DOLPHIN

MICROASSEMBLER

9 October 1981

by

Edward Fiala

Xerox Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, California, 94304

Filed on: [Indigo]<D0Docs>D0Microassembler.Press
Sources on: [Indigo]<D0Source>D0Microassembler.Dm

This manual describes the Dolphin microassembly language, based upon the 16 May 1979 release of the Dolphin Hardware Manual, and hardware changes up to the release date of this manual.

TABLE OF CONTENTS

1. Preliminaries

The machine now called the Dolphin was at one time called the D0, so these two names are used somewhat interchangeably within this manual.

The Dolphin microprogramming language is implemented as a set of definitions on top of the machine-independent assembler Micro; Micro is an Alto program, so assemblies are carried out either on an Alto or on a Dolphin emulating an Alto. The assembly language is based upon the machine description in the 16 May 1979 release of D0 Hardware Manual and hardware changes that have occurred up to the release date of this manual.

Files referred to in this manual are as follows:

Documentation	*When Using the Assembler*
[Indigo]<D0Docs>	[Indigo]<D0Source>
D0Manual.Press	D0Lang.Mc
D0MidasManual.Press	[Maxc2]<Alto>
D0Gotchas.Press	Micro.Run
[Maxc2]<AltoDocs>	MicroD.Run
Micro.Press	

The assembly language is defined by D0Lang.Mc. I have tried to make D0Lang.Mc and this documentation complete, so you should not need to refer to the Micro manual or study D0Lang for further details, except where noted here.

Micro flushes Bravo trailers, so you can use Bravo formatting if you want to. However, the cross reference program, MCross, which is expected to produce primary microprogram documentation, does not handle Bravo trailers. Also, line numbers in Micro error messages may be more difficult to correlate with source statements because of the line breaks inserted by Bravo's hardcopy command. I advise against Bravo formatting for these reasons.

I recommend use of Gacha8 (i.e., a relatively small fixed-pitch font) for printing program listings, and use of Gacha10.Al for editing source files with Bravo. The smaller font is desirable because some statements will be long, and a smaller font will allow you to get these on one text line. Bravo tab stops should be set at precisely 8 character intervals for identical tabulation in Bravo and MCross.

The two relevant lines in USER.CM for BRAVO are:

```
FONT:0 GACHA 8 GACHA 10
TABS: Standard tab width = 1795
```

You will probably want to delete the other Font lines for Bravo in User.Cm.

I also recommend that you read the hardware manual through once or twice before programming; you will be referring regularly to the figures and tables in the hardware manual until you become extremely familiar with the machine.

Note: All arithmetic in this manual and in Dolphin microassembler source files is in *octal*.

2. Assembly Procedures

A microassembly is accomplished as follows:

```
Micro/L D0Lang Source1 Source2 ... SourceN
```

This causes the source files "D0Lang.Mc," "Source1.Mc," ..., "SourceN.Mc" to be assembled. The global switch "/L" causes an expanded assembly listing to be produced on "SourceN.LS"; if "/L" is omitted, no listing is made. The assembler also outputs "SourceN.Dib" (intermediate binary and addresses), "SourceN.Er" (error messages, which are also printed on the display), and "SourceN.St" (the Micro symbol table after assembling SourceN.Mc).

In other words, Micro assembles a sequence of source files with default extension ".Mc" and outputs four files whose extensions are ".Dib", ".Er", ".Ls", and ".St". The default name for these is the name of the last source file to be assembled. Direct output to particular files as follows:

```
Micro Sys/L/B D0Lang Source1 ... SourceN
```

causes the four output files to be "Sys.Ls", "Sys.St", "Sys.Dib", and "Sys.Er." These four files are used as follows: The ".Ls" file is the assembly listing which you will usually not want to create--the MicroD listing file mentioned below is generally of greater value. The ".St" file is a dump of the symbol table at the end of assembly; it can be used in subsequent assemblies to recreate the assembly environment, as discussed below. The ".Dib" file (Dolphin intermediate binary) is the file that will be processed by MicroD later. The ".Er" file contains error messages and other assembly information.

A summary of local and global Micro switches is as follows:

Global:	/L	Produce an expanded listing of the output
	/N	Suppress .Mb file output
	/U	Convert text in all source files to upper case
	/O	Omit .St file
Local:	/R	Recover from symbol table file
	/L	Put expanded listing on named file
	/B	Put binary output on named file with extension .Dib. Default symbol table (.St) and error listing (.Er) to named file.
	/E	Put error listing on named file
	/S	Put symbol table on named file
	/U	Convert text in named file to upper case

Assemblies are slow--it should take about 7 minutes to assemble a 4000_{10}-instruction program.

INSERT[file] statements, described later, can be put in source files so you don't have to type as many source files on the command line. However, this will slow assembly because each INSERT makes a separate call on the directory lookup code (about 1 second), but all names on the command line are looked up at once. A better shortcut is to define command files to carry out your assemblies.

After obtaining an error-free assembly from Micro, you must postprocess the .Dib file with MicroD to transform it appropriately for loading by Midas. This is accomplished as follows:

```
MicroD Sys
```

MicroD displays a progress message while churning away, and requires about 60 seconds to process a 4000_{10}-instruction file (longer when large listings are produced). The local "/O" switch directs the output to the named file rather than to the last-named input file (default extension .Mb), so:

```
MicroD NewSys/O Sys
```

puts the output of MicroD for the input file Sys.Dib onto NewSys.Mb.

In this example, there is only one input file for MicroD (Sys.Dib)--it is also possible to assemble source files independently using the symbol table (.St) file produced by Micro to establish a basis point for further assemblies, thereby reducing assembly time. For example, you can build a GlobalDefs.St file as follows:

```
Micro/U D0Lang GlobalDefs
```

Then do all further assemblies as follows:

```
Micro/O/U GlobalDefs/R Sys/B Source1 ... SourceN
MicroD GlobalDefs Sys
```

Preassembling D0Lang and GlobalDefs in this way might save 10 seconds of assembly time.

MicroD can relocate code in IM (but not in any other memories). On very large programs, such as the system microcode, it is possible to proceed as follows:

```
Micro/U D0Lang GlobalDefs
Micro/O/U GlobalDefs/R Source1
Micro/O/U GlobalDefs/R Source2
  ...
Micro/O/U GlobalDefs/R SourceN
MicroD Sys/O GlobalDefs Source1 Source2 ... SourceN
```

where Source1 ... SourceN may assemble IM locations but must not assemble any RM locations referenced externally--internally referenced RM locations may be defined in Sourcex, but this has to be done in a way that avoids assignments that conflict with other sources. In other words, forward and external references are permitted only in instruction branch clauses, so everything else must be predefined in GlobalDefs.St or at the beginning of a source file.

One advantage of this method is that Source1 ... SourceN can be independently maintained without having to reassemble the entire system for every change; another advantage is that it avoids symbol table overflow--the current system microcode is near to overflowing at present. However, many changes to GlobalDefs will necessitate reassembling everything, and when everything is reassembled the total assembly time will be about 9 minutes rather than 7 minutes.

Note that you do not need to do anything special in your source files to declare labels which are exported (defined here, used elsewhere) or imported (used here, defined elsewhere). Micro assumes that any undefined branch symbol is meant to be imported (but gives you the list just so you can check), and MicroD assumes that all labels are exported. MicroD also discards all but the last definition of a name (e.g., the symbol "." is defined in every file as the address of the last microinstruction).

MicroD produces up to seven output files, depending upon the local and global flags specified on the command line. The name for these files is determined as discussed earlier, followed by the extensions given below (i.e., Sys.Mb, Sys.Dls, ... , SysOccupied.Mc):

.Mb	Binary output--data for memories and address symbols for use when debugging with Midas; this file is produced unless the global /P switch appears on the command line.
.Dls	Listing file always produced--it contains the Executive command line and all strings printed on the display while MicroD is running (i.e., progress information and error messages); this is followed by a table showing the number of free locations on each page of IM, and by a list of data and address symbols in each memory (which can be modified by various global and local switches discussed below).
.Regs	Register allocation listing produced if the global /R switch is specified; it lists in numerical order RM locations and address symbols associated with each.
.CSMap	Control store map produced when the global /M switch is specified; when MicroD input consists of a number of modules (i.e., of .Dib files), the .CSMap file will show for each page in the control store (i.e., each page in IMX) the number of words in each module allocated on that page and the number of free locations in the page.
.CSChart	A file showing which .Dib file contained the instruction at each real address, sorted by real address; this is produced only when the global /E switch is specified and is intended for hardware debugging with a logic analyzer.
.absDLS	A file giving the correlation between real and imaginary IM addresses, sorted by real address; this is produced only when the global /H switch is specified and is intended for hardware debugging with a logic analyzer.
Occupied.Mc	A file which can be assembled to reserve all locations occupied by the current image. It contains IMReserve declarations for every location into which MicroD has placed an instruction. The intent is that this file be used when building overlays to run on top of the current image.

A summary of the local and global MicroD switches is as follows:

Global:	/A	List only absolutely-placed IM locations
	/C	Concise listing--list everything except octal contents of IM
	/D	Debug--print a large amount of debugging information
	/E	Produce a .CSChart file (of Every location)
	/H	Produce a .absDLS file (useful for Hardware debugging)
	/I	Ignore OnPage
	/K	Kludge enable--suppress LoadPage consistency checking
	/M	Produce a .CSMap file
	/N	No listing--IM contents and other memories are not listed
	/O	Produce the Occupied.Mc output file containing IMReserve statements for all locations filled

		by MicroD
	/P	Print only--supresses all MicroD actions and just lists all .Dib files
	/R	Produce a .Regs file
	/S	List symbols for all memories (except Version, RVRel, Disp, IMLock, and IMMask, which are consumed by MicroD). /N prevents /S from listing IM symbols
	/T	Trace--print a trace of calls on the storage allocator
	/X	External--allow references to unbound symbols in the .Mb file
Local:	/A	List only absolutely-place IM locations--overrides global setting
	/C	Concise listing--overrides global setting
	/L	List everything--overrides global setting
	/N	Not IM listing level overrides global setting
	/O	Output file
	/V	Version number
	/Z	Specifies scratch file to use instead of Swatee

Global switches are usually specified on the command line as "MicroD/nmo Sys/O ..." but MicroD accepts "MicroD Sys/O ... ~/nmo" as an alternative. This alternate form is useful with command files because it allows varying switches to be specified at the end of the command line. In other words, if one has prepared a command file Foo.Cm containing "MicroD/n Sys/O ... ," the "~" feature allows variant switches via "@Foo ~/nmo" to the Alto Executive.

Only one of the /A, /C, or /N global switches, which control additional material printed in the .Dls file, can be meaningful--when none of these switches is specified a verbose (/L) listing will be produced. The /A, /C, /L, and /N local switches overrule the global switches for a particular file. The ordering of these is as follows: /L is most verbose; /A prints less IM information than /L; /C prints all other memories but not IM; and /N prints neither IM nor other memory information.

MicroD outputs a ".Mb" file, consisting of blocks of data that can be loaded into various Dolphin memories and of addresses associated with particular locations in memories. The memories are as follows:

IM	44-bit x 10000-word instruction memory (also contains 60 bits/word of placement and other information)
RM	20-bit x 400-word register bank memory

In addition, three other memories called VERSION, IMLOCK, and IMMASK are produced by Micro and consumed by MicroD, but these are invisible to the programmer.

There are at present very limited provisions for microcode overlays, as discussed in a later section.

3. Error Messages

During assembly, error messages and assembly progress messages are output to both the display and the error file.

Micro error messages are in one of two forms, like the following:

```
... source statement ...
218...error message

-or-

... source statement ...
```

```
TAG+39...error message
```

The first example indicates an error on the 218th line of the source file. This form is used for errors that precede the first label in the file. The second form is used afterwards, indicating an error on the 39th line after the label "TAG".

Note that the line count measures <cr>'s in the source, so if you are using Bravo formatting in the source files, you may have trouble distinguishing <cr>'s from line breaks inserted by Bravo's hardcopy command.

The "TITLE" statement in each source outputs a message of the form:

```
1...title..IM.address.=.341
```

This message indicates that the assembler has started working on that source file. "IM.address.=.341" indicates that the first IM location assembled in this source file is the 341st in the program. When a number of source files are assembled into a single .Dib file, this message may be helpful in correlating source statements with error messages from the postprocessor, MicroD.

The most common assembly errors result from references to undefined symbols and from setting a single instruction field multiple times (e.g., attempting to use the F1 field twice in one instruction). I do not believe that you will have any trouble figuring out what these messages mean, so no comments are offered here.

After Micro has finished an assembly, it returns to the Executive leaving a message like "Time: 22 seconds; 0 errors, 0 warnings, 20007 words free" in the system window. Only when the error or warning counts are non-zero do you have to look in the .Er file for detailed information about errors.

MicroD errors are discussed in the appendix.

4. Debugging Microprograms

Microprograms may be debugged directly on the hardware using facilities provided by Midas. To debug programs you will need to load Midas and its auxiliary files as discussed in the Dolphin Midas Manual.

Midas facilities consist of actions to boot the Dolphin; load microprograms; set, clear, and examine breakpoints; start, step, or halt the machine, and examine and modify storage. Addresses defined during assembly may be examined on the display. Midas contains a command file facility that allows you to carry out various procedures such as setting up the display and iterating through a sequence of diagnostic microprograms.

5. Cross Reference Listings

The cross-reference program for Dolphin microprograms is a Tenex subsystem called MCross. It is easier to maintain large programs when cross-reference listings are available, so you may wish to store your sources on a Tenex directory and make your listings using MCross.

Obviously, you can only use MCross if you have a timesharing account on Maxc. If not, you will have to do without cross-reference listings until MCross is implemented on Alto (no one is working on that now).

A typical dialog with MCross is given below. The program is more-or-less self-documenting and will give you a list of its commands if you type "?".

```
@MCROSS
Output File:      LPT:GACHA8.EP
Machine:          0                  (selects Dolphin syntax)
Action:           U                  (convert to upper case)
Action:           N                  (read def's, no printout)
File:             D0LANG<esc>
Action:           CL                 (read def's, produce cross ref)
File:             Source1<esc>
Action:           CL
File:             Source2<esc>
Action:           P                  (print operation usage statistics)
Action:           G                  (print global cross reference)
Action:           E
@
```

6. Comments

Micro ignores all non-printing characters and Bravo trailers. This means that you can freely use spaces, tabs, and carriage returns to format your file for readability without in any way affecting the meaning of the statements.

Comments are handled as follows:

"*" begins a comment terminated by carriage return.

"%" begins a comment terminated by the next "%". This is used for multi-line comments.

";" terminates a statement. Note that if you omit the ";" terminating a statement, and, for example, put a "*" to begin a comment, the same statement will be continued on the next line.

Micro's COMCHAR feature provides one method of producing multi-statement conditional assemblies (This method is now obsolete, replaced by the conditional assembly features discussed in the next section). COMCHAR is used as follows. Suppose you want to have conditional assemblies based on whether the microcode is being assembled for a Pilot or Alto-compatible configuration. To do this define "=" as the comment character for Pilot (i.e., COMCHAR[=];) and "#" as the comment character for Alto-compatible. Then in the source files:

```
*= Alto-compatible configuration only
 ...statements for Alto-compatible configuration...
```

```
*= end of Alto-compatible statements
*# Pilot configuration only
 ...statements for Pilot configuration...
*# end of Pilot statements
```

In other words, "*" followed by the comment character is approximately equivalent to "%" and is terminated by the carriage return following its next occurrence.

7. Conditional Assembly

D0Lang defines IF, UNLESS, ELSEIF, ELSE, and ENDIF macros for doing multi-statement conditional assemblies; IF's may be nested up to four levels deep. The syntax for these is as follows:

```
:IF[Display];
  ... statements assembled if Display is non-zero...
:ELSEIF[OldDisplay];
  ... statements assembled if Display is zero and OldDisplay non-zero...
:ELSE;
  ... statements assembled if both Display and OldDisplay are zero...
:ENDIF;
```

Alternatively, UNLESS can be used instead of IF, as follows:

```
:UNLESS[Display];
  ... statements assembled if Display is zero...
:ENDIF;
```

Note that each of the conditional names must be preceded by ":"; the implementation of these is discussed in the Micro manual. Any number of ELSEIF's may be used after an IF, followed by an optional ELSE and a mandatory ENDIF. The arguments to IF and ELSEIF must be integers.

Warning: The :IF, :UNLESS, :ELSEIF, :ELSE, and :ENDIF must be the first characters in a statement. In the following example:

```
FlshCore:
:IF[MappedStorage];
        ... statements ...
:ELSE;
        ... statements ...
:ENDIF;
```

":IF" is illegal because, due to the "FlshCore" label, ":IF" are not the first characters of a statement.

8. Simplified Parsing Rules

After comments, false conditionals, and non-printing characters are stripped out, the rest of the text forms *statements*.

Statements are terminated by ";". You can have as many statements as you want on a text line, and you can spread statements over as many text lines as you want. Statements may be indefinitely long.

However, the size of Micro's statement buffer limits statements to 500_{10} characters at any one time. If this is exceeded at any time during the assembly of a statement, an error message is output. Since horrendous macro expansions occur during instruction assembly, it is possible that instruction statements may overflow. If this occurs, the size of the statement buffer can be expanded (Tell me.).

The special characters in statements are:

"[" and "]"	for enclosing builtin, macro, field, memory, and address argument lists;
"(" and ")"	for causing nested evaluation;
"_"	as the final character of the token to its left;
":"	to put the address to its left into the symbol table with value equal to the current location and current memory, and as the first character of a statement to be evaluated even in the false arm of a conditional;
","	separates clauses or arguments;
";"	separates statements;
"#"	#1, #2, etc., are the formal parameters inside macro definitions;
"01234567"	are number components (all arithmetic in octal).

All other printing characters are ordinary symbol constituents, so it is perfectly ok to have symbols containing "+", "-", "&", etc. which would be syntactically significant in other languages. Also, don't forget that blanks, carriage returns, and tabs are syntactically meaningless (flushed by the prescan), so "T+Q" = "T + Q", each of which is a single symbol.

The debugger *Midas requires all address symbols to be upper case*; since both Micro and MCross have switches that convert all source file characters to upper case, you can follow your own capitalization conventions but must convert to upper case at assembly time using the /U switch. Experience suggests that *consistent capitalization conventions* are desirable, although there is not much agreement on exactly what conventions should be used. In this manual I follow capitalization conventions which you may consider as a non-binding proposal. My convention is as follows:

> The first letter of each word is capitalized.
>
> When a symbol consists of several words run together, the first letter of each subword is capitalized (e.g., "NextData," "StkP").
>
> When a symbol is formed by running together the first letters from several words, then these are all capitalized (e.g., "MNBR,").

Micro builtins, memory names, and important assembly directives that should stand out in the source, such as TITLE, END, IF, etc. are all capitals.

Midas also *limits address symbols to 13 characters in length*; if you assemble longer addresses, you will still be able to load and run your program with Midas, but you won't be able to examine symbols longer than 13 characters.

Also, *avoid using any of the characters "=," "#," "+," "-," and "!" in address symbols*; these are syntactically significant to Midas and may cause trouble when debugging. Finally, *avoid defining symbols that end with the character "@"*; the internal symbols in D0Lang.Mc by convention end with "@," so you might have name conflicts with reserved words if you also define symbols ending with "@."

Statements are divided into *clauses* separated by commas, which are evaluated right-to-left. An indefinite number of clauses may appear in a statement.

Examples of clauses are:

```
NAME,
NAME[ARG1,ARG2,...,ARGN],
FOO_FOO1_FOO2_P+Q+1,               P+Q+1 is referred to as a "source" while FOO_, FOO1_, and
                                   FOO2_ are "destinations" or "sinks".
P_STEMP,
NAME[N1[N2[ARG]],ARG2]_FOO[X],
```

Further discussion about clause evaluation is postponed until later.

9. Statements Controlling Assembly

Each source file should begin with a TITLE statement as follows:

```
:TITLE[Source1];
```

The TITLE statement:

a. prints a message in the .Er file and on the display which will help you correlate subsequent error messages with source statements which caused them;

b. does a SetTask[0] (discussed later) and resets a number of assembly switches.

At the end of a file you may optionally insert an END statement:

```
:END[Source1];
```

The END statement prints a message like the title statement's and checks for unterminated conditionals, but it does not affect the output of the assembler.

Note that the ":" preceding TITLE and END is optional; inserting the ":" will cause statement evaluation even in the false arm of a conditional, so an appropriate message can be printed to help detect :IF's unmatched by :ENDIF's. The "Source1" argument to :END is also optional.

You may at any place in the program include an INSERT statement:

```
INSERT[SourceX];
```

This is equivalent to the text of the file SourceX.MC.

The message printed on the .Er file by TITLE is most helpful in correlating subsequent error messages if any INSERT statements occur either before the TITLE statement or at the end of the file (before the END statement). INSERT works ok anywhere, but it might be harder to figure out which statement suffered an error if you deviate from this recommendation.

In the event you request a listing by putting "/L" in the Micro command line, the exact stuff printed is determined by declarations that can be put anywhere in your program.

D0Lang selects verbose listing output. However, you will generally *not* want to print this listing. The MicroD listing is normally more useful during debugging. If you want to modify the default listing control in D0Lang for any reason, you can do this using the LIST statement, as follows:

```
LIST[memory,mode];
```

where the "memory" may be any of the ones given earlier and the mode the OR of the following:

20	(TAG) nnnnnn nnnnnn (octal value printout in 16-bit units)
10	alphabetically-ordered list of address symbols
4	numerically-ordered list of address symbols
2	(TAG) FF_3, JCN_4, etc (list of field stores)
1	(TAG) nnnn nnnn nnnn (octal value printout)

Note: The listing output will be incorrect in fields affected by forward references (i.e., references to as yet undefined addresses); such fields will be incorrectly listed as containing their default values.

Micro has a recently added TRACEMODE builtin which you may prefer to use instead of an assembly listing for the purposes of debugging complicated macros. TRACEMODE allows symbol table insertions and macro expansions to be printed in the .Er file. See the Micro manual for details about TRACEMODE.

10. Forward References

Micro and D0Lang have an *extremely limited* ability to handle forward references. The only legal forward references are to instruction labels from branch clauses. Anything else must be defined before it is referenced.

11. Integers

Micro provides builtin operations for manipulating 20-bit assembly-time integers. These have nothing to do with code generation or storage for any memories. Integers are used to implement assembly switches and to control Repeat statements. The operations are given in the table below and there is some additional discussion in the Micro Manual:

Set[NAME,OCT]	Defines NAME as an integer with value OCT. Changes the value of NAME if already defined.
Select[i,C0,...,Cn]	i is an integer 0 to n. Evaluates C0 if i = 0, C1 if i = 1, etc.
Add[O1,...,O8]	Sum of up to 8 integers O1 ... O8.
Sub[O1, ... ,O8]	O1-O2-...-O8
IFE[O1,O2,C1,C2]	Evaluates clause C1 if O1 equals O2, else C2.
IFG[O1,O2,C1,C2]	Evaluates C1 if O1 greater than O2, else C2.
Not[O1]	Ones complement of O1.
Or[O1,O2,...,O8]	Inclusive 'OR' of up to 8 integers.
Xor[O1,O2,...,O8]	Exclusive 'OR' of up to 8 integers.
And[O1,O2,...,O8]	'AND' of up to 8 integers.
LShift[O1,N]	O1 lshift N
RShift[O1,N]	O1 rshift N

OCT in the Set[NAME,OCT] clause, may be any expression which evaluates to an integer, e.g.:

```
Set[NAME, Add[Not[X], And[Y,Z,3], W]]
```

where W, X, Y, and Z are integers.

If you want to do arithmetic on an address, then it must be converted to an integer using the IP operator, e.g.:

IP[FOO]	takes the integer part of the address FOO
Add[3,IP[FOO]]	is legal
Add[3,FOO]	is illegal

Some restrictions on doing arithmetic on IM addresses are discussed later.

12. Repeat Statements

The assortment of macros and junk in the D0Lang file successfully conceals Micro's complicated macro, neutral, memory, field, and address stuff for ordinary use of the assembler.

However, using the Repeat builtin may require you to understand underlying machinery--in a diagnostic you might want to assemble a large block of instructions differing only a little bit from each other, and you want to avoid typing the same instruction over and over.

Instruction statements are assembled relative to a location counter called ".". Originally set to 0 , "." is incremented by one every time an instruction is assembled. To do a Repeat, you must directly reference "." as follows:

```
Repeat[20, . [(instruction statement)]];
```

This would assemble the instruction 20 times. If you want to be bumping some field in the instruction each time, you would proceed as follows:

```
Set[X,0];
Repeat[20, . [(Set[X,Add[X,1]], instruction statement)]];
```

where the instruction statement would use X someplace.

For a complicated Repeat, you may have to know details in D0Lang. For this you will have to delve into it and figure out how things work.

13. Parameters

Parameters are special assembly-time data objects that you may define as building blocks from which constants, RM, or IM data may be constructed. Three macros define parameters:

MP[NAME,OCT];	makes a parameter of NAME with value OCT
SP[NAME,P1,...,P8];	makes NAME a parameter equal to the sum of P1,...,P8, which are parameters or integers.
NSP[NAME,P1,...,P8];	makes NAME a parameter equal to the ones complement of the sum of P1,...,P8, which are parameters or integers.

The parameter "NAME" is defined by the integer "NAME!" (The "!" is a symbol constituent added so that a constant, small constant, or RM address can have an identical NAME.), so it ok to use the NAME again as an address or constant. However, you cannot use it for more than one of these.

14. Constants

The hardware allows a constant to be generated on B that is the 10-bit FF field of the instruction in either the left or right half of the 20-bit data path with 0 in the other 10-bit byte.

"Literal" constants such as "322C" or "32400C" may be inserted in instructions without previous definition.

Negative constants such as "-400C", "-32400C", etc. are also legal.

Left-half-word and right-half-word constants may be constructed from integers by applying, respectively, the HiA and LoA operators to a 20-bit integer. An optional second argument to HiA should be an integer in the range 0 to 17; if provided, this argument is or'ed with bits 0:3 of the first to produce the task number and high four address bits for APCTask&APC _. An example using HiA and LoA is as follows:

```
RTemp _ HiA[DispStartLoc,DispTask];          *Bits 0:7 (DispTask in 0:3)
RTemp _ (RTemp) or (LoA[DispStartLoc]);           *Bits 10:17
```

Alternatively, constants may be constructed from parameters or integers using the following macros:

MC[NAME,P1,...,P8];	defines NAME as a constant whose value is the sum of P1...P8 (integers or parameters).
NMC[NAME,P1,...,P8];	defines NAME as the ones complement of the sum.

Warning: The two macros above also define NAME as a parameter. You must not redefine a parameter with the same name as a constant because the binding of the constant is to the name of its associated parameter (i.e., to "NAME!"), not to its value. In other words, if you redefine a parameter with the same name as a constant, you will redefine the constant also.

Because the definition of a constant also defines a parameter of the same name, it is possible to cascade a number of constant definitions to create various useful values. Here is an example of how several constant definitions can be cascaded:

```
MC[B0,100000];
MC[B1,40000];
MC[B2,20000];
MC[B01,B0,B1];
MC[B02,B01,B2];
```

Occasionally, you may wish to create a constant whose value is an arithmetic expression or an expression including an address in RM. Here are several examples of ways to do this:

IP[RAddr]C	A constant whose value is an RM address
Add[3,LShift[X,4]]C	A constant whose value is a function of the integer X

15. SetTask Statements

The hardware OR's various bits of the task number into fields of the instruction to determine which RM addresses are referenced. You must tell the assembler what task is going to execute each section of microcode, so that it can perform the proper error checks and set up instruction fields appropriately.

This is done with a clause of the form:

```
SetTask[n];          *n = the task number (0 to 17)
```

If you want to refer to task numbers symbolically, you can define integers with values equal to the task numbers. For example:

```
Set[DispTask,3];
```

Then use SetTask[DispTask] to refer to the task.

SetTask controls not only the assembly of instructions, but also the allocation of RM addresses to 100-word sections of RM, as discussed in the next section.

Note: The TITLE statement at the beginning of a file does a SetTask[0].

16. Assembling Data for RM

RM addresses are allocated by RV statements in one of the following ways:

```
RV[name, disp, P1, P2, ... , P7];
RV[name, , P1, P2, ... , P7];
RV[name, disp];
RV[name];
RV[name, , value];
RV2[name0, name1, disp0];
RV4[name0, name1, name2, name3, disp0];
```

The first argument of RV is the "name" of the RM address to which you will subsequently refer in instructions.

The second argument "disp" is a displacement between 0 and 77. This specifies the low six bits of the RM address. The top two bits are determined by the top two bits of the task number, declared by the last SetTask statement. If "disp" is omitted, the RM address is allocated at the last location plus 1.

The remaining 7 arguments are parameters or integers summed to determine the value loaded into that location. If all of these are omitted, then the location will be uninitialized.

RV2 and RV4 macros allow definition, respectively, of two consecutive or four consecutive RM locations. The first two (four) arguments are the names of the consecutive registers, and the final argument is the displacement of the first named register (0 to 77); no provision is made for assembling values into registers defined this way. It is convenient to define registers with RV2/4 rather than one-at-a-time, when they are constrained to be consecutive by some usage. For example, target registers for PFetch2/4 should be defined by RV2/4.

Avoid assigning useless initial values to variables because this will prevent the "Compare" function in Midas (which compares the microstore image against what you loaded) from reporting fictitious errors. In a system microprogram (as opposed to a diagnostic), any occurrence of a variable with an initial value is probably a programming error since it requires reloading the microcode to restore the initial value. Hence, you probably should initialize such registers with your program.

> Also, boot microcode does not allow any memories except IM to be initialized from the boot EPROM, and current software for loading microcode overlays is also limited to IM. For these reasons the usual practice is to put code for initializing RM into a "throwaway" page which can be overwritten by something else after it is executed.

The hardware imposes a number of strange constraints upon RM placement. For example, addresses used as base registers must (usually) be even, addresses for PFetch4/PStore4 must (usually) be origined 0 mod 4, and addresses for PFetch2/PStore2 must (usually) be at even locations (exception: stack double-words). Also, RM is partitioned so that only locations 0 to 77 are accessible to tasks 0 to 3, 100 to 177 to tasks 4 to 7, 200 to 277 to tasks 10 to 13, and 300 to 377 to tasks 14 to 17. Tasks 1 to 3 in each group of 4 are further limited because the task number is OR’ed into high address bits in various ways. These constraints will be a source of many program bugs.

You must be careful to assign a "disp" that satisfies all the uses of each RM address. If you screw up, the assembler will indicate an error when you illegally reference the RM location in an instruction.

Note: Suppose that you want tasks 10, 11, 12, and 13 to share a section of microcode but use independent RM locations. Then you do a SetTask[10] before that section of the program, and you allocate a block of RM locations in the range 100-117 and refer to these locations in the program; you also allocate parallel blocks of RM locations in the ranges 120-137, 140-157, and 160-177 for use by tasks 11, 12, and 13, respectively. In this way, the program will do what you want. If the four tasks have some other RM locations that are shared, allocate these in the range 160-177, so that they will be accessible to all four tasks.

Sometimes you may want to use several different names to refer to the same RM location. To do this, define the first name with RV, as above; you can then define synonyms in several ways, one of which is as follows:

```
RM[FOO1, IP[FOO]];
```

This defines the address FOO1 at the same location as the (previously-defined) address FOO.

17. Assembling Data Items In the Instruction Memory

If you do not want to clutter RM with infrequently referenced constants or variables, and if you are willing to cope with the hardware kludges for reading/writing the instruction memory as data, then you can store data items in IM.

To assemble a table of data in the instruction memory:

```
Set[T1Loc,100];
IMData[(TABLE1: LH[P1, ... , P8] RH[P1, ... , P8], At[T1Loc])];
IMData[(LH[P1, ... , P8] RH[P1, ... , P8], At[T1Loc,1])];
...
```

where TABLE1 is an IM address symbol equal to the location of the first instruction in the table, P1, ..., P8 are parameters or integers. LH stores the sum of up to 8 parameters in the left-half of the IM word and RH, in the right-half; these macros do not allow you to specify the remaining four bits of the 44-bit instruction--the assembler will ensure that the data has valid parity by storing a 0 or 1 in the low-order bit of the RX@ field (which is one of the four bits you cannot specify). "At" is discused in the "Placement" section later.

18. General Comments on Instruction Statements

The general forms of an instruction statement are as follows:

```
TAG:     branch clause, T_rmaddr_(A phrase) and (B phrase), function, placement;
TAG:     branch clause, T_A phrase, B phrase, function, placement;
TAG:     branch clause, A phrase, RMAddr_B phrase, function, placement;
    -or-
TAG:     branch clause, PFetch1[rbase, rdest, f2], placement;
TAG:     branch clause, PFetch1[rbase, rdest], f2, placement;
```

where the first three examples are "regular" instructions and the last two, "memory reference" instructions.

Rule: TAG is an optional IM address symbol or label; it must appear first in the instruction statement.

TAG may be referenced from branch clauses in other instructions, as discussed in the "Branching" section, and it will appear in the output file for use in debugging.

Rule: Clause order is totally arbitrary; it doesn’t matter which ones appear first in the statement. However, you might wish to follow a consistent ordering convention for program readability.

Placement clauses, A phrases, B phrases, ALU clauses, and memory reference clauses are discussed in separate sections and only outlined here.

The ALU operation in the first example is a function of both H1 and H2; it involves only H1 in the second, and only H2 in the third. In this manual, the H1 and H2 data paths are referred to as "A" and "B," respectively, and the output of the ALU, loaded into H3P, is referred to as "LU."

Rule: In an ALU operation involving both A and B, the A phrase must appear to the left of the B phrase.

Rule: An F1 or F2 function that either sources or sinks A will appear in the A phrase; one that either sources or sinks B will appear in the B phrase; one that involves neither A nor B will appear as a separate clause in the instruction.

Data-routing clauses have one or more "_"’s in them and require parentheses in some places to cause evaluation in the correct order. One of these clauses is evaluated from right-to-left, or from "sources" to "sinks."

If there is only one source and one sink in the clause, no problem: simply write "sink_source", e.g.:

T_RMAddr,	The assembler figures out how to route data from the RM address RMAddr onto A, through the ALU, and into T.
RMAddr_34C,	Again the assembler figures out how to construct the constant 34, route it onto B, through the ALU, and into the RM address RMAddr.

When you have A or B phrases embedded in ALU expressions, then you have to use parentheses, e.g.:

T_(StkP_RMAddr)+1,	The assembler routes RMAddr onto A, loads StkP from A with an F2 function, selects the A+1 ALU operation, and sets the LT bit in the instruction so that T will be loaded from the ALU.
T_(LoadTimer[RMAddr])+(SALUF_T)	The assembler routes RMAddr onto A, selects the LoadTimer function in F1, routes T onto B, selects the SALUF_ function in F2, selects the "A+B" ALU operation, and sets the LT bit to load T from the ALU output.

In assembling the first clause above, the assembler proceeds in the following way:

a. RMAddr is looked up first and recognized as an RM address. Error checks ensure that RMAddr can legitimately be referenced by the current task. If so, the proper value is assembled for the RSel, RMod, RX, and MemIns fields of the instruction. The assembler's macros leave a neutral symbol "RB" at the end of this evaluation to check for routing errors later.

b. "StkP_" is looked up next; this macro expands to store the correct code in the F2 field of the instruction and leaves a neutral symbol "A_" to check for routing errors.

c. "A_RB" is looked up; because routing RB onto A is legal, this "connection macro" is defined and expands to leave the neutral "A." (If the routing had been illegal, then Micro would have printed an error message like "A_RB undefined.")

d. "A+1" is looked up; this macro expands to store the correct code in the ALUF instruction field and leaves behind the neutral symbol "LU" to check for routing errors.

e. Then "T_" is looked up; it is a macro that expands to store a 1 into the LT field of the instruction, leaving the neutral "LU_."

f. Finally, "LU_LU" is looked up; it is a "connection" macro that does nothing and leaves behind the neutral "LU"; since there is no more text in the clause this final neutral is thrown away.

Note: The "()" in the first example above are not optional. If you omit them, the assembler would look up "RMAddr+1", which would be undefined.

One general idea in the above is that at each stage the source is routed only as far as necessary to load it into the destination.

Note:	T_StkP_RMAddr,	is legal
	T_StkP_(RMAddr),	is legal
	T_(StkP_RMAddr),	is legal
	StkP_T_RMAddr,	is illegal

The last clause above is illegal because, by the time the assembler recognizes StkP_, it has already routed the source data past A and through the ALU, and there is no path from the ALU to StkP. The assembler is not clever enough to remember that the data originally started on A.

Here are some more "()" examples:

T_(RMAddr)+T,	is legal--"()" are required around RM addresses in combination with other stuff.
T_RMAddr,	is legal--"()" optional around an RM address all by itself
T_(RMAddr)+(T),	is legal--"()" optional around "T."
T_((RMAddr)+T),	is legal--extra "()" around an entire source always OK
(T_(RMAddr)+T),	is legal
T_T+(RMAddr),	is illegal--you cannot reverse the A and B phrases.
RMAddr_(RMAddr)+(2C)	is legal--"()" required around constants in combination with other stuff.

The last two examples at the beginning of this section show the form of "memory reference" instructions. In the first example, the F2 field is used instead of T to specify the displacement for the reference; in the second example, T is used as the displacement so F2 is available to specify some function. Memory reference clauses are discussed later.

19. RM and STK Phrases

The hardware complicates RM references by providing only six bits of RM address in the instruction. The remaining two address bits come from the task number. The programmer must declare the task number with SetTask before any references.

RM addresses can source A and can be used in ALU phrases. In this case, the RM address has to be enclosed in "()."

RM addresses can be used as sinks for ALU operations and ALU sources (which the assembler routes through the ALU). For these simply write the register name followed by "_".

Some examples of clauses involving RM are as follows:

RMAddr_(RMAddr)+T	Use in an ALU expression and as a sink.
RMAddr_T	The assembler automatically routes T through the ALU and into RMAddr.
PCF[RMAddr]_PCF[RMAddr]	You really mean RMAddr_PCF[RMAddr], but due to the vagaries of the assembler you write it this way.
T_(PCF[RMAddr])+T	Use with cycler-masker operation PCF.
T_PCF[RMAddr]+T	"()" not mandatory with these.
T_RMAddr	"()" optional when RMAddr not combined with other text.
DB_RMAddr	Use with an A sink.
T_(SB_RMAddr)+T	Use with an A sink inside an ALU operation.

The assembler will detect all illegal RM references. In other words, if the RM word is unaddressable by the currently selected task, or if it is required to be even or quadaligned, the assembler will check for this and indicate any error.

References to the stack always read the word pointed to by StkP, then adjust the stack pointer in the ways discussed in the "A Phrases" section; if the stack is written, the modified address is used for the write. The various Stack terms may be used interchangeably with RM addresses where they are legal. Also note the following:

Stack&+1_(Stack&+1)+T	is legal.
Stack&+1_(Stack)+T	is illegal.

The last example is illegal because you must use the same StkP modifier in both read and write parts of the instruction.

Occasionally you may wish to create a constant whose value is an RM address. To do this you can use the following kludge:

```
B_IP[RMAddr]C
```

This puts a constant whose value is the address of RMAddr onto B.

20. Cycler-Masker Phrases

The semantic cycler-masker operations, operating on RM data, are as follows:

LSh[RA, n]	Left shift the RM address RA n positions (n = 1 to 17).
RSh[RA, n]	Right shift RA n positions (n = 1 to 17).
LdF[RA, pos, size]	Right-justify or load an arbitrary field from RA; POS is the left bit of the field and SIZE is the number of bits in the field.
RCy[RA, n]	Right cycle RA n positions (n = 1 to 17).
LCy[RA, n]	Left cycle RA n positions (n = 1 to 17).
Dispatch[RA, pos, size]	Loads APC with any field of size less than 4 bits from RA; the following instruction must contain a Disp control clause to carry out the dispatch to the location at Page[0:3],,JA[0:3],,APC[14:17].
RHMask[RA]	= RA & 377.
LHMask[RA]	= RA & 177400.
Zero	= 0.
FixVA[RA]	= RSh[RA,1] & 40100.
Formn[RA]	= RA & n (n = 0 to 10).
Form-n[RA]	= RA & -n (n = 2 to 10).
Nib0Rsh8[RA]	= RSh[RA,10] & 360, which is the first 4-bit nibble in RA right-shifted 10.

For these operations, the assembler fabricates values for F1, F2, and BSel to cause the appropriate shift and mask. RA may be any normal RM address or one of the special ones that have RMod=1 and don’t use F1 or F2 in their specification.

In addition to the cycler-masker phrases discussed here, the various forms of the NextData and NextInst functions discussed later use the cycler-masker to extract either the left or right byte from the selected RM word.

21. A Phrases

All of the A sources use the RMod and RSel instruction fields; some also use the F1 and F2 fields in conjunction with these. An A source can be any of the following:

A	Dummy source for clause splitting. It indicates that the source for some sink is ALUA (You should never need to use this because normally the ALUA sink and source are written together separated by "_" or the ALUA source is embedded in the ALU phrase.).
RAddr	An RM address.
Stack	RM addressed by StkP
Stack&-n	RM addressed by StkP; StkP_StkP-n (n = 1, 2, or 3) after the read (and before the write, if any).
Stack&+n	RM addressed by StkP; StkP_StkP+n (n = 1, 2, or 3) after the read (and before the write, if any).
PCF[RAddr]	RAddr must be quadaligned--i.e., its low two address bits must be 0.
SB[RAddr]	RAdr must be quadaligned.
DB[RAddr]	RAdr must be quadaligned.
NextInst[RAddr]	Instruction dispatch; incorporates both the NextInst F1 function and the PCF[RAddr] cycler masker operation, so RAddr must be quadaligned. See the "NextData and NextInst" section.
CNextInst[RAddr]	Like NextInst; see the "NextData and NextInst" section.
NextData[RAddr]	Incorporates both the NextData F1 function and the PCF[RAddr] cycler masker operation, so RAddr must be quadaligned; see the "NextData and NextInst" section.
CNextData[RAddr]	Like NextData; see the "NextData and NextInst" section.
BBFA[RAddr]	Also a dispatch (no special restrictions on RAddr); uses F1 and F2.
BBFB[RAddr]	Uses F1.
WFA[RAddr]	Uses F1.
WFB[RAddr]	Uses F1.
RF[RAddr]	Uses F1.
BBFX[RAddr]	Uses F1.

The A sources below use RMod and RSel to specify external R-bus sources and use the cycler-masker to extract individual registers from multi-field sources.

GetRSpec[n]	Used to place a multi-field word on A. n is a 7-bit value whose leading bit is placed in RMOD and next 6 bits in RSEL. You should normally prefer to use one of the explicit macros given below instead of GetRSpec[n].
SStkP&NStkP	= GetRSpec[103]
SStkP	Saved StkP; uses F1 and F2.
NStkP	StkP (ones comlemented); uses F1 and F2.
ALUResult&NSALUF	
ALUResult	Uses F1 and F2.
NSALUF	Uses F1 and F2.
MemSyndrome	
MemError	
Cycle&PCXF	= GetRSpec[127]
CycleControl	Uses F1 and F2 (= DBXReg..MWXReg)
DBXReg	Uses F1 and F2.
MWXReg	Uses F1 and F2.
PCXReg	Uses F1 and F2
PCFReg	Uses F1 and F2.
Printer	Uses F2.
Timer	= GetRSpec[133]

DBSB	Uses F2.
RS232	
MNBR	Uses F2.
APCTask&APC	= GetRSpec[143]
APCTask	Uses F1 and F2.
APC	Uses F1 and F2.
APC&APCTask	= GetRSpec[143] (synonym)
CTask&NCIA	
CTask	Uses F1 and F2.
NCIA	Uses F1 and F2 (= CIA ones complemented).
CSData	
Page&Par&Boot	= GetRSpec[157]
Page	Uses F1 and F2.
Parity	Uses F1 and F2.
BootReason	Uses F1 and F2.

Rule: Those of the above sources that accept a normal RM address as an argument, namely PCF[RA], SB[RA], DB[RA], WFA[RA], BBFB[RA], WFB[RA], RF[RA], and BBFBX[RA], require that this same form be used when writing RA in the same instruction; i.e.:

```
PCF[RMAddr]_(PCF[RMAddr])+T          is legal
RMAddr_(PCF[RMaddr])+T               is illegal
```

This unfortunate kludge is required because PCF[RMAddr] will set RMOD to 1, while RMAddr_ will set RMOD to 0, causing an assembly error.

Sinks for A are the ALU and a number of other registers selected by functions. A sinks can be any of the following:

any ALU expression or ALU sink (T_, RAddr_, or LU_)

A_	No-op sink--simply routes the selected A source onto ALUA.
APCTask&APC_	Uses F1 and F2
Restore_	Uses F1 and F2
StkP_	Uses F2
CycleControl_	Uses F2
SB_	Uses F2
DB_	Uses F2
MNBR_	Uses F1 and F2
PCF_	Uses F1 and F2
Printer_	Uses F2

21. B Phrases

B sources can be any of the following:

B	Dummy source for clause splitting
Constants	Uses FF--see earlier constant section
T	

B sinks can be any of the following:

B_	No-op destination--simply routes source onto B.
RS232_	Uses F1
SALUF_	Uses both F1 and F2.
CTD_	Uses both F1 and F2.
any ALU expression or ALU sink	

23. ALU Clauses

In the ALU operations given below, the "A" and "B" components may be any A and B phrases, respectively:

B	
A	
A and B	
A or B	
A xor B	A#B
A and not B	
A or not B	
A xnor B	A=B
A+1	
A+B	
A+B+1	
A-1	
A-B	
A-B-1	
A SALUFOP B	

Rule: For all ALU operations except the "A" and "B" operations, the A phrase must always be enclosed in "()"; the B phrase must be enclosed in parentheses unless it is T; "()" are optional around T.

You must not, for example, write "A and not B" as "A and (not B)"; in other words, it is illegal to put random "()" in the name of the ALU expression, even though that may clarify the meaning. If you tried to do this, the assembler would fail to recognize "not B" and flag an error. The "()" are only legal around the "A" and "B" parts of the ALU phrase.

The arithmetic operations, "A-1," "A+1," "A+B," "A+B+1," "A-B," and "A-B-1" may optionally be accompanied by the standalone clause "UseCOutAsCIn," which causes the ALU carry-in to be the carry-out of the last ALU operation for the same task. The value of the carry-out may be frozen across instructions by means of the FreezeResult standalone function.

"UseCOutAsCIn" cannot be used with the "A" operation because the logical rather than the arithmetic form of the "A" operation is used by the hardware.

Legal sinks for ALU phrases are:

RAddr_	Any RM address

Stack_	Stack sinks
Stack&+n_	
Stack&-n_	
T_	
LU_	This null sink is necessary when the ALU operation must be "A" or "B" for a subsequent branch condition or to interlock/not interlock an RM reference, even though no real destination is being loaded with the ALU output.

Here are some examples of ALU clauses:

T_(RAddr)+T	"()" optional around T, mandatory around RAddr.
T_RAddr	"()" optional around RAddr uncombined.
T_(RAddr)	also ok.
RAddr_(RAddr) xor (377C)	"()" mandatory.
RAddr_T_377C	"()" optional when constant uncombined.
T_RAddr_(377C)	also ok.

24. Memory Reference Instructions

Memory reference instructions have a different form from regular instructions, as discussed in the hardware manual. Branch and placement clauses are identical to those in regular instructions, and the F2 clause, if any, is identical to that in a regular instruction. The rest of the instruction is a single clause in one of the following forms:

PFetch1[RBase, RAddr<, F2>]	OddPFetch1[RBase, RAddr<, F2>]
PFetch2[RBase, RAddr<, F2>]	OddPFetch2[RBase, RAddr<, F2>]
PFetch4[RBase, RAddr<, F2>]	OddPFetch4[RBase, RAddr<, F2>]
PStore1[RBase, RAddr<, F2>]	OddPStore1[RBase, RAddr<, F2>]
PStore2[RBase, RAddr<, F2>]	OddPStore1[RBase, RAddr<, F2>]
PStore4[RBase, RAddr<, F2>]	OddPStore1[RBase, RAddr<, F2>]
IOFetch4[RBase, Device<, F2>]	OddIOFetch4[RBase, RAddr<, F2>]
IOFetch20[RBase, Device<, F2>]	OddIOFetch20[RBase, RAddr<, F2>]
IOStore4[RBase, Device<, F2>]	OddIOStore4[RBase, RAddr<, F2>]
IOStore20[RBase, Device<, F2>]	OddIOStore20[RBase, RAddr<, F2>]
XMap[RBase, RAddr<, F2>]	OddXMap[RBase, RAddr<, F2>]
Input[RAddr<, F2><,OtherRAddr>]	
Output[RAddr<, F2><,OtherRAddr>]	
ReadPipe[RAddr<,F2><,OtherRAddr>]	
Refresh[RAddr]	

IOStore16, OddIOStore16, IOFetch16, and OddIOFetch16 are synonyms for IOStore20, OddIOStore20, IOFetch20, and OddIOFetch20, respectively.

In these clauses, "RBase" is an RM address subject to the same constraints as an RM reference in a regular instruction; i.e., it must be in the group of 100 accessible to the current task and must not be in the subrange 0-17 of that group unless the task selected by SetTask is 0 mod 4; in addition, it must normally be an even address, as discussed in the hardware manual. When it must be even, you write, for example, "PFetch1[...]"; in the rare case when it must be odd, you write "OddPFetch1[...]." Also, the various forms of RM addressing that require RMod=1 cannot be used when specifying a base register (i.e., Stack, Stack&+n, Stack&-n, PCF[..], SB[..], and DB[..]

won’t work); this is illegal because the DF2 bit in memory reference instructions coincides with RMod in regular instructions.

When the optional F2 argument is omitted, the displacement relative to the base register is taken from T, and you may use F2 for an unrelated function in a separate clause. If you supply the F2 argument, which must be an integer less than 20, that value is stored in the F2 field of the instruction and used instead of T. See the hardware manual for details on how this works.

Note that memory instructions use the ALU to add the low-half base register to the displacement; the result of this addition may be tested in the next instruction if desired. For example, it is sometimes useful to branch on the ALU carry in the next instruction, so that the high part of the base register can be updated when crossing a 64K memory boundary.

Input, Output, and ReadPipe references do not use a base register, but address computation is done just as for other references. "OtherRAddr" (default 0 if omitted) specifies the base register, which can be tested by an R<0 or R Odd branch condition in the same instruction, or used to somehow setup an ALU result for a branch condition in the next instruction. In addition, if the base register is the target of an immediately preceding PFetch, then the Input or Output will be held until that reference has finished; specifying a different base register avoids this unnecessary hold--or you can deliberately select the target of the preceding PFetch to interlock the reference.

The RAddr argument may be either "Stack" or an RM address accessible to the task. Since the current task is OR’ed into the top 4 bits of the 8-bit SrcDest field that specifies RAddr, RAddr is further constrained to satisfy the relation: (RAddr or (CTask lshift 4)) = RAddr. In addition, RAddr must not be 0 (which is the value assembled for "Stack"); for PFetch2/PStore2 it must be even and for PFetch4/PStore4 it must be quadaligned.

> "Must be even" and "must be quadaligned" are a little too strong. If you violate this rule, the assembler will print a warning message which you are free to ignore. In this case, the hardware will transfer the double-word/quad-word into the two/four RM locations beginning with the one you specify and wrapping around at 20-word boundaries (i.e., if you specify RM 36 as the address for a PFetch4, the data will be delivered to 36, 37, 20 and 21). The reason for the warning is that RM interlocking will not occur correctly, as discussed in the next section.

25. RM Interlocking

The hardware "interlocks" RM references to ensure that following a memory reference you neither read an RM word before the memory controller has filled it from storage nor overwrite an RM word before the memory controller has deposited it in storage; an instruction will be repeatedly aborted and reexecuted until it is safe to proceed. However, the hardware interlock is not foolproof, so there are some programming requirements of which you must be aware.

First, every instruction is reading some RM word (word 0 in the current 100-word region, if you don’t explicitly mention an RM address), but the hardware interlocks reads only when the ALU operation involves A somehow--no interlock occurs on an LU_T ALU operation, which does not involve A. However, sometimes you will utilize an RM word without routing it into the ALU. For example:

StkP_RAddr, RAddr_T;	Load an A destination from RAddr, LU_T
RAddr, Skip[R Odd];	Use the R<0 or R Odd branch conditions

In these situations, if RAddr is involved in a preceding fetch-type memory reference (PFetch1/2/4, ReadPipe, XMap, or Input) which is still in MC1/MC2, the hardware interlock will not occur and the instruction will erroneously proceed without waiting for the memory controller to fill the word from storage.

The assembler has some features to help avoid programming errors in this situation. First, the default ALU operation in an instruction is LU_T, but when you write a clause which loads some A destination from RM, the default is changed to LU_A which will invoke the read interlock. In other words:

StkP_RAddr;	= LU_StkP_RAddr, so the interlock will occur
A_RAddr;	= LU_A_RAddr, so the interlock will occur
RAddr, Skip[R Odd];	didn't say A_RAddr, so LU_T is still the default and interlock will NOT occur--potential undetected programming error here.
StkP_RAddr, RAddr_T;	LU_A default overruled by LU_T, so no interlock will occur, but the assembler will issue a "No register interlock" warning to you in this case; you can ignore the warning if you want to, but...
StkP_RAddr, RAddr_T, NoRegILockOK;	suppresses the "no register interlock" warning message; you should add the "NoRegILockOK" clause to instructions which do not invoke the register interlock if you are sure that not having the register interlock is ok.

Another set of potential problems occurs when the two RM words involved in PFetch2/PStore2 are not an even-odd pair or when the four RM words affected by a PFetch4/PStore4 are not quadaligned (i.e., the first of the four words does not have two low zeroes in its address). For example, suppose a PFetch2/PStore2 is made to the Stack; if the first of the two RM words is even, all is well and the hardware interlock will abort references to either of the two words involved in the reference; however, if the first of the two RM words is odd, then the hardware will NOT abort references to the second word involved in the reference (It will instead erroneously abort references to the RM word preceding the first word). Similar problems occur when the first RM word of a PFetch4/PStore4 is not quadaligned.

To assembler will flag potential problems in these situations with a warning message "WARNING: RAddr not even" on a PFetch2/PStore2 or "WARNING: RAddr not quadaligned" on a PFetch4/PStore4 for which the RM address is not aligned. These warning will not occur on PFetch2/PStore2 to the stack. If an unaligned reference is legitimate, then you must advise the assembler by placing an "OddOK" or "NonQuadOK" clause to the right of the reference.

When you reference a stack word that is potentially the second word of an unaligned and incomplete PFetch2, you have to be very careful to use the Stack&-n form of reference; when you write any stack word that is potentially the second word of an unaligned and incomplete PStore2, you must use the Stack&+n form of reference. Also note that when you interlock by using Stack&-n or Stack&+n, you do not have to route the stack through the ALU to accomplish the interlock. These are discussed in the hardware manual.

26. Standalone Functions

The following summarizes standalone functions, each written as a separate clause in the instruction:

BreakPoint	Causes the instruction to be breakpointed by Midas when debugging.
Nop	Assembles a no-op instruction.
SpareFunction	Uses F1 and F2 (undefined)
ResetErrors	Uses F1 and F2
IncMPanel	Uses F1 and F2
ClearMPanel	Uses F1 and F2
GenSRClock	Uses F1 and F2
ResetWDT	Uses F1 and F2
Boot	Uses F1 and F2
SetFault	Uses F1 and F2
ResetFault	Uses F1 and F2
UseCTask	Uses F1 and F2
WriteCS0&2	Uses F1 and F2
WriteCS1	Uses F1 and F2
ReadCS	Uses F1 and F2
D0Off	Uses F1 and F2
RegShift	Uses F2 (ordinarily not written explicitly becuase the sources that require it are individually named, so you probably will never use this)
FreezeResult	Uses F2
IOStrobe	Uses F2
ResetMemErrs	Uses F2
UseCOutAsCIn	Uses F2
SkipData	See the "NextData and NextInst" section.
CSkipData	See the "NextData and NextInst" section.
LoadPage[n]	Uses F1 and F2; 0 <= n <= 17, where n is an integer equal to the page number; this is used when an ordinary branch clause is used in the next instruction.
LoadPageExternal[n]	Like LoadPage; this must be used when the branch clause in the next instruction is a GotoExternal or CallExternal.
At[n1,n2]	See the "Placement Declarations" section.
Opcode[n]	See the "Placement Declarations" section.

27. Branching

This section discusses branch and dispatch clauses, which are assembled into the JC and JA fields of the instruction. At assembly time, D0Lang produces the correct JC field and sometimes the correct JA field or correct low bit of JA; however, most branch clauses indicate the required branch linkages in extra fields assembled only for MicroD. MicroD then determines a satisfactory placement for the instructions and fills out JA correctly.

The hardware defines GoTo, Call, Disp, Return, and conditional GoTo, each represented by codes in the JC instruction field; conditional GoTo's may specify any of eight branch conditions using only JC and JA or 6 more branch conditions using F2. The least significant bit of JA is used to modify the branch condition on a conditional GoTo and to modify the action of a Return.

The assembly language encodes these possibilities in the following kinds of branch clauses:

DblGoTo[t1, t2, bc]	Jumps to t1 if the branch condition bc is true, else to t2; MicroD will place t2 at an even location and t1 at the adjacent odd location; the targets must be on the same 400-word microstore page as the instruction containing the DblGoto.
GoTo[t1<, bc>]	Jumps to t1 if the optional bc is either omitted or true, else falls through to the next instruction inline. t1 will be placed on the same page as the current instruction. If bc is given, the next instrucion inline will be placed at an even location and t1 at the adjacent odd location.
Skip[<bc>]	= GoTo[.+2<,bc>]
Call[t1]	Loads TPC[current task] with the address of the next instruction inline and branches to t1; MicroD will place the next instruction inline at (. & 7760)+(.+1 & 17) and will place t1 on the same page.
CallX[t1]	Loads TPC[current task with (. & 7760)+(.+1 & 17) and branches to t1. The next instruction inline is not constrained to lie at any particular location; MicroD will place t1 on the same page.
Disp[t1]	Will dispatch to the the table origined at (t1 & 7760); the programmer must manually place all instructions in the dispatch table with At clauses. The dispatch will transfer control to (t1 & 7760) OR'ed with the low four bits of APC. The instruction immediately preceding Disp must normally be a Dispatch (but conceivably UseCTask or APCTask&APC_ might be used instead of Dispatch in some clever situations).
Return	Returns to the address in APC (JA.7=0).
NIRet	Returns to the address in APC (JA.7=1) which will contain 2001+(4*ByteCode); this form of return is used after NextInst.
Task	Special kludge that does Call[.+1] in the current instruction and forces a Return in the next instruction inline (which must not have any branch clause); the second instruction inline will be placed at .+1.
GoToExternal[n]	Puts a goto code in JC and stores the integer n in JA; no placement constraints imposed and no error checking; usually used after LoadPageExternal[m].
CallExternal[n]	Puts a call code in JC and stores the integer n in JA; no placement constraints imposed and no error checking; usually used after LoadPageExternal[m].
DblGoToP[t1, t2, bc]	= DblGoto with the on-page constraints removed; the preceding instruction must contain an apropriate LoadPage; no error checking.
GoToP[t1, bc]	= Goto with the on-page constraint removed.
CallP[t1]	= Call with the on-page constraint removed.
DispP[t1]	= Disp with the on-page constraint removed.
SkipP[t1]	= Skip with the on-page constraint removed.

In the above forms, t1 and t2 may be instruction labels or one of the following special symbols:

.+3	current address + 3.
.+2	current address + 2.
.+1	current address + 1.
.	the address of the current instruction.
.-1	current address - 1.
.-2	current address - 2.
.-3	current address - 3.

Branch conditions may be any of the following:

Regular	*Complementary*	
ALU#0	ALU=0	
Carry	Carry’	
ALU<0	ALU>=0	
H2Bit8’	H2Bit8	
R<0	R>=0	
R Odd	R Even	
IOAtten’	IOAtten	
MB	MB’	
IntPending	IntPending’	Uses F2
Ovf’	Ovf	Uses F2
BPCChk	BPCChk’	Uses F2
QuadOvf	QuadOvf’	Uses F2
TimeOut	TimeOut’	Uses F2

The complementary form of the branch conditions is provided as a programming convenience. The effect of using a complementary branch condition is to interchange the even-odd constraints on the target pair.

28. NextData and NextInst

The assembler contains a number of macros and placement features to support the NextData and NextInst hardware functions. These functions trap to absolute location 0 in the microstore when the quadword buffer containing data from the opcode stream is exhausted; on a trap, the instruction address in CIA will be saved in TPC iff the instruction containing NextData/NextInst has a Call encoded in its JC field.

> The system microcode deals with this situation as follows: First, the trap instruction at 0 contains a "LoadPageExternal[0], GoToExternal[377]" where the GoTo sends control to location 377 on the page containing the NextData/NextInst that trapped. Next, the instruction at 377 on that page will contain a "PFetch4[PC,IBuf,4], GoToP[BufferRefill]" or whatever, where BufferRefill is on page 0. There are restrictions on what can be done by the trap instruction at location 0 (see hardware manual).

For this reason it is usually necessary to encode a Call in JC to ensure that the trap subroutine will restart at the correct place. However, this will usually be a "fake call" (equivalent to CallX) because there is no intention to return from a subroutine to .+1 in the instruction stream. Consequently, the assembler has a bit in its output data format (the OddCall bit) which tells MicroD that, even though the current instruction contains a call, it is not necessary to place the next instruction inline at .+1 in the microstore. Some of the macros discussed below set this bit to avoid imposing an unnecessary placement constraint on the program.

The macros defined for these two functions are as follows:

A_NextInst[RAddr]	Instruction dispatch on the next opcode in the opcode stream; PCF[RAddr] addressing selects the word in the quadword-buffer beginning at RAddr containing the next byte; the NextInst F1 function causes the cycler-masker to select the byte in that word according to the low bit of PCF; forces a fake Call to be encoded in JC. No branch clause is necessary if the successor is the next instruction inline, but if any branch clause is specified, it must be a (fake) Call or CallX.
A_CNextInst[RAddr]	Like NextInst, but the JC field is not constrained and any branch clause may be used. However, unless either a buffer refill trap is impossible (because the BPCChk branch condition is false or because PCF is known to be odd) or the return address in TPC has already been setup to cope with a trap, you will have to write an explicit Call into the instruction to ensure that a buffer refill trap will return to reexecute this instruction; this will be a real call.
A_NextData[RAddr]	Like NextInst but uses the NextData function rather than the NextInst function to obtain the next byte in the opcode stream.
A_CNextData[RAddr]	Like CNextInst but uses the NextData function.
SkipData	Uses the NextData function to skip the next byte in the opcode stream without referencing it; forces a fake Call to be encoded in JC; does not impose PCF addressing, so any RM address can be referenced (but because the NextData function will use the cycler-masker to select either the left or right byte of the selected RM address, your use of that RM address is limited).
CSkipData	Like SkipData, but does not impose any constraint on JC; any branch clause may be used. If the branch clause is a Call, it will be "real" and the successor will be the next instruction inline; if not a Call, you must be sure that either a trap is impossible or the return address already in TPC is an appropriate return from the trap subroutine.

Here are some examples using these macros:

```
        LU _ NextInst[IBuf];                     *Successor to NextInst is .+1
Push:   Stack&+1 _ T, NIRet;

        LU _ NextInst[IBuf], Call[Push];         *Successor to NextInst is at the label "Push"

        T _ CNextData[IBuf], Call[FixPointer];   *Real call with NextData function
        T _ (Stack&-1) + T, GoTo[Push];

        Call[FixT];                              *Call unnessary because return for possible trap
        T _ CNextData[IBuf], Skip[IntPending];   *already setup

        LU _ Cycle&PCXF, FreezeResult, Skip[R Even];
          CSkipData, DblGoTo[A,B,ALU<0];         *Trap impossible because PCF is odd
        DblGoTo[A,B,ALU<0];
```

Note: It is not strictly necessary for NIRet to occur in the instruction immediately after the NextInst/CNextInst. From the hardware manual, it appears that a subroutine which returned by doing a NIRet could intervene. In other words, the NextInst could be followed by an arbitrary code sequence; when NIRet occurred, the dispatch address for the next opcode would be put into TPC; and the Return or NIRet after that would send control to the next opcode.

29. Placement Declarations

Currently, you must specify the page on which each instruction is placed. This is done by a declaration of the form:

```
OnPage[n];
```

where the integer n is the page number (0 to 17). Subsequently assembled instructions will be placed on the specified page. The OnPage declaration may appear only as a separate statement.

Two other declarations may appear only as clauses in an instruction being assembled; these specify the exact placement of the instruction and also do OnPage, so that subsequently assembled instructions will be placed on the same page. These are:

At[n1]	places the current instruction at location n1.
At[n1, n2]	places the current instruction at location n1+n2.
Opcode[n]	places the current instruction at 2001+(4*n), where n is the opcode (0 to 377)

DispTable[Length,Mask,Value] appearing in an instruction statement causes that instruction to begin a group of Length consecutively-placed statements, where 1 <= Length <= 20. The first instruction will be placed at a location such that And[location,Mask] = Value, where Mask defaults to 17 and Value to 0. It is required that Length+Value be less than 21. Placement of the instruction containing DispTable will be further constrained by whatever OnPage declaration is in force.

Note that DispTable will not handle all dispatch tables; it requires that a table be no longer than 20 with no "holes" and that the instructions in the table appear as consecutive statements in the source program. Whenever these conditions cannot be met, At clauses must be used for the instructions in the table.

Opcode declarations are required for the entry instructions of opcodes; At declarations are required for trap instructions, for instructions in dispatch tables that cannot use DispTable for some reason, and in a few other situations, but MicroD will take care of normal placement constraints automatically.

To assist in defining absolute locations for dispatch tables and manually placed instructions the "LOCA" macro is defined:

Loca[Name,Page,Offset]	defines Name as an integer equal to Page*400 + Offset, where Page and Offset are integers.

In addition to the above placement declarations, there are two macros for "reserving" or "unreserving" locations in IM. These macros, which direct MicroD to avoid placing instructions in particular microstore locations, are as follows:

IMReserve[p,w,n]	where p, w, and n are integers reserves n locations beginning at word w on page p.

IMUnReserve[p,w,n]	unreserves n locations beginning at word w on page p.

For diagnostics, the declaration statement "MidasInit;" may be issued to reserve the several locations on page 0 and all of page 17 for use by the debugger Midas.

30. Tasking Considerations

Since task switching occurs only when an "unrestricted" return is executed, each task must be sure to execute instructions containing returns often enough to satisfy the timing requirements of tasks which are higher (or sometimes lower) in priority. Here "unrestricted" means that the instruction preceding return did not do APCTask&APC_ or UseCTask, which disallow task switching.

When an unrestricted return is executed, the highest priority *different* task requesting service will run, or the emulator if no other tasks are requesting. This has several implications:

First, when a high priority task such as the display driver executes a return when it has not yet blocked, it will lose control to some other task (usually the emulator) until that task does a return. This means that it should not do too many returns else the accumulated interference of lower priority tasks will add up to so large a value that its timing requirements are not satisfied.

Secondly, each task must be sure to execute Returns frequently enough that higher priority tasks receive adequate service.

We can illustrate this with an example. Suppose that the timing requirement of the display driver is that upon issuing a wakeup request it be granted 88 cycles out of the next 288 cycles and that it executes one intermediate return before its final return after blocking. Initial service will be delayed until whatever task is running executes a return, and further delayed until all higher priority tasks are satisfied. At the intermediate return, a lower (or higher task) will run first and then higher tasks until they are satisfied. If page or write protect faults or single-error logging are possible, then some allowance must be made for the fault task, which has a long burst of non-tasking computation at the beginning of each fault. The total interference of all these other tasks must be less than 200 cycles.

The emulator task for the current AMesa system microcode attempts to satisfy this requirement by tasking at least once every 46 cycles.

Another programming consideration is the exact placement of returns. It is frequently possible to choose one of several places; one good place is immediately before an instruction that will be aborted--e.g., before an instruction that will read an RM word for which a PFetch has just been started or write an RM word for which a PStore has just been started. An undesirable placement is where the current task could proceed because it is not issuing memory references but where another task might experience an abort because MC1 is busy.

31. Microcode Overlays

The barest minimum provisions are made for microcode overlays. When assembling basic system microcode, the IMReserve macro discussed in the "Placement Declarations" section may be used to disallow the use of any set of IM locations by MicroD. Also, any pages totally unused by the system microcode or containing throwaway initialization code are available to overlays. Finally, particular instructions in the system overwritten by particular instructions in an overlay must be manually placed with "At" declarations in both the system microcode and the overlay.

An overlay must use IMReserve to prevent its use of any locations on pages that it shares with system microcode. MicroD has a switch that will write a xxOccupied.Mc file containing IMReserve statements.

Any instructions in the system branched to from an overlay must be manually placed with "At" and vice versa. Then, instead of using LoadPage[n] and GoTo[n] or Call[n], you must use LoadPageExternal[n] and GoToExternal[n] or CallExternal[n] in instructions that branch between the system and an overlay.

32. Recent Hardware and Assembler Changes

1. SUB, EQUATE, ASMMODE, TRACEMODE, and WHILE Micro builtins and the "warning" code for the ER builtin.

2. Some names have been assigned to R-bus sources that were formerly referred to by GetRSpec[n] phrases (e.g., Cycle&PCXF, SStkP&NStkP, CTask&NCIA, Page&Par&Boot, ALUResult&NSALUF).

3. Bugs in use of GoTo with read/write control store were fixed.

4. Some error messages for PFetch2/PStore2 and PFetch4/PStore4 were changed to warnings, and the OddOK and NonQuadOK clauses to the right were added to overrule the warnings.

5. The specification of a second RM address with Input[..], Output[..], and ReadPipe[..].

6. OddBaseOK was removed and OddPFetch1, etc. were recently added.

7. SkipData, CSkipData, CNextInst, Loca, IMData, HiA, LoA, RV2, RV4, CallX, negative constants, Formn, Form-n, and Nib0RSh8, and IF, UNLESS, ELSEIF, ELSE, ENDIF conditional assembly stuff are recent assembler additions.

8. The IMMASK memory handled by MicroD and the DispTable macro which outputs placement constraints for dispatch tables are new.

9. Additional MicroD output files.