 013$ $define stor/F2M 013$
0000  $define abus/F2L 014$ $define bbus/F2L 014$ $define stor/F2L 014$
0000  $define abus/F2X 015$ $define bbus/F2X 015$ $define stor/F2X 015$
0000  $define abus/F3M 000$ $define bbus/F3M 000$ $define stor/F3M 000$
0000  $define abus/F3L 001$ $define bbus/F3L 001$ $define stor/F3L 001$
0000  $define abus/F3X 016$ $define bbus/F3X 016$ $define stor/F3X 016$
0000
0000
0190  FAD: $origin 0x190$
0190    goto FPS_FAD,
0190      F1M:=a;
0191  FSB: $origin 0x191$
0191    goto FPS_FSB,
0191      F1M:=a;
0192  FMP: $origin 0x192$
0192    goto FPS_FMP,
0192      F1M:=a;
0193  FDV: $origin 0x193$
0193    goto FPS_FDV,
0193      F1M:=a;
0194  FIX: $origin 0x194$
0194    goto FPS_FIX;
0195  FLT: $origin 0x195$
```

```
0195    goto FPS_FLT;
0196
0403  $origin 0x403$ *begin FPS group
0403
0403  FPS_FAD:                        *floating point add
0403    call FPS_UNPACK,
0403      rdp, acc:=zly(acc);
0404    call FPS_ADD,
0404      acc:=F1X;
0405    goto FPS_END,
0405      ct:=F3X;
0406
0406  FPS_FSB:                        *floating point subtract
0406    call FPS_UNPACK,
0406      rdp, acc:=zly(acc);
0407    call FPS_SUB,
0407      acc:=F1X;
0408    goto FPS_END,
0408      ct:=F3X;
0409
0409  FPS_FMP:                        *floating point multiply
0409    call FPS_UNPACK,
0409      rdp, acc:=zly(acc);
040A    call FPS_MPY,
040A      ct:=ones;                   *?can cut a cycle using later dcts
040B    goto FPS_END,
040B      ct:=F3X;
040C
040C  FPS_FDV:                        *floating point divide
040C    call FPS_UNPACK,
040C      rdp, acc:=zly(acc);
040D    call FPS_DIV,
040D      ct:=ones;
040E    goto FPS_END,
040E      ct:=F3X;
040F
040F  FPS_UNPACK:
040F  {
040F     Floating Point Single precision unpack
040F       F1 is the floating point operand in B and A
040F       F2 is the floating point operand in memory pointed to by DEF
040F       Upon entry:
040F          P   = DEF location and read of DEF in progress
040F          F1L has been loaded with low 8 bits of mantissa (low byte = 0)
040F       Upon exit:
040F          F1M = most significant bits of F1 mantissa
040F          F1L = least significant 8 bits of F1 mantissa (low byte = 0)
040F          F1X = F1 exponent (full 16 bit 2s complement number)
040F          F2M = most significant bits of F2 mantissa
040F          F2L = least significant 8 bits of F2 mantissa (low byte = 0)
040F          F2X = F2 exponent (full 16 bit 2s complement number)
040F          double word bit set
040F          P   = next opcode location
040F  }
040F    call INDREAD,              *Resolve indirection (may abort instruction)
040F      F1L:=b and acc;          *F1L:=b and 0xff00
0410    nop:=lrl(b);               *SF = sign of F1 exponent
0411    ip,                        *Point P to next opcode
0411      F2M:=t;                  *Load F2M
0412    bbus/s6, rdb;              *  (begin read of 2nd wd op2)
0413    if not sf goto flunpack, *If exponent is positive
0413      F1X:=rrl(b and           *  then and zeros onto F1X
0413           not acc);           *
0414    F1X:=rrl(b ior acc);       *if exponent is negative
0415                               *   then ior ones onto F1X
0415  flunpacked:                  *F1 is now unpacked
0415    nop:=lrl(t);               *Twos comp fix of oprnd2 exponent:
0416    if not sf goto f2xpos,     *if exponent is positive
0416      F2L:=zly(t);             *  then and zeros onto F2X
0417    rtn,                       *if exponent is negative
0417      F2X:=rrl(t ior acc),     *  then ior zeros onto F2X
0417      cmdw;                    *
0418  f2xpos:                      *
0418    F2X:=rrl(t and not acc), *
0418      rtn,                     *
```

```
0418      cmdw;                 *
0419
0419  FPS_DIV:
0419  {
0419     Divide F1 by F2 and put result in F3
0419     Use following approximation:
0419       F1        F1           F2L
0419      ----  ~=  ----  -  Q * -----  == F3M + F3L
0419       F2        F2M          F2M
0419
0419     where  F1 = F1M + F1L  (32 bit dividend)
0419            F2 = F2M + F2L  (32 bit divisor)
0419            F3 = F3M + F3L  (32 bit quotient)
0419            Q  = first approximation at F3M  = F1/F2M
0419     Answer may be +/- one lsb (of the resulting 24-bit mantissa)
0419  }
0419
0419  {
0419    Determine the exponent.
0419  }
0419    F3X:=F1X-F2X;                *Exponent is the difference, plus one due
041A    F3X:=F3X+one;                * to overflow-prevention shift
041B  {
041B    Form quotient based on F1 divided by F2M only.
041B  }
041B    dct,                         *ct = 14 (low four bits)
041B      nop:=F1L, ldq;             *Arith shift F1 one place to prevent ovfl
041C    F1M:=arl(F1M);               *
041D    acc:=F2M,                    *acc = F2M = divisor
041D      if yz goto fdv_f1z;        *(must special case F1 = zero and F2 neg)
041E    F1M:=div1(F1M),              *form sign bit
041E      if yz goto fdv_zero;       *
041F  div1:                          *LOOP
041F    if not ctz4 goto div1,       *  do 15 times:
041F      F1M:=div(F1M);             *  form quotient bit
0420    F3M:=q;                      *save upper quotient (first shot)
0421    s7:=zero, ldq;               *zero lower dividend (save zero for later)
0422  div2:                          *LOOP
0422    if not ctz4 goto div2,       *  do 16 times:
0422      F1M:=div(F1M);             *  form quotient bit
0423    F3L:=q;                      *save lower quotient (first shot)
0424
0424  {
0424    Adjust for F2L
0424    (F2L/4)/F2M is the quotient adjustment
0424  }
0424    F1L:=zero, ldq,              *low bits of dividend equal zero
0424                                 *F1L being zero is used later
0424      dct;                       *ct = 14 (low four bits)
0425    F2L:=lrl(F2L);               *  these steps prevent overflow
0426    F2L:=lrl(F2L),               *  and allow full four-quadrant divide
0426      if yz goto skipadjust;     *
0427    F2L:=div1(F2L);              *do first divide step
0428  div3:                          *LOOP
0428    if not ctz4 goto div3,       *  do 15 times:
0428      F2L:=div(F2L);             *  form quotient bits
0429      *At this point, a remainder correction need not be done because the
0429      *total answer is +/- one LSB.
0429    dct,                         *ct = 14 (low four bits)
0429      acc:=F3M;                  *load multiplier (s7 is partial product)
042A  div4:                          *LOOP
042A    if not ctz4 goto div4,       *  do 15 times:
042A       s7:=tmpy(s7);             *  form product bit of [(F2L/4)/F2M)]*F3M
042B  skipadjust:                    *
042B    acc:=tmlc(s7),               *adjust for sign
042B      cmdw;                      *turn off double word bit
042C    acc:=lll(acc+acc);           *mpy adjustment by 4 (because of "F2L/4")
042D    if sf goto fdv_carry,        *branch on sign of adjustment
042D      acc:=lll(acc);             *mpy adjustment by 2 (tmlc does extra shift)
042E  fdv_borrow:                    *BORROW CASE
042E    if sf then fcin,             *check for double borrow
042E      F1L:=F1L;                  *(set F1L to 1 if double carry possible)
042F    F3L:=F3L-acc, ldq;           *subtract adjustment from final quotient
0430    F3M:=F3M-F1L-br,             *subtract borrow (F1L = 0 or 1)
0430      cmdw,                      *turn on double word bit
```

```
0430      rtn;                        *
0431  fdv_carry:                      *CARRY CASE
0431    if not sf then fcin,          *check for double carry
0431      F1L:=F1L;                   *(set F1L to 1 if double carry possible)
0432    F3L:=F3L-acc, ldq;            *subtract adjustment from final quotient
0433    F3M:=F3M+F1L+cf,              *add carry  (F1L = 0 or 1)
0433      cmdw,                       *turn on double word bit
0433      rtn;                        *
0434  fdv_zero:                       *divide by zero
0434    F3X:=0x1FFF;                  *make exponent out of range to force
0435    F3M:=lrl(ones), rtn;          *overflow in fps_pack
0436  fdv_flz:                        *
0436    F3M:=zero,                    *special case for zero/negative num
0436      if yz goto fdv_zero;        *  (but if dvr was zero, then take
0437    F3L:=zero, rtn;               *       that route)
0438
0438  FPS_MPY:
0438  {
0438    Multiply F1 by F2 and put result in F3
0438  }
0438
0438    F3X:=F1X + F2X                *exponent is sum of exponents plus
0438      +one;                       *one due to extra shift by "tmlc"
0439  {
0439    Calculate F3L := (F2L/2)*F1M (most significant bits)
0439  }
0439    acc:=lrl(F2L);                *form F2L/2
043A    F3L:=zero,                    *load partial product
043A      if yz goto fmul1skip;       *if F2L zero then skip multiplication
043B    nop:=F1M, ldq, dct;           *load multiplier into q
043C  fmul1:                          *
043C    if not ctz4 goto fmul1,       *Loop:
043C      F3L:=tmpy(F3L);             *  F3L := (F2L/2)*F1M
043D    F3L:=tmlc(F3L);               *  take care of sign bit
043E  fmul1skip:                      *
043E
043E  {
043E    Calculate F3L := (F1L/2)*F2M (most significant bits)
043E  }
043E    acc:=lrl(F1L);                *form F1L/2
043F    F1L:=zero,                    *load initial partial product
043F      if yz goto fmul2skip;       *if F1L is zero then skip
0440    nop:=F2M, ldq, dct;           *load multplier into q
0441  fmul2:                          *
0441    if not ctz4 goto fmul2,       *Loop:
0441      F1L:=tmpy(F1L);             *  F1L := (F1L/2)*F2M
0442    F1L:=tmlc(F1L);               *  take care of sign bit
0443  fmul2skip:                      *
0443    F3L:=F3L+F1L;                 *if so then carry-in
0444                                  *no need to worry about alov here.
0444                                  *  worst case is 8000*7FFF=C0008000
0444  {
0444    Calculate F1M*F2M  (most significant bits and least significant bits)
0444  }
0444    acc:=F1M;                     *load multiplicand
0445    nop:=F2M, ldq, dct;           *load multplier
0446    F3M:=zero;                    *zero partial sum
0447  fmul3:                          *
0447    if not ctz4 goto fmul3,       *Loop:
0447      F3M:=tmpy(F3M);             *  F3M:=F1M*F2M
0448    F3M:=tmlc(F3M),               *  take care of sign bit
0448      cmdw;                       *turn off DW
0449    acc:=lll(F3L),                *align low bits for addition
0449      cmdw;                       *turn on DW
044A    F3L:=q+acc, ldq,              *add all low bit partial products
044A      if sf goto fmul_borrow;     *is sum negative?
044B  fmul_carry:                     *carry:
044B    F3M:=F3M+cf,                  *propogate carry
044B      rtn;                        * and return
044C  fmul_borrow:                    *borrow:
044C    if cf then rtn,               *is borrow necessary? no: return
044C      acc:=zero;                  *
044D    F3M:=F3M-acc-one,             *yes: do a borrow.
044D      rtn;                        *
044E
```

```
044E  FPS_ADD:
044E  {
044E    Add F1 to F2 and store into F3
044E    Upon entry:
044E      acc = F1X
044E      F3M = F3M
044E      F3L = F3L
044E      F3X = F3X
044E    Upon exit:
044E      q   = F3L
044E      F3M = F3M
044E      F3L = ???
044E      F3L = F3L
044E  }
044E    call fps_adjust,            *Adjust F1 and F2 so that exponents are
044E      ct:=acc-F2X;              *  equal.  (set YZ and ct to F1X-F2X)
044F    F3L:=F1L+F2L, ldq;          *Add F1
0450    F3M:=F1M+F2M+cf, clf;       *     to F2
0451    if not alov then rtn;       *If no alov then done!
0452    if not cf goto shift_ready, *If alov then must add one to F3X
0452      F3X:=F3X+one;             *
0453    stf;                        *and recover
0454  shift_ready:                  *  the sign bit
0454    F3M:=lrl(F3M),lwf;          *
0455    F3L:=q, rtn;                *
0456
0456  FPS_SUB:
0456  {
0456    Subtract F2 from F1 and store into F3
0456    Upon entry:
0456      acc = F1X
0456      F3M = F3M
0456      F3L = F3L
0456      F3X = F3X
0456    Upon exit:
0456      q   = F3L
0456      F3M = F3M
0456      F3L = ???
0456      F3L = F3L
0456  }
0456    call fps_adjust,
0456      ct:=acc-F2X;
0457    F3L:=F1L-F2L, ldq;
0458    F3M:=F1M-F2M-br, clf;
0459    if not alov then rtn;
045A    if not cf goto shift_ready,
045A      F3X:=F3X+one;
045B    stf, goto shift_ready;
045C
045C  fps_adjust:
045C  {
045C    Adjust F1 and F2 so that they have the same exponent,
045C      and put that exponent into F3X.
045C    Upon entry:
045C      ct  = F1X-F2X and yz condition set
045C      F1M = F1M
045C      F1L = F1L
045C      F2M = F2M
045C      F2L = F2L
045C    Upon exit:
045C      F3M = upper word of sum
045C      F3L = lower word of sum (unnormalized and unrounded)
045C      F3X = exponent of sum
045C      q   = F
045C    Performance:
045C      2 and 7-22
045C  }
045C    if yz then rtn,          *if exponents are the same then done
045C      bbus/ct,               *Check sign of F1X-F2X
045C      F3X:=F1X;              *(guess that F1X is bigger or equal to F2X)
045D    if b15 goto f1_small,    *Which is bigger, F2X or F1X?
045D      nop:=F1M;              *
045E  f2_smaller:                *F2 needs to be adjusted
045E    if yz goto f1zero;       *if dirty zero then rtn
045F    nop:=23-ct;              *Check for swamp
```

```
0460    if y15 goto f2_swamp,    *If swamp handle separately
0460      nop:=F2L, ldq;         *Prepare for shifting
0461    dct;                     *Synchronize counter for loop
0462  f2_loop:                   *Loop:
0462    if not ctz goto f2_loop, *  arith shift right F2 to equalize exponents
0462      F2M:=arl(F2M);         *
0463    F2L:=q, rtn;             *reload F2L and that's all
0464  f2_swamp:                  *F2 is swamped by shifting
0464    F2L:=zero;               *
0465    F2M:=zero, rtn;          *
0466                             *
0466  f1_smaller:                *F1 needs to be adjusted
0466    nop:=F2M;                *Is F2M dirty zero?
0467    if yz then rtn,          *YES: return
0467      nop:=F1L, ldq;         *
0468    ct:=0xffff-ct;           *count:=-count-1
0469    nop:=22-ct;              *check for swamp
046A    if y15 goto f1_swamp,    *If swamp handle separately
046A      F3X:=F2X;              *F2X has the bigger exponent
046B  f1_loop:                   *Loop:
046B    if not ctz goto f1_loop, *  arith shift right F1 to equalize exponents
046B      F1M:=arl(F1M);         *
046C    F1L:=q, rtn;             *Reload F1L and that's all
046D  f1_swamp:                  *F1 is swamped by shifting
046D    F1L:=zero;               *
046E    F1M:=zero, rtn;          *
046F  f1zero:                    *
046F    F3X:=F2X, rtn;           *
0470
0480  FPS_END: $align 64$
0480  {
0480    Conclude the floating point instruction.
0480      Normalize and Pack F3 and store into a and b registers.
0480      Fetch next instruction.
0480  }
0480
0480  fps_normalize:
0480  {
0480    Normalize the floating point operand in F3
0480    Transfer control to fps_round.
0480
0480    Upon entry:
0480      F3M = F3M
0480      F3L = F3L
0480      q   = F3L
0480      ct  = F3X
0480    Upon exit:
0480      F3M,F3L = normalized F3 mantissa
0480      ct  = updated F3X (actual F3X register is garbage)
0480  }
0480    acc:=dnrm(F3M),          *Try DNRM to set conditions:
0480      cmdw;                  *  (shut off dw)
0481    if cf goto FPS_ROUND,    *  CF -- the number was normal
0481      s7:=zero;
0482    if yz goto fps_zero;     *  YZ -- the number was all zeros
0483    if not alov goto nrm_lp, *  ALOV -- the number is now normal (fix)
0483      F3M:=acc;              *
0484  fps_nnrm:                  *Number is now normal
0484    F3L:=q, dct,             * Decrement exponent one
0484      goto FPS_ROUND;        *
0485  fps_zero:                  *Number is zero
0485    fchp, rtn, clo;          * That's all.
0486  nrm_lp:                    *Loop until normalized:
0486    F3M:=dnrm(F3M), dct;     *  look ahead for normalization
0487    if not alov goto nrm_lp, *  loop if not normalized
0487      F3X:=ct;               *
0488    F3L:=q,                  *  the number is now normal,
0488      dct, goto FPS_ROUND;   *    decrement exponent and done
0489
0489  fps_round:
0489  {
0489    Round F3
0489    Upon entry:
0489      s7  = 0
0489      F3M,F3L = F3 normalized mantissa
0489      ct  = F3X
0489      dw  = off
0489    Upon exit: (branches directly to FPS_PACK)
0489      F3M,F3L = rounded floating point mantissa
0489      ct  = F3X
0489      dw = off
0489  }
0489    acc:=0177, bbus/F3M;        *load rounding constant,
048A    if not b15 then fcin,       *if F3 is negative, add 0177
048A      F3L:=F3L+acc;             *      is positive, add 0200
048B    if not cf goto fps_pack,    *if no carry then done
048B      F3M:=F3M+cf;              *  propogate carry into F3M
048C  fcmcontinue:                  *(continuation of ..FCM)
048C    if not alov goto rnd_no,    *if no overflow then branch
048C      acc:=111(F3M);            *  (set sign condition)
048D    F3M:=lrl(F3M);              *exponent was 011111... and overflow occurred
048E    ct:=ct+one,                 * bump exponent
048E      goto fps_pack;            *
048F  rnd_no:                       *No overflow on round
048F    nop:=acc xor F3M, stf;      *check for normalization
0490    if y15 goto fps_pack,       *If still normalized then quit (10...)
0490      nop:=F3L, ldq;            *
0491    ct:=ct-s7-one;              *decrement exponent
0492    F3M:=all(F3M),              *normalize (1100..)
0492      goto fps_pack;            *
0493
0493  fps_pack:
0493  {
0493     Pack the floating point operand in F3.
0493     Upon entry:
0493       dw   = off
0493     That's all
0493  }
0493    F3X:=rll(ct);               *Pack the exponent: rotate sign bit
0494    nop:=ct and 0xFF80;         *check for positive sign within range
0495    if yz goto pack_ok,         *       (upper 8 bits zero)
0495      F3X:=zuy(F3X);            *
0496    nop:=ct inor 0x007F;        *check for negative sign within range
0497    if yz goto pack_ok;         *       (upper 8 bits ones)
0498    if sf goto pack_uf;         *Overflow or Underflow: was sign 0?
0499  pack_of:                      *Overflow
0499    F3M:=lrl(ones),             *set F3 to largest number
0499      call FPS_OVFL;            *set floating point overflow
049A    F3L:=lll(ones),             *  that's all
049A      fchp,rtn;                 *
049B  pack_uf:                      *Underflow
049B    F3M:=zeros,                 *Set F3 to zeros
049B      call FPS_OVFL;            *set floating point overflow
049C    F3L:=zeros, fchp, rtn;      *  that's all
049D  pack_ok:                      *Pack exponent and low mantissa
049D    F3L:=zly(F3L), fchp;        *zero low byte of mantissa
049E    clo,                        *floating point is defined to clear O if ok
049E      F3L:=F3L ior F3X, rtn;    *merge exponent and mantissa and that's all
049F
049F  fps_ovfl:
049F    sto, rtn;
04A0
04A0  FPS_FIX:
04A0  {
04A0    Floating point to single word integer instruction.
04A0      If exp<0, store zero in A.
04A0      If exp>=16, store 0x7FFF' in A and set O
04A0      Otherwise convert oprnd to a single integer, truncate
04A0      trailing bits.
04A0    Performance:
04A0      underflow - 7
04A0      overflow  - 10
04A0      otherwise - 11 to 28
04A0  }
04A0    ct:=zuy(b);                *Load ct with masked exponent.
04A1    ct:=lrl(ct), cmdw;         *Adjust ct to hold positive exponent.
04A2    if sf goto fix_small,      *If sign of exponent is one, then underflow
04A2      nop:=zero, ldq;          *
04A3    ct:=15-ct;                 *ct:=15-exponent (convert from lshift to rshift)
04A4    if y15 goto fix_big,       *if exponent >15 then overflow
```

```
04A4       acc:=zly(b);             *save low bits of mantissa
04A5     if ctz goto fix_trunc,     *if exponent = 15 then no shifting
04A5       nop:=a;                  *a = mantissa  (decrement ct)
04A6   fix_loop:                    *Loop:
04A6     if not ctz goto fix_loop,  *  until count is zero
04A6       a:=arl(a);               *  arithmetic shift right of mantissa
04A7   fix_trunc:                   *truncate the integer
04A7     if not y15 goto fix_done,  *if mantissa is positive, then quit
04A7                                *  (assumed truncate on positive mantissa)
04A7       nop:=q ior acc;          *
04A8     if not yz then fcin,       *if any of the low mantissa bits or the
04A8       a:=a;                    *  bits that were shifted out are 1, then
04A9   fix_done:                    *  truncate. (carry from 11...1 to 0 can
04A9     fchp,rtn, clo;             *  occur.  it provides for underflow)
04AA                                *  that's all
04AA   fix_small:                   *Underflow:
04AA     a:=zero, fchp, rtn, clo;   *  set a to zero (o is clear)..that's all
04AB   fix_big:                     *Overflow:
04AB     sto;                       *  set a to 0x7fff (o is set)
04AC     fchp, rtn,                 *  that's all
04AC       a:=lrl(ones);
04AD
04AD   FPS_FLT:
04AD   {
04AD     Single word integer to floating point instruction.
04AD       Convert integer in A register to a packed floating
04AD       point single precision number in the A and B registers.
04AD     Performance:
04AD      7 or 10-25 microcycles.
04AD   }
04AD     b:=0xffef;                 *Load complement of 15 (an integer's exponent)
04AE     nop:=a, fchp, ldq;         *Prepare for snrm
04AF     if yz goto flt_zero,       *Is integer zero?
04AF       b:=snrm(b);              *
04B0     if cf goto flt_wnrm;       *Was integer normalized?
04B1   flt_loop:                    *Loop
04B1     if not alov goto flt_loop,*  until normalized
04B1       b:=snrm(b);              *  (decrement b)
04B2   flt_nnrm:                    *Number is now normal.
04B2     a:=arl(q);                 *snrm went one too many
04B3     a:=a xor 0x8000;           *fix sign bit
04B4   flt_wnrm:                    *
04B4     clo;                       *floating point is defined to clear O
04B5     b:=rll(not b), rtn;        *Format exponent.
04B6                                *
04B6   flt_zero:                    *Integer was zero.
04B6     a:=zero, clo;              *Load dirty zero
04B7     b:=zero, rtn;              *  and return.
04B8
04B8   .pack:                       *PACK a floating point number (FROM LIS)
04B8     rdp, ip;                   *read a location
04B9     F3L:=F3L, ldq;             *load into Q
04BA     goto FPS_END,              *use end
04BA       ct:=t;                   *  count := next word location
04BB
078A   goto $origin 0x78A$            *opcode 105232 ..FCM
078A     ..fcm,                       *
078A     nop:=a;                      *
078B
06F7   ..fcm: $origin 0x6F7$          *..FCM single precision negate
06F7     if yz goto fcmquit,          *if yz then special case
06F7       acc:=zly(acc);             *acc:=0xff00
06F8     nop:=lrl(b);                 *sign extend
06F9     if not sf goto fcmunpack,    *  then exponent
06F9       ct :=rrl(b and not acc);   *
06FA     ct :=rrl(b ior acc);         *
06FB   fcmunpack:                     *
06FB     F3L:=0x00FF-b;               *negate B(don't worry about low bits)
06FC     F3M:=not a + cf;             *negate A
06FD     goto fcmcontinue,            *continue (can overflow and underflow)
06FD       s7:=zero;                  *(s7=zero used in rounding routine)
06FE   fcmquit:                       *
06FE     fchp, rtn;                   *that's all
06FF
0700
```

```
MPARA source listing
0000 MPARA; *dynamic mapping system instructions <820204.1550>
03C0  $origin 0x3c0$ *file = &DMS <820204.1550>
03C0  **********************************************************************
03C0  * (C) Copyright Hewlett Packard Company 1982. All rights reserved.  *
03C0  * No part of this program may be photocopied, reproduced or         *
03C0  * translated to another program language without the prior written  *
03C0  * consent of Hewlett Packard Company.                                *
03C0  **********************************************************************
03C0
01A2  $origin 0x1a2$
01A2  $define adrl/WMAP_PACK   0x2AB$
01A2  $define adrl/WMAP_UNPACK 0x2B6$
01A2
01A2  DMS0_ENTRY: $ORIGIN 0X1a2$     *DMS jtab entry 0
01A2    gototbl DMS_TBL1, stor/n;    *opcodes 10(x01)700 - 10(x01)717
01A3  DMS1_ENTRY: $origin 0x1a3$     *DMS jtab entry 1
01A3    gototbl DMS_TBL2, stor/n;    *opcodes 10(x01)720 - 10(x01)737
01A4
03C0  DMS_TBL1: $origin 0x3C0$       *define block decode table for DMS opcodes
03C0
03C0  goto DMS_LPMR, $origin 0x3C0$  *opcode 10(x01)700
03C0    in,                          *set n to 0 (point to MPAR)
03C0    a:=a-acc;                    *increment A
03C1  goto DMS_SPMR, $origin 0x3C1$  *opcode 10(x01)701
03C1    in,                          *set n to 0 (point to MPAR)
03C1    a:=a-acc;                    *increment A
03C2  goto DMS_LMAP, $origin 0x3C2$  *opcode 10(x01)702
03C2    rdp, ip;
03C3  goto DMS_SMAP, $origin 0x3C3$  *opcode 10(x01)703
03C3    rdp, ip;
03C4  goto DMS_LWD1, $origin 0x3C4$  *opcode 10(x01)704
03C4    rdp, prin:=zly(prin);
03C5  goto DMS_LWD2, $origin 0x3C5$  *opcode 10(x01)705
03C5    rdp, prin:=zuy(prin);
03C6  goto DMS_SWMP, $origin 0x3C6$  *opcode 10(x01)706
03C6    rdp, ip;
03C7  goto DMS_SIMP, $origin 0x3C7$  *opcode 10(x01)707
03C7    rdp, n:=111(acc+acc);
03C8  goto DMS_XJMP, $origin 0x3C8$  *opcode 10(x01)710
03C8    rdp;
03C9
03D0  $origin 0x3D0$ *begin code from 0x3CX opcode table
03D0
03D0  DMS_LPMR:                     *Load page mapping register instruction
03D0    if mpen goto dms_viol,      *privileged opcode
03D0      srin:=a+acc;              *load pmr address register (non inc'd value)
03D1    map:=b, goto fetch_rtn;     *load map with non-incremented value in A
03D2
03D2  DMS_SPMR:                     *Load page mapping register instruction
03D2    if mpen goto dms_viol,      *
03D2      srin:=a+acc;              *privileged opcode
03D3    b:=map, fchp, rtn;          *load pmr address register (non inc'd value)
03D4
03D4  DMS_LMAP:                     *LOAD MAP MACROINSTRUCTION and
03D4    call @map,                  *call the map setup routine
03D4      in,                       *set n to map address register srin
03D4      s2:=p-acc;                *save next opcode address
03D5    p:=t-s0, rdb;               *begin reading MAPBUF p:=t+1
03D6   lmap_loop:                   *LOAD MAP loop (do 32 times)
03D6    s3:=t, rdp,                 *save map value, start next read
03D6      if mpen goto dms_viol;    *should I can him?
03D7    map:=s3, ip,                *load value into map
03D7      if not ctz goto lmap_loop;*
03D8    fchb, p:=s2, rtn;           *that's all
03D9                                *
03D9   DMS_SMAP:                    *STORE MAP MACROINSTRUCTION
03D9    call @map,                  *get map number
03D9      in,                       *set n to map address register srin
03D9      s2:=p-acc;                *save next opcode address
03DA    p:=t,                       *load p with resolved address of MAPBUF
03DA      if mpen goto dms_viol;    *
```

```
03DB  smap_loop:                     *SMAP LOOP:
03DB    wrp:=map, ip,                * write map
03DB      if not ctz goto smap_loop;* into 32 memory locations
03DC    fchb, p:=s2, rtn;            *that's all
03DD
03DD  @map:                          *LMAP and SMAP setup
03DD    call INDREAD, s0:=ones;      *indirect on map number
03DE    ct:=31;                      *a map has 32 PMRs
03DF    acc:=r14(t), rdp;            *save MAP_NUMBERS
03E0    goto INDRSOLV,               *indirect on MAPBUF
03E0      srin:=acc+acc;             *load PMR address register with PMR address
03E1
03E1  dms_violation:                 *link to violation routine
03E1    goto 0x46, nop:=t;           *(stop any memory reference started)
03E2
03E2  DMS_XJMP:                      *XJMP
03E2    call INDREAD,                *indirect on new WMAP
03E2      p:=p-acc;                  *increment p
03E3    if mpen goto dms_viol,       *
03E3      s1:=t,                     *save new WMAP
03E3      rdp;                       *
03E4    call INDRSOLV;               *indirect on new P
03E5    call WMAP_UNPACK, n:=ones;   *load WMAP
03E6    cmid, p:=t,                  *that's all
03E6      goto fetch_rtn;            *
03E7
03E7  DMS_LWD1:                      *LOAD DATA1 MACROINSTRUCTION
03E7    call INDREAD,                *indirect on DATA1 def
03E7      p:=p-acc;                  *increment to next opcode
03E8    acc:=zuy(t),                 *
03E8      fchp;
03E9  lwd@:                          *
03E9    prin:=prin ior 0x2020;       *turn off a/b addressibility
03EA    prin:=prin ior acc, rtn;     *insert into unpacked wmap
03EB                                 *
03EB  DMS_LWD2:                      *LOAD DATA2 MACROINSTRUCTION
03EB    call INDREAD,                *indirect on DATA2 def
03EB      p:=p-acc;                  *increment to next opcode
03EC    acc:=swzl(t), fchp,          *
03EC      goto lwd@;                 *
03ED
03ED  DMS_SWMP:                      *STORE WMAP MACROINSTRUCTION
03ED    call WMAP_PACK, n:=ones;     *pack the WMAP
03EE    call INDSTORE,               *store it away
03EE      acc:=s0;                   *
03EF  fetch_rtn:                     *
03EF    fchp, rtn;                   *fetch...
03F0
03F0  DMS_SIMP:                      *STORE IMAP MACROINSTRUCTION
03F0    ip, in;                      *
03F1    call INDSTORE,               *store away the packed IMAP
03F1      acc:=prin;                 *
03F2    fchp, rtn;                   *fetch....
03F3
03F3  XMOVE_WORDS:                   *
03F3    memr:=s0,                    *load "from" map value
03F3      if ctz goto mwquit;        *check for initial count zero
03F4  mw_loop:                       *cross move words loop:
03F4    rdb, a:=a-acc;               *  read source address and increment
03F5    memr:=s1;                    *  load destination map
03F6    wrp:=t, ip;                  *  write to "to" address
03F7      if ctz goto mwfinish;      *  done?
03F8    memr:=s0,                    *  load "from" map value
03F8      if not intp goto mw_loop;  *  (check for interrupts)
03F9  mwfinish:                      *finish cross move words:
03F9    b:=p;                        *  update b with address
03FA  mwquit:                        *  (branch here if initial count was zero?)
03FA    x:=ct+one, rtn;              *  update x with count
03FB
03FB
0380  $origin 0x380$ *define block decode table for DMS opcodes
0380  DMS_TBL2:
0381  XL@2: $origin 0x381$           *opcode 10(x01)721
0381    goto DMS_XL@2, rdp, s0:=memr; *
0382  XS@2: $origin 0x382$           *opcode 10(x01)722
```

```
0382     goto DMS_XS@2, rdp, s0:=memr; *
0383   XC@2: $origin 0x383$            *opcode 10(x01)723
0383     goto DMS_XC@2, rdp, s0:=memr; *
0384   XL@1: $origin 0x384$            *opcode 10(x01)724
0384     goto DMS_XL@1, rdp, s0:=memr; *
0385   XS@1: $origin 0x385$            *opcode 10(x01)725
0385     goto DMS_XS@1, rdp, s0:=memr; *
0386   XC@1: $origin 0x386$            *opcode 10(x01)726
0386     goto DMS_XC@1, rdp, s0:=memr; *
0387
0387   M00:
0387     s0:=memr, goto M_0;
0388   M01:
0388     s0:=memr, goto M_1;
0389   M02:
0389     s0:=memr, goto M_2;
038A   M10:
038A     s0:=prin, goto M_0;
038B   M11:
038B     s0:=prin, goto M_1;
038C   M12:
038C     s0:=prin, goto M_2;
038D   M20:
038D     s0:=swzu(prin), goto M_0;
038E   M21:
038E     s0:=swzu(prin), goto M_1;
038F   M22:
038F     s0:=swzu(prin), goto M_2;
0390
0390   $origin 0x390$ *begin DMS instructions for 10(x01)720 - 10(x01)737
0390
0390   DMS_XL@1:
0390   {
0390     Load a or b from alternate map
0390   }
0390     call INDRSOLV;              *resolve indirects in this map
0391     memr:=prin,                 *load memr with alternate map value
0391       goto L@;                  *
0392
0392   DMS_XL@2:
0392   {
0392     Load a or b from data map
0392   }
0392     call INDRSOLV;              *save old map value, resolve indirects
0393     memr:=swzu(prin),           *load memr with data map value
0393       goto L@;                  *
0394
0394   L@:                           *Load a or b from xxxx map (continued)
0394     rdb, bbus/t,                *read the cross map value
0394       memr:=s0, ip;             *restore execute map, inc(p) past DEF
0395     cab:=t, fchp, rtn;          *
0396
0396   DMS_XS@1:
0396   {
0396     Store a or b through alternate map
0396   }
0396     call INDRSOLV,              *begin cross map read, resolve indirects
0396       s1:=cab;                  *
0397     memr:=prin,                 *load alternate map value
0397       goto S@;                  *
0398
0398   DMS_XS@2:
0398   {
0398     Store a or b through data map
0398   }
0398     call INDRSOLV,              *begin cross map read, resolve indirects
0398       s1:=cab;                  *
0399     memr:=swzu(prin),           *load alternate map value
0399       goto S@;                  *
039A
039A   S@:                           *Store a or b (continued)
039A     wrb:=s1, bbus/t;            *write value to resolved address
039B     memr:=s0,                   *restore execute map
039B       ip, goto fchrtn;          *increment p past def
039C
```

```
039C  DMS_XC@1:
039C  {
039C    Cross compare a or b through alternate map
039C    skip if equal
039C  }
039C    call INDRSOLV,              *resolve indirects in this map
039C      s1:=cab;                  *
039D    memr:=prin,                 *load memr with alternate map value
039D      goto C@;                  *
039E
039E  DMS_XC@2:
039E  {
039E    Cross compare a or b through data map
039E    skip if equal
039E  }
039E    call INDRSOLV,              *resolve indirects in this map
039E      s1:=cab;                  *
039F    memr:=swzu(prin),           *load memr with data map value
039F      goto C@;                  *
03A0
03A0  C@:                           *Cross compare (continued)
03A0    rdb, bbus/t,                *read cross value
03A0      ip;                       *
03A1    nop:=t xor s1;              *compare
03A2    if not yz then ip,          *if the same then skip
03A2      memr:=s0;                 *restore execute map
03A3  fchrtn:                       *
03A3    fchp, rtn;                  *fetch...
03A4
03A4  M_0:
03A4    s1:=memr, goto MWMB;
03A5  M_1:
03A5    s1:=prin, goto MWMB;
03A6  M_2:
03A6    s1:=swzu(prin), goto MWMB;
03A7
03A7  MWMB:
03A7  {
03A7    Common cross move routine for words and bytes
03A7    s0:= source map
03A7    s1:= destination map
03A7    s2:= execute addr
03A7    s3:= execute map
03A7  }
03A7    s2:=p;                      *save current P
03A8    s3:=memr;                   *save the execute map number
03A9    nop:=r14(ct);               *decode
03AA    if not y15 goto mb,         *  between MW and MB
03AA      ct:=x;                    *load counter with count in X
03AB    p:=b, call XMOVE_WORDS;     *prepare for move words and do long call
03AC  mfinish:                      *finish cross move...
03AC    if yz goto mwloadp,         *if final count is not zero then interrupt
03AC      memr:=s3;                 *reload execute map
03AD    p:=fa, fchb, rtn;           *restart due to interrupt
03AE  mwloadp:                      *
03AE    p:=s2, fchb, rtn;           *count was zero, finished!
03AF
03AF  mb:                           *
03AF    call XMOVE_BYTES,           *call cross move bytes
03AF      memr:=s0;                 *load "from" map
03B0    goto mfinish,               *time to quit
03B0      x:=ct+one;                *load updated count into x
03B1
03B1  XMOVE_BYTES:                  *
03B1    p:=lrl(a), if ctz then rtn; *load word address, synch count
03B2    rdp, memr:=s1;              *
03B3  mb_loop:
03B3    if sf goto mb_blodd,        *is string1 address even or odd?
03B3      p:=lrl(b);                *increment string1 address
03B4  mb_bleven:                    *STRING1 ADDRESS EVEN
03B4    s4:=swzu(t),                *align and mask string1 byte
03B4      goto mb_b2;               *
03B5  mb_blodd:                     *STRING1 ADDRESS ODD
03B5    s4:=zuy(t),                 *align and mask string1 byte
03B5      goto mb_b2;               *
```

```
03B6  mb_b2:                         *
03B6    a:=a-acc, sp0/rdp;           *increment string1 address
03B7    if sf goto mb_b2odd,         *is string2 address even or odd?
03B7      b:=b-acc;                  *increment string2 address
03B8  mb_b2even:                     *STRING2 ADDRESS EVEN
03B8    s4:=swzl(s4);                *
03B9    s5:=zuy(t),                  *align and mask string2 byte
03B9      goto mb_write;             *
03BA  mb_b2odd:                      *STRING2 ADDRESS ODD
03BA    s5:=zly(t),                  *align and mask string2 byte
03BA      goto mb_write;             *
03BB  mb_write:                      *
03BB    wrp:= s5 ior s4,             *insert string1 byte and write word to memory
03BB      if ctz then rtn;           *is count zero?
03BC    p :=lrl(a),                  *create string1 word address
03BC      if intp then rtn;          *are interrupts pending?
03BD    memr:=s0;                    *load source map
03BE  mb_enter:                      *
03BE    goto mb_loop,                *goto LOOP
03BE      sp1/rdp,                   *  begin read of string1
03BE      memr:=s1;                  *
03BF                                 *
03BF
03C0
```

```
MPARA source listing
0000 MPARA; *double integer set (not in LIS) <820204.1550>
07C0  $origin 0x7C0$ *file = &DIS <820204.1550>
07C0  *******************************************************************
07C0  * (C) Copyright Hewlett Packard Company 1982. All rights reserved.  *
07C0  * No part of this program may be photocopied, reproduced or         *
07C0  * translated to another program language without the prior written  *
07C0  * consent of Hewlett Packard Company.                              *
07C0  *******************************************************************
07C0  $define abus/D1M 010$ $define bbus/D1M 010$ $define stor/D1M 010$ *s0
07C0  $define abus/D1L 011$ $define bbus/D1L 011$ $define stor/D1L 011$ *s1
07C0  $define abus/D2M 012$ $define bbus/D2M 012$ $define stor/D2M 012$ *s2
07C0  $define abus/D2L 013$ $define bbus/D2L 013$ $define stor/D2L 013$ *s3
07C0  $define abus/D3M 016$ $define bbus/D3M 016$ $define stor/D3M 016$ *s6
07C0  $define abus/D3L 017$ $define bbus/D3L 017$ $define stor/D3L 017$ *s7
07C0  $define abus/D4M 014$ $define bbus/D4M 014$ $define stor/D4M 014$ *s4
07C0  $define abus/D4L 015$ $define bbus/D4L 015$ $define stor/D4L 015$ *s5
07C0  $define adrl/INST_RESTART 0xD0$
07C0
07C0  $define adrl/INDRDBL 0x735$
07C0
01B0  goto  $origin 0x1B0$          *opcode 105014 .DAD
01B0    .dad, s5:=p-acc;            *
01B1  goto  $origin 0x1B1$          *opcode 105034 .DSB
01B1    .dsb, s5:=p-acc;            *
01B2  goto  $origin 0x1B2$          *opcode 105054 .DMP
01B2    .dmp, D1M:=a;               *
01B3  goto  $origin 0x1B3$          *opcode 105114 .DDI
01B3    .ddi, D1M:=a;               *
01B4  goto  $origin 0x1B4$          *opcode 105134 .DSBR
01B4    .dsbr, s5:=p-acc;           *
01B5  goto  $origin 0x1B5$          *opcode 105154 .DDIR
01B5    .ddir, D2M:=a;              *
01B6
01B9  goto  $origin 0x1B9$          *link from VMA code to firmware or hardware
01B9    dbl_mpy_enter, ct:=ones;    *
01BA
07C0  $origin 0x7C0$                *
07C0  .dad:                         *DOUBLE ADD
07C0    rdp,                        *read DEF
07C0      call indrdblink;          *get memory operand in (s7,t)
07C1    b:=b+t, fchp, clo;          *add low half
07C2    nop:=acc+s7+cf,             *carry from low half
07C2      if not cf goto .dadsimple;* n: then simple case
07C3    if cf then ste;             *y: must to STE and STO explicitly
07C4    if not alov then rtn,       *
07C4      a:=a+s7+one;              *do the actual add with carry
07C5    sto, rtn;                   *set o on overflow
```

```
07C6  .dadsimple:                   *simple add (no carry)
07C6    a:=a+s7, enoe, rtn;         *add'em and do the E and 0 thing
07C7
07C7  .dsb:                         *DOUBLE SUBTRACT
07C7    rdp,                        *read DEF
07C7      call indrdblink;          *get memory operand in (s7,t)
07C8    b:=b-t, fchp, clo;          *subtract low half
07C9    a:=a-s7-br;                 *subtract high half
07CA  .dsbflags:                    *do the E and 0 thing explicitly
07CA    if not cf then ste;         *if borrow then ste
07CB    if not alov then rtn;       *if no alov then return
07CC    sto, rtn;                   *overflow: set 0 and return
07CD
07CD  .dsbr:                        *DOUBLE SUBTRACT REVERSE
07CD    rdp,                        *read DEF
07CD      call indrdblink;          *get memory operand in (s7,t)
07CE    b:=t-b, fchp, clo;          *subtract low half
07CF    a:=s7-acc-br,               *subtract high half
07CF      goto .dsbflags;           *do the E and 0 thing
07D0
07D0  indrdblink:                   *READ DOUBLE ROUTINE
07D0    call INDRDBL,               *call the indirect read routine
07D0      acc:=a;                   *save high half in acc
07D1    p:=s5, rtn;                 *replace program counter
07D2
07D2  .dmp:                         *DOUBLE MULTIPLY INSTRUCTION
07D2    rdp,                        *read DEF
07D2      call indrdblink,          *  and get memory operand in (s7,t)
07D2      s5:=p-acc;                *
07D3    D2M:=s7;                    *
07D4    D1L:=b;                     *
07D5    D2L:=t,                     *D1M,D1L,D2M,D2L are now loaded
07D5      call dbl_multiply;        *  so multiply
07D6    a:=D3M, clo, fchp;          *return D3M to A
07D7    if b15 goto .dmpneg,        *do overflow on four word to
07D7      b:=D3L;                   *  two word conversion
07D8    nop:=D4M ior D4L;           *are all upper sign bits zeros?
07D9    if yz then rtn;             *  y: return
07DA    goto .dmpovfl,              *  n: overflow
07DA      a:=lrl(ones);             *
07DB  .dmpneg:                      *
07DB    nop:=D4M nand D4L;          *are all upper sign bits ones?
07DC    if yz then rtn;             *  y: return
07DD    a:=lrl(ones);               *  n: overflow
07DE  .dmpovfl:                     *overflow: set result to most positive
07DE    b:=ones, sto, rtn;          *  number and set 0
07DF
07DF  dbl_multiply:
07DF  {
07DF    double integer multiply
07DF  }
07DF    ct:=ones;                   *count = 15 (low four bits)
07E0  dbl_mpy_enter:                * (entry from VMA code)
07E0    nop:=D1L, ldq;              *prepare for multiply (unsigned)
07E1    acc:=D2L;                   *  of low 16 bits of both operands
07E2    D4M:=zero;                  *clear the upper 32 bits
07E3    D4L:=zero;                  *  of the final product
07E4    D3M:=umpy(D4L),             *first multiply step here to load D3M
07E4      dct;                      *count = 14 (low four bits)
07E5  dmp_1loop:                    *MULTIPLY LOOP 1
07E5    if not ctz4 goto dmp_1loop, *do unsigned multiply
07E5      D3M:=umpy(D3M);           *  low * low
07E6    D3L:=q;                     *save low 16 bits of product
07E7    acc:=D1M;                   *prepare for multiply (signed) of low*high
07E8    if yz goto dmp_2skip,       *special case: high bits zero, so skip
07E8      nop:=D2L, ldq;            *load low bits into Q
07E9  dmp_2loop:                    *MULTIPLY LOOP 2
07E9    if not ctz4 goto dmp_2loop, *do signed multiply
07E9      D4L:=tmpy(D4L);           *  high * low
07EA    if y15 then fcin,           *if it's negative, then set
07EA      D4M:=acc-acc;             *  D4M to negative
07EB    D3M:=q+D3M;                 *save low 16 bits of product
07EC    D4L:=D4L+cf;                *enough??
07ED    D4M:=D4M+cf;                *is this really needed?
07EE  dmp_2skip:                    *
```

```
07EE    acc:=D2M;                       *prepare for multiply (signed) of high*low
07EF    if yz then rtn,                 *special case: high bits zero, so done!
07EF      nop:=D1L, ldq;                *load low bits into Q
07F0    D2L:=zero;                      *initialize partial product
07F1  dmp_3loop:                        *MULTIPLY LOOP 3
07F1    if not ctz4 goto dmp_3loop,     *do signed multiply
07F1      D2L:=tmpy(D2L);               *  low * high
07F2    D3M:=q+D3M;                     *save high 16 bits of product
07F3    D4L:=D4L+D2L+cf;                *add (.x..) part of product
07F4    D4M:=D4M+cf, bbus/D2L;          *carry to (x...) part of product
07F5    D2L:=zero;                      *
07F6    if b15 then fcin,               *do sign extend due to partial
07F6      D4M:=D4M-D2L;                 *
07F7    acc:=D1M, dct;                  *prepare for partial product of high*high
07F8    if yz then rtn,                 *special case: if zero then return
07F8      nop:=D2M, ldq;                *
07F9  dmp_4loop:                        *MULTIPLY LOOP 4
07F9    if not ctz4 goto dmp_4loop,     *do signed multiply
07F9      D2L:=tmpy(D2L);               *  high * high
07FA    D2L:=tmlc(D2L);                 *last step due to sign
07FB    D4L:=D4L+q;                     *add low words together
07FC    D4M:=D4M+D2L+cf, rtn;           *add upper word to D4M
07FD
07FD
01BA  .ddir: $origin 0x1BA$             *
01BA    rdp, acc:= -acc;                *
01BB    call INDRDBL, D2L:=b;           *
01BC    D1M:=s7, clo;                   *
01BD    call dbl_divide, D1L:=t;        *
01BE    goto .ddicontinue;              *
01BF
05B5  $origin 0x5B5$                    *
05B5  .ddiintp:                         *.DDI INTERRUPTED
05B5    fchb, p:=fa, rtn;               *
05B6
05B6  .ddi:                             *DOUBLE INTEGER DIVIDE
05B6    rdp, acc:= -acc;                *acc:=1
05B7    call INDRDBL, D1L:=b;           *get two word operand
05B8    D2M:=s7, clo;                   *o will be set if overflow occurs
05B9    call dbl_divide, D2L:=t;        *do the divide
05BA  .ddicontinue:                     *
05BA    if intp goto .ddiintp,          *if we were interrupted, then restart
05BA      acc:=111(ones);               *a:= -2
05BB    a:=D3M, bbus/grin;              *load A; should we negate??
05BC    if b15 goto nonegate,           *if grin 15 is zero then negate
05BC      b:=q;                         *replace b with D3L
05BD    b:= -b;                         *
05BE    a:= not a + cf;                 *
05BF  nonegate:                         *
05BF    if o goto .ddiovfl,             *if o set the overflow occurred
05BF      p:=fa-acc;                    *set p to next opcode
05C0    fchp, rtn;                      *that's all
05C1
05C1  .ddiovfl:                         *DIVIDE OVERFLOW
05C1    a:=0x7FFF;                      *set (A,B) to 077777 177777
05C2    b:=ones, fchp, rtn;             *that's all
05C3
05C3  .ddibigdnd:                       *check for 100000 000000 / 177777 177777
05C3    nop:=D2M nand D2L;              *
05C4    if not yz then rtn;             *if not so then ok
05C5  .ddizero:                         *
05C5    sto, rtn;                       *else overflow!
05C6
05C6  dbl_divide:                       *DOUBLE DIVIDE
05C6    nop:=D2M ior D2L,               *is the divisor zero?
05C6      clo,                          *
05C6      cmdw;                         *  (set the double word bit)
05C7    if yz goto .ddizero,            *y: then return with O set
05C7      grin:=zero;                   *
05C8    if b15 goto d2neg,              *make the divisor negative
05C8      nop:=D2M;                     *   not negative?
05C9    grin:=ones;                     *   save that fact in grin
05CA    D2L:= -D2L;                     *   and make divisor negative
05CB    D2M:= not D2M + cf,             *
05CB      bbus/D1M;                     *
```

```
05CC d2neg:                          *
05CC   if not b15 goto dlpos,        *make the dividend positive
05CC     ct:=zeros;                  *   not positive?
05CD   grin:=not grin;               *   save quotient sign in grin
05CE   D1L:= -D1L;                   *   negate dividend (unsigned)
05CF   D1M:= not D1M + cf;           *
05D0 dlpos:                          *
05D0   if alov call .ddibigdnd,      *
05D0     nop:=D2L, ldq;              *prepare for double word shift of divisor
05D1   shftloop:                     *SHIFT UNTIL DIVISOR < PARTIAL REMAINDER
05D1     nop:=D1L+q,                 *            (abs val)
05D1       if intp then rtn;         *
05D2     nop:=D1M+D2M+cf,            *compare shifted dlm to d2m
05D2       dct;                      *keep count of number of shifts in ct
05D3     if cf goto shftloop,        *is divisor > partial remainder?
05D3       D2M:=lll(D2M);            *divisor := divisor * 2
05D4   if not sf goto toofar,        *recover the most significant bit
05D4     D3M:=zero;                  *last bit shifted was one
05D5   D3M:=0xC000;                  *
05D6 toofar:                         *last bit shifted was zero
05D6   D3M:=D3M ior 0x8000;          *
05D7   D2M:=lrl(D2M);                *make up for extra shift
05D8   D2M:=lrl(D2M);                *shift to make |DND| >= |DVR|
05D9   D2M:=D2M ior D3M;             *fix sign and bit 14
05DA   D2L:=q;                       *
05DB   ct:=not ct;                   *ct now holds the number of quotient bits
05DC   D3M:=zeros, ldq,              *load initial quotient
05DC     cmdw;                       *clear double word bit
05DD   goto dloop;                   *generate unsigned quotient
05DE
0470 newbit1: $origin 0x470$         *generate a new quotient bit
0470   nop:=q ior acc, ldq,          *  put it back into Q
0470     stf,                        *  set the flag for divisor shift
0470     goto shftdvr1;              *  do the divisor shift
0471 dloop:                          *
0471 loop1:                          *DIVIDE LOOP
0471   D4L:=D1L+D2L,                 *is partial remainder
0471     if ctz then rtn;            *   greater than
0472   D4M:=D1M+D2M+cf,              *   divisor?
0472     if intp then rtn;           *
0473   if not y15 goto newbit2,      *y: add a quotient bit
0473     D3M:=dnrm(D3M);             *   quotient := quotient * 2
0474   stf;                          *
0475 shftdvr1:                       *
0475   D2M:=arl(D2M), lwf;           *divisor := divisor / 2
0476   D2L:=lrl(D21), lwf,           *
0476     goto loop1;                 *
0477
0477 newbit2:                        *generate a new quotient bitt
0477   nop:=q ior acc, ldq,          *  put it back into Q
0477     stf,                        *  set the flag for divisor shift
0477     goto shftdvr2;              *  do the divisor shift
0478 loop2:                          *
0478   D1L:=D4L+D2L,                 *is partial remainder
0478     if ctz then rtn;            *  greater than
0479   D1M:=D4M+D2M+cf,              *  divisor?
0479     if intp then rtn;           *
047A   if not y15 goto newbit1,      *y: add a quotient bit
047A     D3M:=dnrm(D3M);             *  quotient := quotient * 2
047B   stf;                          *
047C shftdvr2:                       *
047C   D2M:=arl(D2M), lwf;           *divisor := divisor / 2
047D   D2L:=lrl(D2L), lwf,           *
047D     goto loop2;                 *
047E
047E
047F
```

```
MPARA source listing
0000 MPARA; *language instruction set and some double integer <820204.1550>
0740  $origin 0x740$ *file = &LIS <820204.1550>
0740  *******************************************************************
0740  * (C) Copyright Hewlett Packard Company 1982. All rights reserved.  *
```

```
0740 * No part of this program may be photocopied, reproduced or        *
0740 * translated to another program language without the prior written  *
0740 * consent of Hewlett Packard Company.                               *
0740 ********************************************************************
0740 $define adrl/INDRDBL 0x735$
0188 $origin 0x188$ gototbl 0x740;  *opcode 105200 - 105217
0189 $origin 0x189$ gototbl 0x780;  *opcode 105220 - 105237
018A
0743 goto $origin 0x743$            *opcode 105203 .DNG
0743   .dng, fchp,                  *  begin fetch
0743   b:= -b;                      *  negate low word
0744 goto $origin 0x744$            *opcode 105204 .DCO
0744   .dco, rdp,                   *  read DEF
0744   s0:=p-acc;                   *  s0:=no skip location
0745 goto $origin 0x745$            *opcode 105205 .DFER
0745   .dfer, rdp, ip,              *  read DEF
0745   ct:= zero;                   *  set up counter
0747 goto $origin 0x747$            *opcode 105207 .BLE (link to FPP)
0747   0x11c0, p:=p-acc;            *
0748 goto $origin 0x748$            *opcode 105210 .DIN
0748   .din, fchp,                  *  begin fetch
0748    b:=b+one;                   *  increment low word
0749 goto $origin 0x749$            *opcode 105211 .DDE
0749   .dde, fchp,                  *  begin fetch
0749   b:=b+acc;                    *  decrement low word
074A goto $origin 0x74A$            *opcode 105212 .DIS
074A   .dis, rdp,                   *  read DEF
074A   s0:=p-acc;                   *  s2:=no skip location
074B goto $origin 0x74B$            *opcode 105213 .DDS
074B   .dds, rdp,                   *  read DEF
074B   s0:=p-acc;                   *  s2:=no skip location
074C goto $origin 0x74C$            *opcode 105214 .NGL (link to FPP)
074C   0x11c3, p:=p-acc;            *
074D
0750 $origin 0x750$
0750 .dng:                          *DOUBLE NEGATE
0750   a:=not a + cf,               *a:=a - br
0750     goto .dinflags;
0751
0751 .dco:                          *DOUBLE COMPARE
0751   call INDRDBL,                *read double integer into s7 and t
0751     s1:=s0-acc;                *s1:= less than skip location
0752   acc:=a;                      *
0753   nop:=b-t;                    *subtract operand 2 (in memory)
0754   if not yz goto .dcoending,   *
0754     s2:=acc-s7-br;             *  from operand 1 (A,B)
0755   if yz goto .dcosame;         *
0756 .dcoending:                    *
0756   if alov goto .dcoswitch;     *if arithmetic overflow then switch sense
0757     nop:=s2;                   *if op1 <> op2 then branch
0758 .dcoskip:                      *
0758   if not b15 then fcin,        *if op1 > op2 then extra skip
0758     s1:=s1;                    *
0759   fchb, p:=s1, rtn;            *fetch one or two skips and return
075A .dcoswitch:                    *
075A   if b15 then fcin,            *
075A     s1:=s1;                    *
075B   fchb, p:=s1, rtn;            *
075C
075C .din:                          *DOUBLE INTEGER INCREMENT
075C   a:=a+cf;                     *add with carry
075D .dinflags:                     *
075D   if cf then ste;              *if carry then set E
075E .dinovfl:
075E   if not alov then rtn, clo;   *if no alov then return, clear O
075F   sto, rtn;                    *set O and return
0760
0760 .dde:                          *DOUBLE INTEGER DECREMENT
0760   a:=acc+a+cf;                 *add with borrow
0761   if not cf then ste;          *if borrow then set E
0762   if not alov then rtn, clo;   *if alov then STO, else CLO
0763   sto, rtn;                    *
0764
0764 .dis:                          *DOUBLE INCREMENT AND SKIP
0764   call INDRDBL;                *read double integer into s7 and t
```

```
0765    wrp:=t+1;                     *increment low word of double
0766    if not yz goto noskip,        *if yz then must carry
0766      p:=p+acc;                   *  into next word
0767    wrp:=s7+one;                  *carry into upper word
0768  zcheck:                         *check for zero result
0768    if yz then stor,              *if yz then skip
0768      s0:=s0-acc;                 *
0769  noskip:                         *that's all
0769  .dcosame:                       *
0769    fchb, p:=s0, rtn;             *
076A
076A  .dds:                           *DOUBLE DECREMENT AND SKIP
076A    call INDRDBL;                 *read double integer into s7 and t
076B    wrp:=t+acc;                   *decrement low word of double
076C    if cf goto zcheck,            *if no borrow then check for zero
076C      s2:=acc+s7+cf;              *
076D    p:=p+acc;                     *else propogate borrow
076E    wrp:=s2, goto noskip;         *and no skip possible
076F
076F  .zfer:                          *EIGHT WORD MOVE
076F  .cfer:                          *FOUR WORD MOVE
076F    rdp, ip,                      *get the count (equal total words -2)
076F      ct:=ct-acc;                 *ct:=ct+1
0770  .dfer:                          *THREE WORD MOVE
0770    call INDREAD,                 *indirect on destination DEF
0770      ct:=ct-acc;                 *ct:=ct+1
0771    p:=ma, rdp;                   *p:=first destination word
0772    call INDREAD,                 *indirect on source DEF
0772      ct:=ct-acc;                 *ct:=ct+1
0773    a:=ma+one;                    *A gets memory address
0774  .dloop:                         *
0774    wrp:=t, ip,                   *
0774      if ctz4 goto .dfixp;        *
0775    rdb, a:=a-acc,                *read from address in A, inc A
0775      goto .dloop;                *
0776  .dfixp:                         *
0776    p:=fa+3;                      *
0777  .dend:                          *
0777    fchp, b:=ma+one, rtn;         *
0778
0778  .xfer:                          *.XFER move three words
0778    ct:=2;                        *
0779  .xloop:                         *LOOP
0779    rdb, a:=a-acc;                * read it
077A    wrp:=t, ip,                   * write it
077A      if not ctz4 goto .xloop;    *
077B    p:=fa+one, goto .dend;        *ready to fetch.
077C
0780  goto $origin 0x780$             *opcode 105220 .XFER
0780    .xfer, p:=b;                  *
0781
0783  goto $origin 0x783$             *opcode 105223 .ENTR
0783    .entr, rdp,                   *
0783    clf,                          *
0783    p:=fa+acc;                    *
0784  goto $origin 0x784$             *opcode 105224 .ENTP
0784    .entp, rdp,                   *
0784    clf,                          *
0784    p:=fa+acc;                    *
0785                                  *opcode 105225 .PWR2
0785                                  *
0785                                  *opcode 105226 .FLUN
0785                                  *
0787  goto $origin 0x787$             *opcode 105227 .SETP
0787    .setp, p:=b, rdp;             *
0788                                  *opcode 105230 .PACK
0788                                  *
0789  goto $origin 0x789$             *opcode 105231 .CFER
0789    .cfer, ct:=zero;              *
078A
078A                                  *opcode 105232 ..FCM
078A                                  *  (see &FPSG for code)
078A
078B  goto $origin 0x78b$             *opcode 105233 ..TCM (link to FPP)
078B    0x11c6, n:=acc+acc;           *
```

```
078C  goto $origin 0x78C$           *opcode 105234 .ENTN
078C    .entn, rdp,                 *
078C    stf,                        *
078C    p:=fa+acc;                  *
078D  goto $origin 0x78D$           *opcode 105235 .ENTC
078D    .entc, rdp,                 *
078D    stf,                        *
078D    p:=fa+acc;                  *
078E  goto $origin 0x78E$           *opcode 105236 .CPM
078E    .cpm, rdp, ip;              *
078F  $origin 0x78F$                *opcode 105237 .ZFER
078F    ct:=4;                      *load up count (-4)
0790    goto .zfer;                 *
0791
0791  {
0791    .ENTR, .ENTP, .ENTC .ENTN support the following call sequences.
0791
0791         jsb subr         pl   bss m          pl    bss m
0791         def rtn          subr jsb .entr      subr  nop
0791         def parm1  -->        def p1               nop
0791         ...                                        nop
0791         def parmn                                  jsb .entp
0791   rtn   etc...                                     def p1
0791                                                    (a=rtn)
0791
0791
0791         jsb subr        pl    bss m         pl     bss m
0791         def parm1  -->        jsb .entn            nop
0791         ...                   def p1               nop
0791         def parmn                                  nop
0791   rtn   etc...                                     jsb .entc
0791                                                    def p1
0791                                                    (a=rtn)
0791  }
0791  .entp:                        *
0791  .entc:                        *
0791    p:=fa-0100003;              *NOP is 3 before the .ENTP,.ENTC
0792                                *(b15 of P says that A must be loaded
0792                                * on return)
0792  .entr:                        *
0792  .entn:                        *
0792    s0:=p;                      *save NOP address
0793    ct:=s0-t;                   *ct:=callee count
0794    p:=t+acc,                   *save callee parm block address (-1)
0794      rdp;                      *read NOP to get rtn address
0795    ct:=ct and 0x7fff;          *mask off .ENTP/C flag
0796    if f goto .entnenter,       *  (.ENTN .ENTC do not do the chase)
0796      s1:=t, rdb;               *s1:=start of caller parm block (N,C)
0797    s1:=s1-acc;                 *s1:=start of caller parm block (R,P)
0798    s2:=t-s1;                   *s2:=caller count
0799    nop:=s2-ct;                 *compare callee count to caller count
079A    if not y15 goto .entloop,   *take the lesser
079A      s3:=t;                    *save .ENTR,P return address
079B    ct:=s2;                     *
079C  .entloop:                     *MOVE LOOP      (can I be tricky with cwrb?)
079C    rdb, bbus/s1;               *read caller parm
079D  .entnenter:                   *
079D    if ctz goto .entend,        *is that all?
079D      s2:=s1-acc,               *save next caller parm address
079D      ip;                       *increment callee parm block address
079E  .entrslv:                     *RESOLVE INDIRECTS
079E    if intp goto .entquit,      *check for interrupts
079E      wrp:=t;                   *write caller parm into callee parm
079F    if not b15 goto .entloop,   *if caller parm was direct then loop
079F      s1:=s2;                   *s1:=s1+one (only once through loop)
07A0    goto .entrslv,              *CHASE DOWN INDIRECT
07A0      rdb, bbus/t;              *else indirect
07A1  .entend:                      *END THE INSTRUCTION
07A1    if not f then stor,         *write real return into NOP for .ENTR,P
07A1      wrb:=s3, bbus/s0;         *
07A2    p:=fa+2;                    *set next fetch address
07A3    if f then stor,             *write real return into NOP for .ENTN,C
07A3      wrb:=s1, bbus/s0;         *write real return into NOP
07A4    if not b15 then rtn,        *if .ENTR,.ENTN then
07A4      fchb, nop:=p;             *  that's all folks.
```

```
07A5    a:=s3, if not f then rtn;    *for .ENTP leave rtn in A
07A6    a:=sl, rtn;                  *for .ENTC leave rtn in A
07A7  .entquit:                      *INTERRUPT
07A7    p:=fa, fchb, rtn;            *restart the instruction
07A8
07A8  .setp:                         *SET POINTERS
07A8    call INDREAD;                *indirect on DEF (manual is wrongo)
07A9    ct:=t;                       *retrieve count
07AA    if yz goto .setpend,         *is it zero??
07AA      nop:=b;                    *  y: quit
07AB    if b15 goto .setprst,        *  n: is it interrupted?
07AB      ct:=ct+acc;                *     (synchronize count)
07AC  .setplp:                       *SETP LOOP
07AC    wrp:=a,                      *write location specified by B
07AC      if intp goto .setpintp;    *check for interrupts
07AD  .setpcont:                     *
07AD    a:=a-acc, ip,                *increment location
07AD      if not ctz goto .setplp;   *  UNTIL COUNT IS ZERO
07AE  .setpend:                      *END IT
07AE    b:=p and 0x7fff;             *b:=original b + count
07AF    p:=fa+2;                     *point p to next opcode location
07B0  .setpfetch:                    *
07B0  .cpmeq:                        *
07B0    fchp, rtn;                   *
07B1  .setpintp:                     *INTERRUPT
07B1    b:=p ior 0x8000;             *save this location in B (b15 flags intp)
07B2    a:=ct;                       *save count in A
07B3    fchb, p:=fa, rtn;            *that's all
07B4  .setprst:                      *RESUME AFTER INTERRUPT
07B4    ct:=a, rdb, bbus/b;          *count := A, read new value
07B5    p:=b;                        *place this address into P
07B6    a:=t, goto .setpcont;        *replace next value into A
07B7
07B7  .cpm:                          *COMPARE MEMORY LOCATIONS
07B7    call INDREAD;                *acc:= -2
07B8    s0:=t,                       *save first operand
07B8      rdp, ip;                   *read second def
07B9    call INDREAD, sl:=p-acc;     *sl:=less than skip location
07BA    s2:=s0-t;                    *compare them
07BB    if yz goto .cpmeq;           *equal? y: p is set up
07BC    goto .dcoending;             *n: finish up using .dco routine
07BD
07BD
07BE
```

```
MPARA source listing
0000 MPARA; *virtual memory access  <820204.1550>
0600  $origin 0x600$ *file = &VMA <820204.1550>
0600      *Dedicated registers:
0600      *    PRIN(0)=User map 30.
0600      *    PRIN(1)=Page number of page table and working set offset.
0600  {The VM instructions are divided into two groups. .PMAP maps a logical
0600  page in A acording to a page ID in B. The others map a VM address into
0600  page 30 and 31 returning the appropriate logical address in B.
0600    .LBP has the VM address in A and B.
0600    .LBPR has a DEF to the VM address.
0600    .LPX has an address in AB and a DEF to an address.  The addresses are
0600        added to form the VM address.
0600    .LPXR is like .LPX but uses two DEFs.
0600    .IRES resolves array parameters to form a VM address returned in AB.
0600    .IMAP Effectively .IRES then .LBP.
0600    .JRES Like .IRES but double-integer subscripts (and dimension sizes).
0600    .JMAP Like .JRES then .LBP.
0600  All of these have the "local reference" option.  If the VM address is
0600  negative, the least-significant 16 bits are treated as a local address.
0600  Indirects are resolved and the direct address is returned in B.}
```

```
0600     ******************************************************************
0600     *                     EXECUTION TIMES  (250 nsec. cycle)         *
0600     *                         cycles(usec.)                          *
0600     *                                                                *
0600     *     Normal case        Local          First VMA               *
0600     *     (no fault)         reference      after intr.             *
0600     *                                                                *
0600     *  .LBP  25 (6.25)         7 (1.75)     47 (11.75)               *
0600     *                                                                *
0600     *  .LBPR 28 (7.00)        13 (3.25)     53 (13.25)               *
0600     *                                                                *
0600     *  .LPX  31 (7.75)        17 (4.25)     59 (14.75)               *
0600     *                                                                *
0600     *  .LPXR 37 (9.25)        22 (5.50)     71 (17.75)               *
0600     *                                                                *
0600     *  .IMAP 39 (9.75)+I      24(6.00)+I    58(14.50)+I              *
0600     *                                                                *
0600     *  .JMAP 39 (9.75)+J      24(6.00)+J    58(14.50)+J              *
0600     *                                                                *
0600     *  .IRES 20(5.00)+I           -               -                  *
0600     *                                                                *
0600     *  .JRES 20(5.00)+J           -               -                  *
0600     *                               |14(3.50) with FPP|              *
0600     *     I=[#dimensions]x[12(3.00)+|26(6.50) without |]             *
0600     *                               |  or 46(11.50)   |              *
0600     *                                                                *
0600     *                               |14(3.50) with FPP|              *
0600     *                               |26(6.50) without |              *
0600     *     J=[#dimensions]x[16(4.00)+|  or 46(11.50)   |]+2(.50) if  *
0600     *                               |  or 50(12.50)   |    at least *
0600     *                               |  or 88(22.00)   |    one dim. *
0600     *                                                                *
0600     *        For all instructions above except .IRES and .JRES add  *
0600     *        3(.75) if last page in suit (one case in 1024), and    *
0600     *        add 5(1.25) if last page of VMA.  Add 4(1.00) per      *
0600     *        indirect level in local-reference addresses and 2(.50) *
0600     *        per level of indirect in DEF reads                     *
0600     *----------------------------------------------------------------*
0600     *    .PMAP                                                       *
0600     *           Normal                 20 (5.00)                     *
0600     *                                                                *
0600     *           Sign bit                                             *
0600     *            set in A              24 (6.00)                     *
0600     *                                                                *
0600     *           Last-plus-one                                        *
0600     *            page mapped           23 (5.75)                     *
0600     *                                                                *
0600     *           Normal post                                          *
0600     *            interrupt             37 (9.25)                     *
0600     *                                                                *
0600     *           A-register not                                       *
0600     *            in range [0,31]       13 (3.25)                     *
0600     ******************************************************************
0600                 $DEFINE ADRL/PROCESS_DISPATCH  OXDO$
0600                 $DEFINE ADRL/PRIV_TRAP  OXF3$
0600                 $DEFINE ADRL/DIM OX1B9$
0600
018A                 $ORIGIN OX18A$
018A   VMA_ENTR:       GOTOTBL VMA_BLOC, BBUS/A, STOR/N;      ************
018B                 *Location in jump table corresponding to 105240.
018B                 *Ones go to N. BBUS gets A to test for local reference
018B                 *in .LBP.
018B
018B                 *This block decodes individual VMA instructions. In
018B                 *most cases the DEF is read and next address goes to SO.
018B                 *N is bumped to get zero in it.
0600   VMA_BLOC: $ORIGIN OX600$
0600
0600     PMAP:{00}A:=A-ACC, IN, GOTO S1_PMAP;
0601          {01}                 *.PMAP bumps A early.  This must be
0601          {02}                 * accounted for later.
0604          {03} $ORIGIN OX604$
0604     IRES:{04}RDP, P:=P-ACC, GOTO S1_IRES, CLE;
0605     JRES:{05}RDP, P:=P-ACC, GOTO S1_IRES, STE;
0606          {06}
```

```
0608         {07} $ORIGIN 0X608$
0608    IMAP:{08}RDP, P:=P-ACC, GOTO S1_IMAP, CLE;
060A         {09} $ORIGIN 0X60A$
060A    JMAP:{0A}RDP, P:=P-ACC, GOTO S1_IMAP, STE;
060C         {0B} $ORIGIN 0X60C$
060C    LPXR:{0C}RDP, S0:=P-ACC, GOTO S1_LPXR, IN;
060D     LPX:{0D}RDP, S0:=P-ACC, GOTO S1_LPX,  IN;
060E    LBPR:{0E}RDP, S0:=P-ACC, GOTO S1_LBPR, IN;
060F     LBP:{0F}ACC:=0176000, BBUS/A;
0610             S1:=RR1(B AND ACC),
0610                     IF B15 GOTO LR_LBP;
0611             X:=RR1(NOT ACC AND A);
0612             S0:=P, IN;
0613             S7:=ACC IOR B, LGOTO VMA_MAP2;
0614
0614        *.LBP immediately checks for local reference.  Return value
0614        *of P is saved in S0.  N gets zero; ACC gets mask.  Extrac-
0614        *tion of PAGID begins.  S7 gets value that will become
0614        *physical address returned in B.
0614
0614
0614       LR_LBP: IF B15 CALL RES_INDB;    *Resolve any indirect
0615               NOP:=T,      RTN, FCHP;  *address in B and return.
0616     RES_INDB: NOP:=B, RDB;             *Read address in B. Load
0617               B:=T, LGOTO CONT_RIB;    *to B and jump to "continue_
0618                                        *resolve_indirect_in_B."
0618
0618
0618  S1_IRES:  LGOTO S2_IRES, S6:=ZERO;  *Go to "segment_two."  N gets
0619  S1_IMAP:  LGOTO S2_IMAP, N:=ZERO;   *zero in IMAP because we need
061A                                      *to get PRIN(0) to test for
061A                                      *initialization of micro regs.
061A                                      *S6 and S7 will get resolved VM
061A                                      *address; they must be set to 0
061A                                      *before entering resolve loop.
061A                                      *Note that these also link JRES
061A                                      *and JMAP. E is used to flag
061A                                      *I vs. J.
061A
061A  S1_PMAP:  SRIN:=PRIN-ACC, IN, IF B15 GOTO PNEG;
061B            IF YZ GOTO N_INITP1, CLE;
061C            MAP:=PRIN, LGOTO S2_PMAP;
061D                                      *MPAR gets user data map
061D                                      *register 31 to prepare for
061D                                      *PTE read. N increments to 0.
061D                                      *If A had sign bit set on entry
061D                                      *we branch to "PMAP_negative"
061D                                      *code which handles the case
061D                                      *when faults must cause
061D                                      *a P+1 exit. This lets $VMA$
061D                                      *call PMAP without the poss-
061D                                      *ibility of recursion.
061D                                      *If PRIN(0):=ONES we branch to
061D                                      *code that restores A then sets
061D                                      *up PRINs. E is cleared since E
061D                                      *set on return indicates LAST+1.
061D                                      *MAP gets page where PTE is.
061D     PNEG:  LGOTO S_PNEG;
061E  N_INITP1: A:=A+ACC, LGOTO NOT_INIT;
061F
061F  S1_LBPR:  ACC:=0176000;                 *ACC gets mask 6-1s-10-0s.
0620            P:=T, RDB;                    *P gets value of the DEF.
0621            IF B15 CALL RES_INDR, IP;     *Indirects resolved and P
0622            X:=RR1(NOT ACC AND T), RDP;   *bumped to read second word
0623            ********                      *of VM addr. Extraction of
0623            IF B15 GOTO LR_LBPR,          *PAGID begins. LOCAL_REF if
0623                   S1:=RR1(ACC AND T);    *sign of MSW is set. Start
0624            S7:=ACC IOR T,                *generation of log addr in
0624                   LGOTO VMA_MAP2;        *S7(i.e. 111111aaaaaaaaaa).
0625
0625                                          *LOCAL_REF for LBPR. P is
0625     LR_LBPR: B:=T, IF B15 CALL RES_INDB; *restored from S0.
0626                   P:=S0, LGOTO FIN_LR;
0627                                          *RESOLVE_INDIRECT resolves
0627    RES_INDR: P:=T, RDB;                  *indirect addr in P. P is
```

```
0628                IF NOT B15 THEN RTN, IP;    *bumped to read possible
0629                P:=T, RDB;                  *double-word number.
062A    RES_LOOP:   IF NOT B15 THEN RTN, IP;
062B                P:=T, RDB,
062B                  IF NOT INTP GOTO RES_LOOP;
062C         PRC_DSP1: LGOTO PROCESS_DISPATCH, P:=FA;
062D
062D
062D  LOC_REF:     B:=S7, IF B15 CALL RES_INDB; *LOCAL_REF other routines.
062E               P:=S0, LGOTO FIN_LR;         *S7 holds LSW of VM addr.
062F
062F  S1_LPXR:     P:=T-ACC, RDB;               *DEF comes back for MSW
0630               IF B15 CALL RES_INDR;        *of VM base address.
0631               S6:=T, SP0/RDP;              *S6 gets MSW. Read begins
0632               P:=S0, LGOTO S2_LPXR;        *LSW. P gets addr of 2nd
0633                                            *DEF.
0633   S1_LPX:     LGOTO S2_LPX;                *Branch while wait for DEF.
0634
0634                                            *LSW of VM base addr goes to
0634  S2_LPXR:     S7:=T, SP2/RDP;              *S7. Read DEF for VM offset
0635               *********                    *addr. Start read of MSW of
0635               P:=T-ACC, RDB;               *offset. P gets addr of LSW.
0636               IF B15 CALL RES_INDR,        *Resolve indrs. S0 is inc'ed
0636                       S0:=S0-ACC;          *to point to instr after
0637               S6:=S6+T, SP0/RDP;           *LPXR. Offset added to S6,S7.
0638               ACC:=0176000;                *ACC gets mask.
0639               S7:=S7+T, IN;                *N to 1, then 0 for com-
063A               S6:=S6+CF;                   *patibility with IMAP.
063B
063B                                            *PAGID extracted by following
063B                                            *operations. A ten-bit right
063B                                            *shift of double word required
063B
063B                                            *Each word masked and shifted
063B                                            *once. LOCAL_REF test is also
063B  VMA_MAP1:    ABUS/ACC, BBUS/S6, STOR/X,   *done here. Words merged and
063B                   ALU/CAND, SP0/RR1;       *shifted, then byte-swap com-
063C               {X:=RR1(NOT ACC AND S6);}    *pletes shifting.
063C
063C               S1:=RR1(ACC AND S7),
063C                  IF B15 GOTO LOC_REF;
063D
063D
063D               S7:=ACC IOR S7, DN;   **** *This is the first step in
063E  VMA_MAP2:    X:=RR1(X IOR S1);          *calculating the addr returned
063F               X:=SWAP(X);                *in B.
0640
0640               SRIN:=PRIN+ONE;            *N=0 here. MPAR = user map 31.
0641               IF YZ GOTO NOT_INIT,       *If PRIN(0)=ONES registers are
0641                    P:=ACC IOR X, IN;     *not initialized. P gets log
0642               MAP:=PRIN, CLO, DN;        *address in pg 31 for PTE read.
0643                                          *Map register 31 gets page# of
0643                                          *PTE.
0643
0643               SRIN:=PRIN, SP0/RDP, IP;   *MPAR gets user map 30. First
0644               BBUS/P, S4:=075777;        *PTE read begins. P tested to
0645                                          *see if incr results in oflo.
0645                                          *Mask for addr returned in B
0645                                          *goes to S4.
0645               IF NOT B15 GOTO OVER,
0645                  S3:=NOT ACC AND T, IN;  *Mask off segment #. Set 0 if
0646               IF YZ THEN STO,            *Last+1 or null flag. Load map
0646                  MAP:=S3+PRIN;           *register 30 with base(PTE pg)
0647                                          *plus offset.
0647   {At this point mapping of page 30 has occurred. Even though we may
0647    fault on this page, we are OK because the page will be mapped
0647    again when the microcode is restarted.}
0647
0647               S1:=X XOR T, SP0/RDP;      *Do segment #'s match? Start
0648                                          *read of second PTE entry.
0648
0648  FIN_MAP1:    IF 0 GOTO SP_CASE1,        *If 0 set(indicating 0's in
0648                  NOP:=ACC AND S1, CLE;   *lower bits of PTE entry),
0649                                          *branch to special_case code.
0649                                          *Complete test of segment #s.
```

```
0649                                         *E is used as a flag to show
0649                                         *which page caused "problem."
0649    {E is undefined after VMA call other than .PMAP. This code returns
0649     with E clear. Coding restrictions forced me to use E as a flag.}
0649
0649  FIN_MAP2:  IF NOT YZ GOTO PG_FLT1,     *Fault on no match. Mask off
0649                ACC:=NOT ACC AND T;      *segment # of 2nd PTE entry.
064A
064A             IF YZ GOTO SP_CASE2,        *Last+1 or null again. Load
064A                MAP:=ACC+PRIN, STE;      *map 31. Again use E as flag.
064B
064B  FIN_MAP3:  S3:=X XOR T, CLO;           *Segment #s match? Clear O
064C                NOP:=S3 AND 0176000;     *for return.
064D             IF NOT YZ GOTO PG_FLT2,
064D                P:=S0;                   *Restore program counter for
064E             B:=S7 AND S4, CLE,          *return. B gets logical addr.
064E                      FCHP, RTN;         *E is returned clear. Fetch
064F                                         *next instruction.
064F
064F        {The following code handles various "anomalies."}
064F
064F                                         *OVER: handles case when 1st
064F                                         *entry read is last entry of
064F                                         *PTE table.
064F
064F      OVER:  IF YZ THEN STO,             *Last+1 or null. Load map 30.
064F                MAP:=S3+PRIN;            *Save second PTE entry. Set F
0650             S3:=T, STF;                 *so that SP_CASE1 and PG_FLT2
0651                                         *know we have been here.
0651             P:=ACC;                     *Set P to read first PTE
0652                                         *location. (1111110000000000)
0652               S1:=X XOR T, SPO/RDP;     *Segment #s match? Read 2nd
0653               X:=X+ONE, GOTO FIN_MAP1;  *entry. Bump PAGID.
0654
0654                                         *This code handles "LAST+1"
0654                                         *page of VMA and null entry
0654                                         *faults.
0654  SP_CASE1:  IF NOT F THEN STOR,
0654                      S3:=S1 XOR X;      *Determine what the PTE entry
0655  SP_CASE2:  IF E THEN STOR, S3:=T;      *was that resulted in anomaly.
0656             NOP:=S3 INOR 01777;         *A null flag perhaps?
0657             IF YZ GOTO NULL_FLT,
0657                      N:=ZERO;
0658             SRIN:=SRIN-1;               *Must be L+1. Restore MPAR.
0659             MAP:=ACC XNOR ACC, IN,      *Map register is R/W protected.
0659                 IF E GOTO FIN_MAP3;     *Return to VMA_MAP. Fault is
065A             NOP:=ACC AND S1,            *still possible on this page.
065A                 GOTO FIN_MAP2;
065B
065B                                         *The following code handles
065B                                         *the various kinds of faults.
065B                                         *The San Andreas not included.
065B
065B
065B     NF_P1:  NOP:=P AND 01777;           *If OVER incremented PAGID,
065C             IF YZ THEN STOR,            *put it back the way it was!
065C                 X:=X+ACC;
065D             GOTO PG_FLT1;
065E  NULL_FLT:  IF NOT E GOTO NF_P1,        *Fault due to null entry.
065E                 ACC:=ONES;
065F
065F   PG_FLT2:
065F             IF NOT F THEN FCIN, X:=X;   *Bump PAGID if OVER didn't.
0660   PG_FLT1:  Y:=074000;                  *Set Y to logical addr of 1st
0661             N:=ZERO, CLF, CLO;          *PTE entry. $VMA$ manipulates
0662                                         *PTE thru map 30 if non-PMAP
0662                                         *fault on first page. F is
0662                                         *cleared to avoid P+1 return.
0662                                         *O clear cause fault not PMAP.
0662             IF E THEN FCIN, SRIN:=PRIN;*MPAR=USER_MAP_30+E
0663             IF NOT E GOTO FAULT, IN;
0664      PFLT:  Y:=076000;                  *Change Y if PMAP or 2nd page.
0665
0665     FAULT:  P:=4;                       *Where is $VMA$? Addr is at 4.
0666             MAP:=PRIN;                  *Load map reg with page of
```

```
0667            SPO/RDP, ACC:=ONES;          *PTE.  Read 4 and put addr in
0668            P:=T, CLE, IF F GOTO PNF;    *P.  Store fetch address for
0669            WRP:=FA, IP,                 *restart at 1st loc of $VMA$
0669                IF YZ GOTO DM_ABORT;     *and abort if addr given is 0.
066A                                         *Do not store if PMAP P+1.
066A            SPO/RDP;                     *Read 1st instr of $VMA$,
066B            NOP:=T XOR 0104400;          *must be DST, abort if not.
066C            IF NOT YZ GOTO DM_ABORT,
066C                ACC:=ONES;               *F is set if PMAP P+1 exit
066D      PNF:  IF F THEN STOR, P:=FA-ACC;   *required. Fetch next
066E              FCHP, RTN;                 *instruction.
066F
066F
066F                                         *The following code initializes
066F                                         *internal VMA registers after
066F                                         *an interrupt.
066F
066F  NOT_INIT:                              *Save present MEMR.
066F            S1:=MEMR, LGOTO LINK2;
0670
0670      {LINK2: S2:=MEMR and 037;          *Get map set #.
0670               MEMR:=ZERO, LGOTO LINK1;} *Set map set 0.
0670
0670     LINK1: P:=2;                        *Read location 2 of system map.
0671            SPO/RDP, P:=FA;              *This is the page # of PTE.
0672            S2:=RL4(S2), IP;             *Set P to fetch addr plus one.
0673            S2:=LL1(S2);                 *We are going to "restart" the
0674                                         *instruction. Map set # is left
0674                                         *shifted five so that we can
0674                                         *calculate what user map 30 is.
0674
0674            PRIN:=T and 077777;          *Mask sign bit off PTE page #.
0675            DN;                          *N goes to 0. Cond flags stay.
0676            IF B15 GOTO VMN_INIT,        *If PTE page had sign set, VM
0676                          MEMR:=S1;      *system is not initialized. We
0677                                         *must do an "uninitialized
0677                                         *fault."  Also restore MEMR.
0677
0677            PRIN:=S2 IOR 036;            *PRIN(0) gets user map 30.
0678            ACC:=ONES, LGOTO VMA_ENTR;   *Restart instruction.
0679
0679
0679  VMN_INIT: PRIN:=S2 IOR 036;            *PRIN(0) gets user map 30.
067A            SRIN:=PRIN+ONE;              *MPAR gets user map 31.
067B            Y:=0176000;                  *Sign bit set in Y tells $VMA$
067C            GOTO FAULT, IN;              *that this is an "uninitialized
067D                                         *fault."  N goes to one.
067D
067D  DM_ABORT: P:=FA, LGOTO PRIV_TRAP;
067E
067E
067E                                         *.PMAP code follows.
067E
067E
067E   S2_PMAP: P:=B IOR 0176000;            *P gets logical addr for PTE
067F            NOP:=A+0177737;              *read.  Is logical pg to be
0680  CON_PMAP: IF CF GOTO REG_ERR,          *mapped > 31.  If so ERROR!
0680                   RDP, N:=ZERO;         *Read PTE. N gets zero.
0681            SRIN:=PRIN-ACC;              *MPAR gets user map 31.
0682            S3:=B XOR T, CLE;            *Segment #s match?  E is
0683            S2:=T AND 01777;             *returned clear unless L+1.
0684            IF YZ GOTO SP_PMAP,          *Null entry or LAST+1. R/W
0684                   MAP:=ACC;             *prot 31 so user can't trash
0685            S4:=PRIN-31;                 *page table. Calculate abs
0686            SRIN:=A+S4, IN;              *page # given logical page #.
0687                                         *Recall A has been bumped! N=1
0687            NOP:=S3 AND 0176000;         *Segment #s match? Set P to
0688            IF NOT YZ GOTO PMAP_FLT,     *Fetch address plus two.
0688                   P:=FA-ACC;            *Map page.
0689            MAP:=S2+PRIN, IP;            *Bump B.
068A  END_PMAP: B:=B-ACC, FCHP, RTN;         *Fetch next instruction.
068B
068B
068B                                         *PMAP anomalies follow.
068B
```

```
068B
068B    SP_PMAP: NOP:=T INOR 01777;        *Null entry?
068C             IF YZ GOTO PMAP_FLT;
068D             NOP:=S3 AND 0176000;      *Segment #s match?
068E             IF NOT YZ GOTO PMAP_FLT;
068F             S4:=PRIN-31;              *Calculate abs page #. Set
0690             SRIN:=A+S4, STE;          *E to indicate LAST+1.  Set
0691             P:=FA+2;                  *program counter for return.
0692             MAP:=ACC, LGOTO END_PMAP; *R/W protect LAST+1 page.
0693                                       *Restore A.
0693   PMAP_FLT: A:=A+ACC, STO;            *Set 0 so $VMA$ knows
0694             N:=ZERO;                  *PMAP faulted. N gets zero.
0695             SRIN:=PRIN-ACC, IN;       *MPAR = user map 31. N gets 1.
0696             X:=B, LGOTO PFLT;         *X gets PAGID that caused flt.
0697
0697    REG_ERR: A:=80;                    *A gets error code 80 decimal.
0698             P:=FA+ONE;                *Error return at P+1.
0699             FCHP, RTN;                *Fetch next instruction.
069A
069A     S_PNEG: IF YZ GOTO N_INITP2,      *Restore A.
069A                A:=A+ACC;
069B             MAP:=PRIN, CLE, STF;      *Load map reg for PTE read.
069C                                       *Clear E for normal return.
069C                                       *Set F so fault will cause P+1
069C                                       *return.
069C             S1:=A AND 077777;         *Strip sign bit.
069D             A:=A-ACC;                 *Bump A again.
069E             P:=B IOR 0176000;         *P gets log addr for PTE read.
069F             NOP:=S1+0177740;          *Logical page # > 31?
06A0             LGOTO CON_PMAP;
06A1
06A1   N_INITP2: LGOTO NOT_INIT;
06A2
06A2
06A2
06A2
06A2  CONT_RIB:  IF NOT B15 THEN RTN;      *Resolve indirect addresses
06A3             NOP:=B, RDB;              *and store direct addr in B.
06A4             B:=T;
06A5   RIB_LOOP: IF NOT B15 THEN RTN;
06A6             NOP:=B, RDB;
06A7             B:=T, IF NOT INTP
06A7                   GOTO RIB_LOOP;      *Interrupt causes restart.
06A8
06A8             P:=FA, LGOTO PROCESS_DISPATCH;
06A9
06A9
06A9
06A9
06A9    S2_LPX:  P:=T, RDB;                *DEF comes back. Resolve
06AA             IF B15 CALL RES_IND8, IP; *if indr. Load P and bump
06AB             S6:=A+T, SPO/RDP;         *to read 2nd word. Add value
06AC             ACC:=0176000;             *to A,B; store to S6,S7. ACC
06AD             S7:=B+T;                  *gets mask.
06AE             S6:=S6+CF;
06AF             ABUS/ACC, BBUS/S6, STOR/X, *See VMA_MAP1. Must copy code
06AF                 ALU/CAND, SPO/RR1;    *cause long branch required.
06B0             S1:=RR1(ACC AND S7),
06B0                 IF B15 GOTO LR_LPX;
06B1             S7:=ACC IOR S7, LGOTO VMA_MAP2;
06B2
06B2       LR_LPX: LGOTO LOC_REF;
06B3    PRC_DSP2: P:=FA, LGOTO PROCESS_DISPATCH;
06B4    RES_IND8: LGOTO RES_INDR;
06B5
06B5       FIN_LR: NOP:=T, FCHP, RTN;      *Finishes local reference.
06B6                                       *NOP:=T makes sure read
06B6        LINK2: S2:=MEMR AND 037;       *is finished.
06B7               MEMR:=ZERO, LGOTO LINK1;
06B8                                       *See LINK1.
06B8
06B8
06B8
06B8        {Begin array address resolving code.}
```

```
06B8
06B8
06BF               $ORIGIN 0X6BF$          *X gets the addr of the 2nd
06BF  S2_IRES:  X:=T-ACC, RDB, CALL RESOLV;*elem in the array parameter
06C0                                       *table.  Call resolve subr.
06C0
06C0            P:=S0;                     *Save return point in S0.
06C1            A:=S6, FCHP;               *VMA address to S6,S7.
06C2            B:=S7, RTN;                *Fetch next instruction.
06C3
06C3
06C3                                       *The code below has to test for
06C3                                       *uninitialized case before
06C3                                       *resolving the address. .IRES
06C3                                       *and .JRES do not test this at
06C3                                       *all.  This results in different
06C3                                       *entry points to RESOLV.
06C3
06C3  S2_IMAP:  X:=T-ACC, RDB;                 *As above followed by indr
06C4            IF B15 CALL RES_INDX,          *resolver. S6 gets zero.
06C4              S6:=ACC+ONE, BBUS/PRIN;      *Test PRIN(0).  Clear S7.
06C5            IF B15 GOTO INI, S7:=ZERO, IN; *N goes to one.
06C6            Y:= -T, SP2/RDP, CALL MAP_RES; *Y gets -# of dimensions.
06C7            LGOTO VMA_MAP1;                *Read 1st subscript.
06C8
06C8       INI: LGOTO NOT_INIT;
06C9
06C9   RESOLV:  IF B15 CALL RES_INDX, IN, S7:=ZERO; *Resolve indrs loading
06CA            Y:= -T, SP2/RDP;                    *addr to X. Y gets the
06CB                                                *negated loop count.
06CB  MAP_RES:  IF YZ GOTO END_RES, ACC:=ONES, IP;  *If loop count 0, ter-
06CC    RLOOP:  S2:=T-ACC, RDB;                     *minate routine. Indrs
06CD                                                *are resolved for subs
06CD            IF B15 CALL RES_IND2, CT:=LL1(ACC); *DEF. CT gets +1.
06CE            IF NOT E GOTO NOT_J, CT:=NOT CT;
06CF            S6:=S6+T, DCT;                 *CT goes to 0 for JRES/
06D0            RDB, BBUS/S2, IF INTP          * JMAP.  Read 2nd word
06D0                       GOTO PRC_DSP6;      *of subscript in Jxxx.
06D1    NOT_J:  S1:=S7+T;                      *Add word rtned to low
06D2            S0:=S6+CF, BBUS/T;             *word of VM addr. Prop
06D3            IF B15 THEN STOR, S0:=S0-CT;   *carry & sign extend if
06D4            X:=X-ACC, RDB, IF NOT E GOTO I;  *Ixxx. Read next dim size.
06D5            NOP:=Y+ONE;                    *Last loop? The this is #
06D6            S2:=T, IF YZ GOTO I;           *words/elem.  Treat like
06D7            X:=X-ACC, RDB;                 *Ixxx.  Read 2nd word of
06D8            S3:=T, LCALL DIM;              *dim size and call mult.
06D9            Y:=Y+ONE, SP2/RDP, GOTO THERE; *Start read of next subscr
06DA                                           *DEF.  Last loop?
06DA         I: S3:=T, IF INTP GOTO PRC_DSP6;  *Set up Ixxx multiply.
06DB            S2:=ZERO, IF YZ GOTO ENTZ;     *Ixxx treats dim size of
06DC            LCALL DIM;                     *zero as 2**16. Sp-case
06DD      HERE: Y:=Y+ONE, SP2/RDP, GOTO THERE; *this.  Last loop? Start
06DE      ENTZ: S6:=S1;                        *read of next DEF.
06DF            S7:=ZERO, LGOTO HERE;
06E0
06E0     THERE: IF NOT YZ GOTO RLOOP, ACC:=ONES, IP;
06E1
06E1                                      *Finish array addr reso-
06E1                                      *lution. Must add array
06E1                                      *offset.
06E1  END_RES:  BBUS/T, ACC:=0176000;     *Finish pseudoDEF read.
06E2            S0:=P-1;                  *Mask to ACC. Return addr to
06E3            P:=X, RDB;                *S0. Read 1st word of off-
06E4            IP, N:=ZERO;              *set. N will soon be 1.
06E5            S6:=S6+T, SP0/RDP;        *Read 2nd word.
06E6            S7:=S7+T, IN;
06E7            S6:=S6+CF, RTN;           *Propagate carry.
06E8
06E8
06E8
06E8                                      *INDIRECT ADDRESS RESOLVERS.
06E8
06E8   RES_INDX: X:=T-ACC, RDB;           *Table pointer will be in
06E9             IF NOT B15 THEN RTN,     *X.  PRIN must be on BBUS
06E9                NOP:=ACC, BBUS/PRIN;  *to test for NOT_INIT. ALU
```

```
06EA              X:=T-ACC, RDB;              *must not be zero so status
06EB    RSX_LOOP: IF NOT B15 THEN RTN,        *will be updated.
06EB                 NOP:=ACC, BBUS/PRIN;
06EC              X:=T-ACC, RDB,
06EC                 IF NOT INTP GOTO RSX_LOOP;
06ED              P:=FA, LGOTO PROCESS_DISPATCH;
06EE
06EE    RES_IND2: S2:=T-ACC, RDB;             *S2 gets bumped address for
06EF              IF NOT B15 THEN RTN;        *read of possible 2nd word.
06F0              S2:=T-ACC, RDB;
06F1    RS2_LOOP: IF NOT B15 THEN RTN;
06F2              S2:=T-ACC, RDB,
06F2                 IF NOT INTP GOTO RS2_LOOP;
06F3    PRC_DSP6: LGOTO PROCESS_DISPATCH, P:=FA;
06F4
06F5
```

```
MPARA source listing
0000 MPARA; *operating system set <820204.1550>
05E0  $origin 0x5E0$ *file = &OSS <820204.1550>
05E0  ********************************************************************
05E0  * (C) Copyright Hewlett Packard Company 1982. All rights reserved.  *
05E0  * No part of this program may be photocopied, reproduced or         *
05E0  * translated to another program language without the prior written  *
05E0  * consent of Hewlett Packard Company.                                *
05E0  ********************************************************************
05E0
05E0  *820107 SRK updated FWID to REV 2
05E0  *820114 SRK fixed .SIP (was pure NOP, now skip if I/O interrupt)
05E0  *820119 SRK updated FWID to REV 3
05E0  *820128 SRK updated FWID to REV 4
05E0
05E0  $define adrl/INDRDBL 0x735$
05E0  $define dat/ST_FWID  0x0200$
05E0
018C  $origin 0x18C$ goto os_branch, n:=ones; *opcodes 105300 - 105317
018D
05E0  os_table: $origin 0x5e0$        *opcodes 105300 - 105317
05E0  goto  $origin 0x5e0$            *CPU IDENTIFICATION INSTRUCTION
05E0    .cpuid, acc:=111(acc+acc);    *  (loads A=3 for A700)
05E1  goto  $origin 0x5e1$            *FIRMWARE IDENTIFICATION INSTRUCTION
05E1    .fwid,                        *  (branches to the last two words of each
05E1    ct:=b,                        *    1K module noted by B.
05E1    dn;                           *  set n to N_ST for microcode status
05E2  goto  $origin 0x5e2$            *WAIT FOR INTERRUPT INSTRUCTION
05E2    .wfi;                         *
05E3        $origin 0x5e3$            *SKIP IF I/O INTERRUPT PENDING INSTRUCTION
05E3  .sip:                           *
05E3    ist:=ist xor 0x0100;          *unmask I/O interrupts (valid 2 cycles later)
05E4    ist:=ist xor 0x0100;          *remask I/O interrupts
05E5    if intp then ip;              *skip if I/O interrupt
05E6    if intp goto refetch;         *refetch if any other interrupt pending
05E7    fchp, rtn;                    *that's all
05E8
05E8  os_branch:                      *assure that only four words of jumptable
05E8    ct:=ct and 0x0003;            *  are used
05E9    goto os_table, ct30;          *
05EA
05EA  .cpuid:                         *A700 identification code is 3
05EA    fchp, a:=not acc, rtn;        *
05EB
05EB  .fwid:                          *branch to code that identifies each 1k
05EB    acc:=ST_FWID;                 *set special bit to indication that JNM
05EC    call .fwidtable, ct30,        *  may be caused by firmware checkout
05EC      prin:=prin ior acc;         *
05ED    prin:=prin and not acc,       *identification complete, clear the bit
05ED      fchp, rtn;                  *
05EE
05EE  .wfi:                           *
05EE    if not intp goto .wfi;        *wait for interrupt
05EF  refetch:                        *
05EF    fchb, p:=fa, rtn;             *
05F0
```

```
05F0  .fwidtable: $origin 0x5F0$
05F0  {0} goto 0x03FE;                *0k  This code causes a branch to a 1k bank
05F1  {1} goto 0x07FE;                *1k  specified by the contents of the
05F2  {2} goto 0x0BFE;                *2k  B-register.  If firmware is installed,
05F3  {3} goto 0x0FFE;                *3k  it should contain two lines of microcode
05F4  {4} goto 0x13FE;                *4k  to load the revision and package
05F5  {5} goto 0x17FE;                *5k  information into the A-register and rtn.
05F6  {6} goto 0x1BFE;                *6k  If the firmware is not installed, the
05F7  {7} goto 0x1FFE;                *7k  jump-to-nonexistent micromemory code
05F8  {8} goto 0x23FE;                *8k  in &CONTR will set the A-register
05F9  {9} goto 0x27FE;                *9k  to 0xFFFF and fetch the next instruction.
05FA  {A} goto 0x2BFE;                *10k Thus, special selftest opcodes are not
05FB  {B} goto 0x2FFE;                *11k required for checking firmware
05FC  {C} goto 0x33FE;                *12k installation.
05FD  {D} goto 0x37FE;                *13k
05FE  {E} goto 0x3BFE;                *14k
05FF  {F} goto 0x3FFE;                *15k
0600
03FE  $origin 0x3FE$
03FE    a:=0x0401;                    *low byte = firmware package (1=base set)
03FF    rtn;                          *hi  byte = revision         (revision 4)
07FE  $origin 0x7FE$
07FE    a:=0x0401;                    *low byte = firmware package (1=base set)
07FF    rtn;                          *hi  byte = revision         (revision 4)
0800
0801
```

```
MPARA source listing
0000 MPARA; *USER and HP reserved opcodes <820204.1550>
0180  $origin 0x180$ *file = &USER <820204.1550>
0180  ******************************************************************
0180  * (C) Copyright Hewlett Packard Company 1982. All rights reserved.  *
0180  * No part of this program may be photocopied, reproduced or        *
0180  * translated to another program language without the prior written  *
0180  * consent of Hewlett Packard Company.                              *
0180  ******************************************************************
0180
0180  $origin 0x180$ *VIS            *opcodes 105000 - 105017 HP reserved
0181  $origin 0x181$ *VIS            *opcodes 105020 - 105037 HP reserved
0182  $origin 0x182$ *VIS            *opcodes 105040 - 105057 HP reserved
0183  $origin 0x183$ *VIS            *opcodes 105060 - 105077 HP reserved
0184  $origin 0x184$ *reserved       *opcodes 105100 - 105117 HP reserved
0185  $origin 0x185$ *reserved       *opcodes 105120 - 105137 HP reserved
0186  $origin 0x186$  gototbl 0x0900, *opcodes 105140 - 105157 HP reserved
0186                  stor/n,bbus/cab;
0187  $origin 0x187$  gototbl 0x0B00, *opcodes 105160 - 105177 HP reserved
0187                  stor/n,bbus/cab;
0188  $origin 0x188$ *BASE SET: DIS  *opcodes 105200 - 105217 HP reserved
0189  $origin 0x189$ *BASE SET: LIS  *opcodes 105220 - 105237 HP reserved
018A  $origin 0x18A$ *BASE SET: VMA  *opcodes 105240 - 105257 HP reserved
018B  $origin 0x18B$  gototbl 0x2800, *opcodes 105260 - 105277 HP reserved
018B                  stor/n,bbus/cab;
018C  $origin 0x18C$ *BASE SET: OSS  *opcodes 105300 - 105317 HP reserved
018D  $origin 0x18D$ *SIS            *opcodes 105320 - 105337 HP reserved
018E  $origin 0x18E$  gototbl 0x0D00, *opcodes 105340 - 105357 HP reserved
018E                  stor/n,bbus/cab;
018F  $origin 0x18F$  gototbl 0x0F00, *opcodes 105360 - 105377 HP reserved
018F                  stor/n,bbus/cab;
0190
01AE  $origin 0x1ae$  gototbl 0x0800, *opcodes 10(x01)400 - 10(x01)417 HP
01AE                  stor/n,bbus/cab;
01AF  $origin 0x1af$  gototbl 0x0A00, *opcodes 10(x01)420 - 10(x01)437 HP
01AF                  stor/n,bbus/cab;
01AC  $origin 0x1ac$  gototbl 0x0C00, *opcodes 10(x01)440 - 10(x01)457 HP
01AC                  stor/n,bbus/cab;
01AD  $origin 0x1ad$  gototbl 0x0E00, *opcodes 10(x01)460 - 10(x01)477 HP
01AD                  stor/n,bbus/cab;
01AA  $origin 0x1aa$  gototbl 0x3000, *opcodes 10(x01)500 - 10(x01)517 user
01AA                  stor/n,bbus/cab;
01AB  $origin 0x1ab$  gototbl 0x3200, *opcodes 10(x01)520 - 10(x01)537 user
01AB                  stor/n,bbus/cab;
01A8  $origin 0x1a8$  gototbl 0x3400, *opcodes 10(x01)540 - 10(x01)557 user
01A8                  stor/n,bbus/cab;
```

```
01A9  $origin 0x1a9$  gototbl 0x3600, *opcodes 10(x01)560 - 10(x01)577 user
01A9                  stor/n,bbus/cab;
01A6  $origin 0x1a6$  gototbl 0x2000, *opcodes 10(x01)600 - 10(x01)617 HP/user
01A6                  stor/n,bbus/cab;
01A7  $origin 0x1a7$  gototbl 0x2200, *opcodes 10(x01)620 - 10(x01)637 HP/user
01A7                  stor/n,bbus/cab;
01A4  $origin 0x1a4$  gototbl 0x2400, *opcodes 10(x01)640 - 10(x01)657 HP/user
01A4                  stor/n,bbus/cab;
01A5  $origin 0x1a5$  gototbl 0x2600, *opcodes 10(x01)660 - 10(x01)677 HP/user
01A5                  stor/n,bbus/cab;
01A2  $origin 0x1a2$ *BASE SET: DMS   *opcodes 10(x01)700 - 10(x01)717 HP
01A3  $origin 0x1a3$ *BASE SET: DMS   *opcodes 10(x01)720 - 10(x01)737 HP
01A0  $origin 0x1a0$ *BASE SET: EIG   *opcodes 10(x01)740 - 10(x01)757 HP
01A1  $origin 0x1a1$ *BASE SET: EIG   *opcodes 10(x01)760 - 10(x01)777 HP
01A1
01A1  *include opcodes forced out for DIS,FPSG,FPDG
0196  $origin 0x196$  gototbl 0x0900, *opcodes 105140 - 105157.HP reserved
0196                  stor/n,bbus/cab;
0197  $origin 0x197$  gototbl 0x0B00, *opcodes 105160 - 105177 HP reserved
0197                  stor/n,bbus/cab;
019E  $origin 0x19E$  gototbl 0x0900, *opcodes 105140 - 105157 HP reserved
019E                  stor/n,bbus/cab;
019F  $origin 0x19F$  gototbl 0x0B00, *opcodes 105160 - 105177 HP reserved
019F                  stor/n,bbus/cab;
01B6  $origin 0x1B6$  gototbl 0x0900, *opcodes 105140 - 105157 HP reserved
01B6                  stor/n,bbus/cab;
01B7  $origin 0x1B7$  gototbl 0x0B00, *opcodes 105160 - 105177 HP reserved
01B7                  stor/n,bbus/cab;
01B8
01B9
```

```
MPARA source listing
0000 MPARA;  *microcoded selftest and boot routine <820204.1550>
0007  $origin 0x007$ *file = &SELFT <820204.1550>
0007  ***********************************************************************
0007  * (C) Copyright Hewlett Packard Company 1982. All rights reserved.  *
0007  * No part of this program may be photocopied, reproduced or         *
0007  * translated to another program language without the prior written  *
0007  * consent of Hewlett Packard Company.                                *
0007  ***********************************************************************
0007  $define adrl/RESET_PU 0x2F2$
0007  $define adrl/START_UP 0x0CF$
0007  $define adrl/FAILTRAP 0x004$
0007  $define adrl/TEST_RESTART 0x001$
0007
0007  SELFTEST: $origin 0x007$      *link from &CONTR module
0007
0007  PL_DIAGNOSTIC:                *begin lower processor diagnostic
0007
0007  pl_branch:
0007  {
0007    Begin the microcoded selftest
0007    Assure that YZ condition and conditional branches work
0007  }
0007    if not yz goto FAILTRAP,    *YZ condition: test true, no jump
0007      r00:=not acc and acc;     *r00:=zeros
0008    if yz goto pl_b_false,      *              test true, jump
0008      r01:=acc xnor acc;        *
0009  pl_failure:                   *
0009    lgoto FAILTRAP;             *no brains at all! (put brakes on)
000A  pl_b_false:                   *YZ condition set to FALSE
000A    if yz goto FAILTRAP,        *YZ condition: test false, no jump
000A      r02:=r00 xnor acc;        *
000B    if not yz goto              *              test false, jump
000B      pl_regfile,               *
000B      ct:=zeros;                *CT:=zeros for later test.
000C    lgoto FAILTRAP;             *only half crazy!
000D
000D  pl_regfile:
000D  {
000D    Load the register file with unique immediate data.
000D    new microorders used:
000D      - op/imm
000D      - alu/adac;
000D      - stor/r00 - stor/r17
000D  }
000D    r00:=0x0001;              *
000E    r01:=0x0002;              *
000F    r02:=0x0004;              *
0010    r03:=0x0008;              *
0011    r04:=0x0010;              *
0012    r05:=0x0020;              *
0013    r06:=0x0040;              *
0014    r07:=0x0080;              *
0015    r10:=0x0100;              *
0016    r11:=0x0200;              *
0017    r12:=0x0400;              *
0018    r13:=0x0800;              *
0019    r14:=0x1000;              *
001A    r15:=0x2000;              *
001B    r16:=0x4000;              *
001C    r17:=0x8000;              *
001D
001D  {
001D    Do an exclusive or of all of the register file values, in addition
001D      to random checks.  The exclusive or should come up all ones if
001D      the register file checks and the random checks did not interfere.
001D    new microorders used:
001D      - abus or bbus r00-r17
001D      - bbus/q
001D      - stor/q and r00-r17
001D      - sp2/stf,clf,
001D      - sp0/cle,ste,sto,clo,stor, ldq
001D      - alu/xor,ior,addc
001D      - cndx/yz,y15,o,e,f,b15
001D  }
001D    r10:=r10 ior r11,         *r10:=0300
001D      cle,                    *cle (pretest should further test sto in asgs)
001D      stf;                    *stf
001E    acc:=acc xor r05,         *acc:=0030
001E      if e goto fail1,        *check e value
001E      ldq;                    *load q
001F    r17:=r16 xor r17,         *r17:=c000
001F      if not e then ste;      *try sp0t--should ste
0020    acc:=acc+r10,             *acc:=0330
0020      if not e goto fail1,    *check e--should be set
0020      cle;                    *cle
0021    r00:=r00 xor r01,         *r00:=0003
0021      if not e then stor;     *try stor--should store
0022    r00:=acc,                 *r00:=0330
0022      if e then stor;         *try stor--should not store!
0023    r14:=r15 xor r14,         *r14:=3000
0023      if not b15 then clo;    *check b15--should clear o
0024    r07:=r06 xor r07,         *r07:=00c0
0024      if y15 then ste;        *check y15--should be false
0025    r13:=r12 xor r13,         *r13:=0c00
0025      if o then ste;          *check alov--should be clear
0026    r03:=r03 xor r02,         *r03:=000c
0026      sto,                    *sto
0026      clf;                    *clf
0027    r01:=r00 xor r17,         *r01:=c003
0027      if not o then ste;      *check o--should not ste
0028    r11:=r03 xor r14,         *r11:=300c
0028      if f then ste;          *check f--should not ste
0029    r15:=r13 xor r07,         *r15:=0cc0
0029      stf,                    *stf
0029      ldq;                    *load q
002A    acc:=acc xor q,           *acc:=0ff0, use q
002A      if not f goto fail1;    *
002B    r12:=r01 xor r11,         *r12:=F00F
002B      if y15 then ste;        *check b15--should not ste
002C    acc:=acc xor r12,         *acc:=FFFF
002C      if e goto fail1,        *check e--if set then failed test
002C      ldq;                    *
002D    if not y15 goto fail1,    *acc should equal all ones
002D      nop:=not acc,           *nop:=0000
002D      sp1/acf;                *try acf--no carry
002E    if not yz goto fail1,     *test for all ones
```

```
002E      nop:=111(zeros);          *nop:= 0 or 1 based on DW
002F    if yz goto pl_ct_load;      *check for complementing the DW bit
0030    cmdw;                       *complement it to set it to zero
0031
0031  pl_ct_load:
0031  {
0031    check ctz condition by hedging against alu.
0031    new microorders used:
0031      cndx/ctz,ctz4
0031      bbus,stor/ct
0031      spl/fcin
0031  }
0031    ct:=0x5555;                 *Check for stuck at ones, stuck at zeros and
0032    nop:=ct xnor 0xAAAA;        *  stuck to adjacent.
0033    if not yz goto fail1,       *
0033      ct:=not ct, ldq;          *Try other sense. (load q with this pattern)
0034    ct:=ct xor 0xAAAA;          *CT gets zeros.
0035    if not yz goto fail1,       *
0035      acc:=zeros;               *ACC gets zeros
0036    if not ctz4 goto fail1,     *Check for ctz4 condition initially.
0036      nop:=acc+ct;              *ACC and CT should be zero.
0037  pl_ctloop:                    *
0037    if not yz goto fail1,       *If acc+ct do not equal zero, they are out of
0037      acc:=acc+one;             *  step.  Increment acc.
0038    if not ctz goto pl_ctloop,*If ct is zero, then check to make sure
0038      nop:=acc+ct;              *  acc is also zero.  Decrement ct.
0039    if not yz goto fail1,       *
0039      nop:=acc;                 *Check that ctz occured at the same time
003A    if not yz goto fail1,       *  acc was zero.
003A      ct:=ones;
003B
003B  pl_stack:
003B  {
003B     Call through the stack..even overwrite a location in stack.
003B  }
003B  r00:=0xF777;                  *r00=F777
003C  lcall pl_call1,               *Push first rtn address onto stack.
003C    bbus/r00, alu/xor;          *Set up for b15 check (remem status update)
003D  fail1:                        *
003D    lgoto FAILTRAP;             *FAIL: no overwrite.             (stk 1)
003E  pl_call1:                     *
003E    if not b15 call fail1,      *Should not call or push stack.
003E      r01:=111(r00);            *r01=EEEE
003F    if not sf rtn,              *        Check for bad stack pop.
003F      r02:=rrl(r01);            *r02=7777  Do it twice to offset possible
0040    if sf rtn,                  *        bad push at pl_call1.
0040      r03:=rrl(r02);            *r03=BBBB
0041    if not alov call pl_call2,*Should call pl_call2.
0041      r04:=lrl(r03);            *r04=5DDD
0042    rtn,                        *Should not return to stackfail.    (stk 2)
0042      nop:=zeros;               *
0043  pl_call2:                     *
0043    lcall pl_call3;             *Try a long call.
0044    rtn,                        *                                   (stk 3)
0044      rll:=rll(r10), lwf;       *rll=4779 (f was set)
0045  pl_call3:                     *
0045    scall pl_call4,             *Try a short call.                  (stk 4)
0045      r05:=rll(r04);            *r05=BBBA
0046    rtn,                        *
0046      r10:=rrl(r07), lwe;       *r10=9DDD
0047  pl_call5:                     *return down the chain.
0047    rtn,                        *
0047      r07:=rrl(r06), lwe;       *r07=3BBA
0048  pl_call4:                     *
0048    call pl_call5,              *Should call, and overwrite         (stk 1)
0048      r06:=rll(r05);            *r06=7775
0049    if not yz then rtn,         *
0049      r12:=r06 xor rll;         *r12=4CCE
004A    nop:=r12 xor 0x4cce;        *
004B    if not yz goto fail2,       *
004B      r01:=r00+acc;             *
004C
004C  pl_alu:
004C  {
004C    Check the carry mechanism..also check no status update.
004C  }
004C    if not cf goto fail2,       *cf should be true
004C      acc:=r03-acc;             *
004D    if not alov goto fail2,     *alov should be true
004D      r03:=r02 xor acc;         *
004E    r16:=r03 xor 0x2aa9;        *(save zeros for later test)
004F    if not yz goto fail2,       *results should be 2aa9
004F      nop:=sr;                  *
0050
0050  {
0050    End of PL diagnostic
0050    Output 7FFF to lights
0050    Check for looping on selftest
0050  }
0050   {nop:=sr and 0x8000;}       *Check for diagnostic looping.
0050    if not y15 goto            *If no loop then goto PU diagnostic
0050      PU_DIAGNOSTIC,           *
0050      acc:=111(ones);          *set up for blinky lights.
0051  blinky_lights:               *blinky lights routine
0051    pl_dloop:                  *rotate a zero through the lights register
0051      if not ctz goto pl_dloop,*  count to 64k
0051        lr:=acc;               *
0052      if sf goto pl_dloop,     *  shift the zero until done.
0052        acc:=rll(acc);         *
0053    goto TEST_RESTART,         *Restart pl selftest
0053      lr:=zeros;               *  turn all lights on
0054
0054  fail2: goto FAILTRAP;
0055
0055  PU_DIAGNOSTIC:
0055    lr:=not 0x7fff;            *show that PL diagnostic finished
0056
0056  pu_preg:
0056  {
0056    Store every value possible into the p register.
0056    Check ip.
0056    Check integrity of BBUS
0056  }
0056     p:=ones;                   *Set p to -1
0057     acc:=zeros, sp2/ip,        *Set acc to zeros, increment P to 0
0057       goto pu_plpentry;        *  using sp2/ip
0058  pu_ploop:                     *
0058     if not yz goto fail2,      *If acc and P are not equal, then fail.
0058       stor/acc, abus/acc,      *increment acc on ABUS.
0058       fcin,     alu/adac;      *
0059  pu_plpentry:                  *
0059     if not cf  goto pu_ploop,  *If increment of acc rolls over, then done.
0059       n:=acc-p, spl/ip;        *Else, compare acc and P and increment P.
005A
005A  pu_nreg:
005A  {
005A    Rotate a pattern through grin and prin.
005A    Check for dual addressing of register files.
005A    Load and check N register for all values.
005A    Check IN and DN.
005A  }
005A    r17:=1;                     *Load the test pattern
005B  { n:=zero;  }                 *start at first grin,prin location
005B    ct:=ones;                   *will loop 16 times (ctz4)
005C  n_loop:                       *Loop while incrementing N
005C  rin_setloop:                  *Load up registers
005C    grin:=r17, in;              *  load pattern into grin  (try IN)
005D    r17:=rll(r17), dn;          *  rotate pattern left (try DN)
005E    if not sf goto rin_setloop, *  if not done then loop (try IN)
005E      prin:=not r17, in;        *  load complement of test pattern into prin
005F  rin_checkloop:                *Check what was loaded
005F    nop:=grin-r17;              *  compare grin
0060    if not yz goto fail2,       *  if not equal then fail
0060      r17:=rll(r17);            *  rotate pattern
0061    nop:=prin xnor r17,         *  compare grin
0061      in;                       *    (increment N)
0062    if not yz goto fail2,       *  if not equal then fail
0062      acc:=zuy(n);              *    save n for test (upper byte is ?)
0063    if not sf goto              *  if not done then continue test
0063      rin_checkloop,            *    compare ct and acc (n)
```

```
0063       p:=ct xnor acc;          *  (load p with zeros for later test)
0064     if not yz goto fail2,      *  if ct is not complement of n then fail
0064       n:=n+one;                *  increment n (sliding diagonal)
0065     if not ctz4 goto           *  loop 16 times (decrement ct)
0065       n_loop,                  *
0065       memr:=q;                 *  load memr with 0xAAAA for following test
0066
0066   pu_memr:
0066   {
0066     check out memr and swzu,swap,swzl
0066   }
0066     r17:=zuy(memr),            *00aa
0066       if not mpen goto fail2;  *     check mpen condition
0067     memr:=not q,               *5555
0067       rdp, cle;                *     clear abfetch (p == 0)
0068     acc:=swzl(memr);           *5500 start "checksum" of alu functions
0069     p:=srg(r17);               *     try srg PROM
006A     acc:=asg(r17);             *     try ASG PROM
006B
006B   pu_ist:
006B   {
006B     check out ist
006B   }
006B     call RESET_PU,             *reset the IST
006B       acc:=p+acc;              *     continue checksum
006C     ist:=0xA80F;               *     set TBG flag (no FLTO)
006D     acc:=ist+acc,              *     checksum ist (ist lags by 2 cycles)
006D       if intp goto fail2;      *     interrupts should be false
006E     ist:=0xAA0F;               *     set FLTO int
006F     acc:=ist+acc,              *
006F       if not intp goto fail2;  *     interrupts should be true
0070     ist:=0xBA53;               *     set MP int (turn MP on)
0071     acc:=ist+acc,              *
0071       if intf goto fail2;      *     intf should be false
0072     ist:=0x55A1;               *     set PE interrupt (MP on)
0073     acc:=ist+acc,              *
0073       if mpen goto fail2;      *     check if mpen condition works
0074     call RESET_PU,             *     reset him, but checksum the
0074       acc:=ist+acc;            *         parity error code
0075     acc:=acc xor 0x789B;       *     look at the checksum (acc:= 0)
0076     if not yz goto fail2,      *
0076       nop:=111(sr);            *
0077
0077   {
0077     end of PU diagnostic
0077   }
0077    {nop:=sr and 0x4000;}       *If bit 14 is open then loop
0077     if not y15 goto MC_DIAGNOS, *
0077       s3 :=111(ones);          *acc:= FFFE
0078     goto blinky_lights,        *do the blinky lights stuff
0078       acc:=s3;                 *
0079
0079   fail3: goto FAILTRAP;        *
007A                                *
007A   MC_DIAGNOSTIC:               *(not here until checksum is in VCP)
007A     lr:= not 0x3FFF;           *store PU passed code to LEDs
007B     s1:=0x1FFE;                *save passed LED code for LR
007C     ct:=4095;                  *
007D     s4:=0x2000 and sr;         *
007E     n:=9;                      *(prepare to load PRIN LED copy)
007F     p:=020000;                 *
0080   checkloop:                   *Checksum of VCP ROM from 020000-037777
0080     rdp,                       *  read bootrom location
0080       s2:=zly(s1);             *
0081     if not ctz goto checkloop, *  and checksum until count = 0
0081       acc:=t+acc, ip;          *
0082     if not yz goto fail4,      *is checksum zero? n: fail
0082       nop:=s4;                 *
0083     if not yz goto tstlink,    *is MC loop bit set?
0083       lr:=not s1;              *store "passed" LED code to LR
0084     call FL_DIAGNOSTIC,        *
0084       prin:=s2;                *save "passed" LED code in prin reg
0085     goto START_UP;             *
0086
0086   tstlink:                     *go to blinky lights
```

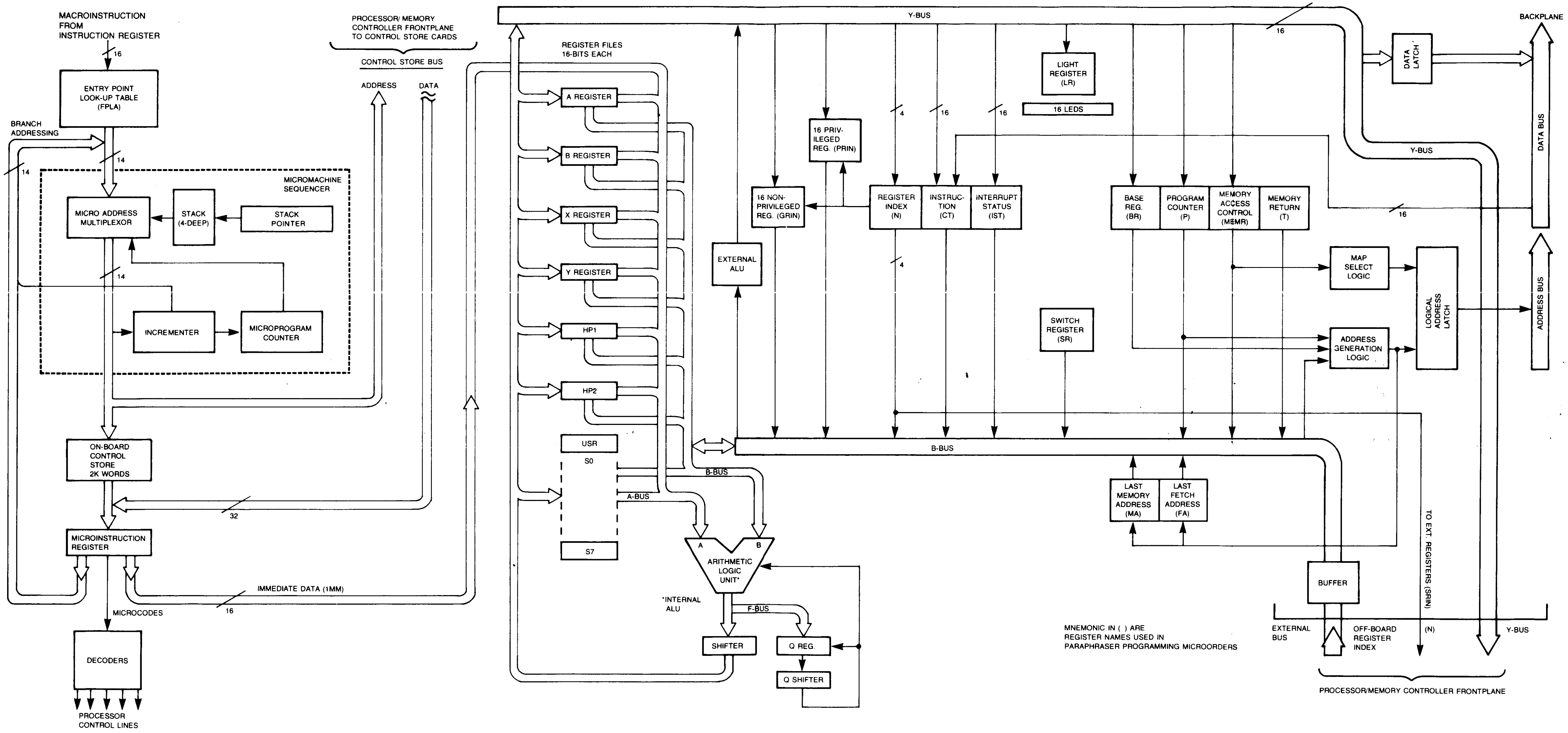
```
0086   goto blinky_lights,         *
0086     acc:=s3;                  *
0087
0087  fail4:                       *
0087    goto FAILTRAP;             *
0088
01B8  FL_DIAGNOSTIC: $origin 0x1b8$ *if FL is installed, it will execute
01B8    rtn;                       *  selftest here and extinguish LEDs
01B9
01B9
01BA
```

# APPENDIX F
## FUNCTIONAL BLOCK DIAGRAM

# FUNCTIONAL BLOCK DIAGRAM

APPENDIX

F



8200-6

Figure F-1. Functional Block Diagram

# APPENDIX G
## DEBUGGING MICROCODE

# DEBUGGING MICROCODE

APPENDIX G

A logic analyzer can be used for efficient debugging of A700 processor microcode. With a logic analyzer, the actual micromachine execution can be followed, and the logic analyzer can be programmed to trace the micromachine execution upon the detection of specified conditions. This appendix describes the signals that are accessible and the method of connecting a logic analyzer.

## G-1. DIAGNOSTIC CONNECTIONS

The most important signals for microcode tracing are on the Microaddress Bus (to follow microcode execution) and the Y-Bus (to follow the results of arithmetic operations). The processor clock (PC−) should be used by the logic analyzer as the data-gathering clock. This clock is frozen during cycles when the micromachine is waiting for memory or map accesses; therefore, data-gathering will not occur during frozen cycles. Signal TESTSC− (system clock) is not frozen during these memory accesses and is equivalent to the backplane clock.

These signals and others are accessible through two connectors on the A700 frontplane. The A700 processor should be used with these cables attached only during microcode debugging and not during normal operation. Be sure to turn power off before connecting or disconnecting diagnostic cables.

Connector J4 on the front plane is used for connection to the WCS and PCS cards. Pins 33 through 46 of this connector contain the control store address bus. A special cable to connect the frontplane to the WCS and also to the logic analyzer must be supplied and installed by the customer for debugging microcode. Table G-1 shows the pin-to-signal identification for connector J4.

Connector J5 on the frontplane is used solely for connection to a logic analyzer. It contains the Y-Bus and the processor clock signals, in addition to some other signals. This cable must be supplied and installed by the customer. Table G-2 shows the pin-to-signal identification for connector J5.

## G-2. PROGRAMMING THE LOGIC ANALYZER

The logic analyzer should be programmed with a format that accommodates these signals and with a trace specification that corresponds to the conditions you want to trace.

The signals of the Y-Bus and control store address bus are positive-true, and the rising edge of the processor clock should trigger data-gathering.

At the rising edge of the processor clock, the control store address bus has the microaddress of the next microinstruction, while the Y-Bus has the output of the ALU from the current microinstruction. This means that the contents of the Y-Bus due to a certain microinstruction are displayed in the cycle after the contents of the microaddress for that microinstruction are displayed.

Table G-1. Connector J4 Signal Identification

| PIN | SIGNAL (* = RECOMMENDED FOR MICROCODE DEBUGGING) |
|---|---|
| 1<br>through<br>32 | Control Store Data Bit 0<br>through<br>Control Store Data Bit 31 |
| *33<br>*through<br>*46 | Control Store Address Bit 0<br>through<br>Control Store Address Bit 13 |
| 47 | Unused |
| 48 | CSIDWC– Bottom of WCS/PCS Priority Chain (negative true) |
| 49, 50 | Ground |

Table G-2. Connector J5 Signal Identification

| PIN | SIGNAL (* = RECOMMENDED FOR MICROCODE DEBUGGING) |
|---|---|
| *1 - 16 | Y-Bus Data Bit 0 through Y-Bus Data Bit 15 |
| 17, 18 | Ground |
| 19 | BBUS/SRIN Microorder (negative true) |
| 20 | CSADIS– Disable Processor Control Store (input for testing) |
| 21 | BPON+ Processor Power-On Line |
| 22 | OP1/JTAB Microorder (negative true) |
| *23 | Processor Clock PC– (negative true, rising edge triggering) |
| 24 | INTP Condition Line |
| 25, 26 | Ground |
| 27 | FREEZE– Memory Access Freeze Line (negative true) |
| 28 | TESTSC– Diagnostic Clock Line (negative true) |
| 29 | ECLK (external clock for testing) |
| 30 | CKDIS– (disable internal clock for testing) |
| 31, 32 | Ground |
| 33 | INTF Condition |
| 34 | STOR/GRIN (negative true) |
| 35 | STOR/SRIN (negative true) |
| 36 | STOR/P (negative true) |
| 37 | STOR/PRIN (negative true) |
| 38 | STOR/IST (negative true) |
| 39 | FFRZ– (for testing) |
| 40 | STOR/MEMR (negative true) |
| 41 | N-Register Bit 1 |
| 42 | N-Register Bit 0 |
| 43 | N-Register Bit 3 |
| 44 | N-Register Bit 2 |
| 45, 46 | CSIDWC– Bottom of WCS/PCS Priority Chain (negative true) |
| 48 | Unused |
| 49, 50 | Ground |

## G-3. DIAGNOSTIC WINDOW MICROCODE

Special microcode is available in the A700 base set for debugging user microcode. This microcode, called the "Diagnostic Window", also provides a look at base set execution. The diagnostic JTAB loop is employed to load the window contents onto the Y-Bus where it is available for display on the logic analyzer. The diagnostic window is selected when switch 5 (labeled DW) on the frontplane is in the "open" position.

The diagnostic JTAB loop is entered only after power-on, reset, or an interrupt. This loop adds about 2 microseconds per macroinstruction execution time; therefore, the execution speed of your A700 processor will be degraded while the diagnostic JTAB loop is selected. Be sure to reset switch DW to the "closed" position when microprogram debugging is completed.

The diagnostic JTAB LOOP microcode is located at 0xE0 through 0xEA. Between macroinstructions, the information is passed to the Y-Bus so that it can be displayed on the logic analyzer. The information is displayed in the line following the line being executed due to the relation of the Y-Bus, the microaddress bus, and the processor clock.

The information at the ten locations is as follows:

a. 0xE0 (shown at location 0xE1). The T-register is on the Y-Bus at this point in the diagnostic JTAB loop, the T-register contains the next macroinstruction to be executed.

b. 0xE1 (shown at location 0xE2). The contents of the FA (fetch address) latch are displayed through the Y-Bus. This is the logical address from which the macroinstruction (opcode) was fetched.

c. 0xE2. A microinstruction containing the JTAB microorder and a jump-to-subroutine is performed to the "jump table area" (locations 0x100 through 0x1FF) of the base set.

d. Microprogram execution.

e. 0xE3 (shown at location 0xE4). Microprograms will return to the JTAB loop at location 0xE3, where the A-register is displayed through the Y-Bus.

f. 0xE4 through 0xE6 (shown at locations 0xE5 through 0xE7). The B, IST, and MEMR registers are displayed successively.

g. 0xE8 (shown at location 0xE9). The contents of the PRIN register that contains interrupt information is displayed.

h. 0xE9. Interrupts are checked before the loop is continued.

Summary of Diagnostic JTAB loop (as would be shown on a logic analyzer):

| | |
|---|---|
| 0xE1 | Macroinstruction to be executed. |
| 0xE2 | Logical Memory Address of macroinstruction |
| — | (Microprogram Execution) |
| 0xE4 | A-register. |
| 0xE5 | B-register. |
| 0xE6 | IST-register. |
| 0xE7 | MEMR-register. |
| 0xE9 | PRIN-register (microcode-kept interrupt status). |

Other microroutines of the base set microcode may be useful for tracing execution of the A700 processor. The interrupt service routine jump table, at locations 0xF0 through 0xFF is useful for tracing interrupt servicing. Microaddress 0 wll be executed on microcode timeout.

Refer to the base set listing for specific information about interrupt handling and instruction execution. (Note: HP reserves the right to change the A700 base set. You should not branch into the A700 base set to make use of existing routines.)

# INDEX

# INDEX

A

A (R00), Macro A-register Microorder, Table 4-2, 4-11, 4-20, 4-21
ACC (R04) Microorder, Table 4-2, 4-11, 4-20, 4-21
Accessories, Microprogramming, 2-1
ACF Microorder, Table 4-2, 4-11
ADAC Microorder, Table 4-2, 4-19
ADBC Microorder, Table 4-2, 4-19
ADDC Microorder, Table 4-2, 4-19
AL1 Microorder, Table 4-2, 4-11
ALOV Microorder, Table 4-2, 4-18
ALU Functions, 2-8, 4-7
AND Microorder, Table 4-2, 4-19
AR1 Microorder, Table 4-2, 4-11
Arithmetic Data Paths, 4-7
Arithmetic Logic Unit (ALU), 2-8
ASG Microorder, Table 4-2, 4-13

B

B (R01), Macro B-Register Microorder, Table 4-2, 4-11, 4-20, 4-21
B15 Microorder, Table 4-2, 4-17
Base Set, 2-18
BFB Microorder, Table 4-2, 4-12
Boot Memory, 2-7
Bus System, 2-10

C

CAB Microorder, Table 4-2, 4-19, 4-21
CAND Microorder, Table 4-2, 4-19
CF Microorder, Table 4-2, 4-18
CK2 Microorder, Table 4-2, 4-12
CK2, Floating Point Data Transfers, 4-12, 11-5
CLE Microorder, Table 4-2, 4-11
CLF Microorder, Table 4-2, 4-16
CLO Microorder, Table 4-2, 4-12
CMAC Microorder, Table 4-2, 4-19
CMBC Microorder, Table 4-2, 4-19
CMDW Microorder, Table 4-2, 4-16
CMID Microorder, Table 4-2, 4-16
Comments, Paraphraser, 7-3
Computer Functions, Microprogram Controllable, 2-1
Control Statement, Paraphaser, 7-3
Control Store, 2-17
Control Store Mapping, 6-1
CT (Register) Microorder, Table 4-2, 4-19, 4-21
CT30 Microorder, Table 4-2, 4-17
CT74 Microorder, Table 4-2, 4-17
CTZ Microorder, Table 4-2, 4-18
CTZ4 Microorder, Table 4-2, 4-18
CWRB Microorder, Table 4-2, 4-21
CXY Microorder, Table 4-2, 4-19, 4-21

D

DCT Microorder, Table 4-2, 4-17
Debugging Microcode, 3-4, G-1
Directives, Paraphaser, 7-5
Directly-Accessed Registers, 2-10
DIV Microorder, Table 4-2, 4-14
DIV1 Microorder, Table 4-2, 4-14
DN Microorder, Table 4-2, 4-17
DNRM Microorder, Table 4-2, 4-14

E

E (Extend Register) Microorder, Table 4-2, 4-18
ENOE Microorders, Table 4-2, 4-12
Error Conditions, Floating Point, 11-5

F

F (Flag) Microorder, Table 4-2, 4-18
FA Microorder, Table 4-2, 4-20
FCHB Microorder, Table 4-2, 4-13
FCHP Microorder, Table 4-2, 4-17
FCIN Microorder, Table 4-2, 4-11
Field, Definition of, 2-15
Floating Point Processor, 11-1
Floating Point, Control Store Address, 11-10
Floating Point, Control Word Format, 11-6
Floating Point, Divide Operation Code, 11-9
Floating Point, Writing Microcode, 11-6

G

GRIN Microorder, Table 4-2, 4-20, 4-21

H

Hardware, Microprogramming, 3-1
HP1 (R05) Microorder, Table 4-2, 4-11, 4-20, 4-22
HP2 (R06) Microorder, Table 4-2, 4-11, 4-20, 4-22

I

IF Conditional Phrases, 7-23
IF Conditional RTN Phrases, 7-23
IF Conditional SP0 Phrases, 7-24
IFCH Microorder, Table 4-2, 4-13
IMM Microorder, Table 4-2, 4-11
IN Microorder, Table 4-2, 4-12
Indirectly-Accessed Registers, 2-12
INOR Microorder, Table 4-2, 4-19
Input/Output Section, 2-10

Interrupt Handling, 6-9
INTF Microorder, Table 4-2,
INTP Microorder, Table 4-2, 4-18
IOR Microorder, Table 4-2, 4-19
IORQ Microorder, Table 4-2, 4-18
IP Microorder, Table 4-2, 4-11, 4-17
IST Microorder, Table 4-2, 4-20, 4-21

J

JMP Microorder, Table 4-2, 4-10
JMPF Microorder, Table 4-2, 4-10
JMPL Microorder, Table 4-2, 4-10
JMPT Microorder, Table 4-2, 4-10
JSB Microorder, Table 4-2, 4-10
JSBF Microorder, Table 4-2, 4-10
JSBL Microorder, Table 4-2, 4-10
JSBT Microorder, Table 4-2, 4-10
JTAB Microorder, Table 4-2, 4-9

L

Labels, Paraphaser, 7-4
LDBR Microorder, Table 4-2, 4-13
LDQ Microorder, Table 4-2, 4-12
LL1 Microorder, Table 4-2, 4-12
LR Microorder, Table 4-2, 4-21
LR1 Microorder, Table 4-2, 4-12
LWE Microorder, Table 4-2, 4-17
LWF Microorder, Table 4-2, 4-17

M

MA Microorder, Table 4-2, 4-20
Macroinstruction Execution, 2-21
Macroinstruction Fetching, 2-19
MAP Microorder, Table 4-2, 4-20, 4-21
Maps, Data and Code, 4-28
Memory Section, 2-9
MEMR Microorder, Table 4-2, 4-20, 4-21
Microcycle, Definition of, 4-15
Microinstruction, Definition of, 4-15
Microorder Definitions, 4-9
Microorder, Definition of, 4-15
Microprogram Control Sections, 2-8
MPEN Microorder, Table 4-2, 4-18

N

N (Register) Microorder, Table 4-2, 4-20, 4-21
NAND Microorder, Table 4-2, 4-19
NOP Microorder, Table 4-2, 4-9, 4-12, 4-17, 4-21
Numbers, Paraphaser, 7-3

O

O (Overflow Register) Microorder, Table 4-2, 4-18

P

P (Register) Microorder, Table 4-2, 4-20, 4-21
Paraphraser, 7-1
Paraphraser Error Messages, 8-5
Paraphraser Rules, 7-2
Paraphraser, Execution of, 8-2
Paraphraser, Floating Point Division Sentence, 11-10
Paraphraser, Floating Point Sentence, 11-8
Phrases, Arithmetic, 7-13
Phrases, Arithmetic with ALU Special, 7-19
Phrases, Arithmetic with Carry-in Modifier, 7-17
Phrases, Arithmetic with Immediate Data, 7-21
Phrases, Arithmetic with Shift or Rotate, 7-16
Phrases, Basic Arithmetic,7-14
Phrases, Branching, 7-11
Phrases, Conditional, 7-23
Phrases, Conditional Branching, 7-23
Phrases, Conditional Return, 7-23
Phrases, Conditional SP0, 7-24
Phrases, Field Forcing, 7-24
Phrases, Paraphaser, 7-9
Phrases, Special, 7-24
Phrases, Types of, 7-11
PON Microorder, Table 4-2, 4-18
Preparatory Steps, Microprogramming, 3-1, 3-4
PRIN Microorder, Table 4-2, 4-20, 4-21
Processor Cards, Overall Circuit Description, 2-5
PROM Burning, 10-1

Q

Q (Register) Microorder, Table 4-2, 4-20

R

R00, Macro A-Register Microorder, Table 4-2, 4-11, 4-20, 4-21
R01, Macro B-Register Microorder, Table 4-2, 4-11, 4-20, 4-21
R02, Macro X-Register Microorder, Table 4-2, 4-11 4-20, 4-21
R03, Macro Y-Register Microorder, Table 4-2, 4-11 4-20, 4-21
R04, ACC Microorder, Table 4-2, 4-11, 4-20, 4-21
RDB Microorder, Table 4-2, 4-12
RDIO Microorder, Table 4-2, 4-17
RDP Microorder, Table 4-2, 4-12
RDPC Microorder, Table 4-2, 4-17
Registers Reserved For The User, 6-10
Restrictions Microcode, 4-27
RL1 Microorder, Table 4-2, 4-12
RL4 Microorder, Table 4-2, 4-14
RR1 Microorder, Table 4-2, 4-12
RTN Microorder, Table 4-2, 4-9
RTNF Microorder, Table 4-2, 4-10
RTNT Microorder, Table 4-2, 4-10

S

S0-S7 Microorders, Table 4-2, 4-11, 4-20, 4-22
SBAC Microorder, Table 4-2, 4-19
SBBC Microorder, Table 4-2, 4-19
Sentence, Microinstruction, 7-4
Sentence, Paraphaser, 7-7
SF Microorder, Table 4-2; 4-18
SM2C Microorder, Table 4-2, 4-15
SNRM Microorder, Table 4-2, 4-15
Software Entry Points, 6-2
Source File Example, 7-8
SP0F Microorder, Table 4-2, 4-10
SP0T Microorder, Table 4-2, 4-10
SPEC Microorder, Table 4-2, 4-19
SR Microorder, Table 4-2, 4-20
SRG Microorder, Table 4-2, 4-15
SRIN Microorder, Table 4-2, 4-20, 4-22
SRIN-C, SRIN-D, Floating Point Input Operands, 11-4
SRIN-E, SRIN-F, Floating Point Results, 11-5
STE Microorder, Table 4-2, 4-12
STF Microorder, Table 4-2, 4-17
STO Microorder, Table 4-2, 4-13
STOR Microorder, Table 4-2, 4-13
Support Software, Microprogramming, 3-2
SWAP Microorder, Table 4-2, 4-15
SWZL Microorder, Table 4-2, 4-15

T

T (Register) Microorder, Table 4-2, 4-21
Time-Out, Microcode, 6-10
Timing, computer, 5-1
Timing, I/O access, 5-1
Timing, memory access, 5-1
Timing, processor clock freeze, 5-2
TMLC Microorder, Table 4-2, 4-16
TMPY Microorder, Table 4-2, 4-16

U

UIG (User Instruction Group), 6-2
UIG Blocks, User Area, 6-4
UIG Mapping, 6-5
UIG Op Codes, 6-4
UMPY Microorder, Table 4-2, 4-16
USR (R07) Microorder, Table 4-2, 4-11, 4-20, 4-22

V

Virtual Control Panel (VCP), 2-7

W

WCS Mapping, 9-1
WLOAD Commands, 9-2
WLOAD, Using, 9-2
Word Type 1, 4-3
Word Type 1: OP Field Microorders, Table 4-2, 4-9
Word Type 2, 4-4
Word Type 2: OP Field Microorders, Table 4-2, 4-10
Word Type 3, 4-5
Word Type 3: OP Field Microorders, Table 4-2, 4-10
Word Type 4, 4-5
Word Type 4: OP Field Microorders, Table 4-2, 4-10
Word Type 5, 4-6
Word Type 5: OP Field Microorder, Table 4-2, 4-10
Word Type 6, 4-6
Word Type 6: OP Field Microorders, Table 4-2, 4-11
Word Types 1-4, ALUS Field Microorders, Table 4-2, 4-13
Word Types 1 and 2: A-Bus Field Microorders, Table 4-2, 4-11
Word Types 1 and 2: SP0 Field Microorders, Table 4-2, 4-2
Word Types 1 and 4: SP2 Field Microorders, Table 4-2, 4-16
Word Types 1-4: SP0 and SP1 Fields Microorders, Table 4-2, 4-11
Word Types 1-6: B-Bus Field Microorders, Table 4-2, 4-19
Word Types 1-6: ALU Field Microorders, Table 4-2, 4-19
Word Types 1-6: STOR Field Microorders, Table 4-2, 4-21
Word Types 2 and 3: CNDX Field Microorders, Table 4-2, 4-17
WRB Microorder, Table 4-2, 4-22
WRIO Microorder, Table 4-2, 4-17
WRP Microorder, Table 4-2, 4-22

X

X (R02), Macro X-Register Microorder, Table 4-2, 4-11, 4-20, 4-21
XNOR Microorder, Table 4-2, 4-19
XOR Microorder, Table 4-2, 4-19

Y

Y (R03), Macro Y-Register Microorder, Table 4-2, 4-11, 4-20, 4-21
Y15 Microorder, Table 4-2, 4-18
YZ Microorder, Table 4-2, 4-18

Z

ZERO Microorder, Table 4-2, 4-19
ZLY Microorder, Table 4-2, 4-16
ZUY, Table 4-2, 4-16

# MANUAL UPDATE

| MANUAL IDENTIFICATION | | UPDATE IDENTIFICATION | |
|---|---|---|---|
| Title: | HP 92045A Microprogramming Package Reference Manual | Update Number: | 2 (July 1982) |
| Part Number: | 92045-90001 | This Packet also Includes: | 1 (May 1982) |

THIS UPDATE GOES WITH: First Edition (February 1982)

THE PURPOSE OF THIS MANUAL UPDATE
is to provide new information for your manual to bring it up to date. This is important because it ensures that your manual accurately documents the current version of the product.

THIS UPDATE CONSISTS OF
this cover sheet, a printing history page, all replacement pages, and write-in instructions (if any). Replacement pages are identified by the update number at the bottom of the page. A vertical line (change bar) in the margin indicates new or changed text material. The change bar is not used for typographical or editorial changes that do not affect the text. New pages to be added do not contain change bars.

TO UPDATE YOUR MANUAL
identify the latest Update (if any) already contained in your manual by referring to the Printing History Page (page ii). Incorporate only the Updates from this packet not already included in your manual. Following the instructions on the back of this page, replace existing pages with the Update pages and insert new pages as indicated. If any page is changed in two or more Updates, such as the Printing History Page which is furnished new for each Update, only the latest page will be included in the Update package. Destroy all replaced pages. If "write-in" instructions are included they are listed on the back of this page.

HEWLETT PACKARD

HEWLETT-PACKARD COMPANY
Data Systems Division
11000 Wolfe Road
Cupertino, California 95014

## TECHNICAL MANUAL UPDATE

## (92045-90001)

**Note that "*" indicates a changed page.**

**UPDATE** **DESCRIPTION**

1 A. Replace the following pages with the pages supplied:

Title page*/ii*
9-1*/9-2

B. Write the following changes of the pages indicated:

1. Page 9-6. In command

   "LB, *input file* or *lu*", delete "or *lu*".

2. Page 9-6. In sentence following "where" for the command in (1) above, delete "or lu is the input device (e.g. cartridge tape) containing microcode."

3. Page 11-10, last sentence. Change referenced manual to "HP 12156A Floating Point Processor Kit Installation and Reference Manual, part no. 12156-90001.

2 A. Replace the following pages with the pages supplied:

3-3*/3-4
12-3/12-4*

# HP 92045A Microprogramming Package

## Reference Manual

Includes:
Paraphraser Programming
WLOAD WCS Loader and
PROM Burn Program

HEWLETT PACKARD

HEWLETT-PACKARD COMPANY
Data Systems Division
11000 Wolfe Road
Cupertino, California 95014

MANUAL PART NO. 92045-90001
Printed in U.S.A. February 1982
U0782

# PRINTING HISTORY

The Printing History below identifies the Edition of this Manual and any Updates that are included. Periodically, Update packages are distributed which contain replacement pages to be merged into the manual, including an updated copy of this Printing History page. Also, the update may contain write-in instructions.

Each reprinting of this manual will incorporate all past Updates, however, no new information will be added. Thus, the reprinted copy will be identical in content to prior printings of the same edition with its user-inserted update information. New editions of this manual will contain new information, as well as all Updates.

To determine what software manual edition and update is compatible with your current software revision code, refer to the appropriate Software Numbering Catalog, Software Product Catalog, or Diagnostic Configurator Manual.

| | |
|---|---|
| First Edition | Feb 1982 |
| Update 1 | May 1982 |
| Update 2 | Jul 1982 |

**NOTICE**

The information contained in this document is subject to change without notice.

HEWLETT-PACKARD MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Hewlett-Packard shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance or use of this material.

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

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

## 3-3. THE PARAPHRASER MICROCODE MICROASSEMBLER

The paraphaser microcoding microassembler language converts a source microprogram into binary object code which may be directed to an output device and/or stored in a disc file. The paraphraser is a necessary tool for preparing microprograms since the microinstruction word length is 32 bits which makes other coding methods difficult.

The source may be input from an input device or disc file. The disc file is easiest since this file can be the same file developed when writing and editing the program with the HP 1000 Editor. The object code will be in the standard microinstruction format which is recognized by the WLOAD utility routine. The program can supply a source listing, a floating field listing of the microinstructions, a label listing, and a list of any errors.

The paraphraser program name is MPARA. MPARA can run with or without the File Manager, and it requires a minimum of 28k words of memory. All information on preparation of microprograms with the paraphraser and output of the microprograms is contained in Sections 7 and 8 of this manual.

## 3-4. DRIVER ID.41

Driver ID.41 must be configured into the RTE system during system generation to provide software linking between MPARA, WLOAD, and the WCS card.

NOTE

> The microprogramming support software can be included either during system generation or loaded into the system when required.

Driver ID.41 drives HP 12153A WCS cards for reads and writes (from and to main memory) and allows control of WCS board functions. The driver utilizes DMA which provides fast data transfer.

When configured in the RTE system, all WCS cards should have a select code of octal 20 or higher. In the system, the driver can be called directly with an EXEC call, or through the WLOAD program (refer to the *RTE Driver ID.41 For HP 12153A WCS Cards Reference Manual).*

## 3-5. WLOAD

The WCS I/O Utility program WLOAD uses driver ID.41 and transfers microprogram object code into WCS when run by the user. Section 9 in this manual contains information on WLOAD used as an I/O utility. WLOAD also includes a PROM "burn tape" function (see paragraph 3-7).

## 3-6. LOADING THE MICROPROGRAMMING SUPPORT SOFTWARE

The WCS driver ID.41 must be loaded at system generation time. (Refer to *RTE Driver ID.41 For HP 12153A WCS Card Reference Manual,* part no. 92045-90002.) The WLOAD program must be loaded on line using the RTE-A.1 LINK program. The MPARA microprogramming language can be loaded on an RTE-A system using the LINK program or on an RTE-6/VM system using the LOADR program.

## 3-7. PROM CODE GENERATOR

The process of loading the microcode into the PROMs (Programmable Read Only Memory) is accomplished for fusing ("burning") the binary bits into the PROM chip. The binary code for the PROMs is generated by the PROM "burn tape" function of WLOAD that uses the final binary object code of the microprogram as input. The program should be tested and debugged by running the program from a WCS card before making expensive PROMs. For additional information on PROM burning, refer to Section 10 of this manual.

## 3-8. PREPARATORY STEPS

Condensed information on the required preparatory steps for microprogramming appear in Table 3-1 along with references to the sections of this manual (or to applicable documents). The letters in the referenced column are keyed to entries in Table 3-2, and the numerals refer to sections in this manual.

Table 3-2 is a list of HP 92045A Microprogramming Software and HP manuals used by the microprogrammer for the HP 1000 A700 computer systems. Section 12 provides examples of the procedures you may want.

In preparation for microprogramming, the WCS cards to be used must be initialized before they can be used.

## 3-9. DEBUGGING MICROCODE

After you have written your source microcode and fixed any errors found by MPARA, load the object code into WCS and try running it. If its performance is not to your satisfaction you will want to "debug" it. Microcode debugging on the A700 processor is most efficiently accomplished through the use of a logic analyzer. Hewlett-Packard logic analyzers are recommended since they were used throughout the development of the base set and floating point microcode and provided the desired results.

A logic analyzer allows the actual micromachine execution to be followed, and it can be programmed to trace the micromachine execution upon detection of certain conditions. Details on connecting a logic analyzer and information on its use are given in Appendix G (Debugging Microcode).

# WRITABLE CONTROL STORE (WCS) SUPPORT SOFTWARE

SECTION 9

The previous section (Section 8) describes a method of preparing a microprogram and storing this source program in a system file. The source program, prepared "off line" or on some other system, could have been stored in a system file by loading it through a system input device. The source program is then translated by the paraphraser (MPARA) microassembler program and filed as binary object code (or microcode) in another system file. This later file is the ready-to-use microinstructions of your program. In order to make use of this microcode it must be moved into the Control Store (micromachine memory) of the computer.

The computer's extended Control Store for user programs is provided by Writable Control Store (WCS) and PROM Control Store (PCS) cards. Normally, the microprogram is initially loaded into a WCS card so that test runs of the program can demonstrate that it has no "bugs" before burning PROMs to install on a PCS card.

The WCS cards are loaded by using a program called "WLOAD." WLOAD is a utility program which loads WCS and generates PROM "burn tape" code under the RTE operating system. An understanding of WCS memory mapping is essential for loading microprograms into it. This subject is summarized below. For additional information on the WCS card, which can be useful to the user for a better understanding of how to load it, refer to the HP 1000 A700 User Control Store Installation and Reference Manual, part no. 02137-90003. The WLOAD PROM "burn tape" function is covered in Section 10.

## 9-1. WCS MAPPING

The micromachine of the HP A700 computer has a microcode address space of 16k words of which the user may use 8k words. The 16k words are conceptually divided into 16 logical 1k modules numbered from 0 through 15. Each WCS card contains four banks of RAMs (Random Access Memory) for a total of 4k-words per card. The banks are numbered 0, 1, 2, and 3.

A mapping RAM on each card maps the logical modules to the physical banks. The map RAM has 16 locations each of which corresponds to a logical module. On each card, the logical module may be assigned a physical block that will be enabled when addressed through mapping, or it may be unmapped. If a logical module is mapped on more than one card at a time, the card which is higher priority in the control store chain will be enabled and will disable the other cards (including the processor control store).

NOTE

> On power-up, the mapping RAM will be in an unknown state. The IN (initialize) command should always be used to unmap all logical modules. Care should always be taken to assure that the WCS has been properly loaded (mapping RAM and data RAMs) before turning WCS on.

## 9-2. USING WLOAD

To load a WCS card using WLOAD, the user assigns an LU to the card to identify its I/O location for program interaction. Next, the appropriate logical to physical mapping is set up, and then a file or files are usually specified from which to download data.

An example of running WLOAD to load a WCS card follows:

```
RU,WLOAD
```

NOTE

> The WLOAD prompt "xx>" will appear on your terminal where xx is the WCS LU which is currently specified. In this example, the WCS LU is specified by the user as 63 is the first step under WLOAD.

Continue with this procedure while running under WLOAD execution:

| PROCEDURE | COMMENTS |
|---|---|
| 0>LU,63 | WLOAD starts up with LU=0; User enters LU of WCS. |
| 63>IN | Initialize. Turn off WCS and unmap all logical modules. |
| 63>EQ,4,0 | User equates the logical module 4 address (1000-13FF hex) to physical bank 0. |
| 63>LB,%EXMPL | User loads microcode from the binary format file %EXMPL (example). |
| 63>ON | User turns WCS on. |
| 63>EX | Exit program. |

## 9-3. WLOAD COMMANDS

WLOAD commands are two characters. Some of the commands require parameters which may be included on the command line separated by commas. If required commands are not included on the command line, WLOAD will prompt for the parameters.

Commands which read from or write to either the data RAMs or the map RAMs require that WCS be turned off. If WCS was on when such a command is executed, WCS will be automatically turned off, and it will be turned back on after execution of the command, unless a WCS I/O error occurs.

Before turning WCS off or executing the command, the input parameters are checked for validity. The following checks are performed as applicable:

1. If the logical module is between 0 and 15.
2. If the physical bank is between 0 and 3.
3. If the WCS address or data is in hex format and the address <16k (4000).
4. If the input file (or LU) can be opened.
5. If the output file (or LU) can be opened or created.

## 12-2. BUFFER INITIALIZATION EXAMPLE

EXAMPLE 1: INITIALIZE BUFFER, FORTRAN PROGRAM

```
FTN7X,L,I,Y
      PROGRAM JOKE3
C
C MAIN PROGRAM:
C
C   Calls assembly routine 'INIT' to initialize a
C   user buffer.  'INIT' invokes the microcode.
C
C RUN STRING:  RUN,JOKE3,START,INC,NUMBR
C
C              START:  Starting value of buffer
C              INC:    Increment between values
C              NUMBR:  Total number of elements
C
      IMPLICIT INTEGER (A-Z)
      DIMENSION BUFF(10000)
      DIMENSION PARMS(5)
      EQUIVALENCE (PARMS(1),START), (PARMS(2),INC)
      EQUIVALENCE (PARMS(3),NUMBR)
C
C Get starting value, increment, and number of elements
C
      CALL RMPAR(PARMS)
      IF (IABS(NUMBR) .GT. 10000) GOTO 999
C
C Initialize the buffer
C
      CALL INIT(BUFF,START,INC,NUMBR)
C
C Print the buffer to scheduling lu
C
      SESN = -1
      LU = LOGLU(SESN)
      WRITE(LU,10)(BUFF(J),J=1,NUMBR)
10    FORMAT(8(2X,I6))
999   END
```

EXAMPLE 1: INITIALIZE BUFFER, ASSEMBLER INTERFACE

```
MACRO,L
      NAM INIT,7
*
* Calls the microcode to initialize the buffer passed
* by the calling program.
*
* CALLING SEQUENCE:
*
*        CALL INIT(BUFF,START,INC,NUMBR)
*
      ENT INIT
      EXT .ENTR,.INIT
*
* .INIT must be declared as an entry point in a seperate
* assembly module as follows:  .INIT  RPL  105500B
*
BUFF  BSS 1            BUFFER ADDRESS
START BSS 1            STARTING VALUE
INC   BSS 1            INCREMENTS
NUMBR BSS 1            NUMBER OF ELEMENTS
*
```

```
*
INIT  NOP
      JSB .ENTR           GET PARAMETERS
      DEF BUFF
*
* Branch to control Store Address (0x3000)
*
      JSB .INIT           USER OPCODE
      DEF @BUFF           BUFFER ADDRESS
      DEF @START          STARTING VALUE
      DEF @INC            INCREMENT
      DEF @NUMBR          NUMBER OF ELEMENTS
*
      JMP @INIT           RETURN
      END INIT
```

## EXAMPLE 1: INITIALIZE BUFFER, MICROPROGRAM

```
MPARA,L,F;
*
UG_INIT: $origin 0x3030$
*
* DESCRIPTION:
*
*        Instruction 'INBUF' initializes a buffer in the user
*        program as specified by the calling program.  The
*        instruction is non-interruptable and does not check
*        for interrupts after every write to user memory.
*        The memory protect logic is enabled so that any memory
*        violation will be detected before the next instruction
*        is executed (ie. before the next JTAB).  Therefore,
*        memory is protected throughout the entire instruction.
*
* CALLING SEQ:
*
*        JSB .INIT
*        DEF BUFF  (,I)
*        DEF START (,I)
*        DEF INC   (,I)
*        DEF NUMBR (,i)
*
* Where: BUFF    is the user buffer
*        START   is the starting value to initialize
*                the buffer with (ie. buff(1)=start)
*        INC     the increment to the next buffer value
*                (ie. buff(2) = start+inc)
*        NUMBR   the number of words to initialize
*                (if numbr <0 use abs(numbr))
*
*
EXTRNL:  $define adrl/inst_restart 0x00D0$
*
*
INT_BUF: rdp, ip;                      *read def buff
         call RSV_IND;                 *resolve indirects
*
* must complete read started by RSV_IND
* s7 has the direct buffer address on return
```

# READER COMMENT SHEET

**HP 92045A MICROPROGRAMMING PACKAGE**
**Reference Manual**

**92045-90001** **February 1982**

**Update No. ________**
**(If Applicable)**

We welcome your evaluation of this manual. Your comments and suggestions help us improve our publications. Please use additional pages if necessary.

---

**FROM:**

**Name** ______________________________

**Company** ______________________________

**Address** ______________________________

______________________________

**Phone No.** ____________________ **Ext.** __________

FOLD FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

**BUSINESS REPLY MAIL**

**FIRST CLASS PERMIT NO. 141 CUPERTINO, CA.**

**— POSTAGE WILL BE PAID BY —**

**Hewlett-Packard Company**
Data Systems Division
11000 Wolfe Road
Cupertino, California 95014
**ATTN: Technical Marketing Dept.**

FOLD FOLD

HEWLETT PACKARD

MANUAL PART NO. 92045-90001
Printed in U.S.A. February 1982

HEWLETT-PACKARD COMPANY
Data Systems Division
11000 Wolfe Road
Cupertino, California 95014