3600 Microcode

Revised December 1983

Table of Contents

Table of Figures

1. Introduction

This document is both to explain the underlying concepts of the microcode compiler and to serve as reference documentation on how to write microcode. The first part of this document explains the philosophy of the microcode compiler and is largely independent of the hardware. The second part describes the microcode operations available; in some sense this constitutes documentation of the hardware, however the reader is assumed to be familiar with the hardware at least at the block-diagram level.

1.1 Structure of the Compiler

The 3600 microcode compiler consists of a front end, a checker, two back ends, and a linker. The input to the compiler is written in a *microcode source language*, which has Lisp-like syntax but the semantics of microcode. This language is considerably higher level than the actual microcode executed by the machine, but is in no sense a high-level or general-purpose language; it is impossible to program in it without general knowledge of the microcode architecture of the machine. The purpose of the microcode source language is to provide a more comfortable syntax, to take care of some elementary bookkeeping, and to provide extensive error checking so that the microprogrammer's knowledge of the microcode architecture need not be perfect for her to program effectively.

The *front end* converts the microcode source language into a primitive form. This primitive form is horizontal microcode expressed in symbolic form, rather than as a bit string; it is a list of microinstructions, where each microinstruction is a list of microinstruction fields and values for those fields. To look at it in another way, each primitive symbolic microinstruction is a list of machine operations to be performed in parallel, and these machine operations are the primitive ones actually implemented by the hardware, rather than any higher-level abstractions that the programmer would use. An effort was made to keep the symbolic microinstructions close to the actual microinstructions executed by the machine, so that the "smarts" of the compiler would be in the front end, and the back end would simply be a trivial transformation from symbolic microinstructions to their actual bit encodings. However, there are some decisions that are difficult to make in the front end, because they require global knowledge. These decisions are made in the back end, which means that the symbolic microinstruction is not identical to the real microinstruction. For example, choices between two different ways of encoding the same function are usually made in the back end. A few additional minor "unrealities" in the symbolic microinstruction are there to simplify the simulator.

The front end is essentially a macro expander. The microcode written by the user consists of nested expressions in the style of Lisp; the `car` of an expression is the name of a macro that defines the operation to be performed. The microcode source language will be discussed in detail later.

The *checker* checks the legality of the primitive symbolic microcode output by the front end. It checks for unknown symbolic field names, for unknown symbolic field values, and for inconsistencies between fields. The symbolic microcode is not perfectly horizontal: there are dependencies between fields. To take some examples, reading the output of a memory requires specifying an address for that memory, many operations are "modulated" by the *magic number* field, and some combinations of fields are not allowed by the hardware. The main purpose of the checker is to detect bugs in the front end (the macros are many in number and possibly user-written). This checker is partly table-driven and partly ad-hoc; it was written in whatever way seemed most convenient at the time.

The front end also does some checking; whenever two things are done in parallel it checks that they are consistent. Part of this check is simply to assure that the same symbolic microinstruction field is not given two different values, and part of the check involves machine-dependent knowledge of how to encode parallel operations. Of course the front end also checks for trivial syntactic errors such as using an undefined macro, using a macro with the wrong number of arguments, etc. Most error messages seen by the user will be from the front end.

One *back end* translates symbolic microcode into Lisp functions to simulate its action, running within a simulator environment that runs in both Maclisp and the Lisp Machine. The simulator allows microcode to be debugged with the same editing and debugging tools as Lisp programs. The simulator is deficient in certain details, mostly having to do with memory and I/O, but simulates the microcode that executes the compiled Lisp instruction set well.

The other back end translates into the actual microcode executed by the machine. This consists of translating symbolic microinstruction fields into the appropriate bit strings, packing the fields together, and making certain decisions when there are overlapping fields in the hardware and/or multiple ways of encoding a symbolic operation.

The *linker* combines separately-compiled microcode modules into an almost-complete image of the hardware memories. Constants are assigned to addresses. Microinstructions are also assigned to addresses; this is a complex process because there are several relationships between the addresses of multiple microinstructions: the hardware microinstruction has only a single successor address field, whereas in general two successors are required (e.g. the address of a subroutine and the address it should return to, or the address of the normal successor and the address of a trap handler which receives control in exceptional cases); the microcode to execute a macroinstruction must be at a certain address determined by the Instruction Fetch Unit; the *skip* and *dispatch* features involve tables of microinstructions located in a block of addresses. The linker makes multiple copies of a microinstruction when necessary to satisfy these constraints, and merges together microinstructions that come from different places in the source code but turn out to execute the same machine operations. The linker generates a symbol table for the microcode debugger, and optionally a report file showing how addresses were assigned.

1.2 Level of Sophistication

The 3600 microcode compiler is primarily analytic rather than synthetic. In other words, it does little *planning* or *scheduling* of the use of hardware resources, and its input is not a general-purpose programming language, but one whose primitives correspond closely to the hardware. It takes a program written by a human and analyzes it to make sure that it will work (i.e. that the human has not out-smarted himself).

The compiler does no scheduling (arrangement of machine operations in time). The programmer must explicitly say which operations are to be done in parallel and which operations are to be done sequentially. The compiler will then say whether or not this "schedule" will work, but it has no idea whether or not this "schedule" is the most optimal one. The compiler knows all the hardware reasons why two operations cannot be done in parallel (e.g. they might require two different data words to be present on a single data path at the same time, or they might require a single microinstruction field to contain two different values). It would be much too difficult (for this project) to write a program that could schedule the performance-critical microcode as well as human ingenuity can.

Things are somewhat simplified because the hardware does some very low-level scheduling. One example of this is that all read/write memories in the processor contain pass-around paths so that a value may be written into the memory and read back correctly in the immediately-following microinstruction (even if it has not yet really been stored into the memory). This means that neither the compiler nor the programmer needs to worry about scheduling issues across multiple microinstructions, but only within a single microinstruction. Another way of putting this is that most of the processor's internal pipelines are invisible to the microprogrammer.

When dealing with main memory, and other Lbus devices, the programmer does need to understand the pipelining and does need to deal with scheduling issues across multiple microinstructions. The compiler includes some error-checking that is designed to detect the programmer's mistakes. The need for the programmer to have enough freedom to be able to optimize use of the memory makes it difficult for the hardware or the compiler to hide the existence of this pipeline from the programmer.

The microcode compiler has about the same lack of intelligence in the space domain as in the time domain. The programmer generally has to have a good idea of which data paths in the machine his microcode is using. The programmer must choose explicitly whether variables and constants reside in the A memory or the B memory (the machine can access one A-memory location and one B-memory location simultaneously, but not two locations in the same memory.) The compiler does not take a high-level description of what is to be done and map it onto the data paths. However, the situation is not hopeless. The compiler does make the simplest data-path planning decisions on its own; for instance it will take advantage of the symmetries of the ALU. This will be discussed in detail later. In addition, if the programmer mistakenly tries to use the data paths in an impossible way, the compiler will detect this.

1.3 Macros and Micros

The source file for a microcode module is a file full of Lisp forms, much like the source file for a Lisp program. There are certain *defining forms*, which are Lisp forms (macros, actually) that define microcode subroutines or other microcode-related things. Inside of a Lisp form that defines a microcode subroutine appears some actual microcode (in the source language form). This microcode could be written directly in the primitive, symbolic microinstruction form, however it is invariably written in a higher-level form, in terms of *micros*. Micros are the macros expanded by the front end of the microcode compiler. They are called micros to distinguish them from normal Lisp macros.

The syntax of the microcode source language is as follows. A valid form (or expression) is one of

a primitive
: A primitive is a list whose `car` is the name of one of the primitive operations defined in the next chapter, and whose `cdr` is the appropriate arguments to that operation.

the invocation of a micro
: This is a list whose `car` is the name of a micro and whose `cdr` is interpreted in a way defined by that micro.

a symbol
: The symbol must be defined as an *atomic micro*. Most atomic micros are used the way variables are used in Lisp. The phrase *atomic micro* is usually abbreviated to *atomicro*.

Note that the Lisp concept of *evaluation* does not apply to the microcode language, even though it looks much like Lisp. A microcode form is processed by *expanding* it into another form, if it is the invocation of a micro or an atomic micro, or by converting it into hardware microcode, if it is a primitive.

The compiler comes with a large number of micros pre-defined. These micros embody knowledge about the hardware architecture (how to get the machine to do things) and about the software architecture implemented by the microcode (conventions about storage layout, names of fields in data structures, etc.) The predefined micros include the usual set of control-structure operations, some additional control operations corresponding to the hardware, and data operations corresponding to all the data manipulations the hardware is able to perform.

There is a defining form (**defmicro**, a Lisp macro) that can be used to define new micros. Its body is a Lisp program that sees the invocation of the micro and computes a new microcode form to serve as its expansion. The front end works by calling the Lisp program associated with each micro it sees, until everything has been expanded into primitives. There are other defining forms for atomic micros.

1.4 Macroinstructions and Microinstructions

A *microinstruction* is the smallest unit of microcode in the machine. On every clock cycle the machine executes one microinstruction and selects which microinstruction will be executed next. Microinstructions are stored in *control memory*.

A *macroinstruction* is the smallest unit of compiled Lisp code. A compiled function consists of a header, a table of constants and external references, and a sequence of macroinstructions. The word *macrocode* is sometimes used to refer to a sequence of macroinstructions, by analogy with microcode. Macroinstructions are stored in main memory or virtual memory.

When the word *instruction* is used without qualification, it generally means macroinstruction.

The principal business of the microcode is to execute macroinstructions; this process is sometimes called *emulation*. The execution of a macroinstruction requires the execution of one or more microinstructions; the microinstructions provide the detailed control of the hardware needed to emulate the higher-level function called for by the macroinstruction. The 3600 is designed so that many simple macroinstructions can be executed by a single microinstruction; in this case control memory serves simply as a lookup table that translates macroinstructions into microinstructions. More complex macroinstructions require the execution of multiple microinstructions.

1.5 Other Work

The general idea of translating a reasonably high-level Lisp-like language into microcode comes from the Scheme Chip project at MIT, as does the idea of having the programmer control the parallelism explicitly in the way to be described below. A number of the details are very different.

1.6 Compiling the Microcode

On a Lisp machine, load the file SYS: L-UCODE; SYSDCL LISP then do (make-system 'micro). This will load the compiler, the simulator, and the microcode. The source files for everything are on the SYS: L-UCODE; directory. Everything resides in the MICRO: package.

The **micro** system consists of two components: **microcompiler,** the compiler and simulator; and **microcode,** the microcode proper. The **microcode** system contains a component system for each version of the microcode (there is a separate version for each incompatible kind of hardware, plus a version for the simulator). These component systems all read roughly the same source files, but produce separate compiled output files. Since the compiled output files are *bin* files, there are separate copies of these files for whichever machine (3600 or LM-2) is used to run the microcode compiler.

The version number of the microcode comes from a *version file,* which is rewritten whenever a version number is allocated. The microcode linker asks you whether to increment the version number, telling you what the current version number is and whether or not microcode with that version number has already been linked. The **:version-file** option of **defsystem** is used to specify the pathname of the version file.

Frequently there are several related microcodes that all share the same version number. For example, the various component systems of **microcode** are all variations of the standard system microcode that differ only in the hardware configuration they run on. These microcodes are maintained together and have synchronized version numbers. Consequently they share a single version file. The **:version-of** option of **defsystem** is used in each of these microcode systems to specify that they use the version file of the **microcode** system.

Currently the component systems of **microcode** are **tmc5-io2-microcode, tmc5-io4-microcode, tmc5-io2-tape-microcode,** and **ifu-io2-microcode. proto-microcode, tmc-microcode,** and **sim-microcode** also exist, but since we no longer have any prototype machines, no longer use the rev-1 TMC, and don't use the simulator much anymore, these are not made components of **microcode** and will not be recompiled automatically.

The **microcompiler** system includes the three architecture-definition files that are also read by several other parts of the 3600 system (the compiler, the cold-load generator, the LIL macros used by the FEP, the debugging console, and the Lisp system itself). The machine architecture might be circularly defined as those aspects of the 3600 that must be understood and agreed upon by all of the above programs.

The SYSDEF file defines all of the constants, byte fields, and storage layouts associated with the machine architecture. The SYSDF1 file defines the areas of storage used to communicate between the various parts of the system, and defines the escape routines (macrocode routines called by the microcode when it needs assistance or needs to trap out to Lisp code). The OPDEF file defines the macroinstruction opcodes and formats. All of these files may be found on the SYS: L-SYS; directory.

To compile microcode, use **make-system**. For example:

```
(make-system 'microcode ':compile)
```

After compiling microcode, you must link it, install it onto the FEP file system, and load it into the machine. These steps are discussed in the following sections.

machine-version-list *Variable*

This variable is bound by **make-system** to control which version of the microcode is being compiled. When doing incremental compilation from the editor, you should set it by hand to an appropriate value. [**(tmc5)** is the usual value.] The value is a list of symbols. Each symbol specifies some feature that is to be included in the microcode. The order of the symbols in this list matters.

One of the following symbols must be included:

sim	Microcode to run in the simulator
proto	Microcode to run on the prototype machine (which has been retired)
tmc	Microcode to run on a machine with a rev-1 temporary memory control board (in place of the IFU)
tmc5	Microcode to run on a machine with a rev-5 temporary memory control board (in place of the IFU). The rev-5 TMC has a macroinstruction format compatible with the IFU.
ifu	Microcode to run on a machine whose memory control board includes the IFU (instruction fetch unit).

One of the following symbols must be included:

io2	Microcode to run with revision 2 or 2A of the IO board.
io4	Microcode to run with revision 4 or higher of the IO board.

Other interesting symbols are:

xsq	Support the XSQ board, which has 16K of control memory instead of the normal 8K.
tape	TD-80 support.

One way to make private or specialized versions of the microcode is to add additional symbols to this list.

When a microcode system is defined (with **defsystem**), the system definition begins with the clause

`(:machine-version` *symbol symbol*...`)`

This controls the types of the compiled files generated by this system, and causes ***machine-version-list*** to be bound to that list of symbols during compilation and loading of this system.

Another **defsystem** clause that may be used is

`(:include-microcode` *system*`)`

This allows one microcode system to be built on another; presumably the two would have different **:machine-version** clauses. For example, the **tmc5-io2-microcode** and **tmc5-io4-microcode** systems are both built on the **tmc5-bare-microcode** system, which contains all of the microcode that is not dependent on the machine's I/O configuration. The **:include-microcode** feature can also be used to add a new microcode module to the standard microcode.

Two other **defsystem** clauses that were mentioned above are

`(:version-file` *pathname*`)`

The file *pathname* contains the highest version number that has been allocated for this microcode, in decimal.

`(:version-of` *system*`)`

The microcode system being defined uses the same version file as *system*.

The following Lisp functions are useful when debugging micros:

ppu *name*

Pretty-print the microcode routine named *name*. The routine could be defined with **defucode, definst,** or their variants. The output is the primitive symbolic microcode.

ppx *code*

Expand all micros in *code* and pretty-print the resulting primitive symbolic microcode.

1.7 Linking the Microcode

When using the simulator, microcode is compiled into Lisp functions, using the normal Lisp compiler, and these Lisp functions are run with the aid of support functions in the **microcompiler** system. When using the real hardware, microcode is compiled into linkable microcode, stored in files called *micrel* files. The type field of the file name depends on the machine version. In general, the type is constructed by concatenating all of the symbols in ***machine-version-list***, separated by hyphens, followed by `-MICREL` for the LM-2, or `-MICREL'` for the 3600. In the interests of brevity, **lo2** is elided. The canonical type mechanism of the Lisp machine pathname system is used to cater to any restrictions placed by particular file systems on allowable types. Thus compiled linkable microcode for the TMC machine typically has a type of `TMC-MICREL`. These files are loaded with **make-system** and the microcode is then processed by the following functions:

compile-the-microcode *system*

Link the microcode and then write it. This is the function that you usually call.

system is a symbol which is the name of a microcode system.

Note that **compile-the-microcode** does not actually compile the microcode source files; use **make-system** for that.

compile-all-microcodes

Call **compile-the-microcode** on all standard microcode systems; these are the component-systems of the **microcode** system.

link-the-microcode *system*
Build a microcode memory image out of all of the microcode that has been loaded. *system* is a symbol which is the name of a microcode system.

linker-summary-report
Print a summary of the results of the most recent **link-the-microcode**. This consists mainly of how many locations were used.

linker-detailed-report
Print a detailed listing of the results of the most recent **link-the-microcode**. This includes complete symbol tables and maps of control memory.

memory-usage-report
Print information about usage of A and B memory. This can be helpful when trying to find locations to reserve with **reserve-scratchpad-memory**.

cmem-bloat-report
Print information about usage of C memory, broken down by microroutine and by source file. This report includes a summary of the splitting of logical microinstructions into multiple physical locations and the combining of multiple logical microinstructions into single physical locations.

file-linker-report *pathname*
Write a file named *pathname* containing the output from **linker-summary-report** and **linker-detailed-report**.

write-the-microcode *system* &optional *link-p name version*
Write the microcode defined by the *system* into a set of microcode binary files (see section 1.9, page 9). If *link-p* is **t**, the microcode is linked first; otherwise the results of the most recent **link-the-microcode** are used.

name is the file name for the files written. It defaults according to the **:machine-version** clause of *system*; a typical name is `TMC5-I04-MIC`.

version is the version number of the microcode, and also the version number for the files written. It defaults to the current version of *system* as specified in its version file. **compile-the-microcode** offers to increment this version number before calling **write-the-microcode**.

When the machine is booted with this microcode, *name* and *version* are stored into the **sys:%microcode-version** communication array.

system is a symbol which is the name of a microcode system.

new-microcode-version *system*
Increment the version number of the microcode defined by *system*. *system* is a symbol which is the name of a microcode system.

1.8 Installing Microcode

To install microcode, use the `si:install-microcode` function to copy the `MIC` file onto the `FEP:` file system. The m-X Copy File command could also be used, but it doesn't set the file's comment and doesn't default its arguments as nicely. Typically `si:install-microcode` is called with one argument, a microcode version number. The pathname of the `MIC` file may also be used as the first argument. The optional second argument to `si:install-microcode` is normally only used in Release 4; it is the name or number of the FEP file to be rewritten. In Release 5 the second argument is omitted and `si:install-microcode` creates a new FEP file with a suitable name. After copying the `MIC` file onto the `FEP:` file system, `si:install-microcode` offers to update the Load Microcode command in the boot file.

Of course, after installing microcode into the `FEP:` file system you must load it into the machine in order to use it. This requires halting the Lisp processor and then warm-booting it. The dialogue looks like:

```
(si:halt)
Lisp stopped itself.
Fep> load microcode >filename.mic
Fep> start
```

1.9 Microcode Binary Files

The microcode is stored in four files, with types `MIC`, `SYM`, `ERR`, and `LOG`. The name of each file is the name of the microcode; a typical name is `TMC5-IO4-MIC`. The version number of each file is the version number of the microcode.

The microcode binary files are stored on the host, device, and directory specified by the **:pathname-default** option of **defsystem** in the microcode system or in one of its component systems. The standard system microcode binary files are stored on `SYS:L-UCODE;`. Specialized microcode may be stored on a different directory. (Note, however, that currently the error-table loading that occurs when the world is booted with a new microcode always looks for the `ERR` file on `SYS:L-UCODE;`.)

`MIC` This file contains the load image for the microcode memories in the machine. This is the only file read by the FEP.

The `MIC` file is a sequence of 8-bit bytes, divided into blocks, each of which starts with a byte containing a type code. All multi-byte data are stored low-order byte first. The type codes and the data that follow each code are as follows:

1 One byte, the microcode file format version number. The current version is 5.

2 Two bytes of microcode version number.

3 Microcode name. One byte of length, followed by that many bytes of the name. (Obviously, the name must be less than 256 characters long.)

4 A-memory contents. Any number of data blocks of the format

2 bytes of length

2 bytes of address

length*5 bytes of data

The A-memory contents block is terminated by a single 2-byte length = 0 field (no address follows).

5 B-memory contents, in the same format as the A-memory contents.

6 C-memory contents. Any number of data blocks of the format

2 bytes of count

2 bytes of address

count number of microword descriptors
Each descriptor consists of 14 bytes of microinstruction, followed by any number of modifier bytes:

0 A zero byte marks the end of the modifiers.

1 Put the slot number of the IO board into the `U AMWA <9:5>` field (microinstruction bits 21-25).

2 Put the slot number of the half-inch magnetic tape interface (MTI or GBUS board) into the `U AMWA <9:5>` field (microinstruction bits 21-25).

The C-memory contents block is terminated by a single 2-byte length = 0 field (no address follows).

7 Type-map contents, in the same format as A-memory contents except that each data word is only one byte rather than five bytes.

8 End of file

9 Hardware configuration description [to be defined].

10 IFU dispatch table. This block is not present in microcodes compiled to run on machines equipped with a TMC instead of an IFU. Any number of data blocks of the format

2 bytes of length

2 bytes of address

length*4 bytes of data

The block is terminated by a single 2-byte length = 0 field (no address follows).

The microcode loader expects the blocks to come in the above order, except that the End of File block is last. Generality was *not* a goal.

SYM This file contains the symbol table, as a series of Lisp lists, each having an identifying symbol in its car. These symbols are:

version	The microcode version is the cadr of the list. It takes the form of a list **(version** *name number***)**.
a-memory	The rest of the list is an a-list associating symbols with A-memory addresses.

b–memory	The rest of the list is an a-list associating symbols with B-memory addresses.
c–memory	The rest of the list is an a-list associating microinstruction names with *lists of* control-memory addresses. A microinstruction can be stored in more than one location when address constraints so dictate. A microinstruction name is either a symbol (specified with **defucode** or **definst**) or a list, which is one path from a symbol-named microinstruction to here. When there can be multiple paths to a microinstruction (because identical microinstructions from different sources were merged by the linker), only one path is remembered.

A path is a list whose first element is a symbol, the name of a microcode routine. Succeeding elements may be

a number	A microinstruction in straight-line code. The number is the number of microinstructions that precede this one.
trap	Diversion of control to an anonymous trap handler.
true	One branch of an **if**.
false	The other branch of an **if**. Note that the **true** and **false** branches may correspond to the *true* and *false* clauses or to the *false* and *true* clauses, respectively, depending on the particular condition being tested.
(*number***)** or **(***symbol***)** or **otherwise**	One clause of a dispatch. Multiple numbers or symbols may appear in the list.

ERR This file contains the error table, which is read by the Lisp system during loading. It tells the error handler how to interpret error traps from the microcode. The format is similar to the **SYM** file. Valid cars of lists are:

version	The microcode version is the cadr of the list. It takes the form of a list **(version** *name number***)**.
error–table	The rest of the list is an a-list associating control-memory locations to error codes, specified by the **signal–error** micro, for example.

LOG This file records the truenames of the source files that were compiled into the microcode, any error messages that were printed during microcode linking, and the size of the microcode.

1.10 Making Your Own Microcode

To make your own microcode you must choose a new machine version symbol, a name for your microcode system, and a file type for your compiled microcode files. The machine version symbol will keep your compiled microcode separate from the normal microcode if both are loaded into the compiler/linker at the same time. The machine version symbol also controls the names of the microcode binary files produced.

Start by loading the file `SYS: L-UCODE; SYSDCL` and the **microcompiler** system. Put your microcode system in the same `MICRO:` package. To make the system declaration for your microcode, you will need to imitate some of the things in the `SYS: L-UCODE; SYSDCL` file. Suppose your new machine version symbol is **mud**. Then the system declaration for the version of your microcode that runs on a machine with TMC rev-5 will start with the clause **(:machine-version mud tmc5)**, assuming that your microcode doesn't depend on what version of IO board the machine has.

Define a new canonical type for your compiled microcode files named **:mud-tmc5-micrel** and make it map into whatever file type you prefer. The **:machine-version** clause will automatically cause the **:micro-compile-load** transformation to compile into files with that type. Like `BIN` files, compiled microcode files are machine-dependent, so if you plan to run the microcode compiler on both 3600s and LM-2s you will need to map the canonical type into different file types on the different machines. Write the **defsystem** for your microcode, incorporating any files from the system microcode that you require. If you are just adding new instructions, you can use **:include-microcode** to include the system microcode version appropriate for your machine. If you need to modify system microcode to call your microcode, you will need to make a complete copy of the modules and transformations in the system microcode's **defsystem**.

You must create a version file for your microcode, to remember the highest version number that has been allocated so far. If you share the standard microcode's version number, your system declaration includes the clause **(:version-of microcode)**. Otherwise, you must create a file with a name such as `version.text` containing the number 1 (or whatever you want the starting version number to be) and include in your system declaration the clause **(:version-file "version.text")**.

The **microcompiler** system includes the three architecture-definition files. You will need to modify these to a larger or smaller extent depending on what your microcode does. For large changes, you will want to make your own copies of these files and your own copy of the **microcompiler** system that uses your files instead of the standard ones. If you are simply adding some new instructions, you can use the standard microcompiler and the standard architecture files; simply make your own file with additional **defopcode** forms. Make sure that you don't duplicate any opcodes used by the system; opcodes 700 through 777 and 1700 through 1777 are reserved for special microcode made by customers; 600 through 677 and 1600 through 1677 are reserved for special microcode made by Symbolics but not part of the standard system microcode. Use the **function** option to **defopcode** to define instructions that can be called as Lisp functions. The compiler will automatically use your new instructions. To make the interpreter use them, use the **defuprim** special form in a Lisp file that you compile and load (see `SYS: SYS; LPRIM`). Read your instruction definition file into the **micro** package after loading the microcompiler and before compiling any of your microcode files. Read the file into the **compiler** package before compiling any Lisp programs that use your new instructions.

An example of how to set up your own microcode system that adds a few new instructions follows. Abstruse issues such as making it possible to run the microcode compiler on both 3600s and LM-2s, making it possible to produce multiple MIC files for different hardware configurations, and arranging for the **microcompiler** system to be loaded automatically are ignored in this example.

```
;; Load the microcode compiler

(load "sys: l-ucode; sysdcl")

(make-system 'microcompiler :noconfirm)

;; Define a microcode system for our microcode

(fs:define-canonical-type :mud-tmc5-micrel "MUD-MICREL")

(defsystem mud-microcode
  (:machine-version mud tmc5)
  (:include-microcode tmc5-io2-microcode)
  (:pathname-default "HOST:>User>Microcode>")
  (:version-file "version.text")
  (:module opcode-definitions ("Opdef") :package "Micro")
  (:module my-microcode ("Mud"))
  (:readfile opcode-definitions)
  (:micro-compile-load my-microcode (:readfile opcode-definitions)))

;; Compile our microcode and load the MICREL files for it and for
;; the standard microcode that it includes

(make-system 'mud-microcode :compile)

;; Link the microcode, assign absolute storage locations, and
;; write out the MIC, SYM, ERR, and LOG files
;; on the HOST:>User>Microcode> directory. This will ask
;; whether to allocate a new microcode version number.

(micro:compile-the-microcode 'mud-microcode)

;; Make our new instructions available to the Lisp compiler

(defsystem mud-compiler
  (:pathname-default "HOST:>User>Microcode>")
  (:module opcode-definitions ("Opdef") :package "Compiler")
  (:readfile opcode-definitions))

(make-system 'mud-compiler)
```

2. Primitives

A micro expands either into another micro expression or into one of four primitives. These primitives are *statements* (a single microcode operation), *sequences* (a list of statements to be performed sequentially), *data* (representing the location of data in the machine), and *predicates* (a special kind of data used as a conditional test).

The output of the micro expansion phase consists of pieces of microcode. Each piece of microcode consists of a name, a sequence (or a single statement), and declarative information such as an address at which the microcode must reside.

We will discuss the primitives first, even though the microprogrammer normally never uses the primitives directly, but always programs in terms of the predefined micros and new micros that she writes.

2.1 Statements

A *statement* is a single microinstruction. The symbolic form of a statement is a list
`(microinstruction` *field value field value...*`)`
The *fields* and *values* are a symbolic representation of the machine microinstruction. The actual microinstruction is simplified somewhat, and made more fully horizontal, to simplify the macros.

For example, the microinstruction

```
(microinstruction abus amem
                  amem-read-addr (stack-pointer 0)
                  xbus abus
                  alu X+1
                  write-amem obus
                  amem-write-addr (stack-pointer 0)
                  write-bmem obus
                  bmem-write-addr 24)
```

specifies that the A-memory location addressed by the stack pointer, with an offset of zero (i.e. the top of the stack), is to be incremented by one and stored back into itself, and also into location 24 in the B memory. Data is to be routed from A memory into the ALU via the Abus and the Xbus.

This document makes no attempt to explain the fields that may be used in a statement. Normally the predefined micros take care of this and the microprogrammer operates at a higher level than the primitive statements.

2.2 Sequences

A *sequence* is an ordered list of microinstructions to be performed one at a time. The symbolic form of a sequence is a list

```
    (microsequence statement statement...)
```

2.3 Data

A *datum* represents a word of bits on some data path in the machine. The exact location in the machine is specified, along with a microinstruction that arranges for the desired data to appear at that place when it is executed. This primitive serves the place of expressions in conventional languages. Thus, a micro that represents an expression with a value expands into a datum, while a micro that represents an imperative command with no value expands into a statement or a sequence.

The symbolic representation of a datum is a list

```
    (microdata place statement)
```

The machine does not execute data; it only executes statements. In other words, the microcode language is a statement language, not an applicative expression language, and the flow of data must be programmed explicitly, with the programmer naming temporary storage locations where they are required. Thus data only appear as intermediate operations during the microcode expansion process. When a datum is used as an argument to a micro (for instance, one that takes two data and adds them together in the ALU), the datum's *place* tells the micro how to generate data-routing microcode, and the datum's *statement* is merged into the generated microinstruction and executed in parallel.

A datum may also be used as the first argument to the **assign** micro, in which case the datum designates a place into which bits will be stored, rather than a place from which bits will be retrieved. This generality increases the symmetry of the source language.

It is permissible to have a sequence (rather than a single microinstruction) inside a datum. This is useful if it takes several sequential machine operations to make the desired datum accessible. However, use of this feature may cause non-intuitive behavior, since something that syntactically appears to be a statement, but contains such a datum, will really be a sequence. In applicative constructs involving data, the order in which operations are written usually is determined by esthetics rather than by the order necessary for things to work. Normally an expression is executed in a single microinstruction (i.e. all parts in parallel), and so the order of operations makes no difference. But if a datum in the expression has a sequence buried inside it, the expression will necessarily be executed in multiple microinstructions, and it may not be obvious to the reader what the order of operations is and which things are done in parallel.

2.4 Predicates

A *predicate* represents a true-or-false condition that can be tested. In the 3600 all such conditions appear on a single condition multiplexor, whose output may be used to divert the flow of control with either a skip or a trap.

The symbolic representation of a predicate is much like a datum: a list

```
(microcondition condition-name sense statement)
```

condition-name is the name of a testable condition in the hardware. *sense* is the symbol **true** or the symbol **false**. **false** indicates that the negation of the hardware condition is represented. Predicates may only be used as arguments to condition-testing micros (such as **if**, **not**, or **trap-if**).

3. Combining Forms

There are two combining forms, which can be used to combine several microcode expressions into one. The expressions being combined will usually be statements (or forms that expand into statements), but it also makes sense for them to be sequences. A datum or a condition may be combined with statements or sequences, which makes a new datum or condition whose statement part is augmented by those statements or sequences.

sequential *form1 form2 ...* *Micro*

The argument forms are to be executed sequentially. To make life easier for micros, if any of the *forms* is **nil** it is ignored.

parallel *form1 form2 ...* *Micro*

The argument forms are to be executed simultaneously, in parallel. This form will expand into a single microinstruction, unless one of the *forms* is a sequence. In that case, the forms written before the sequence will be done in parallel with the first microinstruction in the sequence, and the forms written after the sequence will be done in parallel with the last microinstruction in the sequence. Thus the order of arguments to **parallel** does matter. When microcode is written with this in mind, it will usually be more readable anyway—the parallel clauses will "flow" naturally.

For example,

```
(parallel form1
          form2
          (sequential form3a form3b form3c)
          form4)
```

is equivalent to

```
(sequential (parallel form1
                      form2
                      form3a)
            form3b
            (parallel form3c
                      form4))
```

To make life easier for micros, if any of the *forms* is **nil** it is ignored.

Microcode is usually written in such a way that correct execution does not depend on which combining form is chosen; this only affects speed. (Of course, not everything can be done instantaneously, and so correct execution may be impossible with **parallel**; the compiler detects this.) When "bumming" microcode for maximum performance, sometimes operations will be done in parallel that would not work sequentially. For example, the contents of two registers can be exchanged by doing two **assign** forms in parallel. Another example is the use of micros with hidden dependencies on what is done in parallel with them, for instance **alu-carry** or **obus**.

A problem with **parallel** is that

```
(parallel (if condition (true) (false))
          (code))
```

does **(code)** in parallel with the **if**, rather than duplicating it and paralleling it with both **(true)** and **(false)** as one might intuitively expect. It is usually clearest to put control-transferring micros, such as **if**, **jump**, **call**, and **return**, last in any **parallel** form.

The control-structure micros, such as conditionals and dispatches, are something like combining forms in that their arguments are microinstructions. They are described below in chapter 5, page 24.

for-effect *datum* *Micro*

Convert a datum into a statement. This is useful when the datum has side-effects in its statement part (typically popping the stack), but the value is not needed.

machine-version-case *clause clause...* *Micro*
machine-version-case *clause clause...* *Special Form*
machine-version-case-runtime *clause clause...* *Special Form*

Conditionalize microcode for the various hardware versions. This is available both as a micro—for use inside microcode—and as a Lisp special form (really a macro)—for use as a top-level form. Each *clause* takes the form

(*selector code*)

selector is a parenthesized list of machine version symbols, or is the symbol **otherwise**. See page 6 for the valid machine version symbols. A clause matches if the intersection of its *selector* list and the value of ***machine-version-list** is non-empty. If no **otherwise** clause is present and no specific clause matches (i.e. no code for the current machine version is present), an error is signalled.

code is microcode or a Lisp form, depending on whether this is the **machine-version-case** micro or one of the two Lisp special forms (macros).

The difference between **machine-version-case** and **machine-version-case-runtime**, as Lisp special forms, is that the former does the selection at macro-expansion time, while the latter expands into code to do the selection at run time. Typically **machine-version-case** is used at top level and the clauses contain forms that must be seen by the compiler. On the other hand **machine-version-case-runtime** is typically used inside of a Lisp function, a Lisp macro, or the Lisp-code body of a micro. We want to select the machine version at the time this Lisp code is running as part of a microcode compilation, not at the time the Lisp code is compiled.

4. Defining Forms

defucode *name body...* *Special Form*

Define a microcode routine that is named *name*. The *body* forms are implicitly combined with **sequential**. The microcode routine may be reached by a jump, a subroutine call, or a trap, using *name*.

defucode-at-loc *name loc body...* *Special Form*

This is like **defucode**, but requires that the microcode be stored at a particular address. *loc* is either a number or a list of numbers. The first microinstruction of the body will be stored at that location, or at all of those locations. **defucode-at-loc** is used to set up things like trap handlers whose addresses are known by the hardware.

definst *name attributes body...* *Special Form*

Define the microcode routine to execute a particular macroinstruction. *name* is the name of the macroinstruction. *attributes* is either a list whose first element is the format of the macroinstruction and whose remaining elements are its other attributes, or a symbol which is the format, in the common case where there are no other attributes. The format and attributes are checked against the Opdef file. Formats and attributes are described below (page 21).

definst is essentially **defucode-at-loc**, except the location is automatically computed by looking up *name* in the opcode table. You must explicitly put **(next-instruction)** at the end of the microcode if it is needed.

definst1 *name attributes body...* *Special Form*

A version of **definst** for macroinstructions that can be executed in a single machine cycle. The *body* forms are combined with **parallel** rather than **sequential**, and **(next-instruction)** is automatically appended to the body.

defareg-at-loc *name location &optional initial-value simulator-initial-value* *Special Form*

Define *name* to be the word in A-memory at address *location*. If *initial-value* is supplied, it is a Lisp expression to compute a number to be stored there. *simulator-initial-value* defaults to *initial-value* but allows a different value to be stored when using the simulator (usually it is conceptually the same value but is computed in a different way.)

defareg *name &optional initial-value simulator-initial-value* *Special Form*

Like **defareg-at-loc** but the system chooses the location. If *name* has been previously defined at a specific location, then the same location is used; this is useful because the Sysdf1 file defines a number of A-memory variables at specific locations that are used for communication between microcode and Lisp code. If *name* has not been previously defined at a specific location, a location is assigned from a free pool set up by **reserve-scratchpad-memory**.

defbreg-at-loc *name location &optional initial-value simulator-initial-value* *Special Form*
Like **defareg-at-loc** but for B memory.

defbreg *name location &optional initial-value simulator-initial-value* *Special Form*
Like **defareg** but for B memory.

define-b-temps *name name...* *Special Form*
Define a set of "temporary" B-memory registers. All sets of B-temps are stored in the same actual memory locations; thus one subroutine that uses B-temps should not call another subroutine that also does, and microcode that runs in tasks other than the emulator task should not use B-temps at all. The B-temps use some of the B-memory locations that can be written simultaneously with writing A-memory. The registers named **b-temp, b-temp-2, b-temp-3** use these B-memory locations also, but are guaranteed not to overlap with any registers allocated by **define-b-temps.**

By convention trap handlers that do not pclsr out, but just return to the trapped microcode, do not use any B-temps of either sort. Thus there is no need to worry about these implicitly-called subroutines clobbering their callers' temporaries. These traps are map misses and invisible-pointer traps.

reserve-scratchpad-memory *first-a last-a &optional first-b last-b* *Special Form*
Establish an area of A-memory, and optionally of B-memory, in which variables are to be allocated by **defareg** and **defbreg**. *last-a* and *last-b* are exclusive upper bounds.

A **reserve-scratchpad-memory** form should be put at the front of each microcode file. This kludge is necessary because locations have to be assigned at compile time (rather than when the microcode is linked) for the sake of the simulator.

defmicro *name args body...* *Special Form*
This is much like the Lisp **defmacro,** but defines a micro. The last *body* form should evaluate to a microcode form (a micro invocation or a primitive.) *Note well*: the *body* is not microcode; it is Lisp code that constructs microcode. Use backquote.

args may include the keywords **&optional, &rest, &body,** and **&aux**. The **defmacro** feature that *args* may include more general patterns, not just variables, is *not* supported currently. Optional arguments may have default values (which are Lisp forms to be evaluated, if the argument was not supplied, to produce a piece of microcode to use as the argument).

defatomicro *name expansion* *Special Form*
Define *name* to expand, when it appears by itself as a microcode expression, into *expansion*. Note that *expansion* is a microcode expression, *not* a Lisp form to be evaluated to produce a microcode expression.
[Perhaps I should change this!]

defatomic-byte-field *name byte-specifier register* *Special Form*
Define *name* to be an atomicro that expands into a datum representing a byte of *register*, another datum. The byte is specified by *byte-specifier*, which is either a symbol, or a list of *n-bits* and *bits-over*. A symbol must be the name of a byte defined in the Sysdef file. *n-bits* is the width of the byte in bits. *bits-over* is the position of the byte in bits from the right-hand end of the word (in other words it is the bit number of the least-significant bit in the byte).

def-byte-field *name byte-specifier var* *Special Form*
Define *name* to be a micro that takes an operand as its argument and expands into a datum representing a byte of that operand. *byte-specifier* is the same as in **defatomic-byte-field**. *var* is the dummy variable to be bound to the operand.

associate-dispatch-cues *field-name enumerated-type-name* *Special Form*
Declare that the byte field named *field-name* contains values of an enumerated type defined by **(defenumerated** *enumerated-type-name* **...)** in the Sysdef file. A dispatch (see **dispatch-after-next**, page 25) on the field will allow the symbolic names of the enumerated type values to be used as dispatch cues.

define-enumerated-value-constants *enumerated-type-name* *Special Form*
Declare an enumerated type defined by **(defenumerated** *enumerated-type-name* **...)** in the Sysdef file. Each symbolic value of this type is defined to be an atomicro that expands into a B-memory constant containing the numeric code for that value. This allows symbolic values to be deposited into fields in data structures.

define-storage-word-offset-constants *defstorage-type-name* *Special Form*
Make available to the microcode the symbolic names for the words in a system data structure defined in the Sysdef file with **defstorage**. Each word offset is defined to be an atomicro that expands into a B-memory constant containing the numeric value.

define-sysconstant *name* *Special Form*
Define *name* to be an atomicro that expands into a B-memory constant containing the value of that system constant (defined with **defsysconstant** in the Sysdef file).

There are also internal defining forms, for defining various specialized micros such as those that control the ALU. Presumably all possible uses of these defining forms have already been written, so they don't need to be documented here.

4.1 Macroinstruction Attributes

The following are the macroinstruction formats currently allowed:

unsigned-immediate-operand
The instruction includes an 8-bit immediate constant, which is unsigned. The atomicro to pick up the operand is **macro-unsigned-immediate**.

signed-immediate-operand
The instruction includes an 8-bit immediate constant, which is a 2's-complement signed number. The atomicro to pick up the operand is **macro-signed-immediate**.

10-bit-immediate-operand
The instruction includes a 10-bit immediate constant; the extra two bits are taken out of the opcode. This format is used for certain byte-manipulation instructions only.

address-operand
The instruction addresses the current stack frame; one bit selects between a positive 7-bit displacement from frame-pointer or a negative-or-zero 7-bit displacement from stack-pointer. The atomicro to pick up the operand is **address-operand.**

no-operand
The instruction has no direct operand (usually some operands will be passed on the stack).

quick-external-call
The instruction includes an 8-bit unsigned immediate constant to be interpreted as an index in the system-wide table of quick-external functions.

constant-operand
The instruction includes an 8-bit unsigned immediate constant that is a negative index into the constants table of the current function.

indirect-operand
The instruction includes an 8-bit unsigned immediate constant that is a negative index into the constants table of the current function. The addressed word contains a locative pointer to the value cell or function cell whose contents are the operand.

unsigned-pc-relative
The instruction includes an 8-bit unsigned immediate constant that is a PC-offset (see the **pc-add** micro, page 53) to be used for branching.

signed-pc-relative
The instruction includes an 8-bit signed immediate constant that is a PC-offset (see the **pc-add** micro, page 53) to be used for branching.

constant-pc-relative
Identical to **constant-operand** except that the addressed constant is to be used as an offset from the PC for branching. (See the **pc-add** micro, page 53.)

After the format in a **definst,** any number of attributes may be specified. The following are the currently-defined attributes:

needs-stack
The **top-of-stack** register must be valid when this instruction is entered.

smashes-stack
This instruction leaves the **top-of-stack** register invalid.

branch
This instruction branches unconditionally. It must have **signed-pc-relative** format. This attribute causes the IFU to assume that the next macroinstruction will come from the branch target address.

branch-if
This instruction branches if the datapath condition is true. It must have **signed-pc-relative** format. See the **ifu-branch** micro, page 55.)

branch-if-not
: This instruction branches if the datapath condition is false. It must have **signed-pc-relative** format.

stop-ifu
: Stop the IFU prefetcher while this instruction is executing. This attribute is used with any instruction that is likely to branch and does not have one of the above three attributes. However, **stop-ifu** cannot be used with an instruction in **no-operand** format.

The following attributes are used with **defopcode** in the OPDEF file, but are not used with **definst** in the microcode.

(function *name n-arguments n-values***)**
: This instruction implements the Lisp function named *name* when called with *n-arguments* arguments. *n-values* values are returned on the stack. *n-values* may be omitted and defaults to 1; 0 is commonly specified for functions mainly used for their side-effects. The arguments are passed on the stack except that if the format is not **no-operand** then the operand is the last argument. Multiple instructions may have **function** attributes for the same function; the compiler will choose the appropriate instruction in context.

(operand *type-of-operand***)**
: Additional information about the 8-bit immediate operand, used by the disassembler to print it in a nicer format than just a number. The current list of operand types is:

data-type	An immediate data type code, as used by **%make-pointer**.
byte-pointer	An immediate byte pointer, as used by **ldb**.
argument-number	The sequence number of an argument; 0 is the first argument. This is used by the function-entry instructions.
instance-variable	A reference to an instance variable (mapped or unmapped).

5. Flow of Control

Several of these micros use the concept of *normal successor*. The normal successor of a microinstruction is that microinstruction which is executed immediately afterwards, in the absence of any flow-of-control micros. Only microinstructions embedded in sequences have normal successors (note that **defucode** implicitly wraps **sequential** around its body, thus all microinstructions in the body except the last have a normal successor).

5.1 Jumps and Subroutines

The *ucode* and *return-to* arguments to the following micros are normally names of pieces of microcode, typically defined with **defucode**. It is actually permissible to specify an unnamed statement (or sequence) as one of these arguments.

jump *ucode* *Micro*
Take the next microinstruction from the routine named *ucode*.

call *ucode* *Micro*
Take the next microinstruction from the routine named *ucode* and save as the subroutine return address the normal successor of the current microinstruction.

return *Micro*
Take the next microinstruction from the saved subroutine return address, and pop the subroutine return stack. Each task has 16 stack locations.

next-instruction *Micro*
Take the next microinstruction from the address supplied by the Instruction Fetch Unit. The current microinstruction is the last to be executed on behalf of the current macroinstruction; the next microinstruction will either start the next macroinstruction, handle a sequence break, help the IFU with an instruction fetch, or idle waiting for the IFU to become ready.

In the hardware **next-instruction** and **return** are identical operations; when the outermost subroutine in the emulator task returns, the hardware does a next-instruction operation. The two names for this operation are to make the microprogram more readable. It is entirely legal to call a macroinstruction-execution microroutine as a subroutine.

call-and-return-to *ucode return-to* *Micro*
Take the next microinstruction from the routine named *ucode*, and save *return-to* as the subroutine return address.

call-and-dispatch-upon-return *ucode* *Micro*
A combination of **call** and **take-dispatch** (see page 26). The subroutine's return address is made to be the dispatch set up in the previous microinstruction. In the hardware this is the same as **call**; a separate name is provided to make the microprogram easier to read, to defeat error checking in the microcode linker, and for the benefit of the simulator.

Note that **call** may be executed in parallel with **return** or **next-instruction**. This is quivalent to **jump**. This can be useful when the **call** results from the expansion of one micro d the **return** from the expansion of another.

5.2 Conditionals

if *predicate true false* *Micro*

Test the *predicate*; if it is true, take the next microinstruction from *true*, otherwise take the next microinstruction from *false*. The available predicate micros are described in section 6.3, page 37.

Each clause (*true* or *false*) may be a microcode expression, the form **(goto** *tag***)** which means the microcode routine named *tag*, or the form **(drop-through)** which means the normal successor of the current microinstruction. If a clause is a microcode expression, its normal successor is the **if**'s normal successor, i.e. it rejoins the normal flow of control.

Using **(jump** *tag***)** as a clause is equivalent to **(goto** *tag***)** except that it is one cycle slower, because it generates a microinstruction that does nothing except a **jump**, as opposed to **goto**, which arranges to transfer control directly to the named routine (in some cases this may involve making a copy of that routine; the linker takes care of this).

Compare **if** with **trap-if** (page 26).

call-select *condition true-subroutine false-subroutine* *Micro*

A combination of **if** and **call**. The *condition* is tested and in the next cycle control passes to *true-subroutine* if it was true or *false-subroutine* if it was false. In either case a return address is pushed on the microcode subroutine stack.

call-select-and-return-to *condition true-subroutine false-subroutine return-to* *Micro*

A combination of **call-select** with **call-and-return-to**.

call-and-return-skip *ucode normal-return skip-return* *Micro*

Call *ucode* and allow it to select between two microcode locations when it returns.

return-skip *predicate* *Micro*

Return from a microcode subroutine called by **call-and-return-skip**. If *predicate* is true, control is returned to *normal-return*, otherwise control is returned to *skip-return*. If the sense of *predicate* is inverted in the hardware, an error is signalled because the caller cannot know when to interchange the two returns. In this case you must use **(not** *predicate***)** and change the caller.

5.3 Dispatching

dispatch-after-next *field clauses...* *Micro*

Select one of the *clauses* according to the value of *field*. The current microinstruction's immediate successor may then use the **take-dispatch** micro to transfer control to the selected clause. Note that **dispatch-after-next** and **if** may be used simultaneously, which provides a way to make the taking of the dispatch optional.

The car of a clause, called its *cue*, specifies the condition under which that clause will be selected. This can be a list of symbolic or numeric values for *field*, or the special symbol **otherwise**. The cdr of a clause is a list of microcode expressions; **sequential** is implicitly wrapped around them. As a special case, **(goto** *tag***)** is allowed in dispatch clauses; it works the same way as in **if**. **(drop-through)** is not allowed; its meaning is unclear because of the "after next" nature of dispatching.

Symbolic *field* values that appear in the car of a dispatch clause are defined with the **associate-dispatch-cues** defining form (see page 21).

field selects a field of up to 4 bits in the data path, thus dispatches may select among up to 16 possibilities. Normally the byte-extraction hardware is used to select the field (see the **ldb** micro, page 41). *field* may also be an invocation of the **cdr-code** micro, allowing a 4-way dispatch on the cdr code of an Abus source.

The machine also provides several special-case dispatches, which were added to speed up various critical operations. The **dispatch-after-next** micro recognizes these automatically; they need not be programmed specially. When one of the special-case fields of the Abus is dispatched upon, the byte-extraction hardware is left free, allowing a different byte to be operated on simultaneously, or avoiding the usurpation of microinstruction fields used to control both byte extraction and other things. See the hardware documentation for a list of the special-case fields.

take-dispatch *Micro*

dispatch-after-next only takes effect if **take-dispatch** is executed in the following cycle. In the hardware, dispatching works by storing the address of the selected clause in the NPC register, and **take-dispatch** means to take the next microinstruction from the address in the NPC.

dispatch-after-this *field microinstruction clauses...* *Micro*

Dispatches on *field* into *clauses*, and executes *microinstruction* during the one-cycle dispatch delay. **take-dispatch** is automatically placed in parallel with *microinstruction*.

long-dispatch *address* *Micro*

Jump to the control-memory address given by the low 14 bits of the datum, *address*. The address is stored in the NPC register and the jump only happens if **take-dispatch** is used in the following cycle. **long-dispatch** allows dispatches on more than 4 bits to be done, but more slowly. Currently the dispatch clauses must be defined with **defucode-at-loc**.

5.4 Traps

trap-if *predicate true* *Micro*

If *predicate* is true, take the next microinstruction from *true*; otherwise take the next microinstruction normally (either from the normal successor or under the control of any other flow-of-control micros done in parallel). The *true* clause is exactly like an **if** clause (of course **(drop-through)** is almost useless here).

The difference between **trap-if** and **if** is fourfold: It is legal to do **trap-if** in parallel with other flow-of-control micros, most commonly **next-instruction**. If the *predicate* is true, the side-effects of the current microinstruction are suppressed. If the trap is taken, the current microinstruction takes twice as long to execute as it normally would. If the trap is taken, the address of the trapped microinstruction is placed in the NPC register. This has two effects on the trap handler: It may not call a subroutine in its first microinstruction, because the return address that would normally come from the NPC register is not present.

The trap handler may retain the address of the trapped microinstruction by performing a **trap-save** in its first microinstruction.

The **trap-if** type of trapping does not affect the microcode subroutine stack and does not save the NPC register.

Traps are used to program exception cases while allowing the normal case to run at maximum speed, with no overhead for checking for the exception.

signal-error *error-code...* *Micro*
Abort the current macroinstruction and exit to the error handler, passing the symbolic error message specified by the *error-code* arguments.

signal-error-no-restore-stack *error-code...* *Micro*
Identical to **signal-error** except that the stack-pointer remains at its current setting. **signal-error** would restore it to its value at the start of the macroinstruction.

error-if *condition error-code...* *Micro*
If *condition* is true, trap to the error handler, passing the symbolic error message specified by the *error-code* arguments. This is equivalent to

```
    (trap-if condition (signal-error error-code...))
```

but saves a control-memory location.

error-no-restore-stack *condition error-code...* *Micro*
Identical to **error-if** except that the stack-pointer remains at its current setting. **error-if** would restore it to its value at the start of the macroinstruction.

check-arg-type *location datum type1 type2...* *Micro*
Trap if the data type of *datum* is not one of the types listed. *datum* must be an Abus source. No trap handler is specified; the trap always goes to a fixed location (trap-0 in the type map is used). This micro is typically used by instructions and subroutines to check the types of their arguments. The trap-0 microcode normally passes control to the Lisp error handler.

location is a symbolic specification of where *datum* came from. It is passed along in the error message and used by the error handler to format the error report, to locate the offending datum, and to replace it with a new value if the instruction is retried. In many cases an error is detected by a subroutine used in common by several instructions that get their arguments from different places. The *location* provides a symbolic specification that the error handler uses, in combination with the particular instruction that was being executed, to find the physical location of *datum*.

location should be one of the following:

a number	One of the arguments to the function implemented by the instruction; 0 specifies the first argument.
nil	One of the arguments to the function implemented by the instruction, but it is not specified which one. The error handler will test the arguments and signal an error for the first one whose data type does not match the types specified.

any	Any one of the arguments to a function that takes several arguments all of the same type. The error handler may signal an error complaining about more than one of the arguments simultaneously.
array	The array argument to an array-manipulating instruction. Whether this is the first or second argument depends on the instruction.
subscript	One (or more) of the subscript arguments to an array-manipulating instruction.
top-of-stack	The top value on the stack (i.e. the last argument). This is used by the **funcall** instructions, for example.
rest-arg	The rest-argument being passed by a **lexpr-funcall** instruction.
instance	The instance argument to an instance-variable accessing instruction. This can be either an explicit argument or an implicit one (**self** in the current frame), depending on the particular instruction.
self-mapping-table	The instance-variable mapping table in the current frame. This is an implicit argument to the instance-variable accessing instructions.

More *location* codes are likely to be added in the future.

check-data-type *datum type1 type2...* *Micro*

Trap if the data type of *datum* is not one of the types listed. *datum* must be an Abus source. No trap handler is specified; the trap always goes to a fixed location (trap-0 in the type map is used). **check-data-type** is the same as **check-arg-type** with a *location* of **nil**.

check-data-type-with-error-entry *datum error-code type1 type2...* *Micro*

Identical to **check-data-type** except the *error-code* may be specified.

check-internal-type *location datum type1 type2...* *Micro*

This micro is used to complain about data-type problems found in internal data structure (rather than in arguments to an instruction). It is conceptually similar to **check-arg-type** but is treated differently by the microcode error handler and the Debugger.

Trap if the data type of *datum* is not one of the types listed. *location* is a symbolic specification of where *datum* came from:

return-pc	The current frame's return PC (PC of its caller).
previous-frame	The current frame's previous-frame pointer (frame of its caller).
previous-top	The current frame's previous-stack-top pointer (the highest location in the stack in use by the caller).

instance-size
instance-binding
instance-hash-table
instance-hash-table-entry
Various attributes of some flavor instance being operated upon by the instruction. See **instance** under **check-arg-type** above.

array-length
array-indirect-pointer
array-index-offset
array-dimension
Various fields in the prefix of an array being operated upon by the instruction.

array-word A word containing one or more elements of an array of bytes.

destination-offset
source-offset
Internal variables of **bitblt**.

data-type-trap *datum trap-name type1 type2...* *Micro*
Trap if the data type of *datum* is not one of the types listed. *datum* must be an Abus source. This is the same as **check-data-type** except that you may specify which of the type map traps to use (trap-0, trap-1, trap-2, or trap-3) and no automatic error-table entry is made.

The following micros are essentially special cases of **trap-if** usually used in generic arithmetic macroinstructions.

check-fixnum-2args *a-opnd b-opnd clauses...* *Micro*
a-opnd is an Abus operand and *b-opnd* is a Bbus operand. The data types of these two operands are checked. If both are fixnums, execution proceeds normally. If either is not a number, a trap-0 to signal an error occurs. If both are numbers, but they are not both fixnums, one of the clauses is selected as the trap handler. The clauses look like dispatch clauses. If only an **otherwise** clause is present, no dispatch occurs (i.e. memory is not wasted for a dispatch block of 16 identical microinstructions). If no clauses at all are specified, this micro simply signals an error unless both operands are fixnums. The valid dispatch cues are as follows:

flonum-flonum	Both operands are flonums (immediate floating-point numbers).
fixnum-flonum	*a-opnd* is a fixnum and *b-opnd* is a flonum.
flonum-fixnum	*a-opnd* is a flonum and *b-opnd* is a fixnum.
extnum-extnum	Both operands are extended numbers (anything other than fixnum or flonum, including bignums, rationals, extended-precision floating-point, complex, or what have you.)

fixnum-extnum	
extnum-fixnum	
flonum-extnum	
extnum-flonum	These are analogous.
fixnum-fixnum	Both operands are fixnums, but a trap occurred anyway. This happens if overflow checking is enabled and an overflow occurs (see **add-checking-overflow,** page 36).

If *b-opnd* is an extended number, it does not get fully type-checked; the trap handler should check the type again with **check-arg-type**. This is because the hardware only has full data type checking capability on the Abus. It only checks *b-opnd* for being a fixnum; anything not a fixnum will trap and dispatch. Thus it is possible for the **otherwise** clause to be reached with *b-opnd* having a random data type, and for an *xxx*-**extnum** clause to be reached with *b-opnd* having something whose data type is not **dtp-extended-number**. In an *xxx*-**flonum** clause, *b-opnd* is guaranteed to be a flonum.

check-fixnum-1arg-a *a-opnd clauses...* *Micro*

Analogous to **check-fixnum-2args** when there is only one operand and it is on the Abus. If *clauses* are used to dispatch into a set of trap handlers, the dispatch hardware will still think it is dispatching on two arguments; write the dispatch cues in the clauses appropriately. Typically you put a fixnum on the Bbus and then use dispatch cues such as **fixnum-fixnum** and **flonum-fixnum.**

[illegible]eck-fixnum-1arg-b *b-opnd clauses...* *Micro*

Analogous to **check-fixnum-2args** when there is only one operand and it is on the Bbus. If dispatching into a set of trap handlers is used, the dispatch hardware will still think it is dispatching on two arguments; write the dispatch cues in the clauses appropriately.

Beware! The "condition" bit in the type map is spuriously enabled to cause a trap. Thus **check-fixnum-1arg-b** should not be paralleled with the **data-type?** predicate, nor with **transport** or **store-contents.**

check-fixnum-b *b-opnd* &optional *trap-handler* *Micro*

Trap if *b-opnd* is not a fixnum; it must be a Bbus operand. *trap-handler* defaults to signal a data-type error; it may be specified as a microcode expression to handle the trap, or as **nil** to allow the trap handler to be supplied by something else in the instruction (typically a **trap-if**). This latter feature is used by array referencing, which simultaneously checks that the subscript is within bounds and that it is a fixnum—this would normally be done with **trap-if** and **check-data-type,** but **check-data-type** requires its operand to be on the Abus.

Beware! The "condition" bit in the type map is spuriously enabled to cause a trap. Thus **check-fixnum-b** should not be paralleled with the **data-type?** predicate, nor with **transport** or **store-contents.**

check-data-type-and-dispatch (*a-opnd types...*) *clauses...* *Micro*

If the data type of *a-opnd* is one of the *types* named, take a trap. The trap handler is obtained by dispatching into *clauses* as with **check-fixnum-1arg-a**. Note that the trap occurs if the operand is of the specified type(s), not if it fails to be of the specified type(s)—this is the opposite of **check-data-type**. This micro is probably used only by the **eql** function (it is a different combination of the same primitives that the other micros above use).

5.5 Delay

nop *Micro*

A microinstruction that does nothing. This is useful when an explicit delay is required (usually in connection with main memory).

waiting-for-memory *Micro*

This is the same as **nop**, but is often used to clarify the intention of the microcode when a delay is necessary during the active cycle of a memory reference. This micro should only be used when the delay is actually waiting for memory, since in the future it may be changed to do nothing on IFU machines where the hardware can provide the delay automatically.

declare-speed *speed* *Micro*

Declare a speed restriction; the microinstruction will be at least as slow as specified by *speed*. If two speed declarations are done in parallel, the slower of the two is selected; if **slow-first-half** and **slow-second-half** are done in parallel, both halves are made slow. Usually the micros will supply appropriate speed declarations; explicit use of **declare-speed** is only necessary in odd situations.

The possible symbols for *speed* are:

slow	Slower than normal, but it doesn't matter whether the extra time is in the first half or the second half.
slow-first-half	Extra time (45 ns) in the first half.
slow-second-half	Extra time (30 ns) in the second half.
very-slow	As slow as possible (75 ns extra).

5.6 Trap Handlers

There are two kinds of traps, and the trap handler is entered slightly differently depending on which kind occurs.

Low-level traps trap to fixed addresses and save both NPC and CPC, permitting the trapped microinstruction to be retried. NPC is automatically pushed on the microcode subroutine stack by the hardware. CPC is saved in NPC where it is available to be saved by the trap handler (see the **trap-save** micro below.)

The rest of the traps, such as those generated by **trap-if** and **check-fixnum-2args**, trap to a handler whose address is freely specifiable, and do not save NPC. Thus the trapped microinstruction cannot be retried. However its address is still available to be saved by **trap-save**.

Because of the use of NPC to save CPC at the time of a trap, the first microinstruction of a trap handler may not call a subroutine (the return address for a subroutine call comes from NPC). The microcode linker will complain if there is an attempt to do this.

trap-save *Micro*
Finish the state save initiated by the trapping hardware, by pushing NPC onto the stack. NPC contains the original CPC, i.e. the address of the microinstruction that trapped. NPC at the time of the trap has already been pushed onto the stack (if this is a low-level trap). This micro should be included in the first microinstruction of any trap handler that may retry the trapped microinstruction, or that needs to know where it came from.

trap-no-save *Micro*
This micro is used instead of **trap-save** in the first microinstruction of a trap handler that does not care where it came from or is not for a low-level trap. This applies to destinations of the **trap-if** micro, for example. **trap-no-save** ensures that adequate time is available for the hardware to recover from the trap operation. (In the **proto** hardware it also adjusted the stack, to compensate for the fact that all traps behaved like low-level traps, saving the NPC on the stack.)

trap-restore *cycle-1 cycle-2* *Micro*
Return from a trap handler and retry the microinstruction that trapped. Since this takes two cycles, **trap-restore** takes two arguments, which are microcode to be executed in parallel with the restore. First the saved CPC is popped into NPC. Then the saved NPC is popped into NPC and simultaneously CPC is loaded from NPC.

trap-restore-1 *Micro*
Perform the first cycle of **trap-restore**. The effect of **trap-restore-1** can be undone by **trap-save** executed in the immediately-following cycle; this is useful when conditionally returning from a trap handler.

trap-restore-2 *Micro*
Perform the second cycle of **trap-restore**.

6. Machine Operations

This chapter documents a host of micros that provide access to the various features of the machine. Many of these micros deal with data manipulation and hence expand into a datum rather than a statement; in other words they are used in an applicative style rather than the imperative style of most of the micros described above.

6.1 A and B memory

a–constant *value* *Micro*

value is a Lisp form to be evaluated; its value must be an integer. The **a–constant** micro expands into a datum that is an A-memory location containing that integer.

b–constant *value* *Micro*

value is a Lisp form to be evaluated; its value must be an integer. The **b–constant** micro expands into a datum that is a B-memory location containing that integer.

amem *address* *Micro*

A datum that is an A-memory location as specified by *address*, which may be any of the following:

location	An integer between 0 and 7777 is an absolute address. Normally **defareg** is used to give symbolic names to A-memory locations, rather than using explicit numbers.
(frame–pointer *offset***)** **(stack–pointer** *offset***)** **(xbas** *offset***)**	The specified base register is added to the specified offset (an 8-bit signed integer) to compute the address.
(macrocode)	The address field of the current macroinstruction specifies a base register and an offset. Normally the **address–operand** atomicro is used for this.

All forms of address except the first compute a 10-bit address and take the high 2 bits from the stack base in the data path control register (see page 55). Thus they can only access the current stack buffer, not all of A memory.

stack–pointer *Atomicro*

The stack-pointer register. This is a 28-bit up/down counting register, the low 10 bits of which also serve as a base register for A-memory addresses.

frame–pointer *Atomicro*

The frame-pointer register. This is a 28-bit register, the low 10 bits of which also serve as a base register for A-memory addresses.

xbas *Atomicro*
The extra base register (this can only be written, not read). This is a 10-bit base register for A-memory addresses.

increment-stack-pointer *Micro*
Add one to the stack-pointer.

decrement-stack-pointer *Micro*
Subtract one from the stack-pointer.

stack-adjustment *Atomicro*
A 4-bit register that increments and decrements in parallel with the stack-pointer, and is zeroed at the start of each macroinstruction. This makes it possible to restore the stack-pointer when aborting a trapped macroinstruction; see section 7.5, page 66.

clear-stack-adjustment *Micro*
Zero the **stack-adjustment** register. This is used when a complex macroinstruction reaches an intermediate point to which it can be aborted. Sometimes the first-part-done flag will be set at the same time as **clear-stack-adjustment** is done.

6.2 Arithmetic/Logic Unit

A variety of micros are provided to perform arithmetic and logical operations on 1, 2, or 3 ands (in the 3-operand case, the third operand must be the constant 1; thus x+y+1 and x-y-1 may be computed.) The compiler allows more flexibility about the sources of these operands than is usual. The hardware takes one ALU operand from the Xbus and the other from the Ybus (via the shifter and AluB). Usually Xbus comes from Abus and Ybus comes from Bbus, however the reverse is also possible and in addition Xbus and Ybus each have a special source (Xbus may come from the multiplier, Ybus may come from the "crocks".) See the hardware block diagram on the next page.

The compiler allows either operand to a 2-operand ALU micro to come from either bus, provided only that the two operands come from different busses so that the operation is physically realizable. The exception is subtraction, for which the hardware is deficient: the minuend must come from the Xbus and the subtrahend from the AluB; thus it is not possible to extract a byte and subtract something from it (one could, however, add a negative constant to it.)

The compiler allows the operand to a 1-operand ALU micro to come from either bus; in the cases where the hardware is deficient the compiler will turn it into the 2-operand case, supplying a constant 0 operand on the other bus.

Thus usually the programmer need only be careful to avoid trying to do an ALU operation on two Abus operands or two Bbus operands; the other vagaries of the hardware will be hidden by the compiler. When necessary, the routing of data through the busses may be controlled explicitly; see section 6.6, page 43.

Note that ALU operations are on 32 bits. The output-tagging feature (see section 6.8, page 44) must be used to add a data-type tag.

<LMDP>DPBLK.PLT

When no ALU operation is being performed, but a datum is simply being moved from one place to another, the compiler will generate the appropriate microinstruction to pass the datum unchanged through the ALU and to pass the tag around the ALU; thus all 36 bits will be moved.

1+ *opnd* *Micro*
Add 1 to *opnd*.

1- *opnd* *Micro*
Subtract 1 from *opnd*.

+ *opnd opnd* &optional *one* *Micro*
Take the sum of two operands. If three operands are used, the third (*one*) must be the number 1.

- *opnd* &optional *opnd one* *Micro*
With one operand, take the 2's-complement (negation). With two operands, take the difference. With three operands, the third must be the number 1 and the result is the difference, minus one.

commutative-diff *opnd opnd* &optional *one* *Micro*
The same as - except that the compiler is permitted to interchange the operands, reversing the sign of the result. This is normally used only when all you care about the result is whether or not it is zero.

land *opnd opnd* *Micro*
Bit-by-bit logical and.

logior *opnd opnd* *Micro*
Bit-by-bit logical inclusive or.

logxor *opnd opnd* *Micro*
Bit-by-bit logical exclusive or.

lognand *opnd opnd* *Micro*
The complement of **logand**.

andc2 *opnd1 opnd2* *Micro*
logand with *opnd2* complemented.

This micro currently requires *opnd2* to come from Ybus, but there is no good reason for that restriction.

add-checking-overflow *opnd opnd* *Micro*
+ with overflow checking enabled. The 3-operand case is not allowed because the hardware cannot handle it.

sub-checking-overflow *opnd opnd* *Micro*
- with overflow checking enabled. The 1-operand and 3-operand cases are not allowed because the hardware cannot handle them. (The 1-operand case may be simulated by using a constant 0 as the first operand.)

inc-checking-overflow *opnd* *Micro*
dec-checking-overflow *opnd* *Micro*

1+ and **1-** with overflow checking enabled. See **check-fixnum-2args** (page 29).

Unfortunately these two micros do not work because of a hardware bug. Use **add-checking-overflow** or **sub-checking-overflow** instead.

6.3 Predicates

The micros in this section expand into conditions that may be used with such micros as **if** and **trap-if**. Almost all of them use the ALU and have the same constraints (or lack of constraints) on their operands as the arithmetic and logical micros in the previous section.

not *predicate* *Micro*

Reverse the sense of *predicate*, which must expand into a **microcondition** primitive.

The following predicates operate on 28-bit unsigned numbers (virtual addresses):

equal-pointer *x y* *Micro*

True if the low 28 bits of *x* and *y* are equal.

not-equal-pointer *x y* *Micro*

True if the low 28 bits of *x* and *y* are not equal.

greater-pointer *x y* *Micro*

True if *x* is greater than *y* in the low 28 bits.

greater-or-equal-pointer *x y* *Micro*

True if *x* is greater than *y* in the low 28 bits, or they are equal.

lesser-pointer *x y* *Micro*

True if *x* is less than *y* in the low 28 bits.

lesser-or-equal-pointer *x y* *Micro*

True if *x* is less than *y* in the low 28 bits, or they are equal.

The following predicates operate on 32-bit signed 2's-complement numbers (fixnums):

equal-fixnum *x y* *Micro*

True if the low 32 bits of *x* and *y* are equal.

not-equal-fixnum *x y* *Micro*

True if the low 32 bits of *x* and *y* are not equal.

greater-fixnum *x y* *Micro*

True if *x* is strictly greater than *y* as a 32-bit 2's-complement number.

greater-or-equal-fixnum *x y* *Micro*
True if *x* is not less than *y* as a 32-bit 2's-complement number.

lesser-fixnum *x y* *Micro*
True if *x* is strictly less than *y* as a 32-bit 2's-complement number.

lesser-or-equal-fixnum *x y* *Micro*
True if *x* is not greater than *y* as a 32-bit 2's-complement number.

zero-fixnum *x* *Micro*
True if the low 32 bits of *x* are zero.

not-zero-fixnum *x* *Micro*
True if not all the low 32 bits of*mx*are zero.

minus-fixnum *x* *Micro*
True if bit 31 of *x* is 1 (i.e. *x* is negative as a 32-bit 2's-complement number).

minus-or-zero-fixnum *x* *Micro*
True if *x* is negative or zero as a 32-bit 2's-complement number. Beware! Due to a hardware deficiency, this is *false* for *setz* (the smallest negative number)!

plus-fixnum *x* *Micro*
True if *x* is strictly greater than zero as a 32-bit 2's-complement number. Beware! Due to a hardware deficiency, this is *true* for *setz* (the smallest negative number)!

plus-or-zero-fixnum *x* *Micro*
True if *x* is greater than or equal to zero as a 32-bit 2's-complement number, i.e. bit 31 of *x* is 0.

bit-test *x y* *Micro*
Like the **bit-test** Lisp function, this is true if there is some bit position (among the low 32 bits) in which *x* and *y* are both 1.

bit-test-pointer *x y* *Micro*
Like the **bit-test** Lisp function, this is true if there is some bit position (among the low 28 bits) in which *x* and *y* are both 1.

ldb-bit-test *opnd bit-number* *Micro*
True if the *bit-number*'th bit from the least-significant end of *opnd* is 1. *bit-number* is either a number between 0 and 31. or the symbol **byte-r** (see **ldb**, page 41).

bit *byte-field* *Micro*
byte-field must be a datum that is 1 bit wide. The condition is true if the bit is 1. **(bit** *x***)** is the same as **(zero-fixnum** *x***)**, when *x* is a 1-bit field, but leaves the ALU free and is a little faster.

field-bit *operand field-name* *Micro*

A condition that is true if a bit in *operand* is 1. *field-name* is a symbolic specification of a 1-bit-wide byte. It may be defined with **defatomic-byte-field** or **def-byte-field,** or with **defsysbyte** in the Sysdef file.

bottom-bit *operand* *Micro*

A condition that is true if bit 0 (the least significant bit) of *operand* is 1. This condition involves less possibility of a microinstruction field conflict than

```
(bit (ldb operand 1 0))
```

since it does not mask off the operand to a single bit.

all-ones *x* *Micro*

True if the low 32 bits of *x* are all 1 (*x* is -1 as a 32-bit 2's complement number).

ldb-field-ones *operand field-name* *Micro*

A condition that is true if every bit in the specified field of *operand* is 1. The condition is tested in the ALU, permitting *operand* to be simultaneously stored into B-memory. *field-name* is a symbolic specification of a byte. It may be defined with **defatomic-byte-field** or **def-byte-field,** or with **defsysbyte** in the Sysdef file.

The following predicates operate on 32-bit unsigned integers. There is no such data type in Lisp, but unsigned numbers are used internally in some parts of the microcode, such as floating point. Some of the predicates listed above (**equal-fixnum** for example) are equally meaningful for signed and unsigned integers.

greater-fixnum-unsigned *x y* *Micro*

True if *x* is greater than *y* as a 32-bit unsigned integer.

greater-or-equal-fixnum-unsigned *x y* *Micro*

True if *x* is greater than *y* as a 32-bit unsigned integer or they are equal.

lesser-fixnum-unsigned *x y* *Micro*

True if *x* is less than *y* as a 32-bit unsigned integer.

lesser-or-equal-fixnum-unsigned *x y* *Micro*

True if *x* is less than *y* as a 32-bit unsigned integer or they are equal.

The following predicates operate on typed pointers, which are 34 bits (either 2 bits of type and 32 bits of data or 6 bits of type and 28 bits of address).

equal-typed-pointer *x y* *Micro*

True if the low 34 bits of *x* and *y* are equal.

not-equal-typed-pointer *x y* *Micro*

True if the low 34 bits of *x* and *y* are not equal.

The following predicates are miscellaneous.

ybus-31 *Atomicro*
True if the sign bit of the Y bus is 1. This is used by the microcode for division, but probably is not useful for anything else. This predicate must be **parallel**'ed with something that puts data on the Y bus.

alu-carry *Atomicro*
True if there is a carry out of bit 31 of the ALU. This is useful when doing multiple-word integer arithmetic. This predicate must be **parallel**'ed with something that does an ALU operation.

data-type? *operand types...* *Micro*
True if *operand* (which must be an Abus source) has a data type whose name is one of the specified *types*.

not-data-type? *operand types...* *Micro*
True if *operand* (which must be an Abus source) has a data type whose name is not one of the specified *types*.

cdr-code? *operand code* *Micro*
True if *operand* (which must be an Abus source) has the specified cdr code. *code* may be either the name of a cdr code or a number from 0 to 3.

not-cdr-code? *operand code* *Micro*
True if *operand* (which must be an Abus source) does not have the specified cdr code. *code* may be either the name of a cdr code or a number from 0 to 3.

See also **odd-pc?** (page 53), **ibus-dev-cond** (page 56), and **sequence-break** (page 55).

6.4 Storing Results

assign *destination source* *Micro*
source is any datum, and *destination* is a datum that can be stored into. A statement is generated to store *source* into *destination*. **assign** knows how to store into all the memories and registers in the machine, and also knows how to store into byte fields in a register or memory location. Note however that **assign** is not usually used with main memory, because of garbage collector storage conventions; see **store-contents** (page 48).

obus *Atomicro*
A datum that stands for whatever is on the Obus (the output from the data path). This is useful shorthand when storing the result of the same computation into more than one place simultaneously.

increment *location* &optional *fixnum-p* *Micro*
Add one to *location* and store the result back into *location*. If *fixnum-p* is specified, the data type is forced to be **dtp-fix**; use this when storing into locations that are Lisp variables.

decrement *location* &optional *fixnum-p* *Micro*
Subtract one from *location* and store the result back into *location*. If *fixnum-p* is specified, the data type is forced to be **dtp-fix**; use this when storing into locations that are Lisp variables.

6.5 Shifter

ldb *operand n-bits bits-over* &optional *background* *Micro*
A datum that represents a byte extracted from *operand*. *n-bits* is the width of the byte. *bits-over* is the bit number of the least significant bit in the byte (in other words, the number of bits between the byte and the least-significant end of *operand*). If *background* is specified, it is a datum that supplies the bits of the result outside of the byte; normally these bits are 0. If *background* is the number 0, that is the same as no background.

n-bits is a number from 1 to 40, or the symbol **byte-s**, or the symbol **macro**. **byte-s** means that the **byte-s** register contains one less than the number of bits. **macro** means that the macroinstruction specifies the byte size.

bits-over is a number from 0 to 37, or the symbol **byte-r**, or the symbol **macro**. **byte-r** means that the **byte-r** register contains the number of bits of left rotation (40 minus *bits-over*). **macro** means that the macroinstruction specifies the left rotation.

Not all combinations of non-numeric values for *n-bits* and *bits-over* are supported by the hardware; the compiler will complain if you try to do something illegal.

ldb-field *operand field-name* &optional *background* *Micro*
ldb-field *operand field-name* *Special Form*
A datum that represents a byte extracted from *operand*. *field-name* is a symbolic specification of the byte. It may be defined with **defatomic-byte-field** or **def-byte-field**, or with **defsysbyte** in the Sysdef file.

ldb-field is also available as a Lisp macro; this is mainly of use in constants.

strange-ldb *operand n-bits bits-over* &optional *background* *Micro*
This is the same as **ldb** except with some error-checking turned off. This allows you to use bytes that cross the word boundary and exploit what the hardware does in this case. (The hardware acts as if it first rotates right by *bits-over* and then masks with a mask *n-bits* wide.)

dpb *operand n-bits bits-over background* *Micro*
A datum that represents the result of depositing the low bits of *operand* into a byte in *background*. *n-bits* is the width of the byte and *bits-over* is its position. *background* is either an operand or the number 0, which means that the bits in the result outside of the byte field should be 0.

n-bits is a number from 1 to 40, or the symbol **byte-s**, or the symbol **macro**. **byte-s** means that the **byte-s** register contains one less than the number of bits. **macro** means that the macroinstruction specifies the byte size.

bits-over is a number from 0 to 37, or the symbol **byte-r**, or the symbol **macro**. **byte-r** means that the **byte-r** register contains *bits-over* (the number of bits of left rotation). **macro** means that the macroinstruction specifies the byte position.

Not all combinations of non-numeric values for *n-bits* and *bits-over* are supported by the hardware; the compiler will complain if you try to do something illegal.

dpb-field *operand field-name background* *Micro*
dpb-field *operand field-name background* *Special Form*
A datum that represents a byte from *operand* deposited into *background*. *field-name* is a symbolic specification of the byte. It may be defined with **defatomic-byte-field** or **def-byte-field**, or with **defsysbyte** in the Sysdef file.

dpb-field is also available as a Lisp macro; this is mainly of use in constants.

rotate *operand amount* *Micro*
Rotate *operand* (as a 32-bit number) left by *amount* places. *amount* may be a number from 0 to 37 or the symbol **byte-r**.

byte-mask *ppss* *Special Form*
A Lisp function that converts a byte pointer to an integer containing 1 bits in the selected byte and 0 bits elsewhere. This function is useful in connection with the **a-constant** and **b-constant** micros. *ppss* may be a numeric byte specifier or the symbolic name of a byte field (see **ldb-field** above).

field-mask *field-name* *Special Form*
A Lisp macro that converts a symbolic byte field name to an integer containing 1 bits in the selected byte and 0 bits elsewhere. This is useful in connection with the **a-constant** and **b-constant** micros. See **ldb-field** above for further description of *field-name*.

byte-r *Atomicro*
A 5-bit register that can be used as a source of left-rotation for the byte hardware. **byte-r** can be written from the Obus in the usual manner, or the special statement
```
(assign byte-r array-index-shift-prom)
```
may be done in parallel with a **dispatch-after-next** to load **byte-r** with a function of the field being dispatched upon. This feature is provided specifically to speed up the accessing of packed arrays of bytes and is probably not generally useful.

byte-s *Atomicro*
A 5-bit register that can be used as a source of byte-size-minus-1 for the byte hardware.

complemented-sign-bit *operand* *Micro*
A 1-bit byte that is the complement of bit 31 of *operand*. The background is always 0. Thus the result is 0 if *operand* is negative, or 1 if *operand* is positive or zero.

ybus-crocks-1 *Atomicro*
ybus-crocks-2 *Atomicro*

Two words of data available as input to the shifter. Several special data sources in the machine may be accessed via

```
    (ldb ybus-crocks-n width position)
```

Refer to the hardware documentation (DPYSL2 print) to determine the possible sources and the values of *n*, *width*, and *position* for each source.

6.6 Routing Data

Sometimes it is necessary to specify explicitly which bus (X or Y) is used to bring data from Abus (or Bbus) into the ALU. This is because when two microinstructions are executed in parallel, the decision of which bus to use is made separately for each of the two microinstructions, during micro expansion, before they are placed in parallel. If the decision was made incompatibly, the **parallel** micro will not detect that it could have been made differently, but will simply report a bus-conflict error. The **via-xbus** and **via-ybus** micros may be used to declare the routing of data explicitly to avoid this problem.

The default routing decision, when there is a free choice, is to route Abus data via the Xbus and Bbus data via the Ybus, which avoids the need for explicit declaration of routing in the most cases.

via-xbus *datum* *Micro*
datum on the Xbus.

via-ybus *datum* *Micro*
datum on the Ybus.

6.7 Multiplier

write-mpy-x *source* &optional *signed* *Micro*
Write the low 16 bits of *source* into the X register of the multiplier. If *signed* is non-**nil**, bit 15 of *source* is taken to be a 2's-complement sign bit; otherwise the X register is unsigned. *source* comes in through the Xbus.

write-mpy-y-from-high *source* &optional *signed* *Micro*
Write bits 31-16 (the high 16 bits of a fixnum) of *source* into the Y register of the multiplier. If *signed* is non-**nil**, bit 31 of *source* is a 2's-complement sign bit; otherwise the Y register is unsigned. *source* comes in through the Ybus, thus it may be shifted (with **ldb**, **dpb**, or **rotate**) simultaneously. The multiplier sees the unshifted data.

mpy-product *Atomicro*
The 32-bit product of the X and Y registers. This is an unsigned fixnum if both X and Y were unsigned; otherwise it is a signed fixnum. **mpy-product** is read onto the Xbus. The product may be read in the immediately following microinstruction after loading one or both of the multiplier's input registers; the **mpy-product** atomicro includes the necessary timing specification.

6.8 Output Tagging

These micros control the tag fields of the output from the ALU. When an ALU operation is performed, the tag fields are indeterminate unless these micros are used. When the ALU is just used to pass an Abus or Bbus source, the tag fields come from that source.

set-cdr *operand cdr* *Micro*
A datum that represents *operand* with its cdr code set to *cdr*. *cdr* may be the symbolic name of a cdr code or a number from 0 to 3.

set-type *operand type* *Micro*
A datum that represents *operand* with its data type set to *type*, which must be the symbolic name of a data type. If *type* is **dtp-fix** or **dtp-float**, 32 bits of *operand* are used; otherwise only 28 bits of *operand* appear in the output.

merge-cdr *operand cdr-background* *Micro*
A 36-bit datum that consists of the cdr-code field of *cdr-background* and the type and pointer fields of *operand*.

merge-high-tag *operand tag-background* *Micro*
A 36-bit datum whose low 32 bits come from *operand* and whose high 4 bits are the high 4 bits of *tag-background*.

See section 7.4, page 65 for some other related micros.

6.9 Main Memory

Accessing memory requires interacting with virtual address mapping, with the garbage collector (since the garbage collector "watches" data pass between processor and main memory), and with invisible pointers.

"Main memory" actually is anything on the Lbus that behaves like memory. This includes main memory itself, TV memory, and the control registers of "memory-mapped" I/O devices. The A-memory in the datapath can also masquerade as main memory; the virtual address map can specify, instead of a physical address, one of the 16 physical pages of A memory.

Conceptually there are two registers, **vma** (which holds a virtual address) and **memory-data** (which holds data passing to or from memory). Actually the memory is addressed by physical addresses; the physical address to be referenced may be the result of mapping the contents of **vma** from virtual to physical, which is the usual case, or may come directly from the Abus. This mapping is implemented by tables in main memory, cached in a hardware map cache. Furthermore if bits 27-24 of **vma** are all 1's, **vma** contains a 24-bit physical address and the map is bypassed.

memory-data is not really a register, but represents the processor end of two pipelines, one going into memory and one coming out of memory. In the emulator task, **memory-data** remains valid once it has been read from memory; in I/O tasks **memory-data** may only be read during the **data-cycle**. Note that the read and write faces of **memory-data** do not affect each other.

In general, the timing of a memory cycle is:

1) Load the virtual address into **vma**.

2) Start the memory. If writing, simultaneously output the data to be written. This microinstruction traps if there is a map cache miss or write-protection violation.

3) One microinstruction of delay while the memory is active. It is not legal to start another memory operation in this cycle unless using block mode.

4) The memory read data are available as an operand. This microinstruction traps if there is a bad data type, an invisible pointer, a transporter trap, or a map cache miss during a block read.

For detailed information, refer to the hardware documentation.

memory-data *Atomicro*

A datum that represents data read from memory. Assigning to this stores into memory; however usually **store-contents** (see page 48) should be used instead.

memory-data-held *Atomicro*

A datum that represents data read from memory during some previous cycle. It is only legal to use this in the emulator task. It is not legitimate to use **memory-data-held** during a data cycle; use **memory-data** then. The TMC presents some additional restrictions on the use of **memory-data-held**: assigning to **vma** destroys the contents of **memory-data-held**; when doing a block read, **memory-data-held** may not be used while the memory is active. Thus **memory-data-held** becomes valid three cycles after a **start-memory** and remains valid until another **start-memory** or an assignment to the **vma**.

memory-data-advance *Atomicro*

A datum that represents the first word read from memory as part of a block read operation. The memory control is freed to advance to the next word in the block.

memory-data-held-advance *Atomicro*

A combination of **memory-data-advance** and **memory-data-held**, representing a delayed read of the first word in a block, with permission for the memory control to advance to the next word.

block-memory-data *Atomicro*

A datum that represents a word read from memory as part of a block read operation. **block-memory-data** differs from **memory-data** because on IFU machines it ensures that exceptional conditions, such as microtask interference and page-boundary crossing, are handled correctly. The memory control is not permitted to advance to the next word in the block. It is legal to use **memory-data-held** after using **block-memory-data**. It is permissible to read the *first* word in a block, when not reading it for the last time, with either **block-memory-data** or **memory-data**.

block-memory-data-advance *Atomicro*

A datum that represents a word read from memory as part of a block read operation. The memory control advances to the next word in the block, so the next time one of the **memory-data** sources is read, it will not be the same word as this time. It is *not* legal to use **memory-data-held** after using **block-memory-data-advance**. It is permissible to read the *first* word in a block with either **block-memory-data-advance** or **memory-data-advance**.

advance-md *Micro*

Permit the memory control to advance to the next word of a block read. This is not usually used by itself.

vma *Atomicro*

A datum that represents the 28-bit virtual address register. Assigning to **vma** stores a new address into the register but does not automatically start a memory cycle. It is not legal to read **memory-data-held**, **memory-data-held-advance**, **block-memory-data**, or **block-memory-data-advance** after assigning to **vma** and before a new memory read has been started. Block memory operations increment **vma**; see page 47.

start-memory *modes...* *Micro*

Start a memory cycle. The *modes* specified control the type of cycle to be started. Some *mode* symbols are followed by arguments. The following symbols may be used as modes:

read Start a read.

write Start a write. Either **read** or **write** must be specified. If both are specified, a read is started but write-access is checked in the map. If a non-DMA write is started, the data to be written must be computed and placed on the Obus in parallel with this **start-memory**.

physical *addr*

Take *addr* as the physical address, instead of mapping the virtual address in **vma**. *addr* must be an Abus source.

dma *card subdevice*

Start a DMA cycle. The data read from memory or written to memory goes to/comes from an Lbus device instead of the processor data path. *card* and *subdevice* address the Lbus device; see **select-lbus-dev** (page 56). **dma** must be used in combination with **physical**.

inhibit-page-tags

Prevent the page tags from noticing this cycle. Must be used in combination with **physical**.

address-phtc

Get the physical address by mapping the contents of **vma** through the page-hash-table-cache hash box instead of the normal map. This mode can also be turned on automatically: when a map cache miss occurs, the hardware does

```
        (start-memory read address-phtc)
```

instead of whatever **start-memory** was originally programmed, and traps to appropriate microcode.

instruction-fetch

This must be used in combination with **read** (and possibly **block**) only. The memory data being read are an instruction pair at the address in PC (or the following address when doing the second half of a 2-word TMC instruction fetch). When the instructions arrive from memory, they will be taken by the instruction fetch logic.

instruction-fetch and **block** may not be used together on IFU machines. The IFU does its own fetching of additional words following the word explicitly fetched.

hold-ifu

This must be used in combination with **instruction-fetch**. It forbids the IFU from prefetching additional words.

block Start or continue a block memory write, or continue a block memory read. (Use **first-block** to start a block memory read.)

On TMC machines **block** and **first-block** may not be used with any of the other modifiers except **read, write,** or **instruction-fetch,** may not be used with both **read** and **write** at the same time, and may only be used by the emulator task.

On IFU machines **block** may not be used in the emulator task with any of the other modifiers except **read** or **write,** and may not be used with both **read** and **write** at the same time. On IFU machines **block** may be used in an I/O task in combination with **physical** or with **physical** and **dma** together.

first-block

Start a block memory read.

A block memory operation reads or writes a sequence of memory words at ascending consecutive addresses. A word may be read or written every memory cycle; thus a block operation is considerably faster than a sequence of single operations. The **vma** is incremented as the block operation progresses. A block operation may freely cross page boundaries but reads and writes cannot be mixed in a single block operation.

The TMC and IFU handle block read rather differently, however the microcode compiler attempts to hide this. It cannot hide the fact that the **vma** increments at different times in the two machine versions. The TMC increments the **vma** when a block memory cycle is started, while the IFU increments the **vma** when a block memory cycle is completed. Specifically:

TMC Increments **vma** whenever **(start-memory block...)** or **(start-memory first-block...)** is executed.

IFU Increments **vma** when **(start-memory block write)** is executed, when **(advance-md)** is executed, or when one of the **-advance** versions of the **memory-data** register is read.

A map cache miss in a block read is handled specially. The page fault trap does not occur immediately. Instead, the memory read is suppressed and the page fault trap occurs when an attempt is made to read the result of the memory read that was never started from the **memory-data** register. This makes it possible to do a block read to the end of a page without faulting on the following page unless the block actually extends into that page. When the deferred page fault trap occurs, in the IFU **vma** points at the next word to be read, but in the TMC **vma** may have been incremented one or two extra times—the offset to the correct **vma** is encoded in the address of the trap handler and microcode recovers.

The TMC forbids the use of a non-**block** read after a **block** read, unless an assignment to **vma** intervenes. The TMC cannot do a **block** read from A-memory. The compiler cannot detect these errors.

The IFU allows **block** and **physical** to be used together in an I/O task. When writing several consecutive physical locations, or reading two consecutive physical locations, the IFU requires that **block** be used in the **start-memory** operations after the first; this tells it not to enforce memory-interleaving restrictions. The TMC, on the other hand, does not allow **block** to be used this way, and assumes that the microcode is not violating memory-interleaving restrictions when it starts memory cycles in consecutive microcode cycles.

store-contents *value options...* *Micro*

Store *value* into the currently-addressed memory location (the location **vma** points to, which in most cases will just have been read to check for invisible pointers). **store-contents** puts the word to be stored on the Obus and does a **(start-memory write)** to cause it to be stored. If other memory modes than just **write** are required, an explicit **start-memory** may be paralleled with the **store-contents**, or *options* may be used.

store-contents is different from assigning to **memory-data**, because the latter is a lower-level operation that does not enable the garbage collector tagging hardware. *value* is a Lisp datum; it is decoded by the type map and the GC map (consequently it must be an Abus source) to see whether it is a pointer and if so what it points at. This may cause GC page tags to be set and may cause a gc-write-trap if a pointer to a stack is being stored.

Valid *options* are:

block	Increment **vma** after storing in the location it currently points to.
(cdr *source***)**	Set the cdr code from *source*, which may be a number from 0 to 3, the symbolic name of a cdr-code, or a datum whose cdr-code field is to be used.
cdr-nil **cdr-next** **cdr-normal**	Set the cdr code to the specified symbolic value. If no cdr code is specified, it comes from *value*.
not-pointer	Disable garbage collector tagging; *value* is simply stored. For system storage conventions to be met, *value* must be guaranteed to have a non-pointer data type (typically fixnum). This case is identical with assigning to **memory-data**, except for the cdr-code control and the automatic **(start-memory write)**. *value* need not come from Abus if **not-pointer** is specified.
obus-as-good-as-abus	This kludge declares that *value* may come from Obus (the ALU) even

though the GC map looks at Abus; it is the microprogrammer's responsibility to ensure that the relevant bits (33-14) of the two busses are equivalent. Don't use this unless you are the microcode that it was put in to speed up.

no-amem This kludge declares that we are guaranteed not to be writing into an address that is mapped into A-memory, hence the A-memory write address logic need not be controlled. Don't use this unless you are the microcode that it was put in to speed up.

memread *address* *Micro*

Assign *address* to **vma**, call a subroutine that starts a read, and return with the data available in **memory-data**. Use this if you don't have anything useful to overlap with the wait for memory; it will conserve control memory locations.

memread-write *address* *Micro*

Assign *address* to **vma**, check for write-permission to that address, call a subroutine that starts a read, and return with the data available in **memory-data**. This is useful to guarantee that you will be able to write back into the same address without getting a page fault. Use this if you don't have anything useful to overlap with the wait for memory; it will conserve control memory locations.

transport &optional *type* *Micro*

Use this before or at the same time as picking up data read from memory. **memory-data** is read onto the Abus and decoded by the type and GC maps. A trap occurs if the word read from memory is an invisible pointer, has an invalid data type, or is a pointer to oldspace. The trap handler may restart the memory reference using a new address (e.g. if an invisible pointer is followed); in this case the new address will be stored into **vma**, **a-vma**, and **b-vma**, and the microinstruction containing the **transport** will be re-executed. (**a-vma** and **b-vma** are purely a software convention.)

type specifies the type of transport desired. It must be one of the following symbols:

data This is the default. The word read from memory is going to be used as data (i.e. as a Lisp object.) All invisible pointers are followed, oldspace pointers are detected, and an error occurs if the data type is null (unbound variable) or header (internal data structure scaffolding not valid as a Lisp object).

write The memory read was only done in preparation for a write. All invisible pointers are followed, but no oldspace checking is done and there is no error if a null pointer is detected. A header type causes an error.

cdr The **car** of a cons is being accessed by the **cdr** function, not as data but only to check its cdr code. Only header-forward, element-forward, and body-forward invisible pointers are followed and there is no oldspace check. A header type causes an error.

header The word read from memory is expected to be the header of a structure. Header-forward invisible pointers are followed, an oldspace check is done, and data types illegal as headers signal an error.

header-or-data

Same as header except that no error is signalled if a normal non-header data type is seen (no oldspace check is done either in that case). This is used by the **follow-structure-forwarding** subprimitive.

bind	The memory location is a cell being bound (e.g. a special-variable value cell). All invisible pointers except external-value-cell-pointer are followed, an oldspace check is done, and a header type causes an error (a null type does not).
bind-write	The memory location is a cell whose binding is being restored. All invisible pointers except external-value-cell-pointer are followed, no oldspace check is done, and a header type causes an error (a null type does not).
scav	The memory reference is being performed by the scavenger. An oldspace check is done, but there are no invisible pointers and no data type error checks except that an error is signalled if **dtp-gc-forward** is seen.

Additional types may be added in the future.

Data type errors are signalled via trap-0 from the type map. Invisible pointer following uses trap-2. If there is an invisible pointer to oldspace, the oldspace trap takes priority; the invisible pointer will be followed when the transport is retried after the garbage collector has had its say.

a-vma *Atomicro*
b-vma *Atomicro*

By software convention, **a-vma** and **b-vma** (respectively an A-memory and a B-memory location) sometimes contain copies of the **vma** register. The transporter does not depend on the contents of either of these registers, but if it changes **vma** it also stores the new value into **a-vma** and **b-vma**. The data type is indeterminate. **b-vma** exists to make it possible to combine (add or compare) the address in **vma** with data from the Abus. **a-vma** exists for the benefit of certain microcode that needs to remember two addresses, one of which gets transported.

write-vmas *value* *Micro*

Write *value* into **vma**, **a-vma**, and **b-vma**.

declare-memory-timing *states...* *Micro*

Declare that the current microinstruction (everything paralleled with the **declare-memory-timing**) occurs with the memory read pipeline in the specified states. More than one state can be true simultaneously when using block mode. Valid states are:

active-cycle
block-active-cycle

Memory data will become available in the following cycle. **block-active-cycle** means that the memory data are part of a block read.

data-cycle
block-data-cycle

Memory data are available in this cycle.

(next active-cycle)
(next block-active-cycle)
: The following microinstruction will be an active cycle.

(next data-cycle)
(next block-data-cycle)
: The following microinstruction will be a data cycle (the **active-cycle** state has the same effect).

emulator-after-data-cycle
: This cycle is valid for the use of **memory-data-held**.

The microcode compiler follows the timing of memory reads and gives an error message if **memory-data** is accessed at a time when it is not valid. This micro is provided to turn off such error messages when the compiler cannot follow the timing (for example, when a subroutine is called with a memory cycle already started). Be sure that you know what you are doing, and don't turn off error messages that are telling you about genuine errors. The microcode compiler does not know whether microcode is to be executed in the emulator task, an I/O task, or both; hence it is conservative and gives an error if you use the feature that the emulator task's **memory-data** are held indefinitely after a data cycle (until a new memory read is started).

See also **waiting-for-memory**, page 31.

6.10 Instruction Fetch Unit

As explained in the Flow of Control chapter, a subroutine return from the outermost microcode subroutine in the emulator task transfers control to the address supplied by the Instruction Fetch Unit (IFU). This is usually written **(next-instruction)**, although **(return)** is the same. The address is either the address of the microcode to execute the next macroinstruction, derived from its opcode, or the address of a trap routine (to handle an instruction cache miss, an instruction prefetch page boundary crossing, or a sequence break).

A *sequence break* is a signal, typically from an I/O device, that diverts the emulator task from the normal macroinstruction flow to special macrocode to handle such asynchronous signals. Sequence breaks are generated by the disk when an operation is completed, by the FEP when it needs attention, and by a periodic clock. Other sources of sequence breaks will no doubt be added in the future. Most other computers use the term *interrupt* rather than sequence break. That word is avoided in the 3600 (and the LM-2) because it could be mistaken to include task wakeups and FEP interrupts as well.

The "fake" IFU in the Temporary Memory Control contains no instruction cache and does no autonomous instruction prefetching. Instead, it contains a 4-instruction buffer. When the buffer is exhausted, the next IFU dispatch goes to microcode to access memory and refill the buffer.

In addition to the macroinstruction dispatching feature, the IFU maintains current macroinstruction and program counter (PC) registers. Bits 16-8 of the current macroinstruction are accessible on the data path as an immediate operand, may also be used in the A-memory address calculation, and may be incremented.

macro-unsigned-immediate *Atomicro*
A datum containing the macroinstruction immediate-operand field in bits 7-0, zero in bits 31-8, and fixnum data-type in bits 33-32. This is a Bbus source.

macro-signed-immediate *Atomicro*
A datum containing the macroinstruction immediate-operand field in bits 7-0, a copy of bit 7 in bits 31-8, and fixnum data-type in bits 33-32. This is a Bbus source.

increment-macro-immediate *Micro*
Add one to the immediate-operand field of the current macroinstruction. This is useful when it addresses a multi-word address-operand.

There are two macroinstructions per 36-bit word. Consequently the PC must specify a word address and a halfword-select bit. The PC is represented as a 28-bit word address with a data type tag field of 60 (**dtp-even-pc**) for the even halfword or 70 (**dtp-odd-pc**) for the odd halfword. The PC hardware is capable of incrementing in this format. This form of PC is called a *word-pc*; another useful form is simply a 29-bit halfword address, called a *halfword-pc*. The encodings of **dtp-even-pc** and **dtp-odd-pc** and the high bits supplied when the PC register is read are chosen in such a way that conversion between halfword-PC and word-PC may be done in a single microinstruction using the existing data paths (basically rotating a 32-bit word by one bit position). This facilitates arithmetic on PC values.

pc *Atomicro*
The current PC, i.e. the address of the macroinstruction currently executing. This is a word-pc.

Assigning to **pc** is usually done with the **set-pc** micro (see below), which knows how to get the IFU working on the new instruction stream. In any case assigning to **pc** also assigns to **vma**.

The hardware actually contains several PC registers at different stages of the pipeline. The EPC contains the address of the macroinstruction currently in execution and is the register that is read when **pc** is used as a source. The DPC contains the address of the macroinstruction currently being dispatched; this is the macroinstruction whose dispatch address is available for the microcode to return to when it finishes the current macroinstruction. The IPC contains the address of the macroinstruction currently being fetched from the instruction cache or instruction buffer and decoded. In the "normal" case, the IPC, DPC, and EPC address three consecutive instructions. The APC contains the physical address of the next macroinstruction pair to be prefetched from main memory into the instruction cache. The Temporary Memory Control does not contain an APC. Assigning to **pc** writes into APC, IPC, and DPC; it does not immediately change the contents of EPC. EPC only changes when control advances to the next macroinstruction, or when **accept-restart-pc** or **skip-instruction** is used.

halfword-pc *word-pc* *Micro*
Translate a word-pc into a halfword-pc. The result appears at one of the inputs to the ALU, so a number may be added to it in the same microinstruction.

word-pc *halfword-pc* *Micro*
Translate a halfword-pc into a word-pc.

even-pc *word-address* *Micro*
Translate *word-address* into a pc (word-pc) that points at the first instruction in that word.

odd-pc *word-address* *Micro*
Translate *word-address* into a pc (word-pc) that points at the second instruction in that word.

odd-pc? *pc* *Micro*
A predicate that is true if the operand *pc* points to the second instruction in a word (false if it points to the first instruction).

pc-plus-number *base-pc* *offset* &optional *delta* *Micro*
Add *offset+delta*, a positive or negative number of halfwords, to *base-pc*, a word-pc value. This micro expands into a datum that is the resulting word-pc value. *offset* is a datum. *delta* must be either missing or **1**.

pc-plus-number takes two cycles to execute, and uses **b-temp-3** to hold an intermediate result. Either argument may be **b-temp-3**.

pc-add *base-pc* *magic-offset* *Micro*
Add *magic-offset*, a halfword offset in the magic hardware-dependent format used by branch instructions, to *base-pc* and return the resulting word-pc. Unlike **pc-plus-number** this takes only one cycle to execute.

magic-offset, arithmetically shifted right by one bit, is the word offset and is added to the 28-bit pointer field of *base-pc*. The least-significant bit of *magic-offset* is the halfword select; it is added to the halfword-select bit of *base-pc*, however there is no carry from this addition into the word address. Furthermore, if *magic-offset* is negative, there is a carry into the halfword select addition that has the effect of complementing the least-significant bit of *magic-offset*.

set-pc *new-pc* &optional *other-code* *Micro*
Assign *new-pc* (a word-pc value) to the hardware PC register, synchronize with the IFU, and do a **(next-instruction)**. If *other-code* is specified, it is microcode (a single microinstruction) to be executed in parallel with the wait for the IFU. If a page fault is taken on the instruction fetch, *other-code* will not be executed and the PC will not be changed; thus the macroinstruction doing the **set-pc** will be backed up to its beginning.

set-pc-long *new-pc* &rest *code* *Micro*
First evaluate *new-pc* and check that no instruction-fetch page fault will occur when control branches to that PC. Then execute *code* (any number of microinstructions). Finally, set the PC to the saved value of *new-pc* and start executing that macroinstruction. **set-pc-long** is useful when *code* has irreversible side-effects that cannot be undone if a page fault is taken on the instruction fetch and control is backed up to the start of the macroinstruction. *code* must not use memory, must not touch **a-vma**, and must take at least three cycles; these restrictions are because *code* is actually executed in parallel with an instruction fetch, not in series as the description above would imply.

skip-instruction &optional *last in-function-entry* *Micro*

Advance the hardware PC (and the IFU) to the next macroinstruction without transferring control to the microcode dispatch address for that macroinstruction. In other words, skip the next macroinstruction, causing a subsequent **next-instruction** to take the second following macroinstruction.

Note that **skip-instruction** does not necessarily add 1 to **pc**. It actually copies DPC into EPC; DPC could differ from EPC+1 if an **(assign pc ...)** has been done, or the IFU has predicted a branch.

The optional arguments are only meaningful for IFU machines. *last*, which defaults to **nil**, if **t** means that this is the last **skip-instruction** before a **next-instruction** and therefore the IFU may proceed to prepare the next macroinstruction. *in-function-entry*, which defaults to **t**, if **nil** means to advance to the next byte of a multi-byte instruction. When *in-function-entry* is **nil**, the IFU assumes that you are reading the instruction you are skipping (for example, with **macro-unsigned-immediate** or by using it to generate an A-memory address). If the IFU is held (perhaps because of a page-boundary crossing in the instruction stream) an *advance-miss* trap may occur.

increment-pc *Micro*

This micro is obsolete; it is only supported by TMC machines, not IFU machines! On TMC machines it is identical to **skip-instruction**.

skip-instructions *offset* &optional *delta* *Micro*

Assign to the PC so that the next *offset+delta* macroinstructions will be skipped, and the macroinstruction after that will be dispatched by the next **next-instruction**. *offset* is a datum; *delta* must be either missing or **1**.

(skip-instructions (b-constant 1)) is slower than **(skip-instruction)**.

restart-pc *new-pc* *Micro*

Set the PC at which execution will restart if this macroinstruction is pclsred. No instruction fetch is done since that PC will normally not be used. The PC must be at an even halfword (usually it is an escape function). **accept-restart-pc** must be done in the next cycle, or at any rate some time before it is possible to trap out and pclsr.

accept-restart-pc *Micro*

Accept a PC, previously placed in IPC and DPC by **restart-pc**, into EPC so that it will be used as the PC from which to restart if execution is trapped and pclsred. IPC and DPC are incremented. Normally PC should be set again before doing a **next-instruction**.

push-return-pc &optional *no-top-of-stack* *Micro*

Push onto the stack a PC that points to the next macroinstruction to be executed. This takes an average of 1.5 cycles because the data path cannot increment a word-pc directly. If *no-top-of-stack* is **no-top-of-stack**, then the **top-of-stack** B-register will not be affected.

newtop-return-pc &optional *no-top-of-stack* *Micro*

newtop-return-pc is to **push-return-pc** as **newtop** is to **pushval** (see page 62).

ifu-branch *condition* *Micro*

Cooperate with the IFU to perform a conditional branch macroinstruction. The data path evaluates the predicate *condition* and sends the result to the IFU. The IFU selects the next macroinstruction based on the truth of the predicate and the **branch-if** or **branch-if-not** attribute of the current macroinstruction. **(next-instruction)** may be executed in the cycle following the **ifu-branch**. This micro is only defined for IFU machines (not for TMC machines).

6.11 Datapath Control Register

write-dp-control *datum* *Micro*

Write *datum* into the control register on the DP board. The bits in this register control various random things:

bits 1-0	The stack base. This supplies bits 11-10 of the A-memory address when the address is computed as an offset from stack-pointer, frame-pointer, or xbas.
bit 2	The sequence break flag. This is a testable condition. It is also tested by the IFU; if it is true, the IFU supplies a trap address instead of dispatching to the next macroinstruction. Typically a microtask turns on the sequence break flag in response to a device condition requiring attention from the emulator task.
bits 3,4	Trace flags 1,2. These are testable conditions with no other special hardware features.

sequence-break *Atomicro*

A predicate that is true if the sequence break flag is on.

trace-flag-1 *Atomicro*

A predicate that is true if trace flag 1 is on. This is used for function entry/exit metering.

trace-flag-2 *Atomicro*

A predicate that is true if trace flag 2 is on. This is not used yet.

allow-sequence-break *Micro*

If a sequence break is pending, take it and then retry this macroinstruction. This is a trap, so it can be done in parallel with a **jump** (but not an **if**). An **allow-sequence-break** should be put somewhere inside each microcode loop that could iterate forever.

6.12 Lbus Microdevices

These micros provide primitive microdevice operations. Usually each device will have specific micros for its operations defined in terms of these.

read-lbus-dev *card subdevice* *Micro*

A datum that is a word read from the specified microdevice. *subdevice* is a 5-bit number. *card* selects a card and is either a 5-bit backplane slot number or a symbolic card name, in which case the FEP determines the backplane slot number when the microcode is loaded.

write-lbus-dev *card subdevice datum* *Micro*

Write *datum* into the specified microdevice. *subdevice* is a 5-bit number. *card* selects a card and is either a 5-bit backplane slot number or a symbolic card name, in which case the FEP determines the backplane slot number when the microcode is loaded.

datum may be **nil**, which means that we don't care what is written; the device doesn't look at the Lbus data, and the **write-lbus-dev** is being used simply as a command to the device.

select-lbus-dev *card subdevice* *Micro*

Address a microdevice without doing anything to it. This is normally used internally by other micros. *subdevice* is a 5-bit number. *card* selects a card and is either a 5-bit backplane slot number or a symbolic card name, in which case the FEP determines the backplane slot number when the microcode is loaded.

define-lbus-card *name* &optional *slot* *Special Form*

Declares that *name* is being used as the symbolic name of an Lbus card. If *slot* is specified, it is the Lbus slot number which must contain this card. If *slot* is not specified, the FEP will find the Lbus card and will patch the microcode to contain the appropriate slot number. In this case *name* had better be a name that the FEP recognizes; if it is not, **define-lbus-card** will signal an error.

lbus-dev-cond *Atomicro*

A predicate that is true if the Lbus Dev Cond line on the bus is asserted. A **read-lbus-dev**, **write-lbus-dev**, DMA operation, or FPA operation should be done in parallel to select a device. Doing **select-lbus-dev** by itself is not sufficient.

6.13 Special Sequencer Controls

halt *reason* *Micro*

Stop the machine after executing this microinstruction. *reason* is a string placed in the error table for the use of the Lisp function **dbg:decode-micro-pc**. (In the future *reason* might also be placed in a file to be read by the FEP.)

popj-into-npc *Micro*

Pop the top word off the control stack and put it into the NPC register.

<LMSQ>SQBLK.PLT

pop-control-stack *Micro*

Pop the top word off the control stack and return it as a datum; also put it into the NPC register. Only the low 14 bits of the datum are valid. The remaining bits contain other things; however a **ldb** operation to mask it off cannot be paralled with **pop-control-stack** due to hardware limitations. It is possible to mask it off by **loganding** it with a b-constant.

read-csp &optional *temp* *Micro*

The control-stack pointer. This is a 4-bit read-only register. It takes two cycles to read it, and *temp* is clobbered. *temp* must be a B register; it defaults to **b-temp**.

See also **read-cur-task-and-csp**, page 61.

call-ctos *Micro*

Call a subroutine whose address comes from CTOS (the top of the microcode subroutine stack). Note that you can't pop the control stack and store a return address at the same time, so the address of the subroutine will remain on the stack underneath the return address. If not for that, this might be useful for coroutines.

The **long-dispatch** micro, which writes into the NPC register from the data path, also comes under this category. See page 26.

6.14 Tasking

There are 16 microcode tasks. The hardware automatically schedules execution of these tasks and switches between them when a request arrives for a task of higher priority than the current one, or when the current task is *dismissed*. Larger task numbers denote higher priority. Each microcode task has its own sequencer state, consisting of CPC and NPC microinstruction address registers and a 16-entry subroutine stack. Each time a task is awakened it continues execution from wherever it was the last time it ran. There is no overhead for switching tasks; the machine executes microinstructions continuously during the switch.

The lowest-priority task is called the *emulator task*; it executes Lisp macroinstructions. All tasks other than the emulator task are collectively referred to as *I/O tasks*; they generally deal with peripheral I/O devices.

The emulator task has its own memory data registers, thus its use of the memory pipeline needs no interlocking with other tasks. Only the emulator task is allowed to use virtual memory and the **vma** register. There are no per-task A-memory nor B-memory registers, except as conventionally defined by the microprogrammer.

The tasks are:

Tasks 8-15	High-speed devices. These tasks are used to operate I/O devices and to supervise DMA. The wakeup request for each task comes through the Lbus from the device assigned to it. Each task-using Lbus device can be dynamically assigned to any one of these eight tasks, or to no task.
Task 7	Not used except by diagnostics.
Tasks 1,2,5,6	Software tasks. Wakeup requests are in a register whose bits can be set or cleared by microcode (see **wakeup-task** and **dismiss**). One of these tasks (6 currently)

provides "background" service to the DMA tasks.

Task 4	Low-speed devices. This task can be used by more than one device at the same time; when awakened it must poll the devices. The wakeup request comes from the Lbus. Currently the only device that uses this task is the TV vertical-sync wakeup, used as a periodic clock.
Task 3	FEP service. The wakeup request is settable by the FEP and clearable by microcode. Currently this is used for metering.
Task 0	Emulator task. The wakeup request is always true.

dismiss *Micro*

Dismiss the current task. It will execute *n* additional microinstructions and then stop executing until it is awakened again. Normally *n* is 1 in tasks 8-15, 4 in tasks 3 and 4, and 2 in the remaining tasks. If a **dismiss** occurs simultaneously with a switch to a higher-priority task, *n* will be zero; the additional microinstruction is not executed until the next time the task is awakened. This special case cannot happen if the **dismiss** is in the first microinstruction executed after the task is awakened, or if the **dismiss** is executed in parallel with a **start-memory**. Similarly, if a switch to a higher-priority task occurs during the 2 or 4 extra microinstructions after a **dismiss** in tasks 1-7, when the higher-priority task dismisses itself the wakeup request for the lower-priority task will be gone and the remaining extra microinstructions will not be executed until the task is awakened again.

When dismissing a task whose wakeup request comes from external hardware (any task other than 1, 2, 5, or 6), the **dismiss** should be executed in parallel with the appropriate microdevice operation to clear the task's wakeup request.

dismiss is a no-op in the emulator task (task 0).

wakeup-task *n* *Micro*

Wake the specified task. *n* must be the number of a software-awakened task (1, 2, 5, or 6) or a symbolic task name defined as one of those numbers with **defsysconstant** in the Sysdef file.

disable-tasking *Micro*

Prevent switching to a higher-priority task in the cycle *after next*. The task scheduler may have already committed to switching to a higher-priority task in the next cycle, and **disable-tasking** cannot affect this. To enter an uninterruptible sequence of microinstructions, **disable-tasking** should be used in every microinstruction of the sequence except for the last two, and should be used in one microinstruction before the beginning of the sequence. If either of the last two microinstructions uses the Lbus, and therefore can be forced to wait when the bus is busy, **disable-tasking** must be used in that microinstruction as well as all its predecessors.

There are some interactions between tasking and the memory pipeline. The hardware does an automatic **disable-tasking** in any microinstruction that starts a non-DMA memory read and is not in the emulator task. This ensures that when the memory data arrive in the cycle after next the machine is still in the correct task. This **disable-tasking** is not done in the emulator task since it has a private memory data register. Any attempt to start a memory read or write when the machine is committed to switching to a different task in the next cycle is suppressed; when control returns to this task it will try the memory-starting microinstruction again. This ensures that memory is available in the first cycle of the awakening of any task, allows a task to control what happens in the *active* cycle of its memory operations, and guarantees that tasking cannot cause violations of memory interleaving by ensuring that two different tasks cannot start memory operations in successive cycles.

defer-dismiss-one-cycle *Micro*

Put this in parallel with a **(dismiss)** in a hardware task that dismisses the device with a microdevice operation (**dismiss-disk-task** for example) to ensure that the following microinstruction will be executed before the task is finally dismissed. If the microdevice operation is held up waiting for the bus, and then interrupted by a higher-priority task, so that the dismiss happens in parallel with a task switch, the wrong number of microinstructions will be executed after the dismiss. If the higher-priority task runs for more than two cycles, no microinstructions will be executed after the dismiss (until the next wakeup). But if the higher-priority task runs for only two cycles, two additional microinstructions will be executed after the dismiss, putting the microtask and the device out of synchronization. (This is caused by pipelining in the sequencer's task scheduler).

defer-dismiss-one-cycle, when done in parallel with a microdevice operation, causes the microinstruction to be delayed if a task switch is about to occur. Thus it guarantees that this microinstruction and its successor will be executed consecutively, and then the task will be dismissed.

In normal DMA operation **(defer-dismiss-one-cycle)** is not required, because starting a memory cycle in parallel with a dismiss does the same thing.

write-task-state *n value* *Micro*

Write the saved state of task *n* with the 32-bit datum *value*. *n* is either a number between 0 and 15. or a symbolic task name defined with **defsysconstant** in the Sysdef file. There is currently no provision for *n* to be variable, although the hardware would allow it. *value* is a datum, usually a constant constructed with **build-task-state**.

disable-tasking must be used in the cycle before a **write-task-state**. If this is not done, the state of the current task may be clobbered because **write-task-state** precludes saving the state of the current task if a task switch occurs simultaneously with it.

build-task-state is a special marker that may be used in a constant. It constructs a number that is appropriate as an operand for **write-task-state**. The form is

```
(a-constant '(build-task-state field value field value...))
```

b-constant may be used instead of **a-constant**, of course. The possible *fields* are:

cpc	Address of the first microinstruction to be executed by the task. The *value* may be a number or the name of a microcode routine defined with **defucode**. This field is mandatory.
npc	Value to go in the NPC register. The *value* may be a number, the name of a microcode routine, or a list of the symbol **npc-successor** and either of those. This field is optional and the default is **npc-successor** of whatever is in the **cpc** field.
csp	Value to go in the CSP register (the task's stack pointer). This field is optional and the default is 17, i.e. an empty stack.

read-cur-task &optional *temp* *Micro*

The current task number. This is a 4-bit read-only register. It takes two cycles to read it, and *temp* is clobbered. *temp* must be a B register; it defaults to **b-temp**. It is illegal to use this micro in microcode intended to be executed in any task other than the emulator if *temp* is **b-temp**. In general one must be careful not to use the same B register simultaneously in two different tasks. Consequently this micro is only useful for error checking (to halt if a microcode routine is executed in the wrong task).

read-cur-task-and-csp &optional *temp* *Micro*

An 8-bit read-only register, containing the control-stack pointer in the low 4 bits and the current task number in the high 4 bits. It takes two cycles to read it, and *temp* is clobbered. *temp* must be a B register; it defaults to **b-temp**. It is illegal to use this micro in microcode intended to be executed in any task other than the emulator if *temp* is **b-temp**. In general one must be careful not to use the same B register simultaneously in two different tasks.

7. Architectural Stuff

This chapter describes micros that implement the Lisp architecture. They are less intimately associated with the hardware than those described previously.

7.1 The Stack

The top few pages of the Lisp stack, including the entirety of the current frame, are stored in a part of A memory known as the stack buffer. The stack-pointer and frame-pointer registers contain the virtual addresses of the top of the stack and the current frame, respectively; these same registers, used as A-memory base registers, address the A-memory locations containing those virtual addresses.

The stack may also be addressed as normal virtual memory; references to those pages currently residing in the stack buffer are automatically redirected to A memory.

The top word of the stack is duplicated in a B-memory location. This makes it possible to feed the top two words on the stack, or the top word on the stack and some location in the current frame, into the ALU as a pair of operands.

top-of-stack *Atomicro*
The B-memory location containing the top word on the stack.

p-of-stack-a *Atomicro*
The A-memory location containing the top word on the stack. This is a more concise way of saying **(amem (stack-pointer 0))**.

next-on-stack *Atomicro*
The A-memory location containing the next-to-top word on the stack. This is a more concise way of saying **(amem (stack-pointer -1))**.

address-operand *Atomicro*
The A-memory location addressed by the current macroinstruction. This is a more concise way of saying **(amem (macrocode))**.

The following micros are used to maintain the stack, taking care of the convention that the top word is stored in both A and B memories.

pushval *value* *Micro*
Push *value* onto the stack, with a cdr code of cdr-next. The stack-pointer is incremented. This is the standard way to store the result of an instruction (when there are no arguments to be popped off).

newtop *value* *Micro*
Put *value* into the top of the stack, with a cdr code of cdr-next. The previous top of the stack is replaced, and the stack-pointer does not change. This is the standard way to store the result of an instruction that pops one argument and pushes one result.

pop2push *value* *Micro*

Effectively pop the stack twice and then does a **pushval,** but do it all in a single microinstruction. This is the standard way to store the result of a microinstruction that pops two arguments and pushes one result.

popval *Micro*

A datum that is the word on the top of the stack (as a Bbus source). As a side-effect, the stack is popped; i.e. the stack-pointer is decremented and the B-memory top-of-stack register is updated.

pushval-with-cdr *value* *Micro*

Identical to **pushval** except that *value*'s cdr code is preserved (**pushval** always sets the cdr code of the stack location to cdr-next). *value* would normally be a **set-cdr** expression.

newtop-with-cdr *value* *Micro*

Identical to **newtop** except that *value*'s cdr code is preserved (**newtop** always sets the cdr code of the stack location to cdr-next). *value* would normally be a **set-cdr** expression.

7.2 Standard A and B Registers

a-temp *Atomicro*
a-temp-2 *Atomicro*
b-temp *Atomicro*
b-temp-2 *Atomicro*
b-temp-3 *Atomicro*

General-purpose temporary registers for use in the emulator task. These are generally not assumed to be preserved by subroutines.

define-b-temps may be used to assign names to additional temporary B registers used within a single routine. These are for use in the emulator task only. See page 20.

quote-nil *Atomicro*
b-quote-nil *Atomicro*

These registers contain the symbol **nil** in the standard microcode. **quote-nil** is an A-memory register. **b-quote-nil** is a B-memory register.

quote-t *Atomicro*
b-quote-t *Atomicro*

These registers contain the symbol **t** in the standard microcode. **quote-t** is an A-memory register. **b-quote-t** is a B-memory register.

A large number of registers are set up by the Sysdfl file and will not be discussed here. Most of these registers are the value cells of Lisp variables and are used for communication between Lisp and the microcode.

7.3 The Current Stack Frame

The atomicros in this section define various fields in the header of the current stack frame (pointed to by **frame-pointer**).

frame-function *Atomicro*
The currently-executing function.

frame-misc-data *Atomicro*
A fixnum full of various fields. Accessors for these fields are defined below.

frame-return-pc *Atomicro*
The return PC of this frame's caller.

frame-previous-top *Atomicro*
The address of the top of the previous frame; this is put into stack-pointer when the current frame returns. The cdr code of this word is the value disposition code.

frame-previous-frame *Atomicro*
The address of the previous frame; this is put into frame-pointer when the current frame returns.

Fields in **frame-misc-data**:

frame-number-of-args *Atomicro*
The number of arguments supplied when this frame was called.

frame-cleanup-bits *Atomicro*
If this field is not zero, extra work needs to be done when this frame returns or is thrown through.

frame-buffer-underflow-bit *Atomicro*
1 if the previous frame is not entirely in the stack buffer.

frame-unsafe-reference-bit *Atomicro*
1 if there are pointers to this frame.

frame-catch-bit *Atomicro*
1 if there are catches or unwind-protects in this frame.

frame-bindings-bit *Atomicro*
1 if there is a frame on the binding stack associated with this frame.

frame-trace-bit *Atomicro*
1 if a trap to the debugger is requested when this frame is unwound (either by return or by throw).

frame-meter-bit *Atomicro*
1 if a trap to the metering system is requested when this frame is unwound (either by return or by throw).

frame-bottom-bit *Atomicro*
1 if this is the bottom frame in its stack; trap and do a **stack-group-return** if this frame tries to return.

first-part-done *Atomicro*
1 if an instruction running in this frame was trapped out of and is in an intermediate state (a few instructions look at and set this flag).

frame-lexpr-called *Atomicro*
1 if this frame was called via **apply** or **lexpr-funcall**. The last entry in the caller's copy of the arguments is a list of arguments.

frame-funcalled *Atomicro*
1 if this frame was called via **funcall** or a similar operation; the caller's copy of the arguments is in a slightly different place.

frame-instance-called *Atomicro*
1 if this frame contains a method called by sending a message to an instance. The first two local slots in the frame contain **self** and **self-mapping-table**.

frame-argument-format *Atomicro*
A 2-bit field consisting of **frame-lexpr-called** and **frame-instance-called**.

7.4 Tag Manipulation

The micros described in this section are used to implement the subprimitive instructions that manipulate the tag field in a Lisp "pointer." They are complicated by the fact that the tag field is of variable width: 8 bits normally, but 4 bits in fixnums and flonums.

cdr-field *operand* &optional *background* *Micro*
A 2-bit datum that is the cdr code of *operand*, an Abus source. If *background* is specified, it supplies the rest of the bits, as in **ldb**.

high-type-field *operand* &optional *background* *Micro*
A 2-bit datum that is the high 2 bits of the type field of *operand*, an Abus source. To extract all 6 type bits, you must use **low-tag-field** separately and then combine the results. If *background* is specified, it supplies the rest of the bits, as in **ldb**.

high-tag-field *operand* &optional *background* *Micro*
A 4-bit datum that is the cdr code and high type bits of *operand*, an Abus source. If *background* is specified, it supplies the rest of the bits, as in **ldb**.

low-tag-field *operand* &optional *background* *Micro*
A 4-bit datum that is the low 4 type bits of *operand*, an Abus source. To extract all 6 type bits, you must use **high-type-field** separately and then combine the results. If *background* is specified, it supplies the rest of the bits, as in **ldb**.

pointer-field *operand* &optional *background* *Micro*
A 28-bit datum that is the pointer field of *operand*, an Abus source. If *background* is specified, it supplies the rest of the bits, as in **ldb**.

dpb-tag-field *tag pointer* *Micro*
A 36-bit datum containing *tag* in its tag field and *pointer* in its pointer field. *tag* is an 8-bit Bbus source and *pointer* is a 28-bit Abus source.

dpb-tag-field-high-only *tag fixnum* *Micro*
Like **dpb-tag-field** but only the high 4 bits of the tag come from *tag*; *fixnum* supplies the low 32 bits of the result. Note that the low 4 bits of *tag* are ignored and bits 7-4 are used.

set-low-tag-field *operand tag* *Micro*
A 32-bit datum containing *operand* in its low 28 bits and the constant number *tag* in its high 4 bits (the low 4 bits of the tag field).

dpb-cdr-field *tag operand* *Micro*
A 36-bit datum consisting of *operand* (an Abus source) with its cdr-code field replaced by *tag*. The hardware takes the cdr-code from bits 7-6 of Bbus, so *tag* is required to be a datum that extracts those bits from a Bbus source or the micro will signal an error.

dpb-type-field *tag pointer* *Micro*
A 34-bit datum consisting of *tag* in the data type field and *pointer* in the pointer field. *tag* is a 6-bit Bbus source. *pointer* is a 28-bit Abus source.

See also **merge-cdr** and **merge-high-tag** (page 44).

7.5 Traps

These micros implement trapping out from microcode to macrocode. This includes possibly saving the current PC, possibly resetting stack-pointer to its value at the beginning of the macroinstruction, emptying the microcode subroutine return stack, setting PC to point to the first macroinstruction of the trap handler, and re-entering macroinstruction processing. The trap handler is always an escape function, defined in the Sysdfl file. Escape functions are normally written in "assembly language", because of the low-level things that they must do. **take-function-trap** may be used to invoke an escape function written in Lisp.

Aborting a macroinstruction is called *pclsring* in the microcode, by analogy with the corresponding issue in the ITS operating system. Think of it as a neologism with the same historical status as *cdr*.

take-pre-trap *escape-function-name stack-control* *Micro*

Back out of the current instruction and trap to an escape function. *stack-control* is **preserve-stack** to leave stack-pointer alone or **restore-stack** to undo any pushes or pops that may have been done by this macroinstruction. The PC saved on the stack points to the current macroinstruction, and has a cdr code of cdr-normal as a clue to the debugger. The escape function may exit and retry the instruction by popj'ing to that PC.

take-post-trap *escape-function-name stack-control* *Micro*

Trap to an escape function, logically after the current macroinstruction. *stack-control* is **preserve-stack** to leave stack-pointer alone or **restore-stack** to undo any pushes or pops that may have been done by this macroinstruction. The PC saved on the stack points to the macroinstruction after the current one, and has a cdr code of cdr-normal as a clue to the debugger. The escape function may exit by popj'ing to that PC.

take-jump-trap *escape-function-name stack-control* *Micro*

Trap to an escape function without saving the current PC. *stack-control* is **preserve-stack** to leave stack-pointer alone or **restore-stack** to undo any pushes or pops that may have been done by this macroinstruction.

take-jump-trap-with-continuation *escape-function-name continuation stack-control* *Micro*

Trap to an escape function, pushing the datum *continuation* on the stack as its return PC, with a cdr code of cdr-next. *stack-control* is **preserve-stack** to leave stack-pointer alone or **restore-stack** to undo any pushes or pops that may have been done by this macroinstruction.

take-function-trap *function-name destination n-args* &optional *retry-instruction* *Micro*

Trap out to a macrocode function. The stack-pointer is always left alone by this micro. Before executing a **take-function** trap you must restore the stack (if necessary) and then push the arguments to be passed to the trap function. In the simplest case the arguments may already be in the stack as the arguments to the current macroinstruction. **take-function-trap** is guaranteed not to pclsr before execution enters the macrocode function.

function-name is the name of the function to be called. It must have been declared in the **microcode-constants** area in the SYSDF1 file; this causes the cold-load generator to make the function cell available to the microcode by putting it in a known place in A memory.

destination controls what is done with the result returned by the function. It must be either **effect**, meaning that the result is to be thrown away, or **value**, meaning that the result (one value) is to be pushed on the stack (after popping the arguments).

n-args is the number of arguments.

retry-instruction controls where the function returns to. Specifying **nil** or leaving this argument unsupplied causes the function to return to the next macroinstruction after the current one. Specifying the symbol **retry-instruction** causes the function to return to the current macroinstruction, executing it again.

a-pclsr-top-of-stack *Atomicro*

This A-memory location is used (by software convention) to assist in the restoration of the stack when pclsring (aborting a macroinstruction). If the contents of this register has type tag **dtp-null**, it is empty and has no effect. Otherwise it contains the value that should be restored on the top of the stack if we pclsr. This is used by macroinstructions that pop an argument off the stack and push something else on (smashing the argument) before they are sure that their execution will complete successfully.

check-frame-size *Micro*

Check that the difference between **stack-pointer** and **frame-pointer** is reasonable, and halt the machine if it is not. This takes two cycles. This is really a debugging measure to detect microcode or low-level macrocode bugs that clobber frame pointers, push gobs of stuff into the stack (making a frame that wraps around and clobbers earlier parts of the stack buffer), etc. Clobbers **b-temp**.

check-binary-arithmetic-operands-fast *format index no-operand-version* *Micro*
&optional *float-version fixnum-overflow flonum-fixnum-version*

Do the dispatching required for a two-operand generic arithmetic instruction, optimizing speed rather than control-memory space.

format is the macroinstruction format (e.g. **no-operand** or **address-operand**); see page 21.

index is the arithmetic dispatch table index for this operation, used if it is necessary to call out to macrocode. This is a symbol such as **%arith-op-add**, defined in the SYSDEF file.

no-operand-version is the name of the **no-operand** version of this instruction, which is sometimes used as a subroutine by the other versions.

float-version is the name of the microcode to handle the case of this instruction where both operands are flonums. If unsupplied or **nil**, a trap to macrocode will occur. If one operand is a flonum and the other is a fixnum, the fixnum will be converted to a flonum and then *float-version* will take control.

fixnum-overflow is the name of the microcode to handle a fixnum overflow. If unsupplied or **nil**, an error will be signalled if an overflow occurs. Fixnum overflow is only detected if a suitable micro, such as **add-checking-overflow**, is executed in parallel with the **check-binary-arithmetic-operands-fast**.

flonum-fixnum-version is the name of the microcode to handle the case where the first operand is a flonum and the second is a fixnum. If supplied and non-**nil**, this overrides the conversion of the second operand to a flonum and invocation of *float-version*. This is particularly useful in **signed-immediate-operand** format.

check-binary-arithmetic-operands-slow *format index no-operand-version* *Micro*
&optional *float-version*

Do the dispatching required for a two-operand generic arithmetic instruction, optimizing control-memory space rather than speed. This is also useful when the use of arithmetic-trap-enable by **check-binary-arithmetic-operands-fast** causes a microinstruction field conflict. Note that not all of the optional features of the fast version are provided.

check-unary-arithmetic-operation-fast *format index no-operand-version* &optional *float-version fixnum-overflow* *Micro*

Do the dispatching required for a one-operand generic arithmetic instruction, optimizing speed rather than control-memory space.

format is the macroinstruction format (e.g. **no-operand** or **address-operand**); see page 21.

index is the arithmetic dispatch table index for this operation, used if it is necessary to call out to macrocode. This is a symbol such as **%arith-op-zerop**, defined in the SYSDEF file.

no-operand-version is the name of the **no-operand** version of this instruction, which is sometimes used as a subroutine by the other versions.

float-version is the name of the microcode to handle the case of this instruction where both operands are flonums. If unsupplied or **nil**, a trap to macrocode will occur. If one operand is a flonum and the other is a fixnum, it will be converted to a flonum and then *float-version* will take control.

fixnum-overflow is the name of the microcode to handle a fixnum overflow. If unsupplied or **nil**, an error will be signalled if an overflow occurs. Fixnum overflow is only detected if a suitable micro, such as **add-checking-overflow**, is executed in parallel with the **check-unary-arithmetic-operands-fast**.

8. Esoterica

This chapter is a grab-bag of unorganized documentation of the more obscure parts of the machine. It may not belong in this document at all, but it is here since it may benefit some readers.

8.1 Virtual Address Map

write-both-maps *data* *Micro*
write-lru-map *data* *Micro*
write-map-a *data* *Micro*
write-map-b *data* *Micro*

Write *data* into one or both sectors of the map cache; the map cache location(s) to be written are addressed by **vma**. *data* must be an Abus source. The bits in *data* are the same as the bits in a PHTC entry:

0	Write protect
7-1	Junk
23-8	Physical page number
31-24	VMA tag. Must match VMA<27-20> to be a valid map entry. Must be all 1's to invalidate a map cache entry.
35-32	Ignored (normally fixnum data type).

map-load-successful *Atomicro*

A microcondition that is true if the VMA tag field of the data being written into the map matches the VMA. This is only meaningful if executed in parallel with a write into the map. **map-load-successful** is used when refilling the map cache from the PHTC, to check the validity of the PHTC entry probed. Note that if **map-load-successful** is false, the map cache has been loaded with garbage and should be rewritten with -1 to invalidate the garbage.

map-select-code *Atomicro*

A two bit field describing the result of looking up **vma** in the map cache. Values are:

0	Cache miss
1	Found in map A
2	Found in map B
3	**vma** contains a physical address

map-data *Atomicro*

A 36-bit datum containing the entire contents of the map cache location addressed by **vma**. **map-select-code** is a field in this datum. The bits in this datum are:

23-0	The physical address output from the map. If there is a map cache hit, bits 23-8 come from the selected map cache word and bits 7-0 come from the VMA.
31-24	VMA tag from the selected map cache word.
33-32	Code describing the output from the map (see **map-select-code** above).
34	Write protect
35	Parity error

Note that the bits *read* from the map are not quite the same as the bits that are *written*

<LMTMC>MCABLK.PLT

<LMTMC>MCDBLK.PLT

into the map.

Currently no micro is provided for writing the PHTA, ASN, and IFU-CODE registers. Use **(write-lbus-dev 37 1** *data***)**.

When starting a memory read or write cycle, **vma** is looked up in the map cache. If **vma<27:24>** is all 1's, the map cache is bypassed and the physical address in **vma<23:0>** is used. If a hit occurs in the map cache, the resulting physical page number is combined with **vma<7:0>** to produce the physical address to be referenced. If a map cache miss occurs, the physical address is the base address of the PHTC plus a hash function of the virtual page number in **vma**, a memory read is started (even if the original request was for a write), and the microcode traps to the map-miss handler at location 10001. This microcode normally attempts to reload the map from the PHTC entry, and if that fails takes other action to translate the virtual address.

If a map cache hit occurs, but a write is being requested and the write protect bit in the map cache entry is set, the write cycle is changed to a read and the microcode traps to handle the protection violation. The trap address is 10031 on TMC machines, 10011 on IFU machines.

The IFU Memory Control hides the workings of the memory pipeline from the microcode by inserting delays where necessary. In the Temporary Memory Control there is some unusual interaction between the map cache and block-mode memory cycles, because of the way the memory pipeline works. A map cache miss during a block write traps in the same way as a write-protection violation. A map cache miss during a block read (for locations after the first one in the block) does not trap until the microcode attempts to read the missing data, and does not do a PHTC probe. The microcode trap address is incremented by 8 times the difference (1 or 2) between the contents of **vma** and the address of the missing data; microcode must recover and restart the memory pipeline. Note that if a block read is continued past the desired number of data words, and "accidentally" crosses a page boundary, no page fault will occur since the extra words will never be read.

8.2 Garbage Collector Map

The GC Map contains one 3-bits+parity entry for each quantum (16384 words) of address space. It describes those attributes of a region that need to be handled by hardware. The GC map causes a transporter trap when a pointer to oldspace is read from memory, identifies pointers to temporary space for the benefit of the page tag hardware (see below), can be used to identify pointers to a particular region or set of regions when scanning through memory, and can cause a trap when a pointer to a stack is written into memory.

write-gc-map *address data* *Micro*

Write *data* into the location of the GC map that corresponds to *address*. Bits 27-14 of *address* are relevant. *data* may take on the following meaningful values:

10 Normal

1 Temporary space (causes gc page tag to be set in memory write)

2 Condemned Temporary Space (like Normal, but can be checked for as a microcondition). Note that this does not cause a transport trap and hence should only be used transiently.

13 Oldspace (causes a transport trap in memory read)

4 This stack (can be enabled to trap)

15 Other stack (can be enabled to trap)

[I still need to document slow jumps. There is no micro provided yet for reading the GC map.]

8.3 Page Tags

The 3600 has two *page tag* bits for each physical page of main memory. One bit, the *reference tag*, is set whenever that physical page is referenced. The page replacement algorithm uses the reference tags to determine which pages have not been used recently and are good candidates for removal from main memory. The other bit, the *GC tag*, is set whenever a pointer to a temporary space is written into that physical page. The garbage collector uses the GC tags to locate quickly all the pointers to a region that has been condemned, so that the objects in that region that are still in use may be evacuated.

The page tags see only "normal" memory references. They do not see references by the FEP, non-emulator-task references (including DMA references), nor references with SPEC INHIBIT PAGE TAGS asserted.

[The micros for the page tags may be found in SYS: L-UCODE; MAP LISP, but they are not y general and I think I don't want to document them. Also it uses write-lbus-dev of magic ibers.]

page-referenced Atmicro

(guu p12)

Add sect on 32-36-bit conversions on S-mach See guu p73

Concept Index

Variable Index

Index of Lisp Functions

Index of Atomicros

Index of Micros