PRIM USER'S MANUAL

Louis Gallenson, et al

University of Southern California

Prepared for:

Advanced Research Projects Agency

April 1975

I

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER ISI/TM-75-1	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER AD A009 936
4. TITLE (and Subtitle) PRIM USER'S MANUAL		5. TYPE OF REPORT & PERIOD COVERED Technical
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Louis Gallenson, Joel Goldberg, Ray Mason, Donald Oestreicher, Leroy Richardson		8. CONTRACT OR GRANT NUMBER(s) DAHC 15 72 C 0308
9. PERFORMING ORGANIZATION NAME AND ADDRESS USC/Information Sciences Institute 4676 Admiralty Way Marina del Rey, CA 90291		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS ARPA Order #2223
11. CONTROLLING OFFICE NAME AND ADDRESS Advanced Research Projects Agency 1400 Wilson Blvd. Arlington, Virginia 22209		12. REPORT DATE April 1975
		13. NUMBER OF PAGES 135
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) ----------		15. SECURITY CLASS. (of this report) -------
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE ------
16. DISTRIBUTION STATEMENT (of this Report) This document approved for public release and sale; distribution is unlimited.		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report) --------		
18. SUPPLEMENTARY NOTES --------		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) ARPANET, control memory, microprocessor, microprogramming, microprogramming language, microvisor, MLP-900, operating systems, resource sharing, TENEX, time sharing, writable control memory		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) This document is a four-part technical manual to aid the users of the Programming Research Instrument (PRIM), a major time-shared microprogramming facility which permits individual researchers to create specialized computing systems adapted to their needs. The document consists of an overview, a user's guide, and reference manuals for the General Purpose Microprogramming language (GPM) and the MLP-900 microprocessor.		

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

TO THE USER:

We have worked hard to make this technical manual as accurate and complete as possible. However, since mistakes are known to creep into even the most sincere of efforts, we would appreciate your calling to our attention any technical or typographical errors, omissions, inconsistencies, or ambiguities you notice while perusing it. Postage-paid preaddressed reply cards have been included below for your convenience. Please jot down the problem and the page on which it occurs, tear out the card, and drop it in the mail.

Thank you.

The Publications Group at ISI

ERROR	PAGE
prim	

ERROR	PAGE
prim	

ISI/TM-75-1
April 1975

ARPA ORDER NO. 2331

PRIM

User's Manual

Louis Gallenson
Joel Goldberg
Ray Masor
Donald Oestreicher
Leroy Richardson

INFORMATION SCIENCES INSTITUTE

UNIVERSITY OF SOUTHERN CALIFORNIA

4676 Admiralty Way/Marina del Rey/California 90291
(213) 822-1511

THIS RESEARCH IS SUPPORTED BY THE ADVANCED RESEARCH PROJECTS AGENCY UNDER CONTRACT NO DAHC15 72 C 0308, ARPA ORDER NO 2223. PROGRAM CODE NO 3D30 AND 3P10

ACKNOWLEDGMENT

The members of the PRIM project would like to gratefully acknowledge the interest, cooperation, and support they received from the STANDARD Computer Corporation, manufacturers of the MLP-900 microprocessor. Special thanks are due to the firm's president, Mr. James P. Hynes, whose many efforts on behalf of the project were and are very much appreciated

CONTENTS

FIGURES

Preceding page blank

TABLES

1. OVERVIEW

The Programming Research Instrument (PRIM) project has created a fully protected experimental computing environment with continuous multiuser access. The I/O and user interaction facilities are provided by the TENEX time-sharing system[1,2] of Bolt Beranek and Newman Inc. (BBN). The computation facilities are provided by the MLP-900, a flexible, powerful microprogrammed processor developed by the STANDARD Computer Corporation[3-6]. PRIM's multiaccess system allows each researcher to create his own specialized computing engine that he can change and adapt to his specific needs.

PRIM is implemented on a system that can be viewed on four levels: hardware, software, user interpreter/emulator, and user target program.

The PRIM hardware and software together provide a working environment in which the user can implement his own computer in microcode and run that computer in his target program environment.

1.1 HARDWARE

The hardware system is based on two processors: the Digital Equipment Corporation PDP-10 and the STANDARD Computer Corporation MLP-900 prototype processor. The PDP-10 and MLP-900 share memory as dual processors; the MLP-900 is a device on the PDP-10 I/O bus (see Figure 1.1).

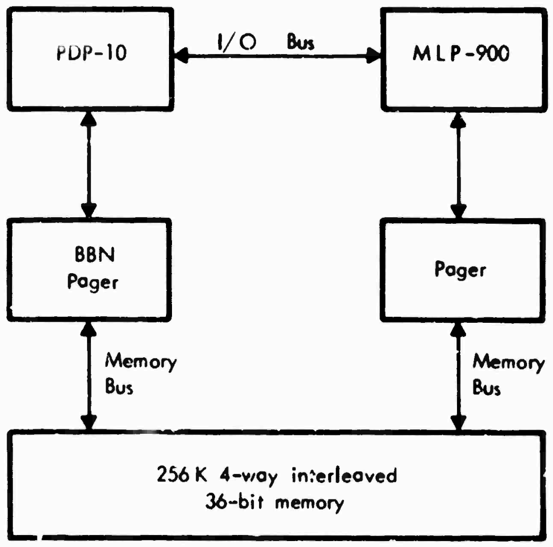


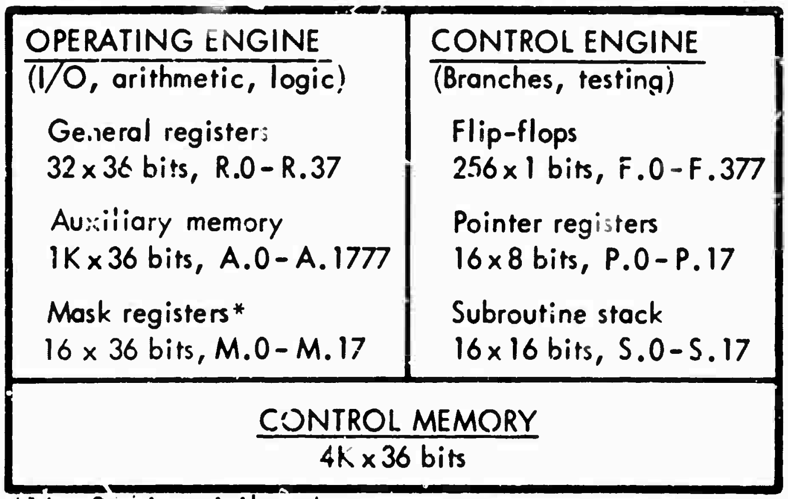
Figure 1.1 Basic PRIM configuration

1.1.1 PDP-10

The PDP-10, connected to the ARPANET, runs under the BBN TENEX time-sharing system on a paged virtual memory. The processor has 256K words of 36-bit memory. The I/O performed by TENEX includes file, terminal, and network handling, swapping, and all other accesses to peripheral devices.

1.1.2 MLP-900

The MLP-900 is a vertical-word microprogrammed processor (microprocessor) that runs synchronously with a 4-MHz clock. It is characterized by two parallel computing engines: the Operating Engine (OE), which performs arithmetic operations, and the Control Engine (CE), which performs control operations (see Figure 1.2). The OE contains 32 36-bit general-purpose registers for operands and 32 36-bit mask registers to specify operand fields. A 1K 36-bit high-speed auxiliary memory is associated with the OE. The CE contains 256 state flip-flops, a 16-word hardware subroutine return stack, and 16 8-bit pointer registers.



*16 x 36 bits privileged

Figure 1.2 MLP-900 configuration

The MLP-900 is accessible only through the PDP-10 as outlined above (i.e., the I/O bus and shared memory); no provisions have been made for direct connection of peripheral devices.

The speed and power of the MLP-900 may be conveniently understood in terms of its ability to emulate better known machines. Emulation of the IBM 360 machine language instructions would produce an estimated execution rate as low as half that of an IBM 360/65. A PDP-10 can be emulated at a rate approximating a KA10 CPU. However, in two high-level languages investigated, an estimated order-of-magnitude increase in execution rate of source statements can be attained by implementing those languages directly rather than emulating an intermediate target machine.

The MLP-900 is particularly well suited for investigating direct language emulation, since it has the flexibility of a large (4096 word x 36 bit) writable control memory. In addition, through the use of special-purpose hardware language boards, the basic architecture of the MLP-900 can be conveniently expanded and its speed increased for specialized language-processing tasks.

The environment of the MLP-900 further promotes easy experimentation and user access. The TENEX host system will provide not only complete I/O handling for the MLP-900 but also a developed (and in many cases familiar) environment for users. Together these two advanced systems should provide a most powerful and useful tool.

1.2 SOFTWARE

The PRIM software consists of the MLP-900 Microprogramming Supervisor (Microvisor), the TENEX Driver for the MLP-900, the TENEX MLP-EXEC program, which provides interactive access to PRIM for a user at a TENEX terminal, and a compiler for the General Purpose Microprogramming Language (GPM).

1.2.1 GPM and the GPM Compiler

GPM is a high-level machine-oriented language designed explicitly for the MLP-900. As a high-level language, GPM offers a block structure and statement syntax similar to PL/1 or Algol. The specific statement types defined in GPM are generalizations of the actual MLP-900 "MINIFLOW" instruction set; constructs completely foreign to MINIFLOW (e.g., multiplication) do not appear in GPM. As a simple example of MINIFLOW generalization, consider that the result of a GEAR (GEneral ARithmetic) ministep may be shifted left or right only by 0, 1, 2, 4, 6, 8, 12, or 16 bits; in GPM, any shift amount may be specified, and the compiler will generate multiple shifts as required.

As the production language for the MLP-900, GPM is constrained to satisfy many of the usual requirements of an assembly language. First, there is a well defined subset of GPM statements that produces exactly one ministep per statement; the subset is capable of generating all possible ministeps. Second, multi-ministep statements do not generate implicit side effects; for example, a complex arithmetic assignment which requires a temporary register for an intermediate result will generate a compile-time error unless the program has explicitly declared some register to be available as a temporary.

1.2.2 MLP-900 Microvisor

The MLP-900 Microprogram Supervisor (Microvisor) is a small, fully protected resident system that controls the MLP-900 and its communication with the PDP-10. It loads and unloads the user's MLP-900 context upon command from the PDP-10, supports paging of the user target program, protects main memory and the rest of the PDP-10 system from user interpreter errors, and provides that interpreter with some services, such as an extended subroutine stack and calls for external communication.

1.2.3 PDP-10 Support Programs

The PDP-10 TENEX software for support of the MLP-900 consists of a Driver to control communication with--and sharing of--the MLP-900, and a subsystem (MLP-EXEC) to allow easy interactive user access to the MLP-900.

MLP-EXEC provides an environment in which the user at a terminal can compile, load, execute, and debug MLP-900 microcode in a manner similar to that used for debugging programs on the PDP-10. In addition, he can create and debug target programs and environments, although these tools must be provided at a very primitive level, since MLP-EXEC cannot know the nature of the target environment.

The MLP-900 Driver is the extension in TENEX of the Microvisor; all communication with the MLP-900 goes through the Driver. While new microcode "machines" can be designed and debugged under the MLP-EXEC, completed ones will work directly through their own terminal subsystems, which will communicate directly with the Driver.

1.2.4 User's Interpreter and Target Program

The user's interpreter is a program written in GPM to run on the MLP-900; it defines a (re-entrant) MLP-900 control memory image. This image and all the nonprivileged registers and flip-flops within the MLP-900 comprise the MLP-900 context; users' contexts are loaded and unloaded as the MLP Driver shares the MLP among different users.

The context defines the user's interpreter (or target machine) and operates upon the user target program in a totally arbitrary way. The only constraint upon the target program is that it fit into a 512K 36-bit (virtual) memory space.

2. USER'S GUIDE

2.1 INTRODUCTION

As explained in Section 1.2 of the previous chapter, the PRIM software consists of the MLP-900 Microprogram Supervisor (Microvisor), the TENEX Driver for the MLP-900, and the TENEX MLP-EXEC program, which provides interactive access to the MLP-900 for a user at a TENEX terminal. This chapter provides a detailed guide to the PRIM software (with the exception of the GPM compiler, which is discussed separately in Chapter 3). Section 2.2 describes MLP-EXEC and the facilities it provides to the user for constructing, running, and debugging both MLP-900 microcode and the associated target system. Section 2.3 describes the MLP-900 Microvisor and the services it provides, as well as the restrictions it places on that microcode. Section 2.4 describes the MLP Driver and the TENEX JSYS's required to communicate with it, which comprise the interface to the MLP-900 used by MLP-EXEC. This section will be of direct interest only to those who wish to replace MLP-EXEC with another subsystem of their own design.

2.2 MLP-EXEC

MLP-EXEC is a TENEX subsystem that allows interactive access to the MLP-900 from a user at a terminal. MLP-EXEC is modelled after the TENEX Exec in its general command format; the specific commands are designed to allow user access to all phases of MLP-900 operation.

2.2.1 Access to MLP-EXEC

As a TENEX subsystem, MLP-EXEC is entered by typing "MLP" to the TENEX Exec program:

```
@MLP
MLP EXEC 1.0
>
```

The MLP-EXEC "prompt" character, ">", signals the user to enter a command. Upon completion of command execution, MLP-EXEC prompts again.

Commands to MLP-EXEC can speci any of several types of actions:

- Control the loading, execution, and debugging of the user's MLP context, a structure which includes both the MLP-900 control memory and all the (nonprivileged) MLP-900 registers. All commands specific to the context are prefixed by a period (.). The context defines the target machine and, in general, its current state.

- Control the loading and debugging of the target system, a 256K virtual memory in which the target machine (as defined by the MLP context) runs. All commands to control the target system are prefixed by a slash (/); in general, these commands are identical to the TENEX Exec commands of the same name.

- Define the input/output files for MLP execution.

- Miscellaneous other commands, such as STATUS, QUITMLP, EXEC, and so forth.

2.2.2 Command Format

A command consists of an initial key word (or portion of a key word) followed by zero or more argument fields. MLP-EXEC prompts for each field required by the user's command. The key word and argument fields are separated from one another by the following field separator characters (separators): space, return, linefeed, tab, formfeed, vertical tab, and Escape.

Additionally, two characters (Control T and Control C) act as complete commands in themselves to control MLP execution and to provide status information on the MLP.

Command Key Words and Recognition

A key word is defined as a sequence of characters other than the separator characters and the semicolon (which is used for comments).

Like TENEX Exec commands, MLP-EXEC commands can be abbreviated to just enough characters to distinguish them from other commands. Similarly, if the abbreviated key word is terminated by an Escape, the MLP-EXEC will, upon recognition, type back the rest of the key word. If the command is not recognized, MLP-EXEC will ring the terminal bell and await additional input.

Command Editing

Certain characters serve to edit a command key word, as follows:

- Control A,
  Delete (DEL) - These backspace one character position, erasing the character from the input.

- Control X - This erases the entire word so far entered.

- Control R - This types out the word so far entered.

These characters are also used for editing command argument fields, except that DEL cannot be used for backspacing a file name argument. An argument field previously completed (i.e., followed by a field separator) cannot be edited.

Comments in Commands

The semicolon (;) is used to begin a comment; everything from the semicolon to the following return or linefeed is ignored by MLP-EXEC (but retyped by Control R).

Example:

```
>;THIS IS THE SAME AS A BLANK LINE↑R
;THIS IS THE SAME AS A BLANK LINE
>EXEC; INVOKE THE TENEX EXEC
  ISI-TENEX 1.31.1 ISI-TENEX EXEC 1.51.3
@
```

Command Termination and Confirmation

Most commands to MLP-EXEC are not executed until a confirming return or linefeed is typed. The confirmation is normally not required, however, if the character terminating the last argument field is a return or linefeed. Some commands require an additional explicit confirmation, since they change or destroy information. A few commands require no confirmation, but are executed upon recognition of the last field.

Control C. Control C is valid at any time, terminating the current operation and returning to the MLP-EXEC command level. During command input, the partial command is aborted. During MLP execution, that execution is interrupted (this is the only way to stop a looping MLP program).

Control T. Control T is valid at any time, and yields a message regarding the state of the MLP and the value of the current address register.

Example:

```
<↑T>
MLP RUNNING AT LOC 451
```

2.2.3 Commands for Control of the MLP Context

These commands begin with a period (.) to distinguish them from similar commands for the target system.

.LOAD

.LOAD prompts for a list of files to be loaded (the file or files should be the output of a GPM compilation). The files are specified as a list of file specifiers, e.g., A.B.*.BIN. The list is terminated by a Return or a Delete (Delete cancels the command).

.LOAD first clears the previous context; each file specified is then loaded into control memory. Any overlap of loaded files is ignored; any overlapped location will have as its value the last item loaded in that location. If any file specifies a starting address, then that address is retained by MLP-EXEC as the starting address for execution.

As a safety feature, any locations not loaded by any of the files are loaded with Halt ministeps. It should also be noted that control memory locations 7000 through 7755 are not part of the user's context; although these locations may be loaded with the .LOAD command, they will not be loaded into the MLP's control memory. These locations may be used to preload certain of the MLP registers; if not otherwise set, they will be set to 0. For more information, see Section 2.4 on the MLP-900 Driver.

Example:

```
>.LOAD
  GPM BINARY FILES: TEST1.BIN,<USER2>*.BIN

LOADED TEST1.BIN;5
LOADED <USER2>TEST3.BIN;4
LOADED <USER2>TEST4.BIN;4
>
```

Errors:
If one or more of the files cannot be loaded, an error message will be given, but loading will continue on the files remaining to be loaded.

.START

.START initiates MLP execution of the context, beginning at the starting address, after amending parts of the context as follows:

```
P.6 ← 2 ! STACK POINTER
S.2 ← Starting address;
S.1 ← 7200 ! ILLEGAL, to detect stack underflow
ARL.5 ← FALSE;
CE.13 ← 0;
INPW ← FALSE;
CE.12 ← 0 (77);
```

After the context is swapped into control memory, microcode execution is always initiated by a RETURN ministep.

Errors:

NO PROGRAM	No MLP context has been loaded

.CONTINUE

.CONTINUE resumes execution of the MLP context "as is" after interruption.

Errors:

NO PROGRAM	No MLP context has been loaded
NOT STARTED	

.RESET

.RESET clears the MLP context. The use of .START at this point will cause the error message "NO PROGRAM" to be typed.

.ENTRY

.ENTRY allows the user to set the starting address manually, as an octal number, or as a hexadecimal number preceded by an apostrophe (').

.RUN

.RUN prompts for the name of the GPM binary files to be run, LOADs them, and .STARTs them at the starting address of the last file loaded.

Example:

```
>.RUN
 GPM BINARY FILES: TEST.MLP
LOADED TEST.MLP;1
```

Errors:
All the errors possible under .LOAD and START are possible.

.SAVE

.SAVE prompts for the file name under which to save the current MLP context, and saves the context on the file so that it can be restored with a subsequent .GET command. Both control memory and all registers are saved.

Example:

```
>.SAVE
FILE NAME: TEST2.MLP[NEW FILE]
>
```

Errors:

```
? NO CONTEXT TO SAVE        No context has been loaded
```

.GET

.GET prompts for the name of a file which was .SAVEd, then restores the MLP context from that file. The starting address is obtained from the restored stack.

Example:

```
>.GET
 FILE: TEST.SAVE[Old version]
>
```

Errors:

FILE NOT GETTABLE	The file was not originally saved in such a way that it can be restored into the MLP context using MLP-EXEC.

.DDT

.DDT invokes MLP DDT to let the user examine and change the MLP context currently loaded. MLP DDT is described separately later in this section.

ARSTATUS

Reports all of the MLP AR's associated with external events. For each active event, the associated AR(s) are specified by an 8-bit mask, with the most significant bit (200) corresponding to F.130 and the least significant bit (001) to F.137.

.EOF

Sets the AR mask associated with the end-of-file condition (or any MLP input channel). The mask is specified as an octal number less than 256.

.INPUT

Sets the AR mask associated with the input-ready condition for a given MLP input channel. The AR(s) is sent to the MLP-900 whenever that channel's input buffer becomes nonempty, or whenever the buffer remains nonempty after a byte is read. (AR(s) is sent once per byte.)

2.2.4 Commands for Control of the Target System

These commands begin with a slash (/) to distinguish them from similar commands for the MLP context.

/LOAD

/LOAD runs the standard TENEX loader to load relocatable binary file(s) into the target system address space. Descriptions of the loader, which is identical to the TENEX Exec "LOADER" command, can be found in Refs. 7 and 8.

Example:
```
        >/LOAD
        */S
        *TEST.MLP
        LOADER 3+3K CORE
        MAX 400 WORDS FREE
        EXIT
        ↑C
        >
```

/GET

/GET clears the current target, then does a GET into the target system address space of a core image saved by SAVE or SSAVE. It is identical to the TENEX Exec "GET" command.

Example:
```
        >/GET
        FILE: TEST.SAV[Old version]
        >
```

/MERGE

/MERGE is similar to GET but does not require initial clearing of the target system. It is identical to the TENEX Exec "MERGE" command.

Example:
```
        >/MERGE
        FILE: TEST3.SAV[Old version]
        >
```

/DDT

/DDT invokes the TENEX DDT package on the target system. It is identical to the TENEX Exec "DDT" command.

/SAVE, /SSAVE

These commands SAVE or SSAVE the core image (except DDT if invoked) on a file. SSAVE is reserved for shared files. These commands differ from TENEX Exec only in saving the entire address space automatically.

Example:
```
        >/SAVE
         TARGET SPACE ON FILE: FOO.SAV [New version]
        >
```

/RESET

/RESET clears the target system. It is identical to the TENEX Exec "RESET" command; it also causes the context to become "NOT STARTED."

/MEMSTAT

/MEMSTAT gives a page-by-page indication of the state of the target system. It is identical to the TENEX Exec "MEMSTAT" command.

2.2.5 Commands For File Input/Output

INPUT, OUTPUT, APPEND

These commands establish a TENEX file for reading, writing, or appending (sequential mode only) by the MLP program on a given channel. Arguments are file name, channel number, and byte size for opening a file. Files can be independently assigned to each of the 16 input and 16 output channels available (channels are numbered 0 through 15).

Example:
```
>INPUT
 FILE: A.B [Old version]
 ON MLP CHANNEL: 0
 WITH BYTE SIZE: 7
>
```

Each file is opened ("thawed") so that reading and writing may be done to the same file simultaneously. If a file is already open on the channel, the MLP-EXEC, after additional confirmation, closes and releases the old file.

CLOSINPUT, CLOSOUTPUT

These commands close a channel; each requires an explicit confirmation.

Example:
```
>CLOSINPUT
 INPUT CHANNEL NUMBER: 4
 CLOSING A.B;5
 [CONFIRM]
>
```

FILESTATUS

This command types the current assignment of files to MLP channels (and to TENEX JFN's).

Example:

```
>FILESTATUS
CHAN:    JFN:     FILE:     POSITION:
INPUT FILES:
0        5        A.B;4     382
OUTPUT FILES:
0        6        A.B;5     1  0
>
```

2.2.6 Other Commands

EXEC

This command loads and starts an inferior TENEX Exec, without affecting the state of the MLP context or target system. The user may return to MLP-EXEC by executing a QUIT from the TENEX Exec.

Example:

```
>EXEC
ISI-TENEX 1.51.0 ISI TENEX EXEC 1.77.6
@;miscellaneous stuff that the user wants to do...
.
.
.
@QUIT
>
```

QUITMLP

This command exits from the MLP-EXEC. The MLP context and target system are cleared before exiting.

LOGOUT

This command clears the context and the target system and logs out the job.

? (The Help Command)

This command lists all the MLP-EXEC commands available.

STATUS

This command prints a brief summary of the state of both context and target system, e.g.,

```
>STATUS
CONTEXT LOADED, ENTRY ADDRESS 0
NO TARGET SYSTEM LOADED
```

(The context can be run without a target system; the first memory reference, if any, will cause termination due to an illegal memory reference.)

2.2.7 MLP DDT

MLP DDT, entered by the .DDT command from MLP-EXEC, allows the user to examine and modify his context (the corresponding facility for the target space is TENEX DDT, invoked by the /DDT command).

Examining MLP Locations

MLP locations are of two kinds: control memory and register locations. Control memory locations are specified by numeric addresses, e.g., 172, or 'A39.* The registers are specified by symbolic addresses, e.g., P.0 or R.36.

To examine a specific location, type its address, followed by a slash (/).

Example:

```
74/
74 0 GEAR 2 360 127 27 R.27←R.27(M.17)*; P.4/
P.4/ 5
```

In this example the user examined control memory location 74 (octal). After the GPM listing-format typeout, the user asked to see the contents of register P.4; P.4 was typed out as an octal number.

Examining Consecutive Locations

After a location is examined, the character linefeed may be used to examine the next location following; the character between " and " may be typed to examine the location preceding.

Example:

```
P.4/    P.4/    5
P.5/     27
M.5/
M.5/     144    ↑
M.4/     67
```

Changing Typeout Modes

The typeout mode is initially octal. To change to hexadecimal, type ESC (the escape key) X; to change back to octal, type ESC O (letter O).

Modification of a Location

The location last examined may be modified. Two methods are available for modification.

---

* A leading apostrophe indicates a hexadecimal value on input.

Direct Modification

An open location (including a control memory location) may be set to a numeric value by typing the value followed by return, linefeed, or "↑". If linefeed or "↑" is typed, the next following or preceding location will be typed out and opened for modification. The new value may be entered in either octal or hexadecimal; as noted earlier, hexadecimal values are indicated by typing a leading apostrophe ('). (Note that if the numeric value given is not a valid octal or hexadecimal integer, a question mark (?) is typed and the modification is not made.)

If the location being modified has fewer significant bits than the number supplied, the least significant bits of the number become the new value.

GPM Modification

To change a control memory address with the aid of GPM, proceed as follows:

- Examine the location (this opens it for modification).
- Type "*": The prompt "GPM:" is made on the next line.
- Type in the new statement (or statements).
- Terminate the change with Control Z.

The GPM statement(s) are compiled and loaded beginning at the currently open location. (Note that more than one consecutive location can be changed in this way; if ORIGIN statements are included, noncontiguous areas of control memory may be changed.)

Before typing Control Z, the change can be aborted by typing Control Q.

Breakpoints

A single breakpoint can be set in control memory, target memory, or both. To set a control memory breakpoint, type

```
<address> ESC B
```

where "<address>" is the control memory address. To clear it, type

```
ESC B
```

(no address).

To set a target memory breakpoint, type

```
<address> ESC T
```

To clear it, type

```
ESC T
```

Action at a Breakpoint

A control memory breakpoint will cause execution of the specified location to halt the MLP and to type out the address of the location at which execution so halts. A target memory breakpoint will cause a similar halt upon any reference to the specified target memory location. The control memory address of the FOP or SAD ministep causing the reference will be the interrupted MLP program counter (PC) value.

Single-Step Execution

To execute a single control memory instruction, type

```
<address> ESC S
```

or simply

```
ESC S
```

(The current location is used for <address>.) After each step the address and contents of the new control memory location are typed out and opened for modification.

Resuming Execution

Normal MLP execution may be resumed by typing

```
<address> ESC P
```

or

```
ESC P
```

2.3 THE MLP-900 MICROPROGRAM SUPERVISOR

The MLP-900 Microprogram Supervisor (Microvisor) performs the usual functions expected of an operating system, except that it is written in microcode and supervises the execution of microcode. The Microvisor interacts only with the user microcode and the TENEX MLP Driver; it does not provide any facilities for--or impose any restrictions upon--the user target system.

User microcode always runs in user mode on the MLP-900; it is subject to the restrictions imposed by the MLP-900 hardware, explained in detail throughout Chapter 4 and summarized here:

- The BLOT ministeps which reference control memory (RCM, WCM, and WBP) are prohibited in user mode. If attempted in user mode, they generate a Supervisor Facilities Action Request (SUPVF AR). User microcode is therefore incapable of modifying itself.

- Certain registers are privileged and can be modified only in supervisor state; an attempt to modify one while in the user state generates a SUPVF AR. The privileged registers include the (paging) translator memory (XLATOR.777), half of the CE miscellaneous registers (MISC.20 thru MISC.37), and seven bytes of the CE flip-flops. These flip-flops and registers control the main memory paging, the I/O bus communication with the PDP-10, the internal AR (interrupt) system, and other critical functions.

- User mode microcode may not branch to a supervisor mode location, except for designated supervisor entry points; an attempt to do so results in a PROT (Protection) AR.

2.3.1 Control Memory

The Microvisor occupies control memory from 7000 to 7755 (octal), inclusive; these locations are not available for user microcode. This includes all the locations associated with AR's of the first four priority levels; all such AR's are handled entirely by the Microvisor. Locations 7756 through 7777 (octal) are associated with the lowest AR priority level (ARL.5) and target system interrupts; these locations are loaded as part the user microcode context.

2.3.2 Main Memory

All main memory references by the user microcode are mapped into the target system virtual memory. Page faults are handled by the Microvisor and the TENEX MLP Driver in the same way that TENEX handles them directly for TENEX processes.

2.3.3 Extended Stack

The Microvisor provides for automatic storing and reloading of the MLP subroutine stack-upon-stack overflow and underflow; no distinction is made between occurrences in user mode and supervisor mode. The extended stack is stored in the last page of auxiliary memory (A.1400 through A.1777), using successive 16-word blocks as needed. The four most significant bits of P.6, the stack pointer, are used as the extended stack block index: 0 selects A.1400-1417, 1 selects A.1420-1437, ... 15 selects A.1760-1777.

Upon stack overflow, the thirteen words at the bottom of the stack (S.1 through S.15) are stored in the parallel words of the current stack extension block and the stack and its pointer adjusted appropriately. Upon underflow, thirteen words are reloaded and the stack again adjusted. Words 0, 14, and 15 of each extension block are neither used nor destroyed; they may be used for other purposes.

An "extended stack overflow" fault is generated, and the microcode halted, whenever a stack overflow uses block zero. There is no provision for detecting extended stack underflow; if desired, underflow protection may be provided by planting an error address in the stack. The maximum amount of stack space available, with P.6 initially set to one, is 209 words (15 stacked blocks of 13 plus 14 more in the actual stack. The minimum amount available, with P.6 initially set to 241 (block 15, word 1), is the 14 words of actual stack; auxiliary memory will not be used except in the case of an (erroneous) overflow or underflow of the stack. Intermediate initial values in P.6 will allow other sizes of effective stack--and commit appropriate amounts of auxiliary memory to the maintenance of that stack. The user's stack requirements must allow not only for the maximum nesting in both main and AR code, but also for four levels of Microvisor stacking.

The nth entry from the top of the stack, 0 <= n < 15 (octal), is located as follows (all numbers are octal):

```
If (P.6 and 17) > n
then S.0 @ (P.6 - n)
else A.1400 @ (P.6 + 15 - n)
```

2.3.4 Microvisor Calls

Microvisor functions are available to the microcode via calls to designated Microvisor entry points. Arguments are passed in register R.37, and R.36 when needed; replies are received in the same registers. The entry names and their locations are known by the GPM compiler; entry names are of the form "MLP.xxx".

CALL MLP.STOP no arguments
Terminates microcode execution and informs the Driver; if continued, execution will resume at the next ministep.

CALL MLP.PUT R.37 contains the output line number.
R.36 contains the data
Transmits the data to the Driver and returns immediately. Any error will result in an asynchronous halt of the microcode at some subsequent point.

CALL MLP.GET R.37 contains the input line number.
Gets a byte of data from the TENEX Driver and returns it in R.36. Any error will result in an immediate halt of the microcode; optionally end-of-file is signaled via a user-level AR.

2.3.5 Communication with TENEX

The microcode can perform I/O on TENEX files through the two Microvisor calls which transmit data to and from the PDP-10. A maximum of sixteen lines are available for input (to the MLP), and sixteen for output. Each Microvisor call transmits one byte of (up to) 36 data bits. Each line actually used must first be defined at the TENEX end (e.g., via the INPUT and OUTPUT commands in MLP-EXEC); the use of an undefined line, or an error on a defined line, causes execution to terminate due to a "Communication Error."

Since I/O is done through the TENEX Driver, it is quite expensive; large data transfers are better done via the shared target system memory.

When the microcode is halted while in an input-wait state, F.162, the input-wait flip-flop, is set; clearing the flip-flop before continuing execution will turn the interrupted GET into a null operation. Conversely, setting the flip-flop will cause an extra GET on the line specified in R.37.

2.3.6 User Microcode Action Requests

The MLP AR's covered by ARL.5 (F.130 through F.137), plus the target system interrupt AR, are entirely at the disposal of the user. The control memory locations (7756 through 7777) and the flip-flops involved are all part of the user MLP context.

User AR's can be generated by the user language board (the null language board does not generate any AR's), by the tracing mechanism, and by direct user ministeps. In addition, the Microvisor will pass an AR to the microcode when an appropriate external event (such as end-of-file) occurs; the particular AR associated with a given event is determined by the AR masks in the MLP context.

Tracing of a Microvisor call results in a total of three trace AR's: the first immediately after the call--or immediately before the first Microvisor ministep--the second and third upon exit from the call, one while still in the Microvisor, and one just before the continuation ministep.

2.4 THE TENEX MLP-900 DRIVER

Access to the MLP-900 from a TENEX process is accomplished via the MLP Driver in TENEX. Communication with the driver is done through a series of JSYS's which mimic (roughly) the JSYS's for subsidiary fork control (see Chapter 6 of the ***TENEX JSYS Manual***). The two principal elements involved in creating and running the MLP are the MLP context (the user microcode together with all the MLP registers) and the target system upon which the context is to operate. The calling process must build both before establishing access to the MLP.

Table 2.1
MLP CONTEXT

Relative Location	Contents
0	Control memory location 0
1	Control memory location 1
...	...
6777	Control memory location 6777
7000	R.0
7001	R.1
...	...
7037	R.37
7040	M.0
...	...
7057	M.17
7060	MISC.0
...	...
7073	MISC.13 (an unimplemented register)
7074	MISC.36 (Target Address Comparand)
7075	MISC.37 (Control Memory Address Comparand)
7076	MISC.16 (VAR)
7077	MISC.17 (MDR)
7100	(CE.0, CE.1), right justified
7101	(CE.2, CE.3)
...	...
7157	(CE.136, CE.137) or S.17
7160-7177	Not assigned
7200	JFN for output line #0
...	...
7217	JFN for output line #17
7220	JFN for input line #0
...	...
7237	JFN for input line #17
7240	AR mask for end-of-file
7241-7257	Other AR masks
7260-7277	Input ready AR masks for lines #0-#17
7300-7677	Internal Driver information
7700-7755	Not assigned
7756	Control memory location 7756
...	...
7777	Control memory location 7777
10000	A.0
...	...
11777	A.1777

2.4.1 MLP-900 Context

The context is a structure that contains all the data necessary to load the MLP and begin (or resume) execution of the desired microcode. It includes not only an image of the MLP-900 control memory, but also the internal MLP-900 registers and some cells used by the Driver to implement MLP-900 communication with the PDP-10.

The context is 10 memory pages (5120 words) long, and must begin on a page boundary in the caller's address space. Its internal form is shown in Table 2.1.

Within the miscellaneous registers, MISC.36 and MISC.37 are mapped into the context in place of MISC.14 and MISC.15, which do not exist. The two comparand registers, although privileged, are loaded as part of the context, as are the two compare arming flip-flops, F.160 and F.161, and the input-wait flip-flop, F.162. The microcode, however, cannot affect either the comparands or the flip-flops.

Each of the AR masks consists of an eight-bit right-justified mask which is OR'ed into the user AR byte (CE.13) by the Microvisor when the given event occurs. If the mask is zero, the microcode cannot detect the condition.

Note that control memory locations 7000 to 7755 are occupied by the Microvisor and are therefore not considered part of the user context.

The output and input JFN's are used for the MLP-900/PDP-10 communication available to the user microcode. When the microcode transmits a word to the PDP-10 over a given line, the driver effectively does a BOUT of the received data to the selected output JFN; similarly, when the microcode requests a word from the PDP-10 over a given line, the driver does a BIN using the selected input JFN.

The JFN's can be any usable JFN except 0, which is used to terminate MLP execution when referenced.

Files must be opened (and positioned if necessary) before MLP execution begins; any file error will terminate MLP execution.

2.4.2 MLP-900 Target System

The target system is the memory upon which the MLP context is to operate. It is defined as a TENEX fork (or process), either the caller or a subsidiary fork established solely for this purpose. Typically, the target system fork will never be started on the PDP-10; it exists to define an address space for MLP execution. The target fork AC's are mapped into locations 0 through 17 of the target memory as seen by the MLP.*

----

* For the convenience of the reader, the presentation of the commands that follow is intended to duplicate the format of the *TENEX User's Manual*[8].

CMLP

Creates MLP context and target system.

ACCEPTS IN	1: the pointer to the MLP context in the caller's address space.
	2: a fork handle for the target system.

CMLP

RETURNS	+1: if unsuccessful, error number in 1
	+2: if successful, MLP handle in 1.

The MLP handle returned is used in succeeding SMLP, HMLP, and RMLPS calls; it remains valid until killed by a KMLP call. The context and the target system are bound to the MLP until the caller executes a subsequent KMLP on the returned handle. Any attempt to re-map context pages or kill the target system fork will yield undefined results.

CMLP ERRORS:

CMLPX1:	context not on page boundary
CMLPX2:	MLP not available
FRKHX1:	illegal fork handle
FRKHX2:	cannot manipulate a superior fork
FRKHX3:	cannot reference multiple forks

KMLP

Kills MLP

ACCEPTS IN 1: MLP handle

KMLP

RETURNS +1: always

Kills the MLP association established by CMLP, releasing the binding of context and target system.

Generates an illegal instruction pseudo-interrupt on error conditions listed below.

KMLP ERRORS:

MLPX1:	invalid MLP handle

IMLP

Interrupt MLP

ACCEPTS IN	1: MLP Handle 2: AR Mask

IMLP

RETURNS	+1: Always

Passes the indicated AR's to the microcode. B28 sets F.130, B29 sets F.131, ... B35 sets F.137. If the microcode is halted, the bits are set in the memory image of the context.

Generates illegal instruction pseudo-interrupt on error conditions listed below.

IMLP ERRORS:

MLPX1:	Invalid MLP handle

SMLP

Starts (or resumes) MLP execution.

ACCEPTS IN	1: MLP handle

SMLP

RETURNS	+1: always

Causes the context bound to the MLP handle to be loaded into the MLP-900 and microcode execution to begin (or resume). The Microvisor passes control to the context microcode via the BORE (Return) ministep; therefore, the start/resume address is defined by the value of P.6 and the appropriate stack word in the context. It does nothing if MLP already started.

Execution of the context microcode continues until either the microcode halts (voluntarily or due to a fault) or the caller does an HMLP; upon termination of execution, the caller is sent a pseudo-interrupt on channel 23. Between an SMLP and the subsequent termination of execution detected by the pseudo-interrupt routine or by a RMLPS--the context "belongs" to the MLP and the Driver; any attempt to read or modify it is invalid.

Generates an illegal instruction pseudo-interrupt on error conditions listed below.

SMLP ERRORS:

MLPX1:	invalid MLP handle

HMLP

Halts MLP execution
ACCEPTS IN 1: MLP handle

HMLP

RETURNS +1: always

Terminates MLP-900 execution of the context microcode. Does nothing if the context is already halted or was not started.

Generates illegal instruction pseudo-interrupt on error conditions listed below.

HMLP ERRORS:
MLPX1: invalid MLP handle

RMLPS

Reads MLP status.

ACCEPTS IN	1: MLP handle

RMLPS

RETURNS	+1: always, with status word in 1, execution time (in milliseconds) in 2.

The MLP status word consists of a state code in the left half and the microcode program counter value in the right half (see Table 2.2).

TABLE 2.2
MLP STATES

Code (Octal)	Status	Context
-1	Unrecoverable Driver Error Stop(*)	Valid
0	Running	Invalid
1	I/O Wait	Invalid
2	Voluntary Termination (CALL STOP by the microcode)	Valid
4	Target System Address Compare Stop	Valid
5	Control Memory Address Compare Stop	Valid
6	Supervisor Facility Violation Fault	Valid
7	Protection Violation Fault	Valid
10	Extended Stack Overflow Fault	Valid
11	Communication Fault	Valid
12	Target System Memory Reference Fault	Valid
13	"Recoverable" MLP Error Stop(*)	Valid

The validity of the context applies to the image of the context in the caller's address space. When it is valid, it may be inspected and/or modified arbitrarily.

In the cases marked (*), the Driver has also printed a message on its primary output file. If an unrecoverable error, the Driver has also been killed, and the MLP handle is no longer valid. This represents a hardware or system software failure which should be reported to system personnel.

RMLPS ERRORS:

MLPX1:	invalid MLP handle

3. GENERAL PURPOSE MICROPROGRAMMING LANGUAGE REFERENCE MANUAL

3.1 INTRODUCTION

The General Purpose Microprogramming Language (GPM), is [illegible]-level language developed by the PRIM project as a machine-dependent microprogramming language for the MLP-900. It contains many special-purpose language forms reflecting actual MLP-900 hardware features.

The assembler philosophy underlies the design of GPM, which allows the programmer to create any instruction sequence and requires no run-time support system, although syntactic block structure and high-level control structures are provided to assist the programmer. GPM is the primary language for the MLP-900 (no assembly language is provided) and, as such, was designed to be used by both the diagnostic programmer and the researcher.

3.2 BASIC LANGUAGE SYMBOLS

GPM programs are composed of five basic symbols or syntactic entities. They are as follows:

- Identifiers (id)
- Reserved identifiers
- Numbers (number)
- Blanks
- Nonalphanumeric characters

3.2.1 Identifiers

```
id ::=
  . word | word | id . subid

subid ::=
  word | number

word ::=
  alpha | word alpha | word digit

number ::=
  digit | number digit

digit ::=
  0 | 1 | ... | 6 | 7

alpha ::=
  & | 9 | A | B | ... | Y | Z |
  a | b | ... | y | z
```

Preceding page blank

An identifier is a string of words (alphanumeric strings) or numbers separated by periods. The first field must not be a number, and words cannot begin with a digit (0 - 7). The last number (all-numeric) field is referred to as the index; it is used extensively for reserved identifiers (e.g., R.0 is general register 0 and R.17 is general register 17). Nonreserved identifiers are used in four places in GPM:

- TITLE statement
- EQUATE statement
- Block name
- Labels

3.2.2 Reserved Identifiers

Reserved identifiers have the same syntax as identifiers and include all special symbols in GPM. In the case of indexed reserved identifiers, they are all assumed to have zero origin and will be referred to in this manual by their upper bound. All reserved identifiers are upper-case.

Example:
There are 32 general registers (R.0 - R.37). R.37 will appear in all descriptions to represent

R.0 | R.1 | ... | R.36 | R.37

Reserved identifiers cannot be used as labels or as the title. A complete list of all reserved identifiers is given in Appendix A.

3.2.3 Numbers

All numbers in GPM, including identifier index fields, are octal. Y.1973 is two identifiers, i.e., Y.1 and 973. The numerals 8 and 9 are letters.

3.2.4 Blanks

All nonprinting characters (space, tab, linefeed, carriage return, and formfeed) are blanks. Blanks separate numbers and identifiers; otherwise they have no syntactic or semantic function. There is one additional blank character, an arbitrary string starting and ending with a percent sign (%). This is not the preferred method of comment, as will be treated in detail in the discussion of the GPM listing format in Appendix B.

3.2.5 Nonalphanumeric Characters

All nonalphanumeric characters are reserved. Except for the period (.), they are all self-terminating and cannot appear as part of any symbol.

3.2.6 Examples of Basic Symbols

The string R.1 ABC*1248X 12A.B;C.3.4.X is interpreted as

R.1	Reserved identifier; index = 1
ABC	Identifier
*	Character
124	Number
8X	Identifier
12	Number
A.B	Identifier
;	Character
C.3.4.X	Identifier; index = 4

3.3 PROGRAM STRUCTURE

```
program ::=
  TITLE id body closing

body ::=
  declarationlist ; statementlist | statementlist

declarationlist ::=
  declaration | declarationlist ; declaration

statementlist ::=
  statement | statementlist ; statement
```

A GPM program starts with a title declaration. The title must be a nonreserved identifier. The body of the program has two parts: a declaration list and statement list. The program ends with a closing or FINISH statement.

3.3.1 Declarations

```
declaration ::=
  pseudostatement | TEMPORARY rlist |
  EQUATE symbol symbol | EQUATE symbol symbol number |
  DEFAULT TEST mode | DEFAULT CLEAR mode |
  DEFAULT MASK M.17

rlist ::=
  R.37 | rlist R.37 | M.17 | rlist M.17

mode ::=
  MODE TRUE | MODE FALSE
```

The declarations define conditions that will be active for the scope of the body in which they are made. They fall into two general groups: The first group (EQUATEs) defines new symbols, and the second (TEMPORARY and DEFAULTs) defines conditions relative to operating engine compilation. Pseudostatements are listed under declarations because they may appear anywhere in the program. They are discussed in Section 3.4.

EQUATE

There are two forms of the EQUATE statement. The first takes two symbols and equates the first to the second. For example, after the declaration EQUATE PC R.3; every occurrence of PC within the scope of the declaration will be interpreted as R.3. The following are legal EQUATE statements:

```
EQUATE INDEX R.6;

EQUATE MINUS.ONE 777777777777;

EQUATE EQ EQUATE;

EQ INFINITE.LOOP.START DO.BEGIN;
```

The second EQUATE form is used to equate blocks of indexed names. For example, after the declaration EQUATE AC.0 R.10 10; every occurrence of AC.0 through AC.7 within the scope of the declaration will be interpreted as R.10 through R.17, respectively.

TEMPORARY

The TEMPORARY declaration declares general registers or mask registers that may be used as temporaries by the code generators. This declaration allows more complicated arithmetic operations and data transfers to be compiled.

DEFAULT

Three conditions associated with arithmetic expressions will be fairly constant over a large number of statements. These may be set by the DEFAULT statement. They are as follows:

- Test Mode. When this is true, no general registers are stored into, though the operations are done and the appropriate status flip-flops are modified. The initial value is FALSE.

- Mask. The mask register defines the active parts of the registers for arithmetic expression evaluation. The initial value is M.0.

- Clear Mode. When this is true, the parts of the register that do not enter into the calculation, as controlled by the mask register value, are cleared to zero. The initial value is FALSE.

3.3.2 Statements

```
statement ::=
  id : statement I substatement
```

The statement types are discussed in detail in Section 3.5. All statements may be tagged by one or more identifiers, which can be used as program labels. Reserved identifiers, numbers, and nonalphanumeric characters may not be used as program labels.

3.3.3 Closing

```
closing :: =
  FINISH I FINISH id
```

The closing statement of a GPM program is the reserved word FINISH, optionally followed by an identifier. This identifier, if present, specifies the starting label of the program to the MLP loader.

3.4 PSEUDOSTATEMENTS

```
pseudostatement ::=
  ORIGIN number I COMMENT (any string not containing a ;) I
  outputcontrol

outputcontrol ::=
  PRINTON I PRINTOFF I outputtype mode

outputtype ::=
  HEXADECIMAL.CODE I NORMAL.CODE I LABEL.TABLE
```

Three classes of pseudostatements may appear anywhere in a GPM program: ORIGIN statement, COMMENT statement, and output control statements.

3.4.1 ORIGIN

The GPM compiler produces absolute code. The ORIGIN statement is provided to allow the programmer to specify where the code should be placed in control memory. The number in the origin statement is the location to receive the next instructions compiled. All succeeding instructions will be placed in consecutive locations. The initial value for the origin is 0.

3.4.2 COMMENT

The COMMENT statement is provided to allow the programmer to document his program. In addition to the COMMENT statement, there is also a feature to allow comments for each statement, as one might use in assembly code. This feature is that any string starting with an exclamation point (!) and terminated by a carriage return is interpreted by the compiler as a semicolon (;).

Example:

```
    COMMENT              comment facility example ;
    R.0 ← 0              !zero general register zero
    R.1 ← R.0 + 1 !      set general register one to one
    COMMENT              end of comment facility example !!!!!!
```

3.4.3 Output Control

Several pseudostatements are provided to control the generation of the output listing. These can be broken into two areas: the source listing and the code listing. A complete listing consists of the following four parts:

- The source file with errors flagged and corrections made (where possible)
- The label table
- The compiled code listed in octal (normal code)
- The compiled code listed in hexadecimal

Source Listing Control

Two pseudostatements control the generation of the source listing: PRINTON and PRINTOFF. PRINTOFF will always turn off the listing; PRINTON will turn on the listing only if there has been one PRINTON for each PRINTOFF, which enables the user to nest PRINTOFF/PRINTON pairs. This is useful with nested INCLUDE files, which usually are not desired in the output listing. There is a compiler switch to allow all PRINTOFFs to be ignored, thus forcing a complete listing.

Code Listing Control

Each of three pseudostatements controls one of the three other parts of the output listing. If several of these statements appear, the last one will be in effect when the listings are generated at the end of the compilation. The initial settings are as follows:

```
LABEL.TABLE MODE FALSE;
NORMAL.CODE MODE FALSE;
HEXADECIMAL.CODE MODE FALSE;
```

However, there are compiler switches (see Section 3.9) to change these initial settings.

3.5 STATEMENTS

```
substatement ::=
  pseudostatement | assignment | control | low level
```

Four classes of statements may appear in GPM programs: pseudostatements, assignment statements, control statements, and miscellaneous statements. Pseudostatements, which are discussed in Section 3.4, do not generate any code and only condition the compilation or listing generation that follows. Assignment statements, which are discussed in Section 3.6, evaluate expressions and move data within the MLP-900. Control statements, which are discussed in Section 3.7, determine the control flow of the program. Low-level statements, which are discussed in Section 3.8, are machine-dependent statements that deal with MLP-900 specific operations but do not fit into the above categories (e.g., input/output).

3.6 ASSIGNMENT STATEMENTS

```
assignment ::=
  arithmetic | boolean | datatransfer
```

The three types of assignment statements are as follows:

- Arithmetic. Assign the value of an arithmetic expression to a General Register (OE).
- Boolean. Assign the value of a boolean expression to a flip-flop (CE).
- Data Transfer. Copy data from one machine register to another (OE and CE).

3.6.1 Arithmetic Assignment

```
arithmetic ::=
  aleft ← arithmetic | aexp | aexp modifiers

aleft ::=
  R.3  | * P.17 | @ P.17

modifiers ::=
  modifier | modifiers modifier

modifier ::=
  ( M.17 ) | [ M.17 ] | # | / number | \ number

aexp ::=
  aterm | aterm aop aexp

aterm ::=
  aprimary | NOT aprimary

aprimary ::=
  aleft | number | P.17 | ( arithmetic )

aop ::=
  + | - | MINUS | PLUS | AND | OR | XOR
```

The arithmetic assignment statement has three parts: result registers (alefts), an arithmetic expression (aexp), and modifiers (modifiers). Only the arithmetic expression must be present. The first two parts define an ordinary arithmetic calculation, while the modifiers condition the evalaution of the expression.

There are three types of modifiers; only one of each may be present. They specify the mask, test mode, and final shift.

Mask

If no mask modifier is specified, the default mask and default clear mode will be used. In nested expressions, the outer specification (if there is one) will replace the default value. The mask (M.17) specifies which mask register will be used for the calculation. The parentheses indicate clear mode false and the brackets indicate clear mode true.

Test Mode

If the test mode symbol (*) is not present, the default or outer specification will be used, as with the mask. If it is present, the new test mode will be the complement of the current default value.

Shift

If no shift is specified, none will occur. Right shift (divide) is specified by a / and left shift (multiply) is specified by a \.

Operators

No precedence is associated with any of the binary operators (aop). The unary one's complement NOT is of highest precedence. If order of evaluation is important, it must be specified with parentheses. The binary operators are

+	Two's complement add
-	Two's complement subtract
PLUS	Long add (see Chapter 4)
MINUS	Long subtract
AND	Logical and
OR	Logical or
XOR	Logical exclusive or

Result

If no result is specified, the operation will be done with test mode true. Both * P.17 and @ P.17 specify indirect references to the general registers. The character @ is a normal indirect; the register number is taken from the five low-order bits of the specified pointer register. The character * is a special indirect; it acts like a norm indirect, except that the low-order bit is forced to 1 in the register number.

Examples:

```
COMMENT if R.4 = R.11 GOTO equal.tag ;
NOT ( R.4 XOR R.11 ) !result will be zero on equals
IF ZSP GOTO EQUAL.TAG ;
```

```
COMMENT M.1 contains 7700, M.2 contains 77770 ;
COMMENT number in R.3 field M.1 added to R.4 field M.2 ;
R.4 ← R.4 + ( R.3 [M.1] /3 ) (M.2) ;
```

3.6.2 Boolean Assignment

```
boolean ::=
  F.377 ← bexp

bexp ::=
  bexpr I boolean

bexpr ::=
  bterm I bexp bop bterm

bterm ::=
  bprimary I NOT bprimary

bprimary ::=
  F.377 I TRUE I FALSE I ( bexp )

bop ::=
  AND I OR I XOR
```

The boolean assignment statement provides a method to set flip-flops to the value of a boolean expression. The boolean expression is composed of flip-flops and the boolean constants TRUE and FALSE. The operators are the logical operators AND, OR, XOR, and NOT.

As in the arithmetic expression, there is no precedence between the binary operators (bop), and the unary one's complement NOT is of highest precedence. If order of evaluation is important, it must be specified with parentheses.

Examples:

```
GI.3 ← GI.3 XOR GI.5 !if GI.5 then complement GI.3
GI.7 ← GI.1 OR GI.2 OR NOT GI.3 ;
GI.11 ← (GI.0 AND GI.5) OR NOT (GI.7 AND GI.6);
```

3.6.3 Data Transfer

```
datatransfer ::=
  dt36lft ← dtnot dt36rt dtmask I
  dt16lft ← dtnot dt16rt dtmask I
  dt8lft ← dtnot dt8rt dtmask

dtnot ::=
  NOT I (empty string)
```

```
dtmask ::=
  ( number ) | [ number ] | (empty string)

dt36lft ::=
  oereg | oepg @ P.17 | oepg * P.17 | XBUS

oereg ::=
  R.37 | MISC.37 | M.17 | A.1777 | LB.1777 |
  SUPVLB.377 | XLATOR.777

oepg ::=
  R.0 | MISC.0 | M.0 | A.PG.3 | LB.PG.3 |
  SUPVLB.0 | XLATOR.PG.1

dt36rt ::=
  dt36lft | number | P.17

dt16lft ::=
  dt36lft H.1 | ( cereg ) | ( cereg , cereg ) | S.17

cereg ::=
  CE.137 | P.17 | XBUS.3

dt16rt ::=
  dt16lft | number

dt8lft ::=
  dt36lft B.3 | cereg

dt8rt ::=
  dt8lft | number | F.377
```

The basic format of a data transfer statement is

left ← not right mask

The left and right fields are data objects of matching size. The possible sizes are 36, 16, and 8 bits. The NOT field contains an optional one's complement NOT.

The mask notation is similar to the arithmetic assignment, except that the mask is specified as a constant number instead of as a mask register. The parentheses specify a normal mask, where all masked-out (zero mask bits) bits remained unchanged. The square brackets specify a clear mask where all masked-out bits are zeroed. If no mask is specified, an all-ones mask of the appropriate size is used.

36-bit transfers

The 36-bit left operands are OE registers. The right operands are either OE registers, constants, or pointer registers. In the case of pointer registers, the high-order 28 bits are zero. The OE registers are as follows:

● R.37	32 general-purpose registers,
● M.17	16 mask registers,
● MISC.37	32 miscellaneous registers,
● A.1777	1024 auxiliary memory registers,
● LB.1777	1024 language board registers,
● SUPVLB.377	256 supervisor language board registers (only Microvisor mode access allowed),
● XLATOR.777	512 translator memory registers (only Microvisor mode access allowed).

In addition to direct references to OE registers, they may be referenced indirectly through the pointer registers. OE registers are divided into pages of up to 256 registers. The 8-bit pointer registers can address any register within a page. It is possible only to indirectly address registers within a fixed page. As with the arithmetic assignment statement, the * indirect operator will force the low-order register number bit to a 1.

16-bit transfers

There are four types of 16-left operands. These and constants comprise the possible right operands. The four left operand types are as follows:

1) OE register Half-words - <dt36lft H.1>
Half-words are numbered from left to right. The high-order four bits are never referenced. Therefore, H.1 refers to the low-order 16 bits and H.0 refers to the next lowest 16 bits. Note that whenever half-word references are used, as the left side of a data transfer, the remainder of the specified OE register is zeroed. Additionally, OE registers may not appear as both left and right operands.

2) CE Double Register - <(cereg)>
The CE register double-register construct references an odd/even pair of CE registers. The CE register explicitly named within the parentheses is the first register of the pair. The two examples following will each cause a swapped data transfer:
R.0 H.1 ← (P.1);
(P.1) ← (P.6);

3) General CE Double Register -
<(cereg, cereg)>
The CE register general double register construct is similar to the double register construct described above except that both CE registers are named explicitly. If the general double register is not an odd/even pair, it cannot be moved to or from an OE register half-word. The following is an impossible data transfer:
(P.1,P.2) ← R.17 H.0;

4) Subroutine Stack Register - <S.17>
The construct S.n is equivalent to (CE.100+2n) or (CE.100+2n,CE.101+2n).

<u>8-bit transfers</u>

There are two types of 8-bit left operands. They are as follows:

1) OE Register Byte -
Bytes are numbered from left to right. The high-order four bits are never referenced. Therefore B.3 refers to the low-order 8 bits, B.2 refers to the next lowest 8 bits, etc. Note that whenever byte references are used as the left side of a data transfer, the remainder of the specified OE register is zeroed. Additionally, OE registers may not appear as left and right operands.

2) CE Register - <cereg>
The CE registers are

- CE.137 All CE registers;
- P.17 pointer registers, (CE.40-CE.57);
- XBUS.3 CE exchange bus, (CE.70 - CE 73 as left operands; CE.64 - CE.67 as right operands).

In addition to the two operand types discussed above, 8-bit right operands may also be either constants or flip-flops. In the case of flip-flops, the right operand is interpreted as an 8-bit quantity, where each bit is a copy of the value of the specified flip-flop.

Examples:
```
    R.0 ← NOT A.173 [777];
    A.PG.0 @ P.1 ← A.PG1 @ P.i;
    M.17 H.1 ← NOT S.12;
    M.1 ← 777777777777;
    R.3 B.3 ← P.17;
    R.3 ← P.17;
    P.17 ← CE.0;
    [illegible]3 ← NOT F.144 (123);
```

3.7 CONTROL STATEMENTS

```
    control ::=
      block | break | branch | do | if | switch
```

There are six control structures in GPM. They are as follows:

● Block	Prototype compound statement form,
● BREAK	Standard block exit mechanism,
● Branches	Unconditional transfer of program control,

● DO	Looping mechanism,
● IF	Conditional execution and compilation,
● Switch	Case analysis (index branch) mechanism.

3.7.1 Block

```
block ::=
  BEGIN name body END name

name ::=
  NAMED symbol | (empty string)
```

The BEGIN END block is the prototype compound statement form in GPM. The IF, DO.BEGIN. and SWITCHON statements are special cases of the BEGIN block. All have the characteristics of the standard block in addition to special features of their own.

Scope

The block specifies the scope for any declarations that may appear in the declaration part of the block body. In the special case blocks, the BEGIN END also determines the scope of the control structure involved.

Names

Blocks can be named by following the BEGIN with "NAMED id," which enables the program to refer to the block by name. This is used for two purposes. First, the END may be named, thus closing all unnamed blocks within the named block; also, the block name is used by the BREAK statement to specify which block to exit.

3.7.2 BREAK

```
break ::=
  BREAK name
```

The BREAK statement will cause program control to branch to the end of a particular block. If no name is supplied to the BREAK, the current block will be exited. If a name is supplied, then control will branch to the end of that block.

This is different from a RETURN statement. The RETURN statement exits a subroutine to the called location (determined at runtime), whereas the BREAK statement exits a block to a block end (determined at compile time).

3.7.3 Branches

```
branch ::=
  RETURN | GOTO label | CALL label

label ::=
  location | < P.17 > | location < P.17 >
location ::=
  id | number | offset | id offset

offset ::=
  + number | - number
```

The three types of unconditional branches are RETURN, CALL, and GOTO. The RETURN statement transfers control to the location on the top of the hardware subroutine stack, and pops the stack. The CALL statement pushes the location of the next sequential instruction in control memory onto the top of the stack and does a GOTO. The GOTO simply branches to the location specified by the label.

In addition to the unconditional branches provided by the branch statements, GPM also has conditional branches. These are special forms of the IF statement described in Section 3.7.

3.7.4 Labels

There are basically two types of labels of branch destinations: relative and absolute. Either type can be indexed by the value of a pointer register: The indexing is always post-indexing, that is, the branch destination is calculated and the value of the pointer register is then added. This addition might cause overflow, in which case the transfer destination will wrap around to low control memory. If the label is only a pointer register, then the index is relative to the next sequential instruction in control memory.

Absolute Labels

An absolute label may transfer a program label identifier (see Section 3.3) or an absolute location specified by a number.

Relative Labels

A relative label may be merely an offset, specifying a transfer relative to the current location in control memory, or an offset from some specified program label identifier.

Examples:

```
      GOTO TAG;
      CALL 100 <P.3>;
   TAG:
      CALL TAG +3;
      RETURN
      GOTO -4;
      CALL +1<P.>;
```

3.7.5 DO.BEGIN

```
do ::=
  DO.BEGIN name body END name
```

The DO.BEGIN statement unconditionally repeats the body of code contained within. This is the looping construct in GPM. The loop is usually terminated with a BREAK

Example:

```
COMMENT construct n-bit mask - n is in general register N ;
R.1 ← 0 ! initialize mask result register
DO.BEGIN
   R.1 ← R.1 + 1 \ 1 !add another bit to the mask
   N ← N - 1 !decrement count
   IF ZSP BREAK !break when count runs out
END; R.1 ← R.1 / 1 !done
```

3.7.6 IF

```
if ::=
  IF bexp THEN.BEGIN name body ELSE statementlist END name I
  IF bexp THEN.BEGIN name body END name I
  IF bexp BREAK name I IF bexp RETURN I
  IF bexp CALL id I IF bexp GOTO id
```

There are two types of IF statements: block structured and conditional branch. The first is for the conditional execution of sections of code and the second for the conditional transfer of control. The first is sufficient in all cases, but the second is easier and more efficient when appropriate.

Block Structured IF Statement

The block structured IF statement has two forms, the most general of which is the IF THEN.BEGIN ELSE END form. In this case the boolean expression is evaluated. If it is true, the body following the THEN.BEGIN is executed. The statement list following the ELSE will not be executed. If the boolean expression is false, the opposite will happen; the body will not be executed and the statement list will be.

Any declarations that follow the THEN.BEGIN will be active for both statements in the body following the THEN.BEGIN and statements in the statement list following the ELSE. The second form of IF simply omits the ELSE sections.

Conditional Compilation

The boolean expression is evaluated at compile time. If it evaluates to a constant TRUE or FALSE, then the IF statement will compile code for the appropriate statements only; no test will be compiled at all. ORIGINs and program label assignments can also be conditionally compiled using this facility. There is no way to conditionally specify declarations for a block.

Conditional Branch IF Statement

These IF statements do not contain either the THEN.BEGIN or the END. Immediately following the boolean expression is a branch statement (BREAK, RETURN, GOTO, CALL). The available branch statements are restricted, and only label names may be used as the GOTO or CALL destinations.

The conditional branch IF statement is provided so programmers may write GOTOless programs without being penalized with inefficient code. Note that a BREAK inside a block-structured IF statement will only BREAK out of the IF block if the BREAK is not NAMED. This means that the following two statements are NOT equivalent:

```
IF ZSP THEN.BEGIN BREAK END;
IF ZSP BREAK;
```

3.7.7 Switch

```
switch ::=
  switchblock | switchtag

switchblock ::=
  SWITCHON < P.17 > INTO.BEGIN name body END name

switchtag ::=
  CASE switchlist | ENTRY switchlist

switchlist ::=
  switchvalue | switchlist , switchvalue

switchvalue ::=
  number | number THRU number | number THRU |
  THRU number | THRU
```

A switch statement has two components: first, a switch block that specifies the pointer register to be used to index into the body of the block and second, a number of switch tags that specify where each index value is to start execution.

Switch Blocks

The switch block specifies a pointer register. The value of this register and the switch tags within the switch block determine where in the body of the switch block execution will begin.

Switch Tags

There are two types of switch tag statements. The ENTRY statement specifies a list of pointer register values that are to start execution following the ENTRY statement. The CASE statement is equivalent to the ENTRY statement, except that the CASE statement compiles a BREAK out of the switch block.

Switch Values

Switch values are either numbers or ranges of numbers. The range of a SWITCHON can be a maximum of 0 through 377. On the THRU version of the switch value 0 is assumed if the start is not specified, and 377 is assumed if the end is not specified. Also, if some particular number has been assigned previously, the THRU specification will ignore it. On the other hand, a single number specification will override.

Efficiency Considerations

The first statement following the INTO.BEGIN (after any declarations) should be an ENTRY statement. A CASE will produce an unnecessary BREAK, and any other statement will never be executed.

Each switch value declared produces one instruction overhead. The switch is assumed to have a 0 origin. For example, a CASE 2 and 4 will have five (0-4) instructions overhead.

Debugging Considerations

No check is made at run time as to the value of the pointer register. Any unspecified values below the maximum specified value will transfer control to the location immediately following the switch block. However, values above the maximum will transfer to a location beyond the switch block, producing strange results.

Examples:
```
SWITCHON <P.1> INTO.BEGIN
  ENTRY 2,4;
  COMMENT CASES 2,4;
  CASE 1 THRU 6,10;
  COMMENT CASES 1,3,6,10;
  ENTRY 5;
  COMMENT CASES 1,3,5,6,10;
  END
```

3.8 LOW-LEVEL STATEMENTS

```
lowlevel ::=
   incr/decr | blot | cede | shift | mul/div
```

The low-level GPM statements include the following:

- INCREMENT/DECREMENT
- BLOT
- CEDE
- SHIFT
- MULTIPLY/DIVIDE

3.8.1 INCREMENT/DECREMENT

```
incr/decr ::=
  inde P.17 BY num
inde ::=
  INCREMENT I DECREMENT
```

This statement allows a constant to be added to or subtracted from a pointer register.

3.8.2 BLOT

```
blot ::=
  blotcode label;
blotcode ::=
  MOE I RSB I WSB I RCM I WCM I WBP
```

See Chapter 4.

3.8.3 CEDE

```
cede ::=
  cedeaddr I cededata I cedecomb

cedeaddr ::=
  addrop 2 left addrsign addrb testmode I
  ROW testmode
addrop ::=
  FIN I FOP I SAD I RMW
addrsign ::=
  + I -
addrb ::=
  aleft I number I P.17
testmode ::=
  .empty. I #

cededata ::=
  dataop dt361f4 testmode
dataop ::=
  WOP I SOP I WOS

cedecomb ::=
  combop aleft, addrb testmode
combop ::=
  WIF I WON I WIN I WOF
```

See Chapter 4.

3 SHIFT

```
shift ::=
shop aleft shdir shamt shmask testmode;
shop ::=
  SHIFT.DE.L | SHIFT.EO.L | SHIFT.SINGLE.L |
  SHIFT.DUAL.L | SHIFT.OE.C | SHIFT.RE.L |
  SHIFT.ER.L | NORMALIZE | SHIFT.RE.C
shdir ::=
  LEFT | RIGHT
shamt ::=
  @ | num
shmask ::=
  .empty | ( M.17 )
```

See Chapter 4.

3.8.5 MULTIPLY/DIVIDE

```
muldiv::=
  mdop aleft BY aright mask testmode
mdop::=
  MULTIPLY | DIVIDE
aright::=
  aleft | number | p.17
mask::=
  (M.17) | empty
testmode::=
  * | empty
```

4. MLP-900 REFERENCE MANUAL

4.1 INTRODUCTION

The MLP-900 is a large vertical-word microprogrammable computer designed to provide a general-purpose emulation host on which each user can create his own target machine. It is a synchronous machine with a 250-nanosecond cycle time, a 4096-word control memory, and a large set of internal registers. A number of original features help make the MLP-900 an exceptionally powerful microprogramming tool; principal among them are a subroutine stack, a multi-level interrupt mechanism, a two-state protection facility, paging and memory protection hardware, and provision for user-specified language boards to provide a hardware assist for particular applications.

The MLP-900 is characterized by two parallel computing engines, known as the Operating Engine (OE) and the Control Engine (CE). The OE is a 36-bit-wide arithmetic and data transfer machine; it includes the hardware for the main memory and external interfaces and the bulk of the register space, including a 1K internal memory. The CE is the instruction sequencing and control unit; it includes the stack handling, interrupt, and protection mechanisms.

MLP-900 instructions are known as "ministeps"; each engine has its own unique instruction set. Ministep execution proceeds sequentially, either singly or in pairs. At the beginning of each cycle, the CE fetches a pair of ministeps from control memory--from the current address and its successor--and examines them. If the first is an OE ministep and the second is a CE ministep, then the pair is executed during this cycle; otherwise only the first ministep is executed (the other will be the first ministep of the next cycle, barring a branch).

With two exceptions, this parallelism is transparent to the user and serves only to increase the effective machine speed: first, interengine data transfers require execution of an OE-CE pair; second, CE registers modified as a side effect of an OE ministep cannot be sensed by a CE ministep immediately following. All changes to the state of the machine occur simultaneously at the end of the cycle ("clock time"); all computations and decisions are therefore based upon the values present at the beginning of the cycle.

The MLP-900 hardware recognizes two distinct execution states, known as user mode and "Microvisor" (microprogram supervisor) mode. User mode microcode is subject to three restrictions: (1) privileged ministeps may not be executed; (2) privileged registers (in both the OE and CE) may not be modified; and (3) a branch to a Microvisor location other than a designated entry point is illegal. Violation of any restriction results in a (privileged) interrupt and suppression of the current cycle. These restrictions fully protect the external interface, the main memory protection and paging facility, and the Microvisor itself from the user microcode; additionally, the microcode is restricted from modifying itself.

The MLP 900 main memory interface includes a memory protection and paging scheme which, together with some Microvisor code, provides the user with a 256K virtual address space. The scheme mimics the memory management provided by the BBN pager on the PDP-10.

The language board facility allows a major application to design its own extension to the MLP-900 hardware, consisting of two PC boards, an OE board and a CE board; the pair is referred to as a language board, and is intended for the exclusive use of that one application. There is physical space for a maximum of four language boards, of which one is the "null" board for general use. Two bits in the CE select the current board. The intended uses of a board include, but are not limited to, target instruction decoding, effective address calculation, and normalization.

Throughout this chapter, registers are referred to by their GPM names, and register sets are referred to by the name of the last register in the set (the index number is always an octal number). Thus R.37 refers to either the 32 general registers or the last one of them, while R.15 refers to the thirteenth register of that set.

4.2 OPERATING ENGINE

The Operating Engine (OE) is a 36-bit data transfer and manipulation engine: it also contains the interfaces with both main memory and the PDP-10 I/O bus. The computational facility consists of a three-input (two operands and a mask) "Primary Adder" capable of various arithmetic and boolean functions, a "Primary Shifter," and an "Extension Shifter" used for double-word shifts. Operands are taken from, and results stored into, the general registers (R.37); masks are taken from the mask registers (M.17). One byte of CE flip-flops (CE.14) is devoted to functions associated with the adder and shifter(s). The interfaces consist of a number of special registers and pseudo-registers (grouped together in MISC.37), the main memory address translator (XLATOR.777), and the memory referencing ministep (CEDE).

Note that in all OE ministeps involving a large constant operand, the ministep takes two control memory words; while the hardware handles the decode automatically, the programmer must be aware of the fact that such a ministep always executes singly. A large constant is one which cannot be expressed in six bits (i.e., not in the range 0-63).

OPERANDS

The OE operands are contained in one sparse 12-bit address space. In addition to the mnemonics shown in Table 4.1, these operands may be addressed as OE.0 - OE.7777.

TABLE 4.1.

OPERATING ENGINE ADDRESS SPACE

Group	Extension	Register	Mnemonic	Description
0000	000	xxxxx	R.37	General Registers
0001	000	0xxxx	M.17	Mask Registers
0010	000	xxxxx	MISC.37	Miscellaneous Reg.
01xx	xxx	xxxxx	A.1777	Auxiliary Memory
1000	000	00000	XBUS	CE Exchange Bus
1001	xxx	xxxxx\	XLATOR.777	(protected)
1010	xxx	xxxxx/		Translator Memory
1011	xxx	xxxxx	SUPVLB.377	(protected)
				Supv. Lang. Board
11xx	xxx	xxxxx	LB.1777	Language Board

Indirect OE Operands. The OE registers may be addressed not only directly, but also indirectly through the Pointer Registers. As the Pointer Registers are only 8 bits wide, the group is still specified in the instruction. There are two types of indirect referencing available. Normal indirect (@) uses the Pointer Register for the lower 8 bits where applicable (i. e., only 5 bits are used when referencing the General Registers). Special indirect (*) is similar, except that the low-order bit is forced to 1.

Examples:
R.0 @ P.5
LB.1400 * P.11
XLATOR.400 @ P.7

The GEAR and SHIN ministeps indirect only to the General Registers, while both CEDE and GENT indirect to all OE registers.

4.2.1 R.37. General Registers

There are 32 general registers (R.0 - R.37), each 36 data bits wide. Four parity bits, one for each 9-bit byte, are maintained with each register. All 32 registers are addressable as inputs to the Primary Adder. Except for R.37, the Shift Extension Register, none of the General Registers has a dedicated function.

4.2.2 M.17. Mask Registers

There are 32 mask registers. However, only 16 of them (M.0-M.17) can be addressed by an OE instruction. The high-order bit of the mask address is CE flip-flop (F/F) MBS (F.167). This F/F is protected and can only be set or reset by a ministep in Microvisor mode. Therefore, user programs see only 16 Mask Registers. The Mask Registers condition the Adder functions to accomplish subword operations.

4.2.3 MISC.37. Miscellaneous Registers

There are thirty-two Miscellaneous Registers (MISC.0 - MISC.37) for different dedicated functions. For addressing purposes, they have been gathered together in one set of registers. The first sixteen (MISC.0 - MISC.17) are available to the user; the second sixteen (MISC.20 - MISC.37) are privileged and can be modified only by the Microvisor, but can be read by the user using a GENT instruction. Some registers are readable and writable, some are read-only. and others are unimplemented. A complete list of the miscellaneous registers, their numbers, and their functions is given below.

0 Data Entry Switches
1 Main Memory Address Switches
2 Processor Address Switches

The above three entries are pseudo-registers which make available the three sets of switches on the console.

3 Unimplemented

The following two registers can be read and written and are highly tied into Language Boards and the CEDE/WIN instruction. These registers can be treated as Auxiliary Memory (Scratch registers) but are unlikely to be, since they are too important in their other functions. For more information on PIR and SIR, see the section on Language Boards and the CEDE/WIN Instruction.

4 Primary Instruction Register (PIR)
5 Secondary Instruction Register (SIR)
6 Unimplemented
.
.
.
15

The folowing two registers are used in memory referencing. For more information, see the CEDE instruction.

16 Virtual Address Register (VAR)
17 Memory Data Register (MDR)

This concludes the registers available to the user. The succeeding registers are privileged.

The next ten registers are involved in paging and page fault handling.

20 Address limit and User Base Register (ALR/UBR)
The ALR/UBR performs the same function as the similar register in the BBN pager.

21 Age and Process Use Register (AGER/PUR)
The AGER/PUR is analogous to the same register in the BBN pager.

22 Generated XLATOR Word
This is a psuedo-register containing the data for loading into translator memory at the completion of a page fault.

23 Real Address Register (RAP)
This register is used by the MLP-900 when in transparent (nontranslate) address mode.

24 Trap Status Word (TSW)
This is a pseudo-register which generates a TSW analogous to that generated by the BBN pager.

25 User Base Address (UBA)
This is a pseudo-register which generates the address for a Microvisor access to the User's Page Table.

26 Core Status Table (CST)
The CST is a pseudo-register which generates the address for a CST reference.

27 Special Page Table (SPT)
The SPT is a pseudo-register which generates the address for a SPT reference.

30 Indirect Page Table (IPT)
The IPT is a pseudo-register which generates the address for a IPT register.

31 Key Register
This contains a 7-bit key value which determines the validity of XLATOR entries.
The following three registers are the control interface with the PDP-10. See Appendix D.

32 DATAO

33 DATAI

34 Command/Status Register

35 Unimplemented

36 Virtual Address Compare Register (VADRC)
VADRC, when enabled by SARM.1, is compared to the virtual address (VAR) at every Main Memory reference, and generates an AR (VADR, F.124) when a match occurs.

37 Control Memory Address Compare Register (CMADRC)
When enabled by SARM.0, CMADRC is compared to the memory address at every control memory reference, and generates an AR (CMADR, F.110) when a match occurs.

A transfer to an unimplemented register is a no-op; a transfer from an unimplemented register yields -1.

4.2.4 A.1777. Auxiliary Memory

There are 1024 words of 200-ns auxiliary memory, which can be used as a scratchpad or cache. This memory can be accessed by the OE instructions CEDE and GENT and the CE instruction BLOT.

4.2.5 XBUS. Exchange Bus

The CE Exchange Bus is a pseudo-register connected to the CE Exchange Bus (see Section 4.3.3.). Data transfers between the engines are accomplished by an OE-CE instruction pair, with the OE instruction either a GENT or a CEDE (which references the Exchange Bus), and the CE instruction either a MOVE (which references the Exchange Bus) or a BLOT (other than MOE). Since these instruction pairs are executed in parallel, the OE instruction (GENT or CEDE) must come first regardless of the transfer direction. In transfers to the OE, any bits not loaded by the CE instruction are transferred as zero. In transfers to the CE, any bits not used by the CE instruction are ignored. A reference to the Exchange Bus without a paired CE instruction is undefined.

4.2.6 XLATOR.777. Translator Memory

The Translator Memory consists of 512 20-bit words used in translating virtual addresses to real addresses. Each word consists of a 7-bit key value, a 9-bit real core address value, a write permit bit, a parity bit, and two unused bits. Whenever translation is performed, the 9 high-order bits of VAR are used as an index into the translator to select a translator word. The word is valid if its key value matches the key register (MISC.31); the write permit bit is "on" if this is a store. The Translator Memory is privileged.*

4.2.7 SUPVLB.377. Supervisor Language Board

These registers do not exist, and are not expected to be added. They are privileged.

4.2.8 LB.1777. User Language Board

Provision is made for up to 256 36-bit registers on each of up to four Language Boards in the MLP-900. The null Language Board, which is always LB.0, has no registers. Other Language Boards, designed for specific users, may have up to 256 registers as needed. Note that the microcode can address all the registers on all the Language Boards and is not limited to the currently active Language Board. See Section 4.4.

OPERATORS

The OE operators are as follows:

- GEAR General Arithmetic. Performs binary arithmetic, logical operations, and single register shifts.
- CEDE Conditional External Data Exchange. Transfers addresses, target instructions, and data between the OE and Main Memory.
- SHIN Shift Instruction. Performs various single and double register shifts, plus the iterated steps of multiply and divide loops.

----

* Caution: a GENT from the translator reads the word selected by the old value of VAR, then modifies the 9 high-order bits of VAR to address the requested word, which is not readable except by coincidence.

- GENT General Data Transfer. Transfers data between the OE registers and to and from the CE.

4.2.9 GEAR. GEnerai ARithmetic

The ministep provides arithmetic and logical capability within the General Registers.

Syntax:

```
gear ::=
      aleft ← aexp amodifier;
aleft ::=
      R.37 | * P.17 | @ P.17
amodifier ::=
      shift mask testmode
shift ::=
      / samount | \ samount | .EMPTY.
samount ::=
      1 | 2 | 4 | 6 | 10 | 14 | 20
mask ::=
      ( M.17 ) | [ M.17 ]
testmode ::=
      * | .EMPTY.
aa is identical to the specified aleft
ab ::=
      aleft | number | P.17
aexp ::=
      aa + ab | aa - ab | ab - aa |
      aa PLUS ab | aa MINUS ab | ab MINUS aa |
      aa AND ab | NOT aa AND ab | aa AND NOT ab |
      aa OR ab | NOT aa OR ab | aa OR NOT ab |
      aa XOR ab | NOT aa XOR ab | ab | NOT ab
```

Examples:

```
R.1 ← R.1 + R.2 (M.0);
R.7 ← R.7 - P.0 /1 [M.1] *;
R.37 ← 173 - R.37 \2 (M.2);
@P.0 ← @P.0 XOR NOT 3 (M.17);
*P.17 ← *P.17 AND P.3 /4 [M.27] *;
@P.3 ← NOT @P.3 OR R.17 \20 (M.21);
@P.1 ← *P.1 MINUS @P.1 (M.3) *;
```

Semantics:

The GEAR ministep is used for arithmetic operations. It selects two operands and a mask and routes them to the primary adder, and then specifies a shift of the result through the primary shifter. The result is then stored into the A operand. This operation is shown in Figure 4.1 below.

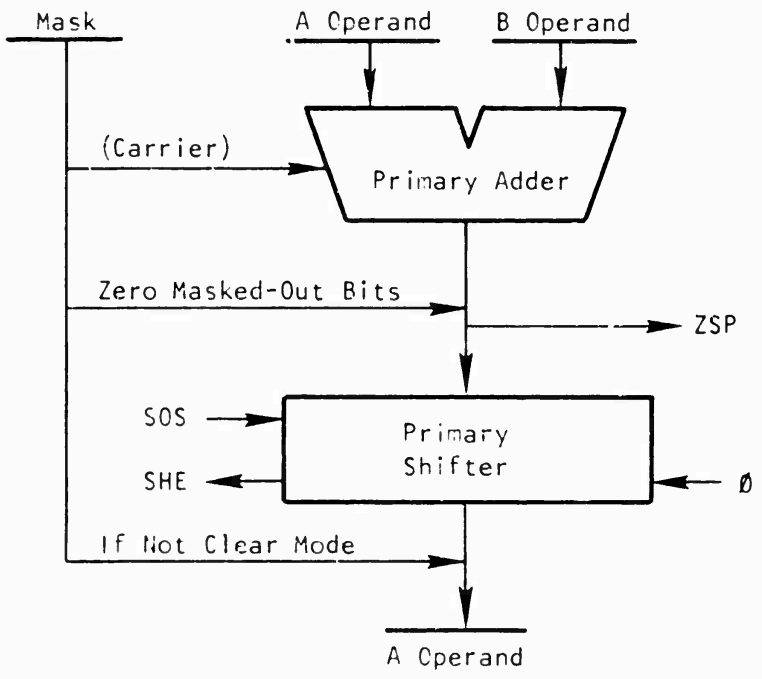


Figure 4.1 Operating Engine: GEAR.

Masks

The requested operation is conditioned by the value of the specified Mask Register. One (1) bit in the mask is a masked-in bit.

Adder. The Primary Adder treats all the masked-in bits as one contiguous operand field; carry generation is suppressed in masked-out bits, and carry propagates over masked-out bits. The masked-out positions are all forced to zero at the Primary Adder output.

Shifter. The shifter ignores the mask

Result Store into A. In Clear mode [M.17], the entire 36-bit ouptut of the primary shifters is stored; if the shift amount is zero, then all masked-out bits are cleared to zero. In normal mode [M.17], only the masked-in bits are stored; the masked-out bits remain unchanged.

Test Mode

If the test mode modifier <*> is present, the store into the A operand is suppressed. However, all applicable F/F's (see Table 4.2) are set.

Operators

All valid operator combinations are listed in the syntax under aexp. All addition and subtraction operators are two's complements. NOT is a logical operator (one's complement). The PLUS and MINUS operators take F/F COF.1 as an initial low-order carry-in. These operators can be used to produce multiple-precision results.

Shifts

All valid shift amounts are listed in the syntax under samount. The prefix / designates a right (divide) shift and the prefix \ designates a left (multiply) shift. The boundary shift conditions are shown in Figure 4.2.

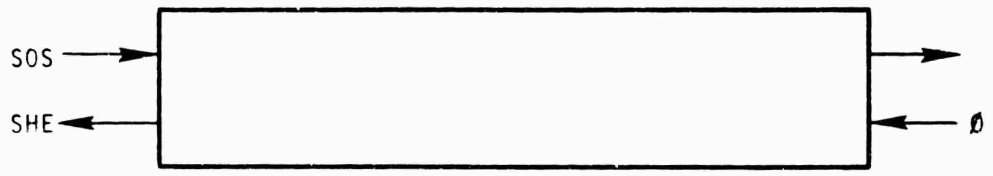


Figure 4.2 Shifter boundary conditions.

Flip-Flops

Table 4.2 lists all F/F's that may be affected by a GEAR.

COP = F.300. This pseudo-F/F contains the carry-out value for +, -, PLUS, and MINUS. It is valid only during the current cycle.

COF.1 = F.140. This F/F contains the carry-out value of the most recent +, -, PLUS, MINUS operation executed.

COF.2 = F.141. This F/F contains a copy of the previous setting of COF.1, and therefore of the second most recent +, -, PLUS, or MINUS executed.

ZSP = F.301. This pseudo-F/F is set if the MASKED output of the Primary Adder of this operation is zero. Active for all GEAR operations, it is valid only during the current cycle.

ZRF.1 = F.142. This F/F contains the most recent setting of ZSP except in the case of PLUS and MINUS, when it is set to the logical product of ZSP and its prior value.

ZRF.2 = F.143. This F/F contains a copy of the previous setting of ZRF.1.

<u>SOS</u> = <u>F.146</u>. If there is a nonzero right (/) shift, SOS is copied into bit 0.

<u>SOF</u> = <u>F.147</u>. If there is a nonzero left (\) shift, the bits shifted out of bit 0 are compared with SOS; if the comparison fails, SOF is set.

<u>SHE</u> = <u>F.145</u>. If there is a nonzero left (\) shift, the last bit shifted out of bit 0 will be in SHE. This happens after the GEAR cycle.

Table 4.2

GEAR Flip-Flops

F/F	Active Condition
COP	+, -, PLUS, MINUS
COF.1	Same as above
COF.2	Same as above
ZSP	All GEAR operations
ZRF.1	Same as above
ZRF.2	Same as above
SOS	Nonzero right (/) shift
SOF	Nonzero left (\) shift
SHE	Same as above

4.2.10 CEDE. Conditional External Data Exchange

CEDE is used to fetch and store main memory. All memory fetches or stores require the execution of two CEDEs. The first provides the virtual or real address, depending on TRBY (F.165), initiates a translate cycle if translating (i.e., if not TRBY), and, if reading, initiates the memory fetch. The second CEDE, which need not follow immediately, provides the data for a store or waits for the operand of a fetch. Some combined forms wait for an operand and then begin a new fetch. Page fault ARs take place at the time of the second instruction (the Wait or Store) and cause that instruction to be suppressed.

Syntax:

```
cede::=
     cedeaddr | cededata | cedecomb | cede b

cedeaddr::=
     addrop addra addsign addrb testmode | ROW testmode
addrop::=
     FIN | FOP | SAD | RMW
addra::=
     aleft (as in GEAR)
addsign::=
     + | -
addrb::=
        aleft | number | P.17
    testmode::=
        * | .EMPTY.

    cededata::=
        dataop dataloc testmode
    dataop::=
        WOP | SOP | WOS
    dataloc::=
        oereg | oereg @ P.17 | oereg * P.17 | XBUS
    oereg::=
        R.37 | M.17 | MISC.37 | A.1777 | LB.1777 |
        XLATOR.777 | SUPVLB.377

    cedecomb::=
        combcode addra, addrb testmode
    combcode::=
        WOF | WON

    cede b::=
        b code addra ← addsign addrb testmode
    bcode::=
        WIN | WIF
```

Examples:

```
FOP R.3 ← R.6;
SAD @ P.0 -2;
WOP XBUS;
WOF R.1, * P.2;
ROW;
SOP M.0 @ P.10;
```

Semantics:

Type	Name	Description

Addr	FIN
	Fetch
	Instruction
		VAR, A ← A +/- B;
		VAR-command-bits ← "read";
		Translate;
		Fetch

Comb	WIN
	Wait for
	Instruction
		Wait;
		[PIR or SIR ← MDR];
		VAR, A ← +/- B;
		[VAR-command-bits ← "read";
		Translate;
		Fetch];

		LB Break-out
Addr	FOP Fetch Operand	Identical to FIN

Addr	SAD
	Set Address
		VAR, A ← A +/- B;
		VAR-command-bits ← "store";
		Translate

Addr	RMW
	*Read-Modify-Write
		VAR, A ← A +/- B;
		VAR-command-bits ← "read" & "store";
		Translate;
		Fetch;
		(Must be followed by WOP and then SOP within time allowed for RMW timeout.)

Comb	WIF
	Wait
	Indirect & Fetch
		Wait;
		A ← MDR;
		VAR ← +/- B;
		[VAR-command-bits ← "read";
		Translate;
		Fetch]

Comb	WOF
	Wait for Operand & Fetch
		Wait;
		A ← MDR;
		VAR ← B;
		VAR-command-bits ← "read";
		Translate;
		Fetch

Data	SOP Store Operand	MDR ← A; Store (Preceding CEDE must be SAD, or the WOP after RMW.)

Data	WOP
	Wait for Operand
		Wait;
		A ← MDR

Data	WOS
	*Wait for Operand, Stream Mode
		Wait;
		A ← MDR;
		(WOS triggers an asynchronous mode of continuous memory fetching from successive

----

* Indicates a privileged CEDE.

		locations on the same memory page at maximum memory rate; WOS must be executed in a loop which is faster than the memory i.e., one MLP-900 cycle, lest data be lost without any indication.)
Addr	ROW *Retry Operation	Translate; If "read" is set in VAR, Fetch (Acts like FOP or SAD, or RMW, depending on the old contents of VAR.)
Comb	WON Wait for Operand and Fetch Instruction	Identical to WOF

FOP and WOP are the basic memory fetch pair, while SAD and SOP are the basic memory store pair.

Translate: Use contents of VAR as index into translator memory, and note (internally) whether the translation is OK.

Fetch: If the translation is OK, initiate Fetch from memory, remember that there is an outstanding Fetch, and increment VAR by one (only the 9 Least Significant Bits are affected; if they were all ones, then they are made zero, but there is no further carry). When the memory responds with the data, it is stored in MDR and the remembered Fetch condition is cleared. The Fetch for RMW does not increment VAR.

Store: If the (most recent) translation is OK, initiate a memory store cycle of the word in MDR; if the translation is not OK, suppress this ministep, and set the PAGE AR request F/F (F.121). If the "store" command is not set in VAR the result is undefined.

Wait: If the last translation is not OK, suppress this ministep and set the PAGE AR request (F.121). If there is still a memory fetch in progress, wait for it to complete (and the data to be in MDR).

[...]: Indicates an action which is LB conditional; is an output from the LB.

LB Break-out: An implicit MINIFLOW branch to a location determined by the Language Board.

4.2.11 SHIN. SHift INstruction

The SHIN ministep provides single- and double-register shifting by both fixed and variable amounts. In addition, two of the variants provide the basic shift-and-add step required for multiplication and division operations.

Syntax:

```
shin ::=
      shop aleft sshift smask testmode |
      mulop aleft BY ab smask testmode
shop ::=
      [ ...as is... ]
aleft ::=
      R.37  |  @ P.17  |  * P.17
sshift ::=
      RIGHT shamount  |  LEFT shamount
shamount ::=
      0  |  1  |  2  |  4  |  6  |  10  |  14  |  20  |  @
smask ::=
      ( M.17 )  |  .empty.
testmode ::=
      *  |  .empty.
mulop ::=
      MULTIPLY  |  DIVIDE
ab ::=
      aleft  |  number  |  P.17
```

(Note that aleft, ab, and test mode are identical to the same constructs in the GEAR ministep; shamount is similar to samount, with the addition of "@", while smask is similar to mask, with the deletion of "[M.17]".)

Examples:

```
SHIFT.EO.L R.12 LEFT 6 ;
SHIFT.OE.C @P.4 RIGHT @ ;
MULTIPLY R.20 BY 12 (M.17) ;
```

Semantics:

The SHIN ministep for the shifting of either a single register (SHIFT.SINGLE.L) or an even/odd register pair (SHIFT.OE.L, SHIFT.OE.L, SHIFT.OE.C, SHIFT.DUAL.L, NORMALIZE, MULTIPLY, and DIVIDE) or a pair comprised of the designated register and the shift-extension register, R.37 (SHIFT.RE.L, SHIFT.ER.L, and SHIFT.RE.C). [illegible]ing is done in two 36-bit shifters, with the designated register entering the primary shifter, and the implied register [illegible]ing the extension shifter; after shifting the primary and extension shifters are copied back into the same two registers. The various shift operations specify various ways of connecting the two shifters.

Aleft: Designates the primary register to be shifted. For the even/odd double shifts, aleft should be even, the next-higher-numbered register is the implied second register of the shift. If aleft is an odd-numbered register, then two copies of its value enter the shift; only the primary shifter value is stored (this allows a circular shift of a single odd register; there is no circular shift of a single even register available). For the register/extension double shifts, R.37 is the implied register; there is no difference between an even aleft and an odd aleft.

Mask: The mask, if any, affects only the aleft register itself; the implied register is always unmasked. Masked-out bits of the register enter the shifter as zero bits; their value is not altered by the shift ministep (as in the GEAR normal mode).

Testmode: Testmode, if set, leaves all the General Registers unchanged; only F/F's (and P.7 in an indirect shift) are affected by the execution of a test mode SHIN.

Shift Direction and Amount: The direction must be specified in the ministep as either RIGHT (/) or LEFT (\); the shift amount (in bits) may be either direct (allowed values are identical with the GEAR) or indirect (@). Vacated bit positions are set to zero in all left shifts, to the value of SOS in all right shifts.

Indirect Shift: The shift amount is taken from the shift counter, P.7; the actual shift amount is 0,1,2,4,10, or 20 (octal)--whichever is the largest value not exceeding the contents of the pointer. The pointer is decremented by the amount of the shift, and, if the new value is zero, the SHD (Shift Done) pseudo-F/F is set. A paired BRAT ministep can be used to create a one-cycle shift loop to shift by an arbitrary shift amount. Note that an indirect shift cannot be paired with a BRAD ministep, since the MLP cannot modify two pointers simultaneously.

Operations:

SHIFT.SINGLE.L is a single register shift identical to the shifting of a GEAR; this SHIN is useful only for an indirect single register shift.

SHIFT.EO.L, SHIFT.OE.L, SHIFT.DUAL.L, SHIFT.OE.C are the straight even/odd shift operations, differing in the connections between the two shift registers:

EO.L (Even into Odd Linear) -- bits shifted out of the even word (primary shifter) enter the odd word (extension shifter), while bits shifted out of the odd word are lost.

OE.L (Odd into Even Linear) -- bits shifted out of the even word are lost, while bits shifted out of the odd word enter the even word.

DUAL.L -- bits leaving either word are lost.

EO.C (Even and Odd Circular) -- bits shifted out of either word enter the other one.

SHIFT.RE.L, SHIFT.ER.L, SHIFT.RE.C are the equivalent operations performed on the register and R.37, the Extension, as a pair:
RE.L (Register into Extension Linear)
ER.L (Extension into Register Linear)
RE.C (Register and Extension Linear)

MULTIPLY is a single step of a multiplication loop, with the even/odd pair representing the multiplicand and partial product, and the second operand representing the multiplier.
MULTIPLY X BY Y (M.Z) is equivalent to the sequence

```
X1 ← X1 AND 1 * ! X1 is the odd reg paired with X
IF ZSP THEN, BEGIN
    X ← Y + 0 (M.Z)
ELSE
    X ← X + Y (M.Z) ! add Y if LSB of X1 is set
END ;
SHIFT.EO.L X RIGHT 1 (M.Z) ;
```

except for timing, and consequently, F/F values.

DIVIDE is a single step of a division loop, with the even/odd pair representing the dividend and quotient, and the second operand representing the divisor.
DIVIDE X BY Y (M.Z) is equivalent to the sequence

```
IF COF.1 THEN.BEGIN ! the current setting selects
    X ← X - Y (M.Z) ! either subtraction
ELSE
    X ← X + Y (M.Z) ! or addition
END ;
SHIFT.OE.L X LEFT 1 (M.Z) ;
IF COF.1 THEN. BEGIN ! the new setting (from above)
    X1 ← X1 OR 1 ! is the new quotient bit in X1
END ;
```

Except for timing, COF.1 must be properly initialized for a divide loop; subsequent iterations use the value set in the previous iteration.

NORMALIZE is a variant on SHIFT.OE.L in which the language board controls the amount of shifting--and presumably the counting up of the exponent. The operation is undefined on the NULL language board.

Flip-Flops

The following F/F's are used uniformly in all SHIN ministeps:

SOS - on all right shifts (including MULTIPLY) a copy of SOS is brought into vacated bit positions--into the unconnected register in a linear shift; into both registers in the dual shift; not used in a circular shift.

SHE - on all linear left shifts, SHE is set to the value of the last bit shifted out of the unconnected register. Not affected by circular or dual shifts.

SOF - on all linear left shifts, SOF is set if any bit shifted out of the unconnected register is different from the setting of SOS. Not affected by circular or dual shifts. SOF is never cleared by a shift.

SHD : pseduo-F/F which is valid only during an indirect shift cycle. SHD is set only during a NORMALIZE cycle.

NMD - pseudo-F/F valid only during a NORMALIZE cycle.

The following F/F's are associated with the adder, and are affected only by the MULTIPLY and DIVIDE operations.

ZSP, ZRF.1 - reflect a zero sum (ZSP is valid this cycle; ZRF.1 next cycle).
ZRF.2 - copy of previous value of ZRF.1.
COP, COF.1 - reflect value of the carry out of the adder. (COF 1 is also an inout to DIVIDE.)
COF.2 - copy of previous value of COF.1.

4.2.12 GENT. GENeral data Transfer

This ministep provides data transfer to and from OE Registers. It is used in conjunction with the CE ministep MOVE to provide interengine data transfers.

Syntax:

```
gent ::=
        genta ← gentb ; | genta ← gentc ; | gentb ← genta ;
genta ::=
        gentar | gentar @ P.17 | gentar * P.17 | XBUS
gentar ::=
        R.37 | MISC.37 | A.1777 | XLATOR.777 |
        SUPVLB.377 | LB.1777
gentb ::=
        gentbr | gentbr @ P.17 | gentbr * P.17 | XBUS
gentbr ::=
        R.37 | M.17 | MISC.37
gentc ::=
        number | P.17
```

Examples:

```
R.12 ← 1234567 ;
MISC.12 ← XBUS ;
A.123 ← P.12 ;
M.12 ← LB.1234 ;
XBUS ← A 1234 ;
```

Semantics:

GENT performs direct transfers of the contents of OE registers (Table 4.1). The contents of the right register is copied into the left register. Where XBUS is used as a destination (left) or a source (right), the GENT must be paired with a corresponding MOVE to transfer data in the CE.

4.3 CONTROL ENGINE

OPERANDS

The Control engine (CE) is the ministep decoding and sequencing unit; it includes the current (ministep) address register. The control memory interface, a 16-word subroutine stack (used for both subroutine calls and interrupts), the interrupt and protection mechanisms, 256 individually addressable F/F's, and 16 8-bit pointer registers.

CE ministeps allow conditional branching, including subroutine call and return, and simple F/F and pointer register computations.

MLP-900 interrupts are known as "Action Requests" (AR's). There are 32 action request levels, of which 24 are privileged. Of the eight remaining levels available to user microcode, only two have dedicated functions; the others can be user-defined.

4.3.1 F.377. Flip-Flops

CE.0-CE.37 are 32 bytes of individually addressable F/F's known as F.0 - F.377. These F/F's are divided into a number of functional groups. F.0 - F.277 are real F/F's; F.300 - F.377 are pseudo-F/F's.

F/F's may be set and tested directly by most of the CE ministeps. Other ministeps affect specific F/F's indirectly as a side effect. For example, GEAR and SHIN use and modify one byte of F/F's, and determine some pseudo-F/F's. Language Boards and AR's also use certain F/F's.

Certain F/F's are protected; that is, the user cannot modify them but can reference them. These protected F/F's are indicated in the tables and text below by an asterisk (*) to the left of the F/F name.

Table 4.3 lists all the F/F's. The F/F number is the sum of the numbers at the top of the column and in the extreme left row in which the F/F is located. Where the F/F number appears (e.g., F.135), the F/F is unassigned; where three dashes (---) appear, it is unimplemented.

The pseudo-F/F in CE 30 (F.300-F.307), plus SHD (F.353), reflect conditions which arise in the current cycle, and are defined only when the appropriate ministeps are being executed; <u>all</u> <u>other</u> <u>F/F's</u> reflect conditions as of the beginning of the current cycle. A reference to any F/F in CE.30 causes a one-cycle "hiccup"; the cycle requires two clocks to execute.

Table 4.3

Flip-Flops (Names and Groups)

	F.0	F.40	F.100	F.140
00	GI.0	LBC.0	POWER*	COF.1
01	.1	.1	PANIC*	.2
02	.2	.2	OPAR*	ZRF.1
03	.3	.3	EPAR*	.2
04	.4	.4	SOVF*	F.144
05	.5	.5	SUNF*	SHE
06	.6	.6	UOVF*	SOS
07	.7	.7	UUNF*	SOF
10	GI.10	LBC.10	CMADR*	ARL.5
11	.11	.11	AERR*	TSIN
12	.12	.12	BERR*	TSL
13	.13	.13	PERR*	ITRAC
14	.14	.14	MMAL*	LBI.0
15	.15	.15	MMNR*	.1
16	.16	.16	MMERR*	.2
17	.17	.17	RMWTIME*	.3
20	GI.20	SLBC.0*	TASK*	SARM.0*
21	.21	.1*	PAGE*	.1*
22	.22	.2*	SUPVF*	F.162*
23	.23	.3*	PROT*	F.163*
24	.24	.4*	VADR*	CKC*
25	.25	.5*	F.125*	TRBY*
26	.26	.6*	F.126*	CKT*
27	.27	.7*	F.127*	MBS*
30	GI.30	SLBC.10*	TRAC	ARL.1*
31	.31	.11*	F.131	.2*
32	.32	.12*	F.132	.3*
33	.33	.13*	LBAR	.4*
34	.34	14*	F.134	MOD.0*
35	.35	.15*	F.135	.1*
36	.36	.16*	F.136	SUPVLB*
37	.37	.17*	F.137	SUPVCT*

* See the AR section following.

Table 4.3 (Continued)

	F.200	F.240	F.300	F.340
00	SI.0	IM.0	COP	SSW.0
01	.1	.1	ZSP	.1
02	.2	.2	---	.2
03	.3	.3	---	.3
04	.4	.4	THZ	.4
05	.5	.5	WAR	.5
06	.6	.6	NMD	.6
07	.7	.7	CCP	.7
10	SI.10	IM.10	TRUE	---
11	.11	.11	---	NPT
12	.12	.12	---	---
13	.13	.13	---	SHD
14	.14	.14	---	OSI.0
15	.15	.15	---	.1
16	.16	.16	---	.2
17	.17	.17	---	.3
20	SI.20	IM.20	OLB.0	ZSI.0
21	.21	.21	.1	.1
22	.22	.22	.2	.2
23	.23	.23	.3	.3
24	.24	.24	CLB.0	.4
25	.25	.25	.1	.5
26	.26	.26	.2	.6
27	.27	.27	.3	.7
30	SI.30	IM.30	CLB.4	ZSI.10
31	.31	.31	.5	.11
32	.32	.32	.6	.12
33	.33	.33	.7	.13
34	.34	.34	.10	TSI.0
35	.35	.35	.11	.1
36	.36	.36	.12	FSI.0
37	.37	.37	.13	.1

The following are real F/F's:

GI.0-37 (F.0-37) General Indicators: available to the user's MINIFLOW for abitrary usage.
LBC.0-17 (F.40-57) Language Board Controls: general-purpose indicators which are also LB inputs.
*SLBC.0-17 (F.60-77) Supervisor Language Board Controls

----

* See the AR section following.

*POWER; PANIC; OPAR;...(F.100-127) Action Requests
TRAC; LBAR;...(F.130-137) AR's (user level): Each F/F represents a specific pending AR which causes a microcode interrupt whenever its appropriate level is enabled. Each bit can be set either by the specific occurrence it represents or by a ministep.*
COF.1,2; ZRF.1,2; SHE, SOS, SOF (F.140-147) Carryout F/F, Zero F/F, Shift Extension, Shift Out Sign, Shift Out Flag: OE-associated (GEAR and SHIN) F/F's; fully described in the GEAR and SHIN sections.
ARL.5 (F.150) AR Lockout: user-level AR lockout.
TSIN (F.151) Target System Inhibit *
TSL (F.152) Target System Lockout *
ITRAC (F.153) Initiate Trace *
LBI.0-3 (F.154-157) Language Board Indicators: four indicators which the Language Board can both sense and set.
*SARM.0,1 (F.160,161) Supervisor AR Masks: control the compare Action Requests.
*CKC (F.164) Clock Control
*TRBY (F.165) Translator Bypass
*CKT (F.166) Check Test
*MBS (F.167) Mask Bank Selector: selects current mask bank.
*ARL.1-4 (F.170-173) AR Lockout: lockouts for privileged AR levels,
*MOD.0,1 (F.174, 175) Mode Bits: stored in Control Memory by a BLOT WCM.
*SUPVLB (F.176) Supervisor LB: selects Supervisor LB.
*SUPVCT (F.177) Supervisor Control: forces MLP-900 into supervisor mode regardless of the mode bit in CM.
SI.0-37 (F.200-237) Target System Interrupt F/F's:*
IM.0-37 (F.240-277) Target System Interrupt Masks*

The following are pseudo-F/F's.

COP (F.300) Carryout Pseudo: see GEAR and SHIN instructions.
ZSP (F.301) Zero Sense Pseudo: see GEAR and SHIN instructions.
THZ (F.304) Through Zero: see BRAD.
WAR (F.305) Wait AR: Wait AR (one of F.133-134) pending.
NMD (F.306) Normalize Done: LB output. See SHIN normalize.
CCP (F.307) Check Carry Pseudo: carryout from the check adder.
TRUE (F.310): always set.
OLB.0-3 (F.320-323) Operating Engine LB outputs
CLB.0-13 (F.324-337) Control Engine LB outputs: sense outputs for the current LB.
SSW.0-7 (F.340-347) Sense Switches: on the MLP control panels.
NPT (F.351) Interrupt Pending: a target system interrupt is pending.
SHD (F.353) Shift Done: see SHIN.
OSI.0-3 (F.354-357) One Sense Indicate: senses a value of -1 (255) in P.0-3, respectively.
ZSI.0-13 (F.360-373) Zero Sense Indicate: senses a value of 0 in P.0-13, respectively.
TSI.0,1 (F.374, 375) Three Sense Indicate: senses a value of 3 in P.0,1, respectively.

----
* See the AR section following.

FSI.0,1 (F.376, 377) Four Sense Indicate: senses a value of 4 in P.0,1, respectively.

4.3.2 P.17. Pointer Registers

There are 16 8-bit Pointer Registers, which can be used in the OE to indirectly address registers (e.g., R.0 @ P.3 is the general register determined by the low-order 5 bits of P.3). The Pointer Registers can be loaded by a MOVE instruction, modified by the BRAD instruction, and tested indirectly through the pointer-sense pseudo-F/F's.

The following pointers have special-purpose functions:

P.0-3: used and modified by the BLOT ministep; otherwise generally available.
P.4-5: no dedicated functions.
P.6: Stack Pointer (See Stack Registers).
P.7: Shift Counter for SHIN (see the SHIN instruction).
P.10-17: Pseudo-pointers set by the current Language Board.

The following pseudo-F/F's are TRUE if and only if the appropriate pointer has exactly the specified value.

OSI.0-OSI.3: sense all ones (i.e., -1 or 377(8)) in P.0 through P.3, respectively.
ZSI.0-ZSI.11: sense zero (0) in P.0 and P.11, respectively.
TSI.0-TSI.1: sense the value three (3) in P.0 and P.1, respectively.
FSI.0-FSI.1: sense the value four (4) in P.0 and P.1, respectively.

If a BRAD modifies a pointer and simultaneously tests that pointer's sense pseudo-F/F's, the old value of the pointer is sensed.

4.3.3 Miscellaneous Registers

The Miscellaneous CE Registers are CE.60-CE.77. Their functions are

MINIFLOW Status Word.

The double register pair (CE.60,CE.61) is the MINIFLOW status word, of which only 2 bits are used.

LB selects the active Language Board set.

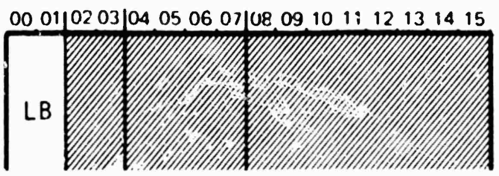


Figure 4.3 MINIFLOW status word.

Current Address Register.

The double register pair (CE.62,CE.63) is the current address register. It contains the address of the current instruction or of the first instruction of a pair. A MOVE to the current Address Register is a no-op.

Exchange Bus In.

CE.64 - CE.67 comprise the Exchange Bus into the CE from the OE. It is addressed as XBUS.0 - XBUS.3 on the left side of an assignment in the MOVE ministep.

Exchange Bus Out.

CE.70 - CE.73 comprise the Exchange Bus out of the CE into the OE. It is addressed as XBUS.0 - XBUS.3 on the right side of the assignment in the MOVE ministep.

Exchange Buses.

The Exchange Buses are pseudo-registers connected to bits 4 to 35 of the Exchange Bus in the OE. XBUS.0 connects to bits 4-11, XBUS.1 to 12-19, XBUS.2 to 20-27, and XBUS.3 to 28-35.

CE.74 - CE.77 do not exist.

4.3.4 S.17. Subroutine Stack

The Subroutine Stack consists of 16 16-bit registers. The Subroutine Stack, together with P.6 (the Stack Pointer), is automatically used in subroutine calls and returns, and AR's. A subroutine call (a BEAD or BENT ministep) branches to the effective address and pushes the return address onto the top of the stack. This is done by incrementing P.6 by 1 and then using the four low-order bits to select the stack word to be loaded with the return address. In addition, if the four low-order bits of P.6 were 16(8) (indicating that the stack is now full), either a supervisor stack overflow (F.104) or a user stack overflow (F.106) is requested, according to the mode of the caller.

Taking an AR consists of pushing the interrupted address onto the stack and branching to the AR entry point, simultaneously setting the appropriate lockout bit (ARL.1-5).

A return (i.e., a BORE ministep) loads the current address register from the top of the stack and then decrements P.6 by 1. If the stack is empty (the four least significant bits of P.6 are 0), and if ARL.2 is off, a stack underflow of the appropriate kind is taken (F.105 if supervisor; F.107 if user). The pointer is left unchanged and the current address (i.e., the address of the BORE instruction) is stacked in S.0.

If the stack is empty but ARL.2 is on, the BORE returns normally, decrementing P.6 as it goes.

OPERATORS

The CE operators are:

- BRAT Branch with Test - Provides conditional jumps.
- BENT Branch and Enter - Provides conditional subroutine calls.
- BORE Branch or Return - Provides conditional subroutine returns.
- BRAD Branch and Modify - Provides loop control.
- BEAD Branch Extended Address - Provides conditional and unconditional subroutine calls and jumps. It has a larger addressing capability than BRAT or BENT.
- BLOT Block Transfer provides loop control together with data transfers with the OE.
- MAST Manipulate Status - Manipulates F/F's.
- MOVE Move CE Registers - This is the general data transfer instruction for the CE.

4.3.5 BRAT. BRAnch with Test

This ministep provides conditional jumps.

Syntax:

```
brat ::=
        /IF booleanexp THEN GOTO relativelabel END;
booleanexp ::=
        ffexp booleanop ffexp |
        (F.377 ← ffexp) | NOT (F.377 ← ffexp)
ffexp ::=
        NOT F.377 | F.377 | TRUE | FALSE
booleanop ::=
        AND | OR | XOR
relativelabel ::=
        sign number | identifier
sign ::=
        + | -
```

Examples:

```
/IF (F.0 ← TRUE) THEN GOTO +200 END;
/IF NOT (F.1 ← FALSE) THEN GOTO -177 END;
/IF F.3 OR F.3 THEN GOTO TAG17 END;
/IF NOT F.4 AND F.5 THEN GOTO +7 END;
/IF F.377 XOR NOT F.377 THEN GOTO -3 END;
/IF NOT F.1 OR NOT F.4 THEN GOTO +166 END;
```

Semantics:

The execution of the BRAT ministep logically [illegible]ace in two parts. First, the boolean expression (booleanexp) is evaluated. If [illegible]lean expression evaluates to true, then the branch is taken, otherwise execution continues with the next instruction.

Boolean expression evaluation. If a store (←) is specified in the boolean expression, the store occurs whether the branch is taken or not. TRUE is the F/F (130) and FALSE is NOT F.130.

Branch destination. The branch destination is relative to the current instruction. The limits on the branch destination are +200 and -177 inclusive. As with all relative branches, addressing beyond or before the ends of control memory will cause a location counter wraparound. Thus a transfer +70 from location 7747 will go to location 0037.

4.3.6 BENT. Branch and ENTer

This ministep provides conditional subroutine calls.

Syntax:

```
 bent ::=
      /IF booleanexp THEN CALL relativelabel END;
```

Examples:

```
/IF (F.17 ← NOT F.1) THEN CALL SUB END;
/IF F.202 OR F.206 THEN CALL +1 END;
/IF F.4 XOR NOT F.77 THEN CALL -27 END;
```

Semantics:

The execution of the BENT ministep is similiar to the BRAT. The only difference is that when the branch is taken, a subroutine entry is executed. The address of the next instruction is loaded into the subroutine stack (S.0 - S.17).

4.3.7 BORE. Branch Or REturn

This ministep provides conditional subroutine returns. (There is no unconditional subroutine return.)

Syntax:

```
    bore ::=
        /IF booleanexp THEN GOTO relativelabel ELSE RETURN END.
```

Examples:

```
  /IF F.1 OR NOT F.3 THEN GOTO -3 ELSE RETURN END;
  /IF TRUE OR F.0 THEN GOTO +1 ELSE RETURN END;
```

Semantics:

The execution of the ministep is identical to BRAT if the boolean expression evaluates to true. If the expression evaluates to false, then instead of continuing at the next instruction, a subroutine return is executed. As with both BRAT and BENT, if a store is indicated, it occurs whether the expression evaluates to true or false.

4.3.8 BRAD. BRanch And moDify pointer

This ministep provides primitive loop control.

Syntax:

```
    brad ::=
        /bradop P.7 BY number ;
        IF ffexp THEN GOTO relativelabel END;
    bradop ::=
        INCREMENT | DECREMENT
```

Examples:

```
      /INCREMENT P.3 BY 7;
    IF F.17 THEN GOTO TAG53 END;

      /DECREMENT P.6 BY 10;
    IF F.103 THEN GOTO +12 END;
```

Semantics.

The BRAD ministep is used for loop and count control. It increments or decrements a counting pointer (P.0 - P.7) and does a conditional relative branch. (Note that BRAD should NOT be executed in a pair with a SHIN ministep using indirect shift.)

<u>Pointer Options</u>. If a noncounting (pseudo) pointer is specified, the contents of the pointer are not modified.

<u>Increment/Decrement Amounts</u>. The largest increment is 7 and the largest decrement is 10. The THrough Zero (THZ) pseudo-F/F is defined only for a BRAD ministep. It is true when the ministep causes the pointer value to pass "through zero"

an INCREMENT which causes overflow or a DECREMENT which causes underflow the new pointer value is correct modulo 400 (8).

4.3.9 BEAD. Branch Extended ADdress

This ministep provides unconditional jumps and subroutine calls. It additionally provides for indexed jumps and subroutine calls. BEAD is the only transfer which can address beyond the relative address range of -200(128) through +177(127).

Syntax:

```
    bead ::=
        bead0 | bead1 | bead2 | bead3
    bead0 ::=
        /IF ffexp THEN transferop label END;
    bead1 ::=
        /transferop label <P.17> ;
    bead2 ::=
        /transferop +1 <P.17> ;
    bead3 ::=
        /IF ffexp THEN transferop sign label END ;
    transferop ::=
        CALL | GOTO
    label ::=
        number | identifier
```

Examples:

```
  /IF F.1 THEN GOTO TAG67 END;
  /IF NOT F.13 THEN CALL 200 END;
  /CALL TAG31 <P.57>;
  /GOTO 277 <P.11>;
  /CALL +1 <P.4>;
  /GOTO +1 <P.11>;
  /IF TRUE THEN GOTO +3711 END;
  /IF NOT F.11 THEN GOTO +TAG67 END;
```

Semantics:

There are four types of BEAD ministeps. The major function of the BEAD is to provide extended addressing capability. All BEADs can address all of control memory All BEADs may optionally execute a subroutine enter. The BEAD types are as follows:

- BEAD0 - Conditional Absolute
- BEAD1 - Absolute plus Pointer
- BEAD2 - Relative plus Pointer
- BEAD3 - Conditional Relative

BEAD0 - Conditional Absolute. If the specified F/F expression is true, control is transferred absolutely to any location (label) in control memory.

BEAD1 - Absolute plus Pointer. Control is unconditionally transferred to the specified location (label) offset by the 8-bit positive quantity in the specified pointer register.

BEAD2 - Relative plus Pointer. Control is unconditionally transferred to the next instruction location p. the 8-bit positive quantity in the specified pointer register. This instruction always tr. sfers in a forward direction.

BEAD3 - Conditional Relative. If the specified F/F expression is true, control is transferred relatively to any location in control memory.

4.3.10 BLOT. BLOck Transfer

BLOT is used to establish loops to transfer blocks of data. The execution of a single BLOT ministep can simultaneously move one word of data, modify some pointers, and conditionally branch. There are six types of BLOTs: one may be used to move data in the OE, two reference the Subroutine Return Stack, and three reference Control Memory (the only instruction that do so).

Syntax:

```
blot::=
  blotcode relativelabel;
blotcode::=
  MOE | RSB | WSB | RCM | WCM | WBP
```

Examples:

```
RCM +7;
WBP -5;
```

Semantics:

There are six types of BLOTs.

- MOE - No CE data is moved (i.e.. step 1 below is null), but steps 2 and 3 (see below) are performed;
- RSB - Move one word from Subroutine Stack to XBUS;
- WSB - Write o word into Subroutine Stack from XBUS;
- RCM - Read one word f om control memory; send to XBUS;
- WCM - Write one word into control memory from XBUS with good parity; and

- WBP - Write one word into control memory from XBUS with bad parity.

Three steps occur simultaneously in all types of BLOT transfers. They are as follows:

1) Moving CE data to or from the XBUS, as specified by the BLOT type

2) Modifying Pointers
Pointer Register modification is identical for all six types of block transfers. P.0. and P.2/P.3 (as a single 16-bit register) are each incremented by one and P.1 is decremented by one.

   Note that the data-move and conditional branch parts of the BLOT, plus any paired OE ministep, use the old values of the Pointer Registers.

3) Conditional Branching.
The conditional branch function is identical for all six types of block transfer. Each time BLOT is executed, P.1 (the word counter) is tested. When a count of one is present, execution continues with the next i[illegible]n. [illegible] P.1 contains any count other than one, the control is transferred to [illegible]ch address. A word count of zero initially loaded into P.1 may be used to [illegible] 256 words.

The data transfer functions for the various BLOTS are

MOE: No CE data is moved, but steps 2 and 3 above are performed.

Example:

Copy the mask registers to memory beginning at the location addressed by R.0 +1.

```
P.0 ← 0              !mask pointer
P.1 ← 20             !loop count
SAD R.0 +1           !Three instruction
SOP M.0@P.0          !store loop
MOE -2;
```

Comment: MOE just provides sequence control. All the data is moved in the OE.

RSB: Read one word from Subroutine Stack into XBUS (XBUS.2; XBUS.3)
WSB: Write one word into Subroutine Stack from XBUS

These BLOT transfers read and write Subroutine Stack words. They are 16 bits wide, read from or written to the rightmost 16 bits (i.e., half-one [H.1]) of XBUS. The low-order 4 bits of P.3 select the stack word (S.0-S.17).

Example:

Copy the subroutine stack to Auxiliary Memory, beginning at A.14000@P.0 (assuming that P.0 already has the correct initial index value).

```
P.1 ← 20                        !loop count
P.3 ← 0                         !Subroutine Stack Pointer
A.1400@P.0 ← XBUS               !Two Instruction
RSB -1                          !GENT/BLOT loop
```

Comment: Now P.0 is 20 greater than at start. P.1 = 0. P.3 = 20;

RCM: Read one word from Control Memory into XBUS
WCM: Write one word into Control Memory from XBUS with good parity
WBP: Write one word into Control Memory from XBUS with bad parity

These BLOT transfers are the only instructions that can reference control memory; they are privileged. They are 36 bits wide, reading and writing to the XBUS using P.2/.3 to select the control memory address.

RCM and WBP are used only in diagnostics. WCM is used for swapping in a new user.

A control memory word is 40 bits wide. Thirty-six data or instruction bits come from the XBUS, two mode bits come from F/F's MOD.0 (F.174) and MOD.1 (F.175). One bit is a parity bit--either good or bad-- and one is unused and is always 0. Parity is generated automatically. WCM generates odd (good) parity; WBP generates even (bad) parity.

RCM will generate a control memory parity AR if parity is bad. If parity is good, then the 36 data/instruction bits are moved to the XBUS; the mode bits cannot be retrieved.

Example:

Load the first 7000 locations in control memory from main memory starting at the location addressed by R.0.

```
P.1 ← 0                   !good for 256 iterations
P.2 ← 0; P.3 ← 0          !control memory address
LOOP:
    FOP R.0 +0;
    WOS XBUS              !Two-instruction loop
    WCM -1                !to read 400(8) words
R.0 ← R.0 + 400;
R.1 B.3 ← P.2;
R.1 XOR 16;
/IF NOT ZSP THEN GOTO LOOP END;
```

4.3.11 MAST. MAnipulate STatus

This ministep manipulates F/F's.

Syntax:

```
    mast ::=
        F.377 ← ffexp booleanop ffexp ; |
        /IF ffexp THEN F.377 ← ffexp END ;
```

Examples:

```
  /F.1 ← F.17 OR NOT F.20 ;
  /F.33 ← NOT F.106 XOR F.13 ;
  /F.106 ← TRUE OR TRUE ;
  /IF F.6 THEN F.111 ← NOT F.4 END ;
  /IF NOT F.11 THEN F.4 ← F.22 END ;
```

Semantics:

There are two types of MAST ministeps, the unconditional and conditional store.

Unconditional MAST. This form of MAST stores a two-term boolean expression into a third F/F. Any F/F's may be used several times. For example, the following will complement F.7:

```
F.7 ← NOT F.7 OR NOT F.7 ;
```

Conditional MAST. This form of MAST is much like the conditional BEAD. If the F/F being tested is true, a store is made. In either case the program continues at the next statement. For example, the following two MAST statements have the same result:

```
  /F.7 ← F.7 OR NOT F.10 ;

  /IF NOT F.7 THEN F.7 ← NOT F.10 END ;
```

4.3.12 MOVE. MOVE CE Registers

This ministep provides data transfer between CE registers; it is also used in conjunction with the OE ministep GENT to provide interengine data transfers.

Syntax:

```
  move ::=
    mi | mff | m | mc | mcl | mdb
  mi ::=
    CE.137 ← number (number) ;
  mff ::=
    CE.137 ← F.377 (number) ;
```

```
m ::=
  CE.137 ← CE.137 (number) ;
mc ::=
  CE.137 ← NOT CE.137 (number) ;
mcl ::=
  CE.137 ← CE.137 [number] ;
mdb ::=
  (CE.137) ← (CE.137) ;
```

Examples:

```
CE.17 ← 5 (7) ;
P.0 ← 17 (75) ;
CE.111 ← F.113 (355) ;
G0R.1 ← G1R.3 (377) ;
XBUS.3 ← NOT CE.4 (11) ;
CE.4 ← XBUS.0 [174] ;
(CE.1) ← (CE.0) ;
5.0 ← (P.0) ;
```

Semantics:

The MOVE ministep moves data within the CE. There are six types of MOVE ministeps. All but one set one CE register, making use of an immediate mask value specified in parentheses or brackets. The mask value is similar to the Mask Register used in the OE; only bits corresponding to one's in the mask are modified. The last type copies an even/odd register pair to another even/odd register pair; the mask is not used.

- Move Immediate - CE.137 ← number (number);
  All masked-in bits of the left CE register receive the corresponding value of the specified right constant operand. As in the GEAR, the mask is specified in ()'s.

- MOVE F/F - CE.137 ← F.377 (number);
  All masked-in bits of the left CE register receive the value of the specified flip-flop.

- MOVE - CE.137 ← CE.137 (number);
  All masked-in bits of the left CE register receive the corresponding value of the specified right CE register.

- Move Complemented - CE.137 ← NOT CE.137 (number);
  All masked-in bits of the left CE register receive the complement of the corresponding value of the specified right CE Register.

- Move and Clear - CE.137 ← CE.137 [number];
  Same as Move (3), but, in addition, the masked-out bits are cleared to zero. Note that the parentheses and brackets (() and []) are used in a manner similar to the GEAR operation.

- Move Double Byte - (CE.137) ← (CE.137);
  Moves one pair of CE registers to another pair of CE registers. The pairs are always an even/odd register pair. Thus (CE.4) and (CE.5) both specify the pair (CE.4,CE.5). When both registers specified are even or both odd, the move will be normal, that is, even to even and odd to odd. However, when the specified registers are one even and one odd, the move will be reversed, that is, even to odd and odd to even. S.0 - S.17 are defined as the appropriate double CE Registers to reference the subroutine stack for the MOVE ministep.

ACTION REQUESTS

There are 32 AR F/F's (F.100-137). Each one is connected to an interrupt location (see address on Table 4.4 below); in addition, each AR is associated with one of five lockout levels (ARL.1-5). ARL.1 locks out all ARs; ARL.2 all ARs on levels 2-5, etc.

When the CE senses the existence of an immediate AR that is not locked out, the current clock cycle is inhibited (i.e., the current instruction/ministep is suppressed) and in the next cycle the MLP-900 takes the AR by performing a call (using the stack to store the interrupted address) to the AR entry point, simultaneously setting the lockout bit of the interrupt level being entered. For those ARs of type "Wait," the AR is left pending until the next CEDE/Wait instruction, when the AR takes place (if not locked out by a higher level), suppressing the CEDE/ Wait instruction. The AR F/F's are not turned off by the act of taking the AR, but must be turned off by the interrupt routine code.

4.3.13 User-Level Action Requests

There are eight AR levels available to the user microcode: three immediate and five wait. Of these eight, two (TRAC and LBAR) have assigned functions.

A user trace function is implemented through the TRAC AR and the ITRAC F/F. Therefore, a TRAC AR routine of the following form:

```
        TRAC ← False;
                  .
                  .        <trace conditions>
                  .
        ARL.5 ← False;
        IF (ITRAC ← True) Return
```

will be entered after every user ministep (except other user AR routines). To initiate tracing, TRAC must be set once.

LBAR is a Language Board output.

4.3.14 Target System Interrupts

A Target System AR takes place only during a CEDE/WIN (which represents the beginning of a new Target System instruction cycles), if any Target System interrupt (F.200-237) and its corresponding mask (F.240-277) are both set; furthermore, all ARL.1-5, TSL, and TSIN lockouts must be clear. In taking a Target System Interrupt, no lockout bit is set. If set by the microcode, TSL prevents all Target System Interrupts until it is cleared by the microcode. TSIN prevents the Target System interrupts at the next CEDE/WIN, at which time TSIN is cleared The pseudo F/F NPT (F.351) is true if any target system interrupt is set and enabled.

Table 4.4

Action Requests

BIT	TYPE	ADDRESS	LEVEL	CAUSE
**				
POWER*	Immediate	7700	ARL.1	Power loss warning
PANIC*	"	7700	"	Interrupt caused by PDP-10 clears immediately
**				
OPAR*	"	7702	ARL.2	Parity error from the odd bank of the Control Memory
EPAR*	"	7704	"	Parity error from the even bank of the Control Memory
SOVF*	"	7706	"	Stack overflow from supervisor mode
SUNF*	"	7710	"	Stack underflow from supervisor mode
UOVF*	"	7712	"	Stack overflow from user mode
UUNF*	"	7714	"	Stack underflow from user mode
**				
CMADR*	"	7716	ARL.3	Control Memory address comparand (Misc.37) matches the Current Address Register while SARM.0 is on
AERR*	"	7720	"	The two adders in the OE differed
BERR*	"	7722	"	Parity error on internal Exchange Bus
PERR*	"	7724	"	Parity error in the translator memory
MMAL*	"	7726	"	Attempt to use VAR beyond that allowed by ALR (Misc.20)

----

* Indicates a privileged AR

Table 4.4 (Continued)

BIT	TYPE	ADDRESS	LEVEL	CAUSE
MMNR*	Immediate	7730	ARL.3	Memory did not respond with correct signal in time designated for Main Memory timeout
MMERR*	"	7732	"	Main Memory parity error
RMWTIME*	"	7734	"	The SOP of a read-modify-write sequence has not occurred within the time designated for RMW timeout
TASK	"	7736	ARL.4	Interrupt from the PDP-10
PAGE*	"	7740	"	A CEDE Wait or Store notes that the last translation is bad
SUPVF*	"	7742	"	Attempt by user mode code to execute a privileged ministep or modify a privileged register
PROT*	"	7744	"	An attempt by user mode code to branch into Microvisor code at other than an entry point
VADR*	"	7746	"	Virtual address comparand (Misc.37) matches VAR while SARM.1 is on
F.125*	"	7750	"	Three unassigned AR's
F.126*	"	7752	"	
F.127*	"	7754	"	
TRAC	"	7756	ARL.5	Set by user microcode, or by ITPAC after a one-cycle delay
F.131	"	7760	"	Two unassigned AR's
F.132	"	7762	"	
LBAR	Wait	7764	"	Language-Board-generated AR
F.134	"	7766	"	Four unassigned AR's
F.135	"	7770	"	
F.136	"	7772	"	
F.137	"	7774	"	
---	Win	7776	---	Some Target System interrupt (TS.0-37) and its mask (IM.0-37) are both set

----
* indicates a privileged AR

APPENDIX A. GPM RESERVED WORDS

Name (Range)	Equivalent
.FIN	
.FOP	
.GAD	
.RMW	
.ROW	
.SAD	
.SOP	
.WIF	
.WIN	
.WOF	
.WON	
.WOP	
.WOS	
.WSS	
A.0 - 1777	OE.2000
A.PG.0 - 3	OE.PG.4
AERR	F.111
AND	
ARL.1 - 4	F.170
ARL.5	F.150
B.0 - 3	
BEGIN	
BERR	F.112
BLOT.0 - 7	
BREAK	
BY	
CALL	
CASE	
CCP	F.307
CE.0 - 377	
CED.0 - 177	
CKC	F 164
CKT	F.166
CLB.0 - 13	F.324
CLEAR	
CMADR	F.110
COF.1 - 2	F.140
COMMENT	
COP	F.300
DATAI	OE.1033
DATAO	OE 1032

Preceding page blank

DECREMENT	
DEFAULT	
DIVIDE	
DO	DO.BEGIN
DO.BEGIN	
ELSE	
END	
ENTRY	
EPAR	F.103
EQUATE	
ERS	F.340
F.0 - 377	
FALSE	
FIN	
FINISH	
FOP	
FSI.0 - 1	F.376
GOR.0 - 17	CE.0
GOR00	CE.0
GOR01	CE.1
GOR02	CE.2
GOR03	CE.3
GOR04	CE.4
GOR05	CE.5
GOR06	CE.6
GOR07	CE.7
GOR08	CE.10
GOR09	CE.11
GOR10	CE.12
GOR11	CE.13
GOR12	CE.14
GOR13	CE.15
GOR14	CE.16
GOR15	CE.17
G1R.0 - 17	CE.20
G1R00	CE.20
G1R01	CE.21
G1R02	CE.22
G1R03	CE.23
G1R04	CE.24
G1R05	CE.25
G1R06	CE.26
G1R07	CE.27
G1R08	CE.30
G1R09	CE.31
G1R10	CE.32

G1R11	CE.33
G1R12	CE.34
G1R13	CE.35
G1R14	CE.36
G1R15	CE.37
GI.0 - 37	F.0
GOTO	
H.0 - 1	
HEXADECIMAL.CODE	
IF	
IM.0 - 37	F.240
INCREMENT	
INDIRECT.0 - 1	
INTO	INTO.BEGIN
INTO.BEGIN	
IOOP.0 - 17	
!TRAC	F.153
LABEL.TABLE	
LB.0 - 1777	OE.6000
LB.PG.0 - 3	OE.PG.14
LBAR	F.133
LBC.0 - 17	F.40
LBI.0 - 3	
LEFT	
M.0 - 17	
MASK	
MBS	F.167
MINUS	
MISC.0 - 37	OE.1000
MMAL	F.114
MMERR	F.116
MMNR	F.115
MOD.0 - 1	F.174
MODE	
MOE	
MULTIPLY	
NAMED	
NMD	F.306
NORMAL.CODE	
NORMALIZE	
NOT	
NPT	F.351
OE.0 - 7777	

OE.PG.0 17	
OLB.0 - 3	F.320
OP.0 - 17	
OPAR	F.102
OR	
ORIGIN	
OSI.0 - 3	F.354
P.0 - 17	
PAGE	F.121
PANIC	F.101
PERR	F.113
PIR	OE.1004
PLUS	
POWER	F.100
PRINTOFF	
PRINTON	
PROT	F.123
R.0 - 37	
RCM	
RETURN	
RIGHT	
RMW	
RMWTIME	F.117
ROW	
RSB	
S.0 - 17	CED.40
SAD	
SARM.0	F.160
SHD	F.353
SHE	F.145
SHIFT.0 - 10	
SHIFT.DUAL.L	
SHIFT.EO.L	
SHIFT.ER.L	
SHIFT.OE.C	
SHIFT.OE.L	
SHIFT.RE.C	
SHIFT.RE.L	
SHIFT.SINGLE.L	
SI.0 - 37	F.200
SIR	OE.1005
SLBC.0 - 17	F.60
SOF	F.147
SOP	
SOS	F.146
SOVF	F.104

SSW.0 - 7	F.340
SUNF	F.105
SUPVCT	F.177
SUPVF	F.122
SUPVLB	F.176
SUPVLB.0 - 377	OE.5400
SWITCHON	
TASK	F.120
TEMPORARY	
TEST	
THEN	THEN.BEGIN
THEN.BEGIN	
THRU	
THZ	F.304
TITLE	
TRAC	F.130
TRBY	F.165
TRUE	
TSI.0 - 1	F.374
TSIN	F.151
TSL	F.152
UOVF	F.106
UUNF	F.107
VADR	F.124
WAR	F.305
WBP	
WCM	
WIF	
WIN	
WOF	
WON	
WOP	
WOS	
WSB	
XBUS	OE.4000
XBUS.0 - 3	
XLATOR.0 - 777	OE.4400
XLATOR.PG.0 - 1	OE.PG.11
XOR	
ZRF.1 - 2	F.142
ZSI.0 - 13	F.360
ZSP	F.301

APPENDIX B

THE GPM COMPILER

The GPM Compiler is a fairly large program written to run under TENEX. This appendix describes use of the compiler, its listing formats, and the INCLUDE feature.

B.1 Using the GPM Compiler

GPM is available as a TENEX subsystem, under the name GPM. The GPM command prompt is "::"; commands consist of a single letter, and are executed immediately. The "C" (compile) command prompts for its source, binary, and listing files. Compilation begins as soon as the last file is confirmed. Using NIL: for the binary file and/or the listing file speeds up compilation considerably and is recommended if either file is not needed.

Example:

```
@GPM

MLP-900 Language System
  Type ? for help
MONDAY, NOVEMBER 11, 1974 14:29:01-PST
USED  0: 0: 0. 5 IN  0: 0: 1.45
Compiler Version GPM.4.74.9

::H     HEXADECIMAL.CODE MODE TRUE
::L     LABEL.TABLE MODE TRUE
::C
source file:PROGRAM.GPM;6 [Old version]
binary file:PROGRAM.BIN;6 [Old version]
listing file:PROGRAM.LST;1 [New version]

↑L
%PROGRAM.NAME        GPM.4.74.9        11-NOV-74 14:30:57 Pg 20 %
        %**No Errors Detected**%

::Q
MONDAY, NOVEMBER 11, 1974 14:31:02-PST
USED  0: 0: 20.20   0: 2: 2.30
```

If no binary file is desired, the binary file should be output to NIL:. The same is true for the listing file. The compilation will run more quickly if no listing is generated.

The listing can be recompiled without any editing. For this reason, it is possible to compile into the source file name. One should be careful, since the compiler will "correct" all errors in the source and they will not appear after recompiling the listing file.

In addition to the "C" command, there are other GPM commands, as follows:

C Compile. Compiles GPM source program (shown in above example).

F Fast compilation. Sets flag for fast syntax check; no code generation.

H HEXADECIMAL.CODE MODE.*

L LABEL.TABLE MODE.*

N NORMAL.CODE MODE.*

P PRINTON Forces complete listing; sets flag to suppress any PRINTOFF statements in the program source.

Q Quit.

S Switch status. Prints the current switch settings as determined by the commands F, H, L, N, and P.

T Teletype Test Compile. Same as C, except binary file is NIL: and both source and listing file are TTY:

B.2 The INCLUDE Feature

The INCLUDE feature may be used anywhere in any GPM source file. It is simply INCLUDE followed by a standard TENEX file name. Neither the INCLUDE nor the file name, but rather the contents of the specified file, are passed to the parser. INCLUDEd files may INCLUDE other files. It is also good practice when working with INCLUDE files to use the proper directory name within the file, so the file can be used by others.

Example:
```
PRINTOFF
COMMENT sample include file ;
BEGIN NAMED INCLUDE.FILE.SAMPLE
EQUATE R.5 INPUT !setup some register definitions
EQUATE R.13 OUTPUT ;
INCLUDE <OESTREICHER>SQUARE-ROOT.INC
COMMENT if this is used when not connected to <OESTREICHER>
  it will still work ;
END NAMED INCLUDE.FILE.SAMPLE !close any open blocks
PRINTON
```

B.3 The GPM Listing Format

---

* Controls generation of appropriate section of GPM listing. Setting alternates every time the command is entered, and the new value is printed. Initial value is false (i.e., no output).

A complete GPM listing contains four parts as follows:

- The source programs with errors flagged and corrections made (where possible).
- The label table.
- The compiled code listed in octal (normal code).
- The compiled code listed in hexadecimal

Section 3.4 discussed the GPM pseudostatements that affect whether or not these listings are produced. This appendix discusses in detail the contents of each part of the listing.

Source Program

The source listing is primarily a formatted copy of the input with a few changes. the most important is that all % text % comments are lost; only the COMMENT statements and ! comments are maintained, because the compiler uses the % text % comments in the listing file for page headings and for error messages.

The output of the GPM compiler can be fed back into the compiler and processed, usually with fewer errors. As the compiler attempts to correct errors, it either "comments out" offending symbols or adds missing ones. If all the corrections made in the output listing (possible new source) are satisfactory, no recompilation is necessary.

Label Table

The label table is output after the FINISH statement and is contained in %'s. It has three columns: octal location, hexadecimal location, and label name.

Example:

```
    %  LABEL TABLE      %
    %  7702      FC2    TAGA   %
    %  7750      FE8    TAGB   %
```

Octal Code

The code listing comes in five columns. The first is the location of the code word in octal, followed by a flag digit and the op code. The fourth column then contains the instruction coding in octal, which is finally followed by a translation of the single instruction back into a GPM statement. This last column is provided to allow easy reading of the compiled code.

The flag digit is not copied to the MLP-900 by the loader. The 4 and 2 flags make ORIGINs and Labels. The 1 flag is of interest because it marks long immediate instructions and causes the location counter column to skip one.

Example:

```
    %7701 0 BEAD 2 121  7027  /IF TRUE THEN GOTO 7027 END;%
    %7702 1 GEAR 4   0 37 77  R.37 ←R.37 OR NOT 777777777657(M.0);%
    %7704 0 GENT 0   2 33 36  MISC.33 ←R.36;%
```

Hexadecimal Code

The hexadecimal listing is the same as the normal, except that the location and instruction coding appear in hexadecimal instead of octal.

Example:

```
    %FC1 0 BEAD 2 91   E17  /IF TRUE THEN GOTO 7027 END;%
    %FC2 1 GEAR 4  0 1F CF  R.37 ←R.37 OR NOT 777777777657 (M.0);%
    %FC4 0 GENT 0  2 1B CB  MISC.33 ←R.36;%
```

APPENDIX C

HARDWARE INSTRUCTION ENCODING

C.1 INTRODUCTION

MLP-900 ministeps are each contained in 32 instruction bits, occupying the least significant bits of the 36-bit control memory word; the four most significant bits are used only in conjunction with the long immediate OE instruction, where the second word contains a 36-bit literal constant. The first four bits of each ministep constitute the op code, and the next four the sub-op; in general, the op code determines the format of the remaining fields of that ministep. The most significant bit of the op code designates the engine: 0 is an OE ministep, 1 is CE.

Four of the eight possible OE op codes are defined. The other four produce undefined results, but the general flavor of their ministep decoding is the same. In particular, the B operand decode applies to ALL OE ministeps (even defined ministeps which have no B operand); whenever the B operand specifies long immediate data, the following word is taken as a 36-bit literal rather than as a ministep.

C.2 FOR THE OPERATING ENGINE

C.2.1 A Operands

An OE A operand represents a reference to a general register (R.0 - R.37) either as an explicitly stated general register or as an indirect reference through a pointer register (P.0 - P.17). The encoding is shown in Figure C.1.

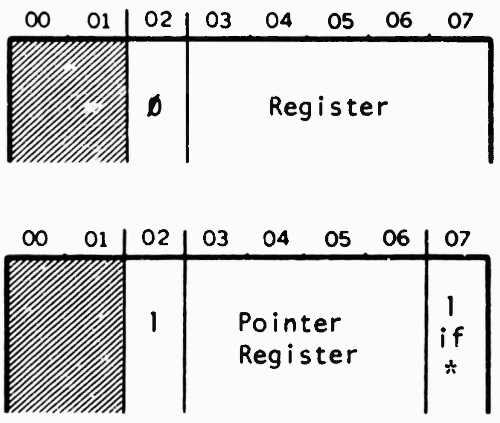


Figure C.1 A operand format.

Examples:

R.13
@ P.11
* P.7

C.2.2 B Operands

An OE B operand represents a reference to a general register (as in an A operand), to a pointer register, or to an immediate operand. The encoding is shown in Figure C.2.

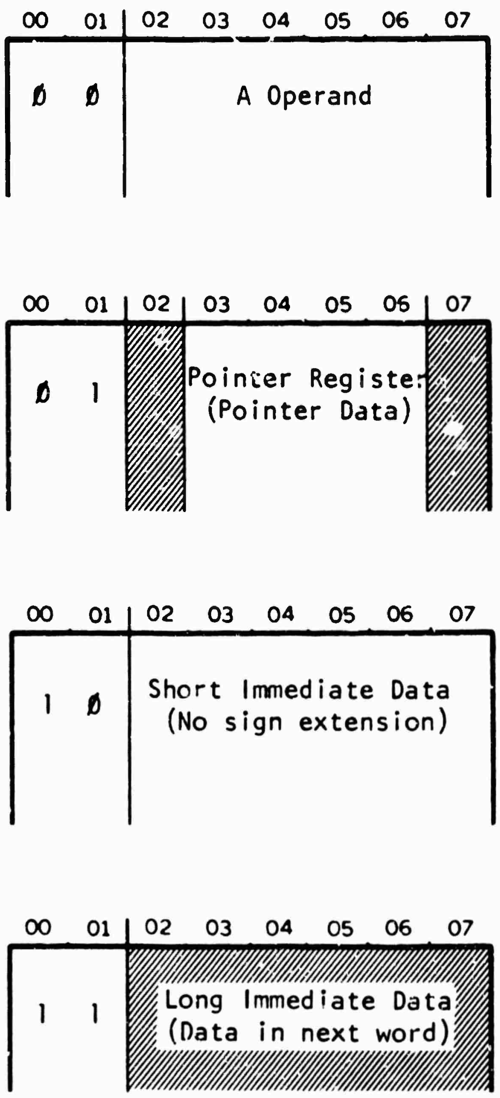


Figure C.2 B operand format.

C.2.3 Shift Amounts

The encoding for shift amounts for GEAR and SHIN ministeps is shown in Table C.1.

Table C.1

Shift Amount Encoding

Shift Amount	Shift Code Left	Shift Code Right
0	10	0
1	11	1
2	12	2
4	13	3
6	14	4
10	15	5
14	16	6
20	17	7

C.2.4 GEAR

aa ← aa op ab shift mask testmode ;

The GEAR internal coding is shown in Figure C.3. The arithmetic codes are listed in Table C.2. The shift amount coding is found in Table C.1. The test mode and clear mode bits are set to 1 to indicate that the mode is active. The A operand (aa) and the B operand (ab) are coded as described in Sections C.2.1 and C.2.2, respectively.

Table C.2

GEAR Arithmetic Codes

Code	Primary Adder Operation
0	aa ← NOT aa OR ab
1	aa ← NOT aa AND ab
2	aa ← ab
3	aa ← aa AND NOT ab
4	aa ← aa OR NOT ab
5	aa ← aa AND ab
6	aa ← aa OR ab
7	aa ← NOT ab
10	aa ← aa XOR NOT ab
11	aa ← aa + ab
12	aa ← ab - aa
13	aa ← aa + ab + COF1
14	aa ← aa - ab + COF1
15	aa ← ab - aa + COF1
16	aa ← aa - ab
17	aa ← aa XOR ab

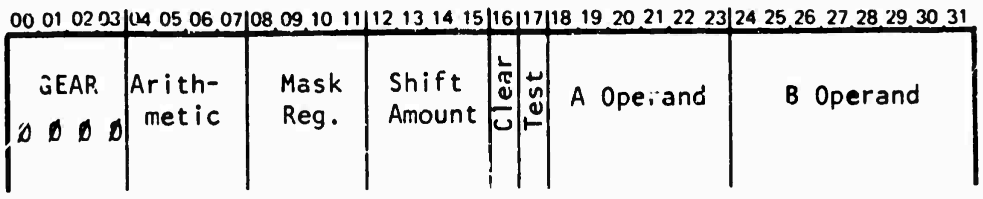


Figure C.3 GEAR ministep.

C.2.5. CEDE

The exchange code determines the CEDE sub-op being executed. The A operand and B operand of all CEDEs, except WOP, SOP, and WOS, are identical to GEAR in the coding of the A and B operands; the Op A Extend and Op A Group are ignored. For these three, the A operand specifies any OE register; the 12-bit address is coded in three sections (the 4-bit group, the 3-bit extension, and the 5-bit register). The operand may also be indirect through a pointer, in which case the indirect addressing is done within the indicated group and the Op A Extend is ignored. These CEDEs ignore the B operand.

Testmode inhibits fetching, storing, translating, and the modification of any register, but waiting and page faulting are still performed.

The subtract bit, when set (i.e., 1), specifies two's complement subtraction instead of addition for those CEDEs that do arithmetic; the subtract bit is ignored for other CEDEs.

For Exchange Codes, see Table C.3 below.

Table C.3
CEDE Exchange Codes

FIN	0
WIN	1
FOP	2
SAD	3
RMW	5
WIF	7
WOF	10
SOP	11
WOP	14
WOS	15
ROW	16
WON	17

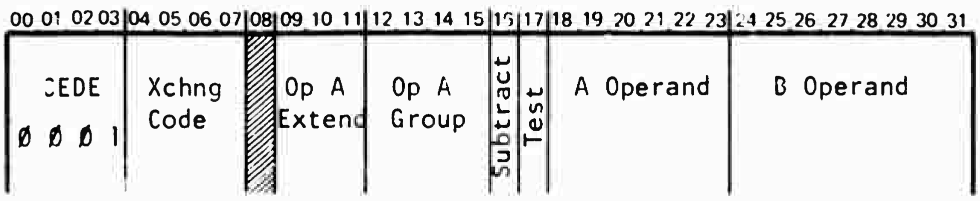


Figure C.4 CEDE ministep.

C.2.6. SHIN

The SHIN internal format is shown in Figure C.5. The shift codes are listed in Table C.4. The Mask, Shift amount, Test, A operand, and B operand (where used) are identical to that of GEAR. Indirect shift, if set, causes the shift amount--though not the shift direction--to be ignored.

Table C.4
SHIN Shift Codes

Code	Shift Operation
0	SHIFT.EO.L (Shift even into odd linear)
1	SHIFT.OE.L (Shift odd into even linear)
2	SHIFT.SINGLE.L
3	SHIFT.DUAL.L
4	SHIFT.EO.C (Shift even and odd circular)
5	SHIFT.RE.L (Shift register into extension linear)
6	SHIFT.ER.L (Shift extension into register linear)
7	SHIFT.RE.C (Shift register and extension circular)
10	NORMALIZE
11	MULTIPLY
12	DIVIDE

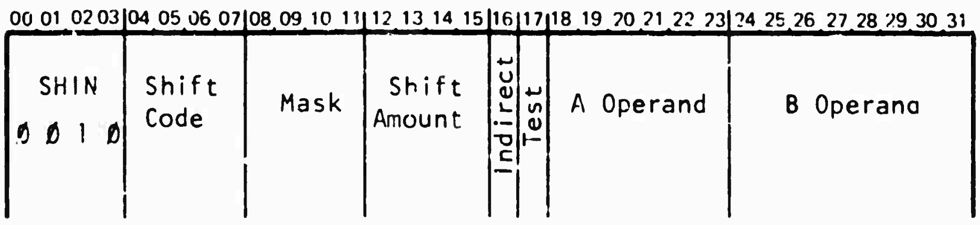


Figure C.5 SHIN ministep.

C.2.7. GENT

gentx ← genty;

The GENT internal coding is shown in Figure C.6. GENT takes two operands: A and B. The direction of the transfer is controlled by the To/From bit as follows:

To/From	Result
0	A ← B
1	B ← A

The 12-bit address for the A operand is coded in three sections as described for CEDE above.

The B operand is coded as described in Section C.2.2, except that when bits 0 and 1 are 0 the operand B group field is used; otherwise, the operand B group field must be zero. The registers addressed by the operand B group field are shown in Table C.5.

If the A operand addresses the mask registers, or the destination is an immediate value or a pointer register, the resulting operation is a no-op.

Table C.5
GENT B Operand Group

Op B Group	Register
0	R.37 - General Registers
1	M.17 - Mask Registers
2	MISC.37 - Misc. Registers
3	XBUS - Exchange Bus

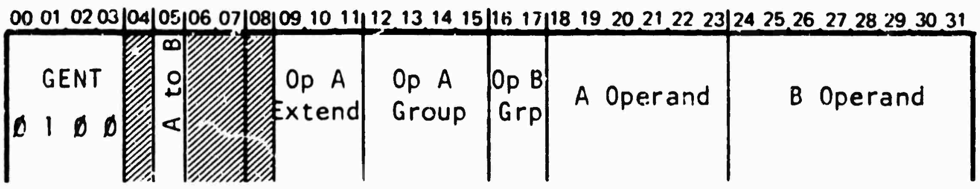


Figure C.6 GENT ministep.

C.3 FOR THE CONTROL ENGINE

C.3.1 Flip-Flops

The F/F's are divided into two groups. F.0 - F.177 are all in group 0, and F.200 - F.377 are all in group 1. Therefore, F.327 is coded as F/F number 127 in group 1. This encoding is shown in Figure C.7.

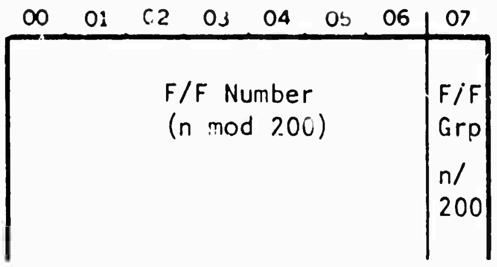


Figure C.7 F.n encoding.

C.3.2 CE Registers

A CE byte register consists of a 4-bit group number and a 4-bit register number within group. This encoding is shown in Figure C.8.

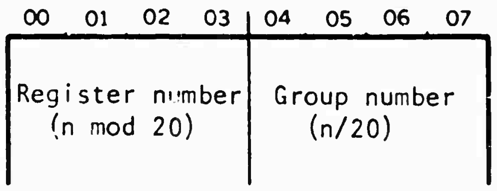


Figure C.8 CE.n encoding.

C.3.3 RELATIVE ADDRESSES

The relative addresses are coded into one byte. They are relative to the continuation address, or the next instruction word. Therefore, a skip is coded as a +1 instead of a +2. The relative offset is two's-complement and signed. The range of the coded possibilities are -200 (10 000 000) through +177 (01 111 111). Because the offset is relative to the continuation address, the effect ranges for relative addresses are -177 through +200.

C.3.4 BOOLEAN EXPRESSIONS

A boolean expression is encoded in two and one half bytes. Two bytes contain the F/F's encoded as shown above. The half byte defines the function. Figure C.9 shows where this information is placed in the instruction word. Table C.6 lists the possible functions.

F/F Expressions - A F/F and its associated true bit are used in BRAT, BENT, BORE, BRAD, BEAD, and MAST to form F/F expressions. If the true bit is on (1), then the actual F/F value is used; if it is off (0), the complement is used.

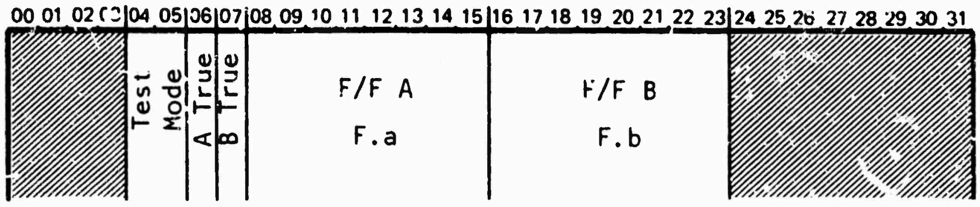


Figure C.9 Boolean expression encoding.

Table C.6
Boolean Expression Types

Test Mode	A True	B True	Boolean Expression
00	0	0	F.b ← NOT F.a
		1	NOT (F.b ← F.a)
	1	0	NOT (F.b ← NOT F.a)
		1	F.b ← F.a
01	0	0	NOT F.b OR NOT F.a
		1	F.b OR NOT F.a
	1	0	NOT F.b OR F.a
		1	F.b OR F.a
10	0	0	NOT F.b AND NOT F.a
		1	F.b AND NOT F.a
	1	0	NOT F.b AND F.a
		1	F.b AND F.a
11	0	0	NOT F.b XOR NOT F.a
		1	F.b XOR NOT F.a
	1	0	NOT F.b XOR F.a
		1	F.b XOR F.a

C.3.5 BRAT

/IF booleanexp THEN GOTO relativelabel END;

The BRAT internal coding (Figure C.10) consists of the BRAT op code, a boolean expression (Figure C.9), and a relative address (Section C.3.3).

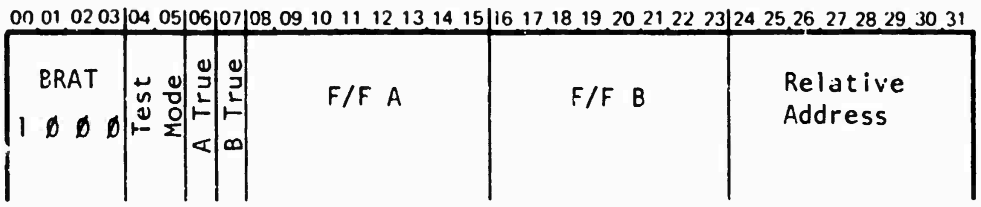


Figure C.10 BRAT ministep.

C.3.6 BENT

/IF booleanexp THEN CALL relativelabel END;

The BENT internal coding (Figure C.11) consists of the BENT op code, a boolean expression (Figure C.6) and a relative address (Section C.3.3).

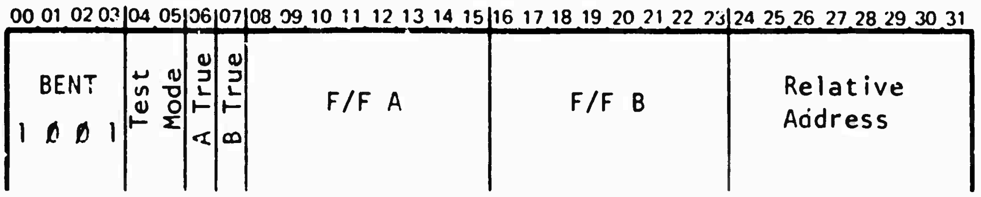


Figure C.11 BENT ministep.

C.3.7 BORE

/If booleanexp THEN GOTO relativelabel ; ELSE RETURN END;

The BORE internal coding (Figure C.12) consists of the BORE op code, a boolean expression (Figure C.6) and a relative address (Section C.3.3).

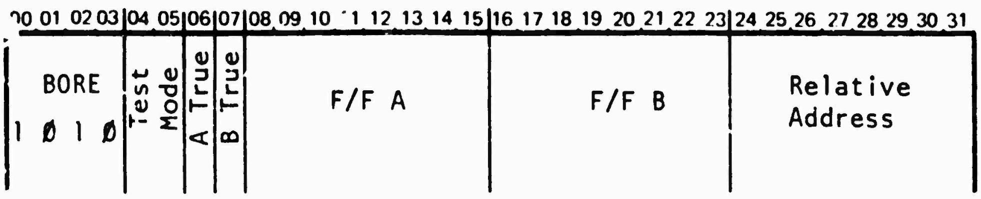


Figure C.12 BORE ministep.

C.3.8 BRAD

/bradop P.7 BY number; IF ffexp THEN GOTO relativelabel END;

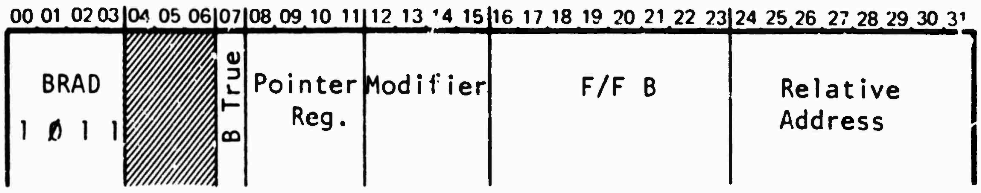


Figure C.13 BRAD ministep.

C.3.9 BEAD

bead0 I bead1 I bead2 I bead3

There are four types of BEAD; they may all be used as a CALL or a GOTO. The Enter bit shown in the four figures below control this. If Enter equals 1, the CALL is done instead of a GOTO.

C.3.9a. BEAD0

/IF ffexp THEN transferop label EN

The BEAD0 internal coding (Figure C.14) consists of a BEAD0 op code, a F/F expression (Section C.3.4) and a 16-bit absolute address.

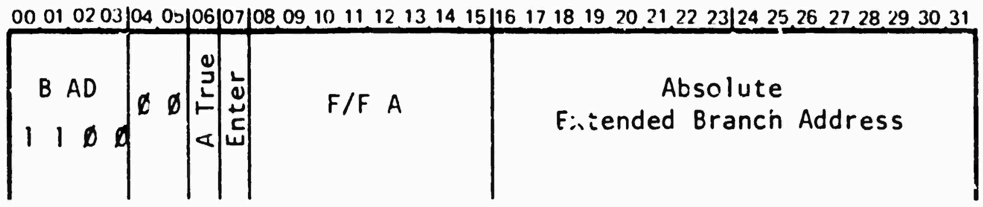


Figure C.14 BEAD0 ministep.

C.3.9b. BEAD1

/transferop label <P.17>;

The BEAD1 internal coding (Figure C.15) consists of a BEAD1 op code, a pointer register number, and a 16-bit absolute address.

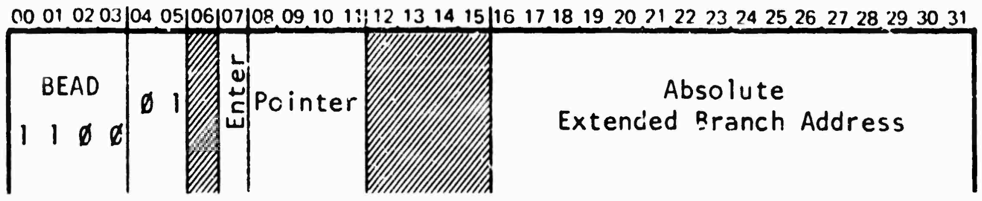


Figure C.15 BEAD1 ministep.

C.3.9c. BEAD2

/transferop +1 <P.17>;

The BEAD2 internal coding (Figure C.16) consists of a BEAD2 op code and a pointer register number.

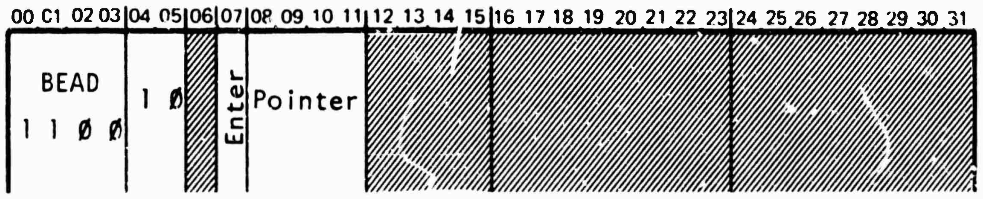


Figure C.16 BEAD2 ministep.

C.3.9d. BEAD3

/IF ffexp THEN transferop sign label END;

The BEAD3 internal coding (Figure C.17) consists of a BEAD3 op code, a F/F expression, and a 16-bit two's-complement relative address. All relative addresses are relative to the next instruction.

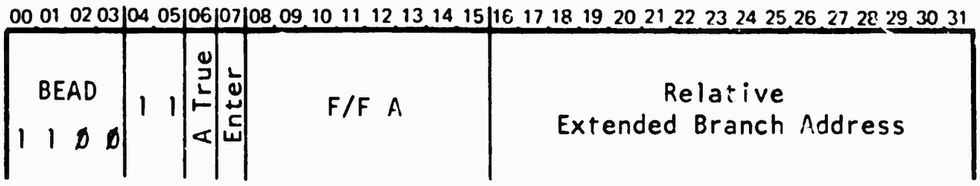


Figure C.17 BEAD3 ministep.

C.3.10. BLOT

blotcode relativelabel

The BLOT internal coding (Figure C.18) consists of the BLOT code and the relative address (Section C.3.3).

Code	Mnemonic
0*	RCM
1*	WCM
2	RSB
3	WSB
4	MOE
5*	WBP

An asterisk (*) indicates a privileged code.

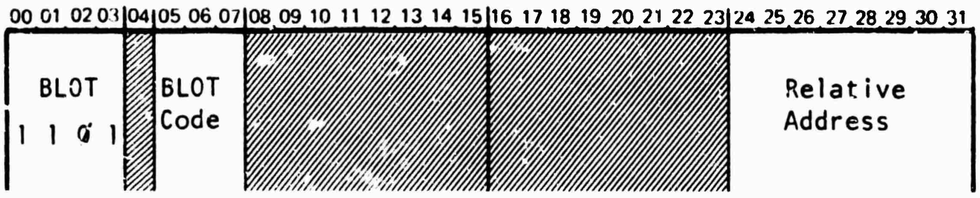


Figure C.18 BLOT ministep.

C.3.11. MAST

F.377 ← ffexpa booleanop ffexpb;
/IF ffexpb THEN F.377 ← ffexpa END;

The MAST internal coding (Figure C.19) consists of a MAST op code, a logical function, two F/F expressions. and a result F/F. The MAST logical function are:

Table C.7: MAST Logical Codes

0	IF ffexpb THEN result ← ffexpa
1	result ← ffexpa OR ffexpb
2	result ← ffexpa AND ffexpb
3	result ← ffexpa XOR ffexpb

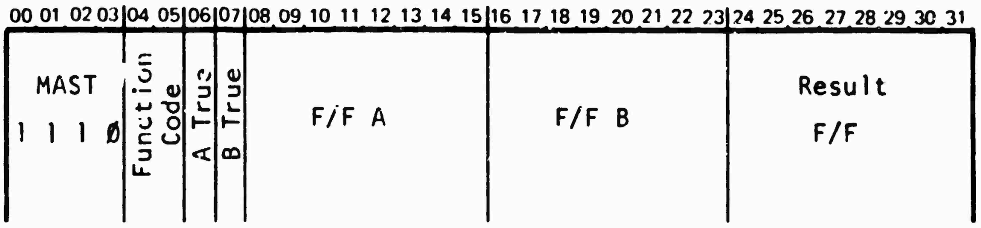


Figure C.19 MAST ministep.

C.3.12. MOVE

```
move ::=
     mi I mff I m I mc I mcl  I mdb
mi ::=
     CE.137 ← number (number) ;
mff ::=
     CE.137 ← F.377 (number) ;
m ::=
     CE.137 ← CE.137 (number) ;
```

```
mc ::=
     CE.137 ← NOT CE.137 (number) ;
mcl ::=
     CE.137 ← CE.137 [number] ;
mdb ::=
     (CE.137) ← (CE.137)  ;
```

The From Address is a constant in the case of MOVE immediate; a F/F in the case of the MOVE F/F; and a CE register for the other four MOVE's. The To Address is always a CE register. The Immediate Mask is an 8-bit constant; it is not used in the MOVE double byte.

Table C.8: MOVE Codes

Code	Mnemonic
0	MSI
1	MOM
2	MAR
3	MAC
4	MCL
5	MDB

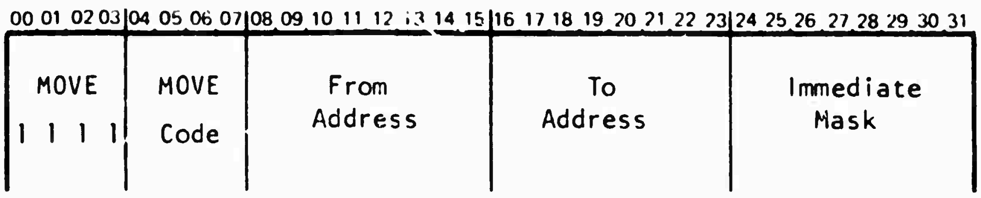


Figure C.20 MOVE ministep.

APPENDIX D. I/O INTERFACE

D.1 Introduction

The I/O interface between the MLP-900 and the PDP-10 contains four registers:

Command/status register	Misc.34
DATAO register	Misc.32
DATAI register	Misc.33
IPL address register	Not addressable

The MLP-900 can read or write these registers as part of the OE miscellaneous register group; writing these registers is allowed only in Microvisor mode. The PDP-10 can read or write these registers via the CONO/I and DATAO/I instructions.

The MLP-900 is recognized as two devices on the I/O bus, MLPA and MLPB, with MLPA intended for all normal communication and MLPB for assistance in saving and restoring the state of the interface.

D.2 Command/Status Register (Misc.34)

The command/status register is a 27-bit register, as shown in Figure D.1.

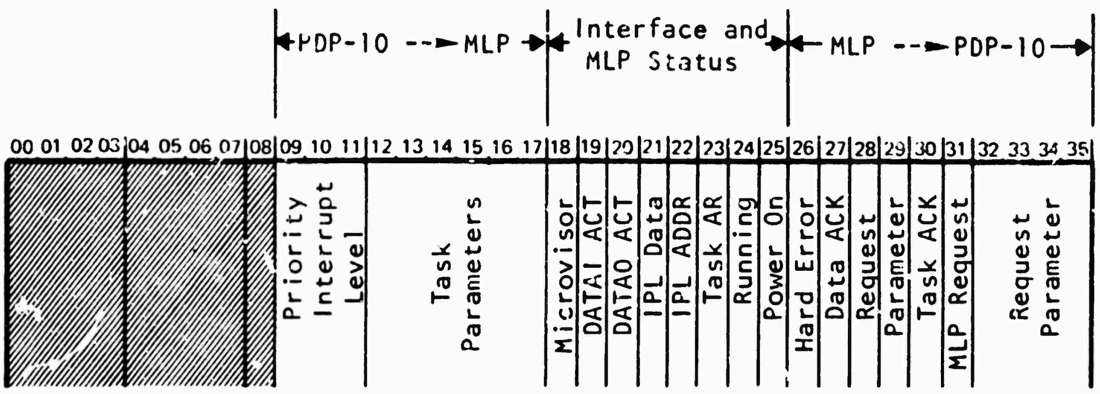


Figure D.1 Command/Status register format.

Bits

9-11	Priority interrupt level
12-17	Task parameter provided by the PDP-10 along with a task AR
18	Microvisor mode

19	DATAI active	I set by writing the register
20	DATAO active	I reset by reading the register
21	IPL data mode	
22	IPL address mode	
23	Task AR pending (F.120)	
24	MLP running (F.164)	
25	MLP Power Up	
26, 27, 30, 31	Priority interrupts: any one causes an attempt to interrupt over the I/O bus 26: Hard error PI 27: Data ack PI 30: Task ack PI 31: MLP request PI	
28, 29, 32-35	Request parameter: expanding on the MLP request PI	

D.3 DATAO (MISC.32) and DATAI (MISC.33)

DATAO and DATAI are both a 36-bit data transmission registers, usable in either direction. Each is accompanied by an active bit in the command/status register. Writing into one of these sets active; reading it resets active (without altering the data). Note that an MLP-900 user may read these registers and, in so doing, clear the active condition.

D.4 MLP-900 Interface Manipulation

The MLP-900 can read the command/status register and the DATAO and DATAI registers via a GENT ministep. In addition, if SUPVLB (F.176), the following command/status fields are available directly:

● Task parameter (bits 12-17)	P.17
● DATAI active (19)	F.326
● DATAO active (20)	F.327
● Hard error PI (26)	F.320
● Data ack PI (27)	F.321
● Task ack PI (30)	F.322
● MLP request PI (31)	F.323

In Microvisor state, the MLP-900 can load any of these three registers via a GENT ministep. Writing the command/status register loads only bits 26-35; bits 0-25 cannot be written directly. Furthermore, if the MLP request PI (bit 31) is zero (new value), the MLP-900 request parameter (bits 28, 29, 32-35) is ignored; that field of the command/status word is cleared. Setting one or more of the four PI bits (26, 27, 30, 31) causes the MLP-900 to interrupt the PDP-10 (if its priority interrupt level is not zero); while their names are function-suggestive, the four PI bits perform identically.

D.5 PDP-10 Interface Manipulation

The PDP-10 recognizes the MLP-900 as two devices on the I/O Bus: MPLA is device 424 and MLPB device 434.

The PDP-10 DATAI and DATAO operations transfer 36-bit values to and from the DATAI and DATAO registers; the active bits are set by a DATAO operation and reset by a DATAI operation.

DATAO-A loads DATAO, and DATAI-A reads DATAI. MPLB is "reversed"; DATAO-B loads DATAI, and DATAI-B reads DATAO.

The PDP-10 CONI and CONO operations transfer 18 bits to and from the command/status register.

CONO-A, MLPA (Commands Out)

Bits	Function
18-20	New priority interrupt level*
21	Set IPL mode
22	Set panic AR (F.101);
23	Set task request (F.120)
24	Set/reset clock (F.164)
25	Reset interface
26	Reset hard error PI
27	Reset data ack PI
28	Reset task ack PI
29	Reset MLP request PI
30-35	New task parameters

CONI-A (Status In)

Bits	Function	
18-25	Bits 18-25 of command/status register	
26,27	Bits 26, 27 of command/status register	I the PI
28,29	Bits 30, 31 of command/status register	I bits
30-35	Bits 28, 29, 32-35 of command/status register; the MLP-900 request parameter	

----

* These two fields are used to alter the appropriate Command/Status fields only if either bit 22 or bit 23 is set in this CONO; otherwise the command/status fields are cleared.

CONO-B, MLPB is a NOP

CONI-B (Read Commands)

<u>Bits</u>	<u>Function</u>
18-20	Priority Interrupt Level
21,22	Zero
23,24	Bits 23, 24 of Command/Status
25-29	Zero
30-35	Bits 12-17 of Command/Status (PDP-10 input parameter)

In general, the MLPB is needed only to save the state of the interface; all "normal" communication is done via MLPA.

D.6 IPL MODE

IPL mode is used to load MLP-900 control memory directly over the I/O bus. IPL mode is initiated by a CONO-A which sets IPL mode (bit 21). This puts the MLP-900 into IPL address mode; the next DATAO-A loads the IPL address register and puts the MLP-900 into IPL data mode. Subsequent DATAO-A's are used to load successive control memory locations, with the mode set to 2 (supervisor mode); the IPL address register is incremented prior to each control memory store.

IPL mode is terminated by any CONO-A. If that CONO itself sets IPL mode, then the MLP-900 is back in IPL address mode.

APPENDIX E. LANGUAGE BOARDS

An MLP-900 language board consists of a pair of boards (one from the OE, one from the CE) which fit into one of four pairs of slots available. The list of available inputs and outputs for each board is fixed (and is identical for each of the four language board positions). The board must obey general MLP-900 hardware conventions regarding board selection, clock time requirements, and the like; the actual construction of language boards must be done by MLP-900 personnel.

The primary functions of the OE board are as follows:

- (Virtual) address transformation for all memory addresses.
- CEDE/WIN and WIF implementation (including indexing, conditional operand fetching, and op code breakout).
- LB register maintenance.

The primary functions of the CE board are as follows:

- Decode of SIR and PIR into pseudo-F/F's and pointers.
- Definition of normalization.

Figure E.1 depicts all the signal lines available to the language board pair. Most of the input lines are the contents of specific registers, a ministep decode signal (indicating the execution of a particular ministep), or a hardware bus. The outputs are divided into pseudo-registers available to the microcode and control signals directed to the MLP hardware.

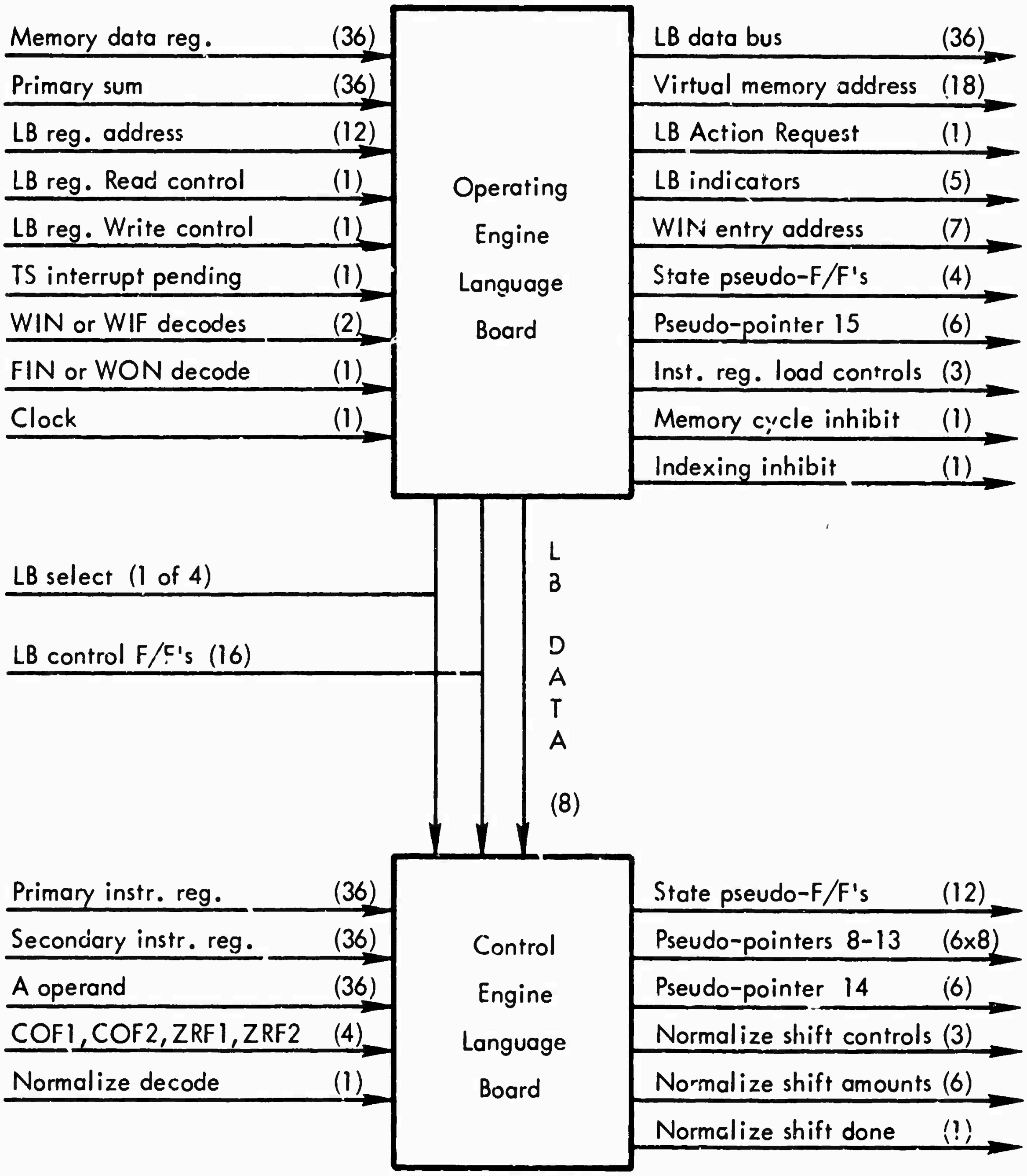


Figure E.1 Language board input/output signals

The input signals are as follows:

Memory data register: MDR

Primary sum: Output of the OE Primary Adder.

LB register address:
OE A operand address (used for referencing language board registers).

LB register Read/Write control:
Control signals set for transfer from or to the LB register (i.e., LB.1777 in a GENT or CEDE).

TS interrupt pending: NPT pseudo-F/F.

WIN/WIF decode:
Control signals for WIN and WIF, respectively.

FIN/WON decode:
Control signal for either FIN or WON; can be used to distinguish instruction and data memory references if desired.

Clock: The MLP-900 clock pulse (for writing into LB registers).

LB select: Decode of the LB select; turns the LB "on".

LB control F/F's: LBC.17 F/F's.

Primary/secondary instruction registers:
PIR and SIR, respectively.

A operand: The OE A operand (for normalization, presumably).

COF1, COF2, ZRF1, ZRF2: The F/F's.

Normalize decode: Control signal for a SHIN Normalize.

The output signals and their definitions on the "null" language board are as follows:

LB data bus:
Used for OE A operand in WIN/WIF, and for the register value in LB register Read operation.
[NULL: Undefined]

Virtual memory address:
The address which actually goes into VAR should this ministep be an address-defining CEDE; the address is presumably a simple transformation of the primary sum. Note that there is no associated control signal.
[NULL: 18 least significant bits of primary sum]

LB Action Request:
    Control signal to set LBAP, F.133.
    [NULL: Never set]

LB indicators:
    Control signal and 4 data bits for LBI.3 F/F's (if control signal is set, the data goes into the four F/F's).
    [NULL: Never set]

WIN entry address:
    Branch address for the WIN ministep (op code breakout), any even location from 0 to 126 (376 octal).
    [NULL: Undefined]

State pseduo-F/F's:
    OLB.3 F/F's.
    [NULL: Undefined]

Pseudo-pointer 15:
    P.17, which is limited to the range 0 through 63.
    [NULL: Undefined]

Instruction register load controls:
    Control signals governing loading of PIR and SIR during WIN.
    [NULL: Undefined]

Memory cycle inhibit:
    Control signal for Fetch inhibit during WIN and WIF.
    [NULL: Undefined]

Indexing inhibit:
    Control signal for B operand inhibit during WIN and WIF.
    [NULL: Undefined]

State pseudo-F/F's:
    CLB.14 F/F's.
    [NULL: Undefined]

Pseudo-pointers 8 - 14:
    P.10 through P.16, of which P.16 is limited to the range 0 through 63.
    [NULL: Undefined]

Normalize shift controls:
    Control signals for normalize shift amount (if amount is indirect).
    [NULL: Undefined]

Normalize shift amounts:
    Increment to P.7 during a Normalize ministep.
    [NULL: Undefined]

Normalize shift done:
The NMD pseudo-F/F.
[NULL: Undefined]

PEFERENCES

1 Bobrow, D. G., J. D. Burch, D. L. Murphy, R. L. Tomlinson, "TENEX, A Paged Time-Sharing System for the PDP-10," *Communications of the ACM*, Vol. 15, No. 3, March 1972, pp. 135-143.

2 Meyer, T. H., J. R. Barnaby, W. W. Plummer, *TENEX Executive Language Manual for Users*, Bolt Beranek and Newman, Inc., Cambridge, Massachussetts, April 1973.

3 *MLP-900 Multilingual Processor--Principles of Operation*, STANDARD Computer Corporation, Santa Ana, California, 1970.

4 *Annual Technical Report*, May 1972-May 1973, USC/Information Sciences Institute, ISI/SR-73-1.

5 *Annual Technical Report*, May 1973-May 1974, USC/Information Sciences Institute, ISI/SR-74-2

6 Oestreicher, Dcnald R., *A Microprogramming Language for the MLP-900*, USC/Information Sciences Institute, ISI/RR-73-7, June 1973.

7 *DEC System-10 Assembly Language Handbook*, Digital Equipment Corporation, Maynard, Massachusetts, 1972.

8 *TENEX User's Guide*, Bolt, Beranek and Newman, Inc., Cambridge, Massachusetts, January 1973.

INDEX