1. Introduction to MMIX. Thirty-eight years have passed since the MIX computer was designed, and computer architecture has been converging during those years towards a rather different style of machine. Therefore it is time to replace MIX with a new computer that contains even less saturated fat than its predecessor.

Exercise 1.3.1–25 in the third edition of *Fundamental Algorithms* speaks of an extended MIX called MixMaster, which is upward compatible with the old version. But MixMaster itself is hopelessly obsolete; although it allows for several gigabytes of memory, we can't even use it with ASCII code to get lowercase letters. And ouch, the standard subroutine calling convention of MIX is irrevocably based on self-modifying code! Decimal arithmetic and self-modifying code were popular in 1962, but they sure have disappeared quickly as machines have gotten bigger and faster. A completely new design is called for, based on the principles of RISC architecture as expounded in *Computer Architecture* by Hennessy and Patterson (Morgan Kaufmann, 1996).

So here is MMIX, a computer that will totally replace MIX in the "ultimate" editions of *The Art of Computer Programming*, Volumes 1–3, and in the first editions of the remaining volumes. I must confess that I can hardly wait to own a computer like this.

How do you pronounce MMIX? I've been saying "em-mix" to myself, because the first 'M' represents a new millennium. Therefore I use the article "an" instead of "a" before the name MMIX in English phrases like "an MMIX simulator."

Incidentally, the *Dictionary of American Regional English* **3** (1996) lists "mommix" as a common dialect word used both as a noun and a verb; to mommix something means to botch it, to bollix it. Only time will tell whether I have mommixed the definition of MMIX.

2. The original MIX computer could be operated without an operating system; you could bootstrap it with punched cards or paper tape and do everything yourself. But nowadays such power is no longer in the hands of ordinary users. The MMIX hardware, like all other computing machines made today, relies on an operating system to get jobs started in their own address spaces and to provide I/O capabilities.

Whenever anybody has asked if I will be writing about operating systems, my reply has always been "Nix." Therefore the name of MMIX's operating system, NNIX, will come as no surprise. From time to time I will necessarily have to refer to things that NNIX does for its users, but I am unable to build NNIX myself. Life is too short. It would be wonderful if some expert in operating system design became inspired to write a book that explains exactly how to construct a nice, clean NNIX kernel for an MMIX chip.

3. I am deeply grateful to the many people who have helped me shape the behavior of MMIX. In particular, John Hennessy and (especially) Dick Sites have made significant contributions.

4. A programmer's introduction to MMIX appears in "Fascicle 1," a booklet containing tutorial material that will ultimately appear in the fourth edition of *The Art of Computer Programming*. The description in the following sections is rather different, because we are concerned about a complete implementation, including all of the features used by the operating system and invisible to normal programs. Here it is important to emphasize exceptional cases that were glossed over in the tutorial, and to consider nitpicky details about things that might go wrong.

5. MMIX basics. `MMIX` is a 64-bit RISC machine with at least 256 general-purpose registers and a 64-bit address space. Every instruction is four bytes long and has the form

OP	X	Y	Z

.

The 256 possible OP codes fall into a dozen or so easily remembered categories; an instruction usually means, "Set register X to the result of Y OP Z." For example,

32	1	2	3

sets register 1 to the sum of registers 2 and 3. A few instructions combine the Y and Z bytes into a 16-bit YZ field; two of the jump instructions use a 24-bit XYZ field. But the three bytes X, Y, Z usually have three-pronged significance independent of each other.

Instructions are usually represented in a symbolic form corresponding to the `MMIX` assembly language, in which each operation code has a mnemonic name. For example, operation 32 is `ADD`, and the instruction above might be written '`ADD $1,$2,$3`'; a dollar sign '`$`' symbolizes a register number. In general, the instruction `ADD $X,$Y,$Z` is the operation of setting $\$X = \$Y + \$Z$. An assembly language instruction with two commas has three operand fields X, Y, Z; an instruction with one comma has two operand fields X, YZ; an instruction with no comma has one operand field, XYZ; an instruction with no operands has $X = Y = Z = 0$.

Most instructions have two forms, one in which the Z field stands for register \$Z, and one in which Z is an unsigned "immediate" constant. Thus, for example, the command '`ADD $X,$Y,$Z`' has a counterpart '`ADD $X,$Y,Z`', which sets $\$X = \$Y + Z$. Immediate constants are always nonnegative. In the descriptions below we will introduce such pairs of instructions by writing just '`ADD $X,$Y,$Z|Z`' instead of naming both cases explicitly.

The operation code for `ADD $X,$Y,$Z` is 32, but the operation code for `ADD $X,$Y,Z` is 33. The `MMIX` assembler chooses the correct code by noting whether the third argument is a register number or not.

Register numbers and constants can be given symbolic names; for example, the assembly language instruction '`x IS $1`' makes `x` an abbreviation for register number 1. Similarly, '`FIVE IS 5`' makes `FIVE` an abbreviation for the constant 5. After these abbreviations have been specified, the instruction `ADD x,x,FIVE` increases \$1 by 5, using opcode 33, while the instruction `ADD x,x,x` doubles \$1 using opcode 32. Symbolic names that stand for register numbers conventionally begin with a lowercase letter, while names that stand for constants conventionally begin with an uppercase letter. This convention is not actually enforced by the assembler, but it tends to reduce a programmer's confusion.

6. A *nybble* is a 4-bit quantity, often used to denote a decimal or hexadecimal digit. A *byte* is an 8-bit quantity, often used to denote an alphanumeric character in ASCII code. The Unicode standard extends ASCII to essentially all the world's languages by using 16-bit-wide characters called *wydes*. (Weight watchers know that two nybbles make one byte, but two bytes make one wyde.) In the discussion below we use the term *tetrabyte* or "tetra" for a 4-byte quantity, and the similar term *octabyte* or "octa" for an 8-byte quantity. Thus, a tetra is two wydes, an octa is two tetras; an octabyte has 64 bits. Each `MMIX` register can be thought of as containing one octabyte, or two tetras, or four wydes, or eight bytes, or sixteen nybbles.

When bytes, wydes, tetras, and octas represent numbers they are said to be either *signed* or *unsigned*. An unsigned byte is a number between 0 and $2^8-1=255$ inclusive; an unsigned wyde lies, similarly, between 0 and $2^{16}-1=65535$; an unsigned tetra lies between 0 and $2^{32}-1=4{,}294{,}967{,}295$; an unsigned octa lies between 0 and $2^{64}-1=18{,}446{,}744{,}073{,}709{,}551{,}615$. Their signed counterparts use the conventions of two's complement notation, by subtracting respectively 2^8, 2^{16}, 2^{32}, or 2^{64} times the most significant bit. Thus, the unsigned bytes 128 through 255 are regarded as the numbers -128 through -1 when they are evaluated as signed bytes; a signed byte therefore lies between -128 and $+127$, inclusive. A signed wyde is a number between -32768 and $+32767$; a signed tetra lies between $-2{,}147{,}483{,}648$ and $+2{,}147{,}483{,}647$; a signed octa lies between $-9{,}223{,}372{,}036{,}854{,}775{,}808$ and $+9{,}223{,}372{,}036{,}854{,}775{,}807$.

The virtual memory of `MMIX` is an array M of 2^{64} bytes. If k is any unsigned octabyte, $\mathrm{M}[k]$ is a 1-byte quantity. `MMIX` machines do not actually have such vast memories, but programmers can act as if 2^{64} bytes are indeed present, because `MMIX` provides address translation mechanisms by which an operating system can maintain this illusion.

We use the notation $\mathrm{M}_{2^t}[k]$ to stand for a number consisting of 2^t consecutive bytes starting at location $k\land(2^{64}-2^t)$. (The notation $k\land(2^{64}-2^t)$ means that the least significant t bits of k are set to 0, and only the least 64 bits of the resulting address are retained. Similarly, the notation $k\lor(2^t-1)$ means that the least significant t bits of k are set to 1.) All accesses to 2^t-byte quantities by `MMIX` are *aligned*, in the sense that the first byte is a multiple of 2^t.

Addressing is always "big-endian." In other words, the most significant (leftmost) byte of $\mathrm{M}_{2^t}[k]$ is $\mathrm{M}_1[k\land(2^{64}-2^t)]$ and the least significant (rightmost) byte is $\mathrm{M}_1[k\lor(2^t-1)]$. We use the notation $\mathrm{s}(\mathrm{M}_{2^t}[k])$ when we want to regard this 2^t-byte number as a *signed* integer. Formally speaking, if $l=2^t$,

$$\mathrm{s}(\mathrm{M}_l[k]) = \bigl(\mathrm{M}_1[k\land(-l)]\,\mathrm{M}_1[k\land(-l)+1]\ \ldots\ \mathrm{M}_1[k\lor(l-1)]\bigr)_{256} - 2^{8l}[\mathrm{M}_1[k\land(-l)]\ge 128].$$

7. Loading and storing. Several instructions can be used to get information from memory into registers. For example, the "load tetra unsigned" instruction `LDTU $1,$4,$5` puts the four bytes $M_4[\$4+\$5]$ into register 1 as an unsigned integer; the most significant four bytes of register 1 are set to zero. The similar instruction `LDT $1,$4,$5`, "load tetra," sets \$1 to the *signed* integer $s(M_4[\$4+\$5])$. (Instructions generally treat numbers as signed unless the operation code specifically calls them unsigned.) In the signed case, the most significant four bytes of the register will be copies of the most significant bit of the tetrabyte loaded; thus they will be all 0s or all 1s, depending on whether the number is ≥ 0 or < 0.

• `LDB $X,$Y,$Z|Z` 'load byte'.
Byte $s(M[\$Y+\$Z])$ or $s(M[\$Y+Z])$ is loaded into register X as a signed number between -128 and $+127$, inclusive.

• `LDBU $X,$Y,$Z|Z` 'load byte unsigned'. Byte $M[\$Y+\$Z]$ or $M[\$Y+Z]$ is loaded into register X as an unsigned number between 0 and 255, inclusive.

• `LDW $X,$Y,$Z|Z` 'load wyde'.
Bytes $s(M_2[\$Y+\$Z])$ or $s(M_2[\$Y+Z])$ are loaded into register X as a signed number between -32768 and $+32767$, inclusive. As mentioned above, our notation $M_2[k]$ implies that the least significant bit of the address $\$Y+\$Z$ or $\$Y+Z$ is ignored and assumed to be 0.

• `LDWU $X,$Y,$Z|Z` 'load wyde unsigned'. Bytes $M_2[\$Y+\$Z]$ or $M_2[\$Y+Z]$ are loaded into register X as an unsigned number between 0 and 65535, inclusive.

• `LDT $X,$Y,$Z|Z` 'load tetra'.
Bytes $s(M_4[\$Y+\$Z])$ or $s(M_4[\$Y+Z])$ are loaded into register X as a signed number between $-2{,}147{,}483{,}648$ and $+2{,}147{,}483{,}647$, inclusive. As mentioned above, our notation $M_4[k]$ implies that the two least significant bits of the address $\$Y+\$Z$ or $\$Y+Z$ are ignored and assumed to be 0.

• `LDTU $X,$Y,$Z|Z` 'load tetra unsigned'.
Bytes $M_4[\$Y+\$Z]$ or $M_4[\$Y+Z]$ are loaded into register X as an unsigned number between 0 and 4,294,967,296, inclusive.

• `LDO $X,$Y,$Z|Z` 'load octa'.
Bytes $M_8[\$Y+\$Z]$ or $M_8[\$Y+Z]$ are loaded into register X. As mentioned above, our notation $M_8[k]$ implies that the three least significant bits of the address $\$Y+\$Z$ or $\$Y+Z$ are ignored and assumed to be 0.

• `LDOU $X,$Y,$Z|Z` 'load octa unsigned'.
Bytes $M_8[\$Y+\$Z]$ or $M_8[\$Y+Z]$ are loaded into register X. There is in fact no difference between the behavior of `LDOU` and `LDO`, since an octabyte can be regarded as either signed or unsigned. `LDOU` is included in `MMIX` just for completeness and consistency, in spite of the fact that a foolish consistency is the hobgoblin of little minds. (Niklaus Wirth made a strong plea for such consistency in his early critique of System/360; see *JACM* **15** (1967), 37–74.)

• `LDHT $X,$Y,$Z|Z` 'load high tetra'.
Bytes $M_4[\$Y+\$Z]$ or $M_4[\$Y+Z]$ are loaded into the most significant half of register X, and the least significant half is cleared to zero. (One use of "high tetra arithmetic" is to detect overflow easily when tetrabytes are added or subtracted.)

• `LDA $X,$Y,$Z|Z` 'load address'.
The address $\$Y+\$Z$ or $\$Y+Z$ is loaded into register X. This instruction is simply another name for the `ADDU` instruction discussed below; it can be used when the programmer is thinking of memory addresses instead of numbers. The `MMIX` assembler converts `LDA` into the same OP-code as `ADDU`.

8. Another family of instructions goes the other way, storing registers into memory. For example, the "store octa immediate" command `STO $3,$2,17` puts the current contents of register 3 into $M_8[\$2+17]$.

• `STB $X,$Y,$Z|Z` 'store byte'.
The least significant byte of register X is stored into byte M[\$Y + \$Z] or M[\$Y + Z]. An integer overflow exception occurs if \$X is not between −128 and +127. (We will discuss overflow and other kinds of exceptions later.)

• `STBU $X,$Y,$Z|Z` 'store byte unsigned'.
The least significant byte of register X is stored into byte M[\$Y + \$Z] or M[\$Y + Z]. `STBU` instructions are the same as `STB` instructions, except that no test for overflow is made.

• `STW $X,$Y,$Z|Z` 'store wyde'.
The two least significant bytes of register X are stored into bytes $M_2[\$Y+\$Z]$ or $M_2[\$Y+Z]$. An integer overflow exception occurs if \$X is not between −32768 and +32767.

• `STWU $X,$Y,$Z|Z` 'store wyde unsigned'.
The two least significant bytes of register X are stored into bytes $M_2[\$Y+\$Z]$ or $M_2[\$Y+Z]$. `STWU` instructions are the same as `STW` instructions, except that no test for overflow is made.

• `STT $X,$Y,$Z|Z` 'store tetra'.
The four least significant bytes of register X are stored into bytes $M_4[\$Y+\$Z]$ or $M_4[\$Y+Z]$. An integer overflow exception occurs if \$X is not between −2,147,483,648 and +2,147,483,647.

• `STTU $X,$Y,$Z|Z` 'store tetra unsigned'.
The four least significant bytes of register X are stored into bytes $M_4[\$Y+\$Z]$ or $M_4[\$Y+Z]$. `STTU` instructions are the same as `STT` instructions, except that no test for overflow is made.

• `STO $X,$Y,$Z|Z` 'store octa'.
Register X is stored into bytes $M_8[\$Y+\$Z]$ or $M_8[\$Y+Z]$.

• `STOU $X,$Y,$Z|Z` 'store octa unsigned'.
Identical to `STO $X,$Y,$Z|Z`.

• `STCO X,$Y,$Z|Z` 'store constant octabyte'.
An octabyte whose value is the unsigned byte X is stored into $M_8[\$Y+\$Z]$ or $M_8[\$Y+Z]$.

• `STHT $X,$Y,$Z|Z` 'store high tetra'.
The most significant four bytes of register X are stored into $M_4[\$Y+\$Z]$ or $M_4[\$Y+Z]$.

9. Adding and subtracting. Once numbers are in registers, we can compute with them. Let's consider addition and subtraction first.

• `ADD $X,$Y,$Z|Z` 'add'.
The sum $\$Y + \$Z$ or $\$Y + Z$ is placed into register X using signed, two's complement arithmetic. An integer overflow exception occurs if the sum is $\geq 2^{63}$ or $< -2^{63}$. (We will discuss overflow and other kinds of exceptions later.)

• `ADDU $X,$Y,$Z|Z` 'add unsigned'.
The sum $(\$Y + \$Z) \bmod 2^{64}$ or $(\$Y + Z) \bmod 2^{64}$ is placed into register X. These instructions are the same as `ADD $X,$Y,$Z|Z` commands except that no test for overflow is made. (Overflow could be detected if desired by using the command `CMPU ovflo,$X,$Y` after addition, where `CMPU` means "compare unsigned"; see below.)

• `2ADDU $X,$Y,$Z|Z` 'times 2 and add unsigned'.
The sum $(2\$Y + \$Z) \bmod 2^{64}$ or $(2\$Y + Z) \bmod 2^{64}$ is placed into register X.

• `4ADDU $X,$Y,$Z|Z` 'times 4 and add unsigned'.
The sum $(4\$Y + \$Z) \bmod 2^{64}$ or $(4\$Y + Z) \bmod 2^{64}$ is placed into register X.

• `8ADDU $X,$Y,$Z|Z` 'times 8 and add unsigned'.
The sum $(8\$Y + \$Z) \bmod 2^{64}$ or $(8\$Y + Z) \bmod 2^{64}$ is placed into register X.

• `16ADDU $X,$Y,$Z|Z` 'times 16 and add unsigned'.
The sum $(16\$Y + \$Z) \bmod 2^{64}$ or $(16\$Y + Z) \bmod 2^{64}$ is placed into register X.

• `SUB $X,$Y,$Z|Z` 'subtract'.
The difference $\$Y - \$Z$ or $\$Y - Z$ is placed into register X using signed, two's complement arithmetic. An integer overflow exception occurs if the difference is $\geq 2^{63}$ or $< -2^{63}$.

• `SUBU $X,$Y,$Z|Z` 'subtract unsigned'.
The difference $(\$Y - \$Z) \bmod 2^{64}$ or $(\$Y - Z) \bmod 2^{64}$ is placed into register X. These two instructions are the same as `SUB $X,$Y,$Z|Z` except that no test for overflow is made.

• `NEG $X,Y,$Z|Z` 'negate'.
The value $Y - \$Z$ or $Y - Z$ is placed into register X using signed, two's complement arithmetic. An integer overflow exception occurs if the result is greater than $2^{63} - 1$. (Notice that in this case `MMIX` works with the "immediate" constant Y, not register Y. `NEG` commands are analogous to the immediate variants of other commands, because they save us from having to put one-byte constants into a register. When $Y = 0$, overflow occurs if and only if $\$Z = -2^{63}$. The instruction `NEG $X,1,2` has exactly the same effect as `NEG $X,0,1`.)

• `NEGU $X,Y,$Z|Z` 'negate unsigned'.
The value $(Y - \$Z) \bmod 2^{64}$ or $(Y - Z) \bmod 2^{64}$ is placed into register X. `NEGU` instructions are the same as `NEG` instructions, except that no test for overflow is made.

10. Bit fiddling. Before looking at multiplication and division, which take longer than addition and subtraction, let's look at some of the other things that `MMIX` can do fast. There are eighteen instructions for bitwise logical operations on unsigned numbers.

• `AND $X,$Y,$Z|Z` 'bitwise and'.
Each bit of register Y is logically anded with the corresponding bit of register Z or of the constant Z, and the result is placed in register X. In other words, a bit of register X is set to 1 if and only if the corresponding bits of the operands are both 1; in symbols, $\$\mathrm{X} = \$\mathrm{Y} \land \$\mathrm{Z}$ or $\$\mathrm{X} = \$\mathrm{Y} \land \mathrm{Z}$. This means in particular that `AND $X,$Y,Z` always zeroes out the seven most significant bytes of register X, because 0s are prefixed to the constant byte Z.

• `OR $X,$Y,$Z|Z` 'bitwise or'.
Each bit of register Y is logically ored with the corresponding bit of register Z or of the constant Z, and the result is placed in register X. In other words, a bit of register X is set to 0 if and only if the corresponding bits of the operands are both 0; in symbols, $\$\mathrm{X} = \$\mathrm{Y} \lor \$\mathrm{Z}$ or $\$\mathrm{X} = \$\mathrm{Y} \lor \mathrm{Z}$.

In the special case $\mathrm{Z} = 0$, the immediate variant of this command simply copies register Y to register X. The `MMIX` assembler allows us to write '`SET $X,$Y`' as a convenient abbreviation for '`OR $X,$Y,0`'.

• `XOR $X,$Y,$Z|Z` 'bitwise exclusive-or'.
Each bit of register Y is logically xored with the corresponding bit of register Z or of the constant Z, and the result is placed in register X. In other words, a bit of register X is set to 0 if and only if the corresponding bits of the operands are equal; in symbols, $\$\mathrm{X} = \$\mathrm{Y} \oplus \$\mathrm{Z}$ or $\$\mathrm{X} = \$\mathrm{Y} \oplus \mathrm{Z}$.

• `ANDN $X,$Y,$Z|Z` 'bitwise and-not'.
Each bit of register Y is logically anded with the complement of the corresponding bit of register Z or of the constant Z, and the result is placed in register X. In other words, a bit of register X is set to 1 if and only if the corresponding bit of register Y is 1 and the other corresponding bit is 0; in symbols, $\$\mathrm{X} = \$\mathrm{Y} \setminus \$\mathrm{Z}$ or $\$\mathrm{X} = \$\mathrm{Y} \setminus \mathrm{Z}$. (This is the *logical difference* operation; if the operands are bit strings representing sets, we are computing the elements that lie in one set but not the other.)

• `ORN $X,$Y,$Z|Z` 'bitwise or-not'.
Each bit of register Y is logically ored with the complement of the corresponding bit of register Z or of the constant Z, and the result is placed in register X. In other words, a bit of register X is set to 1 if and only if the corresponding bit of register Y is greater than or equal to the other corresponding bit; in symbols, $\$\mathrm{X} = \$\mathrm{Y} \lor \overline{\$\mathrm{Z}}$ or $\$\mathrm{X} = \$\mathrm{Y} \lor \overline{\mathrm{Z}}$. (This is the complement of $\$\mathrm{Z} \setminus \$\mathrm{Y}$ or $\mathrm{Z} \setminus \$\mathrm{Y}$.)

• `NAND $X,$Y,$Z|Z` 'bitwise not-and'.
Each bit of register Y is logically anded with the corresponding bit of register Z or of the constant Z, and the complement of the result is placed in register X. In other words, a bit of register X is set to 0 if and only if the corresponding bits of the operands are both 1; in symbols, $\$\mathrm{X} = \$\mathrm{Y} \mathbin{\overline{\land}} \$\mathrm{Z}$ or $\$\mathrm{X} = \$\mathrm{Y} \mathbin{\overline{\land}} \mathrm{Z}$.

• `NOR $X,$Y,$Z|Z` 'bitwise not-or'.
Each bit of register Y is logically ored with the corresponding bit of register Z or of the constant Z, and the complement of the result is placed in register X. In other words, a bit of register X is set to 1 if and only if the corresponding bits of the operands are both 0; in symbols, $\$\mathrm{X} = \$\mathrm{Y} \mathbin{\overline{\lor}} \$\mathrm{Z}$ or $\$\mathrm{X} = \$\mathrm{Y} \mathbin{\overline{\lor}} \mathrm{Z}$.

• `NXOR $X,$Y,$Z|Z` 'bitwise not-exclusive-or'.
Each bit of register Y is logically xored with the corresponding bit of register Z or of the constant Z, and the complement of the result is placed in register X. In other words, a bit of register X is set to 1 if and only if the corresponding bits of the operands are equal; in symbols, $\$\mathrm{X} = \$\mathrm{Y} \mathbin{\overline{\oplus}} \$\mathrm{Z}$ or $\$\mathrm{X} = \$\mathrm{Y} \mathbin{\overline{\oplus}} \mathrm{Z}$.

• `MUX $X,$Y,$Z|Z` 'bitwise multiplex'.
For each bit position j, the jth bit of register X is set either to bit j of register Y or to bit j of the other operand \$Z or Z, depending on whether bit j of the special *mask register* rM is 1 or 0: if M_j then Y_j else Z_j. In symbols, $\$\mathrm{X} = (\$\mathrm{Y} \land \mathrm{rM}) \lor (\$\mathrm{Z} \land \overline{\mathrm{rM}})$ or $\$\mathrm{X} = (\$\mathrm{Y} \land \mathrm{rM}) \lor (\mathrm{Z} \land \overline{\mathrm{rM}})$. (`MMIX` has several such special registers, associated with instructions that need more than two inputs or produce more than one output.)

11. Besides the eighteen bitwise operations, `MMIX` can also perform unsigned bytewise and biggerwise operations that are somewhat more exotic.

- `BDIF $X,$Y,$Z|Z` 'byte difference'.

For each byte position j, the jth byte of register X is set to byte j of register Y minus byte j of the other operand \$Z or Z, unless that difference is negative; in the latter case, byte j of \$X is set to zero.

- `WDIF $X,$Y,$Z|Z` 'wyde difference'.

For each wyde position j, the jth wyde of register X is set to wyde j of register Y minus wyde j of the other operand \$Z or Z, unless that difference is negative; in the latter case, wyde j of \$X is set to zero.

- `TDIF $X,$Y,$Z|Z` 'tetra difference'.

For each tetra position j, the jth tetra of register X is set to tetra j of register Y minus tetra j of the other operand \$Z or Z, unless that difference is negative; in the latter case, tetra j of \$X is set to zero.

- `ODIF $X,$Y,$Z|Z` 'octa difference'.

Register X is set to register Y minus the other operand \$Z or Z, unless \$Z or Z exceeds register Y; in the latter case, \$X is set to zero. The operands are treated as unsigned integers.

The `BDIF` and `WDIF` commands are useful in applications to graphics or video; `TDIF` and `ODIF` are also present for reasons of consistency. For example, if `a` and `b` are registers containing 8-byte quantities, their bytewise maxima `c` and bytewise minima `d` are computed by

```
BDIF x,a,b; ADDU c,x,b; SUBU d,a,x;
```

similarly, the individual "pixel differences" `e`, namely the absolute values of the differences of corresponding bytes, are computed by

```
BDIF x,a,b; BDIF y,b,a; OR e,x,y.
```

To add individual bytes of `a` and `b` while clipping all sums to 255 if they don't fit in a single byte, one can say

```
NOR acomp,a,0; BDIF x,acomp,b; NOR clippedsums,x,0;
```

in other words, complement `a`, apply `BDIF`, and complement the result. The operations can also be used to construct efficient operations on strings of bytes or wydes.

Exercise: Implement a "nybble difference" instruction that operates in a similar way on sixteen nybbles at a time.

Answer: `AND x,a,m; AND y,b,m; ANDN xx,a,m; ANDN yy,b,m; BDIF x,x,y; BDIF xx,xx,yy; OR ans,x,xx` where register `m` contains the mask `#0f0f0f0f0f0f0f0f`.

(The `ANDN` operation can be regarded as a "bit difference" instruction that operates in a similar way on 64 bits at a time.)

12. Three more pairs of bit-fiddling instructions round out the collection of exotics.

• `SADD $X,$Y,$Z|Z` 'sideways add'.
Each bit of register Y is logically anded with the complement of the corresponding bit of register Z or of the constant Z, and the number of 1 bits in the result is placed in register X. In other words, register X is set to the number of bit positions in which register Y has a 1 and the other operand has a 0; in symbols, $\$X = \nu(\$Y \setminus \$Z)$ or $\$X = \nu(\$Y \setminus Z)$. When the second operand is zero this operation is sometimes called "population counting," because it counts the number of 1s in register Y.

• `MOR $X,$Y,$Z|Z` 'multiple or'.
Suppose the 64 bits of register Y are indexed as

$$y_{00}y_{01}\ldots y_{07}y_{10}y_{11}\ldots y_{17}\ldots y_{70}y_{71}\ldots y_{77};$$

in other words, y_{ij} is the jth bit of the ith byte, if we number the bits and bytes from 0 to 7 in big-endian fashion from left to right. Let the bits of the other operand, \$Z or Z, be indexed similarly:

$$z_{00}z_{01}\ldots z_{07}z_{10}z_{11}\ldots z_{17}\ldots z_{70}z_{71}\ldots z_{77}.$$

The `MOR` operation replaces each bit x_{ij} of register X by the bit

$$y_{0j}z_{i0} \vee y_{1j}z_{i1} \vee \cdots \vee y_{7j}z_{i7}.$$

Thus, for example, if register Z contains the constant `#0102040810204080`, `MOR` reverses the order of the bytes in register Y, converting between little-endian and big-endian addressing. (The ith byte of \$X depends on the bytes of \$Y as specified by the ith byte of \$Z or Z. If we regard 64-bit words as 8×8 Boolean matrices, with one byte per column, this operation computes the Boolean product $\$X = \$Y\,\$Z$ or $\$X = \$Y\,Z$. Alternatively, if we regard 64-bit words as 8×8 matrices with one byte per *row*, `MOR` computes the Boolean product $\$X = \$Z\,\$Y$ or $\$X = Z\,\$Y$ with operands in the opposite order. The immediate form `MOR $X,$Y,Z` always sets the leading seven bytes of register X to zero; the other byte is set to the bitwise or of whatever bytes of register Y are specified by the immediate operand Z.)

Exercise: Explain how to compute a mask `m` that is `#ff` in byte positions where `a` exceeds `b`, `#00` in all other bytes. Answer: `BDIF x,a,b; MOR m,minusone,x;` here `minusone` is a register consisting of all 1s. (Moreover, if we `AND` this result with `#8040201008040201`, then `MOR` with $Z = 255$, we get a one-byte encoding of `m`.)

• `MXOR $X,$Y,$Z|Z` 'multiple exclusive-or'.
This operation is like the Boolean multiplication just discussed, but exclusive-or is used to combine the bits. Thus we obtain a matrix product over the field of two elements instead of a Boolean matrix product. This operation can be used to construct hash functions, among many other things. (The hash functions aren't bad, but they are not "universal" in the sense of exercise 6.4–72.)

13. Sixteen "immediate wyde" instructions are available for the common case that a 16-bit constant is needed. In this case the Y and Z fields of the instruction are regarded as a single 16-bit unsigned number YZ.

• `SETH $X,YZ` 'set to high wyde'; `SETMH $X,YZ` 'set to medium high wyde'; `SETML $X,YZ` 'set to medium low wyde'; `SETL $X,YZ` 'set to low wyde'.

The 16-bit unsigned number YZ is shifted left by either 48 or 32 or 16 or 0 bits, respectively, and placed into register X. Thus, for example, `SETML` inserts a given value into the second-least-significant wyde of register X and sets the other three wydes to zero.

• `INCH $X,YZ` 'increase by high wyde'; `INCMH $X,YZ` 'increase by medium high wyde'; `INCML $X,YZ` 'increase by medium low wyde'; `INCL $X,YZ` 'increase by low wyde'.

The 16-bit unsigned number YZ is shifted left by either 48 or 32 or 16 or 0 bits, respectively, and added to register X, ignoring overflow; the result is placed back into register X.

If YZ is the hexadecimal constant #8000, the command `INCH $X,YZ` complements the most significant bit of register X. We will see below that this can be used to negate a floating point number.

• `ORH $X,YZ` 'bitwise or with high wyde'; `ORMH $X,YZ` 'bitwise or with medium high wyde'; `ORML $X,YZ` 'bitwise or with medium low wyde'; `ORL $X,YZ` 'bitwise or with low wyde'.

The 16-bit unsigned number YZ is shifted left by either 48 or 32 or 16 or 0 bits, respectively, and ored with register X; the result is placed back into register X.

Notice that any desired 4-wyde constant `GH IJ KL MN` can be inserted into a register with a sequence of four instructions such as

```
SETH $X,GH; INCMH $X,IJ; INCML $X,KL; INCL $X,MN;
```

any of these `INC` instructions could also be replaced by `OR`.

• `ANDNH $X,YZ` 'bitwise and-not high wyde'; `ANDNMH $X,YZ` 'bitwise and-not medium high wyde'; `ANDNML $X,YZ` 'bitwise and-not medium low wyde'; `ANDNL $X,YZ` 'bitwise and-not low wyde'.

The 16-bit unsigned number YZ is shifted left by either 48 or 32 or 16 or 0 bits, respectively, then complemented and anded with register X; the result is placed back into register X.

If YZ is the hexadecimal constant #8000, the command `ANDNH $X,YZ` forces the most significant bit of register X to be 0. This can be used to compute the absolute value of a floating point number.

14. `MMIX` knows several ways to shift a register left or right by any number of bits.

• `SL $X,$Y,$Z|Z` 'shift left'.

The bits of register Y are shifted left by \$Z or Z places, and 0s are shifted in from the right; the result is placed in register X. Register Y is treated as a signed number, but the second operand is treated as an unsigned number. The effect is the same as multiplication by $2^{\$Z}$ or by 2^{Z}; an integer overflow exception occurs if the result is $\geq 2^{63}$ or $< -2^{63}$. In particular, if the second operand is 64 or more, register X will become entirely zero, and integer overflow will be signaled unless register Y was zero.

• `SLU $X,$Y,$Z|Z` 'shift left unsigned'.

The bits of register Y are shifted left by \$Z or Z places, and 0s are shifted in from the right; the result is placed in register X. Both operands are treated as unsigned numbers. The `SLU` instructions are equivalent to `SL`, except that no test for overflow is made.

• `SR $X,$Y,$Z|Z` 'shift right'.

The bits of register Y are shifted right by \$Z or Z places, and copies of the leftmost bit (the sign bit) are shifted in from the left; the result is placed in register X. Register Y is treated as a signed number, but the second operand is treated as an unsigned number. The effect is the same as division by $2^{\$Z}$ or by 2^{Z} and rounding down. In particular, if the second operand is 64 or more, register X will become zero if \$Y was nonnegative, -1 if \$Y was negative.

• `SRU $X,$Y,$Z|Z` 'shift right unsigned'.

The bits of register Y are shifted right by \$Z or Z places, and 0s are shifted in from the left; the result is placed in register X. Both operands are treated as unsigned numbers. The effect is the same as unsigned division of a 64-bit number by $2^{\$Z}$ or by 2^{Z}; if the second operand is 64 or more, register X will become entirely zero.

15. Comparisons. Arithmetic and logical operations are nice, but computer programs also need to compare numbers and to change the course of a calculation depending on what they find. `MMIX` has four comparison instructions to facilitate such decision-making.

• `CMP $X,$Y,$Z|Z` 'compare'.
Register X is set to -1 if register Y is less than register Z or less than the unsigned immediate value Z, using the conventions of signed arithmetic; it is set to 0 if register Y is equal to register Z or equal to the unsigned immediate value Z; otherwise it is set to 1. In symbols, $\$X = [\$Y > \$Z] - [\$Y < \$Z]$ or $\$X = [\$Y > Z] - [\$Y < Z]$.

• `CMPU $X,$Y,$Z|Z` 'compare unsigned'.
Register X is set to -1 if register Y is less than register Z or less than the unsigned immediate value Z, using the conventions of unsigned arithmetic; it is set to 0 if register Y is equal to register Z or equal to the unsigned immediate value Z; otherwise it is set to 1. In symbols, $\$X = [\$Y > \$Z] - [\$Y < \$Z]$ or $\$X = [\$Y > Z] - [\$Y < Z]$.

16. There also are 32 conditional instructions, which choose quickly between two alternative courses of action.

• `CSN $X,$Y,$Z|Z` 'conditionally set if negative'.
If register Y is negative (namely if its most significant bit is 1), register X is set to the contents of register Z or to the unsigned immediate value Z. Otherwise nothing happens.

• `CSZ $X,$Y,$Z|Z` 'conditionally set if zero'.
• `CSP $X,$Y,$Z|Z` 'conditionally set if positive'.
• `CSOD $X,$Y,$Z|Z` 'conditionally set if odd'.
• `CSNN $X,$Y,$Z|Z` 'conditionally set if nonnegative'.
• `CSNZ $X,$Y,$Z|Z` 'conditionally set if nonzero'.
• `CSNP $X,$Y,$Z|Z` 'conditionally set if nonpositive'.
• `CSEV $X,$Y,$Z|Z` 'conditionally set if even'.

These instructions are entirely analogous to `CSN`, except that register X changes only if register Y is respectively zero, positive, odd, nonnegative, nonzero, nonpositive, or nonodd.

• `ZSN $X,$Y,$Z|Z` 'zero or set if negative'.
If register Y is negative (namely if its most significant bit is 1), register X is set to the contents of register Z or to the unsigned immediate value Z. Otherwise register X is set to zero.

• `ZSZ $X,$Y,$Z|Z` 'zero or set if zero'.
• `ZSP $X,$Y,$Z|Z` 'zero or set if positive'.
• `ZSOD $X,$Y,$Z|Z` 'zero or set if odd'.
• `ZSNN $X,$Y,$Z|Z` 'zero or set if nonnegative'.
• `ZSNZ $X,$Y,$Z|Z` 'zero or set if nonzero'.
• `ZSNP $X,$Y,$Z|Z` 'zero or set if nonpositive'.
• `ZSEV $X,$Y,$Z|Z` 'zero or set if even'.

These instructions are entirely analogous to `ZSN`, except that \$X is set to \$Z or Z if register Y is respectively zero, positive, odd, nonnegative, nonzero, nonpositive, or even; otherwise \$X is set to zero.

Notice that the two instructions `CMPU r,s,0` and `ZSNZ r,s,1` have the same effect. So do the two instructions `CSNP r,s,0` and `ZSP r,s,r`. So do `AND r,s,1` and `ZSOD r,s,1`.

17. Branches and jumps. `MMIX` ordinarily executes instructions in sequence, proceeding from an instruction in tetrabyte $M_4[\lambda]$ to the instruction in $M_4[\lambda+4]$. But there are several ways to interrupt the normal flow of control, most of which use the Y and Z fields of an instruction as a combined 16-bit YZ field. For example, `BNZ $3,@+4000` (branch if nonzero) is typical: It means that control should skip ahead 1000 instructions to the command that appears 4000 bytes after the `BNZ`, if register 3 is not equal to zero.

There are eight branch-forward instructions, corresponding to the eight conditions in the `CS` and `ZS` commands that we discussed earlier. And there are eight similar branch-backward instructions; for example, `BOD $2,@-4000` (branch if odd) takes control to the instruction that appears 4000 bytes *before* this `BOD` command, if register 2 is odd. The numeric OP-code when branching backward is one greater than the OP-code when branching forward; the assembler takes care of this automatically, just as it takes cares of changing `ADD` from 32 to 33 when necessary.

Since branches are relative to the current location, the `MMIX` assembler treats branch instructions in a special way. Suppose a programmer writes '`BNZ $3,Case5`', where `Case5` is the address of an instruction in location l. If this instruction appears in location λ, the assembler first computes the displacement $\delta=\lfloor(l-\lambda)/4\rfloor$. Then if δ is nonnegative, the quantity δ is placed in the YZ field of a `BNZ` command, and it should be less than 2^{16}; if δ is negative, the quantity $2^{16}+\delta$ is placed in the YZ field of a `BNZ` command with OP-code increased by 1, and δ should not be less than -2^{16}.

The symbol `@` used in our examples of `BNZ` and `BOD` above is interpreted by the assembler as an abbreviation for "the location of the current instruction." In the following notes we will define pairs of branch commands by writing, for example, '`BNZ $X,@+4*YZ[-262144]`'; this stands for a branch-forward command that branches to the current location plus four times YZ, as well as for a branch-backward command that branches to the current location plus four times (YZ − 65536).

- `BN $X,@+4*YZ[-262144]` 'branch if negative'.
- `BZ $X,@+4*YZ[-262144]` 'branch if zero'.
- `BP $X,@+4*YZ[-262144]` 'branch if positive'.
- `BOD $X,@+4*YZ[-262144]` 'branch if odd'.
- `BNN $X,@+4*YZ[-262144]` 'branch if nonnegative'.
- `BNZ $X,@+4*YZ[-262144]` 'branch if nonzero'.
- `BNP $X,@+4*YZ[-262144]` 'branch if nonpositive'.
- `BEV $X,@+4*YZ[-262144]` 'branch if even'.

If register X is respectively negative, zero, positive, odd, nonnegative, nonzero, nonpositive, or even, and if this instruction appears in memory location λ, the next instruction is taken from memory location $\lambda+4\mathrm{YZ}$ (branching forward) or $\lambda+4(\mathrm{YZ}-2^{16})$ (branching backward). Thus one can go from location λ to any location between $\lambda-262{,}144$ and $\lambda+262{,}140$, inclusive.

Sixteen additional branch instructions called *probable branches* are also provided. They have exactly the same meaning as ordinary branch instructions; for example, `PBOD $2,@-4000` and `BOD $2,@-4000` both go backward 4000 bytes if register 2 is odd. But they differ in running time: On some implementations of `MMIX`, a branch instruction takes longer when the branch is taken, while a probable branch takes longer when the branch is *not* taken. Thus programmers should use a `B` instruction when they think branching is relatively unlikely, but they should use `PB` when they expect branching to occur more often than not. Here is a list of the probable branch commands, for completeness:

- `PBN $X,@+4*YZ[-262144]` 'probable branch if negative'.
- `PBZ $X,@+4*YZ[-262144]` 'probable branch if zero'.
- `PBP $X,@+4*YZ[-262144]` 'probable branch if positive'.
- `PBOD $X,@+4*YZ[-262144]` 'probable branch if odd'.
- `PBNN $X,@+4*YZ[-262144]` 'probable branch if nonnegative'.
- `PBNZ $X,@+4*YZ[-262144]` 'probable branch if nonzero'.
- `PBNP $X,@+4*YZ[-262144]` 'probable branch if nonpositive'.
- `PBEV $X,@+4*YZ[-262144]` 'probable branch if even'.

18. Locations that are relative to the current instruction can be transformed into absolute locations with `GETA` commands.

- `GETA $X,@+4*YZ[-262144]` 'get address'.

The value $\lambda + 4\mathrm{YZ}$ or $\lambda + 4(\mathrm{YZ} - 2^{16})$ is placed in register X. (The assembly language conventions of branch instructions apply; for example, we can write '`GETA $X,Addr`'.)

19. `MMIX` also has unconditional jump instructions, which change the location of the next instruction no matter what.

- `JMP @+XYZ[-67108864]` 'jump'.

A `JMP` command treats bytes X, Y, and Z as an unsigned 24-bit integer XYZ. It allows a program to transfer control from location λ to any location between $\lambda - 67{,}108{,}864$ and $\lambda + 67{,}108{,}860$ inclusive, using relative addressing as in the `B` and `PB` commands.

- `GO $X,$Y,$Z|Z` 'go to location'.

`MMIX` takes its next instruction from location $\$\mathrm{Y} + \$\mathrm{Z}$ or $\$\mathrm{Y} + \mathrm{Z}$, and continues from there. Register X is set equal to $\lambda + 4$, the location of the instruction that would ordinarily have been executed next. (`GO` is similar to a jump, but it is not relative to the current location. Since `GO` has the same format as a load or store instruction, a loading routine can treat program labels with the same mechanism that is used to treat references to data.)

An old-fashioned type of subroutine linkage can be implemented by saying either '`GO r,subloc,0`' or '`GETA r,@+8; JMP Sub`' to enter a subroutine, then '`GO r,r,0`' to return. But subroutines are normally entered with the instructions `PUSHJ` or `PUSHGO`, described below.

The two least significant bits of the address in a `GO` command are essentially ignored. They will, however, appear in the the value of λ returned by `GETA` instructions, and in the return-jump register rJ after `PUSHJ` or `PUSHGO` instructions are performed, and in the where-interrupted register at the time of an interrupt. Therefore they could be used to send some kind of signal to a subroutine or (less likely) to an interrupt handler.

20. Multiplication and division. Now for some instructions that make `MMIX` work harder.

• `MUL $X,$Y,$Z|Z` 'multiply'.
The signed product of the number in register Y by either the number in register Z or the unsigned byte Z replaces the contents of register X. An integer overflow exception can occur, as with `ADD` or `SUB`, if the result is less than -2^{63} or greater than $2^{63}-1$. (Immediate multiplication by powers of 2 can be done more rapidly with the `SL` instruction.)

• `MULU $X,$Y,$Z|Z` 'multiply unsigned'.
The lower 64 bits of the unsigned 128-bit product of register Y and either register Z or Z are placed in register X, and the upper 64 bits are placed in the special *himult register* rH. (Immediate multiplication by powers of 2 can be done more rapidly with the `SLU` instruction, if the upper half is not needed. Furthermore, an instruction like `4ADDU $X,$Y,$Y` is faster than `MULU $X,$Y,5`.)

• `DIV $X,$Y,$Z|Z` 'divide'.
The signed quotient of the number in register Y divided by either the number in register Z or the unsigned byte Z replaces the contents of register X, and the signed remainder is placed in the special *remainder register* rR. An integer divide check exception occurs if the divisor is zero; in that case $X is set to zero and rR is set to $Y. An integer overflow exception occurs if the number -2^{63} is divided by -1; otherwise integer overflow is impossible. The quotient of y divided by z is defined to be $\lfloor y/z \rfloor$, and the remainder is defined to be $y - \lfloor y/z \rfloor z$ (also written $y \bmod z$). Thus, the remainder is either zero or has the sign of the divisor. Dividing by $z = 2^t$ gives exactly the same quotient as shifting right t via the `SR` command, and exactly the same remainder as anding with $z-1$ via the `AND` command. Division of a positive 63-bit number by a positive constant can be accomplished more quickly by computing the upper half of a suitable unsigned product and shifting it right appropriately.

• `DIVU $X,$Y,$Z|Z` 'divide unsigned'.
The unsigned 128-bit number obtained by prefixing the special *dividend register* rD to the contents of register Y is divided either by the unsigned number in register Z or by the unsigned byte Z, and the quotient is placed in register X. The remainder is placed in the remainder register rR. However, if rD is greater than or equal to the divisor (and in particular if the divisor is zero), then $X is set to rD and rR is set to $Y. (Unsigned arithmetic never signals an exceptional condition, even when dividing by zero.) If rD is zero, unsigned division by $z = 2^t$ gives exactly the same quotient as shifting right t via the `SRU` command, and exactly the same remainder as anding with $z-1$ via the `AND` command. Section 4.3.1 of *Seminumerical Algorithms* explains how to use unsigned division to obtain the quotient and remainder of extremely large numbers.

21. Floating point computations. Floating point arithmetic conforming to the famous IEEE/ANSI Standard 754 is provided for arbitrary 64-bit numbers. The IEEE standard refers to such numbers as "double format" quantities, but `MMIX` calls them simply floating point numbers because 64-bit quantities are the norm.

A positive floating point number has 53 bits of precision and can range from approximately 10^{-308} to 10^{308}. "Denormal numbers" between 10^{-324} and 10^{-308} can also be represented, but with fewer bits of precision. Floating point numbers can be infinite, and they satisfy such identities as $1.0/\infty = +0.0$, $-2.8 \times \infty = -\infty$. Floating point quantities can also be "Not-a-Numbers" or NaNs, which are further classified into signaling NaNs and quiet NaNs.

Five kinds of exceptions can occur during floating point computations, and they each have code letters: Floating overflow (O) or underflow (U); floating divide by zero (Z); floating inexact (X); and floating invalid (I). For example, the multiplication of sufficiently small integers causes no exceptions, and the division of 91.0 by 13.0 is also exception-free, but the division 1.0/3.0 is inexact. The multiplication of extremely large or extremely small floating point numbers is inexact and it also causes overflow or underflow. Invalid results occur when taking the square root of a negative number; mathematicians can remember the I exception by relating it to the square root of -1.0. Invalid results also occur when trying to convert infinity or a quiet NaN to a fixed-point integer, or when any signaling NaN is encountered, or when mathematically undefined operations like $\infty - \infty$ or $0/0$ are requested. (Programmers can be sure that they have not erroneously used uninitialized floating point data if they initialize all their variables to signaling NaN values.)

Four different rounding modes for inexact results are available: round to nearest (and to even in case of ties); round off (toward zero); round up (toward $+\infty$); or round down (toward $-\infty$). `MMIX` has a special *arithmetic status register* rA that specifies the current rounding mode and the user's current preferences for exception handling.

IEEE standard arithmetic provides an excellent foundation for scientific calculations, and it will be thoroughly explained in the fourth edition of *Seminumerical Algorithms*, Section 4.2. For our present purposes, we need not study all the details; but we do need to specify `MMIX`'s behavior with respect to several things that are not completely defined by the standard. For example, the IEEE standard does not fully define the result of operations with NaNs.

When an octabyte represents a floating point number in `MMIX`'s registers, the leftmost bit is the sign; then come 11 bits for an exponent e; and the remaining 52 bits are the fraction part f. We regard e as an integer between 0 and $(11111111111)_2 = 2047$, and we regard f as a fraction between 0 and $(.111\ldots1)_2 = 1 - 2^{-52}$. Each octabyte has the following significance:

$$\begin{aligned} \pm 0.0, &\quad \text{if } e = f = 0 \text{ (zero);} \\ \pm 2^{-1022} f, &\quad \text{if } e = 0 \text{ and } f > 0 \text{ (denormal);} \\ \pm 2^{e-1023}(1+f), &\quad \text{if } 0 < e < 2047 \text{ (normal);} \\ \pm\infty, &\quad \text{if } e = 2047 \text{ and } f = 0 \text{ (infinite);} \\ \pm\mathrm{NaN}(f), &\quad \text{if } e = 2047 \text{ and } 0 < f < 1/2 \text{ (signaling NaN);} \\ \pm\mathrm{NaN}(f), &\quad \text{if } e = 2047 \text{ and } f \ge 1/2 \text{ (quiet NaN).} \end{aligned}$$

Notice that $+0.0$ is distinguished from -0.0; this fact is important for interval arithmetic.

Exercise: What 64 bits represent the floating point number 1.0? Answer: We want $e = 1023$ and $f = 0$, so the answer is #3ff0000000000000.

Exercise: What is the largest finite floating point number? Answer: We want $e = 2046$ and $f = 1 - 2^{-52}$, so the answer is `#7fefffffffffffff` $= 2^{1024} - 2^{971}$.

22. The seven IEEE floating point arithmetic operations (addition, subtraction, multiplication, division, remainder, square root, and nearest-integer) all share common features, called the *standard floating point conventions* in the discussion below: The operation is performed on floating point numbers found in two registers, $Y and $Z, except that square root and integerization involve only one operand. If neither input operand is a NaN, we first determine the exact result, then round it using the current rounding mode found in special register rA. Infinite results are exact and need no rounding. A floating overflow exception occurs if the rounded result is finite but needs an exponent greater than 2046. A floating underflow exception occurs if the rounded result needs an exponent less than 1 and either (i) the unrounded result cannot be represented exactly as a denormal number or (ii) the "floating underflow trip" is enabled in rA. (Trips are discussed below.) NaNs are treated specially as follows: If either $Y or $Z is a signaling NaN, an invalid exception occurs and the NaN is quieted by adding 1/2 to its fraction part. Then if $Z is a quiet NaN, the result is set to $Z; otherwise if $Y is a quiet NaN, the result is set to $Y.

• `FADD $X,$Y,$Z` 'floating add'.
The floating point sum $Y + $Z is computed by the standard floating point conventions just described, and placed in register X. An invalid exception occurs if the sum is $(+\infty)+(-\infty)$ or $(-\infty)+(+\infty)$; in that case the result is NaN(1/2) with the sign of $Z. If the sum is exactly zero and the current mode is not rounding-down, the result is $+0.0$ except that $(-0.0)+(-0.0)=-0.0$. If the sum is exactly zero and the current mode is rounding-down, the result is -0.0 except that $(+0.0)+(+0.0)=+0.0$. These rules for signed zeros turn out to be useful when doing interval arithmetic: If the lower bound of an interval is $+0.0$ or if the upper bound is -0.0, the interval does not contain zero, so the numbers in the interval have a known sign.

Floating point underflow cannot occur unless the U-trip has been enabled, because any underflowing result of floating point addition can be represented exactly as a denormal number.

Silly but instructive exercise: Find all pairs of numbers ($Y, $Z) such that the commands `FADD $X,$Y,$Z` and `ADDU $X,$Y,$Z` both produce the same result in $X (although `FADD` may cause floating exceptions). Answer: Of course $Y or $Z could be zero, if the other one is not a signaling NaN. Or one could be signaling and the other #0008000000000000. Other possibilities occur when they are both positive and less than #0010000000000001; or when one operand is #0000000000000001 and the other is an odd number between #0020000000000001 and #002ffffffffffffd inclusive (rounding to nearest). And still more surprising possibilities exist, such as #7f6001b4c67bc809 + #ff5ffb6a4534a3f7. All eight families of solutions will be revealed some day in the fourth edition of *Seminumerical Algorithms*.

• `FSUB $X,$Y,$Z` 'floating subtract'.
This instruction is equivalent to `FADD`, but with the sign of $Z negated unless $Z is a NaN.

• `FMUL $X,$Y,$Z` 'floating multiply'.
The floating point product $Y × $Z is computed by the standard floating point conventions, and placed in register X. An invalid exception occurs if the product is $(\pm 0.0)\times(\pm\infty)$ or $(\pm\infty)\times(\pm 0.0)$; in that case the result is ±NaN(1/2). No exception occurs for the product $(\pm\infty)\times(\pm\infty)$. If neither $Y nor $Z is a NaN, the sign of the result is the product of the signs of $Y and $Z.

• `FDIV $X,$Y,$Z` 'floating divide'.
The floating point quotient $Y/$Z is computed by the standard floating point conventions, and placed in $X. A floating divide by zero exception occurs if the quotient is (normal or denormal)/(± 0.0). An invalid exception occurs if the quotient is $(\pm 0.0)/(\pm 0.0)$ or $(\pm\infty)/(\pm\infty)$; in that case the result is ±NaN(1/2). No exception occurs for the quotient $(\pm\infty)/(\pm 0.0)$. If neither $Y nor $Z is a NaN, the sign of the result is the product of the signs of $Y and $Z.

If a floating point number in register X is known to have an exponent between 2 and 2046, the instruction `INCH $X,#fff0` will divide it by 2.0.

• `FREM $X,$Y,$Z` 'floating remainder'.
The floating point remainder $Y rem $Z is computed by the standard floating point conventions, and placed in register X. (The IEEE standard defines the remainder to be $Y − n × $Z, where n is the nearest integer to $Y/$Z, and n is an even integer in case of ties. This is not the same as the remainder $Y mod $Z computed by `DIV` or `DIVU`.) A zero remainder has the sign of $Y. An invalid exception occurs if $Y is infinite and/or $Z is zero; in that case the result is NaN(1/2) with the sign of $Y.

• `FSQRT $X,$Z` 'floating square root'.
The floating point square root $\sqrt{\$Z}$ is computed by the standard floating point conventions, and placed in register X. An invalid exception occurs if \$Z is a negative number (either infinite, normal, or denormal); in that case the result is $-\mathrm{NaN}(1/2)$. No exception occurs when taking the square root of -0.0 or $+\infty$. In all cases the sign of the result is the sign of \$Z.

• `FINT $X,$Z` 'floating integer'.
The floating point number in register Z is rounded (if necessary) to a floating point integer, using the current rounding mode, and placed in register X. Infinite values and quiet NaNs are not changed; signaling NaNs are treated as in the standard conventions. Floating point overflow and underflow exceptions cannot occur.

The Y field of `FSQRT` and `FINT` can be used to specify a special rounding mode, as explained below.

23. Besides doing arithmetic, we need to compare floating point numbers with each other, taking proper account of NaNs and the fact that -0.0 should be considered equal to $+0.0$. The following instructions are analogous to the comparison operators `CMP` and `CMPU` that we have used for integers.

• `FCMP $X,$Y,$Z` 'floating compare'.
Register X is set to -1 if $\$Y < \$Z$ according to the conventions of floating point arithmetic, or to 1 if $\$Y > \$Z$ according to those conventions. Otherwise it is set to 0. An invalid exception occurs if either \$Y or \$Z is a NaN; in such cases the result is zero.

• `FEQL $X,$Y,$Z` 'floating equal to'.
Register X is set to 1 if $\$Y = \$Z$ according to the conventions of floating point arithmetic. Otherwise it is set to 0. The result is zero if either \$Y or \$Z is a NaN, even if a NaN is being compared with itself. However, no invalid exception occurs, not even when \$Y or \$Z is a signaling NaN. (Perhaps `MMIX` differs slightly from the IEEE standard in this regard, but programmers sometimes need to look at signaling NaNs without encountering side effects. Programmers who insist on raising an invalid exception whenever a signaling NaN is compared for floating equality should issue the instructions `FSUB $X,$Y,$Y; FSUB $X,$Z,$Z` just before saying `FEQL $X,$Y,$Z`.)

Suppose w, x, y, and z are unsigned 64-bit integers with $w < x < 2^{63} \le y < z$. Thus, the leftmost bits of w and x are 0, while the leftmost bits of y and z are 1. Then we have $w < x < y < z$ when these numbers are considered as unsigned integers, but $y < z < w < x$ when they are considered as signed integers, because y and z are negative. Furthermore, we have $z < y \le w < x$ when these same 64-bit quantities are considered to be floating point numbers, assuming that no NaNs are present, because the leftmost bit of a floating point number represents its sign and the remaining bits represent its magnitude. The case $y = w$ occurs in floating point comparison if and only if y is the representation of -0.0 and w is the representation of $+0.0$.

• `FUN $X,$Y,$Z` 'floating unordered'.
Register X is set to 1 if \$Y and \$Z are unordered according to the conventions of floating point arithmetic (namely, if either one is a NaN); otherwise register X is set to 0. No invalid exception occurs, not even when \$Y or \$Z is a signaling NaN.

The IEEE standard discusses 26 different possible relations on floating point numbers; `MMIX` implements 14 of them with single instructions, followed by a branch (or by a `ZS` to make a "pure" 0 or 1 result); all 26 can be evaluated with a sequence of at most four `MMIX` commands and a subsequent branch. The hardest case to handle is '?>=' (unordered or greater or equal, to be computed without exceptions), for which the following sequence makes $\$X \ge 0$ if and only if \$Y ?>= \$Z:

```
   FUN   $255,$Y,$Z
   BP    $255,1F     % skip ahead if unordered
   FCMP  $X,$Y,$Z    % $X=[$Y>$Z]-[$Y<$Z]; no exceptions will arise
1H CSNZ  $X,$255,1   % $X=1 if unordered
```

24. Exercise: Suppose MMIX had no FINT instruction. Explain how to obtain the equivalent of FINT $X,$Z using other instructions. Your program should do the proper thing with respect to NaNs and exceptions. (For example, it should cause an invalid exception if and only if $Z is a signaling NaN; it should cause an inexact exception only if $Z needs to be rounded to another value.)

Answer: (The assembler prefixes hexadecimal constants by #.)

```
    SETH   $0,#4330 % $0=2^53
    SET    $1,$Z    % $1=$Z
    ANDNH  $1,#8000 % $1=abs($Z)
    ANDN   $2,$Z,$1 % $2=signbit($Z)
    FUN    $3,$Z,$Z % $3=[$Z is a NaN]
    BNZ    $3,1F    % skip ahead if $Z is a NaN
    FCMP   $3,$1,$0 % $3=[abs($Z)>2^53]-[abs($Z)<2^53]
    CSNN   $0,$3,0  % set $0=0 if $3>=0
    OR     $0,$2,$0 % attach sign of $Z to $0
1H  FADD   $1,$Z,$0 % $1=$Z+$0
    FSUB   $X,$1,$0 % $X=$1-$0
```

This program handles most cases of interest by adding and subtracting 2^{53} using floating point arithmetic. It would be incorrect to do this in all cases; for example, such addition/subtraction might fail to give the correct answer when $Z is a small negative quantity (if rounding toward zero), or when $Z is a number like $2^{106}+2^{54}$ (if rounding to nearest).

25. `MMIX` goes beyond the IEEE standard to define additional relations between floating point numbers, as suggested by the theory in Section 4.2.2 of *Seminumerical Algorithms*. Given a nonnegative number ϵ, each normal floating point number $u = (f, e)$ has a *neighborhood*

$$N_\epsilon(u) = \{x \mid |x - u| \le 2^{e-1022}\epsilon\};$$

we also define $N_\epsilon(0) = \{0\}$, $N_\epsilon(u) = \{x \mid |x - u| \le 2^{-1021}\epsilon\}$ if u is denormal; $N_\epsilon(\pm\infty) = \{\pm\infty\}$ if $\epsilon < 1$, $N_\epsilon(\pm\infty) = \{\text{everything except } \mp\infty\}$ if $1 \le \epsilon < 2$, $N_\epsilon(\pm\infty) = \{\text{everything}\}$ if $\epsilon \ge 2$. Then we write

$$\begin{aligned}
&u \prec v\ (\epsilon), \text{ if } u < N_\epsilon(v) \text{ and } N_\epsilon(u) < v;\\
&u \sim v\ (\epsilon), \text{ if } u \in N_\epsilon(v) \text{ or } v \in N_\epsilon(u);\\
&u \approx v\ (\epsilon), \text{ if } u \in N_\epsilon(v) \text{ and } v \in N_\epsilon(u);\\
&u \succ v\ (\epsilon), \text{ if } u > N_\epsilon(v) \text{ and } N_\epsilon(u) > v.
\end{aligned}$$

• `FCMPE $X,$Y,$Z` 'floating compare (with respect to epsilon)'.
Register X is set to -1 if \$Y $\prec$ \$Z (rE) according to the conventions of *Seminumerical Algorithms* as stated above; it is set to 1 if \$Y $\succ$ \$Z (rE) according to those conventions; otherwise it is set to 0. Here rE is a floating point number in the special *epsilon register*, which is used only by the floating point comparison operations `FCMPE`, `FEQLE`, and `FUNE`. An invalid exception occurs, and the result is zero, if any of \$Y, \$Z, or rE are NaN, or if rE is negative. If no such exception occurs, exactly one of the three conditions \$Y $\prec$ \$Z, \$Y $\sim$ \$Z, \$Y $\succ$ \$Z holds with respect to rE.

• `FEQLE $X,$Y,$Z` 'floating equivalent (with respect to epsilon)'.
Register X is set to 1 if \$Y $\approx$ \$Z (rE) according to the conventions of *Seminumerical Algorithms* as stated above; otherwise it is set to 0. An invalid exception occurs, and the result is zero, if any of \$Y, \$Z, or rE are NaN, or if rE is negative. Notice that the relation \$Y $\approx$ \$Z computed by `FEQLE` is stronger than the relation \$Y $\sim$ \$Z computed by `FCMPE`.

• `FUNE $X,$Y,$Z` 'floating unordered (with respect to epsilon)'.
Register X is set to 1 if \$Y, \$Z, or rE are exceptional as discussed for `FCMPE` and `FEQLE`; otherwise it is set to 0. No exceptions occur, even if \$Y, \$Z, or rE is a signaling NaN.

Exercise: What floating point numbers does `FCMPE` regard as ~ 0.0 with respect to $\epsilon = 1/2$, when no exceptions arise? Answer: Zero, denormal numbers, and normal numbers with $f = 0$. (The numbers similar to zero with respect to ϵ are zero, denormal numbers with $f \le 2\epsilon$, normal numbers with $f \le 2\epsilon - 1$, and $\pm\infty$ if $\epsilon >= 1$.)

26. The IEEE standard also defines 32-bit floating point quantities, which it calls "single format" numbers. MMIX calls them *short floats*, and converts between 32-bit and 64-bit forms when such numbers are loaded from memory or stored into memory. A short float consists of a sign bit followed by an 8-bit exponent and a 23-bit fraction. After it has been loaded into one of MMIX's registers, its 52-bit fraction part will have 29 trailing zero bits, and its exponent e will be one of the 256 values 0, $(01110000001)_2 = 897$, $(01110000010)_2 = 898$, $\ldots$, $(10001111110)_2 = 1150$, or 2047, unless it was denormal; a denormal short float loads into a normal number with $874 \le e \le 896$.

• `LDSF $X,$Y,$Z|Z` 'load short float'.
Register X is set to the 64-bit floating point number corresponding to the 32-bit floating point number represented by $\mathrm{M}_4[\$\mathrm{Y} + \$\mathrm{Z}]$ or $\mathrm{M}_4[\$\mathrm{Y} + \mathrm{Z}]$. No arithmetic exceptions occur, not even if a signaling NaN is loaded.

• `STSF $X,$Y,$Z|Z` 'store short float'.
The value obtained by rounding register X to a 32-bit floating point number is placed in $\mathrm{M}_4[\$\mathrm{Y} + \$\mathrm{Z}]$ or $\mathrm{M}_4[\$\mathrm{Y} + \mathrm{Z}]$. Rounding is done with the current rounding mode, in a manner exactly analogous to the standard conventions for rounding 64-bit results, except that the precision and exponent range are limited. In particular, floating overflow, underflow, and inexact exceptions might occur; a signaling NaN will trigger an invalid exception and it will become quiet. The fraction part of a NaN is truncated if necessary to a multiple of 2^{-23}, by ignoring the least significant 29 bits.

If we load any two short floats and operate on them once with either `FADD`, `FSUB`, `FMUL`, `FDIV`, `FREM`, `FSQRT`, or `FINT`, and if we then store the result as a short float, we obtain the results required by the IEEE standard for single format arithmetic, because the double format can be shown to have enough precision to avoid any problems of "double rounding." But programmers are usually better off sticking to 64-bit arithmetic unless they have a strong reason to emulate the precise behavior of a 32-bit computer; 32 bits do not offer much precision.

27. Of course we need to be able to go back and forth between integers and floating point values.

• `FIX $X,$Z` 'convert floating to fixed'.
The floating point number in register Z is converted to an integer as with the `FINT` instruction, and the resulting integer (mod 2^{64}) is placed in register X. An invalid exception occurs if \$Z is infinite or a NaN; in that case \$X is simply set equal to \$Z. A float-to-fix exception occurs if the result is less than -2^{63} or greater than $2^{63} - 1$.

• `FIXU $X,$Z` 'convert floating to fixed unsigned'.
This instruction is identical to `FIX` except that no float-to-fix exception occurs.

• `FLOT $X,$Z|Z` 'convert fixed to floating'.
The integer in \$Z or the immediate constant Z is converted to the nearest floating point value (using the current rounding mode) and placed in register X. A floating inexact exception occurs if rounding is necessary.

• `FLOTU $X,$Z|Z` 'convert fixed to floating unsigned'.
`FLOTU` is like `FLOT`, but \$Z is treated as an unsigned integer.

• `SFLOT $X,$Z|Z` 'convert fixed to short float'; `SFLOTU $X,$Z|Z` 'convert fixed to short float unsigned'.
The `SFLOT` instructions are like the `FLOT` instructions, except that they round to a floating point number whose fraction part is a multiple of 2^{-23}. (Thus, the resulting value will not be changed by a "store short float" instruction.) Such conversions appear in MMIX's repertoire only to establish complete conformance with the IEEE standard; a programmer needs them only when emulating a 32-bit machine.

28. Since the variants of `FIX` and `FLOT` involve only one input operand (\$Z or Z), their Y field is normally zero. A programmer can, however, force the mode of rounding used with these commands by setting

$$\begin{aligned}
\mathrm{Y}&=1, &&\texttt{ROUND_OFF};\\
\mathrm{Y}&=2, &&\texttt{ROUND_UP};\\
\mathrm{Y}&=3, &&\texttt{ROUND_DOWN};\\
\mathrm{Y}&=4, &&\texttt{ROUND_NEAR};
\end{aligned}$$

for example, the instruction `FLOTU $X,ROUND_OFF,$Z` will set the exponent e of register X to $1086 - l$ if \$Z is a nonzero quantity with l leading zero bits. Thus we can count leading zeros by continuing with `SETL $0,1086`; `SR $X,$X,52`; `SUB $X,$0,$X`; `CSZ $X,$Z,64`.

The Y field can also be used in the same way to specify any desired rounding mode in the other floating point instructions that have only a single operand, namely `FSQRT` and `FINT`. An illegal instruction interrupt occurs if Y exceeds 4 in any of these commands.

29. Subroutine linkage. `MMIX` has a several special operations designed to facilitate the process of calling and implementing subroutines. The key notion is the idea of a hardware-supported *register stack*, which can coexist with a software-supported stack of variables that are not maintained in registers. From a programmer's standpoint, `MMIX` maintains a potentially unbounded list $S[0]$, $S[1]$, …, $S[\tau-1]$ of octabytes holding the contents of registers that are temporarily inaccessible; initially $\tau = 0$. When a subroutine is entered, registers can be "pushed" on to the end of this list, increasing τ; when the subroutine has finished its execution, the registers are "popped" off again and τ decreases.

Our discussion so far has treated all 256 registers \$0, \$1, …, \$255 as if they were alike. But in fact, `MMIX` maintains two internal one-byte counters L and G, where $0 \le L \le G < 256$, with the property that

registers 0, 1, …, $L-1$ are "local";
registers L, $L+1$, …, $G-1$ are "marginal";
registers G, $G+1$, …, 255 are "global."

A marginal register is zero when its value is read.

The G counter is normally set to a fixed value once and for all when a program is loaded, thereby defining the number of program variables that will live entirely in registers rather than in memory during the course of execution. A programmer may, however, change G dynamically using the `PUT` instruction described below.

The L counter starts at 0. If an instruction places a value into a register that is currently marginal, namely a register x such that $L \le x < G$, the value of L will increase to $x+1$, and any newly local registers will be zero. For example, if $L = 10$ and $G = 200$, the instruction `ADD $5,$15,1` would simply set \$5 to 1. But the instruction `ADD $15,$5,$200` would set \$10, \$11, …, \$14 to zero, \$15 to \$5 + \$200, and L to 16. (The process of clearing registers and increasing L might take quite a few machine cycles in the worst case. We will see later that `MMIX` is able to take care of any high-priority interrupts that might occur during this time.)

- `PUSHJ $X,@+4*YZ[-262144]` 'push registers and jump'.
- `PUSHGO $X,$Y,$Z|Z` 'push registers and go'.

Suppose first that $\mathrm{X} < L$. Register X is set equal to the number X, then registers 0, 1, …, X are pushed onto the register stack as described below. If this instruction is in location λ, the value $\lambda+4$ is placed into the special *return-jump register* rJ. Then control jumps to instruction $\lambda + 4\mathrm{YZ}$ or $\lambda + 4\mathrm{YZ} - 262144$ or $\$\mathrm{Y} + \$\mathrm{Z}$ or $\$\mathrm{Y} + \mathrm{Z}$, as in a `JMP` or `GO` command.

Pushing the first $\mathrm{X}+1$ registers onto the stack means essentially that we set $S[\tau] \leftarrow \$0$, $S[\tau+1] \leftarrow \$1$, …, $S[\tau+\mathrm{X}] \leftarrow \$\mathrm{X}$, $\tau \leftarrow \tau+\mathrm{X}+1$, $\$0 \leftarrow \$(\mathrm{X}+1)$, …, $\$(L-\mathrm{X}-2) \leftarrow \$(L-1)$, $L \leftarrow L-\mathrm{X}-1$. For example, if $\mathrm{X} = 1$ and $L = 5$, the current contents of \$0 and the number 1 are placed on the register stack, where they will be temporarily inaccessible. Then control jumps to a subroutine with L reduced to 3; the registers that we had been calling \$2, \$3, and \$4 appear as \$0, \$1, and \$2 to the subroutine.

If $\mathrm{X} \ge L$ the actions are similar, except that *all* of the local registers \$0, …, $\$(L-1)$ are placed on the register stack followed by the number L, and L is reset to zero. In particular, the instruction `PUSHGO $255,$Y,$Z` pushes all the local registers onto the stack and sets L to zero, regardless of the previous value of L.

We will see later that `MMIX` is able to achieve the effect of pushing and renaming local registers without actually doing very much work at all.

- `POP X,YZ` 'pop registers and return from subroutine'.

This command preserves X of the current local registers, undoes the effect of the most recent `PUSHJ` or `PUSHGO`, and jumps to the instruction in $\mathrm{M}_4[4\mathrm{YZ} + \mathrm{rJ}]$. If $\mathrm{X} > 0$, the value of $\$(\mathrm{X}-1)$ goes into the "hole" position where `PUSHJ` or `PUSHGO` stored the number of registers previously pushed.

The formal details of `POP` are slightly complicated, but we will see that they make sense: If $\mathrm{X} > L$, we first replace X by $L+1$. Then we set $x \leftarrow S[\tau-1] \bmod 256$; this is the effective value of the X field in the push instruction that is being undone. Stack position $S[\tau-1]$ is now set to $\$(\mathrm{X}-1)$ if $0 < \mathrm{X} \le L$, otherwise it is set to zero. Then we essentially set $L \leftarrow \min(x+\mathrm{X}, G)$, $\$(L-1) \leftarrow \$(L-x-2)$, …, $\$(x+1) \leftarrow \$0$, $\$x \leftarrow S[\tau-1]$, …, $\$0 \leftarrow S[\tau-x-1]$, $\tau \leftarrow \tau-x-1$. The operating system should arrange things so that a memory-protection interrupt will occur if a program does more pops than pushes. (If $x > G$, these formulas don't make sense as written; we actually set $\$j \leftarrow S[\tau-x-1+j]$ for $L > j \ge 0$ in that rare case.)

Suppose, for example, that a subroutine has three input parameters ($\$0, \$1, \$2$) and produces two outputs ($\$0, \$1$). If the subroutine does not call any other subroutines, it can simply end with `POP 2,0`, because rJ will contain the return address. Otherwise it should begin by saving rJ, for example with the instruction `GET $4,rJ` if it will be using local registers \$0 through \$3, and it should use `PUSHJ $5` or `PUSHGO $5` when calling sub-subroutines; finally it should `PUT rJ,$4` before saying `POP 2,0`. To call the subroutine from another routine that has, say, 6 local registers, we would put the input arguments into \$7, \$8, and \$9, then issue the command `PUSHGO $6,base,Subr`; in due time the outputs of the subroutine will appear in \$7 and \$6.

Notice that the push and pop commands make use of a one-place "hole" in the register stack, between the registers that are pushed down and the registers that remain local. (The hole is position \$6 in the example just considered.) `MMIX` needs this hole position to remember the number of registers that are pushed down. A subroutine with no outputs ends with `POP 0,0` and the hole disappears (becomes marginal). A subroutine with one output \$0 ends with `POP 1,0` and the hole gets the former value of \$0. A subroutine with two outputs ($\$0, \$1$) ends with `POP 2,0` and the hole gets the former value of \$1; in this case, therefore, the relative order of the two outputs has been switched on the register stack. If a subroutine has, say, five outputs ($\$0, \ldots, \$4$), it ends with `POP 5,0` and \$4 goes into the hole position, where it is followed by ($\$0, \$1, \$2, \$3$). `MMIX` makes this curious permutation in the case of multiple outputs because the hole is most easily plugged by moving one value down (namely \$4) instead of by sliding each of five values down in the stack.

These conventions for parameter passing are admittedly a bit confusing in the general case, and I suppose people who use them extensively might someday find themselves talking about "the infamous `MMIX` register shuffle." However, there is good use for subroutines that convert a sequence of register contents like (x, a, b, c) into (f, a, b, c) where f is a function of a, b, and c but not x. Moreover, `PUSHGO` and `POP` can be implemented with great efficiency, and subroutine linkage tends to be a significant bottleneck when other conventions are used.

Information about a subroutine's calling conventions needs to be communicated to a debugger. That can readily be done at the same time as we inform the debugger about the symbolic names of addresses in memory.

A subroutine that uses 50 local registers will not function properly if it is called by a program that sets G less than 50. `MMIX` does not allow the value of G to become less than 32. Therefore any subroutine that avoids global registers and uses at most 32 local registers can be sure to work properly regardless of the current value of G.

The rules stated above imply that a `PUSHJ` or `PUSHGO` instruction with $\mathrm{X} = 255$ pushes all of the currently defined local registers onto the stack and sets L to zero. This makes G local registers available for use by the subroutine jumped to. If that subroutine later returns with `POP 0,0`, the former value of L and the former contents of $\$0, \ldots, \$(L-1)$ will be restored.

A `POP` instruction with $\mathrm{X} = 255$ preserves all the local registers as outputs of the subroutine (provided that the total doesn't exceed G after popping), and puts zero into the hole (unless $L = G = 255$). The best policy, however, is almost always to use `POP` with a small value of X, and in general to keep the value of L as small as possible by decreasing it when registers are no longer active. A smaller value of L means that `MMIX` can change context more easily when switching from one process to another.

30. System considerations. High-performance implementations of `MMIX` gain speed by keeping *caches* of instructions and data that are likely to be needed as computation proceeds. [See M. V. Wilkes, *IEEE Transactions* **EC-14** (1965), 270–271; J. S. Liptay, *IBM System J.* **7** (1968), 15–21.] Careful programmers can make the computer run even faster by giving hints about how to maintain such caches.

• `LDUNC $X,$Y,$Z|Z` 'load octa uncached'.
These instructions, which have the same meaning as `LDO`, also inform the computer that the loaded octabyte (and its neighbors in a cache block) will probably not be read or written in the near future.

• `STUNC $X,$Y,$Z|Z` 'store octa uncached'.
These instructions, which have the same meaning as `STO`, also inform the computer that the stored octabyte (and its neighbors in a cache block) will probably not be read or written in the near future.

• `PRELD X,$Y,$Z|Z` 'preload data'.
These instructions have no effect on registers or memory, but they inform the computer that many of the $X + 1$ bytes $M[\$Y + \$Z]$ through $M[\$Y + \$Z + X]$, or $M[\$Y + Z]$ through $M[\$Y + Z + X]$, will probably be loaded and/or stored in the near future. No protection failure occurs if the memory is not accessible.

• `PREGO X,$Y,$Z|Z` 'prefetch to go'.
These instructions have no effect on registers or memory, but they inform the computer that many of the $X + 1$ bytes $M[\$Y + \$Z]$ through $M[\$Y + \$Z + X]$, or $M[\$Y + Z]$ through $M[\$Y + Z + X]$, will probably be used as instructions in the near future. No protection failure occurs if the memory is not accessible.

• `PREST X,$Y,$Z|Z` 'prestore data'.
These instructions have no effect on registers or memory if the computer has no data cache. But when such a cache exists, they inform the computer that all of the $X + 1$ bytes $M[\$Y + \$Z]$ through $M[\$Y + \$Z + X]$, or $M[\$Y + Z]$ through $M[\$Y + Z + X]$, will definitely be stored in the near future before they are loaded. (Therefore it is permissible for the machine to ignore the present contents of those bytes. Also, if those bytes are being shared by several processors, the current processor should try to acquire exclusive access.) No protection failure occurs if the memory is not accessible.

• `SYNCD X,$Y,$Z|Z` 'synchronize data'.
When executed from nonnegative locations, these instructions have no effect on registers or memory if neither a write buffer nor a "write back" data cache are present. But when such a buffer or cache exists, they force the computer to make sure that all data for the $X+1$ bytes $M[\$Y+\$Z]$ through $M[\$Y+\$Z+X]$, or $M[\$Y+Z]$ through $M[\$Y + Z + X]$, will be present in memory. (Otherwise the result of a previous store instruction might appear only in the cache; the computer is being told that now is the time to write the information back, if it hasn't already been written. A program can use this feature before outputting directly from memory.) No protection failure occurs if the memory is not accessible.

The action is similar when `SYNCD` is executed from a negative address, but in this case the specified bytes are also removed from the data cache (and from a secondary cache, if present). The operating system can use this feature when a page of virtual memory is being swapped out, or when data is input directly into memory.

• `SYNCID X,$Y,$Z|Z` 'synchronize instructions and data'.
When executed from nonnegative locations these instructions have no effect on registers or memory if the computer has no instruction cache separate from a data cache. But when such a cache exists, they force the computer to make sure that the $X + 1$ bytes $M[\$Y + \$Z]$ through $M[\$Y + \$Z + X]$, or $M[\$Y + Z]$ through $M[\$Y+Z+X]$, will be interpreted correctly if used as instructions before they are next modified. (Generally speaking, an `MMIX` program is not expected to store anything in memory locations that are also being used as instructions. Therefore `MMIX`'s instruction cache is allowed to become inconsistent with respect to its data cache. Programmers who insist on executing instructions that have been fabricated dynamically, for example when setting a breakpoint for debugging, must first `SYNCID` those instructions in order to guarantee that the intended results will be obtained.) A `SYNCID` command might be implemented in several ways; for example, the machine might update its instruction cache to agree with its data cache. A simpler solution, which is good enough because the need for `SYNCID` ought to be rare, removes instructions in the specified range from the instruction cache, if present, so that they will have to be fetched from memory the next time

they are needed; in this case the machine also carries out the effect of a `SYNCD` command. No protection failure occurs if the memory is not accessible.

The behavior is more drastic, but faster, when `SYNCID` is executed from a negative location. Then all bytes in the specified range are simply removed from all caches, and the memory corresponding to any "dirty" cache blocks involving such bytes is *not* brought up to date. An operating system can use this version of the command when pages of virtual memory are being discarded (for example, when a program is being terminated).

31. `MMIX` is designed to work not only on a single processor but also in situations where several processors share a common memory. The following commands are useful for efficient operation in such circumstances.

- `CSWAP $X,$Y,$Z|Z` 'compare and swap octabytes'.

If the octabyte $M_8[\$Y + \$Z]$ or $M_8[\$Y + Z]$ is equal to the contents of the special *prediction register* rP, it is replaced in memory with the contents of register X, and register X is set equal to 1. Otherwise the octabyte in memory replaces rP and register X is set to zero. This is an atomic (indivisible, uninterruptible) operation, useful for interprocess communication when independent computers are sharing the same memory.

The compare-and-swap operation was introduced by IBM in late models of the System/370 architecture, and it soon spread to several other machines. Significant ways to use it are discussed, for example, in section 7.2.3 of Harold Stone's *High-Performance Computer Architecture* (Reading, Massachusetts: Addison–Wesley, 1987), and in sections 8.2 and 8.3 of *Transaction Processing* by Jim Gray and Andreas Reuter (San Francisco: Morgan Kaufmann, 1993).

- `SYNC XYZ` 'synchronize'.

If $XYZ = 0$, the machine drains its pipeline (that is, it stalls until all preceding instructions have completed their activity). If $XYZ = 1$, the machine controls its actions less drastically, in such a way that all store instructions preceding this `SYNC` will be completed before all store instructions after it. If $XYZ = 2$, the machine controls its actions in such a way that all load instructions preceding this `SYNC` will be completed before all load instructions after it. If $XYZ = 3$, the machine controls its actions in such a way that all *load or store* instructions preceding this `SYNC` will be completed before all load or store instructions after it. If $XYZ = 4$, the machine goes into a power-saver mode, in which instructions may be executed more slowly (or not at all) until some kind of "wake-up" signal is received. If $XYZ = 5$, the machine empties its write buffer and cleans its data caches, if any (including a possible secondary cache); the caches retain their data, but the cache contents also appear in memory. If $XYZ = 6$, the machine clears its virtual address translation caches (see below). If $XYZ = 7$, the machine clears its instruction and data caches, discarding any information in the data caches that wasn't previously in memory. ("Clearing" is stronger than "cleaning"; a clear cache remembers nothing. Clearing is also faster, because it simply obliterates everything.) If $XYZ > 7$, an illegal instruction interrupt occurs.

Of course no `SYNC` is necessary between a command that loads from or stores into memory and a subsequent command that loads from or stores into exactly the same location. However, `SYNC` might be necessary in certain cases even on a one-processor system, because input/output processes take place in parallel with ordinary computation.

The cases $XYZ > 3$ are *privileged*, in the sense that only the operating system can use them. More precisely, if a `SYNC` command is encountered with $XYZ = 4$ or $XYZ = 5$ or $XYZ = 6$ or $XYZ = 7$, a "privileged instruction interrupt" occurs unless that interrupt is currently disabled. Only the operating system can disable interrupts (see below).

32. Trips and traps. Special register rA records the current status information about arithmetic exceptions. Its least significant byte contains eight "event" bits called DVWIOUZX from left to right, where D stands for integer divide check, V for integer overflow, W for float-to-fix overflow, I for invalid operation, O for floating overflow, U for floating underflow, Z for floating division by zero, and X for floating inexact. The next least significant byte of rA contains eight "enable" bits with the same names DVWIOUZX and the same meanings. When an exceptional condition occurs, there are two cases: If the corresponding enable bit is 0, the corresponding event bit is set to 1. But if the corresponding enable bit is 1, `MMIX` interrupts its current instruction stream and executes a special "exception handler." Thus, the event bits record exceptions that have not been "tripped."

Floating point overflow always causes two exceptions, O and X. (The strictest interpretation of the IEEE standard would raise exception X on overflow only if floating overflow is not enabled, but `MMIX` always considers an overflowed result to be inexact.) Floating point underflow always causes both U and X when underflow is not enabled, and it might cause both U and X when underflow is enabled. If both enable bits are set to 1 in such cases, the overflow or underflow handler is called and the inexact handler is ignored. All other types of exceptions arise one at a time, so there is no ambiguity about which exception handler should be invoked unless exceptions are raised by "ropcode 2" (see below); in general the first enabled exception in the list DVWIOUZX takes precedence.

What about the six high-order bytes of the status register rA? At present, only two of those 48 bits are defined; the others must be zero for compatibility with possible future extensions. The two bits corresponding to 2^{17} and 2^{16} in rA specify a rounding mode, as follows: 00 means round to nearest (the default); 01 means round off (toward zero); 10 means round up (toward positive infinity); and 11 means round down (toward negative infinity).

33. The execution of `MMIX` programs can be interrupted in several ways. We have just seen that arithmetic exceptions will cause interrupts if they are enabled; so will illegal or privileged instructions, or instructions that are emulated in software instead of provided by the hardware. Input/output operations or external timers are another common source of interrupts; the operating system knows how to deal with all gadgets that might be hooked up to an `MMIX` processor chip. Interrupts occur also when memory accesses fail—for example if memory is nonexistent or protected. Power failures that force the machine to use its backup battery power in order to keep running in an emergency, or hardware failures like parity errors, all must be handled as gracefully as possible.

Users can also force interrupts to happen by giving explicit `TRAP` or `TRIP` instructions:

- `TRAP X,Y,Z` 'trap'; `TRIP X,Y,Z` 'trip'.

Both of these instructions interrupt processing and transfer control to a handler. The difference between them is that `TRAP` is handled by the operating system but `TRIP` is handled by the user. More precisely, the X, Y, and Z fields of `TRAP` have special significance predefined by the operating system kernel. For example, a system call—say an I/O command, or a command to allocate more memory—might be invoked by certain settings of X, Y, and Z. The X, Y, and Z fields of `TRIP`, on the other hand, are definable by users for their own applications, and users also define their own handlers. "Trip handler" programs invoked by `TRIP` are interruptible, but interrupts are normally inhibited while a `TRAP` is being serviced. Specific details about the precise actions of `TRIP` and `TRAP` appear below, together with the description of another command called `RESUME` that returns control from a handler to the interrupted program.

Only two variants of `TRAP` are predefined by the `MMIX` architecture: If XYZ $= 0$ in a `TRAP` command, a user process should terminate. If XYZ $= 1$, the operating system should provide default action for cases in which the user has not provided any handler for a particular kind of interrupt (see below).

A few additional variants of `TRAP` are predefined in the rudimentary operating system used with `MMIX` simulators. These variants, which allow simple input/output operations to be done, all have X $= 0$, and the Y field is a small positive constant. For example, Y $= 1$ invokes the `Fopen` routine, which opens a file. (See the program MMIX-SIM for full details.)

34. Non-catastrophic interrupts in `MMIX` are always *precise*, in the sense that all legal instructions before a certain point have effectively been executed, and no instructions after that point have yet been executed. The current instruction, which may or may not have been completed at the time of interrupt and which may or may not need to be resumed after the interrupt has been serviced, is put into the special *execution register* rX, and its operands (if any) are placed in special registers rY and rZ. The address of the following instruction is placed in the special *where-interrupted register* rW. The instruction in rX may not be the same as the instruction in location rW − 4; for example, it may be an instruction that branched or jumped to rW. It might also be an instruction inserted internally by the `MMIX` processor. (For example, the computer silently inserts an internal instruction that increases L before an instruction like `ADD $9,$1,$0` if L is currently less than 10. If an interrupt occurs, between the inserted instruction and the `ADD`, the instruction in rX will say `ADD`, because an internal instruction retains the identity of the actual command that spawned it; but rW will point to the *real* `ADD` command.)

When an instruction has the normal meaning "set \$X to the result of \$Y op \$Z" or "set \$X to the result of \$Y op Z," special registers rY and rZ will relate in the obvious way to the Y and Z operands of the instruction; but this is not always the case. For example, after an interrupted store instruction, the first operand rY will hold the virtual memory address (\$Y plus either \$Z or Z), and the second operand rZ will be the octabyte to be stored in memory (including bytes that have not changed, in cases like `STB`). In other cases the actual contents of rY and rZ are defined by each implementation of `MMIX`, and programmers should not rely on their significance.

Some instructions take an unpredictable and possibly long amount of time, so it may be necessary to interrupt them in progress. For example, the `FREM` instruction (floating point remainder) is extremely difficult to compute rapidly if its first operand has an exponent of 2046 and its second operand has an exponent of 1. In such cases the rY and rZ registers saved during an interrupt show the current state of the computation, not necessarily the original values of the operands. The value of rY rem rZ will still be the desired remainder, but rY may well have been reduced to a number that has an exponent closer to the exponent of rZ. After the interrupt has been processed, the remainder computation will continue where it left off. (Alternatively, an operation like `FREM` or even `FADD` might be implemented in software instead of hardware, as we will see later.)

Another example arises with an instruction like `PREST` (prestore), which can specify prestoring up to 256 bytes. An implementation of `MMIX` might choose to prestore only 32 or 64 bytes at a time, depending on the cache block size; then it can change the contents of rX to reflect the unfinished part of a partially completed `PREST` command.

Commands that decrease G, pop the stack, save the current context, or unsave an old context also are interruptible. Register rX is used to communicate information about partial completion in such a way that the interruption will be essentially "invisible" after a program is resumed.

35. Three kinds of interruption are possible: trips, forced traps, and dynamic traps. We will discuss each of these in turn.

A `TRIP` instruction puts itself into the right half of the execution register rX, and sets the 32 bits of the left half to #`80000000`. (Therefore rX is *negative*; this fact will tell the `RESUME` command not to `TRIP` again.) The special registers rY and rZ are set to the contents of the registers specified by the Y and Z fields of the `TRIP` command, namely $Y and $Z. Then $255 is placed into the special *bootstrap register* rB, and $255 is set to zero. `MMIX` now takes its next instruction from virtual memory address 0.

Arithmetic exceptions interrupt the computation in essentially the same way as `TRIP`, if they are enabled. The only difference is that their handlers begin at the respective addresses 16, 32, 48, 64, 80, 96, 112, and 128, for exception bits D, V, W, I, O, U, Z, and X of rA; registers rY and rZ are set to the operands of the interrupted instruction as explained earlier.

A 16-byte block of memory is more than enough for a sequence of commands like

```
PUSHJ 255,Handler; GET $255,rB; RESUME
```

which will invoke a user's handler. And if the user does not choose to provide a custom-designed handler, the operating system provides a default handler via the instructions

```
TRAP 1; GET $255,rB; RESUME.
```

A trip handler might simply record the fact that tripping occurred. But the handler for an arithmetic interrupt might want to change the default result of a computation. In such cases, the handler should place the desired substitute result into rZ, and it should change the most significant byte of rX from #`80` to #`02`. This will have the desired effect, because of the rules of `RESUME` explained below, *unless* the exception occurred on a command like `STB` or `STSF`. (A bit more work is needed to alter the effect of a command that stores into memory.)

Instructions in *negative* virtual locations do not invoke trip handlers, either for `TRIP` or for arithmetic exceptions. Such instructions are reserved for the operating system, as we will see.

36. A `TRAP` instruction interrupts the computation essentially like `TRIP`, but with the following modifications: (i) $255 is set to the contents of the special "trap address register" rT, not zero; (ii) the interrupt mask register rK is cleared to zero, thereby inhibiting interrupts; (iii) control jumps to virtual memory address rT, not zero; (iv) information is placed in a separate set of special registers rBB, rWW, rXX, rYY, and rZZ, instead of rB, rW, rX, rY, and rZ. (These special registers are needed because a trap might occur while processing a `TRIP`.)

Another kind of forced trap occurs on implementations of `MMIX` that emulate certain instructions in software rather than in hardware. Such instructions cause a `TRAP` even though their opcode is something else like `FREM` or `FADD` or `DIV`. The trap handler can tell what instruction to emulate by looking at the opcode, which appears in rXX. In such cases the lefthand half of rXX is set to #`02000000`; the handler emulating `FADD`, say, should compute the floating point sum of rYY and rZZ and place the result in rZZ. A subsequent `RESUME 1` will then place the value of rZZ in the proper register.

Implementations of `MMIX` might also emulate the process of virtual-address-to-physical-address translation described below, instead of providing for page table calculations in hardware. Then if, say, a `LDB` instruction does not know the physical memory address corresponding to a specified virtual address, it will cause a forced trap with the left half of rXX set to #`03000000` and with rYY set to the virtual address in question. The trap handler should place the physical page address into rZZ; then `RESUME 1` will complete the `LDB`.

37. The third and final kind of interrupt is called a *dynamic* trap. Such interruptions occur when one or more of the 64 bits in the the special *interrupt request register* rQ have been set to 1, and when at least one corresponding bit of the special *interrupt mask register* rK is also equal to 1. The bit positions of rQ and rK have the general form

24	8	24	8
low-priority I/O	program	high-priority I/O	machine

where the 8-bit "program" bits are called `rwxnkbsp` and have the following meanings:

`r` bit: instruction tries to load from a page without read permission;
`w` bit: instruction tries to store to a page without write permission;
`x` bit: instruction appears in a page without execute permission;
`n` bit: instruction refers to a negative virtual address;
`k` bit: instruction is privileged, for use by the "kernel" only;
`b` bit: instruction breaks the rules of `MMIX`;
`s` bit: instruction violates security (see below);
`p` bit: instruction comes from a privileged (negative) virtual address.

Negative addresses are for the use of the operating system only; a security violation occurs if an instruction in a nonnegative address is executed without the `rwxnkbsp` bits of rK all set to 1. (In such cases the `s` bits of both rQ and rK are set to 1.)

The eight "machine" bits of rQ and rK represent the most urgent kinds of interrupts. The rightmost bit stands for power failure, the next for memory parity error, the next for nonexistent memory, the next for rebooting, etc. Interrupts that need especially quick service, like requests from a high-speed network, also are allocated bit positions near the right end. Low priority I/O devices like keyboards are assigned to bits at the left. The allocation of input/output devices to bit positions will differ from implementation to implementation, depending on what devices are available.

Once $\mathrm{rQ} \land \mathrm{rK}$ becomes nonzero, the machine waits briefly until it can give a precise interrupt. Then it proceeds as with a forced trap, except that it uses the special "dynamic trap address register" rTT instead of rT. The trap handler that begins at location rTT can figure out the reason for interrupt by examining $\mathrm{rQ} \land \mathrm{rK}$. (For example, after the instructions

```
GET $0,rQ; GET $1,rK; AND $0,$0,$1; SUBU $1,$0,1; XOR $2,$0,$1; ANDN $1,$0,$1; SADD $2,$2,0
```

the highest-priority offending bit will be in \$1 and its position will be in \$2.)

If the interrupted instruction contributed 1s to any of the `rwxnkbsp` bits of rQ, the corresponding bits are set to 1 also in rX. A dynamic trap handler might be able to use this information (although it should service higher-priority interrupts first if the right half of $\mathrm{rQ} \land \mathrm{rK}$ is nonzero).

The rules of `MMIX` are rigged so that only the operating system can execute instructions with interrupts suppressed. Therefore the operating system can in fact use instructions that would interrupt an ordinary program. Control of register rK turns out to be the ultimate privilege, and in a sense the only important one.

An instruction that causes a dynamic trap is usually executed before the interruption occurs. However, an instruction that traps with bits `x`, `k`, or `b` does nothing; a load instruction that traps with `r` or `n` loads zero; a store instruction that traps with any of `rwxnkbsp` stores nothing.

38. After a trip handler or trap handler has done its thing, it generally invokes the following command.

• `RESUME Z` 'resume after interrupt'; the X and Y fields must be zero.

If the Z field of this instruction is zero, `MMIX` will use the information found in special registers rW, rX, rY, and rZ to restart an interrupted computation. If the execution register rX is negative, it will be ignored and instructions will be executed starting at virtual address rW; otherwise the instruction in the right half of the execution register will be inserted into the program as if it had appeared in location rW − 4, subject to certain modifications that we will explain momentarily, and the *next* instruction will come from rW.

If the Z field of `RESUME` is 1 and if this instruction appears in a negative location, registers rWW, rXX, rYY, and rZZ are used instead of rW, rX, rY, and rZ. Also, just before resuming the computation, mask register rK is set to $255 and $255 is set to rBB. (Only the operating system gets to use this feature.)

An interrupt handler within the operating system might choose to allow itself to be interrupted. In such cases it should save the contents of rBB, rWW, rXX, rYY, and rZZ on some kind of stack, before making rK nonzero. Then, before resuming whatever caused the base level interrupt, it must again disable all interrupts; this can be done with `TRAP`, because the trap handler can tell from the virtual address in rWW that it has been invoked by the operating system. Once rK is again zero, the contents of rBB, rWW, rXX, rYY, and rZZ are restored from the stack, the outer level interrupt mask is placed in $255, and `RESUME 1` finishes the job.

Values of Z greater than 1 are reserved for possible later definition. Therefore they cause an illegal instruction interrupt (that is, they set the '`b`' bit of rQ) in the present version of `MMIX`.

If the execution register rX is nonnegative, its leftmost byte controls the way its righthand half will be inserted into the program. Let's call this byte the "ropcode." A ropcode of 0 simply inserts the instruction into the execution stream; a ropcode of 1 is similar, but it substitutes rY and rZ for the two operands, assuming that this makes sense for the operation considered.

Ropcode 2 inserts a command that sets $X to rZ, where X is the second byte in the right half of rX. This ropcode is normally used with forced-trap emulations, so that the result of an emulated instruction is placed into the correct register. It also uses the third-from-left byte of rX to raise any or all of the arithmetic exceptions DVWIOUZX, at the same time as rZ is being placed in $X. Emulated instructions and explicit `TRAP` commands can therefore cause overflow, say, just as ordinary instructions can. (Such new exceptions may, of course, spawn a trip interrupt, if any of the corresponding bits are enabled in rA.)

Finally, ropcode 3 is the same as ropcode 0, except that it also tells `MMIX` to treat rZ as the page table entry for the virtual address rY. (See the discussion of virtual address translation below.) Ropcodes greater than 3 are not permitted; moreover, only `RESUME 1` is allowed to use ropcode 3.

The ropcode rules in the previous paragraphs should of course be understood to involve rWW, rXX, rYY, and rZZ instead of rW, rX, rY, and rZ when the ropcode is seen by `RESUME 1`. Thus, in particular, ropcode 3 always applies to rYY and rZZ, never to rY and rZ.

Special restrictions must hold if resumption is to work properly: Ropcodes 0 and 3 must not insert a `RESUME` instruction; ropcode 1 must insert a "normal" instruction, namely one whose opcode begins with one of the hexadecimal digits `#0`, `#1`, `#2`, `#3`, `#6`, `#7`, `#C`, `#D`, or `#E`. (See the opcode chart below.) Some implementations may also allow ropcode 1 with `SYNCD[I]` and `SYNCID[I]`, so that those instructions can conveniently be interrupted. Moreover, the destination register $X used with ropcode 1 or 2 must not be marginal. All of these restrictions hold automatically in normal use; they are relevant only if the programmer tries to do something tricky.

Notice that the slightly tricky sequence

```
LDA $0,Loc; PUT rW,$0; LDT $1,Inst; PUT rX,$1; RESUME
```

will execute an almost arbitrary instruction `Inst` as if it had been in location `Loc-4`, and then will jump to location `Loc` (assuming that `Inst` doesn't branch elsewhere).

39. Special registers. Quite a few special registers have been mentioned so far, and MMIX actually has even more. It is time now to enumerate them all, together with their internal code numbers:

rA, arithmetic status register [21];
rB, bootstrap register (trip) [0];
rC, cycle counter [8];
rD, dividend register [1];
rE, epsilon register [2];
rF, failure location register [22];
rG, global threshold register [19];
rH, himult register [3];
rI, interval counter [12];
rJ, return-jump register [4];
rK, interrupt mask register [15];
rL, local threshold register [20];
rM, multiplex mask register [5];
rN, serial number [9];
rO, register stack offset [10];
rP, prediction register [23];
rQ, interrupt request register [16];
rR, remainder register [6];
rS, register stack pointer [11];
rT, trap address register [13];
rU, usage counter [17];
rV, virtual translation register [18];
rW, where-interrupted register (trip) [24];
rX, execution register (trip) [25];
rY, Y operand (trip) [26];
rZ, Z operand (trip) [27];
rBB, bootstrap register (trap) [7];
rTT, dynamic trap address register [14];
rWW, where-interrupted register (trap) [28];
rXX, execution register (trap) [29];
rYY, Y operand (trap) [30];
rZZ, Z operand (trap) [31];

In this list rG and rL are what we have been calling simply G and L; rC, rF, rI, rN, rO, rS, rU, and rV have not been mentioned before.

40. The *cycle counter* rC advances by 1 on every "clock pulse" of the `MMIX` processor. Thus if `MMIX` is running at 500 MHz, the cycle counter increases every 2 nanoseconds. There is no need to worry about rC overflowing; even if it were to increase once every nanosecond, it wouldn't reach 2^{64} until more than 584.55 years have gone by.

The *interval counter* rI is similar, but it *decreases* by 1 on each cycle, and causes an *interval interrupt* when it reaches zero. Such interrupts can be extremely useful for "continuous profiling" as a means of studying the empirical running time of programs; see Jennifer M. Anderson, Lance M. Berc, Jeffrey Dean, Sanjay Ghemawat, Monika R. Henzinger, Shun-Tak A. Leung, Richard L. Sites, Mark T. Vandevoorde, Carl A. Waldspurger, and William E. Weihl, *ACM Transactions on Computer Systems* **15** (1997), 357–390. The interval interrupt is achieved by setting the leftmost bit of the "machine" byte of rQ equal to 1; this is the eighth-least-significant bit.

The *usage counter* rU consists of three fields (u_p, u_m, u_c), called the usage pattern u_p, the usage mask u_m, and the usage count u_c. The most significant byte of rU is the usage pattern; the next most significant byte is the usage mask; and the remaining 48 bits are the usage count. Whenever an instruction whose $\mathrm{OP} \wedge u_m = u_p$ has been executed, the value of u_c increases by 1 (mod 2^{48}). Thus, for example, the OP-code chart below implies that all instructions are counted if $u_p = u_m = 0$; all loads and stores are counted together with `GO` and `PUSHGO` if $u_p = (10000000)_2$ and $u_m = (11000000)_2$; all floating point instructions are counted together with fixed point multiplications and divisions if $u_p = 0$ and $u_m = (11100000)_2$; fixed point multiplications and divisions alone are counted if $u_p = (00011000)_2$ and $u_m = (11111000)_2$; completed subroutine calls are counted if $u_p =$ `POP` and $u_m = (11111111)_2$. Instructions in negative locations, which belong to the operating system, are exceptional: They are included in the usage count only if the leading bit of u_c is 1.

Incidentally, the 64-bit counters rC and rI can be implemented rather cheaply with only two levels of logic, using an old trick called "carry-save addition" [see, for example, G. Metze and J. E. Robertson, *Proc. International Conf. Information Processing* (Paris: 1959), 389–396]. One nice embodiment of this idea is to represent a binary number x in a redundant form as the difference $x' - x''$ of two binary numbers. Any two such numbers can be added without carry propagation as follows: Let

$$f(x,y,z) = (x \wedge \bar{y}) \vee (x \wedge z) \vee (\bar{y} \wedge z), \qquad g(x,y,z) = x \oplus y \oplus z.$$

Then it is easy to check that $x - y + z = 2f(x,y,z) - g(x,y,z)$; we need only verify this in the eight cases when x, y, and z are 0 or 1. Thus we can subtract 1 from a counter $x' - x''$ by setting

$$(x', x'') \leftarrow (f(x', x'', -1) \ll 1,\ g(x', x'', -1));$$

we can add 1 by setting $(x', x'') \leftarrow (g(x'', x', -1), f(x'', x', -1) \ll 1)$. The result is zero if and only if $x' = x''$. We need not actually compute the difference $x' - x''$ until we need to examine the register. The computation of $f(x,y,z)$ and $g(x,y,z)$ is particularly simple in the special cases $z = 0$ and $z = -1$. A similar trick works for rU, but extra care is needed in that case because several instructions might finish at the same time. (Thanks to Frank Yellin for his improvements to this paragraph.)

41. The special *serial number register* rN is permanently set to the time this particular instance of `MMIX` was created (measured as the number of seconds since 00:00:00 Greenwich Mean Time on 1 January 1970), in its five least significant bytes. The three most significant bytes are permanently set to the *version number* of the `MMIX` architecture that is being implemented together with two additional bytes that modify the version number. This quantity serves as an essentially unique identification number for each copy of `MMIX`.

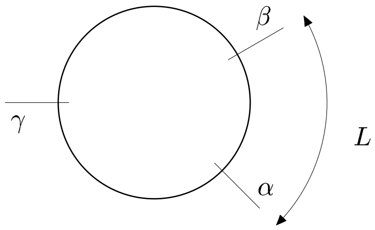
Version 1.0.0 of the architecture is described in the present document. Version 1.0.1 is similar, but simplified to avoid the complications of pipelines and operating systems. Other versions may become necessary in the future.

42. The *register stack offset* rO and *register stack pointer* rS are especially interesting, because they are used to implement MMIX's register stack $S[0]$, $S[1]$, $S[2]$, ...

The operating system initializes a register stack by assigning a large area of virtual memory to each running process, beginning at an address like #6000000000000000. If this starting address is σ, stack entry $S[k]$ will go into the octabyte $\mathrm{M}_8[\sigma+8k]$. Stack underflow will be detected because the process does not have permission to read from $\mathrm{M}[\sigma-1]$. Stack overflow will be detected because something will give out—either the user's budget or the user's patience or the user's swap space—long before 2^{61} bytes of virtual memory are filled by a register stack.

The MMIX hardware maintains the register stack by having two banks of 64-bit general-purpose registers, one for globals and one for locals. The global registers g[32], g[33], ..., g[255] are used for register numbers that are $\geq G$ in MMIX commands; recall that G is always 32 or more. The local registers come from another array that contains 2^n registers for some n where $8 \leq n \leq 10$; for simplicity of exposition we will assume that there are exactly 512 local registers, but there may be only 256 or there may be 1024.

The local register slots l[0], l[1], ..., l[511] act as a cyclic buffer with addresses that wrap around mod 512, so that l[512] = l[0], l[513] = l[1], etc. This buffer is divided into three parts by three pointers, which we will call α, β, and γ.



Registers $\mathrm{l}[\alpha]$, $\mathrm{l}[\alpha+1]$, ..., $\mathrm{l}[\beta-1]$ are what program instructions currently call \$0, \$1, ..., \$$(L-1)$; registers $\mathrm{l}[\beta]$, $\mathrm{l}[\beta+1]$, ..., $\mathrm{l}[\gamma-1]$ are currently unused; and registers $\mathrm{l}[\gamma]$, $\mathrm{l}[\gamma+1]$, ..., $\mathrm{l}[\alpha-1]$ contain items of the register stack that have been pushed down but not yet stored in memory. Special register rS holds the virtual memory address where $\mathrm{l}[\gamma]$ will be stored, if necessary. Special register rO holds the address where $\mathrm{l}[\alpha]$ will be stored; this always equals 8τ plus the address of $S[0]$. We can deduce the values of α, β, and γ from the contents of rL, rO, and rS, because

$$\alpha = (\mathrm{rO}/8) \bmod 512, \qquad \beta = (\alpha + \mathrm{rL}) \bmod 512, \qquad \text{and} \qquad \gamma = (\mathrm{rS}/8) \bmod 512.$$

To maintain this situation we need to make sure that the pointers α, β, and γ never move past each other. A PUSHJ or PUSHGO operation simply advances α toward β, so it is very simple. The first part of a POP operation, which moves β toward α, is also very simple. But the next part of a POP requires α to move downward, and memory accesses might be required. MMIX will decrease rS by 8 (thereby decreasing γ by 1) and set $\mathrm{l}[\gamma] \leftarrow \mathrm{M}_8[\mathrm{rS}]$, one or more times if necessary, to keep α from decreasing past γ. Similarly, the operation of increasing L may cause MMIX to set $\mathrm{M}_8[\mathrm{rS}] \leftarrow \mathrm{l}[\gamma]$ and increase rS by 8 (thereby increasing γ by 1) one or more times, to keep β from increasing past γ. If many registers need to be loaded or stored at once, these operations are interruptible.

[A somewhat similar scheme was introduced by David R. Ditzel and H. R. McLellan in *SIGPLAN Notices* **17**, 4 (April 1982), 48–56, and incorporated in the so-called CRISP architecture developed at AT&T Bell Labs. An even more similar scheme was adopted in the late 1980s by Advanced Micro Devices, in the processors of their Am29000 series—a family of computers whose instructions have essentially the format 'OP X Y Z' used by MMIX.]

Limited versions of MMIX, having fewer registers, can also be envisioned. For example, we might have only 32 local registers l[0], l[1], ..., l[31] and only 32 global registers g[224], g[225], ..., g[255]. Such a machine could run any MMIX program that maintains the inequalities $L < 32$ and $G \geq 224$.

43. Access to MMIX's special registers is obtained via the GET and PUT commands.

• `GET $X,Z` 'get from special register'; the Y field must be zero.
Register X is set to the contents of the special register identified by its code number Z, using the code numbers listed earlier. An illegal instruction interrupt occurs if $Z \geq 32$.

Every special register is readable; MMIX does not keep secrets from an inquisitive user. But of course only the operating system is allowed to change registers like rK and rQ (the interrupt mask and request registers). And not even the operating system is allowed to change rC (the cycle counter) or rN (the serial number) or the stack pointers rO and rS.

• `PUT X,$Z|Z` 'put into special register'; the Y field must be zero.
The special register identified by X is set to the contents of register Z or to the unsigned byte Z itself, if permissible. Some changes are, however, impermissible: Bits of rA that are always zero must remain zero; the leading seven bytes of rG and rL must remain zero, and rL must not exceed rG; special registers 8–11 (namely rC, rN, rO, and rS) must not change; special registers 12–18 (namely rI, rK, rQ, rT, rU, rV, and rTT) can be changed only if the privilege bit of rK is zero; and certain bits of rQ (depending on available hardware) might not allow software to change them from 0 to 1. Moreover, any bits of rQ that have changed from 0 to 1 since the most recent `GET x,rQ` will remain 1 after `PUT rQ,z`. The PUT command will not increase rL; it sets rL to the minimum of the current value and the new value. (A program should say `SETL $99,0` instead of `PUT rL,100` when rL is known to be less than 100.)

Impermissible PUT commands cause an illegal instruction interrupt, or (in the case of rI, rK, rQ, rT, rU, rV, and rTT) a privileged operation interrupt.

• `SAVE $X,0` 'save process state'; `UNSAVE 0,$Z` 'restore process state'; the Y field must be 0, and so must the Z field of SAVE, the X field of UNSAVE.
The SAVE instruction stores all registers and special registers that might affect the computation of the currently running process. First the current local registers \$0, \$1, ..., $\$(L-1)$ are pushed down as in `PUSHGO $255`, and L is set to zero. Then the current global registers $\$G$, $\$(G+1)$, ..., \$255 are placed above them in the register stack; finally rB, rD, rE, rH, rJ, rM, rR, rP, rW, rX, rY, and rZ are placed at the very top, followed by registers rG and rA packed into eight bytes:

8	24	32
rG	0	rA

The address of the topmost octabyte is then placed in register X, which must be a global register. (This instruction is interruptible. If an interrupt occurs while the registers are being saved, we will have $\alpha = \beta = \gamma$ in the ring of local registers; thus rO will equal rS and rL will be zero. The interrupt handler essentially has a new register stack, starting on top of the partially saved context.) Immediately after a SAVE the values of rO and rS are equal to the location of the first byte following the stack just saved. The current register stack is effectively empty at this point; thus one shouldn't do a POP until this context or some other context has been unsaved.

The UNSAVE instruction goes the other way, restoring all the registers when given an address in register Z that was returned by a previous SAVE. Immediately after an UNSAVE the values of rO and rS will be equal. Like SAVE, this instruction is interruptible.

The operating system uses SAVE and UNSAVE to switch context between different processes. It can also use UNSAVE to establish suitable initial values of rO and rS. But a user program that knows what it is doing can in fact allocate its own register stack or stacks and do its own process switching.

Caution: UNSAVE is destructive, in the sense that a program can't reliably UNSAVE twice from the same saved context. Once an UNSAVE has been done, further operations are likely to change the memory record of what was saved. Moreover, an interrupt during the middle of an UNSAVE may have already clobbered some of the data in memory before the UNSAVE has completely finished, although the data will appear properly in all registers.

44. Virtual and physical addresses. Virtual 64-bit addresses are converted to physical addresses in a manner governed by the special *virtual translation register* rV. Thus M[A] really refers to m[ϕ(A)], where m is the physical memory array and $\phi(A)$ is determined by the physical mapping function ϕ. The details of this conversion are rather technical and of interest mainly to the operating system, but two simple rules are important to ordinary users:

• Negative addresses are mapped directly to physical addresses, by simply suppressing the sign bit:

$$\phi(A) = A + 2^{63} = A \land \texttt{\#7fffffffffffffff}, \qquad \text{if } A < 0.$$

All accesses to negative addresses are privileged, for use by the operating system only. (Thus, for example, the trap addresses in rT and rTT should be negative, because they are addresses inside the operating system.) Moreover, all physical addresses $\geq 2^{48}$ are intended for use by memory-mapped I/O devices; values read from or written to such locations are never placed in a cache.

• Nonnegative addresses belong to four *segments*, depending on whether the three leading bits are 000, 001, 010, or 011. These 2^{61}-byte segments are traditionally used for a program's text, data, dynamic memory, and register stack, respectively, but such conventions are not mandatory. There are four mappings ϕ_0, ϕ_1, ϕ_2, and ϕ_3 of 61-bit addresses into 48-bit physical memory space, one for each segment:

$$\phi(A) = \phi_{\lfloor A/2^{61} \rfloor}(A \bmod 2^{61}), \qquad \text{if } 0 \leq A < 2^{63}.$$

In general, the machine is able to access smaller addresses of a segment more efficiently than larger addresses. Thus a programmer should let each segment grow upward from zero, trying to keep any of the 61-bit addresses from becoming larger than necessary, although arbitrary addresses are legal.

45. Now it's time for the technical details of virtual address translation. The mappings ϕ_0, ϕ_1, ϕ_2, and ϕ_3 are defined by the following rules.

(1) The first two bytes of rV are four nybbles called b_1, b_2, b_3, b_4; we also define $b_0 = 0$. Segment i has at most $1024^{\,b_{i+1}-b_i}$ pages. In particular, segment i must have at most one page when $b_i = b_{i+1}$, and it must be entirely empty if $b_i > b_{i+1}$.

(2) The next byte of rV, s, specifies the current *page size*, which is 2^s bytes. We must have $s \ge 13$ (hence at least 8192 bytes per page). Values of s larger than, say, 20 or so are of use only in rather large programs that will reside in main memory for long periods of time, because memory protection and swapping are applied to entire pages. The maximum legal value of s is 48.

(3) The remaining five bytes of rV are a 27-bit *root location* r, a 10-bit *address space number* n, and a 3-bit *function field* f:

	4	4	4	4	8	27	10	3
rV =	b_1	b_2	b_3	b_4	s	r	n	f

Normally $f = 0$; if $f = 1$, virtual address translation will be done by software instead of hardware, and the b_1, b_2, b_3, b_4, and r fields of rV will be ignored by the hardware. (Values of $f > 1$ are reserved for possible future use; if $f > 1$ when `MMIX` tries to translate an address, a memory-protection failure will occur.)

(4) Each page has an 8-byte *page table entry* (PTE), which looks like this:

	16	$48-s$	$s-13$	10	3
PTE =	x	a	y	n	p

Here x and y are ignored (thus they are usable for any purpose by the operating system); a is the physical address of byte 0 on the page; and n is the address space number (which must match the number in rV). The final three bits are the *protection bits* $p_r\,p_w\,p_x$; the user needs $p_r = 1$ to load from this page, $p_w = 1$ to store on this page, and $p_x = 1$ to execute instructions on this page. If n fails to match the number in rV, or if the appropriate protection bit is zero, a memory-protection fault occurs.

Page table entries should be writable only by the operating system. The 16 ignored bits of x imply that physical memory size is limited to 2^{48} bytes (namely 256 large terabytes); that should be enough capacity for awhile, if not for the entire new millennium.

(5) A given 61-bit address A belongs to page $\lfloor A/2^s \rfloor$ of its segment, and

$$\phi_i(A) = 2^s\,a + (A \bmod 2^s)$$

if a is the address in the PTE for page $\lfloor A/2^s \rfloor$ of segment i.

(6) Suppose $\lfloor A/2^s \rfloor = (a_4a_3a_2a_1a_0)_{1024}$ in the radix-1024 number system. In the common case $a_4 = a_3 = a_2 = a_1 = 0$, the PTE is simply the octabyte $\mathrm{m}_8[2^{13}(r+b_i)+8a_0]$; this rule defines the mapping for the first 1024 pages. The next million or so pages are accessed through an auxiliary *page table pointer*

	1	50	10	3
PTP =	1	c	n	q

in $\mathrm{m}_8[2^{13}(r+b_i+1)+8a_1]$; here the sign must be 1 and the n-field must match rV, but the q bits are ignored. The desired PTE for page $(a_1a_0)_{1024}$ is then in $\mathrm{m}_8[2^{13}c+8a_0]$. The next billion or so pages, namely the pages $(a_2a_1a_0)_{1024}$ with $a_2 \ne 0$, are accessed similarly, through an auxiliary PTP at level two; and so on.

Notice that if $b_3 = b_4$, there is just one page in segment 3, and its PTE appears all alone in physical location $2^{13}(r+b_3)$. Otherwise the PTEs appear in 1024-octabyte blocks. We usually have $0 < b_1 < b_2 < b_3 < b_4$, but the null case $b_1 = b_2 = b_3 = b_4 = 0$ is worthy of mention: In this special case there is only one page, and the segment bits of a virtual address are ignored; the other $61-s$ bits of each virtual address must be zero.

If $s = 13$, $b_1 = 3$, $b_2 = 2$, $b_3 = 1$, and $b_4 = 0$, there are at most 2^{30} pages of 8192 bytes each, all belonging to segment 0. This is essentially the virtual memory setup in the Alpha 21064 computers with DIGITAL UNIX™.

I know these rules look extremely complicated, and I sincerely wish I could have found an alternative that would be both simple and efficient in practice. I tried various schemes based on hashing, but came to the conclusion that "trie" methods such as those described here are better for this application. Indeed, the page tables in most contemporary computers are based on very similar ideas, but with significantly smaller virtual addresses and without the shortcut for small page numbers. I tried also to find formats for rV and the page tables that would match byte boundaries in a more friendly way, but the corresponding page sizes did not work well. Fortunately these grungy details are almost always completely hidden from ordinary users.

46. Of course `MMIX` can't afford to perform a lengthy calculation of physical addresses every time it accesses memory. The machine therefore maintains a *translation cache* (TC), which contains the translations of recently accessed pages. (In fact, there usually are two such caches, one for instructions and one for data.) A TC holds a set of 64-bit translation keys

1	2	$61 - s$	$s - 13$	10	3
0	i	v	0	n	0

associated with 38-bit translations

$48 - s$	$s - 13$	3
a	0	p

representing the relevant parts of the PTE for page v of segment i. Different processes typically have different values of n, and possibly also different values of s. The operating system needs a way to keep such caches up to date when pages are being allocated, moved, swapped, or recycled. The operating system also likes to know which pages have been recently used. The `LDVTS` instructions facilitate such operations:

• `LDVTS $X,$Y,$Z|Z` 'load virtual translation status'.

The sum $Y + $Z or $Y + Z should have the form of a translation cache key as above, except that the rightmost three bits need not be zero. If this key is present in a TC, the rightmost three bits replace the current protection code p; however, if p is thereby set to zero, the key is removed from the TC. Register X is set to 0 if the key was not present in any translation cache, or to 1 if the key was present in the TC for instructions, or to 2 if the key was present in the TC for data, or to 3 if the key was present in both. This instruction is for the operating system only.

47. We mentioned earlier that cheap versions of MMIX might calculate the physical addresses with software instead of hardware, using forced traps when the operating system needs to do page table calculations. Here is some code that could be used for such purposes; it defines the translation process precisely, given a nonnegative virtual address in register rYY. First we must unpack the fields of rV and compute the relevant base addresses for PTEs and PTPs:

```
GET    virt,rYY
GET    $7,rV          % $7=(virtual translation register)
AND    $1,$7,#7       % $1=rightmost three bits
BNZ    $1,Fail        % those bits should be zero
SRU    $1,virt,61     % $1=i (segment number of virtual address)
SLU    $1,$1,2
NEG    $1,52,$1       % $1=52-4i
SRU    $1,$7,$1
SLU    $2,$1,4
SETL   $0,#f000
AND    $1,$1,$0       % $1=b[i]<<12
AND    $2,$2,$0       % $2=b[i+1]<<12
SLU    $3,$7,24
SRU    $3,$3,37
SLU    $3,$3,13       % $3=(r field of rV)
ORH    $3,#8000       % make $3 a physical address
2ADDU  base,$1,$3     % base=address of first page table
2ADDU  limit,$2,$3    % limit=address after last page table
SRU    s,$7,40
AND    s,s,#ff        % s=(s field of rV)
CMP    $0,s,13
BN     $0,Fail        % s must be 13 or more
CMP    $0,s,49
BNN    $0,Fail        % s must be 48 or less
SETH   mask,#8000
ORL    mask,#1ff8     % mask=(sign bit and n field)
ORH    $7,#8000       % set sign bit for PTP validation below
SRU    $0,virt,s      % $0=a4a3a2a1a0 (page number of virt)
ZSZ    $1,$0,1        % $1=[page number is zero]
ADD    limit,limit,$1 % increase limit if page number is zero
```

The next part of the routine finds the "digits" of the page number $(a_4a_3a_2a_1a_0)_{1024}$, from right to left:

```
OR   $5,base,0      OR   $5,base,0      OR   $5,base,0      OR   $5,base,0
SRU  $1,$0,10       SRU  $2,$1,10       SRU  $3,$2,10       SRU  $4,$3,10
PBZ  $1,1F          PBZ  $2,2F          PBZ  $3,3F          PBZ  $4,4F
AND  $0,#3ff        AND  $1,#3ff        AND  $2,#3ff        AND  $3,#3ff
INCL base,#2000     INCL base,#2000     INCL base,#2000     INCL base,#2000
```

Then the process cascades back through PTPs.

```
OR    $5,base,0        ANDNL  base,#1fff       ANDNL  base,#1fff       ANDNL  base,#1fff
8ADDU $6,$4,base    4H 8ADDU  $6,$3,base    3H 8ADDU  $6,$2,base    2H 8ADDU  $6,$1,base
LDO   base,$6,0        LDO    base,$6,0        LDO    base,$6,0        LDO    base,$6,0
XOR   $6,base,$7       XOR    $6,base,$7       XOR    $6,base,$7       XOR    $6,base,$7
AND   $6,$6,mask       AND    $6,$6,mask       AND    $6,$6,mask       AND    $6,$6,mask
BNZ   $6,Fail          BNZ    $6,Fail          BNZ    $6,Fail          BNZ    $6,Fail
```

Finally we obtain the PTE and communicate it to the machine. If errors have been detected, we set the translation to zero; actually any translation with permission bits zero would have the same effect.

```
      ANDNL  base,#1fff    % remove low 13 bits of PTP
1H    8ADDU  $6,$0,base
      LDO    base,$6,0     % base=PTE
      XOR    base,base,$7
      ANDN   $6,base,#7
      SLU    $6,$6,51
      BNZ    $6,Fail       % branch if n doesn't match
      CMP    $6,$5,limit
      BN     $6,Ready      % did we run off the end of the page table?
Fail  SETL   base,0        % errors lead to PTE of zero
Ready PUT    rZZ,base
      LDO    $255,IntMask  % load the desired setting of rK
      RESUME 1             % now the machine will digest the translation
```

All loads and stores in this program deal with negative virtual addresses. This effectively shuts off memory mapping and makes the page tables inaccessible to the user.

The program assumes that the ropcode in rXX is 3 (which it is when a forced trap is triggered by the need for virtual translation).

The translation from virtual pages to physical pages need not actually follow the rules for PTPs and PTEs; any other mapping could be substituted by operating systems with special needs. But people usually want compatibility between different implementations whenever possible. The only parts of rV that MMIX really needs are the s field, which defines page sizes, and the n field, which keeps TC entries of one process from being confused with the TC entries of another.

48. The complete instruction set. We have now described all of `MMIX`'s special registers—except one: The special *failure location register* rF is set to a physical memory address when a parity error or other memory fault occurs. (The instruction leading to this error will probably be long gone before such a fault is detected; for example, the machine might be trying to write old data from a cache in order to make room for new data. Thus there is generally no connection between the current virtual program location rW and the physical location of a memory error. But knowledge of the latter location can still be useful for hardware repair, or when an operating system is booting up.)

49. One additional instruction proves to be useful.

- `SWYM X,Y,Z` 'sympathize with your machinery'.

This command lubricates the disk drives, fans, magnetic tape drives, laser printers, scanners, and any other mechanical equipment hooked up to `MMIX`, if necessary. Fields X, Y, and Z are ignored.

The `SWYM` command was originally included in `MMIX`'s repertoire because machines occasionally need grease to keep in shape, just as human beings occasionally need to swim or do some other kind of exercise in order to maintain good muscle tone. But in fact, `SWYM` has turned out to be a "no-op," an instruction that does nothing at all; the hypothetical manufacturers of our hypothetical machine have pointed out that modern computer equipment is already well oiled and sealed for permanent use. Even so, a no-op instruction provides a good way for software to send signals to the hardware, for such things as scheduling the way instructions are issued on superscalar superpipelined buzzword-compliant machines. Software programs can also use no-ops to communicate with other programs like symbolic debuggers.

When a forced trap computes the translation rZZ of a virtual address rYY, ropcode 3 of `RESUME 1` will put (rYY, rZZ) into the TC for instructions if the opcode in rXX is `SWYM`; otherwise (rYY, rZZ) will be put into the TC for data.

50. The running time of MMIX programs depends to a great extent on changes in technology. MMIX is a mythical machine, but its mythical hardware exists in cheap, slow versions as well as in costly high-performance models. Details of running time usually depend on things like the amount of main memory available to implement virtual memory, as well as the sizes of caches and other buffers.

For practical purposes, the running time of an MMIX program can often be estimated satisfactorily by assigning a fixed cost to each operation, based on the approximate running time that would be obtained on a high-performance machine with lots of main memory; so that's what we will do. Each operation will be assumed to take an integer number of υ, where υ (pronounced "oops") is a unit that represents the clock cycle time in a pipelined implementation. The value of υ will probably decrease from year to year, but I'll keep calling it υ. The running time will also depend on the number of memory references or *mems* that a program uses; this is the number of load and store instructions. For example, each LDO (load octa) instruction will be assumed to cost $\mu + \upsilon$, where μ is the average cost of a memory reference. The total running time of a program might be reported as, say, $35\mu + 1000\upsilon$, meaning 35 mems plus 1000 oops. The ratio μ/υ will probably increase with time, so mem-counting is likely to become increasingly important. [See the discussion of mems in *The Stanford GraphBase* (New York: ACM Press, 1994).]

Integer addition, subtraction, and comparison all take just 1υ. The same is true for SET, GET, PUT, SYNC, and SWYM instructions, as well as bitwise logical operations, shifts, relative jumps, comparisons, conditional assignments, and correctly predicted branches-not-taken or probable-branches-taken. Mispredicted branches or probable branches cost 3υ, and so do the POP and GO commands. Integer multiplication takes 10υ; integer division weighs in at 60υ. TRAP, TRIP, and RESUME cost 5υ each.

Most floating point operations have a nominal running time of 4υ, although the comparison operators FCMP, FEQL, and FUN need only 1υ. FDIV and FSQRT cost 40υ each. The actual running time of floating point computations will vary depending on the operands; for example, the machine might need one extra υ for each denormal input or output, and it might slow down greatly when trips are enabled. The FREM instruction might typically cost $(3 + \delta)\upsilon$, where δ is the amount by which the exponent of the first operand exceeds the exponent of the second (or zero, if this amount is negative). A floating point operation might take only 1υ if at least one of its operands is zero, infinity, or NaN. However, the fixed values stated at the beginning of this paragraph will be used for all seat-of-the-pants estimates of running time, since we want to keep the estimates as simple as possible without making them terribly out of line.

All load and store operations will be assumed to cost $\mu + \upsilon$, except that CSWAP costs $2\mu + 2\upsilon$. (This applies to all OP codes that begin with #8, #9, #A, and #B, except #98–#9F and #B8–#BF. It's best to keep the rules simple, because μ is just an approximate device for estimating average memory cost.) SAVE and UNSAVE are charged $20\mu + \upsilon$.

Of course we must remember that these numbers are very rough. We have not included the cost of fetching instructions from memory. Furthermore, an integer multiplication or division might have an effective cost of only 1υ, if the result is not needed while other numbers are being calculated. Only a detailed simulation can be expected to be truly realistic.

51. If you think that `MMIX` has plenty of operation codes, you are right; we have now described them all. Here is a chart that shows their numeric values:

<table>
<tr><th></th><th>#0</th><th>#1</th><th>#2</th><th>#3</th><th>#4</th><th>#5</th><th>#6</th><th>#7</th><th></th></tr>
<tr><td rowspan="2">#0x</td><td>TRAP</td><td>FCMP</td><td>FUN</td><td>FEQL</td><td>FADD</td><td>FIX</td><td>FSUB</td><td>FIXU</td><td rowspan="2">#0x</td></tr>
<tr><td colspan="2">FLOT[I]</td><td colspan="2">FLOTU[I]</td><td colspan="2">SFLOT[I]</td><td colspan="2">SFLOTU[I]</td></tr>
<tr><td rowspan="2">#1x</td><td>FMUL</td><td>FCMPE</td><td>FUNE</td><td>FEQLE</td><td>FDIV</td><td>FSQRT</td><td>FREM</td><td>FINT</td><td rowspan="2">#1x</td></tr>
<tr><td colspan="2">MUL[I]</td><td colspan="2">MULU[I]</td><td colspan="2">DIV[I]</td><td colspan="2">DIVU[I]</td></tr>
<tr><td rowspan="2">#2x</td><td colspan="2">ADD[I]</td><td colspan="2">ADDU[I]</td><td colspan="2">SUB[I]</td><td colspan="2">SUBU[I]</td><td rowspan="2">#2x</td></tr>
<tr><td colspan="2">2ADDU[I]</td><td colspan="2">4ADDU[I]</td><td colspan="2">8ADDU[I]</td><td colspan="2">16ADDU[I]</td></tr>
<tr><td rowspan="2">#3x</td><td colspan="2">CMP[I]</td><td colspan="2">CMPU[I]</td><td colspan="2">NEG[I]</td><td colspan="2">NEGU[I]</td><td rowspan="2">#3x</td></tr>
<tr><td colspan="2">SL[I]</td><td colspan="2">SLU[I]</td><td colspan="2">SR[I]</td><td colspan="2">SRU[I]</td></tr>
<tr><td rowspan="2">#4x</td><td colspan="2">BN[B]</td><td colspan="2">BZ[B]</td><td colspan="2">BP[B]</td><td colspan="2">BOD[B]</td><td rowspan="2">#4x</td></tr>
<tr><td colspan="2">BNN[B]</td><td colspan="2">BNZ[B]</td><td colspan="2">BNP[B]</td><td colspan="2">BEV[B]</td></tr>
<tr><td rowspan="2">#5x</td><td colspan="2">PBN[B]</td><td colspan="2">PBZ[B]</td><td colspan="2">PBP[B]</td><td colspan="2">PBOD[B]</td><td rowspan="2">#5x</td></tr>
<tr><td colspan="2">PBNN[B]</td><td colspan="2">PBNZ[B]</td><td colspan="2">PBNP[B]</td><td colspan="2">PBEV[B]</td></tr>
<tr><td rowspan="2">#6x</td><td colspan="2">CSN[I]</td><td colspan="2">CSZ[I]</td><td colspan="2">CSP[I]</td><td colspan="2">CSOD[I]</td><td rowspan="2">#6x</td></tr>
<tr><td colspan="2">CSNN[I]</td><td colspan="2">CSNZ[I]</td><td colspan="2">CSNP[I]</td><td colspan="2">CSEV[I]</td></tr>
<tr><td rowspan="2">#7x</td><td colspan="2">ZSN[I]</td><td colspan="2">ZSZ[I]</td><td colspan="2">ZSP[I]</td><td colspan="2">ZSOD[I]</td><td rowspan="2">#7x</td></tr>
<tr><td colspan="2">ZSNN[I]</td><td colspan="2">ZSNZ[I]</td><td colspan="2">ZSNP[I]</td><td colspan="2">ZSEV[I]</td></tr>
<tr><td rowspan="2">#8x</td><td colspan="2">LDB[I]</td><td colspan="2">LDBU[I]</td><td colspan="2">LDW[I]</td><td colspan="2">LDWU[I]</td><td rowspan="2">#8x</td></tr>
<tr><td colspan="2">LDT[I]</td><td colspan="2">LDTU[I]</td><td colspan="2">LDO[I]</td><td colspan="2">LDOU[I]</td></tr>
<tr><td rowspan="2">#9x</td><td colspan="2">LDSF[I]</td><td colspan="2">LDHT[I]</td><td colspan="2">CSWAP[I]</td><td colspan="2">LDUNC[I]</td><td rowspan="2">#9x</td></tr>
<tr><td colspan="2">LDVTS[I]</td><td colspan="2">PRELD[I]</td><td colspan="2">PREGO[I]</td><td colspan="2">GO[I]</td></tr>
<tr><td rowspan="2">#Ax</td><td colspan="2">STB[I]</td><td colspan="2">STBU[I]</td><td colspan="2">STW[I]</td><td colspan="2">STWU[I]</td><td rowspan="2">#Ax</td></tr>
<tr><td colspan="2">STT[I]</td><td colspan="2">STTU[I]</td><td colspan="2">STO[I]</td><td colspan="2">STOU[I]</td></tr>
<tr><td rowspan="2">#Bx</td><td colspan="2">STSF[I]</td><td colspan="2">STHT[I]</td><td colspan="2">STCO[I]</td><td colspan="2">STUNC[I]</td><td rowspan="2">#Bx</td></tr>
<tr><td colspan="2">SYNCD[I]</td><td colspan="2">PREST[I]</td><td colspan="2">SYNCID[I]</td><td colspan="2">PUSHGO[I]</td></tr>
<tr><td rowspan="2">#Cx</td><td colspan="2">OR[I]</td><td colspan="2">ORN[I]</td><td colspan="2">NOR[I]</td><td colspan="2">XOR[I]</td><td rowspan="2">#Cx</td></tr>
<tr><td colspan="2">AND[I]</td><td colspan="2">ANDN[I]</td><td colspan="2">NAND[I]</td><td colspan="2">NXOR[I]</td></tr>
<tr><td rowspan="2">#Dx</td><td colspan="2">BDIF[I]</td><td colspan="2">WDIF[I]</td><td colspan="2">TDIF[I]</td><td colspan="2">ODIF[I]</td><td rowspan="2">#Dx</td></tr>
<tr><td colspan="2">MUX[I]</td><td colspan="2">SADD[I]</td><td colspan="2">MOR[I]</td><td colspan="2">MXOR[I]</td></tr>
<tr><td rowspan="2">#Ex</td><td>SETH</td><td>SETMH</td><td>SETML</td><td>SETL</td><td>INCH</td><td>INCMH</td><td>INCML</td><td>INCL</td><td rowspan="2">#Ex</td></tr>
<tr><td>ORH</td><td>ORMH</td><td>ORML</td><td>ORL</td><td>ANDNH</td><td>ANDNMH</td><td>ANDNML</td><td>ANDNL</td></tr>
<tr><td rowspan="2">#Fx</td><td colspan="2">JMP[B]</td><td colspan="2">PUSHJ[B]</td><td colspan="2">GETA[B]</td><td colspan="2">PUT[I]</td><td rowspan="2">#Fx</td></tr>
<tr><td>POP</td><td>RESUME</td><td>SAVE</td><td>UNSAVE</td><td>SYNC</td><td>SWYM</td><td>GET</td><td>TRIP</td></tr>
<tr><td></td><td>#8</td><td>#9</td><td>#A</td><td>#B</td><td>#C</td><td>#D</td><td>#E</td><td>#F</td><td></td></tr>
</table>

The notation '`[I]`' indicates an operation with an "immediate" variant in which the Z field denotes a constant instead of a register number. Similarly, '`[B]`' indicates an operation with a "backward" variant in which a relative address has a negative displacement. Simulators and other programs that need to present `MMIX` instructions in symbolic form will say that opcode `#20` is `ADD` while opcode `#21` is `ADDI`; they will say that `#F2` is `PUSHJ` while `#F3` is `PUSHJB`. But the `MMIX` assembler uses only the forms `ADD` and `PUSHJ`, not `ADDI` or `PUSHJB`.

To read this chart, use the hexadecimal digits at the top, bottom, left, and right. For example, operation code `A9` in hexadecimal notation appears in the lower part of the #`Ax` row and in the #`1`/#`9` column; it is `STTI`, 'store tetrabyte immediate'.

52. Index. (References are to section numbers, not page numbers.)

MMIX