DORADO

MICROASSEMBLER

21 July 1980

by

Edward Fiala

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

Filed on: [Ivy]<DoradoDocs>DoradoMicroassembler.Press
Sources on: [Ivy]<DoradoSource>DoradoMicroassembler.Dm

This manual describes the Dorado microassembly language, based upon the 8 October 1979 release of the Dorado Hardware Manual, and hardware changes up to the release date of this manual.

TABLE OF CONTENTS

1. Preliminaries

The Dorado 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 some other machine (e.g., D0 or Dorado) emulating an Alto. The assembly language is based upon the machine description in the 8 October 1979 release of Dorado Hardware Manual and several hardware changes that have occurred since then.

Files referred to in this manual are as follows:

Documentation	*When Using the Assembler*
[Ivy]<DoradoDocs>	[Ivy]<DoradoSource>
DoradoManual-A.Press	D1Lang.Mc
DoradoManual-B.Press	D1Alu.Mc
DoradoManual-Figs.Press	[Maxc1]<Alto>
DoradoMidasManual.Press	Micro.Run
[Maxc1]<AltoDocs>	MicroD.Run
Micro.Press	

The assembly language is defined by D1Lang.Mc and D1Alu.Mc; you must modify D1Alu.Mc as discussed later. I have tried to make D1Lang.Mc and this documentation complete, so you should not need to refer to the Micro manual or study D1Lang 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 and begin programming with Figure 1 of the hardware manual in front of you for reference.

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

2. Assembly Procedures

DlAlu.Mc must be edited to define the ALU definitions for your program as discussed in the "Assembling For ALUFM" section. Suppose that you put these definitions on AluDefs.Mc. Then a microassembly is accomplished as follows:

```
Micro/L DlLang AluDefs Source1 Source2 ... SourceN
```

This causes the source files "DlLang.Mc", "AluDefs.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 DlLang AluDefs Source1 ... SourceN
```

causes the four output files to be "Sys.Ls", "Sys.St", "Sys.Dib", and "Sys.Er."

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 9 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 55 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 AluDefs.St file as follows:

```
Micro/U DlLang AluDefs
```

Then do all further assemblies as follows:

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

Preassembling DlLang and AluDefs in this way saves about 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 DlLang AluDefs RegDefs
Micro/O/U RegDefs/R Source1
Micro/O/U RegDefs/R Source2
   ...
Micro/O/U RegDefs/R SourceN
MicroD Sys/O RegDefs Source1 Source2 ... SourceN
```

where Source1 ... SourceN may assemble IM and IFUM locations but must not assemble any RM or ALUFM locations; i.e., forward and external references are permitted only in instruction branch clauses and in target addresses for IFUM locations, so everything else must be predefined in RegDefs.St. 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--some large programs are near to overflowing at present. However, whenever any RM or ALUFM assignments change, it will still be necessary to reassemble everything, and when everything is reassembled the total assembly time will be about 9 minutes rather than 6 minutes.

Note that you do not need 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 "ILC" 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 switches 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 consisting of address symbols for debugging and binary data for various memories; this file is produced unless the global /P switch on the command line suppresses it.

.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 mage.

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	Unused for Dorado
	/M	Produce a .CSMap file
	/N	No listing--IM contents are not listed
	/O	Produce the Occupied.Mc output file containing IMReserve statements for all locations filled by MicroD
	/P	Print only--suppresses 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-placed IM locations--overrides global setting
	/C	Concise listing--overrides global setting
	/L	List everything--overrides global setting
	/N	No IM listing--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 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 Dorado memories and of addresses associated with particular locations in memories. The memories are as follows:

IM	44-bit x 10000-word instruction memory (placement and other information brings total width to 140 bits)
RM	20-bit x 400-word register bank memory
STK	20-bit x 400-word stack memory
IFUM	40-bit x 2000-word instruction fetch unit memory
ALUFM	10-bit x 20-word ALU control memory
BR	40-bit x 40-word base register memory (only for debugging symbols)
BRX	40-bit x 4-word MemBX-relative base register memory (only for debugging symbols)
DEVICE	20-bit x 400-word fake memory for device address symbols
TASKN	20-bit x 20-word fake memory for task number symbols

In addition, four other memories called VERSION, RVREL, IMLOCK, and IMMASK are produced by Micro and consumed by MicroD, but these are invisible to the programmer. Only the data contents for IM and IFUM are affected by the operation of MicroD; address symbols and data for other memories are transmitted to the output file exactly as they are received by MicroD.

There are at present no facilities provided for microcode overlays. However, such a facility will be provided eventually.

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 microprogram. This will 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 FF 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 error messages are intended to be self-explanatory.

4. Debugging Microprograms

There is no simulator for Dorado. Microprograms are debugged directly on the hardware using facilities provided by Midas. To debug microprograms you will need to load Midas and its auxiliary files as discussed in the Midas manual.

Midas facilities consist of a number of hardware tests, a loader for Dorado microprograms, set/clear breakpoints, start, step, or halt the machine, and examine and modify storage. Addresses defined during assembly may be examined on the display. Midas works with both the imaginary IM addresses defined in your source program and with the absolute IM addresses assigned to instructions by MicroD.

5. Cross Reference Listings

The cross-reference program for Dorado microprograms is a Tenex subsystem called MCross. It is significantly easier to maintain large microprograms when cross-reference listings are available, so you are advised 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:        D               (selects Dorado syntax)
Action:         U               (convert to upper case)
Action:         N               (read def's, no printout)
File:           D1LANG<esc>
Action:         CL              (read def's, produce cross ref)
File:           Src1<esc>
Action:         CL
File:           Src2<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). COMCHAR is used as follows. Suppose you want to have conditional assemblies based on whether the microcode is being assembled for a 4K or 16K Dorado configuration. To do this define "~" as the comment character for 4K (i.e., COMCHAR[~];) and "!" as the comment character for 16K. Then in the source files:

```
*! 4K configuration only
  ...statements for 4K configuration...
*!
*~ 16K configuration only
  ...statements for 16K configuration...
*~
```

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

7. Conditional Assembly

DlLang defines IF, ELSEIF, ELSE, 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;
```

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, :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-decimal 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., "FreezeBC," "StkP").

When a symbol is formed by running together the first letters from several words, then these are all capitalized (e.g., "MC").

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. Although 13-character addresses are acceptable, Midas must fit an address symbol, an offset, and the value at that location in a screen window; if total text length exceeds window width then the offset and name are truncated to fit, producing a confusing display image. For this reason, ordinarily limit IM addresses to 9 characters to avoid truncation.

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. puts the assembler in "Emulator" mode and "Subroutine" mode (discussed later).

The final file to be assembled should be terminated with an END statement:

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

Currently, the END statement assembles "MIDASBREAK: Return, BreakPoint, At[7776]" and prints some statistical information about the program on the .Er file. MIDASBREAK is needed to implement the subroutine-call feature which you might use when debugging with Midas. When you are going to independently assemble a number of source files, and then load the .Dib files into MicroD, you should put an END statement *only on the last file to be loaded.*

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.

The hardware has some operations executed differently for emulator, fault, and input/output tasks. For example, stack and ←ID operations are only available to the emulator task; Map← and Flush← only to the emulator and fault tasks; IOFetch← and IOStore← only to io tasks. The TITLE statement initializes to assemble for the emulator task. This can be done independently by the statement:

```
Set[XTask,0];
```

The assembler bases its error-checking upon the value of XTask. The programmer should correctly set XTask to 0 (emulator), 17 (fault task), or some value in the range 1 to 16 (io tasks).

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.

DlLang 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 DlLang 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.

When an instruction statement does not contain a branch clause, the assembler (i.e., MicroD) must cause it to branch to the next instruction inline. For programs which nearly fill the microstore, it is important to allow MicroD flexibility in locating the next instruction. If it is permissible to smash the Link register, then MicroD has more locations to choose from and might be able to pack the microstore tighter.

To tell the assembler whether or not it is ok to smash Link, two declaration statements are provided:

```
Subroutine;
```

tells the assembler that it must not smash Link. The TITLE statement puts the assembler in Subroutine mode.

```
Top Level;
```

tells the assembler that it is ok to smash Link.

There is more detailed discussion on these in the section on branching.

10. Forward References

Micro and DlLang have an *extremely limited* ability to handle forward references. The only legal forward references are to instruction labels from either branch clauses or IFUM assembly statements. 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 such as XTask and to control Repeat statements. The operations given in the table below are included here for completeness, but hopefully you will not have to use any of them except SET:

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 D1Lang 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 ILC. This is originally set to 0 and is bumped every time an instruction is assembled. To do a Repeat, you must directly reference ILC as follows:

```
Repeat[20,ILC[(...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,ILC[(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 DlLang. 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, small 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, small constant, 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 and either 0 or 377 in the other 10-bit byte.

"Literal" constants such as "322C", "177422C", "32400C", or "32377C" may be inserted in instructions without previous definition.

Negative constants such as "-1C", "-55C", etc. are also legal.

Alternatively, constants may be constructed from parameters, integers, or addresses 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. Small Constants

The hardware allows low bits of the FF field in an instruction to be used as operands for functions selected by high bits of the FF field. When used this way, the low bits of FF are called a *small constant.*

"Literal" small constants such as "17S", "1S", etc. may appear in instructions without previous definition. Alternatively, small constants can be defined in advance by the following macro:

```
MSC[NAME,P1,...,P4];        *defines NAME as a small constant with value = sum of P1...P4.
```

Warning: MSC also defines NAME as a parameter. If you redefine the parameter with the same name as the small constant, you will also redefine the small constant.

The RBase[RMADDR] macro is used to construct correct small constants for loading into the RBase← register prior to referencing the RM address or RM region RMADDR; RM references will be discussed later.

Some example clauses using small constants are as follows:

```
RBase←RBase[RTEMP],         *Addresses the RM region containing address RTEMP
Cnt←13S,                    *Loads Cnt with 13
```

Small constants cannot be used as B constants--they are not equivalent to constants.

16. Assembling Base Register Names

Base registers are referred to when loading MemBase and in assembling for IFUM. Base registers are defined by clauses of the form:

```
BR[MDS,1];                  *Define MDS as base register 1
BR[CODEBASE,37];            *Define CODEBASE as base register 37
```

Since your program can also refer to the four base registers pointed at by the MemBX register, a separate construct is provided to define names for this purpose:

```
BRX[LOCAL,n];               *Define LOCAL as MemBX-relative base register n = 0 to 3
```

Address symbols defined by BR and BRX can then be used as in "MemBase←MDS" or "MemBaseX←LOCAL" in the program, and these names can be used in IFUM assembly statements as discussed later. In addition, the address symbols will be used by Midas when pretty-printing microinstructions.

17. Assembling Device Numbers for TIOA

Device numbers are defined symbolically by clauses of the form:

```
DEVICE[DSP,120];
```

This defines DSP as an address symbol that will be used in symbolic printout by Midas when the number 120 appears in the TIOA register.

During assembly the name DSP can then be used as a constant. For example,

```
T←DSP;
TIOA←T;
```

causes TIOA to be loaded with the full 8-bit device address. Alternatively, in situations where you are sure that the high 5 bits of TIOA contain the correct value, you can use the short method of loading TIOA, as follows:

```
TIOA[DSP];
```

This loads the low three bits of TIOA with the low three bits of DSP while leaving the high five bits of TIOA unchanged.

18. Symbolic Use of Task Numbers

Task numbers are defined symbolically by clauses of the form:

```
TASKN[FLT,17];          *Define the fault task as 17
```

FLT can then be referred to during assembly by the following kinds of clauses:

```
RdTPC←FLT;              *Read TPC[FLT] into Link
LdTPC←FLT;              *Load TPC[FLT] from the value in Link
Wakeup[FLT];            *Initiate a wakeup for FLT
```

FLT will also be passed as an address to Midas and will be used in printing the contents of CTASK, TASK, NEXT, etc. symbolically. Since Midas may eventually print task specific register addresses symbolically also (e.g. TIOA DSPTSK 120, RBASE DSPTSK 10), it is desirable to keep the task names fairly short to avoid using too much of the display window during debugging.

The task names EMU for the emulator (task 0) and FLT for the fault task (17) are predefined in D1Lang.mc.

19. Assembling Data for RM

The assembler can assemble data and assign addresses in RM in several ways discussed in this section.

The hardware allows any one of the 20 registers pointed to by RBase to be read/written by an instruction. To define 20-long regions of RM:

```
RMRegion[RGNNAME];        *allocates a 20-long region RGNNAME
```

New regions are allocated in blocks of 20 starting at 0 and ascending by 20 for each new region. I.e., the first region you define is 0 to 17, next 20 to 37, etc. up to 360 to 377.

After defining a region with RMRegion, you can proceed immediately to define addresses in that memory using the macros given below. Alternatively, you can reselect that region at a later time:

```
SetRMRegion[RGNNAME];     *reselects a region RGNNAME for allocation
```

"Literals" in RM, such as "32224R", may be referenced in instructions without previous definition. The first reference to an RM literal will assemble the literal value into a register in the current region. Subsequent references to the literal will refer to the same location in RM. It is illegal to duplicate a literal in more than one region.

Other ways of assigning storage in a region are as follows:

```
RV[NAME,P1,...,P8];       *define NAME as an address in the current region with value = sum of
                          *parameters.
RVN[NAME];                *Defines NAME in current region without initial value.
Reserve[N];               *Skips N (an integer < 20) words in the region
```

Define variables with RVN rather than a useless initial value because this will prevent the "Cmpr" action in Midas (which compares the microstore image against what you loaded) from reporting fictitious errors. In system microcode (as opposed to diagnostics), any occurrence of a variable with an initial value is probably a programming error since it requires a reload to restore the initial value. Hence, if you have variables with initial values, you probably should store the initial values elsewhere (in IM, for example), define the variables with RVN, and copy the initial values into the registers during initialization.

The assembler checks at assembly-time that the base for an RM address agrees with the value believed to be in RBase. This is controlled by the following macros:

```
KnowRBase[RGNNAME];       *declares RGNNAME to be in RBase
DontKnowRBase;            *declares that the contents of RBase are unknown
```

The small-constant clauses that load RBase (see the Small Constants section) change the assembly-time parameter for the current region.

It may be convenient to code subroutines that use temporary storage in RM and are called with several possible values in RBase. For example, a programming convention might reserve the first two words in each of several regions as temporary storage for subroutines. Then the subroutine will want to reference these words in a regionless way.

To do this, first reserve temporaries for each region (using Reserve). Then define a regionless RM address as follows:

```
RVREL[NAME,DISP];        *declares NAME a regionless address in RM with displacement DISP
                         *relative to the current value in RBase.
```

The subroutine would then reference NAME. These references would not cause "Invalid RBase" error messages.

20. Assembling Data for STK

The hardware allows the 400-word STK memory to be treated as four separate stacks of 100 words each. We expect at least one of these four stacks to be used as the Mesa evaluation stack, and perhaps all four will be used for this purpose, representing different procedure frames or something.

However, it is also possible that one or more of the four STK regions will be used to hold data in some way other than as a stack. For example, STK is the best available memory for (emulator-only) tables because it is the only memory with an indexable address.

The assembler has several macros that allow data to be assembled for STK. These are:

```
STKRegion[i];            *where i = 0 to 3 selects one of the four STK regions for allocation.
STKWrd[i];               *where i = 0 to 77, sets the word for the next allocation.
STKVal[p0,p1, ... , p8]; *sums up to 9 parameters and stores the result in the currently selected
                         *STK word, and then advances to the next word.
```

21. 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];
Data[(TABLE1: Byt0[P1,...,P8] Byt1[...] Byt2[...] Byt3[...], At[T1Loc])];
Data[(Byt0[P1,...,P8] Byt1[...] Byt2[...] Byt3[...], 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. Byt0, Byt1, Byt2, and Byt3 assemble for the different 9-bit bytes of the instruction and correspond to the bits read by ReadIM[0], ReadIM[1], ReadIM[2], and ReadIM[3]. "At" is discused in the "Placement" section later. Sample sequences for reading and writing IM are given in the "Programming Examples" section.

It is important to note that while Byt0 and Byt2 may assemble data items up to 9 bits wide, Byt1 and Byt3 are limited to 8-bit wide items because the 9th bits for these are parity bits, and the assembler will not let you load bad parity into IM as part of a data item.

22. Assembling for IFUM

Micro assembles data for an imaginary IFU identical to the Dorado IFU except that the address of the target location in IM is a full 14-bit address rather than the compact 12-bit form used by Dorado. MicroD imposes the necessary placement constraints on IFU target addresses in IM and transforms the imaginary IFU instruction into the form expected by Dorado.

Before using any of the IFU macros discussed below, you must declare the instruction set number as follows:

```
InsSet[0,1];    *Declares instruction set 0 with 1 instruction per entry vector
InsSet[1,4];    *Declares instruction set 1 with 4 instructions per entry vector
InsSet[2,4];    *Declares instruction set 2 with 4 instructions per entry vector
InsSet[3,1];    *Declares instruction set 3 with 1 instruction per entry vector
```

You can define parameters or integers to use in place of the literal arguments to InsSet, if you want to.

It is desirable, though not required, for an IFUM assembly statement to be given *after* its target instruction has been assembled to avoid a forward reference. This reduces work required by both Micro and MicroD, so the assembly will run faster.

There are three macros defined for assembling IFUm words. These are:

```
IFUReg   [opcode, length, memb, rbaseb, ifad, n, sign, pa];   *Regular opcode
IFUPause [opcode, length, memb, rbaseb, ifad, n, sign, pa];   *Pause opcode
IFUJmp   [opcode, 2,      memb, rbaseb, ifad, sign];          *2-byte jump opcode
IFUJmp   [opcode, 1,      memb, rbaseb, ifad, disp];          *1-byte jump opcode
```

In these:

Argument	*Value*	*Effect*
opcode	0 to 377	Together with InsSet, opcode specifies the IFUM word to be loaded.
length	1 to 3	Number of bytes in the opcode.
n	0 to 16	First operand delivered by ←ID -or-
	17	if no operand.
sign	0 or 1	Operand sign or sign extension (see hardware documentation).
pa	0 or 1	0 if not packed-alpha, 1 if packed-alpha.
memb	BRX address	0..MemBX[0:1]..address loaded into MemBase -or-
	BR address	The BR address must identify BR 34 to BR 37; a value of 4 to 7 is assembled into memb; 34+memb[1:2] is loaded into MemBase at t_0 of the entry instruction.
rbaseb	0 or 1	RM region loaded into RBase at t_0 of the entry instruction.
ifad	IM address	the tag on the first instruction in the IFU entry vector for the opcode.
disp	-40 to +37	The displacement of a one-byte jump opcode (fills in the n, sign, and pa fields appropriately)

"ifad" is the address of the first instruction in the entry vector for the opcode. The last InsSet pseudooperation specified the number of entry instructions per vector. You must put these instructions in sequence in the source file beginning with the one whose tag is "ifad." **Warning:** The assembler does not check that you have done this.

For one-byte jump opcodes, the assembler will correctly transform the "disp" argument, which should be an integer in the range -40 to +37, into proper values for the "n," "sign," and "pa" fields of the IFUM word (see hardware manual). For two-byte jumps, only the "sign" field is utilized by the hardware, and the assembler will put don't-care values into "n" and "pa."

Since the above macros are fairly jaw-breaking (six or eight arguments), you may wish to use Micro's macro-definition capability to define shorter forms for your own use. You may also wish to use the "Repeat" builtin to define blocks of opcodes. Here are two examples:

```
*Macro to define Byte Lisp opcodes of length 1
*Byte Lisp is instruction set number 1
M[RegOp,IFUReg[#1,1,MDS,0,#2,17,0,0]];

RegOp[0,CONS];
RegOp[1,RPLACA];
RegOp[2,RPLACD];

*Generate 20 one-byte jump instructions with displacements from 1 to 20
Set[DX,1];
Repeat[20,IFUJmp[Add[77,DX],1,MDS,JMP,DX]  Set[DX,Add[DX,1]]];
```

I think that assembling data for IFUM is the only place where falling back to basic Micro constructs such as Macro definition will be desirable.

23. Assembling for ALUFM

The processor's Alu provides all 20 boolean operations on A and B and 20 arithmetic operations, each with an optional carry. Only about 11 of the 40 possible arithmetic functions seem to me to have any hope of application (See the MECL System Design Handbook specification for the MC10181 ic's, if you are curious about the others.)

The 20-word ALUFM memory contains six Alu control signals in each word. Since only 20 of the 31 potentially useful Alu operations can be stored in ALUFM, you must define a suitable subset by editing the DlAlu.Mc file according to instructions on its first page; this file must be assembled after DlLang as discussed in the "Assembly Procedures" section. This will assemble storage for ALUFM and define all macros pertinent to each Alu operation. 16 operations believed most useful are *'ed in the table below, leaving two operations unspecified.

Addresses 16 and 17 are special--these are the two addresses used by shift operations. By convention, ALUFM 16 must contain the controls for "not A" (so the shifter output on A, which is inverted, will go through the Alu to the masker). ALUFM 17 is not presently constrained (Perhaps the "A" operation, which would allow shifter output to be inverted before masking, is a good choice for ALUFM 17).

Note: You must define both "A" and "B" because those sources which can be optionally routed over either A or B (i.e., RM, T, Q, and MD) require that both paths be available through the Alu.

Both "not A" and "not B" must be defined for the same reason (and "not A" is required for the shifter).

The XorCarry and XorSavedCarry functions modify the input carry and the Carry20 function OR's a 1 into the carry out of the low-order four-bit ALU slice; these are written as standalone clauses; since carries affect only arithmetic, they are useless noops with logical operations.

BitBlt is expected to use ALUFM 15 and 17 as variables. However, if BitBlt restores these upon exit, and if they are preserved across map faults and other traps, the emulator will be able to use two other operations; you should edit DlAlu.Mc to define these operations as "emulator-only" so that an error will be flagged if they are erroneously used by any task other than the emulator. To support restoration of these prior to exit from BitBlt, DlLang defines symbols AFMn (n = 0 to 17) as constants equal to the normal contents of corresponding ALUFM locations. These allow normal contents of an ALUFM location to be restored as follows:

```
T←AFM17;                *T ← normal contents of ALUFM location 17
ALUFMRW←T, ALUF[17];
```

The AFMn constants are also used to initialize ALUFM during boot-loading.

Below, the potentially useful Alu operations are listed:

	A0	0
	A1	-1
*	A	A straight through (arithmetic)
	A	A straignt through (logical)
*	B	B straight through
*	not A	ones complement of A
*	not B	ones complement of B
*	A and B	logical and
	not A and B	parsed (not A) and B
	not A and not B	parsed (not A) and (not B)
	A and not B	parsed A and (not B)
*	A or B	inclusive 'OR'
	A or not B	parsed A or (not B)
	not A or B	parsed (not A) or B
	not A or not B	parsed (not A) or (not B)
*	A#B	exclusive 'OR'
*	A xor B	exclusive 'OR' (alternate syntax)
	A=B	equivalence or exclusive nor
	A xnor B	equivalence (alternate syntax)
	A eqv B	equivalence (alternate syntax)
*	A+B	sum
*	A+B+1	
*	A-B-1	equals A+(not B)
*	A-B	difference
*	A-1	
*	A+1	
	2A	A+A
	2A+1	

* Indicates that these are defined for system microcode

24. General Comments on Instruction Statements

The general layout of an instruction statement is as follows:

```
TAG:  branch clause, T←MD, rmaddr←(A phrase) and (B phrase), function, placement;
```

TAG is an instruction memory address symbol. This may be used in branch clauses as discussed later. Micro places instructions sequentially starting at 0; then the postprocessing program MicroD relocates, or places, the instructions at their runtime locations.

Branch and placement clauses are discussed later. They present no special problems in understanding and are easy to program.

Functions that don't involve routing data from one place to another are also easy to program--you just write the appropriate name as a separate clause in the instruction, as discussed in the "Standalone Clauses" section.

However, clauses that involve moving data from one place to another are tricky. This section tries to present the general concepts behind programming these clauses.

Data-routing clauses all 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 "source" to "destinations".

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

MemBase←MDS,	
T←RMADDR,	The assembler figures out how to route data from the RM address RMADDR to B (or A if B has been used) then through the Alu and into T.
RMADDR←34C,	Again the assembler figures out how to construct the constant 34, route it onto B and through the Alu, and into the correct RM address.

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

T←(Fetch←RMADDR)+1,	The assembler routes the contents of the RM address RMADDR onto both A and Mar, does A+1 in the Alu, routes the Alu onto PD, and stores PD in T.
T←(Fetch←T)+(Q←RMADDR)	The assembler routes RM address RMADDR onto B and into Q, T onto both A and Mar, performs "A+B" in the Alu, routes the Alu onto PD, and loads PD into T.

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. The proper value is assembled for the RStk field of the instruction. At this point data from RMADDR might be routed over either A or B.

b. Fetch← is looked up next. The ASel field is set to cause the fetch, and RM is routed onto A using FF[0:1] (There are some complications when FF has been used as a constant.).

c. A+1 is recognized as an Alu operation, AluF is set to cause A+1, and the data is now at PD.

d. Finally, the assembler identifies T←PD and modifies the LC field appropriately.

Note: The "()" in the above example 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←Fetch←RMADDR,	is legal
T←Fetch←(RMADDR),	is legal
T←(Fetch←RMADDR),	is legal
Fetch←T←RMADDR,	is illegal

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

Here are some more "()" examples:

T←T+(RMADDR),	is legal--"()" are mandatory around all A and B sources except "T", "MD", "Q", and "ID".
T←(T)+(RMADDR),	is legal--"()" optional around "T", "MD", "Q", and "ID".
T←(T+(RMADDR)),	is legal--extra "()" around an entire source always OK
(T←T+(RMADDR)),	is legal
T←(T+(RMADDR)) rsh 1,	is legal--"()" are mandatory
T←T+(RMADDR) rsh 1,	is illegal--the assembler will evaluate RMADDR OK, but it won't recognize the rest.

You must also write clauses in the correct order. Since the assembler evaluates clauses from right-to-left, whenever there are different ways to do something, the assembler will pick one of the ways, and you must be sure that it makes the correct choice by putting the clauses in the correct order.

The five decisions of interest are:

a. Branch clauses--the assembler has to know whether FF is available for a branch condition, long goto, or long call. The assembler has special stuff to figure this out, so you can position the branch clause anywhere in the instruction statement.

b. RM, T, MD, or Q routed through Alu clauses--the assembler will use B for this routing unless you have already used B for something else. Hence, if you are using B for something else, put that clause to the right in the instruction statement.

c. Different RM addresses for read and write--Be sure that the RADDR2← destination is to the *left* of the RADDR1 source in the instruction statement.

d. Shifter operations that OR shifter output with another A source before routing through the Alu--this is rarely done, ordinarily an error. If you really want to do this, the *shifter clause should be to the right* in the instruction statement.

e. Put FF>77 clauses to the right of A clauses. If you violate this rule, correct statements will still assemble correctly, but some erroneous statements won't get checked by the assembler.

I think that the above cases are the only ones where the assembler's decision algorithms might cause trouble. Obeying (b) through (f) above nearly always produces a statement that will assemble correctly, and I think that any legal instruction can be written in a way that satisfies these. If you encounter problems with these rules, see me.

25. Small Constant Clauses

The hardware allows low bits of FF in instructions to be used as literal values in several places. The valid clauses for these uses are:

MemBase←SC,	MemBase loaded from small constant (unusual) -or-
MemBase←MDS	MemBase loaded from value of BR address symbol (usual)
RBase←SC,	RBase loaded from small constant (unusual)
RBase←RBase[RADDR]	-or-
RBase←RBase[RGNNAME]	Normal ways to load RBase
CNT←SC,	Loop counter from small constant
MemBX←SC,	MemBX loaded with 0 to 3
MemBaseX←SC,	MemBase loaded with 0..MemBX[0:1]..SC (unusual) -or-
MemBaseX←LOCAL	where LOCAL is a BRX address (usual) -or-
MemBase←LOCAL	same as MemBase←LOCAL
A←SC,	A from 0 to 17 (usually embedded in Alu expression)
Fetch←SC,	Memory reference from small constant

In the above "SC" can be any of the small constant tokens discussed in the earlier section on small constants, namely, a literal like "14S" or a name defined by MSC.

The blocks of FF decodes for waking up tasks and loading the low bits of TIOA are not written as small constant clauses. Instead Wakeup[task] and TIOA[device] are used (i.e., standalone clauses); the arguments to these should be address symbols in the TASKN and DEVICE memories, respectively, as discussed earlier. An alternative syntax for Wakeup[task] is Notify[n], where n is an integer (0 to 17) specifying the task to be awakened; this alternate syntax is not expected to be useful except for diagnostics checking out the control section.

26. A Clauses

If you study the way in which various operations are encoded in the ASel and FF fields of the instruction (see hardware manual), you will discover that activity on A is usually encoded in the ASel field, leaving FF available for encoding another function. Memory references use FF[0:1] as well as ASel, leaving FF[2:7] available for encoding one of 100 common functions and branch conditions.

When FF is used literally as a constant on B, the A source is limited to RM, T, or ID, and the source for memory references is limited to T or RM.

Shift operations are hardware sources of A, but the assembler treats these as PD outputs. In other words, the various shift-and-mask statements assemble the "shift" value into ASel and select either the "not A" Alu operation (AluF = 16) or the variable Alu operation (AluF = 17), as discussed in the shifter section.

An A source can be any of the following:

A	Dummy source for clause splitting. It indicates that the source for some destination is A (You probably won't ever use this because normally the A source and destination are written as a single clause.).
RADDR	An RM address (never uses FF except to OR with shifts)
T	Uses only ASel if not with a memory reference, only FF[0:1] with Fetch← or Store←, or FF (< 100) with other memory references
ID	Uses only ASel if not with a memory reference, only FF[0:1] with Fetch← or Store←, undefined for other memory references
MD	Uses ASel and FF
Q	Uses FF (< 100)
0S to 17S	Uses FF (< 100)

Rule: You have to enclose small constant and RM addresses in "()" when they are embedded in an Alu expression; "()" are optional for other A sources.

Destinations for A are the Alu, the eight memory references and other control functions in the memory section, and RF← and WF← for shifter-setup. A destinations can be any of the following:

Fetch←	Start memory fetch--uses ASel only
IFetch←	Start memory fetch in which ID from the IFU replaces low bits of BR (see hardware manual)--uses ASel and FF[0:1] (emulator only).
LongFetch←	Start memory fetch in which B extends the displacement on Mar (see hardware manual)--uses ASel and FF[0:1].
Store←	Start memory store--uses ASel only (data for write must be put on B in a separate clause)
IOFetch←	Start 20-word fetch on behalf of io device--uses ASel and FF[0:1] (illegal in emulator or fault task)
IOStore←	Start 20-word store on behalf of io device--uses ASel and FF[0:1] (illegal in emulator or fault task)
PreFetch←	Put 20-word data unit in cache--uses ASel and FF[0:1]
Map←	Write Map for VA, data on B--uses ASel and FF[0:1] (emulator or fault task only)
RMap←	Does the Map← reference together with the ReadMAP function, suppressing the write of the Map so that old data can be read non-destructively in the Pipe; since the FF field is used for ReadMAP, the displacement for the reference can only come from an RM/STK address (emulator or fault task only).
Flush←	Purge VA from cache--uses ASel and FF[0:1] (emulator or fault task only)

DummyRef←	Loads VA into the Pipe (for reading BR's)
any Alu expression	or destination
BRLo←	Uses FF (> 77)
BRHi←	Uses FF (> 77)
CFlags←	Uses FF (> 77)
RF←	Uses FF (> 77). Read-field setup for shifter.
WF←	Uses FF (> 77). Write-field setup for shifter.
A←	No-op destination--simply routes the source onto A

The primary memory operations Fetch← and Store←, do not prevent FF from being used as a B constant, an "insert field" or "extract field" value, or a long goto/call. FF can be used in these ways if the A source is an RM address or T. However, when FF is not used in these ways, FF[0:1] must encode the source for the reference (RM, T, or ID). The eight secondary memory references (IFetch←, LongFetch←, Map←, Flush←, IOFetch←, IOStore←, DummyRef←, and PreFetch←) require FF[0:1] for encoding. This means that only functions 0 to 77 can be encoded in the same instruction with a memory operation, and secondary memory references consume FF to specify any source other than an RM address.

The above can be combined arbitrary ways and used in Alu expressions, e.g.:

```
RADDR←T←Fetch←ID,
T←RADDR←Fetch←2S,
T←(Fetch←T)+(RADDR),
```

The MCR register has some bits loaded from A and some from B. This is encoded by a stand-alone clause of the form "LoadMCR[A,B]", where A and B are A and B source clauses respectively, as discussed later.

Warning: If you illegally write an expression that uses one of the FF decodes between 100 and 377 to the left of a memory reference clause with RM or T as the source, the assembler will not detect the error. .

27. B Clauses

The common B sources can be selected by the BSel field in the instruction, leaving FF free for other uses. Less common sources are encoded by functions (FF > 77), and for these BSel may optionally encode the Q← destination. With common sources, B destinations are selected by functions.

The implications of this arrangement are as follows:

You can route any common B source (RM, T, MD, or Q) through the Alu into RM or T while using FF for something else.

You can route any common source to any destination by using FF.

You can route any uncommon source through the Alu or to Q← by using FF.

B sources can be any of the following (all of the FF's are > 77):

B	Dummy source for clause splitting (You probably will never use this since it is normally possible to embed the B phrase inside the B arm of the Alu phrase.)
Constants	Uses FF--see earlier constant section
RADDR	
T	
MD	Memory data
Q	
Link	Uses FF
RWCPReg	Uses FF (does Link←B' and B←CPReg, intended for Midas)--overrules loading of Link by Call or Return in the same instruction
DBuf	Uses FF
FaultInfo'	Uses FF
Pipe0	Uses FF (VAHi is a synonym)
Pipe1	Uses FF (VALo is a synonym)
Pipe2'	Uses FF
Pipe3'	Uses FF (Map' is a synonym)
Pipe4'	Uses FF (Errors' is a synonym)
Config'	Module indicators and IC size stuff--uses FF
Pipe5	Uses FF (PRef is a synonym)--cache stuff and reference flags
PCX'	Uses FF
EvCntA'	Uses FF
EvCntB'	Uses FF
IFUMRH'	Low part of IFU data--uses FF
IFUMLH'	High part of IFU data--uses FF

Note: All FF-encoded B sources except PCX', EvCntA', and EvCntB' (i.e., those from the IFU board) are too slow for Alu arithmetic; the assembler will flag an error if you use any other FF-encoded B source in an arithmetic expression.

Rule: When used in an Alu phrase, B sources must be enclosed in "()", except that "()" are optional around "T", "Q", and "MD".

B destinations can be any of the following:

B←	No-op destination--simply routes source onto B
DBuf←	A no-op destination. Actual loading of Dbuf is enabled by the Store← memory operation; this is just for readability.
MapBuf←	Another no-op destination. Actual loading of MapBuf is enabled by the Map← memory operation.
Q←	Uses FF (< 100) ordinarily, or BSel with FF-encoded B sources
Cnt←	Uses FF (> 77)
Link←	Uses FF (> 77)--overrules loading of Link by Call, Return, or CoReturn in same instruction
PCF←	Uses FF (> 77)--loads PC and starts IFU
StkP←	Uses FF (> 77)
TIOA←	Uses FF (> 77)--TIOA loaded from B[0:7]
Output←	Uses FF (< 100)
MemBase←	Uses FF (> 77); loads from B[3:7].
RBase←	Uses FF (> 77); loads from B[14:17].

Pointers←	Uses FF (> 77); does both MemBase←B[3:7] and RBase←B[14:17].
ShC←	Uses FF (see "Shifter Clauses")
Hold&TaskSim←	Uses FF (> 77)
InsSetOrEvent←	Uses FF (> 77)
MOS←	Uses FF (> 77), synonym for InsSetOrEvent←
GenOut←	Uses FF (> 77), general output to io devices
EventCntB←	Uses FF (> 77), synonym for GenOut←
IFUMRH←	Uses FF (> 77) (B must remain good through the following instruction--see hardware manual).
IFUMLH←	Uses FF (> 77) (B must regain good through the following instruction)
BrkIns←	Uses FF (> 77); loads from B[0:7]
IFUTest←	Uses FF (> 77)
ALUFMRW←	Uses FF (> 77)--this one is simultaneously a B destination and an ALU source--the clause "AluF[n]" must appear in the same instruction where n (0 to 17) selects the location in ALUFM that is read and written
LdTPC←	Uses JCN, forces low bit of RStk to be 1. Data on B is the task number whose PC is loaded from Link.
RdTPC←	Uses JCN, forces low bit of RStk to be 0. Data on B is the task number whose PC is read into Link.
IMLHR0Pok←	
IMLHR0Pbad←	
IMLHR0'Pok←	
IMLHR0'Pbad←	
IMRHBPok←	
IMRHBPbad←	
IMRHB'Pok←	
IMRHB'Pbad←	Use JCN and RStk[1:3]. These 10 kludge destinations are various forms for writing IM, where the parity (POK or PBAD), the 21st data bit (RStk.0 or RStk.0' for the left-half, Block or Block' for the right-half), and the half-word (left or right) are specified in the macro name and assembled into RStk[1:3]. The other 20 data bits are written from B.
BDispatch←	Uses FF (< 100). 8-way dispatch on B[13:15]
BigBDispatch←	Uses FF (< 100). 256-way dispatch on B[8:15].
MidasStrobe←	Uses FF (> 77). Shifts out DMux addess bit in B[4].
any ALU destination	

Some examples of instructions which read and write ALUFM are as follows:

T←(ALUFMRW←T), A+B+1;	Loads ALUFM[X] with value in T, where X is the location assembled with control for A+B+1; the old contents of ALUFM[X] are saved in T.
T←ALUFMEM, AluF[3];	Reads ALUFM[3] through PD into T

28. RM and STK Clauses

The hardware complicates references to RM by providing only four bits of RM address in the instruction (The Block bit in combination with these four encodes stack reference options for the emulator task). The remaining four address bits come from the RBase register which the programmer must load appropriately before the reference.

Micro will flag an error if an RM read reference is not in the 20-word RM region believed to be pointed at by RBase, and it will use the change-RBase-for-write FF decodes (FF > 77) for RM write references outside the current region. To disable error checking, the programmer must define and reference regionless RVREL addresses, as discussed in the "Assembling Data For RM" section; RVREL addresses should be used only in a section of code which has multiple entries that setup RBase with different values.

The current region is specified by the following statements:

```
DontKnowRBase;          *Contents of RBase is unknown--any read reference is an error
KnowRBase[RGNNAME];     *RBase points at the RM region RGNNAME
KnowRBase[RADDR];       *RBase points at the RM region containing the address RADDR
```

The TITLE statement also declares DontKnowRBase.

In addition, RBase may be loaded, as discussed earlier, by a clause (FF > 77) of the form:

```
RBase←RBase[RADDR], -or-
RBase←RBase[RGNNAME],
```

These both load RBase and declare KnowRBase[RGNNAME], so subsequent instructions will be assembled assuming that the newly-loaded region is in RBase; this is normally what the programmer wants.

Since (at t_0 of the entry instruction for an opcode) the IFU initializes RBase to point at the RM 0 or RM 20 region, the programmer should usually insert a KnowRBase[RGN0] or KnowRBase[RGN1] declaration before this.

It is permitted to both read and write RM in one instruction. Normally, the read and write addresses are identical. However, a block of 20 functions (FF < 100) changes the RM address for the write, permitting different registers in the current region to be read and written. These functions are also used when the stack is referenced during the read portion of the instruction and a register in the current RM region during the write part. However, there is no way to read from RM and write the STK in the same instruction.

Another block of 20 functions (FF > 77) changes the RBase part of the RM write address. The assembler will output one of these functions when it believes that RBase does not contain the proper value for writing the selected register.

Warning: If you inadvertently write an illegal statement like:

```
RADDR2←Fetch←RADDR1; -or-
RADDR2←Fetch←T; -or-
RADDR2←Store←T, MD←Q;
```

where RADDR2 is not in the current RM region, the assembler will produce a garbage instruction because the change-RBase-for-write function produced for RADDR2← will clobber the FF[0:1] field which must not be clobbered by anything to the left of the Fetch← or Store←. The assembler will not flag this error.

RM addresses can be used as sources for A or B destinations, and this doesn't require any extra fields in the instruction. RM addresses can be used in Alu phrases, and, in this case, the RM address has to be enclosed in "()".

RM addresses can be destinations for Alu operations, for the Input, InputNoPE, Cnt, Pointers, TIOA&StkP, ALUFMem, and ALUFMRW functions, and for MD--they can also be destinations for Alu sources. For these simply write the register name followed by "←".

Some examples of RM clauses are the following:

```
RTEMP1←RTEMP0,
        uses FF because address written different from address read

Fetch←RTEMP0, Q←RTEMP0,
        routes RTEMP0 onto A for a Fetch and onto B to save it in Q. These have to be written as
        separate clauses.

RTEMP0←(ALUFMRW←RTEMP0), AluF[17]
        loads ALUFM[17] from RTEMP0 and saves the old value of ALUFM[17] in RTEMP0.

RTEMP0←Input
        routes IOB data onto PD and store in RTEMP0.
```

It is (just barely) conceivable that you may wish to create a constant whose value is an RM address. To do this you can use the following kludge:

```
B←IP[RADDR]C
```

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

References to STK are illegal except in the emulator task. The operations read the word pointed to by StkP, then adjust the stack pointer in the ways below, then write the selected item (if any) at the modified or unmodified address. When you write at the modified address, the assembler will automatically supply the ModStkPBeforeW FF decode.

StkP[0:1] select one of two separate stacks, and StkP[2:7] address the word in the stack selected by StkP[0:1].

The hardware StkError signal occurs, waking up task 17, before any instruction in which StkP overflows or underflows. However, when StkP is initially 0, underflow should usually occur when TOS (top-of-stack) is referenced in that instruction but not when the pointer is incremented without reference. Hence, the assembler outputs a different code in RStk when incrementing StkP without reference than when incrementing in conjunction with a read of TOS. In addition, an example in the "Instruction Fetch Unit" chapter of the hardware manual shows several situations where TOS is copied into T without knowing whether the stack is empty; in this case the programmer wants to disable the underflow check, even though he is referencing TOS.

The stuff provided by the assembler assumes that your program will use the stack in the manner envisioned by the hardware design, as follows: An empty stack is represented by StkP containing 0; StkP will sensibly point at either the last item pushed or the item before that, according to programming convention; a push increments StkP and writes in one instruction (or increments StkP in one instruction and writes in the next according to programming convention); a pop can read the item being popped in the same instruction if desired.

Names in the first column below are "sources" for reading the top STK entry; the name modifier, "&+n" or "&-n" controls StkP modification, which always occurs *after* the top stack entry is read. Second column names are used instead of first column names when it is permissible for the stack to be empty--this aims at the case when TOS is copied into T without knowing whether or not the stack is empty. The third column are "destinations" for writing the top STK entry; here also StkP is modified *after* the top stack entry is written. The fourth column are destinations that modify StkP *before* writing; they use the ModStkPBeforeW function (FF<100) to do this. Finally, the fifth column are used to modify StkP in an instruction which makes no reference to STK.

Read Stack	*Mod StkP No StkP=0 UFL*	*Mod StkP After write*	*Mod StkP Before write*	*Mod StkP No ref*
Stack&+3	StackNOUFL&+3	Stack&+3←	Stack+3←	StkP+3
Stack&+2	StackNOUFL&+2	Stack&+2←	Stack+2←	StkP+2
Stack&+1	StackNOUFL&+1	Stack&+1←	Stack+1←	StkP+1
Stack	StackNOUFL	Stack←		
Stack&-1		Stack&-1←	Stack-1←	StkP-1
Stack&-2		Stack&-2←	Stack-2←	StkP-2
Stack&-3		Stack&-3←	Stack-3←	StkP-3
Stack&-4		Stack&-4←	Stack-4←	StkP-4

The RStk[0] bit will wind up equal to 1 whenever the StkP=0 underflow check should be made by the hardware. When the stack appears as both a source and a destination in the instruction, the modifiers must match, so the Stack&+i source can only be used with the Stack+i← or Stack&+i← destinations.

StkP← may be loaded from B. RestoreStkP is a standalone function, written as a separate clause.

29. Shifter Clauses

The shifter may be used in two ways. The first way specifies the shift operation and the other control information for the shift in a single instruction. This uses up the ASel, BSel, and FF fields of the instruction while allowing data in either T or a selected RM address to source the shifter.

The second method specifies that shift controls loaded into SHC by a previous instruction (via ShC←B, WF←A, or RF←A) be used. In this case only the ASel field is used up by the shift operation.

The semantic shifter operations when shift controls are specified in FF are as follows (The rightmost 20 bits of the 40-bit quantity participating in a cycle are the result.):

a. Left shift data in RM or in T by 0-17;
b. Right shift data in RM or in T by 0-17;
c. Right-justify (or load) an arbitrary field from RM or T;
d. Right cycle the 40-bit quantity RM.T, T.RM, RM.RM, or T.T by 0-17;
e. Left cycle the 40-bit quantity RM.T, T.RM, RM.RM, or T.T by 0-17;
f. Deposit RM or T into an arbitrary field of a word coming from MD.
g. Deposit RM or T into an arbitrary field of a word of zeroes.

For these, the assembler fabricates an FF value describing the shift, sets BSel to cause the FF-controlled shift, and forces ASel to select a shift. "LdF" stands for "load-field" and "DpF" for "deposit-field". The assembler defines the following macros:

```
T←Lsh[x,count,y];      *Invokes ShiftRMask (or ShMDRMask)
T←Rsh[x,count,y];      *Invokes ShiftLMask (or ShMDLMask)
T←LdF[x,size,pos,y];   *Invokes ShiftLMask (or ShMDLMask)
T←DpF[x,size,pos,y];   *Invokes ShiftBothMasks (or ShMDBothMasks)
T←Rcy[u,v,count];      *Invokes ShiftNoMask
T←Lcy[u,v,count];      *Invokes ShiftNoMask
```

In the above:

count =	distance of the shift or cycle (0 <= count <= 17)
size =	number of bits in the field
pos =	number of bits to the right of the field
x =	source for the shift (RADDR or T)
y =	value replacing masked-out bits (0 or MD, defaulted to 0 if the arg is omitted)
u and v =	T and an RADDR in any combination
count =	distance of the shift.

The macros given above invoke the Alu operation in ALUFM 16 which should be "not A"; the equivalent macros named XLsh, XRsh, XLdF, XDpF, XRcy, and XLcy invoke the Alu operation in ALUFM 17.

The above pseudo-operations include all of the conceptual shift options which are possible when the shift function is carried out in the same instruction with loading ShC. There are no other clever uses of FF-controlled shifts that I am aware of.

Notes:

1) The hardware does not allow an arithmetic Alu operation in conjunction with the Shift...Mask, and the assembler does not check for this.

2) The Alu<0, Alu=0, Carry', and Overflow' branch conditions apply to the output of the Alu *before* it has gone through the masker, so the value tested by these is generally different from the value produced by the shift-and-mask. However, when ShiftNoMask is used, the Alu=0 branch condition will still apply.

The hardware also has three ways of loading ShC←. These are implemented by functions. For these you write a separate clause as follows:

```
RF←A,       Read-field (do ShiftLMask later)
WF←A,       Write-field (do ShMDBothMasks later)
ShC←B,      General
```

where "B" is any B source and "A" any A source.

You should study the shift control figure in the hardware manual to absorb how these work. The most general control of the shifter (ShC←B) allows the 40-bit input quantity to be specified as either T..T, T..R, R..T, or R..R. In other words, you can carry out either a 20-bit or 40-bit cycle by choosing the shifter input control bits appropriately. In addition, if you use the RIsID or TIsID (FF < 100) function in the instruction that carries out the shift, you can replace either the T or RM/STK component of the shift by ID, conceivably useful.

The read-field and write-field operations were intended to support corresponding Mesa operations, which put 8-bit field descriptors on A (usually from ID). Since these use both FF and A, while providing no capabilities beyond that of the FF-controlled shift, it won't normally be convenient to use them in other contexts.

For situations when you need more flexibility than is provided by the FF-controlled shifts, the assembler defines the following macro for constructing complete 20-bit shifter-control constants in RM:

```
RVSH[NAME,LMASK,RMASK,TRSEL,COUNT];
```

"NAME" is the name of the RM variable, TRSEL is 0 to specify R..R as the 40-bit input to the shifter, 1 to specify R..T, 2 for T..R, and 3 for T..T. The interpretation of the other arguments is discussed in the hardware manual. Having constructed such a descriptor, you can load SHC← from it as follows:

```
ShC←NAME
```

The shift hardware forces the Alu operation to be the one defined by ALUFM 16 or ALUFM 17. By convention, ALUFM 16 contains the "not A" Alu operation--this is the one ordinarily required because the shifter output appears complemented on A and is normally routed straight through the Alu to the masker.

ALUFM 17 is (will probably be) reserved as a variable. BitBlt will load ALUFM 17 with assorted controls to accomplish the strange things it does.

The Shift...Mask and ShMD...Mask functions may be written with the RADDR input to the shifter as an argument, or the RADDR argument may be omitted if irrelevant or specified elsewhere in the instruction, as follows:

For Aluf = 16	*For Aluf = 17*
ShiftNoMask[RADDR]	XShiftNoMask[RADDR]
ShiftLMask[RADDR]	XShiftLMask[RADDR]
ShiftRMask[RADDR]	XShiftRMask[RADDR]
ShiftBothMasks[RADDR]	XShiftBothMasks[RADDR]
ShMDLMask[RADDR]	XShMDLMask[RADDR]
ShMDRMask[RADDR]	XShMDRMask[RADDR]
ShMDBothMasks[RADDR]	XShMDBothMask[RADDR]

where RADDR may be any RM address including the stack sources discussed earlier. The difference between Shift... and ShMD... is that the former replaces masked out bits with 0, while the latter replaces masked out bits with MD from the memory. All of these shift-and-mask functions are treated like Alu operations, so the result may be routed into an RM address or T.

Here are some examples:

RTEMP←ShiftLMask	RM address specified elsewhere in the instruction
T←ShiftRMask[RTEMP]	RM address specified in the shift expression
RTEMP←Lsh[RTEMP,3,0];	Use of Lsh macro--masked out bits replaced by zeroes
RTEMP←Lsh[RTEMP,3]	Use of Lsh macro--masked out bits replaced by zeroes
RTEMP←Lsh[T,3];	Use of Lsh macro with source data for shift from T
T←Lsh[T,3,MD];	Use of Lsh with MD replacing masked-out bits
T←Rsh[RTEMP,17];	Right-shift data in RTEMP 17 positions
T←Rcy[RTEMP,T,3];	Right-cycle the 40-bit quantity RTEMP..T by 3 positions
T←Lcy[T,RTEMP,3];	Left-cycle T..RTEMP 3 positions
T←DpF[T,3,10,MD];	Deposit T[15:17] into MD[6:10]
T←LdF[RTEMP,2,12];	Right-justify RTEMP[4:5] and leave result in T

When the shifter is used, the 4-way multiplexor for other A inputs is normally disabled by the hardware. However, if you write another A source clause to the left of the shift-and-mask clause in the instruction statement, then that source (coded by the FF field) will be ORed with the shifter output on A before going through the Alu, e.g.:

A←T, T←ShiftLMask[RTEMP]	'Or' shifter output with data from T

would shift the 40-bit quantity RTEMP..T according to the control in ShC, OR that with T, NOT this in the Alu, clear some of the left-most bits in the result according to the LMask specified in ShC, and finally load this into T. Since the shifter output is complemented on A, the actual data appearing at the masker inputs is [shifter and not T]. Hence, the shifted data is masked once by "not T" and then again by the selected masks.

30. ALU Clauses

The operations performed by the Alu were given in the section on ALUFM. In "not A and not B", for example, "A" and "B" may be, respectively, any A or B sources.

Rule: The safe way to write the Alu phrase is to enclose the A and B sources in "()", e.g., "not (RADDR) and not (T)". However, you may omit the "()" around "ID", "T", "Q" or "MD", if you want to (and assembly is slightly quicker when you omit the "()").

Rule: You must not write "not A and B" as "(not A) and B". In other words, it is illegal to put random "()" in the Alu phrase, even though that may clarify the meaning. If you tried to do this, the assembler would recognize "not A" as an Alu phrase and then give you an error like "PDANDB undefined". The "()" are only legal around the "A" and "B" parts of ALU expressions.

Rule: When used in conjunction with the Alu Lsh 1, Lcy 1, Rsh 1, Rcy 1, BRsh 1, or ARsh 1 functions, the above would be written like "(not (RADDR) and not (T)) Lsh 1"; i.e., the entire Alu operation is enclosed in "()" followed by the "Lsh 1" or whatever.

The last carry out of the Alu can be xor'ed with the carry from the ALUFM memory. This is caused by the XorSavedCarry function, written as a separate clause. Carry20 is also written as a separate clause.

If you have not defined the arithmetic Alu operation with the kind of carry bit you need, then you must explicitly write XorCarry as a separate clause in the instruction. For example,

```
(RTEMP)+T, XorCarry;          *Equivalent to (RTEMP)+T+1
(RTEMP)-T, XorCarry;          *Equivalent to (RTEMP)-T-1
(RTEMP)-T-1, XorCarry;        *Equivalent to (RTEMP)-T
```

The legal destinations for an Alu source expression are:

RADDR←	Any RM destination
Stack←	or any other stack destination--emulator only
T←	
PD←	This no-op destination is necessary when the Alu operation is "A" or "B" (i.e., A straight through or B straight through) and the Alu output is not being loaded by any real destination. It ensures that the source gets routed through the Alu, which might be necessary for an Alu branch condition in the next instruction.

The Input and InputNoPE functions (FF < 100), ALUFMem, ALUFMRW←, Cnt, Pointers, ShC, TIOA&StkP functions (FF > 77) and shifter operations discussed in the last section may be used instead of an Alu operation (These are alternative inputs to the hardware's PD path.). Hence, they can feed RADDR← or T← destinations.

Here are some examples of other Alu clauses:

```
T←((RTEMP)+T) Lsh 1          Use of Alu lshift 1 function
T←T Rsh 1                    Use of Alu rshift 1 function
T←(T) Rsh 1                  "()" are optional around "T"
T←RTEMP                      "()" are optional around a single source
T←(RTEMP) Lcy 1              "()" required around an RM address with anything else
T←((RTEMP)+T) Rcy 1
RTEMP←(RTEMP) BRsh 1         Alu rshift 1 bringing Alu carry into bit 0
```

```
RTEMP←(RTEMP) ARsh 1          Arithmetic rshift 1 preserving sign
T←T xor (377C)
T←ID, Carry20                 Carry20 function is separate clause
T←ALUFMem, Aluf[17]           Save ALUFM[17] in T
T←0H                          Literal reference to then contents of the ALUFM location containing 0
T←(ALUFMRW←RTEMP), Aluf[17]   Saves ALUFM[17] in T, loads it from RTEMP
```

Warning: If you erroneously write an instruction statement that routes the Alu through PD and also routes Input or ALUFMem through PD, the assembler won't give you any error message.

31. Memory References

Memory references are initiated with A clauses as discussed earlier--the assembler does not make any distinction between the hardware's Mar bus and the A bus.

From the viewpoint of what can be encoded in an instruction, it is convenient to distinguish Fetch← and Store← from other references; only these two allow the displacement to be sourced from T or an RM address while FF remains available for use as a constant or a long branch; only these two allow the displacement to be sourced from T, ID, MD, or Q using only FF[0:1], so FF[2:7] remain available for encoding another function. All other references require FF to be used when specifying any source other than an RM address. Since FF[0:1] encode alternate sources for the displacement or alternate references, only the first 100 functions can be used in the same instruction; functions 100 to 377 cannot be encoded.

The Store← and Map← references not only use a displacement on A but also accept data on B. However, if you forget to route data onto B, the assembler won't flag your error.

In the same statement with Store←, you should normally write "DBuf←bsource" to show explicitly that the data on B (bsource) is intended for the Store←; similarly with Map←, you should write "MapBuf←bsource." The "DBuf←" and "MapBuf←" are just for readability, since the hardware will load from B regardless of what you write in the instruction statement (DBuf/MapBuf are the names of the buffer registers in the memory section that get loaded when you do a Store←/Map←).

At t_0 of the entry instruction for an opcode, MemBase is loaded by the IFU with either a MemBX-relative value between 0 and 3 or an absolute number between 34 and 37. Base register 37 is used as the code base by the IFU. The FlipMemBase function loads MemBase with its current value xor 1 and MemBase←small constant loads with any value between 0 and 37.

After a Fetch←, MD may be read in any of the following ways:

```
RTEMP←MD          Load into any RM or STK register
T←MD
adest←MD          Any A destination (i.e., Alu or another memory request)
bdest←MD          Any B destination
Implicit use by ShMD...Mask
```

The time required for a memory reference not confined to the cache (i.e., a cache reference that misses or an io reference) is about 1.7 μs. A Fetch← reference confined to the cache finishes in two cycles, which means that MD can be loaded into RM or T in the next instruction, or onto A or B or used in ShMD...Mask in the second instruction after the Fetch← without being held.

32. Standalone Functions, Block, and Breakpoint

Rule: Standalone clauses should be put *to the right of A clauses* in an instruction statement; the assembler will generate correct output regardless of where the clause appears, but an FF > 77 function to the left of a memory reference clause (which is an error) will not be flagged, if you violate the rule.

The following summarizes standalone functions, each written as a separate clause.

BreakPoint	(Not a function)--causes loading with bad parity in both halves of IM, interpreted as a breakpoint by Midas
Block	(Not a function)--causes the Block bit to be set in the instruction (only legal in non-emulator tasks)
RestoreStkP	FF > 77--restores StkP to the value saved after the last IFU dispatch
XorCarry	FF < 100--complement the carryin from the ALUFM ram
XorSavedCarry	FF < 100--xor carryin from the ALUFM ram with the carryout of the last instruction executed by this task.
Carry20	FF < 100--OR 1 into the carry into Alu[13] (The hardware Alu is composed of four four-bit IC's; this function OR's 1 into the carry out of the low-order IC.)
FreezeBC	FF < 100--freeze task-specific branch conditions
TIsID	FF < 100
RIsID	FF < 100
FlipMemBase	FF < 100--MemBase←MemBase xor 1
Multiply	FF < 100
TaskingOn	FF > 77--enabling tasking is delayed until the next instruction for the current task is executed
TaskingOff	FF > 77
UseDMD	FF > 77--execute the manifold operation for the current DMux address
Divide	FF > 77
CDivide	FF > 77
IFUReset	FF > 77--reset the IFU
LoadMCR[A,B]	FF > 77--Routes first arg onto A, second onto B; loads MCR from the appropriate bits off of each bus.
LoadTestSyndrome	FF > 77--loads TestSyndrome from DBuf (used after a Store←).
Reschedule	FF > 77--cause the second IFUJump to enter a trap vector
RescheduleNow	FF > 77--cause the next IFUJump to enter a trap vector
NoReschedule	FF > 77--turns off the ReSchedule condition
IFUTick	FF > 77--generates the next clock for the IFU testing stuff.
AckJunkTW	FF > 77
TIOA[device]	FF > 77--loads TIOA[5:7] from FF[5:7].
Wakeup[taskx]	FF = 360 to 377--issue a wakeup to taskx, previously defined by TASKN[taskx,n].
Notify[n]	same as notify, but n is an integer 0 to 17; the Wakeup form should ordinarily be used (exception: control section diagnostics)
Cnt-1	FF < 100--uses the Cnt=0&-1 branch condition function for its side-effect without imposing any placement constraint on the successor. The successor must be forced to lie at an odd location (e.g., by using DispTable).

* The TGetsMD, ModStkPBeforeW, and ReadMAP functions are never written explicitly by programmers; the ReadMAP function is imposed automatically by the RMap← reference, ModStkPBeforeW by the appropriate stack operations, and TGetsMD when both T and an RM address are loaded from MD.

33. Branching

This section discusses branch clauses in instruction statements, declarations which affect branching, and dispatch clauses.

Micro assembles instructions for an imaginary machine identical to Dorado but with additional fields assembled for its postprocessor. The imaginary machine is characterized by full-size 14-bit branch addresses in instructions and 14-bit program addresses in IFUM. MicroD places instructions and transforms the .Dib file for the imaginary machine into a .Mb file for Dorado. Algorithms used by MicroD are described in the appendix.

33.1. What the Branch Hardware Does

Dorado implements three kinds of control transfers determined by the value in the JCN field of an instruction: jumps, returns, and IFU jumps; jumps may be "local," "global," "long," or "conditional." The processor always branches or returns--the hardware contains no concept of not-branching or of falling through to the next instruction.

Returns and IFU jumps load the Link register unconditionally; jumps load Link iff the target address is 0 mod 20. For all of these, the value loaded into Link, if any, is ((.+1) & 77) + (. & 7700); i.e., Link is loaded with caller's address +1 (carries not propagating beyond the low six bits).

For reasons that will be apparent, it is convenient to view the microstore as composed of 100 pages of 100 words each. Local jumps transfer control to any location on the current page, global jumps to location 0 on any page, and long jumps to any location in the microstore (using the FF field to extend JCN).

An explicit branch clause may be unconditional or conditional. When conditional, the branch address is executed next, if the condition is false, or the branch address OR 1, if the condition is true. The decision to load Link (i.e., Call or Goto) is based upon the false branch address.

Branch conditions may be encoded as functions (FF < 100), in the JCN field, or both (when two BC's are specified, the true path takes if either condition is true). When encoded in JCN, the false branch address must be at locations 4, 6, 10, ... , or 36 in the current page. When the BC is coded only in FF, the false branch address can be at any even location in the same page or at a global location.

The locations which are multiples of 4 are IFU targets. Namely, it is possible to origin an IFU entry vector at these points.

Dispatches allow an instruction to modify the branch address of the next instruction for the same task. The address modification consists of "OR"ing bits computed by the dispatch with the branch address computed in cycle i+1.

33.2. Branch Clauses

The assembly language has IFUJump and Return constructs analogous to the underlying hardware operations. However, the complications surrounding jumps are, for the most part, concealed from the programmer.

If the programmer doesn't specify any branch clause in the instruction statement, the assembler will fabricate a jump to the next instruction inline. Several constructs of the form:

Goto[ba, bc1, bc2] -or-
DblGoto[batrue,bafalse,bc1,bc2]

are defined (see below), where both branch conditions are optional in the "Goto" form, and the second branch condition is optional in the "DblGoto" form. "Goto" indicates that the Link register must not be modified and "Call" that Link must loaded with the address of the next instruction inline; "Branch" is deliberately indefinite about whether a "Goto" or "Call" is done.

Branch addresses for these may be either instruction tags or one of the following special symbols: .-3 .-2 .-1 . .+1 .+2 .+3, where "." refers to the current instruction and the others are relative to this inline. [It is obviously possible to define .-4, .+4, .-5, etc., but my feeling is that it is bad style to jump further than +/- 3 without using a tag. If anyone finds this inconvenient, please let me know.]

Branch condition arguments may be either "regular" (one of the 10 in the hardware manual) or "complementary" (complements of the 10 in the hardware manual). The branch conditions are named as follows:

Regular	*Complementary*	
Alu=0	Alu#0	
Alu<0	Alu>=0	
Cnt=0&-1	Cnt#0&-1	Also decrements Cnt after testing for the branch
R<0	R>=0	
R odd	R even	
Carry'	Carry	
Reschedule	Reschedule'	(Emulator task only)
IOAtten'	IOAtten	(io tasks only--same encoding as Reschedule)
Overflow	Overflow'	(FF encoding only)
Alu<=0	Alu>0	Combination of Alu=0 in FF and Alu<0 in JCN

When complementary branch conditions are used, the assembler simply reverses the order of the branch tags. Hence, DblGoto[T1,T2,com C1, com C2] = DblGoto[T2,T1,C1,C2]. This is provided as a programming convenience.

Warning: If two branch conditions appear in a statement, they must be *both regular* or *both complementary.* When two regular branch conditions are used, the true path takes if either is true. However, when two complementary branch conditions are used, the true path takes only when both are true. Don't get confused by this.

The "Top Level" and "Subroutine" declarations control assembler error checking. In Top Level mode, calls and dispatches are legal, returns are illegal, and branches may have target addresses that lie on either call or goto locations. In Subroutine mode, calls and dispatches are illegal, returns are legal, and branch targets are required to be at goto locations.

The assembler constructs are given below, where "<>" denote optional args; C1 and C2 either two hardware branch conditions or complements of two hardware branch conditions:

Return[<C1>]	To Link and smashes Link--illegal in Top Level mode. A branch condition (uses FF < 100) makes sense only when the caller has skip/noskip return points created by an SCall, DblSCall, or SCoReturn.
CoReturn[<C1>]	Like Return but Link←.+1 and next instruction inline placed at .+1.
DblBranch[T1,T2,C1<,C2>]	To T1 if C1 or C2 true, else to T2. T1 will be placed at T2 OR 1; placement of T2 is limited to goto locations in Subroutine mode, else unconstrained.
DblGoto[T1,T2,C1<,C2>]	like DblBranch[T1,T2,C1<,C2>] constraining T2 placement to goto locations.
DblCall[T1,T2,C1<,C2>]	like DblBranch[T1,T2,C1<,C2>], constraining next instruction inline to be at .+1, and limiting T2 to call locations. Illegal in Subroutine mode.
Branch[T1<,C1<,C2>>]	To T1 if C1 or C2 is true or if both branch conditions are omitted; otherwise to next instruction inline. When conditional, T1 will be placed at .+1 OR 1. In Subroutine mode, either .+1 (conditional) or T1 (unconditional) constrained to goto locations.
Goto[T1<,C1<,C2>>]	like Branch[T1<,C1<,C2>>] constrains either T1 (unconditional) or next instruction inline (conditional) to goto locations.
Call[T1<,C1<,C2>>]	like Branch[T1<,C1<,C2>>]; illegal in Subroutine mode; complementary BC's illegal; constrains next instruction inline to be at .+1; constrains placement of either T1 (unconditional) or next instruction inline (conditional) to call locations. Discussed below.
IFUJump[i<,C1>]	Dispatch to the i'th entry vector of the next opcode (An error is flagged if i >= the entry vector size specified by the last InsSet declaration). A branch condition would only be used if a conditional exit programming convention is followed, as discussed in the hardware manual; complementary BC's are illegal.
DblSCall[T1,T2,C1<,C2>]	= DblCall[...] and forces odd placement of the instruction and placement of the next two instructions inline at .+1 and .+2 so that the subroutine can do skip/noskip Return by using a branch condition.
SCall[T1<,C1<,C2>>]	= Call[T1<,C1<,C2>>] and forces odd placement of the instruction and placement of the next two instructions inline at .+1 and .+2 so that the subroutine being called can do skip/noskip Return by using a branch condition. Complementary branch conditions are illegal.
SCoReturn[<C1>]	= CoReturn but forces placement at an odd location and placement of the next two instructions inline at .+1 and .+2. A complementary branch condition is illegal and use of any branch condition only makes sense when the caller entered by means of SCall or SCoReturn.
--	No branch clause = Branch[.+1]

A Branch while Top Level is in force imposes less placement constraints on the target instruction(s) because it permits either the Call or Goto locations to be used.

DblBranch, DblGoto, and DblCall are expected to be less frequent than Branch, Goto, and Call

because programmers ordinarily think of branching or falling-through rather than branching to one of a pair of instructions.

For an unconditional top level Branch, MicroD outputs a long call or long goto if the FF field is unused, and imposes no constraint on the placement of either the instruction or its target. If FF is used, then the branch target will have to be in one of the 100 same-page or 100 global branch locations reachable by a JCN branch.

An unconditional Call is assembled as a long call if FF is unused. In this case, the branch address may be any call location. If FF is used, then the target address has to be one of the 3 call locations in the same page or one of the 100 global call locations. The next instruction inline is placed at .+1 within the page.

A conditional Call is just barely possible. It requires the next instruction inline to be simultaneously at the true branch address xor 1 and at the address of the caller +1. Since the true branch address must be at a location with four low bits equal 0001, these conditions are only met at three positions within a page (e.g., the Call, false target, and true target may be placed at 17, 20, and 21; at 37, 40, and 41; or at 57, 60, and 61 in the page). This implies that complementary BC's are illegal with Call, nor can you encode two consecutive instructions each containing a conditional Call, nor can you have more than one conditional call to a single subroutine.

It is also impossible to have a Call in an instruction which is the false target of a conditional Branch because the return of the Call would be to the true target of the previous conditional branch.

An unconditional Return branches to Link, normally containing the address of the caller +1. There is no placement constraint on an instruction containing a Return.

A conditional Return goes to Link if the branch condition is false or to Link or'ed with 1 if the condition is true. This allows a skip/noskip return to the caller, which only makes sense if the caller imposed the necessary placement constraints on his two successor instructions by using an SCall, DblSCall, or SCoReturn.

An unconditional Goto is assembled as a long goto if FF is unused. If FF is used, then the branch address has to be one of the 74 goto locations in the same page.

Suggestion: In programs that nearly fill the control store, or in smaller programs that have large instruction clusters, you should carefully use Branch rather than Goto in Top Level mode to give MicroD greater freedom in placing instructions; in Subroutine mode, you may use either Branch or Goto, but I suggest that you pick a consistent convention: either always use Goto or always Branch.

33.3. Dispatch Clauses

The assembly language defines the following dispatch clauses:

BDispatch←B	10-way dispatch on B[15:17]
BigBDispatch←B	400-way dispatch on B[10:17]

Multiply is also a dispatch. Dispatches OR bits into the branch address computed by the next instruction for the same task. This means that the programmer must impose the necessary constraints on the target instructions in the dispatch table himself--MicroD won't do it for him. This is done using placement declarations as discussed in the next section.

34. Placement Declarations

When an IFU location is assembled, the address to which it dispatches is automatically marked as an IFU entry--no explicit declaration is required when assembling that instruction. In other words:

```
IFUReg[n,TAG,..otherjunk..];    *Opcode n PUSH2
```

automatically makes TAG an IFU entry. When you specified the opcode set with InsSet[i,n], you declared that there would be n entries for each IFU dispatch in the instruction set. You must put the n entries in sequence in the source, with the first at TAG and the others after that.

Global entries are declared by a "Global" clause in a statement, e.g.:

```
DONEXT: Return, T←ID, Global;
```

Global declarations cause placement at one of the 100 global call locations in the microstore. Global placement must be explicitly declared--MicroD handles most placement automatically, but it does not automatically assign globals.

Placement requirements for instructions in a dispatch table (i.e., of instuctions which are the targets of BDispatch←B, BigBDispatch←B, etc., or of a computed Return) may be declared either through using "At" on every instruction in the table (see below), or, under suitable conditions, using the DispTable macro.

DispTable[LENGTH,MASK,VALUE] appearing as a clause in an instruction statement causes that statement to begin a group of LENGTH consecutively-placed statements, 1 <= LENGTH <= 20. The first statement is placed so that [address and MASK] = VALUE. MASK defaults to [next power of 2 >= LENGTH] - 1, and VALUE defaults to 0. Note that LENGTH+VALUE must be <= 20.

A 10-way BDispatch might be written as follows:

```
            BDispatch←RTEMP;
            ... , Goto[SWITCH];

SWITCH:     ... , DispTable[10];          *B[15:17] = 0
            ... ;                         *B[15:17] = 1
            ...
            ... ;                         *B[15:17]=7
```

where the three instructions in the dispatch need not be consecutive in the assembly source.

An instruction containing the clause "At[N]" will be forced by the assembler to appear at absolute location N in the microstore. "At[N1,N2]" in an instruction is equivalent to At[Add[N1,N2]]. "At" will be necessary for the special IFU trap locations and for instructions in dispatch tables that do not meet the constraints of the DispTable macro above. The currently reserved locations in IM are given in the hardware manual.

Warning: In addition, because instruction addressess are unknown during assembly, it is illegal to create parameters, constants, or RM data referring in any way to absolute locations. To do this, you must manually locate each affected instruction with "At" and do arithmetic on integers with the same values as the instruction locations. This will probably be required for the startup instructions of all tasks, which must be loaded into Link for a LdTPC←.

Normally, MicroD automatically chooses the page assignment for an instruction not constrained by "At"--the TITLE statement enables automatic page assignment. However, the following macros are available for constraining the page on which an instruction is placed:

OnPage[n];	as a separate statement will constrain MicroD to place subsequently assembled instructions on locations on page n (i.e., in the range n*100 through n*100 + 77).
AutoPage[n];	undoes OnPage and allows MicroD freedom to place anywhere.

OnPage may be useful in dealing with microcode overlays, as discussed later.

In addition to the above placement macros, there are two macros for "reserving" and "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.

35. Microcode Overlays

The barest minimum provisions are made for microcode overlays. Because we cannot handle dynamic relocation, non-conflicting placements must be made for the resident system and all overlays that may be used with it. There are several ways that safe placement may be accomplished:

First, MicroD can write an xxOccupied.Mc file in which all locations used in the resident system loadup are indicated by IMReserve statements. The xxOccupied.Mc file can be loaded with an overlay to ensure safe placement by MicroD of code in the overlay. A disadvantage of this method is that whenever the system microcode is modified, all overlays using this method must be regenerated.

Next, the resident microcode can itself reserve regions of the microstore with IMReserve so that overlays confining themselves to the reserved area need not be regenerated when a new system is released.

Thirdly, throwaway initialization code may be manually placed by means of OnPage or At

declarations that totally fill some pages of the microstore, and these pages are available to overlays after execution. If this method is used in conjunction with one of the first two methods, IMUnreserve declarations can be used to free up the pages filled with the throwaway code.

Finally, particular instructions in the resident system may be overwritten by particular instructions in an overlay; these must be manually placed with "At" declarations in both the system microcode and the overlay.

36. Instruction Memory Read-Write

The hardware provides an efficient method for loading the instruction memory (which might be common if microcode overlays are used) and a painful method of reading the instruction memory (unlikely to be dynamically frequent). Each instruction that reads or writes IM takes three cycles.

IM read/write is encoded in the JCN and RSTK fields of the instruction, so you may not program any control clause in the same instruction. The instruction after the one doing the read or write must be at .+1 within the page, and the assembler automatically imposes this constraint, so you do not have to use "At[N]". Tasking must be off.

For loading IM, the address to be written is first loaded into Link←, then the left or right half is written from a B source. RSTK[1:3] control left/right half, good/bad parity, and the 17th data bit (RSTK.0 or Block), so there is little flexibility in selecting an RM address for use with the write--you probably should source the data from T, Q, or Cnt. Link is smashed with .+1 after the write, so it has to be reloaded before writing the other half of IM. The following sequence is an example:

```
%Have 16 bits of left-half data at STK[StkP], RStk.0 and JCN.7 value in the sign bit and low bit of
STK[StkP-1], respectively, and 16 bits of right-half data in STK[StkP-2]. Write this data into the IM address
in Q with good parity.
%
                T←Stack&-1, Link←Q;
                TaskingOff, Stack&-1, Branch[R0TRUE,R<0];
                IMLHR0'POK←T;
                T←Stack&+1, Link←Q;
WRH:            Stack&-2, Branch[BTRUE,R ODD];
                IMRHB'POK←T;
IMWFIN:         TaskingOn;
                ...

R0TRUE:         IMLHR0POK←T;
                T←Stack&+1, Link←Q, Branch[WRH];

BTRUE:          IMRHBPOK←T;
                T←Stack&-2, Branch[IMWFIN];
```

The IM write instructions take three cycles each but are otherwise indistinguishable from ordinary instructions. This means that there are no strange restrictions on other actions carried out in the same instruction.

IM data are read nine bits at-a-time, with the address again coming from Link and the byte number from RStk[2:3]. The data arrangement is shown in a figure of the hardware manual and is read back by the B←Link function in the cycle immediately after the read.

```
%Have IM address in RM location RTempl. Read the left-half of IM to RTemp3 right-half to RTemp2
using RTemp0 as temp storage. Assume RBasc points at correct region of RM at call. The extra bits of IM
(P.16, P.17, RStk.0, and Block) are flushed. RTemp0 to RTemp3 are RM locations whose low bits are 0, 1, 2,
and 3, respectively. RRetn is another RM location in the same region as RTemp0 to RTemp3
%
```

```
Subroutine;

RDIMD:      RRetn←Link;                    *Save return to caller of subroutine

Top Level;

            Link←RTemp1;
            TaskingOff;
            ReadIM[1];
            RTemp0←Link;                   *RTemp0[7:17]←byte 1
            Link←RTemp1;
            ReadIM[0], T←Lsh[RTemp0,10];   *T[0:7]←byte 1 flushing parity bit
            RTemp3←Link;                   *RTemp3[7:17]←byte 0
            Link←RTemp1;
*RTemp3←byte0,,byte 1 (flushing RSTK.0 bit)
            ReadIM[3], RTemp3←Lcy[RTemp3,T,10];
            RTemp2←Link;                   *RTemp2[7:17]←byte 3
            Link←RTemp1;
            ReadIM[2], T←Lsh[RTemp2,10];   *T[0:7]←byte 3, flushing parity bit
            RTemp2←Link;                   *RTemp2[7:17]←byte 2
            TaskingOn;
            Link←RRetn;                    *Restore return address to LINK

Subroutine;

*RTemp2←byte2,,byte3 flushing JCN.7
            Return, RTemp2←Lcy[T,RTemp2,10];
```

37. Reading and Loading Task PC's

The method by which task PC's (TPC) are loaded for specific tasks, like the IM read/write, is a funny format RETURN in the JCN field of the instruction. Consequently, control clauses are verboten in the instructions that do this, and, again, the successor to the instruction that does the TPC read/load must be at .+1 within the page (automatically constrained by the assembler). Tasking must be off.

Data for TPC goes to/from LINK, task number from B[14:17].

Two macros are defined for reading and writing TPC from Link (These are B destinations.):

```
LdTPC←      Loads TPC for task from LINK, task number from B[14:17]
RdTPC←      Reads TPC for task into LINK, task number from B[14:17]
```

These macros fill in JCN appropriately. Normally, the task number on B will be a constant produced by mentioning the name of a task you have previously defined with TASKN, e.g.:

```
TASKN[DSP,14];                     *Define DSP as the display task

            Link←DSPST;            *Load Link with starting address
            TaskingOff;            *Require tasking off during LdTPC←
            LdTPC←DSP;             *is then equivalent to LdTPC←14C;
            TaskingOn;
```

38. Divide and Multiply

The Dorado hardware defines special standalone functions Multiply, Divide, and CDivide which allow multiplication to be carried out in a one-cycle loop and division in a two cycle loop.

The hardware actions caused by these functions are as follows:

```
Multiply:
        PD←ALUCarry,.ALU/2
        Q←ALU[17]..Q/2
        Next branch address← whatever it is 'or' 2 if Q[16] is 1

Divide:
        PD←2*ALU..Q[00]
        Q←2*Q..ALUCarry'
```

The following examples show how these are used:

```
%At entry:
RTemp/ multiplier (20-bit unsigned)
T/ multiplicand (20-bit unsigned)
At exit:
RTemp..Q/ 40-bit result

The first step is outside the inner loop. It moves the multiplier into Q and tests Q[17]. The second step, also
outside the inner loop, tests Q[16] with the Multiply function and initializes the result (computed in RTemp)
to 0. It enters the inner loop at the "add" or "no-add" position based upon step 1. The Multiply function
also causes a dispatch, so the inner loop is entered with the "add" or "no add" decision already made for the
two low bits of multiplier, 0 in Q[0], and untested multiplier bits in Q[1:16]. The inner loop does 16 useful
Multiply steps, 1 useless step testing the 0 that started out in Q[0], and then the exit instruction does a final
Multiply testing the low bit of the result, leaving the result in RTemp[0:17]..Q[0:17]. Instruction placement is
critical. The two exit instructions have to be located so that 'or'ing 2 into their locations doesn't change the
location. The inner loop instructions have to be located so that the first is a fast-goto location and a multiple
of 4. This can only be satisfied if the low four bits of address are 4, 10, or 14.
%

MULT:       Q←RTemp;
            Goto[.+2,R Even], B←RTemp, Cnt←16S;
            Goto[M1], RTemp←T-T, Multiply;
            Goto[M0], RTemp←T-T, Multiply;

DTABLE[MulX,0,7770]; *Dispatch table origin 0 mod 10 (0 and 1 unused).
MXIT0:      Dat[MulX,2];
            Return, Dat[MulX,3];

*Here after Q[16] was 0 (no add)
M0:         DblGoto[M0,M0E,Cnt#0&-1], RTemp←RTemp, Multiply, Dat[MulX,4];
M0E:        Goto[MXIT0], RTemp←RTemp, Multiply, Dat[MulX,5];
*Here after Q[16] was 1 (add)
M1:         DblGoto[M0,M0E,Cnt#0&-1], RTemp←(RTemp)+T, Multiply, Dat[MulX,6];
            Goto[MXIT0], RTemp←(RTemp)+T, Multiply, Dat[MulX,7];

%At entry:
RTemp/ most significant 20 bits of 40-bit unsigned dividend
Q/ least significant part of dividend
T/ divisor (20-bit unsigned)
At exit:
Q/ quotient (20-bit unsigned)
RTemp/ remainder (20-bit unsigned)
```

Each divide step shifts Q[0] from the low part of the dividend into the high part of the dividend while doing the Divide function and testing for exit. The second instruction chooses between add or subtract, based upon whether or not the last add/subtract "succeeded".

The duplicated instructions are required because they are part of branch condition pairs.
%

```
DIV:        (RTemp)-T, Cnt←17S;
            Goto[DivOK,Carry'];              *Test whether the divide is possible
            Return, RTemp←T-T;               *Return 0 indicating impossible

DivOK:      PD←T;
            Goto[BigDiv,Alu<0];              *Branch for the hard case
            DblGoto[DvExit,DvTest,Cnt=0&-1], RTemp←(RTemp)-T, Divide;

*Easy case--divisor bit 0 is 0
DvTest:     DblGoto[Dv0,Dv1,Carry'];
Dv0:        DblGoto[DvExit,DvTest,Cnt=0&-1], RTemp←(RTemp)+T, Divide;
Dv1:        DblGoto[DvExit,DvTest,Cnt=0&-1], RTemp←(RTemp)-T, Divide;

DvExit:     Goto[DvXit0,Carry'];
DvXit1:     (RTemp)-T, Divide;
            DblGoto[DvXitFix,DvXitOK,Alu<0], RTemp←(RTemp)-T;
DvXit0:     (RTemp)+T, Divide;
            DblGoto[DvXitFix,DvXitOK,Alu<0], RTemp←(RTemp)+T;
*Fix for having subtracted too much in last step
DvXitFix:
            Return, RTemp←(RTemp)+T;         *Adjust remainder
DvXitOK:
            Return;

*Hard case--bit 0 of divisor is 1
BigDiv:     DblGoto[BigDvd,BDvLp1,Alu<0], RTemp←(RTemp)-T, Divide;
BDL2:       DblGoto[BigDvd,BDvLp1,Alu<0], RTemp←(RTemp)-T, Divide;
BDvLp:      DblGoto[BigDvd,BDvLp1,Alu<0], RTemp←(RTemp)-T, Divide;
BDvLp1:     RTemp←(RTemp)+T;
            Goto[BDvLp,Cnt#0&-1], RTemp←(RTemp)+T;
BDvXit:     (RTemp)-T, Divide;
BDvXit0:
            DblGoto[DvXitFix,DvXitOK,Carry'], RTemp←(RTemp)-T;
BRDX:       Goto[BDvXit0],(RTemp)-T,Divide;

*Big partial dividend, check for carry
BigDvd:     Goto[BigDvH,Carry'];
            DblGoto[BDvXit,BDvLp,Cnt=0&-1], PD←RTemp;

*Most complicated case--big R and no carry
BigDvH:     RTemp←(RTemp)+T, Goto[.+2,Cnt#0&-1];   *R+2T-T
BRDX1:      PD←(A←RTemp), CDivide, Return;

*Force carry to 0--1 bit in Q
BigRLP:     Goto[BrDvXit,Cnt=0&-1], RTemp←(A←RTemp), CDivide;
            Goto[BDL2,Alu>=0], PD←RTemp;
            Goto[BigRLP], RTemp←(RTemp)-T;

*Exit for the hard cases
BrDvXit:
            Goto[BRDX,Alu>=0];
            RTemp←(RTemp)-T, Goto[BRDX1];
```

39. Programming Tips and Examples

Experience suggests that it is necessary to worry about availability of FF for use in long branches. For this reason you should try to leave the FF field free for a long branch when this doesn't add extra instructions.

Another issue to be concerned with is usage of Alu operations. Preliminary versions of the Mesa and Alto emulators have suggested that the 15 operations *'ed in the "Assembling for ALUFM" section will be required. At the moment, A0 is also defined. However, try to avoid using A0 and other doubtful operations unless you really need them. In those places where A0 would be the simplest, try to use A-B with the same source for both A and B instead. Similarly, try to use A-B-1 rather than A1 and XOR rather than EQV. If you need an extra operation to save time or space, go ahead and use it, but don't do this needlessly in case we decide to change the selection of operations later.

Also, BitBlt uses two ALUFM locations as variable operations but should restore these to standard values before exiting to the next opcode. If these two operations are restored, the emulator will have 17 Alu operations available, though other tasks will have only 15 available. The comments in the D1Alu.Mc file show how to define the two "emulator only" operations so that the assembler will flag an error when one of these is used from an io task.

It is also important to take full advantage of the various numbers which can be delivered by ←ID when programming emulators. These are the operand, argument bytes alpha and beta, and then instruction length endlessly. For example, on Mesa DIVIDE, it was possible to use length=1 to negate the quotient and remainder with (ID)-T-1 (etc.). Also, the same instruction can be used for NOT and NEG opcodes and the same exit instruction for ADD and SUB. Try to exploit the various options afforded by this.

The examples below will be augmented as more code is available.

```
*Mesa Read-Field opcode
RDFLD:      IFetch←Stack, TIsID;              *Calc. pointer as MDS+α+Stack
            Stack←MD, RF←ID;                  *IFU supplies β
            IFUJump[0], Stack←ShiftLMask;     *Shift and mask, Stack←result

*Opcode 23, type = regular, length = 3 bytes, MemBase←MDS, RBase←0, no operand
IFUReg[23,3,MDS,0,RDFLD,17,0,0];

*Mesa Write-Field opcode
WRTFLD:     T←(IFetch←Stack&-1)+T, TIsID;     *Calc pointer and save in T
            WF←ID, RTemp←T;                   *T←field descriptor
            T←ShMDBothMasks[Stack&-1];        *Deposit Stack in MD and pop
            IFUJump[0], Store←RTemp, DBuf←T;  *Store result, exit

*Opcode 24, type regular, length = 3 bytes, MemBase←MDS, RBase←0, no operand
IFUREG[24,3,MDS,0,WRTFLD,17,0,0];

*Random number generator using 8 words of RM as storage for the "state" of
*the generator.
```

```
RMRegion[Other];
RV[RGState,0];          RV[Rand,0];

RMRegion[Random];
RV[R0,134134];    RV[R1,054206];
RV[R2,036711];    RV[R3,103625];
RV[R4,117253];    RV[R5,154737];
RV[R6,041344];    RV[R7,006712];

SET[X,20];        *A "call" location
RGen:             Goto[RGen1], T←R0, RBase←RBase[Rand], At[X];
                  Goto[RGen1], T←R1, RBase←RBase[Rand], At[X,1];
                  Goto[RGen1], T←R2, RBase←RBase[Rand], At[X,2];
                  Goto[RGen1], T←R3, RBase←RBase[Rand], At[X,3];
                  Goto[RGen1], T←R4, RBase←RBase[Rand], At[X,4];
                  Goto[RGen1], T←R5, RBase←RBase[Rand], At[X,5];
                  Goto[RGen1], T←R6, RBase←RBase[Rand], At[X,6];
                  Goto[RGen1], T←R7, RBase←RBase[Rand], At[X,7];

RGEN1:            Return, T←Rand←(Rand)+T;

*The calls are as follows:

                  RGState←(RGState)+1, BDispatch←RGState;
                  Call[RGEN], RBase←RBase[Random];     *Return random number in T

*Test-and-set in one instruction for use by different tasks that control
*each other.  Sign bit of RM register RFlag is the lock.
                  RFlag←(RFlag) or (100000C), Branch[AlreadyLocked,R<0];

*Alternative lock procedure:  store -1 in RFlag when unlocked; then:
                  RFlag←(RFlag)+1, Branch[AlreadyLocked,R>=0];
```

Appendix 1. MicroD

MicroD transforms .Dib files produced by Micro into .Mb files. Since instruction placement is fairly tedious, the display shows a progress message, so you can monitor progress of the load. The sequence of progress messages is as follows:

```
Loading File1...
Loading File2...
   ...
Loading FileN...
N instructions, M words for symbols
Linking...
Building allocation lists...
Assigning locations...
Reloading binaries...
Checking assignment...
Writing .MB...
N words free
```

Error messages may appear at any time. Some of these immediately abort the load, but most errors do not abort until the end of the current progress step. In other words, errors during "Linking...", will usually abort at the end of this loading phase; errors during "Building allocation lists...", usually abort at the end of this phase, etc.

After "Building allocation lists..." has completed, all bugs will have been detected except conflicting absolute addresses (two AT's at same location) and various overflows (too many globals, too many IFU entries, too many instructions on a page, etc.).

The data printout for IFUM and RM is in two columns. For RM the address symbol(s) associated with a location are printed to the right of the data. For IFUM, the IM target symbol is printed to the right of the data. For IM, the printout is like the following:

```
345   457    23456   23457   FOO
346   601   233333  144444
```

meaning that the 345th instruction assembled by Micro with label "FOO" was placed at absolute location 457 and the two 16-bit numbers are the octal contents of the instruction.

The error messages produced by MicroD contain the symbolic address of the instruction at which the error was detected, when relevant.

Micro Output for the Imaginary Machine

Micro outputs stuff for IM, RM, IFUM, ALUFM, STK, and fake memories called BR, BRX, DEVICE, TASKN, VERSION, RVREL, IMLOCK, and IMMASK.

MicroD transforms only IM and IFUM data. Addresses in all memories and data in all memories except IM and IFUM pass through MicroD to the .Mb output file unchanged--this excludes data and addresses for VERSION, RVREL, IMLOCK, and IMMASK, which are fake memories whose contents and address symbols are consumed and flushed by MicroD.

Data are output for IM, IFUM, ALUFM, RM, and STK in the form expected by MicroD and Midas, as given below. BR, BRX, DEVICE, and TASKN have address symbols useful when debugging with Midas but no data are output for these memories. In summary, we have:

IM	Transformed by MicroD--see below
IFUM	Transformed by MicroD--see below
RM	20-bits per word
STK	20-bits per word (Most programs don't assemble anything for this memory, but provision is made for this.)
ALUFM	10-bits per word with 0 and 3:7 containing the 6 bits loaded into the ALUFM ram
BR	base register address symbols for debugging
BRX	MemBX-relative base register address symbols for debugging
DEVICE	io device address symbols for debugging
TASKN	task address symbols for debugging
VERSION	1-word memory defining the machine as Dorado for MicroD.
IMLOCK	10000-word x 1-bit memory; a 1 in an IMLOCK word prevents MicroD from placing any instruction in the corresponding location of the microstore.
IMMASK	10000-word x 24-bit memory defining dispatch table length and allowable placement of first word.

IM and IFUM parity bits expected by the hardware are computed by neither Micro nor MicroD; Midas computes these at the time it does the load.

Micro outputs a modified form of Dorado IFUM words, as follows:

PA	1 bit	Packed-α bit
NEnt	2 bits	Number of instructions in target sequence
	1 bit	Unused
IFAD	14 bits	Imaginary address of target instruction
Sign	1 bit	
	3 bits	Unused (parity bits filled in by Midas)
Length'	2 bits	opcode length (1, 2, or 3 bytes)
RBaseB'	1 bit	RBase initialization
MemB	3 bits	MemBase initialization
Pause'	1 bit	
Jump'	1 bit	
N	4 bits	

All of the bits are located in positions compatible with IFUMRH←/IFUMLH← except for IFAD, which has two extra bits. These extra bits are positioned to avoid conflict with real IFUMRH←/IFUMLH← data bits.

MicroD will transform IFAD into a real address and output the proper 12 bits in 0 to 11 of the first word, as well as zeroing the extraneous bits.

Micro outputs for each instruction assembled the 42-bit (+2 parity bits) instruction and four extra words of stuff needed by MicroD as follows:

Dorado instr.	42 bits	Complete except for branch address stuff
P016	1 bit	Load bad parity into IM[0:20]
P2141	1 bit	Load bad parity into IM[21:41]
	14 bits	unused
	1 bit	unused
W0@	1 bit	Place at location W0
Glb@	1 bit	Place at a global location
OnPg@	1 bit	Place on the page specified in W0
W0	14 bits	Location for placement if W0@ or OnPg@ = 1
Returns	1 bit	This instruction does a Return, CoReturn, or IFUJump (or IM or TPC read/write)
Calls	1 bit	This instruction does a Call or CoReturn
JBC	1 bit	This instruction has a branch condition in JCN
UsesFF	1 bit	FF field unavailable for long goto or long call
W1	14 bits	Imaginary address of unconditional or false branch (7777 defaults this to .+1)
Branches	1 bit	This instruction does a Branch
Goes	1 bit	This instruction does a Goto
Emul	1 bit	Print as emulator instruction
IsCond	1 bit	This instr has a branch condition (i.e., W2 at W1 OR 1)
W2	14 bits	Imaginary true address of conditional branch (7777 defaults this to .+1)

W1 and W2 may receive automatic Micro fixups if they are forward references.

Micro finishes assembly for all bits of the instruction except those referring to instruction locations. In other words, the only job of MicroD is assigning absolute locations for the instructions and storing appropriate stuff in the JCN fields (and for long calls, in the FF fields) of the instructions and in the address fields of IFU words.

For conditional branches, the branch condition(s) are already in FF or in JCN, so MicroD does not fix up those parts of the instruction. For Return, CoReturn, IFUJump, IM read/write, and TPC read/write, JCN is also complete.

A more precise meaning for some of these bits is as follows:

IsCond	The instruction at imaginary address W2 must be placed at the absolute location assigned to W1 xor 1.
Returns	JCN has been completely assembled by Micro; W1 and W2 are irrelevant.
Calls	The next instruction in sequence must be at .+1 within the same page, and, unless Returns is also 1, the instruction W1 must be placed at a call location in the microstore.

Instruction Placement

The discussion here describes the original design of MicroD by E. Fiala. The actual MicroD, designed and implemented by L. Deutsch, differs from this description in a number of ways. There is presently no description of the existing program.

The "Load" pass of MicroD loads the .Dib file output by Micro into simulated memories and executes fixups. After loading, all addresses and all data not needed during placement computations are flushed; after placement computation is finished, the .Dib binaries are reread, modified with the placement information and output on the .MB output file.

After loading, several passes are made over IM data as described below. During the "Link" pass simulated memory for an instruction is viewed as follows:

AlcPtr	20 bits	Points at alist header (now 0)
Link	20 bits	Pointer to next alist item (now 0)
	4 bits	tail of Dorado instruction
	14 bits	Unused
	1 bit	Unused
Place	3 bits	0 = W0 is the absolute address of this instruction 1 = Place at a global location 2 = Place at a global and place W0 at . xor 1 3 = IFU entry 4 = Place at even location and place W0 at . xor 1 5 = IFU entry and place W0 at . xor 1 6 = Place at odd location and place W0 at . xor 1 7 = None of the above
W0	14 bits	Absolute addr of this instr if Place = 0 Imaginary addr of instr at . xor 1 if Place indicates it
Returns	1 bit	JCN field fully assembled; ignore W1 and W2

Calls	1 bit	Place next instr. at (.+1 & 77)+(. & 7700); require W1 to lie at a call location unless Returns is 1
JBC	1 bit	Place W1 and W2 at a reachable JCN branch condition target
UsesFF	1 bit	FF field unavailable for long goto or long call
W1	14 bits	Imaginary address of branch from this instruction
Branches	1 bit	Does a Branch
Goes	1 bit	Does a Goto
	1 bit	Unused
IsCond	1 bit	Has a branch condition
W2	14 bits	Second imaginary address of DBLxxx or .+1
JBCT	1 bit	The target of a JCN-encoded conditional branch
GoedTo	1 bit	The target of an unconditional or false conditional Goto
Called	1 bit	Target of unconditional or false conditional Call
jbcLink	15 bits	7777 if no JCN conditional branch else 10000+imag addr

"Link" then scans IM, doing the following for each word:

a. AlcPtr and Link are initialized to 0.

b. If W2 is relevant (= IsCond & not Returns), then W2 must be at W1 xor 1, so the W0 and Place fields are set appropriately for both words, making error checks for inconsistent constraints.

c. The JBCT, GoedTo, and Called bits are set in W1 as appropriate (ignored if Returns eq 1).

d. The word containing W2, now disposed of, is converted into a brLink. If Returns or (not IsCond & not UsesFF), then W1 can be anywhere and no restriction is propagated. Otherwise, W1 must be in the same page as this instruction. Either brLink or jbcLink is set to 10000+W1 and the other is set to 7777 (= empty). jbcLink is used when the branch target must be a reachable JCN-encoded conditional branch location.

While propagating xor1 relationships, error checks ensure that no situations where different instructions must be xor1 to the same instruction occur. If such errors are detected, error messages are output, and at the end of "Link" assembly terminates.

"Link" then scans simulated IFUM and, for IFU entries which have been loaded, sets the IFUE state in Place for addresses branched to from the IFU; if NEnt is greater than 1, then Calls is set 1 in the first NEnt-1 instructions of the entry vector.

At the end of "Link" simulated memory is as follows:

AlcPtr	20 bits	Pointer to the alist header
Link	20 bits	Pointer to next alist item
	1 bit	Unused
Place	3 bits	Placement constraint
W0	14 bits	Absolute address of this instruction if Place eq 0

Returns	1 bit	JCN is correct and W1 is irrelevant
Calls	1 bit	the next imaginary instr must be placed at .+1
JBC	1 bit	A branch condition is in JCN
UsesFF	1 bit	FF field not available for long call or long goto
W1	14 bits	Imaginary address of branch from this instruction
State	3 bits	State of allocation list (now 0)
brLink	15 bits	Imaginary addr of next instr in page or 7777B if empty
JBCT	1 bit	Target of branch with condition in JCN
GoedTo	1 bit	Place at a goto location
Called	1 bit	Place at a call location
jbcLink	15 bits	Imaginary addr of next instr in subpage or 7777B if empty

At the end of "Link" each instruction contains a collection of flags and W0 describing restrictions on its placement, and the lists beginning at jbcLink and brLink thread through instructions on the same subpage or same page. W1, UsesFF, and Returns indicate how the JCN (and sometimes the FF) field must be filled in for its own branch. Calls connects it to the instruction at .+1, and Calls in the preceding imaginary word may connect it to .-1.

The "AList" pass of MicroD transforms data structures left by "Link" into a form more amenable to allocation. The word containing W0 and the JBCT, GoedTo, and Called bits are processed so that the placement constraints are contained in a one-word "Mask" and in the three-bit "State" field. The jbcLink and brLink lists are transformed into circular lists as follows:

a. Initially, xxLink contains 7777B (empty) or 10000+imaginary addr, interpreted as an "unmarked" pointer.

b. Imaginary addr in xxLink is interpreted as a "marked" pointer (which implies that imaginary address 7777 is unusable--sorry about that, but the allocator is unlikely to be good enough to assemble 100% of the microstore anyway).

c. During the scan of IM, if xxLink is already marked, skip it. Otherwise,

d. Follow and mark the xxLinks until either 7777 (empty) or a marked link is encountered. If empty, change that to a marked pointer to the starting xxLink. If a marked pointer, splice the list just scanned in at that place, except that if the marked pointer is at the original xxLink then done (List was already circular).

Next, "alists" of instructions connected by Calls or Xor1 are built. Alists have the property that the placement of every instruction in the alist is determined unambiguously by the placement of any other element. Alists begin at a header and thread through the Link words of IM entries in the alist. The interpretation of State is as follows:

0 Absolute--list contains absolutely located instructions -or-
Page-relative--alist contains instructions whose low 6 bits are located

1 Other--placement constraint encoded in Mask (currently unused)

2 Xor1--two-instruction alist with instructions at an xor1 pair, legal placements encoded in Mask

3 Plus1--multi-instruction alist with instructions bearing a .+1 relationship to predecessors

4 AnyCall--one-instruction at any call location

5 AnyGo--one-instruction at any goto location

6 AnyIFUE--one-instruction at any IFU entry

7 Any--one-instruction arbitrarily located

Legal alists containing arbitary combinations of Calls and Xor1 constraints are transformable into a "Plus1" list. Header locations for the alists are determined as follows:

a. Absolutely-located alists have their header in the PageTab entry (see below) for the appropriate absolute page. All absolutely-located instructions in that page are on that single alist.

b. Page-relative alists (i.e., ones containing a Global) have header in GlobTab.

c. AnyCall, AnyGo, AnyIFUE, and Any instructions which have both jbcLink and brLink equal to 7777 (empty), are combined onto single lists. These are not considered to be part of any instruction cluster and are allocated at the last possible moment. Instructions which are only reached by long Goto/Call or IFU dispatch and which themselves do long Goto/Call, Return, or IFUJump wind up on these lists.

d. All other alists have their headers in AlcTab.

The AlcPtr word in each IM word's structure points at the alist header. This is needed for clustering instructions into pages.

The "Cluster" pass of MicroD groups and sorts the alists into clusters of instructions that must appear on the same 64-word page of the microstore. This is done in the following steps:

a. Absolute clusters for pages 0-77 are collected and sorted by size.
b. Global clusters are collected and sorted by size.
c. Global clusters are merged into page 0-77 clusters.
d. Remaining clusters are collected and sorted by size.
e. Remaining clusters are merged into page 0-77 clusters.
f. The page-independent AnyCall, AnyGo, AnyIFUE, and Any alists are allocated.

The "seed" alist for the cluster gathering procedures is obtained as follows:

a. The PageTab entry for a page contains its absolutely-located instructions.
b. GlobTab entries not absorbed during (a) are seeds for global clusters.
c. Take AlcTab entries not absorbed collecting other clusters in an arbitrary order.

Note: The circular jbcLink and brLink lists form a fully-connected structure, so the cluster gathering process can begin with an arbitrary seed alist. The purpose of collecting the clusters in the careful order described above is to avoid unnecessary sorting of the clusters and avoid undesirable thrashing by the cluster-merging heuristic.

As a cluster is collected, the alists composing it are aggregated into adjacent AlcTab locations, and the single-instruction alists (probably 80% of all instructions are on single-instruction alists) are rplaca'd onto special lists for the cluster. The PageTab or ClusTab structure describing a cluster is as follows:

a. Pointer to first AlcTab alist.
b. Count of alists in AlcTab.
c. Header for AnyCall instructions in cluster.
d. Header for AnyGoto instructions in cluster.
e. Header for Anywhere instructions in cluster.
f. Header for absolutely-located/page-relative instructions in page/cluster.
g. Count of total instructions in cluster.

PageTab only:
h. Count of total goto locations occupied by current allocation of page.
i. Count of total call locations.
j. Count of total JCN locations.
k. Count of total JCN conditional branch goto locations.
l. Count of total JCN conditional branch call locations.
m. 4-word bit table for allocation.

This information is needed by the allocate-and-merge heuristic. A rough sketch of the heuristic is as follows:

a. Initially, each PageTab entry contains the assorted lists described above and an empty bit table for the page.

b. The alists in AlcTab are sorted into a desired allocation order (undecided how this works at present).

c. The AlcTab alists are allocated, the bit table bits filled in, and the tentatively assigned location stored in brLink (which is no longer needed).

d. The assorted counts are filled in by counting the ones in the bit table appropriately. To these counts are added the lengths of the Anyxx lists.

e. Merges are considered in the order of decreasing size. Namely, the can-I-merge question is asked for the largest entry in ClusTab with the largest entry in PageTab and then successively smaller PageTab entries until the answer is "yes".

f. If either the PageTab or the ClusTab entry contains only alists beginning in the Anyxx headers (i.e., there are no AlcTab alists for the cluster), then the merge question can be answered by considering only the assorted counts. Otherwise, the counts will provide a certain negative answer for most situations when the merge is impossible.

g. If the PageTab entry is empty (i.e., the page hasn't been used yet), then the merge is ok, so the page-relative alists in the cluster are converted to absolute, the AlcTab alists are sorted into position and the bit table and counts are filled in as above. This may result in an error if the cluster is too big for one page.

h. If the counts indicate that a merge is probably ok, then the bit table in PageTab is copied and an attempt is made to allocate the alists in the cluster without changing any location assignments already made for the PageTab alists. If this succeeds the clusters are merged with the cluster's AlcTab alists being appended after the ones already in PageTab.

i. If (h) fails the answer is presently assumed to be "no". (This can be improved later by resorting the alists in PageTab and in the cluster, but maybe the heuristic will work well enough without resorting to this time-consuming reallocation.)

j. If the answer is "no" then loop to the next smaller PageTab entry.

The "Allocate" pass of MicroD is carried out as follows: Each entry in PageTab now represents instructions that will wind up on a single absolute page. AlcTab alists have already been assigned absolute locations (assignment in brLink). Absolute locations are now assigned to the remaining instructions on the AnyCall, AnyGoto, and Any lists for each page. Then the instructions on the page-independent AnyCall, AnyGoto, and Any lists are allocated wherever there is space.

The "Relocation" pass of MicroD rereads the .Dib binaries, checks assignments of IM words, and outputs a .Mb file in the form expected by Midas. Memory definitions, addresses, and data for all memories except IM and IFUM are output unchanged in the order read, except that the fake memories intended only for MicroD (RVREL, IMLOCK, VERSION, and IMLOCK) are flushed. IM addresses are also output unchanged--they are not relocated because Midas works with the unrelocated addresses.

However, modified definitions for IM and IFUM are output, and MicroD builds an in-core data structure for IM and IFUM words so that these memories can be listed on the .Dls file. To do this, it compresses Dorado instructions into the form shown below; IM address symbols are appended to the appropriate symbol chain.

MicroD fills in JCN (and sometimes FF) fields of instructions and IFUM words with absolute information. In filling in JCN the rules are as follows:

1. If the instruction has a branch condition in JCN, only JCN[1:4], the 4 bits selecting from 16 possible target addresses, are filled in by MicroD (other bits were filled by Micro.).

2. If the instruction has Returns=1, no fixup is made.

3. Otherwise, all bits JCN[0:7] are set by MicroD to the correct values, and for long gotos/calls FF[0:7] are also set.

MicroD must also zero the extraneous bits in each IFUM word.

After this, representation of the IM words is as follows:

Dorado instr.	44 bits	
	14 bits	unused
	2 bits	unused
Undef	1 bit	This bit must be 0
Emul	1 bit	Print out as an emulator instruction
AbsAddr	14 bits	
SymLink	20 bits	Pointer to chain of symbols

Then:

a. Memory definition blocks compatible with Midas are output for all memories on the .Mb file; the sizes expected by Midas are as follows:

IM	10000 words x 100-bits (IM representation given above with SymLink removed)
IFUM	2000 words x 40-bits
other	passed through MicroD unchanged

b. Data blocks are output on the .Mb file for all memories. IM words are represented by the 14-bit absolute address as well as the data, so that both the imaginary and absolute addresses are available to Midas during debugging.

c. IM words are output as data blocks beginning at 0 and extending to the last imaginary location used by the program.

d. Finally, the Micro endblock is output.

Appendix 2. Recent Hardware and Assembler Changes

1. The Micro "While" builtin has been added (affecting DlLang internally but probably uninteresting to programmers).

2. The DispTable placement macro has been added, supported by the IMMask memory in MicroD.

3. The StackNOUFL, StackNOUFL&+1, StackNOUFL&+2, and StackNOUFL&+3 macros have been added to read the top stack entry without checking for a StkP=0 underflow condition.

4. The macros for restoring an ALUFM entry that has been smashed have been added; the nH literals have been removed.

5. The "Cnt-1" macro has been added to use the Cnt=0&-1 branch condition for its side effect.

6. "SetRMRegion" has been added so that the definition of an RM region can be in a different file from definitions for registers in that region.

7. The BRX (fake) memory has been added for use in contexts which specify the MemBX-relative loading of MemBase (e.g., in defining IFUM entries, MemBaseX←SC)

8. The "IMReserve" and "IMUnreserve" macros have been added to prevent/allow MicroD use of absolute microstore locations.

9. The "OnPage" and "AutoPage" macros have been added to force MicroD placement on a particular page and to allow general placement (primarily for microcode overlays).