**1. Introduction.** This program simulates a simplified version of the `MMIX` computer. Its main goal is to help people create and test `MMIX` programs for *The Art of Computer Programming* and related publications. It provides only a rudimentary terminal-oriented interface, but it has enough infrastructure to support a cool graphical user interface — which could be added by a motivated reader. (Hint, hint.)

`MMIX` is simplified in the following ways:

- There is no pipeline, and there are no caches. Thus, commands like `SYNC` and `SYNCD` and `PREGO` do nothing.
- Simulation applies only to user programs, not to an operating system kernel. Thus, all addresses must be nonnegative; "privileged" commands such as `PUT rK,z` or `RESUME 1` or `LDVTS x,y,z` are not allowed; instructions should be executed only from addresses in segment 0 (addresses less than `#2000000000000000`). Certain special registers remain constant: rF = 0, rK = `#ffffffffffffffff`, rQ = 0; rT = `#8000000500000000`, rTT = `#8000000600000000`, rV = `#369c200400000000`.
- No trap interrupts are implemented, except for a few special cases of `TRAP` that provide rudimentary input-output.
- All instructions take a fixed amount of time, given by the rough estimates stated in the `MMIX` documentation. For example, `MUL` takes $10v$, `LDB` takes $\mu + v$; all times are expressed in terms of $\mu$ and $v$, "mems" and "oops." The simulated clock increases by $2^{32}$ for each $\mu$ and 1 for each $v$. But the interval counter rI decreases by 1 for each $v$; and the usage count field of rU may increase by 1 (modulo $2^{47}$) for each instruction.

**2.** To run this simulator, assuming UNIX conventions, you say '`mmix` ⟨ options ⟩ `progfile args...`', where `progfile` is an output of the `MMIXAL` assembler, `args...` is a sequence of optional command line arguments passed to the simulated program, and ⟨ options ⟩ is any subset of the following:

• `-t<n>` Trace each instruction the first $n$ times it is executed. (The notation `<n>` in this option, and in several other options and interactive commands below, stands for a decimal integer.)

• `-e<x>` Trace each instruction that raises an arithmetic exception belonging to the given bit pattern. (The notation `<x>` in this option, and in several other commands below, stands for a hexadecimal integer.) The exception bits are DVWIOUZX as they appear in rA, namely `#80` for D (integer divide check), `#40` for V (integer overflow), ..., `#01` for X (floating inexact). The option `-e` by itself is equivalent to `-eff`, tracing all eight exceptions.

• `-r` Trace details of the register stack. This option shows all the "hidden" loads and stores that occur when octabytes are written from the ring of local registers into memory, or read from memory into that ring. It also shows the full details of `SAVE` and `UNSAVE` operations.

• `-l<n>` List the source line corresponding to each traced instruction, filling gaps of length $n$ or less. For example, if one instruction came from line 10 of the source file and the next instruction to be traced came from line 12, line 11 would be shown also, provided that $n \geq 1$. If `<n>` is omitted it is assumed to be 3.

• `-s` Show statistics of running time with each traced instruction.

• `-P` Show the program profile (that is, the frequency counts of each instruction that was executed) when the simulation ends.

• `-L<n>` List the source lines corresponding to each instruction that appears in the program profile, filling gaps of length $n$ or less. This option implies `-P`. If `<n>` is omitted it is assumed to be 3.

• `-v` Be verbose: Turn on all options. (More precisely, the `-v` option is shorthand for `-t9999999999 -e -r -s -l10 -L10`.)

• `-q` Be quiet: Cancel all previously specified options.

• `-i` Go into interactive mode before starting the simulation.

• `-I` Go into interactive mode when the simulated program halts or pauses for a breakpoint.

• `-b<n>` Set the buffer size of source lines to $\max(72, n)$.

• `-c<n>` Set the capacity of the local register ring to $\max(256, n)$; this number must be a power of 2.

• `-f<filename>` Use the named file for standard input to the simulated program. This option should be used whenever the simulator is not being used interactively, because the simulator will not recognize end of file when standard input has been defined in any other way.

• `-D<filename>` Prepare the named file for use by other simulators, instead of actually doing a simulation.

• `-?` Print the "`Usage`" message, which summarizes the command line options.

The author recommends `-t2 -l -L` for initial offline debugging.

While the program is being simulated, an *interrupt* signal (usually control-C) will cause the simulator to break and go into interactive mode after tracing the current instruction, even if `-i` and `-I` were not specified on the command line.

**3.** In interactive mode, the user is prompted '`mmix>`' and a variety of commands can be typed online. Any command line option can be given in response to such a prompt (including the '`-`' that begins the option), and the following operations are also available:

• Simply typing ⟨return⟩ or `n`⟨return⟩ to the `mmix>` prompt causes one `MMIX` instruction to be executed and traced; then the user is prompted again.

• `c` continues simulation until the program halts or reaches a breakpoint. (Actually the command is '`c`⟨return⟩', but we won't bother to mention the ⟨return⟩ in the following description.)

• `q` quits (terminates the simulation), after printing the profile (if it was requested) and the final statistics.

• `s` prints out the current statistics (the clock times and the current instruction location). We have already discussed the `-s` option on the command line, which causes these statistics to be printed automatically; but a lot of statistics can fill up a lot of file space, so users may prefer to see the statistics only on demand.

• `l<n><t>`, `g<n><t>`, `$<n><t>`, `rA<t>`, `rB<t>`, ..., `rZZ<t>`, and `M<x><t>` will show the current value of a local register, global register, dynamically numbered register, special register, or memory location. Here `<t>` specifies the type of value to be displayed; if `<t>` is '`!`', the value will be given in decimal notation; if `<t>` is '`.`' it will be given in floating point notation; if `<t>` is '`#`' it will be given in hexadecimal, and if `<t>` is '`"`' it will be given as a string of eight one-byte characters. Just typing `<t>` by itself will repeat the most recently shown value, perhaps in another format; for example, the command '`l10#`' will show local register 10 in hexadecimal notation, then the command '`!`' will show it in decimal and '`.`' will show it as a floating point number. If `<t>` is empty, the previous type will be repeated; the default type is decimal. Register `rA` is equivalent to `g21`, according to the numbering used in `GET` and `PUT` commands.

The '`<t>`' in any of these commands can also have the form '`=<value>`', where the value is a decimal or floating point or hexadecimal or string constant. (The syntax rules for floating point constants appear in MMIX-ARITH. A string constant is treated as in the `BYTE` command of `MMIXAL`, but padded at the left with zeros if fewer than eight characters are specified.) This assigns a new value before displaying it. For example, '`l10=.1e3`' sets local register 10 equal to 100; '`g250="ABCD",#a`' sets global register 250 equal to `#`000000414243440a; '`M1000=-Inf`' sets $M_8[\#1000] = \#$`fff0000000000000`, the representation of $-\infty$. Special registers other than rI cannot be set to values disallowed by `PUT`. Marginal registers cannot be set to nonzero values.

The command '`rI=250`' sets the interval counter to 250; this will cause a break in simulation after $250v$ have elapsed.

• `+<n><t>` shows the next $n$ octabytes following the one most recently shown, in format `<t>`. For example, after '`l10#`' a subsequent '`+30`' will show `l11`, `l12`, ..., `l40` in hexadecimal notation. After '`g200=3`' a subsequent '`+30`' will set `g201`, `g202`, ..., `g230` equal to 3, but a subsequent '`+30!`' would merely display `g201` through `g230` in decimal notation. Memory addresses will advance by 8 instead of by 1. If `<n>` is empty, the default value $n = 1$ is used.

• `@<x>` sets the address of the next tetrabyte to be simulated, sort of like a `GO` command.

• `t<x>` says that the instruction in tetrabyte location $x$ should always be traced, regardless of its frequency count.

• `u<x>` undoes the effect of `t<x>`.

• `b[rwx]<x>` sets breakpoints at tetrabyte $x$; here `[rwx]` stands for any subset of the letters `r`, `w`, and/or `x`, meaning to break when the tetrabyte is read, written, and/or executed. For example, '`bx1000`' causes a break in the simulation just after the tetrabyte in `#1000` is executed; '`b1000`' undoes this breakpoint; '`brwx1000`' causes a break just after any simulated instruction loads, stores, or appears in tetrabyte number `#1000`.

• `T`, `D`, `P`, `S` sets the "current segment" to `Text_Segment`, `Data_Segment`, `Pool_Segment`, or `Stack_Segment`, respectively, namely to `#0`, `#2000000000000000`, `#4000000000000000`, or `#6000000000000000`. The current segment, initially `#0`, is added to all memory addresses in `M`, `@`, `t`, `u`, and `b` commands.

• `B` lists all current breakpoints and tracepoints.

• `i<filename>` reads a sequence of interactive commands from the specified file, one command per line, ignoring blank lines. This feature can be used to set many breakpoints or to display a number of key

registers, etc. Included lines that begin with `%` or `i` are ignored; therefore an included file cannot include *another* file. Included lines that begin with a blank space are reproduced in the standard output, otherwise ignored.

- `h` (help) reminds the user of the available interactive commands.

**4. Rudimentary I/O.** Input and output are provided by the following ten primitive system calls:

• `Fopen`(*handle*, *name*, *mode*). Here *handle* is a one-byte integer, *name* is the address of the first byte of a string, and *mode* is one of the values `TextRead`, `TextWrite`, `BinaryRead`, `BinaryWrite`, `BinaryReadWrite`. An `Fopen` call associates *handle* with the external file called *name* and prepares to do input and/or output on that file. It returns 0 if the file was opened successfully; otherwise returns the value $-1$. If *mode* is `TextWrite`, `BinaryWrite`, or `BinaryReadWrite`, any previous contents of the named file are discarded. If *mode* is `TextRead` or `TextWrite`, the file consists of "lines" terminated by "newline" characters, and it is said to be a text file; otherwise the file consists of uninterpreted bytes, and it is said to be a binary file.

Text files and binary files are essentially equivalent in cases where this simulator is hosted by an operating system derived from UNIX; in such cases files can be written as text and read as binary or vice versa. But with other operating systems, text files and binary files often have quite different representations, and certain characters with byte codes less than '␣' are forbidden in text. Within any `MMIX` program, the newline character has byte code `#0a` $= 10$.

At the beginning of a program three handles have already been opened: The "standard input" file `StdIn` (handle 0) has mode `TextRead`, the "standard output" file `StdOut` (handle 1) has mode `TextWrite`, and the "standard error" file `StdErr` (handle 2) also has mode `TextWrite`. When this simulator is being run interactively, lines of standard input should be typed following a prompt that says '`StdIn>`␣', unless the `-f` option has been used. The standard output and standard error files of the simulated program are intermixed with the output of the simulator itself.

The input/output operations supported by this simulator can perhaps be understood most easily with reference to the standard library `stdio` that comes with the C language, because the conventions of C have been explained in hundreds of books. If we declare an array **FILE** $*file[256]$ and set $file[0] = stdin$, $file[1] = stdout$, and $file[2] = stderr$, then the simulated system call `Fopen`(*handle*, *name*, *mode*) is essentially equivalent to the C expression

$$(file[handle]?\ (file[handle] = freopen(name, mode\_string[mode], file[handle])):\\ (file[handle] = fopen(name, mode\_string[mode]))?\ 0:\ -1,$$

if we set $mode\_string$[`TextRead`] = `"r"`, $mode\_string$[`TextWrite`] = `"w"`, $mode\_string$[`BinaryRead`] = `"rb"`, $mode\_string$[`BinaryWrite`] = `"wb"`, and $mode\_string$[`BinaryReadWrite`] = `"wb+"`.

• `Fclose`(*handle*). If the given file handle has been opened, it is closed—no longer associated with any file. Again the result is 0 if successful, or $-1$ if the file was already closed or unclosable. The C equivalent is

$$fclose(file[handle])\ ?\ -1 : 0$$

with the additional side effect of setting $file[handle] = \Lambda$.

• `Fread`(*handle*, *buffer*, *size*). The file handle should have been opened with mode `TextRead`, `BinaryRead`, or `BinaryReadWrite`. The next *size* characters are read into `MMIX`'s memory starting at address *buffer*. If an error occurs, the value $-1 - size$ is returned; otherwise, if the end of file does not intervene, 0 is returned; otherwise the negative value $n - size$ is returned, where $n$ is the number of characters successfully read and stored. The statement

$$fread(buffer, 1, size, file[handle]) - size$$

has the equivalent effect in C, in the absence of file errors.

• `Fgets`(*handle*, *buffer*, *size*). The file handle should have been opened with mode `TextRead`, `BinaryRead`, or `BinaryReadWrite`. Characters are read into `MMIX`'s memory starting at address *buffer*, until either $size - 1$ characters have been read and stored or a newline character has been read and stored; the next byte in memory is then set to zero. If an error or end of file occurs before reading is complete, the memory contents are undefined and the value $-1$ is returned; otherwise the number of characters successfully read and stored is returned. The equivalent in C is

$$fgets(buffer, size, file[handle])\ ?\ strlen(buffer) : -1$$

if we assume that no null characters were read in; null characters may, however, precede a newline, and they are counted just like other characters.

• `Fgetws`(*handle*, *buffer*, *size*). This command is the same as `Fgets`, except that it applies to wyde characters instead of one-byte characters. Up to *size* − 1 wyde characters are read; a wyde newline is `#000a`. The C version, using conventions of the ISO multibyte string extension (MSE), is approximately

$$fgetws(buffer, size, file[handle])\ ?\ wcslen(buffer) : -1$$

where *buffer* now has type **wchar_t** ∗.

• `Fwrite`(*handle*, *buffer*, *size*). The file handle should have been opened with one of the modes `TextWrite`, `BinaryWrite`, or `BinaryReadWrite`. The next *size* characters are written from `MMIX`'s memory starting at address *buffer*. If no error occurs, 0 is returned; otherwise the negative value $n - size$ is returned, where $n$ is the number of characters successfully written. The statement

$$fwrite(buffer, 1, size, file[handle]) - size$$

together with *fflush*(*file*[*handle*]) has the equivalent effect in C.

• `Fputs`(*handle*, *string*). The file handle should have been opened with mode `TextWrite`, `BinaryWrite`, or `BinaryReadWrite`. One-byte characters are written from `MMIX`'s memory to the file, starting at address *string*, up to but not including the first byte equal to zero. The number of bytes written is returned, or −1 on error. The C version is

$$fputs(string, file[handle]) \ge 0\ ?\ strlen(string) : -1,$$

together with *fflush*(*file*[*handle*]).

• `Fputws`(*handle*, *string*). The file handle should have been opened with mode `TextWrite`, `BinaryWrite`, or `BinaryReadWrite`. Wyde characters are written from `MMIX`'s memory to the file, starting at address *string*, up to but not including the first wyde equal to zero. The number of wydes written is returned, or −1 on error. The C+MSE version is

$$fputws(string, file[handle]) \ge 0\ ?\ wcslen(string) : -1$$

together with *fflush*(*file*[*handle*]), where *string* now has type **wchar_t** ∗.

• `Fseek`(*handle*, *offset*). The file handle should have been opened with mode `BinaryRead`, `BinaryWrite`, or `BinaryReadWrite`. This operation causes the next input or output operation to begin at *offset* bytes from the beginning of the file, if *offset* ≥ 0, or at −*offset* − 1 bytes before the end of the file, if *offset* < 0. (For example, *offset* = 0 "rewinds" the file to its very beginning; *offset* = −1 moves forward all the way to the end.) The result is 0 if successful, or −1 if the stated positioning could not be done. The C version is

$$fseek(file[handle],\ offset < 0\ ?\ offset + 1 : offset,\ offset < 0\ ?\ \texttt{SEEK\_END} : \texttt{SEEK\_SET})?\ -1 :\ 0.$$

If a file in mode `BinaryReadWrite` is used for both reading and writing, an `Fseek` command must be given when switching from input to output or from output to input.

• `Ftell`(*handle*). The file handle should have been opened with mode `BinaryRead`, `BinaryWrite`, or `BinaryReadWrite`. This operation returns the current file position, measured in bytes from the beginning, or −1 if an error has occurred. In this case the C function

$$ftell(file[handle])$$

has exactly the same meaning.

Although these ten operations are quite primitive, they provide the necessary functionality for extremely complex input/output behavior. For example, every function in the `stdio` library of C, with the exception of the two administrative operations *remove* and *rename*, can be implemented as a subroutine in terms of the six basic operations `Fopen`, `Fclose`, `Fread`, `Fwrite`, `Fseek`, and `Ftell`.

Notice that the `MMIX` function calls are much more consistent than those in the C library. The first argument is always a handle; the second, if present, is always an address; the third, if present, is always a size. *The result returned is always nonnegative if the operation was successful, negative if an anomaly arose.* These common features make the functions reasonably easy to remember.

**5.** The ten input/output operations of the previous section are invoked by `TRAP` commands with X = 0, Y = `Fopen` or `Fclose` or ... or `Ftell`, and Z = `Handle`. If there are two arguments, the second argument is placed in \$255. If there are three arguments, the address of the second is placed in \$255; the second argument is $M_8[\$255]$ and the third argument is $M_8[\$255 + 8]$. The returned value will be in \$255 when the system call is finished. (See the example below.)

**6.** The user program starts at symbolic location `Main`. At this time the global registers are initialized according to the `GREG` statements in the `MMIXAL` program, and \$255 is set to the numeric equivalent of `Main`. Local register \$0 is initially set to the number of *command line arguments*; and local register \$1 points to the first such argument, which is always a pointer to the program name. Each command line argument is a pointer to a string; the last such pointer is $M_8[\$0 \ll 3+\$1-8]$, and $M_8[\$0 \ll 3+\$1]$ is zero. (Register \$1 will point to an octabyte in `Pool_Segment`, and the command line strings will be in that segment too.) Location M[`Pool_Segment`] will be the address of the first unused octabyte of the pool segment.

Registers rA, rB, rD, rE, rF, rH, rI, rJ, rM, rP, rQ, and rR are initially zero, and rL = 2.

A subroutine library loaded with the user program might need to initialize itself. If an instruction has been loaded into tetrabyte $M_4[\#$`f0`$]$, the simulator actually begins execution at `#f0` instead of at `Main`; in this case \$255 holds the location of `Main`. (The routine at `#f0` can pass control to `Main` without increasing rL, if it starts with the slightly tricky sequence

```
PUT rW, $255;  PUT rB, $255;  SETML $255,#F700;  PUT rX,$255
```

and eventually says `RESUME`; this `RESUME` command will restore \$255 and rB. But the user program should *not* really count on the fact that rL is initially 2.)

**7.** The main program ends when `MMIX` executes the system call `TRAP 0`, which is often symbolically written '`TRAP 0,Halt,0`' to make its intention clear. The contents of \$255 at that time are considered to be the value "returned" by the main program, as in the *exit* statement of C; a nonzero value indicates an anomalous exit. All open files are closed when the program ends.

**8.** Here, for example, is a complete program that copies a text file to the standard output, given the name of the file to be copied. It includes all necessary error checking.

```
* SAMPLE PROGRAM: COPY A GIVEN FILE TO STANDARD OUTPUT
t        IS   $255
argc     IS   $0
argv     IS   $1
s        IS   $2
Buf_Size IS   1000
         LOC  Data_Segment
Buffer   LOC  @+Buf_Size
         GREG @
Arg0     OCTA 0,TextRead
Arg1     OCTA Buffer,Buf_Size
         LOC  #200              main(argc,argv) {
Main     CMP  t,argc,2          if (argc==2) goto openit
         PBZ  t,OpenIt
         GETA t,1F              fputs("Usage: ",stderr)
         TRAP 0,Fputs,StdErr
         LDOU t,argv,0          fputs(argv[0],stderr)
         TRAP 0,Fputs,StdErr
         GETA t,2F              fputs(" filename\n",stderr)
Quit     TRAP 0,Fputs,StdErr
         NEG  t,0,1             quit: exit(-1)
         TRAP 0,Halt,0
1H       BYTE "Usage: ",0
         LOC  (@+3)&-4          align to tetrabyte
2H       BYTE " filename",#a,0
OpenIt   LDOU s,argv,8          openit: s=argv[1]
         STOU s,Arg0
         LDA  t,Arg0            fopen(argv[1],"r",file[3])
         TRAP 0,Fopen,3
         PBNN t,CopyIt          if (no error) goto copyit
         GETA t,1F              fputs("Can't open file ",stderr)
         TRAP 0,Fputs,StdErr
         SET  t,s               fputs(argv[1],stderr)
         TRAP 0,Fputs,StdErr
         GETA t,2F              fputs("!\n",stderr)
         JMP  Quit              goto quit
1H       BYTE "Can't open file ",0
         LOC  (@+3)&-4          align to tetrabyte
2H       BYTE "!",#a,0
CopyIt   LDA  t,Arg1            copyit:
         TRAP 0,Fread,3         items=fread(buffer,1,buf_size,file[3])
         BN   t,EndIt           if (items < buf_size) goto endit
         LDA  t,Arg1            items=fwrite(buffer,1,buf_size,stdout)
         TRAP 0,Fwrite,StdOut
         PBNN t,CopyIt          if (items >= buf_size) goto copyit
Trouble  GETA t,1F              trouble: fputs("Trouble w...!",stderr)
         JMP  Quit              goto quit
1H       BYTE "Trouble writing StdOut!",#a,0
EndIt    INCL t,Buf_Size
         BN   t,ReadErr         if (ferror(file[3])) goto readerr
         STO  t,Arg1+8
         LDA  t,Arg1            n=fwrite(buffer,1,items,stdout)
         TRAP 0,Fwrite,StdOut
         BN   t,Trouble         if (n < items) goto trouble
         TRAP 0,Halt,0          exit(0)
ReadErr  GETA t,1F              readerr: fputs("Trouble r...!",stderr)
         JMP  Quit              goto quit }
1H       BYTE "Trouble reading!",#a,0
```

**9. Basics.** To get started, we define a type that provides semantic sugar.

```
⟨ Type declarations 9 ⟩ ≡
  typedef enum {
    false, true
  } bool;
```

See also sections 10, 16, 38, 39, 54, 55, 59, 64, and 135.

This code is used in section 141.

**10.** This program for the 64-bit `MMIX` architecture is based on 32-bit integer arithmetic, because nearly every computer available to the author at the time of writing (1999) was limited in that way. It uses subroutines from the MMIX-ARITH module, assuming only that type **tetra** represents unsigned 32-bit integers. The definition of **tetra** given here should be changed, if necessary, to agree with the definition in that module.

```
⟨ Type declarations 9 ⟩ +≡
  typedef unsigned int tetra;      /* for systems conforming to the LP-64 data model */
  typedef struct {
    tetra h, l;
  } octa;      /* two tetrabytes make one octabyte */
  typedef unsigned char byte;      /* a monobyte */
```

**11.** We declare subroutines twice, once with a prototype and once with the old-style C conventions. The following hack makes this work with new compilers as well as the old standbys.

```
⟨ Preprocessor macros 11 ⟩ ≡
#ifdef __STDC__
#define ARGS(list) list
#else
#define ARGS(list) ()
#endif
```

See also sections 43 and 46.

This code is used in section 141.

**12.**

```
⟨ Subroutines 12 ⟩ ≡
  void print_hex ARGS((octa));
  void print_hex(o)
      octa o;
  {
    if (o.h) printf("%x%08x", o.h, o.l);
    else printf("%x", o.l);
  }
```

See also sections 13, 15, 17, 20, 26, 27, 42, 45, 47, 50, 82, 83, 91, 114, 117, 120, 137, 140, 143, 148, 154, 160, 162, 165, and 166.

This code is used in section 141.

**13.** Most of the subroutines in MMIX-ARITH return an octabyte as a function of two octabytes; for example, *oplus*$(y,z)$ returns the sum of octabytes $y$ and $z$. Division inputs the high half of a dividend in the global variable *aux* and returns the remainder in *aux*.

```
⟨ Subroutines 12 ⟩ +≡
  extern octa zero_octa;      /* zero_octa.h = zero_octa.l = 0 */
  extern octa neg_one;      /* neg_one.h = neg_one.l = -1 */
  extern octa aux, val;      /* auxiliary data */
  extern bool overflow;      /* flag set by signed multiplication and division */
  extern int exceptions;      /* bits set by floating point operations */
  extern int cur_round;      /* the current rounding mode */
  extern char *next_char;      /* where a scanned constant ended */
  extern octa oplus ARGS((octa y, octa z));      /* unsigned y + z */
  extern octa ominus ARGS((octa y, octa z));      /* unsigned y - z */
  extern octa incr ARGS((octa y, int delta));      /* unsigned y + δ (δ is signed) */
  extern octa oand ARGS((octa y, octa z));      /* y ∧ z */
  extern octa shift_left ARGS((octa y, int s));      /* y ≪ s, 0 ≤ s ≤ 64 */
  extern octa shift_right ARGS((octa y, int s, int u));      /* y ≫ s, signed if ¬u */
  extern octa omult ARGS((octa y, octa z));      /* unsigned (aux, x) = y × z */
  extern octa signed_omult ARGS((octa y, octa z));      /* signed x = y × z */
  extern octa odiv ARGS((octa x, octa y, octa z));      /* unsigned (x, y)/z; aux = (x, y) mod z */
  extern octa signed_odiv ARGS((octa y, octa z));      /* signed x = y/z */
  extern int count_bits ARGS((tetra z));      /* x = ν(z) */
  extern tetra byte_diff ARGS((tetra y, tetra z));      /* half of BDIF */
  extern tetra wyde_diff ARGS((tetra y, tetra z));      /* half of WDIF */
  extern octa bool_mult ARGS((octa y, octa z, bool xor));      /* MOR or MXOR */
  extern octa load_sf ARGS((tetra z));      /* load short float */
  extern tetra store_sf ARGS((octa x));      /* store short float */
  extern octa fplus ARGS((octa y, octa z));      /* floating point x = y ⊕ z */
  extern octa fmult ARGS((octa y, octa z));      /* floating point x = y ⊗ z */
  extern octa fdivide ARGS((octa y, octa z));      /* floating point x = y ⊘ z */
  extern octa froot ARGS((octa, int));      /* floating point x = √z */
  extern octa fremstep ARGS((octa y, octa z, int delta));      /* floating point x rem z = y rem z */
  extern octa fintegerize ARGS((octa z, int mode));      /* floating point x = round(z) */
  extern int fcomp ARGS((octa y, octa z));      /* -1, 0, 1, or 2 if y < z, y = z, y > z, y ‖ z */
  extern int fepscomp ARGS((octa y, octa z, octa eps, int sim));
    /* x = sim? [y ~ z (ϵ)] : [y ≈ z (ϵ)] */
  extern octa floatit ARGS((octa z, int mode, int unsgnd, int shrt));      /* fix to float */
  extern octa fixit ARGS((octa z, int mode));      /* float to fix */
  extern void print_float ARGS((octa z));      /* print octabyte as floating decimal */
  extern int scan_const ARGS((char *buf));      /* val = floating or integer constant; returns the type */
```

**14.** Here's a quick check to see if arithmetic is in trouble.

```
#define panic(m)
        { fprintf(stderr, "Panic: %s!\n", m); exit(-2); }
⟨ Initialize everything 14 ⟩ ≡
  if (shift_left(neg_one, 1).h ≠ #ffffffff) panic("Incorrect implementation of type tetra");
```

See also sections 18, 24, 32, 41, 77, and 147.

This code is used in section 141.

**15.** Binary-to-decimal conversion is used when we want to see an octabyte as a signed integer. The identity $\lfloor (an+b)/10 \rfloor = \lfloor a/10 \rfloor n + \lfloor ((a \bmod 10)n + b)/10 \rfloor$ is helpful here.

```
#define sign_bit ((unsigned) #80000000)
⟨ Subroutines 12 ⟩ +≡
  void print_int ARGS((octa));
  void print_int(o)
      octa o;
  {
    register tetra hi = o.h, lo = o.l, r, t;
    register int j;
    char dig[20];
    if (lo ≡ 0 ∧ hi ≡ 0) printf("0");
    else {
      if (hi & sign_bit) {
        printf("-");
        if (lo ≡ 0) hi = −hi;
        else lo = −lo, hi = ∼hi;
      }
      for (j = 0; hi; j++) {      /* 64-bit division by 10 */
        r = ((hi % 10) ≪ 16) + (lo ≫ 16);
        hi = hi/10;
        t = ((r % 10) ≪ 16) + (lo & #ffff);
        lo = ((r/10) ≪ 16) + (t/10);
        dig[j] = t % 10;
      }
      for ( ; lo; j++) {
        dig[j] = lo % 10;
        lo = lo/10;
      }
      for (j−−; j ≥ 0; j−−) printf("%c", dig[j] + '0');
    }
  }
```

**16. Simulated memory.** Chunks of simulated memory, 2048 bytes each, are kept in a tree structure organized as a *treap*, following ideas of Vuillemin, Aragon, and Seidel [*Communications of the ACM* **23** (1980), 229–239; *IEEE Symp. on Foundations of Computer Science* **30** (1989), 540–546]. Each node of the treap has two keys: One, called *loc*, is the base address of 512 simulated tetrabytes; it follows the conventions of an ordinary binary search tree, with all locations in the left subtree less than the *loc* of a node and all locations in the right subtree greater than that *loc*. The other, called *stamp*, can be thought of as the time the node was inserted into the tree; all subnodes of a given node have a larger *stamp*. By assigning time stamps at random, we maintain a tree structure that almost always is fairly well balanced.

Each simulated tetrabyte has an associated frequency count and source file reference.

```
⟨ Type declarations 9 ⟩ +≡
  typedef struct {
    tetra tet;      /* the tetrabyte of simulated memory */
    tetra freq;     /* the number of times it was obeyed as an instruction */
    unsigned char bkpt;      /* breakpoint information for this tetrabyte */
    unsigned char file_no;      /* source file number, if known */
    unsigned short line_no;      /* source line number, if known */
  } mem_tetra;
  typedef struct mem_node_struct {
    octa loc;      /* location of the first of 512 simulated tetrabytes */
    tetra stamp;      /* time stamp for treap balancing */
    struct mem_node_struct *left, *right;      /* pointers to subtrees */
    mem_tetra dat[512];      /* the chunk of simulated tetrabytes */
  } mem_node;
```

**17.** The *stamp* value is actually only pseudorandom, based on the idea of Fibonacci hashing [see *Sorting and Searching*, Section 6.4]. This is good enough for our purposes, and it guarantees that no two stamps will be identical.

```
⟨ Subroutines 12 ⟩ +≡
  mem_node *new_mem ARGS((void));
  mem_node *new_mem()
  {
    register mem_node *p;
    p = (mem_node *) calloc(1, sizeof(mem_node));
    if (¬p) panic("Can't␣allocate␣any␣more␣memory");
    p→stamp = priority;
    priority += #9e3779b9;      /* ⌊2^32(φ − 1)⌋ */
    return p;
  }
```

**18.** Initially we start with a chunk for the pool segment, since the simulator will be putting command line information there before it runs the program.

```
⟨ Initialize everything 14 ⟩ +≡
  mem_root = new_mem();
  mem_root→loc.h = #40000000;
  last_mem = mem_root;
```

**19.** ⟨ Global variables 19 ⟩ ≡

```
tetra priority = 314159265;     /* pseudorandom time stamp counter */
mem_node *mem_root;     /* root of the treap */
mem_node *last_mem;     /* the memory node most recently read or written */
octa sclock;     /* simulated clock */
```

See also sections 25, 31, 40, 48, 52, 56, 61, 65, 76, 110, 113, 121, 129, 139, 144, and 151.

This code is used in section 141.

**20.** The *mem_find* routine finds a given tetrabyte in the simulated memory, inserting a new node into the treap if necessary.

⟨ Subroutines 12 ⟩ +≡

```
mem_tetra *mem_find ARGS((octa));
mem_tetra *mem_find(addr)
    octa addr;
{
  octa key;
  register int offset;
  register mem_node *p = last_mem;
  key.h = addr.h;
  key.l = addr.l & #fffff800;
  offset = addr.l & #7fc;
  if (p->loc.l ≠ key.l ∨ p->loc.h ≠ key.h)
    ⟨ Search for key in the treap, setting last_mem and p to its location 21 ⟩;
  return &p->dat[offset ≫ 2];
}
```

**21.** ⟨ Search for *key* in the treap, setting *last_mem* and *p* to its location 21 ⟩ ≡

```
{ register mem_node **q;
  for (p = mem_root; p; ) {
    if (key.l ≡ p->loc.l ∧ key.h ≡ p->loc.h) goto found;
    if ((key.l < p->loc.l ∧ key.h ≤ p->loc.h) ∨ key.h < p->loc.h) p = p->left;
    else p = p->right;
  }
  for (p = mem_root, q = &mem_root; p ∧ p->stamp < priority; p = *q) {
    if ((key.l < p->loc.l ∧ key.h ≤ p->loc.h) ∨ key.h < p->loc.h) q = &p->left;
    else q = &p->right;
  }
  *q = new_mem();
  (*q)->loc = key;
  ⟨ Fix up the subtrees of *q 22 ⟩;
  p = *q;
found: last_mem = p;
}
```

This code is used in section 20.

**22.** At this point we want to split the binary search tree $p$ into two parts based on the given $key$, forming the left and right subtrees of the new node $q$. The effect will be as if $key$ had been inserted before all of $p$'s nodes.

```
⟨ Fix up the subtrees of *q 22 ⟩ ≡
  {
    register mem_node **l = &(*q)→left, **r = &(*q)→right;
    while (p) {
      if ((key.l < p→loc.l ∧ key.h ≤ p→loc.h) ∨ key.h < p→loc.h) *r = p, r = &p→left, p = *r;
      else *l = p, l = &p→right, p = *l;
    }
    *l = *r = Λ;
  }
```

This code is used in section 21.

**23. Loading an object file.** To get the user's program into memory, we read in an `MMIX` object, using modifications of the routines in the utility program `MMOtype`. Complete details of `mmo` format appear in the program for MMIXAL; a reader who hopes to understand this section ought to at least skim that documentation. Here we need to define only the basic constants used for interpretation.

```
#define mm #98 /* the escape code of mmo format */
#define lop_quote #0 /* the quotation lopcode */
#define lop_loc #1 /* the location lopcode */
#define lop_skip #2 /* the skip lopcode */
#define lop_fixo #3 /* the octabyte-fix lopcode */
#define lop_fixr #4 /* the relative-fix lopcode */
#define lop_fixrx #5 /* extended relative-fix lopcode */
#define lop_file #6 /* the file name lopcode */
#define lop_line #7 /* the file position lopcode */
#define lop_spec #8 /* the special hook lopcode */
#define lop_pre #9 /* the preamble lopcode */
#define lop_post #a /* the postamble lopcode */
#define lop_stab #b /* the symbol table lopcode */
#define lop_end #c /* the end-it-all lopcode */
```

**24.** We do not load the symbol table. (A more ambitious simulator could implement `MMIXAL`-style expressions for interactive debugging, but such enhancements are left to the interested reader.)

```
⟨ Initialize everything 14 ⟩ +≡
  mmo_file = fopen(mmo_file_name, "rb");
  if (¬mmo_file) {
    register char *alt_name = (char *) calloc(strlen(mmo_file_name) + 5, sizeof(char));
    if (¬alt_name) panic("Can't␣allocate␣file␣name␣buffer");
    sprintf(alt_name, "%s.mmo", mmo_file_name);
    mmo_file = fopen(alt_name, "rb");
    if (¬mmo_file) {
      fprintf(stderr, "Can't␣open␣the␣object␣file␣%s␣or␣%s!\n", mmo_file_name, alt_name);
      exit(-3);
    }
    free(alt_name);
  }
  byte_count = 0;
```

**25.** ⟨ Global variables 19 ⟩ +≡

```
  FILE *mmo_file; /* the input file */
  int postamble; /* have we encountered lop_post? */
  int byte_count; /* index of the next-to-be-read byte */
  byte buf[4]; /* the most recently read bytes */
  int yzbytes; /* the two least significant bytes */
  int delta; /* difference for relative fixup */
  tetra tet; /* buf bytes packed big-endianwise */
```

**26.** The tetrabytes of an `mmo` file are stored in friendly big-endian fashion, but this program is supposed to work also on computers that are little-endian. Therefore we read four successive bytes and pack them into a tetrabyte, instead of reading a single tetrabyte.

```
#define mmo_err
        {
          fprintf(stderr,"Bad␣object␣file!␣(Try␣running␣MMOtype.)\n");
          exit(-4);
        }
⟨ Subroutines 12 ⟩ +≡
  void read_tet ARGS((void));
  void read_tet()
  {
    if (fread(buf,1,4,mmo_file) ≠ 4) mmo_err;
    yzbytes = (buf[2] ≪ 8) + buf[3];
    tet = (((buf[0] ≪ 8) + buf[1]) ≪ 16) + yzbytes;
  }
```

**27.**
```
⟨ Subroutines 12 ⟩ +≡
  byte read_byte ARGS((void));
  byte read_byte()
  {
    register byte b;
    if (¬byte_count) read_tet();
    b = buf[byte_count];
    byte_count = (byte_count + 1) & 3;
    return b;
  }
```

**28.**
```
⟨ Load the preamble 28 ⟩ ≡
  read_tet();      /* read the first tetrabyte of input */
  if (buf[0] ≠ mm ∨ buf[1] ≠ lop_pre) mmo_err;
  if (ybyte ≠ 1) mmo_err;
  if (zbyte ≡ 0) obj_time = #ffffffff;
  else {
    j = zbyte - 1;
    read_tet(); obj_time = tet;      /* file creation time */
    for ( ; j > 0; j--) read_tet();
  }
```

This code is used in section 32.

**29.** ⟨ Load the next item 29 ⟩ ≡

```
{
  read_tet();
loop: if (buf[0] ≡ mm)
    switch (buf[1]) {
    case lop_quote: if (yzbytes ≠ 1) mmo_err;
      read_tet(); break;
    ⟨ Cases for lopcodes in the main loop 33 ⟩
    case lop_post: postamble = 1;
      if (ybyte ∨ zbyte < 32) mmo_err;
      continue;
    default: mmo_err;
    }
  ⟨ Load tet as a normal item 30 ⟩;
}
```

This code is used in section 32.

**30.** In a normal situation, the newly read tetrabyte is simply supposed to be loaded into the current location. We load not only the current location but also the current file position, if *cur_line* is nonzero and *cur_loc* belongs to segment 0.

```
#define mmo_load(loc, val) ll = mem_find(loc), ll→tet ⊕= val
```

⟨ Load *tet* as a normal item 30 ⟩ ≡

```
{
  mmo_load(cur_loc, tet);
  if (cur_line) {
    ll→file_no = cur_file;
    ll→line_no = cur_line;
    cur_line++;
  }
  cur_loc = incr(cur_loc, 4); cur_loc.l &= −4;
}
```

This code is used in section 29.

**31.** ⟨ Global variables 19 ⟩ +≡

```
octa cur_loc;      /* the current location */
int cur_file = −1;     /* the most recently selected file number */
int cur_line;      /* the current position in cur_file, if nonzero */
octa tmp;      /* an octabyte of temporary interest */
tetra obj_time;      /* when the object file was created */
```

**32.** ⟨ Initialize everything 14 ⟩ +≡

```
cur_loc.h = cur_loc.l = 0;
cur_file = −1;
cur_line = 0;
⟨ Load the preamble 28 ⟩;
do ⟨ Load the next item 29 ⟩ while (¬postamble);
⟨ Load the postamble 37 ⟩;
fclose(mmo_file);
cur_line = 0;
```

**33.** We have already implemented *lop_quote*, which falls through to the normal case after reading an extra tetrabyte. Now let's consider the other lopcodes in turn.

```
#define ybyte buf[2]    /* the next-to-least significant byte */
#define zbyte buf[3]    /* the least significant byte */
⟨ Cases for lopcodes in the main loop 33 ⟩ ≡
case lop_loc: if (zbyte ≡ 2) {
    j = ybyte; read_tet(); cur_loc.h = (j ≪ 24) + tet;
  } else if (zbyte ≡ 1) cur_loc.h = ybyte ≪ 24;
  else mmo_err;
  read_tet(); cur_loc.l = tet;
  continue;
case lop_skip: cur_loc = incr(cur_loc, yzbytes); continue;
```

See also sections 34, 35, and 36.

This code is used in section 29.

**34.** Fixups load information out of order, when future references have been resolved. The current file name and line number are not considered relevant.

```
⟨ Cases for lopcodes in the main loop 33 ⟩ +≡
case lop_fixo: if (zbyte ≡ 2) {
    j = ybyte; read_tet(); tmp.h = (j ≪ 24) + tet;
  } else if (zbyte ≡ 1) tmp.h = ybyte ≪ 24;
  else mmo_err;
  read_tet(); tmp.l = tet;
  mmo_load(tmp, cur_loc.h);
  mmo_load(incr(tmp, 4), cur_loc.l);
  continue;
case lop_fixr: mmo_load(incr(cur_loc, -yzbytes ≪ 2), yzbytes);
  continue;
case lop_fixrx: j = yzbytes; if (j ≠ 16 ∧ j ≠ 24) mmo_err;
  read_tet(); if (tet & #fe000000) mmo_err;
  delta = (tet ≥ #1000000 ? (tet & #ffffff) − (1 ≪ j) : tet);
  mmo_load(incr(cur_loc, -delta ≪ 2), tet);
  continue;
```

**35.** The space for file names isn't allocated until we are sure we need it.

```
⟨ Cases for lopcodes in the main loop 33 ⟩ +≡
case lop_file: if (file_info[ybyte].name) {
    if (zbyte) mmo_err;
    cur_file = ybyte;
  } else {
    if (¬zbyte) mmo_err;
    file_info[ybyte].name = (char *) calloc(4 * zbyte + 1, 1);
    if (¬file_info[ybyte].name) {
      fprintf(stderr, "No room to store the file name!\n"); exit(−5);
    }
    cur_file = ybyte;
    for (j = zbyte, p = file_info[ybyte].name; j > 0; j−−, p += 4) {
      read_tet();
      *p = buf[0]; *(p + 1) = buf[1]; *(p + 2) = buf[2]; *(p + 3) = buf[3];
    }
  }
  cur_line = 0; continue;
case lop_line: if (cur_file < 0) mmo_err;
  cur_line = yzbytes; continue;
```

**36.** Special bytes are ignored (at least for now).

```
⟨ Cases for lopcodes in the main loop 33 ⟩ +≡
case lop_spec: while (1) {
    read_tet();
    if (buf[0] ≡ mm) {
      if (buf[1] ≠ lop_quote ∨ yzbytes ≠ 1) goto loop;      /* end of special data */
      read_tet();
    }
  }
```

**37.** Since a chunk of memory holds 512 tetrabytes, the *ll* pointer in the following loop stays in the same chunk (namely, the first chunk of segment 3, also known as `Stack_Segment`).

```
⟨ Load the postamble 37 ⟩ ≡
  aux.h = #60000000; aux.l = #18;
  ll = mem_find(aux);
  (ll − 1)→tet = 2;      /* this will ultimately set rL = 2 */
  (ll − 5)→tet = argc;      /* and $0 = argc */
  (ll − 4)→tet = #40000000;
  (ll − 3)→tet = #8;      /* and $1 = Pool_Segment + 8 */
  G = zbyte; L = 0; O = 0;
  for (j = G + G; j < 256 + 256; j++, ll++, aux.l += 4) read_tet(), ll→tet = tet;
  inst_ptr.h = (ll − 2)→tet, inst_ptr.l = (ll − 1)→tet;      /* Main */
  (ll + 2 * 12)→tet = G ≪ 24;
  g[255] = incr(aux, 12 * 8);      /* we will UNSAVE from here, to get going */
```

This code is used in section 32.

**38. Loading and printing source lines.** The loaded program generally contains cross references to the lines of symbolic source files, so that the context of each instruction can be understood. The following sections of this program make such information available when it is desired.

Source file data is kept in a **file_node** structure:

```
⟨ Type declarations 9 ⟩ +≡
  typedef struct {
    char *name;      /* name of source file */
    int line_count;      /* number of lines in the file */
    long *map;      /* pointer to map of file positions */
  } file_node;
```

**39.** In partial preparation for the day when source files are in Unicode, we define a type **Char** for the source characters.

```
⟨ Type declarations 9 ⟩ +≡
  typedef char Char;      /* bytes that will become wydes some day */
```

**40.**
```
⟨ Global variables 19 ⟩ +≡
  file_node file_info[256];      /* data about each source file */
  int buf_size;      /* size of buffer for source lines */
  Char *buffer;
```

**41.** As in `MMIXAL`, we prefer source lines of length 72 characters or less, but the user is allowed to increase the limit. (Longer lines will silently be truncated to the buffer size when the simulator lists them.)

```
⟨ Initialize everything 14 ⟩ +≡
  if (buf_size < 72) buf_size = 72;
  buffer = (Char *) calloc(buf_size + 1, sizeof(Char));
  if (¬buffer) panic("Can't␣allocate␣source␣line␣buffer");
```

**42.** The first time we are called upon to list a line from a given source file, we make a map of starting locations for each line. Source files should contain at most 65535 lines. We assume that they contain no null characters.

```
⟨ Subroutines 12 ⟩ +≡
  void make_map ARGS((void));
  void make_map()
  {
    long map[65536];
    register int k, l;
    register long *p;
    ⟨ Check if the source file has been modified 44 ⟩;
    for (l = 1; l < 65536 ∧ ¬feof(src_file); l++) {
      map[l] = ftell(src_file);
    loop: if (¬fgets(buffer, buf_size, src_file)) break;
      if (buffer[strlen(buffer) − 1] ≠ '\n') goto loop;
    }
    file_info[cur_file].line_count = l;
    file_info[cur_file].map = p = (long *) calloc(l, sizeof(long));
    if (¬p) panic("No␣room␣for␣a␣source-line␣map");
    for (k = 1; k < l; k++) p[k] = map[k];
  }
```

**43.** We want to warn the user if the source file has changed since the object file was written. The standard C library doesn't provide the information we need; so we use the UNIX system function *stat*, in hopes that other operating systems provide a similar way to do the job.

```
⟨ Preprocessor macros 11 ⟩ +≡
#include <sys/types.h>
#include <sys/stat.h>
```

**44.**
```
⟨ Check if the source file has been modified 44 ⟩ ≡
  {
    struct stat stat_buf;
    if (stat(file_info[cur_file].name, &stat_buf) ≥ 0)
      if ((tetra) stat_buf.st_mtime > obj_time)
        fprintf(stderr, "Warning: File %s was modified; it may not match the program!\n",
            file_info[cur_file].name);
  }
```
This code is used in section 42.

**45.** Source lines are listed by the *print_line* routine, preceded by 12 characters containing the line number. If a file error occurs, nothing is printed—not even an error message; the absence of listed data is itself a message.

```
⟨ Subroutines 12 ⟩ +≡
  void print_line ARGS((int));
  void print_line(k)
      int k;
  {
    char buf[11];
    if (k ≥ file_info[cur_file].line_count) return;
    if (fseek(src_file, file_info[cur_file].map[k], SEEK_SET) ≠ 0) return;
    if (¬fgets(buffer, buf_size, src_file)) return;
    sprintf(buf, "%d:    ", k);
    printf("line %.6s %s", buf, buffer);
    if (buffer[strlen(buffer) − 1] ≠ '\n') printf("\n");
    line_shown = true;
  }
```

**46.**
```
⟨ Preprocessor macros 11 ⟩ +≡
#ifndef SEEK_SET
#define SEEK_SET 0      /* code for setting the file pointer to a given offset */
#endif
```

**47.** The *show_line* routine is called when we want to output line *cur_line* of source file number *cur_file*, assuming that *cur_line* $\neq 0$. Its job is primarily to maintain continuity, by opening or reopening the *src_file* if the source file changes, and by connecting the previously output lines to the new one. Sometimes no output is necessary, because the desired line has already been printed.

```
⟨ Subroutines 12 ⟩ +≡
  void show_line ARGS((void));
  void show_line()
  {
    register int k;
    if (shown_file ≠ cur_file) ⟨ Prepare to list lines from a new source file 49 ⟩
    else if (shown_line ≡ cur_line) return;      /* already shown */
    if (cur_line > shown_line + gap + 1 ∨ cur_line < shown_line) {
      if (shown_line > 0) {
        if (cur_line < shown_line) printf("--------\n");      /* indicate upward move */
        else printf("␣␣␣␣␣...\n"); }      /* indicate the gap */
      print_line(cur_line);
    } else for (k = shown_line + 1; k ≤ cur_line; k++) print_line(k);
    shown_line = cur_line;
  }
```

**48.**
```
⟨ Global variables 19 ⟩ +≡
  FILE *src_file;      /* the currently open source file */
  int shown_file = -1;      /* index of the most recently listed file */
  int shown_line;      /* the line most recently listed in shown_file */
  int gap;      /* minimum gap between consecutively listed source lines */
  bool line_shown;      /* did we list anything recently? */
  bool showing_source;      /* are we listing source lines? */
  int profile_gap;      /* the gap when printing final frequencies */
  bool profile_showing_source;      /* showing_source within final frequencies */
```

**49.**
```
⟨ Prepare to list lines from a new source file 49 ⟩ ≡
  {
    if (¬src_file) src_file = fopen(file_info[cur_file].name, "r");
    else freopen(file_info[cur_file].name, "r", src_file);
    if (¬src_file) {
      fprintf(stderr, "Warning:␣I␣can't␣open␣file␣%s;␣source␣listing␣omitted.\n",
          file_info[cur_file].name);
      showing_source = false;
      return;
    }
    printf("\"%s\"\n", file_info[cur_file].name);
    shown_file = cur_file;
    shown_line = 0;
    if (¬file_info[cur_file].map) make_map();
  }
```

This code is used in section 47.

**50.** Here is a simple application of *show_line*. It is a recursive routine that prints the frequency counts of all instructions that occur in a given subtree of the simulated memory and that were executed at least once. The subtree is traversed in symmetric order; therefore the frequencies appear in increasing order of the instruction locations.

```
⟨ Subroutines 12 ⟩ +≡
  void print_freqs ARGS((mem_node *));
  void print_freqs(p)
      mem_node *p;
  {
    register int j;
    octa cur_loc;
    if (p→left) print_freqs(p→left);
    for (j = 0; j < 512; j++)
      if (p→dat[j].freq) ⟨ Print frequency data for location p→loc + 4 * j 51 ⟩;
    if (p→right) print_freqs(p→right);
  }
```

**51.** An ellipsis (. . .) is printed between frequency data for nonconsecutive instructions, unless source line information intervenes.

```
⟨ Print frequency data for location p→loc + 4 * j 51 ⟩ ≡
  {
    cur_loc = incr(p→loc, 4 * j);
    if (showing_source ∧ p→dat[j].line_no) {
      cur_file = p→dat[j].file_no, cur_line = p→dat[j].line_no;
      line_shown = false;
      show_line();
      if (line_shown) goto loc_implied;
    }
    if (cur_loc.l ≠ implied_loc.l ∨ cur_loc.h ≠ implied_loc.h)
      if (profile_started) printf("␣␣␣␣␣␣␣␣␣0.␣␣␣␣␣␣␣␣␣...\n");
  loc_implied: printf("%10d.␣%08x%08x:␣%08x␣(%s)\n", p→dat[j].freq, cur_loc.h, cur_loc.l, p→dat[j].tet,
        info[p→dat[j].tet ≫ 24].name);
    implied_loc = incr(cur_loc, 4); profile_started = true;
  }
```

This code is used in section 50.

**52.** ⟨ Global variables 19 ⟩ +≡

```
  octa implied_loc;      /* location following the last shown frequency data */
  bool profile_started;  /* have we printed at least one frequency count? */
```

**53.** ⟨ Print all the frequency counts 53 ⟩ ≡

```
  {
    printf("\nProgram␣profile:\n");
    shown_file = cur_file = −1; shown_line = cur_line = 0;
    gap = profile_gap;
    showing_source = profile_showing_source;
    implied_loc = neg_one;
    print_freqs(mem_root);
  }
```

This code is used in section 141.

**54. Lists.** This simulator needs to deal with 256 different opcodes, so we might as well enumerate them now.

⟨ Type declarations 9 ⟩ +≡

```
typedef enum {
  TRAP, FCMP, FUN, FEQL, FADD, FIX, FSUB, FIXU,
  FLOT, FLOTI, FLOTU, FLOTUI, SFLOT, SFLOTI, SFLOTU, SFLOTUI,
  FMUL, FCMPE, FUNE, FEQLE, FDIV, FSQRT, FREM, FINT,
  MUL, MULI, MULU, MULUI, DIV, DIVI, DIVU, DIVUI,
  ADD, ADDI, ADDU, ADDUI, SUB, SUBI, SUBU, SUBUI,
  IIADDU, IIADDUI, IVADDU, IVADDUI, VIIIADDU, VIIIADDUI, XVIADDU, XVIADDUI,
  CMP, CMPI, CMPU, CMPUI, NEG, NEGI, NEGU, NEGUI,
  SL, SLI, SLU, SLUI, SR, SRI, SRU, SRUI,
  BN, BNB, BZ, BZB, BP, BPB, BOD, BODB,
  BNN, BNNB, BNZ, BNZB, BNP, BNPB, BEV, BEVB,
  PBN, PBNB, PBZ, PBZB, PBP, PBPB, PBOD, PBODB,
  PBNN, PBNNB, PBNZ, PBNZB, PBNP, PBNPB, PBEV, PBEVB,
  CSN, CSNI, CSZ, CSZI, CSP, CSPI, CSOD, CSODI,
  CSNN, CSNNI, CSNZ, CSNZI, CSNP, CSNPI, CSEV, CSEVI,
  ZSN, ZSNI, ZSZ, ZSZI, ZSP, ZSPI, ZSOD, ZSODI,
  ZSNN, ZSNNI, ZSNZ, ZSNZI, ZSNP, ZSNPI, ZSEV, ZSEVI,
  LDB, LDBI, LDBU, LDBUI, LDW, LDWI, LDWU, LDWUI,
  LDT, LDTI, LDTU, LDTUI, LDO, LDOI, LDOU, LDOUI,
  LDSF, LDSFI, LDHT, LDHTI, CSWAP, CSWAPI, LDUNC, LDUNCI,
  LDVTS, LDVTSI, PRELD, PRELDI, PREGO, PREGOI, GO, GOI,
  STB, STBI, STBU, STBUI, STW, STWI, STWU, STWUI,
  STT, STTI, STTU, STTUI, STO, STOI, STOU, STOUI,
  STSF, STSFI, STHT, STHTI, STCO, STCOI, STUNC, STUNCI,
  SYNCD, SYNCDI, PREST, PRESTI, SYNCID, SYNCIDI, PUSHGO, PUSHGOI,
  OR, ORI, ORN, ORNI, NOR, NORI, XOR, XORI,
  AND, ANDI, ANDN, ANDNI, NAND, NANDI, NXOR, NXORI,
  BDIF, BDIFI, WDIF, WDIFI, TDIF, TDIFI, ODIF, ODIFI,
  MUX, MUXI, SADD, SADDI, MOR, MORI, MXOR, MXORI,
  SETH, SETMH, SETML, SETL, INCH, INCMH, INCML, INCL,
  ORH, ORMH, ORML, ORL, ANDNH, ANDNMH, ANDNML, ANDNL,
  JMP, JMPB, PUSHJ, PUSHJB, GETA, GETAB, PUT, PUTI,
  POP, RESUME, SAVE, UNSAVE, SYNC, SWYM, GET, TRIP
} mmix_opcode;
```

**55.** We also need to enumerate the special names for special registers.

⟨ Type declarations 9 ⟩ +≡

```
typedef enum {
  rB, rD, rE, rH, rJ, rM, rR, rBB, rC, rN, rO, rS, rI, rT, rTT, rK, rQ, rU, rV, rG, rL, rA, rF, rP, rW,
      rX, rY, rZ, rWW, rXX, rYY, rZZ
} special_reg;
```

**56.** ⟨ Global variables 19 ⟩ +≡

```
char *special_name[32] = {"rB", "rD", "rE", "rH", "rJ", "rM", "rR", "rBB", "rC", "rN", "rO", "rS",
    "rI", "rT", "rTT", "rK", "rQ", "rU", "rV", "rG", "rL", "rA", "rF", "rP", "rW", "rX", "rY", "rZ",
    "rWW", "rXX", "rYY", "rZZ"};
```

**57.** Here are the bit codes for arithmetic exceptions. These codes, except `H_BIT`, are defined also in MMIX-ARITH.

```
#define X_BIT (1 ≪ 8)    /* floating inexact */
#define Z_BIT (1 ≪ 9)    /* floating division by zero */
#define U_BIT (1 ≪ 10)   /* floating underflow */
#define O_BIT (1 ≪ 11)   /* floating overflow */
#define I_BIT (1 ≪ 12)   /* floating invalid operation */
#define W_BIT (1 ≪ 13)   /* float-to-fix overflow */
#define V_BIT (1 ≪ 14)   /* integer overflow */
#define D_BIT (1 ≪ 15)   /* integer divide check */
#define H_BIT (1 ≪ 16)   /* trip */
```

**58.** The *bkpt* field associated with each tetrabyte of memory has bits associated with forced tracing and/or breaking for reading, writing, and/or execution.

```
#define trace_bit (1 ≪ 3)
#define read_bit (1 ≪ 2)
#define write_bit (1 ≪ 1)
#define exec_bit (1 ≪ 0)
```

**59.** To complete our lists of lists, we enumerate the rudimentary operating system calls that are built in to `MMIXAL`.

```
#define max_sys_call Ftell
⟨ Type declarations 9 ⟩ +≡
  typedef enum {
    Halt, Fopen, Fclose, Fread, Fgets, Fgetws, Fwrite, Fputs, Fputws, Fseek, Ftell
  } sys_call;
```

**60. The main loop.** Now let's plunge in to the guts of the simulator, the master switch that controls most of the action.

```
⟨ Perform one instruction 60 ⟩ ≡
  {
    if (resuming) loc = incr(inst_ptr, −4), inst = g[rX].l;
    else ⟨ Fetch the next instruction 63 ⟩;
    op = inst ≫ 24; xx = (inst ≫ 16) & #ff; yy = (inst ≫ 8) & #ff; zz = inst & #ff;
    f = info[op].flags; yz = inst & #ffff;
    x = y = z = a = b = zero_octa; exc = 0; old_L = L;
    if (f & rel_addr_bit) ⟨ Convert relative address to absolute address 70 ⟩;
    ⟨ Install operand fields 71 ⟩;
    if (f & X_is_dest_bit) ⟨ Install register X as the destination, adjusting the register stack if necessary 80 ⟩;
    w = oplus(y, z);
    if (loc.h ≥ #20000000) goto privileged_inst;
    switch (op) {
    ⟨ Cases for individual MMIX instructions 84 ⟩;
    }
    ⟨ Check for trip interrupt 122 ⟩;
    ⟨ Update the clocks 127 ⟩;
    ⟨ Trace the current instruction, if requested 128 ⟩;
    if (resuming ∧ op ≠ RESUME) resuming = false;
  }
```

This code is used in section 141.

**61.** Operands $x$ and $a$ are usually destinations (results), computed from the source operands $y$, $z$, and/or $b$.

```
⟨ Global variables 19 ⟩ +≡
  octa w, x, y, z, a, b, ma, mb;      /* operands */
  octa *x_ptr;      /* destination */
  octa loc;      /* location of the current instruction */
  octa inst_ptr;      /* location of the next instruction */
  tetra inst;      /* the current instruction */
  int old_L;      /* value of L before the current instruction */
  int exc;      /* exceptions raised by the current instruction */
  unsigned int tracing_exceptions;      /* exception bits that cause tracing */
  int rop;      /* ropcode of a resumed instruction */
  int round_mode;      /* the style of floating point rounding just used */
  bool resuming;      /* are we resuming an interrupted instruction? */
  bool halted;      /* did the program come to a halt? */
  bool breakpoint;      /* should we pause after the current instruction? */
  bool tracing;      /* should we trace the current instruction? */
  bool stack_tracing;      /* should we trace details of the register stack? */
  bool interacting;      /* are we in interactive mode? */
  bool interact_after_break;      /* should we go into interactive mode? */
  bool tripping;      /* are we about to go to a trip handler? */
  bool good;      /* did the last branch instruction guess correctly? */
  tetra trace_threshold;      /* each instruction should be traced this many times */
```

**62.** ⟨ Local registers 62 ⟩ ≡

```
register mmix_opcode op;      /* operation code of the current instruction */
register int xx, yy, zz, yz;      /* operand fields of the current instruction */
register tetra f;      /* properties of the current op */
register int i, j, k;      /* miscellaneous indices */
register mem_tetra *ll;      /* current place in the simulated memory */
register char *p;      /* current place in a string */
```

See also section 75.

This code is used in section 141.

**63.** ⟨ Fetch the next instruction 63 ⟩ ≡

```
{
  loc = inst_ptr;
  ll = mem_find(loc);
  inst = ll→tet;
  cur_file = ll→file_no;
  cur_line = ll→line_no;
  ll→freq++;
  if (ll→bkpt & exec_bit) breakpoint = true;
  tracing = breakpoint ∨ (ll→bkpt & trace_bit) ∨ (ll→freq ≤ trace_threshold);
  inst_ptr = incr(inst_ptr, 4);
}
```

This code is used in section 60.

**64.** Much of the simulation is table-driven, based on a static data structure called the **op_info** for each operation code.

⟨ Type declarations 9 ⟩ +≡

```
typedef struct {
  char *name;      /* symbolic name of an opcode */
  unsigned char flags;      /* its instruction format */
  unsigned char third_operand;      /* its special register input */
  unsigned char mems;      /* how many μ it costs */
  unsigned char oops;      /* how many υ it costs */
  char *trace_format;      /* how it appears when traced */
} op_info;
```

**65.** For example, the *flags* field of *info*[*op*] tells us how to obtain the operands from the X, Y, and Z fields of the current instruction. Each entry records special properties of an operation code, in binary notation: #1 means Z is an immediate value, #2 means rZ is a source operand, #4 means Y is an immediate value, #8 means rY is a source operand, #10 means rX is a source operand, #20 means rX is a destination, #40 means YZ is part of a relative address, #80 means a push or pop or unsave instruction.

The *trace_format* field will be explained later.

```
#define Z_is_immed_bit #1
#define Z_is_source_bit #2
#define Y_is_immed_bit #4
#define Y_is_source_bit #8
#define X_is_source_bit #10
#define X_is_dest_bit #20
#define rel_addr_bit #40
#define push_pop_bit #80
```

⟨ Global variables 19 ⟩ +≡

```
op_info info[256] = {⟨ Info for arithmetic commands 66 ⟩, ⟨ Info for branch commands 67 ⟩, ⟨ Info for
    load/store commands 68 ⟩, ⟨ Info for logical and control commands 69 ⟩};
```

**66.** ⟨Info for arithmetic commands 66⟩ ≡

```
{"TRAP",#0a,255,0,5,"%r"},
    {"FCMP",#2a,0,0,1,"%l␣=␣%.y␣cmp␣%.z␣=␣%x"},
    {"FUN",#2a,0,0,1,"%l␣=␣[%.y(||)%.z]␣=␣%x"},
    {"FEQL",#2a,0,0,1,"%l␣=␣[%.y(==)%.z]␣=␣%x"},
    {"FADD",#2a,0,0,4,"%l␣=␣%.y␣%(+%)␣%.z␣=␣%.x"},
    {"FIX",#26,0,0,4,"%l␣=␣%(fix%)␣%.z␣=␣%x"},
    {"FSUB",#2a,0,0,4,"%l␣=␣%.y␣%(-%)␣%.z␣=␣%.x"},
    {"FIXU",#26,0,0,4,"%l␣=␣%(fix%)␣%.z␣=␣%#x"},
    {"FLOT",#26,0,0,4,"%l␣=␣%(flot%)␣%z␣=␣%.x"},
    {"FLOTI",#25,0,0,4,"%l␣=␣%(flot%)␣%z␣=␣%.x"},
    {"FLOTU",#26,0,0,4,"%l␣=␣%(flot%)␣%#z␣=␣%.x"},
    {"FLOTUI",#25,0,0,4,"%l␣=␣%(flot%)␣%z␣=␣%.x"},
    {"SFLOT",#26,0,0,4,"%l␣=␣%(sflot%)␣%z␣=␣%.x"},
    {"SFLOTI",#25,0,0,4,"%l␣=␣%(sflot%)␣%z␣=␣%.x"},
    {"SFLOTU",#26,0,0,4,"%l␣=␣%(sflot%)␣%#z␣=␣%.x"},
    {"SFLOTUI",#25,0,0,4,"%l␣=␣%(sflot%)␣%z␣=␣%.x"},
    {"FMUL",#2a,0,0,4,"%l␣=␣%.y␣%(*%)␣%.z␣=␣%.x"},
    {"FCMPE",#2a,rE,0,4,"%l␣=␣%.y␣cmp␣%.z␣(%.b)␣=␣%x"},
    {"FUNE",#2a,rE,0,1,"%l␣=␣[%.y(||)%.z␣(%.b)]␣=␣%x"},
    {"FEQLE",#2a,rE,0,4,"%l␣=␣[%.y(==)%.z␣(%.b)]␣=␣%x"},
    {"FDIV",#2a,0,0,40,"%l␣=␣%.y␣%(/%)␣%.z␣=␣%.x"},
    {"FSQRT",#26,0,0,40,"%l␣=␣%(sqrt%)␣%.z␣=␣%.x"},
    {"FREM",#2a,0,0,4,"%l␣=␣%.y␣%(rem%)␣%.z␣=␣%.x"},
    {"FINT",#26,0,0,4,"%l␣=␣%(int%)␣%.z␣=␣%.x"},
    {"MUL",#2a,0,0,10,"%l␣=␣%y␣*␣%z␣=␣%x"},
    {"MULI",#29,0,0,10,"%l␣=␣%y␣*␣%z␣=␣%x"},
    {"MULU",#2a,0,0,10,"%l␣=␣%#y␣*␣%#z␣=␣%#x,␣rH=%#a"},
    {"MULUI",#29,0,0,10,"%l␣=␣%#y␣*␣%z␣=␣%#x,␣rH=%#a"},
    {"DIV",#2a,0,0,60,"%l␣=␣%y␣/␣%z␣=␣%x,␣rR=%a"},
    {"DIVI",#29,0,0,60,"%l␣=␣%y␣/␣%z␣=␣%x,␣rR=%a"},
    {"DIVU",#2a,rD,0,60,"%l␣=␣%#b%0y␣/␣%#z␣=␣%#x,␣rR=%#a"},
    {"DIVUI",#29,rD,0,60,"%l␣=␣%#b%0y␣/␣%z␣=␣%#x,␣rR=%#a"},
    {"ADD",#2a,0,0,1,"%l␣=␣%y␣+␣%z␣=␣%x"},
    {"ADDI",#29,0,0,1,"%l␣=␣%y␣+␣%z␣=␣%x"},
    {"ADDU",#2a,0,0,1,"%l␣=␣%#y␣+␣%#z␣=␣%#x"},
    {"ADDUI",#29,0,0,1,"%l␣=␣%#y␣+␣%z␣=␣%#x"},
    {"SUB",#2a,0,0,1,"%l␣=␣%y␣-␣%z␣=␣%x"},
    {"SUBI",#29,0,0,1,"%l␣=␣%y␣-␣%z␣=␣%x"},
    {"SUBU",#2a,0,0,1,"%l␣=␣%#y␣-␣%#z␣=␣%#x"},
    {"SUBUI",#29,0,0,1,"%l␣=␣%#y␣-␣%z␣=␣%#x"},
    {"2ADDU",#2a,0,0,1,"%l␣=␣%#y␣<<1+␣%#z␣=␣%#x"},
    {"2ADDUI",#29,0,0,1,"%l␣=␣%#y␣<<1+␣%z␣=␣%#x"},
    {"4ADDU",#2a,0,0,1,"%l␣=␣%#y␣<<2+␣%#z␣=␣%#x"},
    {"4ADDUI",#29,0,0,1,"%l␣=␣%#y␣<<2+␣%z␣=␣%#x"},
    {"8ADDU",#2a,0,0,1,"%l␣=␣%#y␣<<3+␣%#z␣=␣%#x"},
    {"8ADDUI",#29,0,0,1,"%l␣=␣%#y␣<<3+␣%z␣=␣%#x"},
    {"16ADDU",#2a,0,0,1,"%l␣=␣%#y␣<<4+␣%#z␣=␣%#x"},
    {"16ADDUI",#29,0,0,1,"%l␣=␣%#y␣<<4+␣%z␣=␣%#x"},
    {"CMP",#2a,0,0,1,"%l␣=␣%y␣cmp␣%z␣=␣%x"},
    {"CMPI",#29,0,0,1,"%l␣=␣%y␣cmp␣%z␣=␣%x"},
    {"CMPU",#2a,0,0,1,"%l␣=␣%#y␣cmp␣%#z␣=␣%x"},
```

```
{"CMPUI", #29, 0, 0, 1, "%l␣=␣%#y␣cmp␣%z␣=␣%x"},
{"NEG", #26, 0, 0, 1, "%l␣=␣%y␣-␣%z␣=␣%x"},
{"NEGI", #25, 0, 0, 1, "%l␣=␣%y␣-␣%z␣=␣%x"},
{"NEGU", #26, 0, 0, 1, "%l␣=␣%y␣-␣%#z␣=␣%#x"},
{"NEGUI", #25, 0, 0, 1, "%l␣=␣%y␣-␣%z␣=␣%#x"},
{"SL", #2a, 0, 0, 1, "%l␣=␣%y␣<<␣%#z␣=␣%x"},
{"SLI", #29, 0, 0, 1, "%l␣=␣%y␣<<␣%z␣=␣%x"},
{"SLU", #2a, 0, 0, 1, "%l␣=␣%#y␣<<␣%#z␣=␣%#x"},
{"SLUI", #29, 0, 0, 1, "%l␣=␣%#y␣<<␣%z␣=␣%#x"},
{"SR", #2a, 0, 0, 1, "%l␣=␣%y␣>>␣%#z␣=␣%x"},
{"SRI", #29, 0, 0, 1, "%l␣=␣%y␣>>␣%z␣=␣%x"},
{"SRU", #2a, 0, 0, 1, "%l␣=␣%#y␣>>␣%#z␣=␣%#x"},
{"SRUI", #29, 0, 0, 1, "%l␣=␣%#y␣>>␣%z␣=␣%#x"}
```

This code is used in section 65.

**67.** ⟨ Info for branch commands 67 ⟩ ≡

```
{"BN",#50,0,0,1,"%b<0?␣%t%g"},
    {"BNB",#50,0,0,1,"%b<0?␣%t%g"},
    {"BZ",#50,0,0,1,"%b==0?␣%t%g"},
    {"BZB",#50,0,0,1,"%b==0?␣%t%g"},
    {"BP",#50,0,0,1,"%b>0?␣%t%g"},
    {"BPB",#50,0,0,1,"%b>0?␣%t%g"},
    {"BOD",#50,0,0,1,"%b␣odd?␣%t%g"},
    {"BODB",#50,0,0,1,"%b␣odd?␣%t%g"},
    {"BNN",#50,0,0,1,"%b>=0?␣%t%g"},
    {"BNNB",#50,0,0,1,"%b>=0?␣%t%g"},
    {"BNZ",#50,0,0,1,"%b!=0?␣%t%g"},
    {"BNZB",#50,0,0,1,"%b!=0?␣%t%g"},
    {"BNP",#50,0,0,1,"%b<=0?␣%t%g"},
    {"BNPB",#50,0,0,1,"%b<=0?␣%t%g"},
    {"BEV",#50,0,0,1,"%b␣even?␣%t%g"},
    {"BEVB",#50,0,0,1,"%b␣even?␣%t%g"},
    {"PBN",#50,0,0,1,"%b<0?␣%t%g"},
    {"PBNB",#50,0,0,1,"%b<0?␣%t%g"},
    {"PBZ",#50,0,0,1,"%b==0?␣%t%g"},
    {"PBZB",#50,0,0,1,"%b==0?␣%t%g"},
    {"PBP",#50,0,0,1,"%b>0?␣%t%g"},
    {"PBPB",#50,0,0,1,"%b>0?␣%t%g"},
    {"PBOD",#50,0,0,1,"%b␣odd?␣%t%g"},
    {"PBODB",#50,0,0,1,"%b␣odd?␣%t%g"},
    {"PBNN",#50,0,0,1,"%b>=0?␣%t%g"},
    {"PBNNB",#50,0,0,1,"%b>=0?␣%t%g"},
    {"PBNZ",#50,0,0,1,"%b!=0?␣%t%g"},
    {"PBNZB",#50,0,0,1,"%b!=0?␣%t%g"},
    {"PBNP",#50,0,0,1,"%b<=0?␣%t%g"},
    {"PBNPB",#50,0,0,1,"%b<=0?␣%t%g"},
    {"PBEV",#50,0,0,1,"%b␣even?␣%t%g"},
    {"PBEVB",#50,0,0,1,"%b␣even?␣%t%g"},
    {"CSN",#3a,0,0,1,"%l␣=␣%y<0?␣%z:␣%b␣=␣%x"},
    {"CSNI",#39,0,0,1,"%l␣=␣%y<0?␣%z:␣%b␣=␣%x"},
    {"CSZ",#3a,0,0,1,"%l␣=␣%y==0?␣%z:␣%b␣=␣%x"},
    {"CSZI",#39,0,0,1,"%l␣=␣%y==0?␣%z:␣%b␣=␣%x"},
    {"CSP",#3a,0,0,1,"%l␣=␣%y>0?␣%z:␣%b␣=␣%x"},
    {"CSPI",#39,0,0,1,"%l␣=␣%y>0?␣%z:␣%b␣=␣%x"},
    {"CSOD",#3a,0,0,1,"%l␣=␣%y␣odd?␣%z:␣%b␣=␣%x"},
    {"CSODI",#39,0,0,1,"%l␣=␣%y␣odd?␣%z:␣%b␣=␣%x"},
    {"CSNN",#3a,0,0,1,"%l␣=␣%y>=0?␣%z:␣%b␣=␣%x"},
    {"CSNNI",#39,0,0,1,"%l␣=␣%y>=0?␣%z:␣%b␣=␣%x"},
    {"CSNZ",#3a,0,0,1,"%l␣=␣%y!=0?␣%z:␣%b␣=␣%x"},
    {"CSNZI",#39,0,0,1,"%l␣=␣%y!=0?␣%z:␣%b␣=␣%x"},
    {"CSNP",#3a,0,0,1,"%l␣=␣%y<=0?␣%z:␣%b␣=␣%x"},
    {"CSNPI",#39,0,0,1,"%l␣=␣%y<=0?␣%z:␣%b␣=␣%x"},
    {"CSEV",#3a,0,0,1,"%l␣=␣%y␣even?␣%z:␣%b␣=␣%x"},
    {"CSEVI",#39,0,0,1,"%l␣=␣%y␣even?␣%z:␣%b␣=␣%x"},
    {"ZSN",#2a,0,0,1,"%l␣=␣%y<0?␣%z:␣0␣=␣%x"},
    {"ZSNI",#29,0,0,1,"%l␣=␣%y<0?␣%z:␣0␣=␣%x"},
    {"ZSZ",#2a,0,0,1,"%l␣=␣%y==0?␣%z:␣0␣=␣%x"},
```

```
{"ZSZI", #29, 0, 0, 1, "%l␣=␣%y==0?␣%z:␣0␣=␣%x"},
{"ZSP", #2a, 0, 0, 1, "%l␣=␣%y>0?␣%z:␣0␣=␣%x"},
{"ZSPI", #29, 0, 0, 1, "%l␣=␣%y>0?␣%z:␣0␣=␣%x"},
{"ZSOD", #2a, 0, 0, 1, "%l␣=␣%y␣odd?␣%z:␣0␣=␣%x"},
{"ZSODI", #29, 0, 0, 1, "%l␣=␣%y␣odd?␣%z:␣0␣=␣%x"},
{"ZSNN", #2a, 0, 0, 1, "%l␣=␣%y>=0?␣%z:␣0␣=␣%x"},
{"ZSNNI", #29, 0, 0, 1, "%l␣=␣%y>=0?␣%z:␣0␣=␣%x"},
{"ZSNZ", #2a, 0, 0, 1, "%l␣=␣%y!=0?␣%z:␣0␣=␣%x"},
{"ZSNZI", #29, 0, 0, 1, "%l␣=␣%y!=0?␣%z:␣0␣=␣%x"},
{"ZSNP", #2a, 0, 0, 1, "%l␣=␣%y<=0?␣%z:␣0␣=␣%x"},
{"ZSNPI", #29, 0, 0, 1, "%l␣=␣%y<=0?␣%z:␣0␣=␣%x"},
{"ZSEV", #2a, 0, 0, 1, "%l␣=␣%y␣even?␣%z:␣0␣=␣%x"},
{"ZSEVI", #29, 0, 0, 1, "%l␣=␣%y␣even?␣%z:␣0␣=␣%x"}
```

This code is used in section 65.

**68.** ⟨ Info for load/store commands 68 ⟩ ≡

```
{"LDB",#2a,0,1,1,"%l␣=␣M1[%#y+%#z]␣=␣%x"},
    {"LDBI",#29,0,1,1,"%l␣=␣M1[%#y%?+]␣=␣%x"},
    {"LDBU",#2a,0,1,1,"%l␣=␣M1[%#y+%#z]␣=␣%#x"},
    {"LDBUI",#29,0,1,1,"%l␣=␣M1[%#y%?+]␣=␣%#x"},
    {"LDW",#2a,0,1,1,"%l␣=␣M2[%#y+%#z]␣=␣%x"},
    {"LDWI",#29,0,1,1,"%l␣=␣M2[%#y%?+]␣=␣%x"},
    {"LDWU",#2a,0,1,1,"%l␣=␣M2[%#y+%#z]␣=␣%#x"},
    {"LDWUI",#29,0,1,1,"%l␣=␣M2[%#y%?+]␣=␣%#x"},
    {"LDT",#2a,0,1,1,"%l␣=␣M4[%#y+%#z]␣=␣%x"},
    {"LDTI",#29,0,1,1,"%l␣=␣M4[%#y%?+]␣=␣%x"},
    {"LDTU",#2a,0,1,1,"%l␣=␣M4[%#y+%#z]␣=␣%#x"},
    {"LDTUI",#29,0,1,1,"%l␣=␣M4[%#y%?+]␣=␣%#x"},
    {"LDO",#2a,0,1,1,"%l␣=␣M8[%#y+%#z]␣=␣%x"},
    {"LDOI",#29,0,1,1,"%l␣=␣M8[%#y%?+]␣=␣%x"},
    {"LDOU",#2a,0,1,1,"%l␣=␣M8[%#y+%#z]␣=␣%#x"},
    {"LDOUI",#29,0,1,1,"%l␣=␣M8[%#y%?+]␣=␣%#x"},
    {"LDSF",#2a,0,1,1,"%l␣=␣(M4[%#y+%#z])␣=␣%.x"},
    {"LDSFI",#29,0,1,1,"%l␣=␣(M4[%#y%?+])␣=␣%.x"},
    {"LDHT",#2a,0,1,1,"%l␣=␣M4[%#y+%#z]<<32␣=␣%#x"},
    {"LDHTI",#29,0,1,1,"%l␣=␣M4[%#y%?+]<<32␣=␣%#x"},
    {"CSWAP",#3a,0,2,2,"%l␣=␣[M8[%#y+%#z]==%a]␣=␣%x,␣%r"},
    {"CSWAPI",#39,0,2,2,"%l␣=␣[M8[%#y%?+]==%a]␣=␣%x,␣%r"},
    {"LDUNC",#2a,0,1,1,"%l␣=␣M8[%#y+%#z]␣=␣%#x"},
    {"LDUNCI",#29,0,1,1,"%l␣=␣M8[%#y%?+]␣=␣%#x"},
    {"LDVTS",#2a,0,0,1,""},
    {"LDVTSI",#29,0,0,1,""},
    {"PRELD",#0a,0,0,1,"[%#y+%#z␣..␣%#x]"},
    {"PRELDI",#09,0,0,1,"[%#y%?+␣..␣%#x]"},
    {"PREGO",#0a,0,0,1,"[%#y+%#z␣..␣%#x]"},
    {"PREGOI",#09,0,0,1,"[%#y%?+␣..␣%#x]"},
    {"GO",#2a,0,0,3,"%l␣=␣%#x,␣->␣%#y+%#z"},
    {"GOI",#29,0,0,3,"%l␣=␣%#x,␣->␣%#y%?+"},
    {"STB",#1a,0,1,1,"M1[%#y+%#z]␣=␣%b,␣M8[%#w]=%#a"},
    {"STBI",#19,0,1,1,"M1[%#y%?+]␣=␣%b,␣M8[%#w]=%#a"},
    {"STBU",#1a,0,1,1,"M1[%#y+%#z]␣=␣%#b,␣M8[%#w]=%#a"},
    {"STBUI",#19,0,1,1,"M1[%#y%?+]␣=␣%#b,␣M8[%#w]=%#a"},
    {"STW",#1a,0,1,1,"M2[%#y+%#z]␣=␣%b,␣M8[%#w]=%#a"},
    {"STWI",#19,0,1,1,"M2[%#y%?+]␣=␣%b,␣M8[%#w]=%#a"},
    {"STWU",#1a,0,1,1,"M2[%#y+%#z]␣=␣%#b,␣M8[%#w]=%#a"},
    {"STWUI",#19,0,1,1,"M2[%#y%?+]␣=␣%#b,␣M8[%#w]=%#a"},
    {"STT",#1a,0,1,1,"M4[%#y+%#z]␣=␣%b,␣M8[%#w]=%#a"},
    {"STTI",#19,0,1,1,"M4[%#y%?+]␣=␣%b,␣M8[%#w]=%#a"},
    {"STTU",#1a,0,1,1,"M4[%#y+%#z]␣=␣%#b,␣M8[%#w]=%#a"},
    {"STTUI",#19,0,1,1,"M4[%#y%?+]␣=␣%#b,␣M8[%#w]=%#a"},
    {"STO",#1a,0,1,1,"M8[%#y+%#z]␣=␣%b"},
    {"STOI",#19,0,1,1,"M8[%#y%?+]␣=␣%b"},
    {"STOU",#1a,0,1,1,"M8[%#y+%#z]␣=␣%#b"},
    {"STOUI",#19,0,1,1,"M8[%#y%?+]␣=␣%#b"},
    {"STSF",#1a,0,1,1,"%(M4[%#y+%#z]%)␣=␣%.b,␣M8[%#w]=%#a"},
    {"STSFI",#19,0,1,1,"%(M4[%#y%?+]%)␣=␣%.b,␣M8[%#w]=%#a"},
    {"STHT",#1a,0,1,1,"M4[%#y+%#z]␣=␣%#b>>32,␣M8[%#w]=%#a"},
```

```
{"STHTI", #19, 0, 1, 1, "M4[%#y%?+]␣=␣%#b>>32,␣M8[%#w]=%#a"},
{"STCO", #0a, 0, 1, 1, "M8[%#y+%#z]␣=␣%b"},
{"STCOI", #09, 0, 1, 1, "M8[%#y%?+]␣=␣%b"},
{"STUNC", #1a, 0, 1, 1, "M8[%#y+%#z]␣=␣%#b"},
{"STUNCI", #19, 0, 1, 1, "M8[%#y%?+]␣=␣%#b"},
{"SYNCD", #0a, 0, 0, 1, "[%#y+%#z␣..␣%#x]"},
{"SYNCDI", #09, 0, 0, 1, "[%#y%?+␣..␣%#x]"},
{"PREST", #0a, 0, 0, 1, "[%#y+%#z␣..␣%#x]"},
{"PRESTI", #09, 0, 0, 1, "[%#y%?+␣..␣%#x]"},
{"SYNCID", #0a, 0, 0, 1, "[%#y+%#z␣..␣%#x]"},
{"SYNCIDI", #09, 0, 0, 1, "[%#y%?+␣..␣%#x]"},
{"PUSHGO", #aa, 0, 0, 3, "%lrO=%#b,␣rL=%a,␣rJ=%#x,␣->␣%#y+%#z"},
{"PUSHGOI", #a9, 0, 0, 3, "%lrO=%#b,␣rL=%a,␣rJ=%#x,␣->␣%#y%?+"}
```

This code is used in section 65.

**69.** ⟨ Info for logical and control commands 69 ⟩ ≡

```
{"OR",#2a,0,0,1,"%l␣=␣%#y␣|␣%#z␣=␣%#x"},
    {"ORI",#29,0,0,1,"%l␣=␣%#y␣|␣%z␣=␣%#x"},
    {"ORN",#2a,0,0,1,"%l␣=␣%#y␣|~␣%#z␣=␣%#x"},
    {"ORNI",#29,0,0,1,"%l␣=␣%#y␣|~␣%z␣=␣%#x"},
    {"NOR",#2a,0,0,1,"%l␣=␣%#y␣~|␣%#z␣=␣%#x"},
    {"NORI",#29,0,0,1,"%l␣=␣%#y␣~|␣%z␣=␣%#x"},
    {"XOR",#2a,0,0,1,"%l␣=␣%#y␣^␣%#z␣=␣%#x"},
    {"XORI",#29,0,0,1,"%l␣=␣%#y␣^␣%z␣=␣%#x"},
    {"AND",#2a,0,0,1,"%l␣=␣%#y␣&␣%#z␣=␣%#x"},
    {"ANDI",#29,0,0,1,"%l␣=␣%#y␣&␣%z␣=␣%#x"},
    {"ANDN",#2a,0,0,1,"%l␣=␣%#y␣\\␣%#z␣=␣%#x"},
    {"ANDNI",#29,0,0,1,"%l␣=␣%#y␣\\␣%z␣=␣%#x"},
    {"NAND",#2a,0,0,1,"%l␣=␣%#y␣~&␣%#z␣=␣%#x"},
    {"NANDI",#29,0,0,1,"%l␣=␣%#y␣~&␣%z␣=␣%#x"},
    {"NXOR",#2a,0,0,1,"%l␣=␣%#y␣~^␣%#z␣=␣%#x"},
    {"NXORI",#29,0,0,1,"%l␣=␣%#y␣~^␣%z␣=␣%#x"},
    {"BDIF",#2a,0,0,1,"%l␣=␣%#y␣bdif␣%#z␣=␣%#x"},
    {"BDIFI",#29,0,0,1,"%l␣=␣%#y␣bdif␣%z␣=␣%#x"},
    {"WDIF",#2a,0,0,1,"%l␣=␣%#y␣wdif␣%#z␣=␣%#x"},
    {"WDIFI",#29,0,0,1,"%l␣=␣%#y␣wdif␣%z␣=␣%#x"},
    {"TDIF",#2a,0,0,1,"%l␣=␣%#y␣tdif␣%#z␣=␣%#x"},
    {"TDIFI",#29,0,0,1,"%l␣=␣%#y␣tdif␣%z␣=␣%#x"},
    {"ODIF",#2a,0,0,1,"%l␣=␣%#y␣odif␣%#z␣=␣%#x"},
    {"ODIFI",#29,0,0,1,"%l␣=␣%#y␣odif␣%z␣=␣%#x"},
    {"MUX",#2a,rM,0,1,"%l␣=␣%#b?␣%#y:␣%#z␣=␣%#x"},
    {"MUXI",#29,rM,0,1,"%l␣=␣%#b?␣%#y:␣%z␣=␣%#x"},
    {"SADD",#2a,0,0,1,"%l␣=␣nu(%#y\\%#z)␣=␣%x"},
    {"SADDI",#29,0,0,1,"%l␣=␣nu(%#y%?\\)␣=␣%x"},
    {"MOR",#2a,0,0,1,"%l␣=␣%#y␣mor␣%#z␣=␣%#x"},
    {"MORI",#29,0,0,1,"%l␣=␣%#y␣mor␣%z␣=␣%#x"},
    {"MXOR",#2a,0,0,1,"%l␣=␣%#y␣mxor␣%#z␣=␣%#x"},
    {"MXORI",#29,0,0,1,"%l␣=␣%#y␣mxor␣%z␣=␣%#x"},
    {"SETH",#20,0,0,1,"%l␣=␣%#z"},
    {"SETMH",#20,0,0,1,"%l␣=␣%#z"},
    {"SETML",#20,0,0,1,"%l␣=␣%#z"},
    {"SETL",#20,0,0,1,"%l␣=␣%#z"},
    {"INCH",#30,0,0,1,"%l␣=␣%#y␣+␣%#z␣=␣%#x"},
    {"INCMH",#30,0,0,1,"%l␣=␣%#y␣+␣%#z␣=␣%#x"},
    {"INCML",#30,0,0,1,"%l␣=␣%#y␣+␣%#z␣=␣%#x"},
    {"INCL",#30,0,0,1,"%l␣=␣%#y␣+␣%#z␣=␣%#x"},
    {"ORH",#30,0,0,1,"%l␣=␣%#y␣|␣%#z␣=␣%#x"},
    {"ORMH",#30,0,0,1,"%l␣=␣%#y␣|␣%#z␣=␣%#x"},
    {"ORML",#30,0,0,1,"%l␣=␣%#y␣|␣%#z␣=␣%#x"},
    {"ORL",#30,0,0,1,"%l␣=␣%#y␣|␣%#z␣=␣%#x"},
    {"ANDNH",#30,0,0,1,"%l␣=␣%#y␣\\␣%#z␣=␣%#x"},
    {"ANDNMH",#30,0,0,1,"%l␣=␣%#y␣\\␣%#z␣=␣%#x"},
    {"ANDNML",#30,0,0,1,"%l␣=␣%#y␣\\␣%#z␣=␣%#x"},
    {"ANDNL",#30,0,0,1,"%l␣=␣%#y␣\\␣%#z␣=␣%#x"},
    {"JMP",#40,0,0,1,"->␣%#z"},
    {"JMPB",#40,0,0,1,"->␣%#z"},
    {"PUSHJ",#e0,0,0,1,"%lr0=%#b,␣rL=%a,␣rJ=%#x,␣->␣%#z"},
```

```
{"PUSHJB", #e0, 0, 0, 1, "%lrO=%#b,␣rL=%a,␣rJ=%#x,␣->␣%#z"},
{"GETA", #60, 0, 0, 1, "%l␣=␣%#z"},
{"GETAB", #60, 0, 0, 1, "%l␣=␣%#z"},
{"PUT", #02, 0, 0, 1, "%s␣=␣%r"},
{"PUTI", #01, 0, 0, 1, "%s␣=␣%r"},
{"POP", #80, rJ, 0, 3, "%lrL=%a,␣rO=%#b,␣->␣%#y%?+"},
{"RESUME", #00, 0, 0, 5, "{%#b}␣->␣%#z"},
{"SAVE", #20, 0, 20, 1, "%l␣=␣%#x"},
{"UNSAVE", #82, 0, 20, 1, "%#z:␣rG=%x,␣...,␣rL=%a"},
{"SYNC", #01, 0, 0, 1, ""},
{"SWYM", #00, 0, 0, 1, ""},
{"GET", #20, 0, 0, 1, "%l␣=␣%s␣=␣%#x"},
{"TRIP", #0a, 255, 0, 5, "rW=%#w,␣rX=%#x,␣rY=%#y,␣rZ=%#z,␣rB=%#b,␣g[255]=%#a"}
```

This code is used in section 65.

**70.** ⟨ Convert relative address to absolute address 70 ⟩ ≡

```
{
  if ((op & #fe) ≡ JMP) yz = inst & #ffffff;
  if (op & 1) yz −= (op ≡ JMPB ? #1000000 : #10000);
  y = inst_ptr; z = incr(loc, yz ≪ 2);
}
```

This code is used in section 60.

**71.** ⟨ Install operand fields 71 ⟩ ≡

```
if (resuming ∧ rop ≠ RESUME_AGAIN)
  ⟨ Install special operands when resuming an interrupted operation 126 ⟩
else {
  if (f & X_is_source_bit) ⟨ Set b from register X 74 ⟩;
  if (info[op].third_operand) ⟨ Set b from special register 79 ⟩;
  if (f & Z_is_immed_bit) z.l = zz;
  else if (f & Z_is_source_bit) ⟨ Set z from register Z 72 ⟩
  else if ((op & #f0) ≡ SETH) ⟨ Set z as an immediate wyde 78 ⟩;
  if (f & Y_is_immed_bit) y.l = yy;
  else if (f & Y_is_source_bit) ⟨ Set y from register Y 73 ⟩;
}
```

This code is used in section 60.

**72.** There are 256 global registers, $g[0]$ through $g[255]$; the first 32 of them are used for the special registers $rA$, $rB$, etc. There are $lring\_mask + 1$ local registers, usually 256 but the user can increase this to a larger power of 2 if desired.

The current values of rL, rG, rO, and rS are kept in separate variables called $L$, $G$, $O$, and $S$ for convenience. (In fact, $O$ and $S$ actually hold the values rO/8 and rS/8, modulo $lring\_size$.)

⟨ Set $z$ from register Z 72 ⟩ ≡

```
{
  if (zz ≥ G) z = g[zz];
  else if (zz < L) z = l[(O + zz) & lring_mask];
}
```

This code is used in section 71.

**73.** ⟨ Set $y$ from register Y 73 ⟩ ≡

```
{
  if (yy ≥ G) y = g[yy];
  else if (yy < L) y = l[(O + yy) & lring_mask];
}
```

This code is used in section 71.

**74.** ⟨ Set $b$ from register X 74 ⟩ ≡

```
{
  if (xx ≥ G) b = g[xx];
  else if (xx < L) b = l[(O + xx) & lring_mask];
}
```

This code is used in section 71.

**75.** ⟨ Local registers 62 ⟩ +≡

```
register int G, L, O;    /* accessible copies of key registers */
```

**76.** ⟨ Global variables 19 ⟩ +≡

```
octa g[256];    /* global registers */
octa *l;    /* local registers */
int lring_size;    /* the number of local registers (a power of 2) */
int lring_mask;    /* one less than lring_size */
int S;    /* congruent to rS ≫ 3 modulo lring_size */
```

**77.** Several of the global registers have constant values, because of the way `MMIX` has been simplified in this simulator.

Special register rN has a constant value identifying the time of compilation. (The macro `ABSTIME` is defined externally in the file `abstime.h`, which should have just been created by ABSTIME; ABSTIME is a trivial program that computes the value of the standard library function *time*(Λ). We assume that this number, which is the number of seconds in the "UNIX epoch," is less than $2^{32}$. Beware: Our assumption will fail in February of 2106.)

```
#define VERSION 1    /* version of the MMIX architecture that we support */
#define SUBVERSION 0    /* secondary byte of version number */
#define SUBSUBVERSION 3    /* further qualification to version number */
```

⟨ Initialize everything 14 ⟩ +≡

```
g[rK] = neg_one;
g[rN].h = (VERSION ≪ 24) + (SUBVERSION ≪ 16) + (SUBSUBVERSION ≪ 8);
g[rN].l = ABSTIME;    /* see comment and warning above */
g[rT].h = #80000005;
g[rTT].h = #80000006;
g[rV].h = #369c2004;
if (lring_size < 256) lring_size = 256;
lring_mask = lring_size − 1;
if (lring_size & lring_mask) panic("The number of local registers must be a power of 2");
l = (octa *) calloc(lring_size, sizeof(octa));
if (¬l) panic("No room for the local registers");
cur_round = ROUND_NEAR;
```

**78.** In operations like `INCH`, we want $z$ to be the $yz$ field, shifted left 48 bits. We also want $y$ to be register X, which has previously been placed in $b$; then `INCH` can be simulated as if it were `ADDU`.

```
⟨ Set z as an immediate wyde 78 ⟩ ≡
  {
    switch (op & 3) {
    case 0: z.h = yz ≪ 16; break;
    case 1: z.h = yz; break;
    case 2: z.l = yz ≪ 16; break;
    case 3: z.l = yz; break;
    }
    y = b;
  }
```

This code is used in section 71.

**79.**

```
⟨ Set b from special register 79 ⟩ ≡
  b = g[info[op].third_operand];
```

This code is used in section 71.

**80.**

```
⟨ Install register X as the destination, adjusting the register stack if necessary 80 ⟩ ≡
  {
    if (xx ≥ G) {
      sprintf(lhs, "$%d=g[%d]", xx, xx);
      x_ptr = &g[xx];
    } else {
      while (xx ≥ L) ⟨ Increase rL 81 ⟩;
      sprintf(lhs, "$%d=l[%d]", xx, (O + xx) & lring_mask);
      x_ptr = &l[(O + xx) & lring_mask];
    }
  }
```

This code is used in section 60.

**81.**

```
⟨ Increase rL 81 ⟩ ≡
  {
    l[(O + L) & lring_mask] = zero_octa;
    L = g[rL].l = L + 1;
    if (((S − O − L) & lring_mask) ≡ 0) stack_store();
  }
```

This code is used in section 80.

**82.** The *stack_store* routine advances the "gamma" pointer in the ring of local registers, by storing the oldest local register into memory location rS and advancing rS.

```
#define test_store_bkpt(ll) if ((ll)→bkpt & write_bit) breakpoint = tracing = true
⟨ Subroutines 12 ⟩ +≡
  void stack_store ARGS((void));
  void stack_store()
  {
    register mem_tetra *ll = mem_find(g[rS]);
    register int k = S & lring_mask;
    ll→tet = l[k].h; test_store_bkpt(ll);
    (ll + 1)→tet = l[k].l; test_store_bkpt(ll + 1);
    if (stack_tracing) {
      tracing = true;
      if (cur_line) show_line();
      printf("␣␣␣␣␣␣␣␣␣␣␣␣␣␣M8[#%08x%08x]=l[%d]=#%08x%08x,␣rS+=8\n", g[rS].h, g[rS].l, k, l[k].h, l[k].l);
    }
    g[rS] = incr(g[rS], 8), S++;
  }
```

**83.** The *stack_load* routine is essentially the inverse of *stack_store*.

```
#define test_load_bkpt(ll) if ((ll)→bkpt & read_bit) breakpoint = tracing = true
⟨ Subroutines 12 ⟩ +≡
  void stack_load ARGS((void));
  void stack_load()
  {
    register mem_tetra *ll;
    register int k;
    S--, g[rS] = incr(g[rS], -8);
    ll = mem_find(g[rS]);
    k = S & lring_mask;
    l[k].h = ll→tet; test_load_bkpt(ll);
    l[k].l = (ll + 1)→tet; test_load_bkpt(ll + 1);
    if (stack_tracing) {
      tracing = true;
      if (cur_line) show_line();
      printf("␣␣␣␣␣␣␣␣␣␣␣␣␣␣rS-=8,␣l[%d]=M8[#%08x%08x]=#%08x%08x\n", k, g[rS].h, g[rS].l, l[k].h, l[k].l);
    }
  }
```

**84. Simulating the instructions.** The master switch branches in 256 directions, one for each MMIX instruction.

Let's start with ADD, since it is somehow the most typical case—not too easy, and not too hard. The task is to compute $x = y + z$, and to signal overflow if the sum is out of range. Overflow occurs if and only if $y$ and $z$ have the same sign but the sum has a different sign.

Overflow is one of the eight arithmetic exceptions. We record such exceptions in a variable called *exc*, which is set to zero at the beginning of each cycle and used to update rA at the end.

The main control routine has put the input operands into octabytes $y$ and $z$. It has also made *x_ptr* point to the octabyte where the result should be placed.

```
⟨ Cases for individual MMIX instructions 84 ⟩ ≡
case ADD: case ADDI: x = w;      /* w = oplus(y,z) */
  if (((y.h ⊕ z.h) & sign_bit) ≡ 0 ∧ ((y.h ⊕ x.h) & sign_bit) ≠ 0) exc |= V_BIT;
store_x: *x_ptr = x; break;
```

See also sections 85, 86, 87, 88, 89, 90, 92, 93, 94, 95, 96, 97, 101, 102, 104, 106, 107, 108, and 124.

This code is used in section 60.

**85.** Other cases of signed and unsigned addition and subtraction are, of course, similar. Overflow occurs in the calculation $x = y - z$ if and only if it occurs in the calculation $y = x + z$.

```
⟨ Cases for individual MMIX instructions 84 ⟩ +≡
case SUB: case SUBI: case NEG: case NEGI: x = ominus(y,z);
  if (((x.h ⊕ z.h) & sign_bit) ≡ 0 ∧ ((x.h ⊕ y.h) & sign_bit) ≠ 0) exc |= V_BIT;
  goto store_x;
case ADDU: case ADDUI: case INCH: case INCMH: case INCML: case INCL: x = w; goto store_x;
case SUBU: case SUBUI: case NEGU: case NEGUI: x = ominus(y,z); goto store_x;
case IIADDU: case IIADDUI: case IVADDU: case IVADDUI: case VIIIADDU: case VIIIADDUI:
  case XVIADDU: case XVIADDUI: x = oplus(shift_left(y,((op & #f) ≫ 1) − 3),z); goto store_x;
case SETH: case SETMH: case SETML: case SETL: case GETA: case GETAB: x = z; goto store_x;
```

**86.** Let's get the simple bitwise operations out of the way too.

```
⟨ Cases for individual MMIX instructions 84 ⟩ +≡
case OR: case ORI: case ORH: case ORMH: case ORML: case ORL: x.h = y.h | z.h; x.l = y.l | z.l;
  goto store_x;
case ORN: case ORNI: x.h = y.h | ∼z.h; x.l = y.l | ∼z.l; goto store_x;
case NOR: case NORI: x.h = ∼(y.h | z.h); x.l = ∼(y.l | z.l); goto store_x;
case XOR: case XORI: x.h = y.h ⊕ z.h; x.l = y.l ⊕ z.l; goto store_x;
case AND: case ANDI: x.h = y.h & z.h; x.l = y.l & z.l; goto store_x;
case ANDN: case ANDNI: case ANDNH: case ANDNMH: case ANDNML: case ANDNL: x.h = y.h & ∼z.h;
  x.l = y.l & ∼z.l; goto store_x;
case NAND: case NANDI: x.h = ∼(y.h & z.h); x.l = ∼(y.l & z.l); goto store_x;
case NXOR: case NXORI: x.h = ∼(y.h ⊕ z.h); x.l = ∼(y.l ⊕ z.l); goto store_x;
```

**87.** The less simple bit manipulations are almost equally simple, given the subroutines of MMIX-ARITH. The `MUX` operation has three inputs; in such cases the inputs appear in $y$, $z$, and $b$.

**#define** *shift_amt* $(z.h \vee z.l \geq 64 \,?\, 64 : z.l)$

⟨ Cases for individual `MMIX` instructions 84 ⟩ +≡
**case** `SL`: **case** `SLI`: $x = shift\_left(y, shift\_amt)$;
  $a = shift\_right(x, shift\_amt, 0)$;
  **if** $(a.h \neq y.h \vee a.l \neq y.l)$ $exc \mid= \texttt{V\_BIT}$;
  **goto** *store_x*;
**case** `SLU`: **case** `SLUI`: $x = shift\_left(y, shift\_amt)$; **goto** *store_x*;
**case** `SR`: **case** `SRI`: **case** `SRU`: **case** `SRUI`: $x = shift\_right(y, shift\_amt, op \,\&\, \#2)$; **goto** *store_x*;
**case** `MUX`: **case** `MUXI`: $x.h = (y.h \,\&\, b.h) \mid (z.h \,\&\, {\sim}b.h)$; $x.l = (y.l \,\&\, b.l) \mid (z.l \,\&\, {\sim}b.l)$;
  **goto** *store_x*;
**case** `SADD`: **case** `SADDI`: $x.l = count\_bits(y.h \,\&\, {\sim}z.h) + count\_bits(y.l \,\&\, {\sim}z.l)$; **goto** *store_x*;
**case** `MOR`: **case** `MORI`: $x = bool\_mult(y, z, false)$; **goto** *store_x*;
**case** `MXOR`: **case** `MXORI`: $x = bool\_mult(y, z, true)$; **goto** *store_x*;
**case** `BDIF`: **case** `BDIFI`: $x.h = byte\_diff(y.h, z.h)$; $x.l = byte\_diff(y.l, z.l)$; **goto** *store_x*;
**case** `WDIF`: **case** `WDIFI`: $x.h = wyde\_diff(y.h, z.h)$; $x.l = wyde\_diff(y.l, z.l)$; **goto** *store_x*;
**case** `TDIF`: **case** `TDIFI`: **if** $(y.h > z.h)$ $x.h = y.h - z.h$;
*tdif_l*: **if** $(y.l > z.l)$ $x.l = y.l - z.l$; **goto** *store_x*;
**case** `ODIF`: **case** `ODIFI`: **if** $(y.h > z.h)$ $x = ominus(y, z)$;
  **else if** $(y.h \equiv z.h)$ **goto** *tdif_l*;
  **goto** *store_x*;

**88.** When an operation has two outputs, the primary output is placed in $x$ and the auxiliary output is placed in $a$.

⟨ Cases for individual `MMIX` instructions 84 ⟩ +≡
**case** `MUL`: **case** `MULI`: $x = signed\_omult(y, z)$;
*test_overflow*: **if** $(overflow)$ $exc \mid= \texttt{V\_BIT}$;
  **goto** *store_x*;
**case** `MULU`: **case** `MULUI`: $x = omult(y, z)$; $a = g[rH] = aux$; **goto** *store_x*;
**case** `DIV`: **case** `DIVI`: **if** $(\neg z.l \wedge \neg z.h)$ $aux = y$, $exc \mid= \texttt{D\_BIT}$, $overflow = false$;
  **else** $x = signed\_odiv(y, z)$;
  $a = g[rR] = aux$; **goto** *test_overflow*;
**case** `DIVU`: **case** `DIVUI`: $x = odiv(b, y, z)$; $a = g[rR] = aux$; **goto** *store_x*;

**89.** The floating point routines of MMIX-ARITH record exceptional events in a variable called *exceptions*. Here we simply merge those bits into the *exc* variable. The `U_BIT` is not exactly the same as "underflow," but the true definition of underflow will be applied when *exc* is combined with rA.

```
⟨ Cases for individual MMIX instructions 84 ⟩ +≡
case FADD: x = fplus(y, z);
fin_float: round_mode = cur_round;
store_fx: exc |= exceptions; goto store_x;
case FSUB: a = z; if (fcomp(a, zero_octa) ≠ 2) a.h ⊕= sign_bit;
  x = fplus(y, a); goto fin_float;
case FMUL: x = fmult(y, z); goto fin_float;
case FDIV: x = fdivide(y, z); goto fin_float;
case FREM: x = fremstep(y, z, 2500); goto fin_float;
case FSQRT: x = froot(z, y.l);
fin_unifloat: if (y.h ∨ y.l > 4) goto illegal_inst;
  round_mode = (y.l ? (int) y.l : cur_round); goto store_fx;
case FINT: x = fintegerize(z, y.l); goto fin_unifloat;
case FIX: x = fixit(z, y.l); goto fin_unifloat;
case FIXU: x = fixit(z, y.l); exceptions &= ∼W_BIT; goto fin_unifloat;
case FLOT: case FLOTI: case FLOTU: case FLOTUI: case SFLOT: case SFLOTI: case SFLOTU:
  case SFLOTUI: x = floatit(z, y.l, op & #2, op & #4); goto fin_unifloat;
```

**90.** We have now done all of the arithmetic operations except for the cases that compare two registers and yield a value of −1 or 0 or 1.

```
#define cmp_zero store_x /* x is 0 by default */
⟨ Cases for individual MMIX instructions 84 ⟩ +≡
case CMP: case CMPI: if ((y.h & sign_bit) > (z.h & sign_bit)) goto cmp_neg;
  if ((y.h & sign_bit) < (z.h & sign_bit)) goto cmp_pos;
case CMPU: case CMPUI: if (y.h < z.h) goto cmp_neg;
  if (y.h > z.h) goto cmp_pos;
  if (y.l < z.l) goto cmp_neg;
  if (y.l ≡ z.l) goto cmp_zero;
cmp_pos: x.l = 1; goto store_x;
cmp_neg: x = neg_one; goto store_x;
case FCMPE: k = fepscomp(y, z, b, true);
  if (k) goto cmp_zero_or_invalid;
case FCMP: k = fcomp(y, z);
  if (k < 0) goto cmp_neg;
cmp_fin: if (k ≡ 1) goto cmp_pos;
cmp_zero_or_invalid: if (k ≡ 2) exc |= I_BIT;
  goto cmp_zero;
case FUN: if (fcomp(y, z) ≡ 2) goto cmp_pos; else goto cmp_zero;
case FEQL: if (fcomp(y, z) ≡ 0) goto cmp_pos; else goto cmp_zero;
case FEQLE: k = fepscomp(y, z, b, false);
  goto cmp_fin;
case FUNE: if (fepscomp(y, z, b, true) ≡ 2) goto cmp_pos; else goto cmp_zero;
```

**91.** We have now done all the register-register operations except for the conditional commands. Conditional commands and branch commands all make use of a simple subroutine that determines whether a given octabyte satisfies the condition of a given opcode.

```
⟨ Subroutines 12 ⟩ +≡
  int register_truth ARGS((octa, mmix_opcode));
  int register_truth(o, op)
      octa o;
      mmix_opcode op;
  { register int b;
    switch ((op ≫ 1) & #3) {
    case 0: b = o.h ≫ 31; break;      /* negative? */
    default: case 1: b = (o.h ≡ 0 ∧ o.l ≡ 0); break;      /* zero? */
    case 2: b = (o.h < sign_bit ∧ (o.h ∨ o.l)); break;      /* positive? */
    case 3: b = o.l & #1; break;      /* odd? */
    }
    if (op & #8) return b ⊕ 1;
    else return b;
  }
```

**92.** The $b$ operand will be zero on the `ZS` operations; it will be the contents of register X on the `CS` operations.

```
⟨ Cases for individual MMIX instructions 84 ⟩ +≡
case CSN: case CSNI: case CSZ: case CSZI:
case CSP: case CSPI: case CSOD: case CSODI:
case CSNN: case CSNNI: case CSNZ: case CSNZI:
case CSNP: case CSNPI: case CSEV: case CSEVI:
case ZSN: case ZSNI: case ZSZ: case ZSZI:
case ZSP: case ZSPI: case ZSOD: case ZSODI:
case ZSNN: case ZSNNI: case ZSNZ: case ZSNZI:
case ZSNP: case ZSNPI: case ZSEV: case ZSEVI:
  x = register_truth(y, op) ? z : b; goto store_x;
```

**93.** Didn't that feel good, when 32 opcodes reduced to a single case? We get to do it one more time. Happiness!

```
⟨ Cases for individual MMIX instructions 84 ⟩ +≡
case BN: case BNB: case BZ: case BZB:
case BP: case BPB: case BOD: case BODB:
case BNN: case BNNB: case BNZ: case BNZB:
case BNP: case BNPB: case BEV: case BEVB:
case PBN: case PBNB: case PBZ: case PBZB:
case PBP: case PBPB: case PBOD: case PBODB:
case PBNN: case PBNNB: case PBNZ: case PBNZB:
case PBNP: case PBNPB: case PBEV: case PBEVB:
  x.l = register_truth(b, op);
  if (x.l) {
    inst_ptr = z;
    good = (op ≥ PBN);
  } else good = (op < PBN);
  if (good) good_guesses++;
  else {
    bad_guesses++, sclock.l += 2;      /* penalty is 2υ for bad guess */
    if (g[rI].l ≤ 2 ∧ g[rI].l ∧ g[rI].h ≡ 0) tracing = breakpoint = true;
    g[rI] = incr(g[rI], −2);
  }
  break;
```

**94.** Memory operations are next on our agenda. The memory address, $y + z$, has already been placed in $w$.

```
⟨ Cases for individual MMIX instructions 84 ⟩ +≡
case LDB: case LDBI: case LDBU: case LDBUI:
  i = 56; j = (w.l & #3) ≪ 3;
  goto fin_ld;
case LDW: case LDWI: case LDWU: case LDWUI:
  i = 48; j = (w.l & #2) ≪ 3;
  goto fin_ld;
case LDT: case LDTI: case LDTU: case LDTUI:
  i = 32; j = 0; goto fin_ld;
case LDHT: case LDHTI: i = j = 0;
fin_ld: ll = mem_find(w); test_load_bkpt(ll);
  x.h = ll→tet;
  x = shift_right(shift_left(x, j), i, op & #2);
check_ld: if (w.h & sign_bit) goto privileged_inst;
  goto store_x;
case LDO: case LDOI: case LDOU: case LDOUI: case LDUNC: case LDUNCI: w.l &= −8;
  ll = mem_find(w);
  test_load_bkpt(ll); test_load_bkpt(ll + 1);
  x.h = ll→tet; x.l = (ll + 1)→tet;
  goto check_ld;
case LDSF: case LDSFI: ll = mem_find(w); test_load_bkpt(ll);
  x = load_sf(ll→tet); goto check_ld;
```

**95.** ⟨ Cases for individual `MMIX` instructions 84 ⟩ +≡

```
case STB: case STBI: case STBU: case STBUI:
  i = 56; j = (w.l & #3) ≪ 3;
  goto fin_pst;
case STW: case STWI: case STWU: case STWUI:
  i = 48; j = (w.l & #2) ≪ 3;
  goto fin_pst;
case STT: case STTI: case STTU: case STTUI:
  i = 32; j = 0;
fin_pst: ll = mem_find(w);
  if ((op & #2) ≡ 0) {
    a = shift_right(shift_left(b, i), i, 0);
    if (a.h ≠ b.h ∨ a.l ≠ b.l) exc |= V_BIT;
  }
  ll→tet ⊕= (ll→tet ⊕ (b.l ≪ (i − 32 − j))) & ((((tetra) −1) ≪ (i − 32)) ≫ j);
  goto fin_st;
case STSF: case STSFI: ll = mem_find(w);
  ll→tet = store_sf(b); exc = exceptions;
  goto fin_st;
case STHT: case STHTI: ll = mem_find(w); ll→tet = b.h;
fin_st: test_store_bkpt(ll);
  w.l &= −8; ll = mem_find(w);
  a.h = ll→tet; a.l = (ll + 1)→tet;      /* for trace output */
  goto check_st;
case STCO: case STCOI: b.l = xx;
case STO: case STOI: case STOU: case STOUI: case STUNC: case STUNCI: w.l &= −8;
  ll = mem_find(w);
  test_store_bkpt(ll); test_store_bkpt(ll + 1);
  ll→tet = b.h; (ll + 1)→tet = b.l;
check_st: if (w.h & sign_bit) goto privileged_inst;
  break;
```

**96.** The `CSWAP` operation has elements of both loading and storing. We shuffle some of the operands around so that they will appear correctly in the trace output.

⟨ Cases for individual `MMIX` instructions 84 ⟩ +≡

```
case CSWAP: case CSWAPI: w.l &= −8; ll = mem_find(w);
  test_load_bkpt(ll); test_load_bkpt(ll + 1);
  a = g[rP];
  if (ll→tet ≡ a.h ∧ (ll + 1)→tet ≡ a.l) {
    x.h = 0, x.l = 1;
    test_store_bkpt(ll); test_store_bkpt(ll + 1);
    ll→tet = b.h, (ll + 1)→tet = b.l;
    strcpy(rhs, "M8[%#w]=%#b");
  } else {
    b.h = ll→tet, b.l = (ll + 1)→tet;
    g[rP] = b;
    strcpy(rhs, "rP=%#b");
  }
  goto check_ld;
```

**97.** The `GET` command is permissive, but `PUT` is restrictive.

```
⟨ Cases for individual MMIX instructions 84 ⟩ +≡
case GET: if (yy ≠ 0 ∨ zz ≥ 32) goto illegal_inst;
  x = g[zz];
  goto store_x;
case PUT: case PUTI: if (yy ≠ 0 ∨ xx ≥ 32) goto illegal_inst;
  strcpy(rhs, "%z␣=␣%#z");
  if (xx ≥ 8) {
    if (xx ≤ 11 ∧ xx ≠ 8) goto illegal_inst;      /* can't change rN, rO, rS */
    if (xx ≤ 18) goto privileged_inst;
    if (xx ≡ rA) ⟨ Get ready to update rA 100 ⟩
    else if (xx ≡ rL) ⟨ Set L = z = min(z, L) 98 ⟩
    else if (xx ≡ rG) ⟨ Get ready to update rG 99 ⟩;
  }
  g[xx] = z; zz = xx; break;
```

**98.**

```
⟨ Set L = z = min(z, L) 98 ⟩ ≡
  {
    x = z; strcpy(rhs, z.h ? "min(rL,%#x)␣=␣%z" : "min(rL,%x)␣=␣%z");
    if (z.l > (unsigned int) L ∨ z.h) z.h = 0, z.l = L;
    else old_L = L = z.l;
  }
```

This code is used in section 97.

**99.**

```
⟨ Get ready to update rG 99 ⟩ ≡
  {
    if (z.h ≠ 0 ∨ z.l > 255 ∨ z.l < (unsigned int) L ∨ z.l < 32) goto illegal_inst;
    for (j = z.l; j < G; j++) g[j] = zero_octa;
    G = z.l;
  }
```

This code is used in section 97.

**100.**

```
#define ROUND_OFF 1
#define ROUND_UP 2
#define ROUND_DOWN 3
#define ROUND_NEAR 4
⟨ Get ready to update rA 100 ⟩ ≡
  {
    if (z.h ≠ 0 ∨ z.l ≥ #40000) goto illegal_inst;
    cur_round = (z.l ≥ #10000 ? z.l ≫ 16 : ROUND_NEAR);
  }
```

This code is used in section 97.

**101.** Pushing and popping are rather delicate, because we want to trace them coherently.

```
⟨ Cases for individual MMIX instructions 84 ⟩ +≡
case PUSHGO: case PUSHGOI: inst_ptr = w; goto push;
case PUSHJ: case PUSHJB: inst_ptr = z;
push: if (xx ≥ G) {
    xx = L++;
    if (((S − O − L) & lring_mask) ≡ 0) stack_store();
  }
  x.l = xx; l[(O + xx) & lring_mask] = x;      /* the "hole" records the amount pushed */
  sprintf(lhs, "l[%d]=%d,␣", (O + xx) & lring_mask, xx);
  x = g[rJ] = incr(loc, 4);
  L −= xx + 1; O += xx + 1;
  b = g[rO] = incr(g[rO], (xx + 1) ≪ 3);
sync_L: a.l = g[rL].l = L; break;
case POP: if (xx ≠ 0 ∧ xx ≤ L) y = l[(O + xx − 1) & lring_mask];
  if (g[rS].l ≡ g[rO].l) stack_load();
  k = l[(O − 1) & lring_mask].l & #ff;
  while ((tetra)(O − S) ≤ (tetra) k) stack_load();
  L = k + (xx ≤ L ? xx : L + 1);
  if (L > G) L = G;
  if (L > k) {
    l[(O − 1) & lring_mask] = y;
    if (y.h) sprintf(lhs, "l[%d]=#%x%08x,␣", (O − 1) & lring_mask, y.h, y.l);
    else sprintf(lhs, "l[%d]=#%x,␣", (O − 1) & lring_mask, y.l);
  } else lhs[0] = '\0';
  y = g[rJ]; z.l = yz ≪ 2; inst_ptr = oplus(y, z);
  O −= k + 1; b = g[rO] = incr(g[rO], −((k + 1) ≪ 3));
  goto sync_L;
```

**102.** To complete our simulation of MMIX's register stack, we need to implement SAVE and UNSAVE.

```
⟨ Cases for individual MMIX instructions 84 ⟩ +≡
case SAVE: if (xx < G ∨ yy ≠ 0 ∨ zz ≠ 0) goto illegal_inst;
  l[(O + L) & lring_mask].l = L, L++;
  if (((S − O − L) & lring_mask) ≡ 0) stack_store();
  O += L; g[rO] = incr(g[rO], L ≪ 3);
  L = g[rL].l = 0;
  while (g[rO].l ≠ g[rS].l) stack_store();
  for (k = G; ; ) {
    ⟨ Store g[k] in the register stack 103 ⟩;
    if (k ≡ 255) k = rB;
    else if (k ≡ rR) k = rP;
    else if (k ≡ rZ + 1) break;
    else k++;
  }
  O = S, g[rO] = g[rS];
  x = incr(g[rO], −8); goto store_x;
```

**103.** This part of the program naturally has a lot in common with the *stack_store* subroutine. (There's a little white lie in the section name; if $k$ is $rZ+1$, we store rG and rA, not $g[k]$.)

```
⟨ Store g[k] in the register stack 103 ⟩ ≡
  ll = mem_find(g[rS]);
  if (k ≡ rZ + 1) x.h = G ≪ 24, x.l = g[rA].l;
  else x = g[k];
  ll→tet = x.h; test_store_bkpt(ll);
  (ll + 1)→tet = x.l; test_store_bkpt(ll + 1);
  if (stack_tracing) {
    tracing = true;
    if (cur_line) show_line();
    if (k ≥ 32)
      printf("␣␣␣␣␣␣␣␣␣␣␣␣␣␣M8[#%08x%08x]=g[%d]=#%08x%08x,␣rS+=8\n", g[rS].h, g[rS].l, k, x.h, x.l);
    else printf("␣␣␣␣␣␣␣␣␣␣␣␣␣␣M8[#%08x%08x]=%s=#%08x%08x,␣rS+=8\n", g[rS].h, g[rS].l,
        k ≡ rZ + 1 ? "(rG,rA)" : special_name[k], x.h, x.l);
  }
  S++, g[rS] = incr(g[rS], 8);
```

This code is used in section 102.

**104.**

```
⟨ Cases for individual MMIX instructions 84 ⟩ +≡
case UNSAVE: if (xx ≠ 0 ∨ yy ≠ 0) goto illegal_inst;
  z.l &= −8; g[rS] = incr(z, 8);
  for (k = rZ + 1; ; ) {
    ⟨ Load g[k] from the register stack 105 ⟩;
    if (k ≡ rP) k = rR;
    else if (k ≡ rB) k = 255;
    else if (k ≡ G) break;
    else k−−;
  }
  S = g[rS].l ≫ 3;
  stack_load();
  k = l[S & lring_mask].l & #ff;
  for (j = 0; j < k; j++) stack_load();
  O = S; g[rO] = g[rS];
  L = k > G ? G : k;
  g[rL].l = L; a = g[rL];
  g[rG].l = G; break;
```

**105.** ⟨ Load $g[k]$ from the register stack 105 ⟩ ≡

```
g[rS] = incr(g[rS], -8);
ll = mem_find(g[rS]);
test_load_bkpt(ll); test_load_bkpt(ll + 1);
if (k ≡ rZ + 1) {
  x.l = G = g[rG].l = ll→tet ≫ 24, a.l = g[rA].l = (ll + 1)→tet & #3ffff;
  if (G < 32) x.l = G = g[rG].l = 32;
} else g[k].h = ll→tet, g[k].l = (ll + 1)→tet;
if (stack_tracing) {
  tracing = true;
  if (cur_line) show_line();
  if (k ≥ 32) printf("␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣rS-=8,␣g[%d]=M8[#%08x%08x]=#%08x%08x\n", k, g[rS].h, g[rS].l,
      ll→tet, (ll + 1)→tet);
  else if (k ≡ rZ + 1)
    printf("␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣(rG,rA)=M8[#%08x%08x]=#%08x%08x\n", g[rS].h, g[rS].l, ll→tet, (ll + 1)→tet);
  else printf("␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣rS-=8,␣%s=M8[#%08x%08x]=#%08x%08x\n", special_name[k], g[rS].h,
      g[rS].l, ll→tet, (ll + 1)→tet);
}
```

This code is used in section 104.

**106.** The cache maintenance instructions don't affect this simulation, because there are no caches. But if the user has invoked them, we do provide a bit of information when tracing, indicating the scope of the instruction.

⟨ Cases for individual MMIX instructions 84 ⟩ +≡

```
case SYNCID: case SYNCIDI: case PREST: case PRESTI: case SYNCD: case SYNCDI: case PREGO:
  case PREGOI: case PRELD: case PRELDI: x = incr(w, xx); break;
```

**107.** Several loose ends remain to be nailed down.

⟨ Cases for individual MMIX instructions 84 ⟩ +≡

```
case GO: case GOI: x = inst_ptr; inst_ptr = w; goto store_x;
case JMP: case JMPB: inst_ptr = z;
case SWYM: break;
case SYNC: if (xx ≠ 0 ∨ yy ≠ 0 ∨ zz > 7) goto illegal_inst;
  if (zz ≤ 3) break;
case LDVTS: case LDVTSI: privileged_inst: strcpy(lhs, "!privileged");
  goto break_inst;
illegal_inst: strcpy(lhs, "!illegal");
break_inst: breakpoint = tracing = true;
  if (¬interacting ∧ ¬interact_after_break) halted = true;
  break;
```

**108. Trips and traps.** We have now implemented 253 of the 256 instructions: all but `TRIP`, `TRAP`, and `RESUME`.

The `TRIP` instruction simply turns `H_BIT` on in the *exc* variable; this will trigger an interruption to location 0.

The `TRAP` instruction is not simulated, except for the system calls mentioned in the introduction.

```
⟨ Cases for individual MMIX instructions 84 ⟩ +≡
case TRIP: exc |= H_BIT; break;
case TRAP: if (xx ≠ 0 ∨ yy > max_sys_call) goto privileged_inst;
  strcpy(rhs, trap_format[yy]);
  g[rWW] = inst_ptr;
  g[rXX].h = sign_bit, g[rXX].l = inst;
  g[rYY] = y, g[rZZ] = z;
  z.h = 0, z.l = zz;
  a = incr(b, 8);
  ⟨ Prepare memory arguments ma = M[a] and mb = M[b] if needed 111 ⟩;
  switch (yy) {
  case Halt: ⟨ Either halt or print warning 109 ⟩; g[rBB] = g[255]; break;
  case Fopen: g[rBB] = mmix_fopen((unsigned char) zz, mb, ma); break;
  case Fclose: g[rBB] = mmix_fclose((unsigned char) zz); break;
  case Fread: g[rBB] = mmix_fread((unsigned char) zz, mb, ma); break;
  case Fgets: g[rBB] = mmix_fgets((unsigned char) zz, mb, ma); break;
  case Fgetws: g[rBB] = mmix_fgetws((unsigned char) zz, mb, ma); break;
  case Fwrite: g[rBB] = mmix_fwrite((unsigned char) zz, mb, ma); break;
  case Fputs: g[rBB] = mmix_fputs((unsigned char) zz, b); break;
  case Fputws: g[rBB] = mmix_fputws((unsigned char) zz, b); break;
  case Fseek: g[rBB] = mmix_fseek((unsigned char) zz, b); break;
  case Ftell: g[rBB] = mmix_ftell((unsigned char) zz); break;
  }
  x = g[255] = g[rBB]; break;
```

**109.**
```
⟨ Either halt or print warning 109 ⟩ ≡
  if (¬zz) halted = breakpoint = true;
  else if (zz ≡ 1) {
    if (loc.h ∨ loc.l ≥ #90) goto privileged_inst;
    print_trip_warning(loc.l ≫ 4, incr(g[rW], −4));
  } else goto privileged_inst;
```
This code is used in section 108.

**110.**
```
⟨ Global variables 19 ⟩ +≡
  char arg_count[] = {1, 3, 1, 3, 3, 3, 3, 2, 2, 2, 1};
  char *trap_format[] = {"Halt(%z)", "$255␣=␣Fopen(%!z,M8[%#b]=%#q,M8[%#a]=%p)␣=␣%x",
      "$255␣=␣Fclose(%!z)␣=␣%x", "$255␣=␣Fread(%!z,M8[%#b]=%#q,M8[%#a]=%p)␣=␣%x",
      "$255␣=␣Fgets(%!z,M8[%#b]=%#q,M8[%#a]=%p)␣=␣%x",
      "$255␣=␣Fgetws(%!z,M8[%#b]=%#q,M8[%#a]=%p)␣=␣%x",
      "$255␣=␣Fwrite(%!z,M8[%#b]=%#q,M8[%#a]=%p)␣=␣%x", "$255␣=␣Fputs(%!z,%#b)␣=␣%x",
      "$255␣=␣Fputws(%!z,%#b)␣=␣%x", "$255␣=␣Fseek(%!z,%b)␣=␣%x", "$255␣=␣Ftell(%!z)␣=␣%x"};
```

**111.** ⟨ Prepare memory arguments $ma = \mathrm{M}[a]$ and $mb = \mathrm{M}[b]$ if needed 111 ⟩ ≡

```
if (arg_count[yy] ≡ 3) {
  ll = mem_find(b); test_load_bkpt(ll); test_load_bkpt(ll + 1);
  mb.h = ll→tet, mb.l = (ll + 1)→tet;
  ll = mem_find(a); test_load_bkpt(ll); test_load_bkpt(ll + 1);
  ma.h = ll→tet, ma.l = (ll + 1)→tet;
}
```

This code is used in section 108.

**112.** The input/output operations invoked by `TRAP`s are done by subroutines in an auxiliary program module called MMIX-IO. Here we need only declare those subroutines, and write three primitive interfaces on which they depend.

**113.** ⟨ Global variables 19 ⟩ +≡

```
extern void mmix_io_init ARGS((void));
extern octa mmix_fopen ARGS((unsigned char, octa, octa));
extern octa mmix_fclose ARGS((unsigned char));
extern octa mmix_fread ARGS((unsigned char, octa, octa));
extern octa mmix_fgets ARGS((unsigned char, octa, octa));
extern octa mmix_fgetws ARGS((unsigned char, octa, octa));
extern octa mmix_fwrite ARGS((unsigned char, octa, octa));
extern octa mmix_fputs ARGS((unsigned char, octa));
extern octa mmix_fputws ARGS((unsigned char, octa));
extern octa mmix_fseek ARGS((unsigned char, octa));
extern octa mmix_ftell ARGS((unsigned char));
extern void print_trip_warning ARGS((int, octa));
extern void mmix_fake_stdin ARGS((FILE *));
```

**114.** The subroutine *mmgetchars*(*buf*, *size*, *addr*, *stop*) reads characters starting at address *addr* in the simulated memory and stores them in *buf*, continuing until *size* characters have been read or some other stopping criterion has been met. If *stop* < 0 there is no other criterion; if *stop* = 0 a null character will also terminate the process; otherwise *addr* is even, and two consecutive null bytes starting at an even address will terminate the process. The number of bytes read and stored, exclusive of terminating nulls, is returned.

```
⟨ Subroutines 12 ⟩ +≡
  int mmgetchars ARGS((char *, int, octa, int));
  int mmgetchars(buf, size, addr, stop)
      char *buf;
      int size;
      octa addr;
      int stop;
  {
    register char *p;
    register int m;
    register mem_tetra *ll;
    register tetra x;
    octa a;
    for (p = buf, m = 0, a = addr; m < size; ) {
      ll = mem_find(a); test_load_bkpt(ll);
      x = ll→tet;
      if ((a.l & #3) ∨ m > size − 4) ⟨ Read and store one byte; return if done 115 ⟩
      else ⟨ Read and store up to four bytes; return if done 116 ⟩
    }
    return size;
  }
```

**115.**

```
⟨ Read and store one byte; return if done 115 ⟩ ≡
  {
    *p = (x ≫ (8 * ((∼a.l) & #3))) & #ff;
    if (¬*p ∧ stop ≥ 0) {
      if (stop ≡ 0) return m;
      if ((a.l & #1) ∧ *(p − 1) ≡ '\0') return m − 1;
    }
    p++, m++, a = incr(a, 1);
  }
```

This code is used in section 114.

**116.**

```
⟨ Read and store up to four bytes; return if done 116 ⟩ ≡
  {
    *p = x ≫ 24;
    if (¬*p ∧ (stop ≡ 0 ∨ (stop > 0 ∧ x < #10000))) return m;
    *(p + 1) = (x ≫ 16) & #ff;
    if (¬*(p + 1) ∧ stop ≡ 0) return m + 1;
    *(p + 2) = (x ≫ 8) & #ff;
    if (¬*(p + 2) ∧ (stop ≡ 0 ∨ (stop > 0 ∧ (x & #ffff) ≡ 0))) return m + 2;
    *(p + 3) = x & #ff;
    if (¬*(p + 3) ∧ stop ≡ 0) return m + 3;
    p += 4, m += 4, a = incr(a, 4);
  }
```

This code is used in section 114.

**117.** The subroutine *mmputchars*(*buf*, *size*, *addr*) puts *size* characters into the simulated memory starting at address *addr*.

```
⟨ Subroutines 12 ⟩ +≡
  void mmputchars ARGS((unsigned char *, int, octa));
  void mmputchars(buf, size, addr)
      unsigned char *buf;
      int size;
      octa addr;
  {
    register unsigned char *p;
    register int m;
    register mem_tetra *ll;
    octa a;
    for (p = buf, m = 0, a = addr; m < size; ) {
      ll = mem_find(a); test_store_bkpt(ll);
      if ((a.l & #3) ∨ m > size − 4) ⟨ Load and write one byte 118 ⟩
      else ⟨ Load and write four bytes 119 ⟩;
    }
  }
```

**118.**
```
⟨ Load and write one byte 118 ⟩ ≡
  {
    register int s = 8 * ((∼a.l) & #3);
    ll→tet ⊕= (((ll→tet ≫ s) ⊕ *p) & #ff) ≪ s;
    p++, m++, a = incr(a, 1);
  }
```
This code is used in section 117.

**119.**
```
⟨ Load and write four bytes 119 ⟩ ≡
  {
    ll→tet = (*p ≪ 24) + (*(p + 1) ≪ 16) + (*(p + 2) ≪ 8) + *(p + 3);
    p += 4, m += 4, a = incr(a, 4);
  }
```
This code is used in section 117.

**120.** When standard input is being read by the simulated program at the same time as it is being used for interaction, we try to keep the two uses separate by maintaining a private buffer for the simulated program's `StdIn`. Online input is usually transmitted from the keyboard to a C program a line at a time; therefore an *fgets* operation works much better than *fread* when we prompt for new input. But there is a slight complication, because *fgets* might read a null character before coming to a newline character. We cannot deduce the number of characters read by *fgets* simply by looking at *strlen*(*stdin_buf*).

```
⟨ Subroutines 12 ⟩ +≡
  char stdin_chr ARGS((void));
  char stdin_chr()
  {
    register char *p;
    while (stdin_buf_start ≡ stdin_buf_end) {
      if (interacting) {
        printf("StdIn>␣"); fflush(stdout);
      }
      if (¬fgets(stdin_buf, 256, stdin))
        panic("End␣of␣file␣on␣standard␣input;␣use␣the␣-f␣option,␣not␣<");
      stdin_buf_start = stdin_buf;
      for (p = stdin_buf; p < stdin_buf + 254; p++)
        if (*p ≡ '\n') break;
      stdin_buf_end = p + 1;
    }
    return *stdin_buf_start++;
  }
```

**121.**
```
⟨ Global variables 19 ⟩ +≡
  char stdin_buf[256];      /* standard input to the simulated program */
  char *stdin_buf_start;    /* current position in that buffer */
  char *stdin_buf_end;      /* current end of that buffer */
```

**122.** Just after executing each instruction, we do the following. Underflow that is exact and not enabled is ignored. (This applies also to underflow that was triggered by `RESUME_SET`.)

```
⟨ Check for trip interrupt 122 ⟩ ≡
  if ((exc & (U_BIT + X_BIT)) ≡ U_BIT ∧ ¬(g[rA].l & U_BIT)) exc &= ∼U_BIT;
  if (exc) {
    if (exc & tracing_exceptions) tracing = true;
    j = exc & (g[rA].l | H_BIT);      /* find all exceptions that have been enabled */
    if (j) ⟨ Initiate a trip interrupt 123 ⟩;
    g[rA].l |= exc ≫ 8;
  }
```

This code is used in section 60.

**123.** ⟨ Initiate a trip interrupt 123 ⟩ ≡

```
{
  tripping = true;
  for (k = 0; ¬(j & H_BIT); j ≪= 1, k++) ;
  exc &= ∼(H_BIT ≫ k);      /* trips taken are not logged as events */
  g[rW] = inst_ptr;
  inst_ptr.h = 0, inst_ptr.l = k ≪ 4;
  g[rX].h = sign_bit, g[rX].l = inst;
  if ((op & #e0) ≡ STB) g[rY] = w, g[rZ] = b;
  else g[rY] = y, g[rZ] = z;
  g[rB] = g[255];
  g[255] = g[rJ];
  if (op ≡ TRIP) w = g[rW], x = g[rX], a = g[255];
}
```

This code is used in section 122.

**124.** We are finally ready for the last case.

⟨ Cases for individual MMIX instructions 84 ⟩ +≡

```
case RESUME: if (xx ∨ yy ∨ zz) goto illegal_inst;
  inst_ptr = z = g[rW];
  b = g[rX];
  if (¬(b.h & sign_bit)) ⟨ Prepare to perform a ropcode 125 ⟩;
  break;
```

**125.** Here we check to see if the ropcode restrictions hold. If so, the ropcode will actually be obeyed on the next fetch phase.

```
#define RESUME_AGAIN 0    /* repeat the command in rX as if in location rW − 4 */
#define RESUME_CONT 1     /* same, but substitute rY and rZ for operands */
#define RESUME_SET 2      /* set register $X to rZ */
```

⟨ Prepare to perform a ropcode 125 ⟩ ≡

```
{
  rop = b.h ≫ 24;      /* the ropcode is the leading byte of rX */
  switch (rop) {
  case RESUME_CONT: if ((1 ≪ (b.l ≫ 28)) & #8f30) goto illegal_inst;
  case RESUME_SET: k = (b.l ≫ 16) & #ff;
    if (k ≥ L ∧ k < G) goto illegal_inst;
  case RESUME_AGAIN: if ((b.l ≫ 24) ≡ RESUME) goto illegal_inst;
    break;
  default: goto illegal_inst;
  }
  resuming = true;
}
```

This code is used in section 124.

**126.** ⟨ Install special operands when resuming an interrupted operation 126 ⟩ ≡

```
if (rop ≡ RESUME_SET) {
  op = ORI;
  y = g[rZ];
  z = zero_octa;
  exc = g[rX].h & #ff00;
  f = X_is_dest_bit;
} else {      /* RESUME_CONT */
  y = g[rY];
  z = g[rZ];
}
```

This code is used in section 71.

**127.** We don't want to count the UNSAVE that bootstraps the whole process.

⟨ Update the clocks 127 ⟩ ≡

```
if (sclock.l ∨ sclock.h ∨ ¬resuming) {
  sclock.h += info[op].mems;      /* clock goes up by 2^32 for each μ */
  sclock = incr(sclock, info[op].oops);      /* clock goes up by 1 for each υ */
  if ((¬(loc.h & sign_bit) ∨ (g[rU].h & #8000)) ∧
        ((op & (g[rU].h ≫ 16)) ≡ (g[rU].h ≫ 24))) {
    g[rU].l++;
    if (g[rU].l ≡ 0) { g[rU].h++; if ((g[rU].h & #7fff) ≡ 0) g[rU].h −= #8000; }
  }      /* usage counter counts matched instructions simulated */
  if (g[rI].l ≤ info[op].oops ∧ g[rI].l ∧ g[rI].h ≡ 0) tracing = breakpoint = true;
  g[rI] = incr(g[rI], −info[op].oops);      /* interval υ timer counts down */
}
```

This code is used in section 60.

**128. Tracing.** After an instruction has been executed, we often want to display its effect. This part of the program prints out a symbolic interpretation of what has just happened.

```
⟨ Trace the current instruction, if requested 128 ⟩ ≡
  if (tracing) {
    if (showing_source ∧ cur_line) show_line();
    ⟨ Print the frequency count, the location, and the instruction 130 ⟩;
    ⟨ Print a stream-of-consciousness description of the instruction 131 ⟩;
    if (showing_stats ∨ breakpoint) show_stats(breakpoint);
    just_traced = true;
  } else if (just_traced) {
    printf("␣.....\n");
    just_traced = false;
    shown_line = −gap − 1;      /* gap will not be filled */
  }
```

This code is used in section 60.

**129.**

```
⟨ Global variables 19 ⟩ +≡
  bool showing_stats;      /* should traced instructions also show the statistics? */
  bool just_traced;      /* was the previous instruction traced? */
```

**130.**

```
⟨ Print the frequency count, the location, and the instruction 130 ⟩ ≡
  if (resuming ∧ op ≠ RESUME) {
    switch (rop) {
    case RESUME_AGAIN:
      printf("␣␣␣␣␣␣␣␣␣␣␣(%08x%08x:␣%08x␣(%s))␣", loc.h, loc.l, inst, info[op].name); break;
    case RESUME_CONT:
      printf("␣␣␣␣␣␣␣␣␣␣␣(%08x%08x:␣%04xrYrZ␣(%s))␣", loc.h, loc.l, inst ≫ 16, info[op].name); break;
    case RESUME_SET:
      printf("␣␣␣␣␣␣␣␣␣␣␣(%08x%08x:␣..%02x..rZ␣(SET))␣", loc.h, loc.l, (inst ≫ 16) & #ff); break;
    }
  } else {
    ll = mem_find(loc);
    printf("%10d.␣%08x%08x:␣%08x␣(%s)␣", ll→freq, loc.h, loc.l, inst, info[op].name);
  }
```

This code is used in section 128.

**131.** This part of the simulator was inspired by ideas of E. H. Satterthwaite, *Software—Practice and Experience* **2** (1972), 197–217. Online debugging tools have improved significantly since Satterthwaite published his work, but good offline tools are still valuable; alas, today's algebraic programming languages do not provide tracing facilities that come anywhere close to the level of quality that Satterthwaite was able to demonstrate for ALGOL in 1970.

```
⟨ Print a stream-of-consciousness description of the instruction 131 ⟩ ≡
  if (lhs[0] ≡ '!') printf("%s␣instruction!\n", lhs + 1);      /* privileged or illegal */
  else {
    ⟨ Print changes to rL 132 ⟩;
    if (z.l ≡ 0 ∧ (op ≡ ADDUI ∨ op ≡ ORI)) p = "%l␣=␣%y␣=␣%#x";      /* LDA, SET */
    else p = info[op].trace_format;
    for ( ; *p; p++) ⟨ Interpret character *p in the trace format 133 ⟩;
    if (exc) printf(",␣rA=#%05x", g[rA].l);
    if (tripping) tripping = false, printf(",␣->␣#%02x", inst_ptr.l);
    printf("\n");
  }
```

This code is used in section 128.

**132.** Push, pop, and UNSAVE instructions display changes to rL and rO explicitly; otherwise the change is implicit, if $L \neq old\_L$.

```
⟨ Print changes to rL 132 ⟩ ≡
  if (L ≠ old_L ∧ ¬(f & push_pop_bit)) printf("rL=%d,␣", L);
```

This code is used in section 131.

**133.** Each `MMIX` instruction has a *trace format* string, which defines its symbolic representation. For example, the string for `ADD` is `"%l␣=␣%y␣+␣%z␣=␣%x"`; if the instruction is, say, `ADD $1,$2,$3` with $\$2 = 5$ and $\$3 = 8$, and if the stack offset is 100, the trace output will be `"$1=l[101]␣=␣5␣+␣8␣=␣13"`.

Percent signs (`%`) induce special format conventions, as follows:

• `%a`, `%b`, `%p`, `%q`, `%w`, `%x`, `%y`, and `%z` stand for the numeric contents of octabytes $a$, $b$, $ma$, $mb$, $w$, $x$, $y$, and $z$, respectively; a "style" character may follow the percent sign in this case, as explained below.

• `%(` and `%)` are brackets that indicate the mode of floating point rounding. If *round_mode* = `ROUND_NEAR`, `ROUND_OFF`, `ROUND_UP`, `ROUND_DOWN`, the corresponding brackets are `(` and `)`, `[` and `]`, `^` and `^`, `_` and `_`. Such brackets are placed around a floating point operator; for example, floating point addition is denoted by '`[+]`' when the current rounding mode is rounding-off.

• `%l` stands for the string *lhs*, which usually represents the "left hand side" of the instruction just performed, formatted as a register number and its equivalent in the ring of local registers (e.g., '`$1=l[101]`') or as a register number and its equivalent in the array of global registers (e.g., '`$255=g[255]`'). The `POP` instruction uses *lhs* to indicate how the "hole" in the register stack was plugged.

• `%r` means to switch to string *rhs* and continue formatting from there. This mechanism allows us to use variable formats for opcodes like `TRAP` that have several variants.

• `%t` means to print either '`Yes, ->loc`' (where `loc` is the location of the next instruction) or '`No`', depending on the value of $x$.

• `%g` means to print '` (bad guess)`' if *good* is *false*.

• `%s` stands for the name of special register $g[zz]$.

• `%?` stands for omission of the following operator if $z = 0$. For example, the memory address of `LDBI` is described by '`%#y%?+`'; this means to treat the address as simply '`%#y`' if $z = 0$, otherwise as '`%#y+%z`'. This case is used only when $z$ is a relatively small number ($z.h = 0$).

⟨ Interpret character $*p$ in the trace format 133 ⟩ ≡

```
{
  if (*p ≠ '%') fputc(*p, stdout);
  else {
    style = decimal;
  char_switch:
    switch (*++p) {
    ⟨ Cases for formatting characters 134 ⟩;
    default: printf("BUG!!");      /* can't happen */
    }
  }
}
```

This code is used in section 131.

**134.** Octabytes are printed as decimal numbers unless a "style" character intervenes between the percent sign and the name of the octabyte: '`#`' denotes hexadecimal notation, prefixed by `#`; '`0`' denotes hexadecimal notation with no prefixed `#` and with leading zeros not suppressed; '`.`' denotes floating decimal notation; and '`!`' means to use the names `StdIn`, `StdOut`, or `StdErr` if the value is 0, 1, or 2.

⟨ Cases for formatting characters 134 ⟩ ≡

```
case '#': style = hex; goto char_switch;
case '0': style = zhex; goto char_switch;
case '.': style = floating; goto char_switch;
case '!': style = handle; goto char_switch;
```

See also sections 136 and 138.

This code is used in section 133.

**135.** ⟨ Type declarations 9 ⟩ +≡
```
typedef enum {
  decimal, hex, zhex, floating, handle
} fmt_style;
```

**136.** ⟨ Cases for formatting characters 134 ⟩ +≡
```
case 'a': trace_print(a); break;
case 'b': trace_print(b); break;
case 'p': trace_print(ma); break;
case 'q': trace_print(mb); break;
case 'w': trace_print(w); break;
case 'x': trace_print(x); break;
case 'y': trace_print(y); break;
case 'z': trace_print(z); break;
```

**137.** ⟨ Subroutines 12 ⟩ +≡
```
fmt_style style;
char *stream_name[] = {"StdIn","StdOut","StdErr"};
void trace_print ARGS((octa));
void trace_print(o)
    octa o;
{
  switch (style) {
  case decimal: print_int(o); return;
  case hex: fputc('#',stdout); print_hex(o); return;
  case zhex: printf("%08x%08x",o.h,o.l); return;
  case floating: print_float(o); return;
  case handle: if (o.h ≡ 0 ∧ o.l < 3) printf("%s",stream_name[o.l]);
    else print_int(o); return;
  }
}
```

**138.** ⟨ Cases for formatting characters 134 ⟩ +≡
```
case '(': fputc(left_paren[round_mode],stdout); break;
case ')': fputc(right_paren[round_mode],stdout); break;
case 't': if (x.l) printf("␣Yes,␣->␣#"),print_hex(inst_ptr);
  else printf("␣No"); break;
case 'g': if (¬good) printf("␣(bad␣guess)"); break;
case 's': printf("%s",special_name[zz]); break;
case '?': p++; if (z.l) printf("%c%d",*p,z.l); break;
case 'l': printf("%s",lhs); break;
case 'r': p = switchable_string; break;
```

**139.** **#define** *rhs* *&switchable_string*[1]

⟨ Global variables 19 ⟩ +≡
```
char left_paren[] = {0,'[','^','_','('};    /* denotes the rounding mode */
char right_paren[] = {0,']','^','_',')'};    /* denotes the rounding mode */
char switchable_string[48];    /* holds rhs; position 0 is ignored */
  /* switchable_string must be able to hold any trap_format */
char lhs[32];
int good_guesses, bad_guesses;    /* branch prediction statistics */
```

**140.** ⟨ Subroutines 12 ⟩ +≡

```
void show_stats ARGS((bool));
void show_stats(verbose)
    bool verbose;
{
  octa o;
  printf("␣␣%d␣instruction%s,␣%d␣mem%s,␣%d␣oop%s;␣%d␣good␣guess%s,␣%d␣bad\n",g[rU].l,
      g[rU].l ≡ 1 ? "" : "s",
      sclock.h,sclock.h ≡ 1 ? "" : "s",
      sclock.l,sclock.l ≡ 1 ? "" : "s",
      good_guesses,good_guesses ≡ 1 ? "" : "es",bad_guesses);
  if (¬verbose) return;
  o = halted ? incr(inst_ptr,-4) : inst_ptr;
  printf("␣␣(%s␣at␣location␣#%08x%08x)\n",halted ? "halted" : "now",o.h,o.l);
}
```

**141. Running the program.** Now we are ready to fit the pieces together into a working simulator.

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>
#include <signal.h>
#include "abstime.h"
  ⟨Preprocessor macros 11⟩
  ⟨Type declarations 9⟩
  ⟨Global variables 19⟩
  ⟨Subroutines 12⟩
  int main(argc, argv)
      int argc;
      char *argv[];
  {
    ⟨Local registers 62⟩;
    mmix_io_init();
    ⟨Process the command line 142⟩;
    ⟨Initialize everything 14⟩;
    ⟨Load the command line arguments 163⟩;
    ⟨Get ready to UNSAVE the initial context 164⟩;
    while (1) {
      if (interrupt ∧ ¬breakpoint) breakpoint = interacting = true, interrupt = false;
      else {
        breakpoint = false;
        if (interacting) ⟨Interact with the user 149⟩;
      }
      if (halted) break;
      do ⟨Perform one instruction 60⟩ while ((¬interrupt ∧ ¬breakpoint) ∨ resuming);
      if (interact_after_break) interacting = true, interact_after_break = false;
    }
  end_simulation: if (profiling) ⟨Print all the frequency counts 53⟩;
    if (interacting ∨ profiling ∨ showing_stats) show_stats(true);
    return g[255].l;      /* provide rudimentary feedback for non-interactive runs */
  }
```

**142.** Here we process the command line options; when we finish, $*cur\_arg$ should be the name of the object file to be loaded and simulated.

We assume that $argv[0]$ is never null. (The author believes strongly that the wizards who decided to allow $argc = 0$ were mistaken when they defined the C89 standard; hence he has taken no pains to avoid system crashes when people try to invoke any of his programs with a null environment. Null invocations are contrary to the intent of C's designers.)

```
#define mmo_file_name *cur_arg
⟨Process the command line 142⟩ ≡
  myself = argv[0];
  for (cur_arg = argv + 1; *cur_arg ∧ (*cur_arg)[0] ≡ '-'; cur_arg++) scan_option(*cur_arg + 1, true);
  if (¬*cur_arg) scan_option("?", true);      /* exit with usage note */
  argc -= cur_arg - argv;      /* this is the argc of the user program */
```

This code is used in section 141.

**143.** Careful readers of the following subroutine will notice a little white bug: A tracing specification like `t1000000000` or even `t0000000000` or even `t!!!!!!!!!!` is silently converted to `t4294967295`.

The `-b` and `-c` options are effective only on the command line, but they are harmless while interacting.

```
⟨ Subroutines 12 ⟩ +≡
  void scan_option ARGS((char *, bool));
  void scan_option(arg, usage)
      char *arg;      /* command line argument (without the '-') */
      bool usage;     /* should we exit with usage note if unrecognized? */
  {
    register int k;
    switch (*arg) {
    case 't': if (strlen(arg) > 10) trace_threshold = #ffffffff;
      else if (sscanf(arg + 1, "%d", (int *) &trace_threshold) ≠ 1) trace_threshold = 0;
      return;
    case 'e': if (¬*(arg + 1)) tracing_exceptions = #ff;
      else if (sscanf(arg + 1, "%x", &tracing_exceptions) ≠ 1) tracing_exceptions = 0;
      return;
    case 'r': stack_tracing = true; return;
    case 's': showing_stats = true; return;
    case 'l': if (¬*(arg + 1)) gap = 3;
      else if (sscanf(arg + 1, "%d", &gap) ≠ 1) gap = 0;
      showing_source = true; return;
    case 'L': if (¬*(arg + 1)) profile_gap = 3;
      else if (sscanf(arg + 1, "%d", &profile_gap) ≠ 1) profile_gap = 0;
      profile_showing_source = true;
    case 'P': profiling = true; return;
    case 'v': trace_threshold = #ffffffff; tracing_exceptions = #ff;
      stack_tracing = true; showing_stats = true;
      gap = 10, showing_source = true;
      profile_gap = 10, profile_showing_source = true, profiling = true;
      return;
    case 'q': trace_threshold = tracing_exceptions = 0;
      stack_tracing = showing_stats = showing_source = false;
      profiling = profile_showing_source = false;
      return;
    case 'i': interacting = true; return;
    case 'I': interact_after_break = true; return;
    case 'b': if (sscanf(arg + 1, "%d", &buf_size) ≠ 1) buf_size = 0; return;
    case 'c': if (sscanf(arg + 1, "%d", &lring_size) ≠ 1) lring_size = 0; return;
    case 'f': ⟨ Open a file for simulated standard input 145 ⟩; return;
    case 'D': ⟨ Open a file for dumping binary output 146 ⟩; return;
    default: if (usage) {
        fprintf(stderr, "Usage: %s <options> progfile command line-args...\n", myself);
        for (k = 0; usage_help[k][0]; k++) fprintf(stderr, "%s", usage_help[k]);
        exit(-1);
      } else for (k = 0; usage_help[k][1] ≠ 'b'; k++) printf("%s", usage_help[k]);
      return;
    }
  }
```

**144.** ⟨ Global variables 19 ⟩ +≡

```
char *myself;  /* argv[0], the name of this simulator */
char **cur_arg;  /* pointer to current place in the argument vector */
bool interrupt;  /* has the user interrupted the simulation recently? */
bool profiling;  /* should we print the profile at the end? */
FILE *fake_stdin;  /* file substituted for the simulated StdIn */
FILE *dump_file;  /* file used for binary dumps */
char *usage_help[] = {
    " with these options: (<n>=decimal number, <x>=hex number)\n",
    "-t<n> trace each instruction the first n times\n",
    "-e<x> trace each instruction with an exception matching x\n",
    "-r     trace hidden details of the register stack\n",
    "-l<n> list source lines when tracing, filling gaps <= n\n",
    "-s     show statistics after each traced instruction\n",
    "-P     print a profile when simulation ends\n",
    "-L<n> list source lines with the profile\n",
    "-v     be verbose: show almost everything\n",
    "-q     be quiet: show only the simulated standard output\n",
    "-i     run interactively (prompt for online commands)\n",
    "-I     interact, but only after the program halts\n",
    "-b<n> change the buffer size for source lines\n",
    "-c<n> change the cyclic local register ring size\n",
    "-f<filename> use given file to simulate standard input\n",
    "-D<filename> dump a file for use by other simulators\n", ""};
char *interactive_help[] = {
    "The interactive commands are:\n",
    "<return>  trace one instruction\n",
    "n         trace one instruction\n",
    "c         continue until halt or breakpoint\n",
    "q         quit the simulation\n",
    "s         show current statistics\n",
    "l<n><t>   set and/or show local register in format t\n",
    "g<n><t>   set and/or show global register in format t\n",
    "rA<t>     set and/or show register rA in format t\n",
    "$<n><t>   set and/or show dynamic register in format t\n",
    "M<x><t>   set and/or show memory octabyte in format t\n",
    "+<n><t>   set and/or show n additional octabytes in format t\n",
    "_<t> is ! (decimal) or . (floating) or # (hex) or \" (string)\n",
    "      or <empty> (previous <t>) or =<value> (change value)\n",
    "@<x>      go to location x\n",
    "b[rwx]<x> set or reset breakpoint at location x\n",
    "t<x>      trace location x\n",
    "u<x>      untrace location x\n",
    "T         set current segment to Text_Segment\n",
    "D         set current segment to Data_Segment\n",
    "P         set current segment to Pool_Segment\n",
    "S         set current segment to Stack_Segment\n",
    "B         show all current breakpoints and tracepoints\n",
    "i<file>   insert commands from file\n",
    "-<option> change a tracing/listing/profile option\n",
    "-?        show the tracing/listing/profile options\n", ""};
```

**145.** ⟨ Open a file for simulated standard input 145 ⟩ ≡

```
if (fake_stdin) fclose(fake_stdin);
fake_stdin = fopen(arg + 1, "r");
if (¬fake_stdin) fprintf(stderr, "Sorry, I can't open file %s!\n", arg + 1);
else mmix_fake_stdin(fake_stdin);
```

This code is used in section 143.

**146.** ⟨ Open a file for dumping binary output 146 ⟩ ≡

```
dump_file = fopen(arg + 1, "wb");
if (¬dump_file) fprintf(stderr, "Sorry, I can't open file %s!\n", arg + 1);
```

This code is used in section 143.

**147.** ⟨ Initialize everything 14 ⟩ +≡

```
signal(SIGINT, catchint);      /* now catchint will catch the first interrupt */
```

**148.** ⟨ Subroutines 12 ⟩ +≡

```
void catchint ARGS((int));
void catchint(n)
    int n;
{
  if (n ≠ SIGINT) return;
  interrupt = true;
  signal(SIGINT, catchint);      /* now catchint will catch the next interrupt */
}
```

**149.** ⟨ Interact with the user 149 ⟩ ≡

```
{ register int repeating;
interact: ⟨ Put a new command in command_buf 150 ⟩;
  p = command_buf;
  repeating = 0;
  switch (*p) {
  case '\n': case 'n': breakpoint = tracing = true;      /* trace one inst and break */
  case 'c': goto resume_simulation;      /* continue until breakpoint */
  case 'q': goto end_simulation;
  case 's': show_stats(true); goto interact;
  case '-': k = strlen(p); if (p[k-1] ≡ '\n') p[k-1] = '\0';
    scan_option(p+1, false); goto interact;
  ⟨ Cases that change cur_disp_mode 152 ⟩;
  ⟨ Cases that define cur_disp_type 153 ⟩;
  ⟨ Cases that set and clear tracing and breakpoints 161 ⟩;
  default: what_say: k = strlen(command_buf);
    if (k < 10 ∧ command_buf[k-1] ≡ '\n') command_buf[k-1] = '\0';
    else strcpy(command_buf + 9, "...");
    printf("Eh? Sorry, I don't understand '%s'. (Type h for help)\n", command_buf);
    goto interact;
  case 'h': for (k = 0; interactive_help[k][0]; k++) printf("%s", interactive_help[k]);
    goto interact;
  }
  if (*p ≠ '\n') {
    if (¬*p)
    incomplete_str: printf("Syntax error: Incomplete command!\n");
    else {
      p[strlen(p) - 1] = '\0';
      printf("Syntax error; I'm ignoring '%s'!\n", p);
    }
  }
  while (repeating) ⟨ Display and/or set the value of the current octabyte 156 ⟩;
  goto interact;
resume_simulation: ;
}
```

This code is used in section 141.

**150.** ⟨ Put a new command in *command_buf* 150 ⟩ ≡

```
{ register bool ready = false;
incl_read: while (incl_file ∧ ¬ready)
    if (¬fgets(command_buf, command_buf_size, incl_file)) {
      fclose(incl_file);
      incl_file = Λ;
    } else if (command_buf[0] ≠ '\n' ∧ command_buf[0] ≠ 'i' ∧ command_buf[0] ≠ '%') {
      if (command_buf[0] ≡ '␣') printf("%s", command_buf); else ready = true; }
  while (¬ready) {
    printf("mmix>␣"); fflush(stdout);
    if (¬fgets(command_buf, command_buf_size, stdin)) command_buf[0] = 'q';
    if (command_buf[0] ≠ 'i') ready = true;
    else {
      command_buf[strlen(command_buf) − 1] = '\0';
      incl_file = fopen(command_buf + 1, "r");
      if (incl_file) goto incl_read;
      if (isspace(command_buf[1])) incl_file = fopen(command_buf + 2, "r");
      if (incl_file) goto incl_read;
      printf("Can't␣open␣file␣`%s'!\n", command_buf + 1);
    }
  }
}
```

This code is used in section 149.

**151.** `#define` *command_buf_size* 1024 /* make it plenty long, for floating point tests */

⟨ Global variables 19 ⟩ +≡

```
char command_buf[command_buf_size];
FILE *incl_file;      /* file of commands included by `i' */
char cur_disp_mode = 'l';      /* 'l' or 'g' or '$' or 'M' */
char cur_disp_type = '!';      /* '!' or '.' or '#' or '"' */
bool cur_disp_set;      /* was the last <t> of the form =<val>? */
octa cur_disp_addr;      /* the h half is relevant only in mode 'M' */
octa cur_seg;      /* current segment offset */
char spec_reg_code[] = {rA, rB, rC, rD, rE, rF, rG, rH, rI, rJ, rK, rL, rM, rN, rO, rP, rQ, rR, rS, rT,
    rU, rV, rW, rX, rY, rZ};
char spec_regg_code[] = {0, rBB, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, rTT, 0, 0, rWW, rXX, rYY, rZZ};
```

**152.** ⟨ Cases that change *cur_disp_mode* 152 ⟩ ≡

```
case 'l': case 'g': case '$': cur_disp_mode = *p++;
  for (cur_disp_addr.l = 0; isdigit(*p); p++) cur_disp_addr.l = 10 * cur_disp_addr.l + *p - '0';
  goto new_mode;
case 'r': p++; cur_disp_mode = 'g';
  if (*p < 'A' ∨ *p > 'Z') goto what_say;
  if (*(p + 1) ≠ *p) cur_disp_addr.l = spec_reg_code[*p - 'A'], p++;
  else if (spec_regg_code[*p - 'A']) cur_disp_addr.l = spec_regg_code[*p - 'A'], p += 2;
  else goto what_say;
  goto new_mode;
case 'M': cur_disp_mode = *p;
  cur_disp_addr = scan_hex(p + 1, cur_seg); cur_disp_addr.l &= -8; p = next_char;
new_mode: cur_disp_set = false;      /* the '=' is remembered only by '+' */
  repeating = 1;
  goto scan_type;
case '+': if (¬isdigit(*(p + 1))) repeating = 1;
  for (p++; isdigit(*p); p++) repeating = 10 * repeating + *p - '0';
  if (repeating) {
    if (cur_disp_mode ≡ 'M') cur_disp_addr = incr(cur_disp_addr, 8);
    else cur_disp_addr.l++;
  }
  goto scan_type;
```

This code is used in section 149.

**153.** ⟨ Cases that define *cur_disp_type* 153 ⟩ ≡

```
case '!': case '.': case '#': case '"': cur_disp_set = false;
  repeating = 1;
set_type: cur_disp_type = *p++; break;
scan_type: if (*p ≡ '!' ∨ *p ≡ '.' ∨ *p ≡ '#' ∨ *p ≡ '"') goto set_type;
  if (*p ≠ '=') break;
  goto scan_eql;
case '=': repeating = 1;
scan_eql: cur_disp_set = true;
  val = zero_octa;
  if (*++p ≡ '#') cur_disp_type = *p, val = scan_hex(p + 1, zero_octa);
  else if (*p ≡ '"' ∨ *p ≡ '\'') goto scan_string;
  else cur_disp_type = (scan_const(p) > 0 ? '.' : '!');
  p = next_char;
  if (*p ≠ ',') break;
  val.h = 0; val.l &= #ff;
scan_string: cur_disp_type = '"';
  ⟨ Scan a string constant 155 ⟩; break;
```

This code is used in section 149.

**154.** ⟨ Subroutines 12 ⟩ +≡

```
octa scan_hex ARGS((char *, octa));
octa scan_hex(s, offset)
    char *s;
    octa offset;
{
  register char *p;
  octa o;
  o = zero_octa;
  for (p = s; isxdigit(*p); p++) {
    o = incr(shift_left(o, 4), *p - '0');
    if (*p ≥ 'a') o = incr(o, '0' - 'a' + 10);
    else if (*p ≥ 'A') o = incr(o, '0' - 'A' + 10);
  }
  next_char = p;
  return oplus(o, offset);
}
```

**155.** ⟨ Scan a string constant 155 ⟩ ≡

```
while (*p ≡ ',') {
  if (*++p ≡ '#') {
    aux = scan_hex(p + 1, zero_octa), p = next_char;
    val = incr(shift_left(val, 8), aux.l & #ff);
  } else if (isdigit(*p)) {
    for (k = *p++ - '0'; isdigit(*p); p++) k = (10 * k + *p - '0') & #ff;
    val = incr(shift_left(val, 8), k);
  }
  else if (*p ≡ '\n') goto incomplete_str;
}
if (*p ≡ '\'' ∧ *(p + 2) ≡ *p) *p = *(p + 2) = '"';
if (*p ≡ '"') {
  for (p++; *p ∧ *p ≠ '\n' ∧ *p ≠ '"'; p++) val = incr(shift_left(val, 8), *p);
  if (*p ∧ *p++ ≡ '"')
    if (*p ≡ ',') goto scan_string;
}
```

This code is used in section 153.

**156.** ⟨ Display and/or set the value of the current octabyte 156 ⟩ ≡

```
{
  if (cur_disp_set) ⟨ Set the current octabyte to val 157 ⟩;
  ⟨ Display the current octabyte 159 ⟩;
  fputc('\n', stdout);
  repeating--;
  if (¬repeating) break;
  if (cur_disp_mode ≡ 'M') cur_disp_addr = incr(cur_disp_addr, 8);
  else cur_disp_addr.l++;
}
```

This code is used in section 149.

**157.** ⟨ Set the current octabyte to *val* 157 ⟩ ≡

```
switch (cur_disp_mode) {
case 'l': l[cur_disp_addr.l & lring_mask] = val; break;
case '$': k = cur_disp_addr.l & #ff;
  if (k < L) l[(O + k) & lring_mask] = val; else if (k ≥ G) g[k] = val;
  break;
case 'g': k = cur_disp_addr.l & #ff;
  if (k < 32) ⟨ Set g[k] = val only if permissible 158 ⟩;
  g[k] = val; break;
case 'M': if (¬(cur_disp_addr.h & sign_bit)) {
    ll = mem_find(cur_disp_addr);
    ll→tet = val.h; (ll + 1)→tet = val.l;
  } break;
}
```

This code is used in section 156.

**158.** Here we essentially simulate a `PUT` command, but we simply **break** if the `PUT` is illegal or privileged.

⟨ Set $g[k] = val$ only if permissible 158 ⟩ ≡

```
if (k ≥ 9 ∧ k ≠ rI) {
  if (k ≤ 19) break;
  if (k ≡ rA) {
    if (val.h ≠ 0 ∨ val.l ≥ #40000) break;
    cur_round = (val.l ≥ #10000 ? val.l ≫ 16 : ROUND_NEAR);
  } else if (k ≡ rG) {
    if (val.h ≠ 0 ∨ val.l > (unsigned int) 255 ∨ val.l < (unsigned int) L ∨ val.l < 32) break;
    for (j = val.l; j < G; j++) g[j] = zero_octa;
    G = val.l;
  } else if (k ≡ rL) {
    if (val.h ≡ 0 ∧ val.l < (unsigned int) L) L = val.l;
    else break;
  }
}
```

This code is used in section 157.

**159.** ⟨ Display the current octabyte 159 ⟩ ≡

```
switch (cur_disp_mode) {
case 'l': k = cur_disp_addr.l & lring_mask;
  printf("l[%d]=",k); aux = l[k]; break;
case '$': k = cur_disp_addr.l & #ff;
  if (k < L) printf("$%d=l[%d]=",k,(O+k) & lring_mask), aux = l[(O+k) & lring_mask];
  else if (k ≥ G) printf("$%d=g[%d]=",k,k), aux = g[k];
  else printf("$%d=",k), aux = zero_octa;
  break;
case 'g': k = cur_disp_addr.l & #ff;
  printf("g[%d]=",k); aux = g[k]; break;
case 'M': if (cur_disp_addr.h & sign_bit) aux = zero_octa;
  else {
    ll = mem_find(cur_disp_addr);
    aux.h = ll→tet; aux.l = (ll + 1)→tet;
  }
  printf("M8[#"); print_hex(cur_disp_addr); printf("]="); break;
}
switch (cur_disp_type) {
case '!': print_int(aux); break;
case '.': print_float(aux); break;
case '#': fputc('#',stdout); print_hex(aux); break;
case '"': print_string(aux); break;
}
```

This code is used in section 156.

**160.** ⟨ Subroutines 12 ⟩ +≡

```
void print_string ARGS((octa));
void print_string(o)
    octa o;
{
  register int k, state, b;
  for (k = state = 0; k < 8; k++) {
    b = ((k < 4 ? o.h ≫ (8 * (3 - k)) : o.l ≫ (8 * (7 - k)))) & #ff;
    if (b ≡ 0) {
      if (state) printf("%s,0",state > 1 ? "\"" : ""), state = 1;
    } else if (b ≥ ' ' ∧ b ≤ '~') printf("%s%c",state > 1 ? "" : state ≡ 1 ? ",\"" : "\"",b), state = 2;
    else printf("%s#%x",state > 1 ? "\"," : state ≡ 1 ? "," : "",b), state = 1;
  }
  if (state ≡ 0) printf("0");
  else if (state > 1) printf("\"");
}
```

**161.** ⟨ Cases that set and clear tracing and breakpoints 161 ⟩ ≡

```
case '@': inst_ptr = scan_hex(p + 1, cur_seg); p = next_char;
  halted = false; break;
case 't': case 'u': k = *p;
  val = scan_hex(p + 1, cur_seg); p = next_char;
  if (val.h < #20000000) {
    ll = mem_find(val);
    if (k ≡ 't') ll→bkpt |= trace_bit;
    else ll→bkpt &= ~trace_bit;
  }
  break;
case 'b': for (k = 0, p++; ¬isxdigit(*p); p++)
    if (*p ≡ 'r') k |= read_bit;
    else if (*p ≡ 'w') k |= write_bit;
    else if (*p ≡ 'x') k |= exec_bit;
  val = scan_hex(p, cur_seg); p = next_char;
  if (¬(val.h & sign_bit)) {
    ll = mem_find(val);
    ll→bkpt = (ll→bkpt & −8) | k;
  }
  break;
case 'T': cur_seg.h = 0; goto passit;
case 'D': cur_seg.h = #20000000; goto passit;
case 'P': cur_seg.h = #40000000; goto passit;
case 'S': cur_seg.h = #60000000; goto passit;
case 'B': show_breaks(mem_root);
passit: p++; break;
```

This code is used in section 149.

**162.** ⟨ Subroutines 12 ⟩ +≡

```
  void show_breaks ARGS((mem_node *));
  void show_breaks(p)
      mem_node *p;
  {
    register int j;
    octa cur_loc;
    if (p→left) show_breaks(p→left);
    for (j = 0; j < 512; j++)
      if (p→dat[j].bkpt) {
        cur_loc = incr(p→loc, 4 * j);
        printf("␣␣%08x%08x␣%c%c%c%c\n", cur_loc.h, cur_loc.l,
            p→dat[j].bkpt & trace_bit ? 't' : '-', p→dat[j].bkpt & read_bit ? 'r' : '-',
            p→dat[j].bkpt & write_bit ? 'w' : '-', p→dat[j].bkpt & exec_bit ? 'x' : '-');
      }
    if (p→right) show_breaks(p→right);
  }
```

**163.** We put pointers to the command line strings in $M_8[\texttt{Pool\_Segment} + 8 * (k + 1)]$ for $0 \le k < argc$; the strings themselves are octabyte-aligned, starting at $M_8[\texttt{Pool\_Segment} + 8 * (argc + 2)]$. The location of the first free octabyte in the pool segment is placed in $M_8[\texttt{Pool\_Segment}]$.

```
⟨ Load the command line arguments 163 ⟩ ≡
  x.h = #40000000, x.l = #8;
  loc = incr(x, 8 * (argc + 1));
  for (k = 0; k < argc; k++, cur_arg++) {
    ll = mem_find(x);
    ll→tet = loc.h, (ll + 1)→tet = loc.l;
    ll = mem_find(loc);
    mmputchars((unsigned char *) *cur_arg, strlen(*cur_arg), loc);
    x.l += 8, loc.l += 8 + (strlen(*cur_arg) & -8);
  }
  x.l = 0; ll = mem_find(x); ll→tet = loc.h, (ll + 1)→tet = loc.l;
```

This code is used in section 141.

**164.**

```
⟨ Get ready to UNSAVE the initial context 164 ⟩ ≡
  x.h = 0, x.l = #f0;
  ll = mem_find(x);
  if (ll→tet) inst_ptr = x;
  resuming = true;
  rop = RESUME_AGAIN;
  g[rX].l = ((tetra) UNSAVE ≪ 24) + 255;
  if (dump_file) {
    x.l = 1;
    dump(mem_root);
    dump_tet(0), dump_tet(0);
    exit(0);
  }
```

This code is used in section 141.

**165.** The special option '`-D<filename>`' can be used to prepare binary files needed by the `MMIX-in-MMIX` simulator of Section 1.4.3′. (See *The Art of Computer Programming*, Volume 1, Fascicle 1.) This option puts big-endian octabytes into a given file; a location $l$ is followed by one or more nonzero octabytes $M_8[l]$, $M_8[l+8]$, $M_8[l+16]$, ..., followed by zero. The simulated simulator knows how to load programs in such a format (see exercise 1.4.3′–20), and so does the meta-simulator MMMIX.

```
⟨ Subroutines 12 ⟩ +≡
  void dump ARGS((mem_node *));
  void dump_tet ARGS((tetra));
  void dump(p)
      mem_node *p;
  {
    register int j;
    octa cur_loc;
    if (p→left) dump(p→left);
    for (j = 0; j < 512; j += 2)
      if (p→dat[j].tet ∨ p→dat[j + 1].tet) {
        cur_loc = incr(p→loc, 4 * j);
        if (cur_loc.l ≠ x.l ∨ cur_loc.h ≠ x.h) {
          if (x.l ≠ 1) dump_tet(0), dump_tet(0);
          dump_tet(cur_loc.h); dump_tet(cur_loc.l); x = cur_loc;
        }
        dump_tet(p→dat[j].tet);
        dump_tet(p→dat[j + 1].tet);
        x = incr(x, 8);
      }
    if (p→right) dump(p→right);
  }
```

**166.**

```
⟨ Subroutines 12 ⟩ +≡
  void dump_tet(t)
      tetra t;
  {
    fputc(t ≫ 24, dump_file);
    fputc((t ≫ 16) & #ff, dump_file);
    fputc((t ≫ 8) & #ff, dump_file);
    fputc(t & #ff, dump_file);
  }
```

## 167. Index.

May 1, 2025 at 02:23

# MMIX-SIM