1. Definition of MMIXAL. This program takes input written in `MMIXAL`, the `MMIX` assembly language, and translates it into binary files that can be loaded and executed on `MMIX` simulators. `MMIXAL` is much simpler than the "industrial strength" assembly languages that computer manufacturers usually provide, because it is primarily intended for the simple demonstration programs in *The Art of Computer Programming*. Yet it tries to have enough features to serve also as the back end of compilers for C and other high-level languages.

Instructions for using the program appear at the end of this document. First we will discuss the input and output languages in detail; then we'll consider the translation process, step by step; then we'll put everything together.

2. A program in `MMIXAL` consists of a series of *lines*, each of which usually contains a single instruction. However, lines with no instructions are possible, and so are lines with two or more instructions.

Each instruction has three parts called its label field, opcode field, and operand field; these fields are separated from each other by one or more spaces. The label field, which is often empty, consists of all characters up to the first blank space. The opcode field, which is never empty, runs from the first nonblank after the label to the next blank space. The operand field, which again might be empty, runs from the next nonblank character (if any) to the first blank or semicolon that isn't part of a string or character constant. If the operand field is followed by a semicolon, possibly with intervening blanks, a new instruction begins immediately after the semicolon; otherwise the rest of the line is ignored. The end of a line is treated as a blank space for the purposes of these rules, with the additional proviso that string or character constants are not allowed to extend from one line to another.

The label field must begin with a letter or a digit; otherwise the entire line is treated as a comment. Popular ways to introduce comments, either at the beginning of a line or after the operand field, are to precede them by the character `%` as in TeX, or by `//` as in C++; `MMIXAL` is not very particular. However, Lisp-style comments introduced by single semicolons will fail if they follow an instruction, because they will be assumed to introduce another instruction.

3. `MMIXAL` has no built-in macro capability, nor does it know how to include header files and such things. But users can run their files through a standard C preprocessor to obtain `MMIXAL` programs in which macros and such things have been expanded. (Caution: The preprocessor also removes C-style comments, unless it is told not to do so.) Literate programming tools could also be used for preprocessing.

If a line begins with the special form '`#` ⟨integer⟩ ⟨string⟩', this program interprets it as a *line directive* emitted by a preprocessor. For example,

```
# 13 "foo.mms"
```

means that the following line was line 13 in the user's source file `foo.mms`. Line directives allow us to correlate errors with the user's original file; we also pass them to the output, for use by simulators and debuggers.

4. `MMIXAL` deals primarily with *symbols* and *constants*, which it interprets and combines to form machine language instructions and data. Constants are simplest, so we will discuss them first.

A *decimal constant* is a sequence of digits, representing a number in radix 10. A *hexadecimal constant* is a sequence of hexadecimal digits, preceded by `#`, representing a number in radix 16:

$$\begin{aligned}
&\langle\,\text{digit}\,\rangle \longrightarrow \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9}\\
&\langle\,\text{hex digit}\,\rangle \longrightarrow \langle\,\text{digit}\,\rangle \mid \texttt{A} \mid \texttt{B} \mid \texttt{C} \mid \texttt{D} \mid \texttt{E} \mid \texttt{F} \mid \texttt{a} \mid \texttt{b} \mid \texttt{c} \mid \texttt{d} \mid \texttt{e} \mid \texttt{f}\\
&\langle\,\text{decimal constant}\,\rangle \longrightarrow \langle\,\text{digit}\,\rangle \mid \langle\,\text{decimal constant}\,\rangle\langle\,\text{digit}\,\rangle\\
&\langle\,\text{hex constant}\,\rangle \longrightarrow \texttt{\#}\langle\,\text{hex digit}\,\rangle \mid \langle\,\text{hex constant}\,\rangle\langle\,\text{hex digit}\,\rangle
\end{aligned}$$

Constants whose value is 2^{64} or more are reduced modulo 2^{64}.

5. A *character constant* is a single character enclosed in single quote marks; it denotes the ASCII or Unicode number corresponding to that character. For example, `'a'` represents the constant `#61`, also known as `97`. The quoted character can be anything except the character that the C library calls `\n` or *newline*; that character should be represented as `#12`.

$$\langle\,\text{character constant}\,\rangle \longrightarrow \texttt{'}\langle\,\text{single byte character except newline}\,\rangle\texttt{'}$$
$$\langle\,\text{constant}\,\rangle \longrightarrow \langle\,\text{decimal constant}\,\rangle \mid \langle\,\text{hex constant}\,\rangle \mid \langle\,\text{character constant}\,\rangle$$

Notice that `'''` represents a single quote, the code `#27`; and `'\'` represents a backslash, the code `#5c`. `MMIXAL` characters are never "quoted" by backslashes as in the C language.

In the present implementation a character constant will always be at most 255, since wyde character input is not supported. The present program does not support Unicode directly because basic software for inputting and outputting 16-bit characters was still in a primitive state at the time of writing. But the data structures below are designed so that a change to Unicode will not be difficult when the time is ripe.

6. A *string constant* like `"Hello"` is an abbreviation for a sequence of one or more character constants separated by commas: `'H','e','l','l','o'`. Any character except newline or the double quote mark `"` can appear between the double quotes of a string constant.

7. A *symbol* in `MMIXAL` is any sequence of letters and digits, beginning with a letter. A colon '`:`' or underscore symbol '`_`' is regarded as a letter, for purposes of this definition. All extended-ASCII characters like '`é`', whose 8-bit code exceeds 126, are also treated as letters.

$$\langle\,\text{letter}\,\rangle \longrightarrow \texttt{A} \mid \texttt{B} \mid \cdots \mid \texttt{Z} \mid \texttt{a} \mid \texttt{b} \mid \cdots \mid \texttt{z} \mid \texttt{:} \mid \texttt{\_} \mid \langle\,\text{character with code value} > 126\,\rangle$$
$$\langle\,\text{symbol}\,\rangle \longrightarrow \langle\,\text{letter}\,\rangle \mid \langle\,\text{symbol}\,\rangle\langle\,\text{letter}\,\rangle \mid \langle\,\text{symbol}\,\rangle\langle\,\text{digit}\,\rangle$$

In future implementations, when `MMIXAL` is used with Unicode, all wyde characters whose 16-bit code exceeds 126 will be regarded as letters; thus `MMIXAL` symbols will be able to involve Greek letters or Chinese characters or thousands of other glyphs.

8. A symbol is said to be *fully qualified* if it begins with a colon. Every symbol that is not fully qualified is an abbreviation for the fully qualified symbol obtained by placing the *current prefix* in front of it; the current prefix is always fully qualified. At the beginning of an `MMIXAL` program the current prefix is simply the single character '`:`', but the user can change it with the `PREFIX` command. For example,

```
ADD     x,y,z             % means ADD :x,:y,:z
PREFIX  Foo:              % current prefix is :Foo:
ADD     x,y,z             % means ADD :Foo:x,:Foo:y,:Foo:z
PREFIX  Bar:              % current prefix is :Foo:Bar:
ADD     :x,y,:z           % means ADD :x,:Foo:Bar:y,:z
PREFIX  :                 % current prefix reverts to :
ADD     x,Foo:Bar:y,Foo:z % means ADD :x,:Foo:Bar:y,:Foo:z
```

This mechanism allows large programs to avoid conflicts between symbol names, when parts of the program are independent and/or written by different users. The current prefix conventionally ends with a colon, but this convention need not be obeyed.

9. A *local symbol* is a decimal digit followed by one of the letters `B`, `F`, or `H`, meaning "backward," "forward," or "here":

$$\langle\,\text{local operand}\,\rangle \longrightarrow \langle\,\text{digit}\,\rangle\,\texttt{B} \mid \langle\,\text{digit}\,\rangle\,\texttt{F}$$
$$\langle\,\text{local label}\,\rangle \longrightarrow \langle\,\text{digit}\,\rangle\,\texttt{H}$$

The `B` and `F` forms are permitted only in the operand field of `MMIXAL` instructions; the `H` form is permitted only in the location field. A local operand such as `2B` stands for the last local label `2H` in instructions before the current one, or 0 if `2H` has not yet appeared as a label. A local operand such as `2F` stands for the first `2H` in instructions after the current one. Thus, in a sequence such as

```
2H JMP 2F; 2H JMP 2B
```

the first instruction jumps to the second and the second jumps to the first.

Local symbols are useful for references to nearby points of a program, in cases where no meaningful name is appropriate. They can also be useful in special situations where a redefinable symbol is needed; for example, an instruction like

```
9H IS 9B+1
```

will maintain a running counter.

10. Each symbol receives a value called its *equivalent* when it appears in the label field of an instruction; it is said to be *defined* after its equivalent has been established. A few symbols, like `rA` and `ROUND_OFF` and `Fopen`, are predefined because they refer to fixed constants associated with the `MMIX` hardware or its rudimentary operating system; otherwise every symbol should be defined exactly once. The two appearances of '`2H`' in the example above do not violate this rule, because the second '`2H`' is not the same symbol as the first.

A predefined symbol can be redefined (given a new equivalent). After it has been redefined it acts like an ordinary symbol and cannot be redefined again. A complete list of the predefined symbols appears in the program listing below.

Equivalents are either *pure* or *register numbers*. A pure equivalent is an unsigned octabyte, but a register number equivalent is a one-byte value, between 0 and 255. A dollar sign is used to change a pure number into a register number; for example, '`$20`' means register number 20.

11. Constants and symbols are combined into *expressions* in a simple way:

⟨ primary expression ⟩ ⟶ ⟨ constant ⟩ | ⟨ symbol ⟩ | ⟨ local operand ⟩ | `@` |
  `(`⟨ expression ⟩`)` | ⟨ unary operator ⟩⟨ primary expression ⟩
⟨ term ⟩ ⟶ ⟨ primary expression ⟩ | ⟨ term ⟩⟨ strong operator ⟩⟨ primary expression ⟩
⟨ expression ⟩ ⟶ ⟨ term ⟩ | ⟨ expression ⟩⟨ weak operator ⟩⟨ term ⟩
⟨ unary operator ⟩ ⟶ `+` | `-` | `~` | `$` | `&`
⟨ strong operator ⟩ ⟶ `*` | `/` | `//` | `%` | `<<` | `>>` | `&`
⟨ weak operator ⟩ ⟶ `+` | `-` | `|` | `^`

Each expression has a value that is either pure or a register number. The character `@` stands for the current location, which is always pure. The unary operators `+`, `-`, `~`, `$`, and `&` mean, respectively, "do nothing," "subtract from zero," "complement the bits," "change from pure value to register number," and "take the serial number." Only the first of these, `+`, can be applied to a register number. The last unary operator, `&`, applies only to symbols, and it is of interest primarily to system programmers; it converts a symbol to the unique positive integer that is used to identify it in the binary file output by `MMIXAL`.

Binary operators come in two flavors, strong and weak. The strong ones are essentially concerned with multiplication or division: `x*y`, `x/y`, `x//y`, `x%y`, `x<<y`, `x>>y`, and `x&y` stand respectively for $(x \times y) \bmod 2^{64}$ (multiplication), $\lfloor x/y \rfloor$ (division), $\lfloor 2^{64}x/y \rfloor$ (fractional division), $x \bmod y$ (remainder), $(x \times 2^y) \bmod 2^{64}$ (left shift), $\lfloor x/2^y \rfloor$ (right shift), and $x \wedge y$ (bitwise and) on unsigned octabytes. Division is legal only if $y > 0$; fractional division is legal only if $x < y$. None of the strong binary operations can be applied to register numbers.

The weak binary operations `x+y`, `x-y`, `x|y`, and `x^y` stand respectively for $(x + y) \bmod 2^{64}$ (addition), $(x-y) \bmod 2^{64}$ (subtraction), $x \vee y$ (bitwise or), and $x \oplus y$ (bitwise exclusive-or) on unsigned octabytes. These operations can be applied to register numbers only in four contexts: ⟨ register ⟩ + ⟨ pure ⟩, ⟨ pure ⟩ + ⟨ register ⟩, ⟨ register ⟩ − ⟨ pure ⟩ and ⟨ register ⟩ − ⟨ register ⟩. For example, if `x` denotes `$1` and `y` denotes `$10`, then `x+3` and `3+x` denote `$4`, and `y-x` denotes the pure value `9`.

Register numbers within expressions are allowed to be arbitrary octabytes, but a register number assigned as the equivalent of a symbol should not exceed 255.

(Incidentally, one might ask why the designer of `MMIXAL` did not simply adopt the existing rules of C for expressions. The primary reason is that the designers of C chose to give `<<`, `>>`, and `&` a lower precedence than `+`; but in `MMIXAL` we want to be able to write things like `o<<24+x<<16+y<<8+z` or `@+yz<<2` or `@+(#100-@)&#ff`. Since the conventions of C were inappropriate, it was better to make a clean break, not pretending to have a close relationship with that language. The new rules are quite easily memorized, because `MMIXAL` has just two levels of precedence, and the strong binary operations are all essentially multiplicative by nature while the weak binary operations are essentially additive.)

12. A symbol is called a *future reference* until it has been defined. `MMIXAL` restricts the use of future references, so that programs can be assembled quickly in one pass over the input; therefore all expressions can be evaluated when the `MMIXAL` processor first sees them.

The restrictions are easily stated: Future references cannot be used in expressions together with unary or binary operators (except the unary `+`, which does nothing); moreover, future references can appear as operands only in instructions that have relative addresses (namely branches, probable branches, `JMP`, `PUSHJ`, `GETA`) or in octabyte constants (the pseudo-operation `OCTA`). Thus, for example, one can say `JMP 1F` or `JMP 1B-4`, but not `JMP 1F-4`.

13. We noted earlier that each `MMIXAL` instruction contains a label field, an opcode field, and an operand field. The label field is either empty or a symbol or local label; when it is nonempty, the symbol or local label receives an equivalent. The operand field is either empty or a sequence of expressions separated by commas; when it is empty, it is equivalent to the simple operand field '`0`'.

$$\begin{aligned}&\langle\,\text{instruction}\,\rangle \longrightarrow \langle\,\text{label}\,\rangle\langle\,\text{opcode}\,\rangle\langle\,\text{operand list}\,\rangle\\&\langle\,\text{label}\,\rangle \longrightarrow \langle\,\text{empty}\,\rangle \mid \langle\,\text{symbol}\,\rangle \mid \langle\,\text{local label}\,\rangle\\&\langle\,\text{operand list}\,\rangle \longrightarrow \langle\,\text{empty}\,\rangle \mid \langle\,\text{expression list}\,\rangle\\&\langle\,\text{expression list}\,\rangle \longrightarrow \langle\,\text{expression}\,\rangle \mid \langle\,\text{expression list}\,\rangle\texttt{,}\langle\,\text{expression}\,\rangle\end{aligned}$$

The opcode field either contains a symbolic `MMIX` operation name (like `ADD`), or an *alias operation*, or a *pseudo-operation*. Alias operations are alternate names for `MMIX` operations whose standard names are inappropriate in certain contexts. Pseudo-operations do not correspond directly to `MMIX` commands, but they govern the assembly process in important ways.

There are two alias operations:

• `SET $X,$Y` is equivalent to `OR $X,$Y,0`; it sets register X to register Y. Similarly, `SET $X,Y` (when `Y` is not a register) is equivalent to `SETL $X,Y`.

• `LDA $X,$Y,$Z` is equivalent to `ADDU $X,$Y,$Z`; it loads the address of memory location $\$\mathrm{Y} + \$\mathrm{Z}$ into register X. Similarly, `LDA $X,$Y,Z` is equivalent to `ADDU $X,$Y,Z`.

The symbolic operation names for genuine `MMIX` operations should not include the suffix `I` for an immediate operation or the suffix `B` for a backward jump; `MMIXAL` determines such things automatically. Thus, one never writes `ADDI` or `JMPB` in the source input to `MMIXAL`, although such opcodes might appear when a simulator or debugger or disassembler is presenting a numeric instruction in symbolic form.

$$\begin{aligned}&\langle\,\text{opcode}\,\rangle \longrightarrow \langle\,\text{symbolic }\texttt{MMIX}\text{ operation}\,\rangle \mid \langle\,\text{alias operation}\,\rangle\\&\qquad\qquad\qquad\qquad\qquad \mid \langle\,\text{pseudo-operation}\,\rangle\\&\langle\,\text{symbolic }\texttt{MMIX}\text{ operation}\,\rangle \longrightarrow \texttt{TRAP} \mid \texttt{FCMP} \mid \cdots \mid \texttt{TRIP}\\&\langle\,\text{alias operation}\,\rangle \longrightarrow \texttt{SET} \mid \texttt{LDA}\\&\langle\,\text{pseudo-operation}\,\rangle \longrightarrow \texttt{IS} \mid \texttt{LOC} \mid \texttt{PREFIX} \mid \texttt{GREG} \mid \texttt{LOCAL} \mid \texttt{BSPEC} \mid \texttt{ESPEC}\\&\qquad\qquad\qquad\qquad\qquad \mid \texttt{BYTE} \mid \texttt{WYDE} \mid \texttt{TETRA} \mid \texttt{OCTA}\end{aligned}$$

14. `MMIX` operations like `ADD` require exactly three expressions as operands. The first two must be register numbers. The third must be either a register number or a pure number between 0 and 255; in the latter case, `ADD` becomes `ADDI` in the assembled output. Thus, for example, the command "set register 1 to the sum of register 2 and register 3" could be expressed as

```
ADD $1,$2,$3
```

or as, say,

```
ADD x,y,y+1
```

if the equivalent of `x` is `$1` and the equivalent of `y` is `$2`. The command "subtract 5 from register 1' could be expressed as

```
SUB $1,$1,5
```

or as

```
SUB x,x,5
```

but not as '`SUBI $1,$1,5`' or '`SUBI x,x,5`'.

`MMIX` operations like `FLOT` require either three operands (register, pure, register/pure) or only two (register, register/pure). In the first case the middle operand is the rounding mode, which is best expressed in terms of the predefined symbolic values `ROUND_CURRENT`, `ROUND_OFF`, `ROUND_UP`, `ROUND_DOWN`, `ROUND_NEAR`, for $(0,1,2,3,4)$ respectively. In the second case the middle operand is understood to be zero (namely, `ROUND_CURRENT`).

`MMIX` operations like `SETL` or `INCH`, which involve a wyde intermediate constant, require exactly two operands, (register, pure). The value of the second operand should fit in two bytes.

`MMIX` operations like `BNZ`, which mention a register and a relative address, also require two operands. The first operand should be a register number. The second operand, when subtracted from the current location and divided by four, should yield a result r in the range $-2^{16} \le r < 2^{16}$. The second operand might also be undefined; in that case, the eventual value must satisfy the restriction stated for defined values. The opcodes `GETA` and `PUSHJ` are similar, except that the first operand to `PUSHJ` might also be pure (see below). The `JMP` operation is also similar, but it has only one operand, and it allows the larger address range $-2^{24} \le r < 2^{24}$.

`MMIX` operations that refer to memory, like `LDO` and `STHT` and `GO`, are treated like `ADD` if they have three operands, except that the first operand should be pure (not a register number) in the case of `PRELD`, `PREGO`, `PREST`, `STCO`, `SYNCD`, and `SYNCID`. These opcodes also accept a special two-operand form, in which the second operand stands for a *base address* and an immediate offset (see below).

The first operand of `PUSHJ` and `PUSHGO` can be either a pure number or a register number. In the first case ('`PUSHJ 2,Sub`' or '`PUSHGO 2,Sub`') the programmer might be thinking "let's push down two registers"; in the second case ('`PUSHJ $2,Sub`' or '`PUSHGO $2,Sub`') the programmer might be thinking "let's make register 2 the hole position for this subroutine call." Both cases result in the same assembled output.

The remaining `MMIX` opcodes are idiosyncratic:

```
NEG r,p,z;
PUT s,z;
GET r,s;
POP p,yz;
RESUME xyz;
SAVE r,0;
UNSAVE r;
SYNC xyz;
TRAP x,y,z  or  TRAP x,yz  or  TRAP xyz;
```

`SWYM` and `TRIP` are like `TRAP`. Here `s` is an integer between 0 and 31, preferably given by one of the predefined symbols `rA`, `rB`, ... for special register codes; `r` is a register number; `p` is a pure byte; `x`, `y`, and `z` are either register numbers or pure bytes; `yz` and `xyz` are pure values that fit respectively in two and three bytes.

All of these rules can be summarized by saying that `MMIXAL` treats each `MMIX` opcode in the most natural way. When there are three operands, they affect fields X, Y, and Z of the assembled `MMIX` instruction; when there are two operands, they affect fields X and YZ; when there is just one operand, it affects field XYZ.

15. In all cases when the opcode corresponds to an `MMIX` operation, the `MMIXAL` instruction tells the assembler to carry out four steps: (1) Align the current location so that it is a multiple of 4, by adding 1, 2, or 3 if necessary; (2) Define the equivalent of the label field to be the current location, if the label is nonempty; (3) Evaluate the operands and assemble the specified `MMIX` instruction into the current location; (4) Increase the current location by 4.

16. Now let's consider the pseudo-operations, starting with the simplest cases.

• ⟨label⟩ `IS` ⟨expression⟩ defines the value of the label to be the value of the expression, which must not be a future reference. The expression may be either pure or a register number.

• ⟨label⟩ `LOC` ⟨expression⟩ first defines the label to be the value of the current location, if the label is nonempty. Then the current location is changed to the value of the expression, which must be pure.

For example, '`LOC #1000`' will start assembling subsequent instructions or data in location whose hexadecimal value is #`1000`. '`X LOC @+500`' defines `X` to be the address of the first of 500 bytes in memory; assembly will continue at location `X` + 500. The operation of aligning the current location to a multiple of 256, if it is not already aligned in that way, can be expressed as '`LOC @+(256-@)&255`'.

A less trivial example arises if we want to emit instructions and data into two separate areas of memory, but we want to intermix them in the `MMIXAL` source file. We could start by defining `8H` and `9H` to be the starting addresses of the instruction and data segments, respectively. Then, a sequence of instructions could be enclosed in '`LOC 8B`; ...; `8H IS @`'; a sequence of data could be enclosed in '`LOC 9B`; ...; `9H IS @`'. Any number of such sequences could then be combined. Instead of the two pseudo-instructions '`8H IS @; LOC 9B`' one could in fact write simply '`8H LOC 9B`' when switching from instructions to data.

• `PREFIX` ⟨symbol⟩ redefines the current prefix to be the given symbol (fully qualified). The label field should be blank.

17. The next pseudo-operations assemble bytes, wydes, tetrabytes, or octabytes of data.

• ⟨label⟩ `BYTE` ⟨expression list⟩ defines the label to be the current location, if the label field is nonempty; then it assembles one byte for each expression in the expression list, and advances the current location by the number of bytes. The expressions should all be pure numbers that fit in one byte.

String constants are often used in such expression lists. For example, if the current location is #`1000`, the instruction `BYTE "Hello",0` assembles six bytes containing the constants `'H'`, `'e'`, `'l'`, `'l'`, `'o'`, and `0` into locations #`1000`, ..., #`1005`, and advances the current location to #`1006`.

• ⟨label⟩ `WYDE` ⟨expression list⟩ is similar, but it first makes the current location even, by adding 1 to it if necessary. Then it defines the label (if a nonempty label is present), and assembles each expression as a two-byte value. The current location is advanced by twice the number of expressions in the list. The expressions should all be pure numbers that fit in two bytes.

• ⟨label⟩ `TETRA` ⟨expression list⟩ is similar, but it aligns the current location to a multiple of 4 before defining the label; then it assembles each expression as a four-byte value. The current location is advanced by $4n$ if there are n expressions in the list. Each expression should be a pure number that fits in four bytes.

• ⟨label⟩ `OCTA` ⟨expression list⟩ is similar, but it first aligns the current location to a multiple of 8; it assembles each expression as an eight-byte value. The current location is advanced by $8n$ if there are n expressions in the list. Any or all of the expressions may be future references, but they should all be defined as pure numbers eventually.

18. Global registers are important for accessing memory in `MMIX` programs. They could be allocated by hand, and defined with `IS` instructions, but `MMIXAL` provides a mechanism that is usually much more convenient:

• ⟨label⟩ `GREG` ⟨expression⟩ allocates a new global register, and assigns its number as the equivalent of the label. At the beginning of assembly, the current global threshold G is \$255. Each distinct `GREG` instruction decreases G by 1; the final value of G will be the initial value of rG when the assembled program is loaded.

The value of the expression will be loaded into the global register at the beginning of the program. *If this value is nonzero, it should remain constant throughout the program execution*; such global registers are considered to be *base addresses*. Two or more base addresses with the same constant value are assigned to the same global register number.

Base addresses can simplify memory accesses in an important way. Suppose, for example, five octabyte values appear in a data segment, and their addresses are called `AA`, `BB`, `CC`, `DD`, and `EE`:

```
AA LOC @+8;BB LOC @+8;CC LOC @+8;DD LOC @+8;EE LOC @+8
```

Then if you say `Base GREG AA`, you will be able to write simply '`LDO $1,AA`' to bring `AA` into register `$1`, and '`LDO $2,CC`' to bring `CC` into register `$2`.

Here's how it works: Whenever a memory operation such as `LDO` or `STB` or `GO` has only two operands, the second operand should be a pure number whose value can be expressed as $b + \delta$, where $0 \le \delta < 256$ and b is the value of a base address in one of the preceding `GREG` commands. The `MMIXAL` processor will find the closest base address and manufacture an appropriate command. For example, the instruction '`LDO $2,CC`' in the example of the preceding paragraph would be converted automatically to '`LDO $2,Base,16`'.

If no base address is close enough, an error message will be generated, unless this program is run with the `-x` option on the command line. The `-x` option inserts additional instructions if necessary, using global register 255, so that any address is accessible. For example, if there is no base address that allows `LDO $2,FF` to be implemented in a single instruction, but if `FF` equals `Base+1000`, then the `-x` option would assemble two instructions,

```
SETL $255,1000; LDO $2,Base,$255
```

in place of `LDO $2,FF`. Caution: The `-x` feature makes the number of actual `MMIX` instructions hard to predict, so extreme care must be used if your style of coding includes relative branch instructions in dangerous forms like '`BNZ x,@+8`'.

This base address convention can be used also with the alias operation `LDA`. For example, '`LDA $3,CC`' loads the address of `CC` into register 3, by assembling the instruction '`ADDU $3,Base,16`'.

`MMIXAL` also allows a two-operand form for memory operations such as

```
LDO $1,$2
```

to be an abbreviation for '`LDO $1,$2,0`'.

When `MMIXAL` programs use subroutines with a memory stack in addition to the built-in register stack, they usually begin with the instructions '`sp GREG 0;fp GREG 0`'; these instructions allocate a *stack pointer* `sp=$254` and a *frame pointer* `fp=$253`. However, subroutine libraries are free to implement any conventions for global registers and stacks that they like.

19. Short programs rarely run out of global registers, but long programs need a mechanism to check that `GREG` hasn't been used too often. The following pseudo-instruction provides the necessary safety valve:

• `LOCAL` ⟨expression⟩ ensures that the expression will be a local register in the program being assembled. The expression should be a register number, and the label field should be blank. At the close of assembly, `MMIXAL` will report an error if the final value of G does not exceed all register numbers that are declared local in this way.

A `LOCAL` instruction need not be given unless the register number is 32 or more. (`MMIX` always considers `$0` through `$31` to be local, so `MMIXAL` implicitly acts as if the instruction '`LOCAL $31`' were present.)

20. Finally, there are two pseudo-instructions to pass information and hints to the loading routine and/or to debuggers that will be using the assembled program.

- `BSPEC` ⟨expression⟩ begins "special mode"; the ⟨expression⟩ should have a value that fits in two bytes, and the label field should be blank.
- `ESPEC` ends "special mode"; the operand field is ignored, and the label field should be blank.

All material assembled between `BSPEC` and `ESPEC` is passed directly to the output, but not loaded as part of the assembled program. Ordinary `MMIX` instructions cannot appear in special mode; only the pseudo-operations `IS`, `PREFIX`, `BYTE`, `WYDE`, `TETRA`, `OCTA`, `GREG`, and `LOCAL` are allowed. The operand of `BSPEC` should have a value that fits in two bytes; this value identifies the kind of data that follows. (For example, `BSPEC 0` might introduce information about subroutine calling conventions at the current location, and `BSPEC 1` might introduce line numbers from a high-level-language program that was compiled into the code at the current place. System routines often need to pass such information through an assembler to the operating system, hence `MMIXAL` provides a general-purpose conduit.)

21. A program should begin at the special symbolic location `Main` (more precisely, at the address corresponding to the fully qualified symbol `:Main`). This symbol always has serial number 1, and it must always be defined.

Locations should not receive assembled data more than once. (More precisely, the loader will load the bitwise xor of all the data assembled for each byte position; but the general rule "do not load two things into the same byte" is safest.) All locations that do not receive assembled data are initially zero, except that the loading routine will put register stack data into segment 3, and the operating system may put command-line data and debugger data into segment 2. (The rudimentary `MMIX` operating system starts a program with the number of command-line arguments in \$0, and a pointer to the beginning of an array of argument pointers in \$1.) Segments 2 and 3 should not get assembled data, unless the user is a true hacker who is willing to take the risk that such data might crash the system.

22. Binary MMO output. When the MMIXAL processor assembles a file called foo.mms, it produces a binary output file called foo.mmo. (The suffix mms stands for "MMIX symbolic," and mmo stands for "MMIX object.") Such mmo files have a simple structure consisting of a sequence of tetrabytes. Some of the tetrabytes are instructions to a loading routine; others are data to be loaded.

Loader instructions are distinguished from tetrabytes of data by their first (most significant) byte, which has the special escape-code value #98, called *mm* in the program below. This code value corresponds to MMIX's opcode LDVTS, which is unlikely to occur in tetras of data. The second byte X of a loader instruction is the loader opcode, called the *lopcode*. The third and fourth bytes, Y and Z, are operands. Sometimes they are combined into a single 16-bit operand called YZ.

#define *mm* #98

23. A small, contrived example will help explain the basic ideas of mmo format. Consider the following input file, called test.mms:

```
% A peculiar example of MMIXAL
     LOC   Data_Segment      % location #2000000000000000
     OCTA  1F                % a future reference
a    GREG  @                 % $254 is base address for ABCD
ABCD BYTE  "ab"              % two bytes of data
     LOC   #123456789        % switch to the instruction segment
Main JMP   1F                % another future reference
     LOC   @+#4000           % skip past 16384 bytes
2H   LDB   $3,ABCD+1         % use the base address
     BZ    $3,1F; TRAP       % and refer to the future again
# 3 "foo.mms"                % this comment is a line directive
     LOC   2B-4*10           % move 10 tetras before previous location
1H   JMP   2B                % resolve previous references to 1F
     BSPEC 5                 % begin special data of type 5
     TETRA &a<<8             % four bytes of special data
     WYDE  a-$0              % two more bytes of special data
     ESPEC                   % end a special data packet
     LOC   ABCD+2            % resume the data segment
     BYTE  "cd",#98          % assemble three more bytes of data
```

It defines a silly program that essentially puts 'b' into register 3; the program halts when it gets to an all-zero TRAP instruction following the BZ. But the assembled output of this file illustrates most of the features of MMIX objects, and in fact test.mms was the first test file tried by the author when the MMIXAL processor was originally written.

The binary output file test.mmo assembled from test.mms consists of the following tetrabytes, shown in hexadecimal notation with brief comments. Fuller explanations appear with the descriptions of individual lopcodes below.

```
98090101  lop_pre 1,1 (preamble, version 1, 1 tetra)
36f4a363  (the file creation time)
98012001  lop_loc #20,1 (data segment, 1 tetra)
00000000  (low tetrabyte of address in data segment)
00000000  (high tetrabyte of OCTA 1F)
00000000  (low tetrabyte, will be fixed up later)
61620000  ("ab", padded with trailing zeros)
```

98010002	*lop_loc* 0, 2 (instruction segment, 2 tetras)
00000001	(high tetrabyte of address in instruction segment)
2345678c	(low tetrabyte of address, after alignment)
98060002	*lop_file* 0, 2 (file name 0, 2 tetras)
74657374	("test")
2e6d6d73	(".mms")
98070007	*lop_line* 7 (line 7 of the current file)
f0000000	(JMP 1F, will be fixed up later)
98024000	*lop_skip* #4000 (advance 16384 bytes)
98070009	*lop_line* 9 (line 9 of the current file)
8103fe01	(LDB $3,b,1, uses base address b)
42030000	(BZ $3,1F, will be fixed later)
9807000a	*lop_line* 10 (stay on line 10)
00000000	(TRAP)
98010002	*lop_loc* 0, 2 (instruction segment, 2 tetras)
00000001	(high tetrabyte of address in instruction segment)
2345a768	(low tetrabyte of address 1H)
98050010	*lop_fixrx* 16 (fix 16-bit relative address)
0100fff5	(fixup for location @-4*-11)
98040ff7	*lop_fixr* #ff7 (fix @-4*#ff7)
98032001	*lop_fixo* #20, 1 (data segment, 1 tetra)
00000000	(low tetrabyte of data segment address to fix)
98060102	*lop_file* 1, 2 (file name 1, 2 tetras)
666f6f2e	("foo.")
6d6d7300	("mms",0)
98070004	*lop_line* 4 (line 4 of the current file)
f000000a	(JMP 2B)
98080005	*lop_spec* 5 (begin special data of type 5)
00000200	(TETRA &a<<8)
00fe0000	(WYDE a-$0)
98012001	*lop_loc* #20, 1 (data segment, 1 tetra)
0000000a	(low tetrabyte of address in data segment)
00006364	("cd" with leading zeros, because of alignment)
98000001	*lop_quote* (don't treat next tetrabyte as a lopcode)
98000000	(BYTE #98, padded with trailing zeros)
980a00fe	*lop_post* $254 (begin postamble, G is 254)
20000000	(high tetrabyte of the initial contents of $254)
00000008	(low tetrabyte of base address $254)
00000001	(high tetrabyte of the initial contents of $255)
2345678c	(low tetrabyte of $255, is address of Main)
980b0000	*lop_stab* (begin symbol table)
203a5040	(compressed form for symbol table as a ternary trie)
50404020	
41204220	
43094408	
83404020	(ABCD = #2000000000000008, serial 3)
4d206120	
69056e01	
2345678c	
81400f61	(Main = #000000012345678c, serial 1)
fe820000	(a = $254, serial 2)
980c000a	*lop_end* (end symbol table, 10 tetras)

24. When a tetrabyte of the `mmo` file does not begin with the escape code, it is loaded into the current location λ, and λ is increased to the next higher multiple of 4. (If λ is not a multiple of 4, the tetrabyte actually goes into location $\lambda \wedge (-4) = 4\lfloor\lambda/4\rfloor$, according to `MMIX`'s usual conventions.) The current line number is also increased by 1, if it is nonzero.

When a tetrabyte does begin with the escape code, its next byte is the lopcode defining a loader instruction. There are thirteen lopcodes:

• *lop_quote*: X = #`00`, YZ = 1. Treat the next tetra as an ordinary tetrabyte, even if it begins with the escape code.

• *lop_loc*: X = #`01`, Y = high byte, Z = tetra count (Z = 1 or 2). Set the current location to the 64-bit address defined by the next Z tetras, plus 2^{56}Y. Usually Y = 0 (for the instruction segment) or Y = #`20` (for the data segment). If Z = 2, the high tetra appears first.

• *lop_skip*: X = #`02`, YZ = delta. Increase the current location by YZ.

• *lop_fixo*: X = #`03`, Y = high byte, Z = tetra count (Z = 1 or 2). Load the value of the current location λ into octabyte P, where P is the 64-bit address defined by the next Z tetras plus 2^{56}Y as in *lop_loc*. (The octabyte at P was previously assembled as zero because of a future reference.)

• *lop_fixr*: X = #`04`, YZ = delta. Load YZ into the YZ field of the tetrabyte in location P, where P is $\lambda - 4$YZ, namely the address that precedes the current location by YZ tetrabytes. (This tetrabyte was previously loaded with an `MMIX` instruction that takes a relative address: a branch, probable branch, `JMP`, `PUSHJ`, or `GETA`. Its YZ field was previously assembled as zero because of a future reference.)

• *lop_fixrx*: X = #`05`, Y = 0, Z = 16 or 24. Proceed as in *lop_fixr*, but load δ into tetrabyte P $= \lambda - 4\delta$ instead of loading YZ into P $= \lambda - 4$YZ. Here δ is the value of the tetrabyte following the *lop_fixrx* instruction; its leading byte will either 0 or 1. If the leading byte is 1, δ should be treated as the *negative* number $(\delta \wedge$ #`ffffff`$) - 2^{\rm Z}$ when calculating the address P. (The latter case arises only rarely, but it is needed when fixing up a relative "future" reference that ultimately leads to a "backward" instruction. The value of δ that is xored into location P in such cases will change `BZ` to `BZB`, or `JMP` to `JMPB`, etc.; we have Z = 24 when fixing a `JMP`, Z = 16 otherwise.)

• *lop_file*: X = #`06`, Y = file number, Z = tetra count. Set the current file number to Y and the current line number to zero. If this file number has occurred previously, Z should be zero; otherwise Z should be positive, and the next Z tetrabytes are the characters of the file name in big-endian order. Trailing zeros follow the file name if its length is not a multiple of 4.

• *lop_line*: X = #`07`, YZ = line number. Set the current line number to YZ. If the line number is nonzero, the current file and current line should correspond to the source location that generated the next data to be loaded, for use in diagnostic messages. (The `MMIXAL` processor gives precise line numbers to the sources of tetrabytes in segment 0, which tend to be instructions, but not to the sources of tetrabytes assembled in other segments.)

• *lop_spec*: X = #`08`, YZ = type. Begin special data of type YZ. The subsequent tetrabytes, continuing until the next loader operation other than *lop_quote*, comprise the special data. A *lop_quote* instruction allows tetrabytes of special data to begin with the escape code.

• *lop_pre*: X = #`09`, Y = 1, Z = tetra count. A *lop_pre* instruction, which defines the "preamble," must be the first tetrabyte of every `mmo` file. The Y field specifies the version number of `mmo` format, currently 1; other version numbers may be defined later, but version 1 should always be supported as described in the present document. The Z tetrabytes following a *lop_pre* command provide additional information that might be of interest to system routines. If Z > 0, the first tetra of additional information records the time that this `mmo` file was created, measured in seconds since 00:00:00 Greenwich Mean Time on 1 Jan 1970.

• *lop_post*: X = #`0a`, Y = 0, Z = G (must be 32 or more). This instruction begins the *postamble*, which follows all instructions and data to be loaded. It causes the loaded program to begin with rG equal to the stated value of G, and with \$G, G + 1, ..., \$255 initially set to the values of the next $(256 - {\rm G}) * 2$ tetrabytes. These tetrabytes specify $256 - {\rm G}$ octabytes in big-endian fashion (high half first).

• *lop_stab*: X = #0b, YZ = 0. This instruction must appear immediately after the $(256 - G) * 2$ tetrabytes following *lop_post*. It is followed by the symbol table, which lists the equivalents of all user-defined symbols in a compact form that will be described later.

• *lop_end*: X = #0c, YZ = tetra count. This instruction must be the very last tetrabyte of each mmo file. Furthermore, exactly YZ tetrabytes must appear between it and the *lop_stab* command. (Therefore a program can easily find the symbol table without reading forward through the entire mmo file.)

A separate routine called MMOtype is available to translate binary mmo files into human-readable form.

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

25. Many readers will have noticed that MMIXAL has no facilities for relocatable output, nor does mmo format support such features. The author's first drafts of MMIXAL and mmo did allow relocatable objects, with external linkages, but the rules were substantially more complicated and therefore inconsistent with the goals of *The Art of Computer Programming*. The present design might actually prove to be superior to the current practice, now that computer memory is significantly cheaper than it used to be, because one-pass assembly and loading are extremely fast when relocatability and external linkages are disallowed. Different program modules can be assembled together about as fast as they could be linked together under a relocatable scheme, and they can communicate with each other in much more flexible ways. Debugging tools are enhanced when open-source libraries are combined with user programs, and such libraries will certainly improve in quality when their source form is accessible to a larger community of users.