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 10υ, `LDB` takes $\mu + \upsilon$; all times are expressed in terms of μ and υ, "mems" and "oops." The simulated clock increases by 2^{32} for each μ and 1 for each υ. But the interval counter rI decreases by 1 for each υ; 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 g22, 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 250υ 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 $offset \ge 0$, 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 $M_8[\$0 \ll 3+\$1]$, and $M_8[\$0 \ll 3+\$1+8]$ 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
```