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:

⟨ digit ⟩ ⟶ `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`
⟨ hex digit ⟩ ⟶ ⟨ digit ⟩ | `A` | `B` | `C` | `D` | `E` | `F` | `a` | `b` | `c` | `d` | `e` | `f`
⟨ decimal constant ⟩ ⟶ ⟨ digit ⟩ | ⟨ decimal constant ⟩⟨ digit ⟩
⟨ hex constant ⟩ ⟶ `#`⟨ hex digit ⟩ | ⟨ hex constant ⟩⟨ hex digit ⟩

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 `#a`.

⟨ character constant ⟩ ⟶ `'`⟨ single byte character except newline ⟩`'`
⟨ constant ⟩ ⟶ ⟨ decimal constant ⟩ | ⟨ hex constant ⟩ | ⟨ character constant ⟩

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.

⟨ letter ⟩ ⟶ `A` | `B` | ⋯ | `Z` | `a` | `b` | ⋯ | `z` | `:` | `_` | ⟨ character with code value >126 ⟩
⟨ symbol ⟩ ⟶ ⟨ letter ⟩ | ⟨ symbol ⟩⟨ letter ⟩ | ⟨ symbol ⟩⟨ digit ⟩

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":

⟨ local operand ⟩ ⟶ ⟨ digit ⟩ `B` | ⟨ digit ⟩ `F`
⟨ local label ⟩ ⟶ ⟨ digit ⟩ `H`

The `B` and `F` forms are permitted only in the operand field of `MMIXAL` instructions; the `H` form is permitted only in the label 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 $x \mathbin{\&} y$ (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), $x \mid y$ (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`'.

⟨ instruction ⟩ ⟶ ⟨ label ⟩⟨ opcode ⟩⟨ operand list ⟩
⟨ label ⟩ ⟶ ⟨ empty ⟩ | ⟨ symbol ⟩ | ⟨ local label ⟩
⟨ operand list ⟩ ⟶ ⟨ empty ⟩ | ⟨ expression list ⟩
⟨ expression list ⟩ ⟶ ⟨ expression ⟩ | ⟨ expression list ⟩`,`⟨ expression ⟩

The opcode field contains either 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 $Y + $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.

⟨ opcode ⟩ ⟶ ⟨ symbolic `MMIX` operation ⟩ | ⟨ alias operation ⟩
| ⟨ pseudo-operation ⟩
⟨ symbolic `MMIX` operation ⟩ ⟶ `TRAP` | `FCMP` | ··· | `TRIP`
⟨ alias operation ⟩ ⟶ `SET` | `LDA`
⟨ pseudo-operation ⟩ ⟶ `IS` | `LOC` | `PREFIX` | `GREG` | `LOCAL` | `BSPEC` | `ESPEC`
| `BYTE` | `WYDE` | `TETRA` | `OCTA`

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 should yield a result r in the range $-2^{16} \le r < 2^{16}$ when the current location is subtracted from it and the result is divided by 4. 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 #20,1 (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 $(\delta \wedge \#\texttt{ffffff}) - 2^{\text{Z}}$ 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 $(256-\text{G})*2$ tetrabytes. These tetrabytes specify 256 − 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.

26. Basic data types. This program for the 64-bit `MMIX` architecture is based on 32-bit integer arithmetic, because nearly every computer available to the author at the time of writing was limited in that way. Details of the basic arithmetic appear in a separate program module called MMIX-ARITH, because the same routines are needed also for the simulators. The definition of type **tetra** should be changed, if necessary, to conform with the definitions found in MMIX-ARITH.

```
⟨ Type definitions 26 ⟩ ≡
  typedef unsigned int tetra;      /* assumes that an int is exactly 32 bits wide */
  typedef struct {
    tetra h, l;
  } octa;      /* two tetrabytes make one octabyte */
  typedef enum {
    false, true
  } bool;
```

See also sections 30, 54, 58, 62, 68, and 82.

This code is used in section 136.

27.
```
⟨ Global variables 27 ⟩ ≡
  extern octa zero_octa;      /* zero_octa.h = zero_octa.l = 0 */
  extern octa neg_one;      /* neg_one.h = neg_one.l = −1 */
  extern octa aux;      /* auxiliary output of a subroutine */
  extern bool overflow;      /* set by certain subroutines for signed arithmetic */
```

See also sections 33, 36, 37, 43, 46, 51, 56, 60, 63, 67, 69, 77, 83, 90, 105, 120, 133, 139, and 143.

This code is used in section 136.

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

```
⟨ Subroutines 28 ⟩ ≡
  extern octa oplus ARGS((octa y, octa z));      /* unsigned y + z */
  extern octa ominus ARGS((octa y, octa z));      /* unsigned y − z */
  extern octa incr ARGS((octa y, int delta));      /* unsigned y + δ (δ is signed) */
  extern octa oand ARGS((octa y, octa z));      /* y ∧ z */
  extern octa shift_left ARGS((octa y, int s));      /* y ≪ s, 0 ≤ s ≤ 64 */
  extern octa shift_right ARGS((octa y, int s, int u));      /* y ≫ s, signed if ¬u */
  extern octa omult ARGS((octa y, octa z));      /* unsigned (aux, x) = y × z */
  extern octa odiv ARGS((octa x, octa y, octa z));      /* unsigned (x, y)/z; aux = (x, y) mod z */
```

See also sections 41, 42, 44, 45, 47, 48, 49, 50, 52, 55, 57, 59, 73, and 74.

This code is used in section 136.

29. Here's a rudimentary check to see if arithmetic is in trouble.

```
⟨ Initialize everything 29 ⟩ ≡
  acc = shift_left(neg_one, 1);
  if (acc.h ≠ #ffffffff) panic("Type␣tetra␣is␣not␣implemented␣correctly");
```

See also sections 32, 61, 71, 84, 91, and 140.

This code is used in section 136.

30. Future versions of this program will work with symbols formed from Unicode characters, but the present code limits itself to an 8-bit subset. The type **Char** is defined here in order to ease the later transition: At present, **Char** is the same as **char**, but **Char** can be changed to a 16-bit type in the Unicode version.

Other changes will also be necessary when the transition to Unicode is made; for example, some calls of *fprintf* will become calls of *fwprintf*, and some occurrences of `%s` will become `%ls` in print formats. The switchable type name **Char** provides at least a first step towards a brighter future with Unicode.

⟨ Type definitions 26 ⟩ +≡

```
typedef char Char;      /* bytes that will become wydes some day */
```

31. While we're talking about classic systems versus future systems, we might as well define the `ARGS` macro, which makes function prototypes available on ANSI C systems without making them uncompilable on older systems. Each subroutine below is declared first with a prototype, then with an old-style definition.

⟨ Preprocessor definitions 31 ⟩ ≡

```
#ifdef __STDC__
#define ARGS(list) list
#else
#define ARGS(list) ()
#endif
```

See also section 39.

This code is used in section 136.

32. Basic input and output. Input goes into a buffer that is normally limited to 72 characters. This limit can be raised, by using the `-b` option when invoking the assembler; but short buffers will keep listings from becoming unwieldy, because a symbolic listing adds 19 characters per line.

```
⟨ Initialize everything 29 ⟩ +≡
  if (buf_size < 72) buf_size = 72;
  buffer = (Char *) calloc(buf_size + 1, sizeof(Char));
  lab_field = (Char *) calloc(buf_size + 1, sizeof(Char));
  op_field = (Char *) calloc(buf_size, sizeof(Char));
  operand_list = (Char *) calloc(buf_size, sizeof(Char));
  err_buf = (Char *) calloc(buf_size + 60, sizeof(Char));
  if (¬buffer ∨ ¬lab_field ∨ ¬op_field ∨ ¬operand_list ∨ ¬err_buf) panic("No␣room␣for␣the␣buffers");
```

33.

```
⟨ Global variables 27 ⟩ +≡
  Char *buffer; /* raw input of the current line */
  Char *buf_ptr; /* current position within buffer */
  Char *lab_field; /* copy of the label field of the current instruction */
  Char *op_field; /* copy of the opcode field of the current instruction */
  Char *operand_list; /* copy of the operand field of the current instruction */
  Char *err_buf; /* place where dynamic error messages are sprinted */
```

34.

```
⟨ Get the next line of input text, or break if the input has ended 34 ⟩ ≡
  if (¬fgets(buffer, buf_size + 1, src_file)) break;
  line_no ++;
  line_listed = false;
  j = strlen(buffer);
  if (buffer[j − 1] ≡ '\n') buffer[j − 1] = '\0'; /* remove the newline */
  else if ((j = fgetc(src_file)) ≠ EOF) ⟨ Flush the excess part of an overlong line 35 ⟩;
  if (buffer[0] ≡ '#') ⟨ Check for a line directive 38 ⟩;
  buf_ptr = buffer;
```

This code is used in section 136.

35.

```
⟨ Flush the excess part of an overlong line 35 ⟩ ≡
  { while (j ≠ '\n' ∧ j ≠ EOF) j = fgetc(src_file);
    if (¬long_warning_given)
    { long_warning_given = true;
      err("*trailing␣characters␣of␣long␣input␣line␣have␣been␣dropped");
      fprintf(stderr, "(say␣`-b␣<number>'␣to␣increase␣the␣length␣of␣my␣input␣buffer)\n");
    } else err("*trailing␣characters␣dropped");
  }
```

This code is used in section 34.

36.

```
⟨ Global variables 27 ⟩ +≡
  int cur_file; /* index of the current file in filename */
  int line_no; /* current position in the file */
  bool line_listed; /* have we listed the buffer contents? */
  bool long_warning_given; /* have we given the hint about -b? */
```

37. We keep track of source file name and line number at all times, for error reporting and for synchronization data in the object file. Up to 256 different source file names can be remembered.

```
⟨ Global variables 27 ⟩ +≡
  Char *filename[257];      /* source file names, including those in line directives */
  int filename_count;       /* how many filename entries have we filled? */
```

38. If the current line is a line directive, it will also be treated as a comment by the assembler.

```
⟨ Check for a line directive 38 ⟩ ≡
  { for (p = buffer + 1; isspace(*p); p++) ;
    for (j = 0; isdigit(*p); p++) j = 10 * j + *p - '0';
    for ( ; isspace(*p); p++) ;
    if (*p ≡ '\"')
    { if (¬filename[filename_count])
      { filename[filename_count] = (Char *) calloc(FILENAME_MAX + 1, sizeof(Char));
        if (¬filename[filename_count]) panic("Capacity␣exceeded:␣Out␣of␣filename␣memory");
      }
      for (p++, k = 0; *p ∧ *p ≠ '\"' ∧ k < FILENAME_MAX; p++, k++) filename[filename_count][k] = *p;
      if (k ≡ FILENAME_MAX) panic("Capacity␣exceeded:␣File␣name␣too␣long");
      if (*p ≡ '\"' ∧ *(p − 1) ≠ '\"')
      {       /* yes, it's a line directive */
        filename[filename_count][k] = '\0';
        for (k = 0; strcmp(filename[k], filename[filename_count]) ≠ 0; k++) ;
        if (k ≡ filename_count)
        { if (filename_count ≡ 256) panic("Capacity␣exceeded:␣More␣than␣256␣file␣names");
          filename_count++;
        }
        cur_file = k;
        line_no = j − 1;
      }
    }
  }
```

This code is used in section 34.

39. Archaic versions of the C library do not define `FILENAME_MAX`.

```
⟨ Preprocessor definitions 31 ⟩ +≡
#ifndef FILENAME_MAX
#define FILENAME_MAX  256
#endif
```

40.

```
⟨ Local variables 40 ⟩ ≡
  register Char *p, *q;      /* the place where we're currently scanning */
```

See also section 65.

This code is used in section 136.

41. The next several subroutines are useful for preparing a listing of the assembled results. In such a listing, which the user can request with a command-line option, we fill the leftmost 19 columns with a representation of the output that has been assembled from the input in the buffer. Sometimes the assembled output requires more than one line, because we have room to output only a tetrabyte per line.

The *flush_listing_line* subroutine is called when we have finished generating one line's worth of assembled material. Its parameter is a string to be printed between the assembled material and the buffer contents, if the input line hasn't yet been echoed. The length of this string should be 19 minus the number of characters already printed on the current line of the listing.

```
⟨ Subroutines 28 ⟩ +≡
  void flush_listing_line ARGS((char *));
  void flush_listing_line(s)
      char *s;
  { if (line_listed) fprintf(listing_file, "\n");
    else
    { fprintf(listing_file, "%s%s\n", s, buffer);
      line_listed = true;
    }
  }
```

42. Only the three least significant hex digits of a location are shown on the listing, unless the other digits have changed. The following subroutine prints an extra line when a change needs to be shown.

```
⟨ Subroutines 28 ⟩ +≡
  void update_listing_loc ARGS((int));
  void update_listing_loc(k)
      int k;      /* the location to display, mod 4 */
  { if (cur_loc.h ≠ listing_loc.h ∨ ((cur_loc.l ⊕ listing_loc.l) & #fffff000))
    { fprintf(listing_file, "%08x%08x:", cur_loc.h, (cur_loc.l & -4) | k);
      flush_listing_line("␣␣");
    }
    listing_loc.h = cur_loc.h; listing_loc.l = (cur_loc.l & -4) | k;
  }
```

43.
```
⟨ Global variables 27 ⟩ +≡
  octa cur_loc;      /* current location of assembled output */
  octa listing_loc;      /* current location on the listing */
  unsigned char hold_buf[4];      /* assembled bytes */
  unsigned char held_bits;      /* which bytes of hold_buf are active? */
  unsigned char listing_bits;      /* which of them haven't been listed yet? */
  bool spec_mode;      /* are we between BSPEC and ESPEC? */
  tetra spec_mode_loc;      /* number of bytes in the current special output */
```

44. When bytes are assembled, they are placed into the *hold_buf*. More precisely, a byte assembled for a location that is j plus a multiple of 4 is placed into *hold_buf*$[j]$; two auxiliary variables, *held_bits* and *listing_bits*, are then increased by $1 \ll j$. Furthermore, *listing_bits* is increased by $^{\#}\texttt{10} \ll j$ if that byte is a future reference to be resolved later.

The bytes are held until we need to output them. The *listing_clear* routine lists any that have been held but not yet shown. It should be called only when *listing_bits* $\neq 0$.

```
⟨ Subroutines 28 ⟩ +≡
  void listing_clear ARGS((void));
  void listing_clear( )
  { register int j, k;
    for (k = 0; k < 4; k++)
      if (listing_bits & (1 ≪ k)) break;
    if (spec_mode) fprintf(listing_file, "␣␣␣␣␣␣␣␣␣");
    else
    { update_listing_loc(k);
      fprintf(listing_file, "␣...%03x:␣", (listing_loc.l & #ffc) | k);
    }
    for (j = 0; j < 4; j++)
      if (listing_bits & (#10 ≪ j)) fprintf(listing_file, "xx");
      else if (listing_bits & (1 ≪ j)) fprintf(listing_file, "%02x", hold_buf[j]);
      else fprintf(listing_file, "␣␣");
    flush_listing_line("␣␣");
    listing_bits = 0;
  }
```

45. Error messages are written to *stderr*. If the message begins with '`*`' it is merely a warning; if it begins with '`!`' it is fatal; otherwise the error is probably serious enough to make manual correction necessary, yet it is not tragic. Errors and warnings appear also on the optional listing file.

```
#define err(m)
        { report_error(m); if (m[0] ≠ '*') goto bypass; }
#define derr(m,p)
        { sprintf(err_buf,m,p);
          report_error(err_buf); if (err_buf[0] ≠ '*') goto bypass; }
#define dderr(m,p,q)
        { sprintf(err_buf,m,p,q);
          report_error(err_buf); if (err_buf[0] ≠ '*') goto bypass; }
#define panic(m)
        { sprintf(err_buf,"!%s",m); report_error(err_buf); }
#define dpanic(m,p)
        { err_buf[0] = '!'; sprintf(err_buf+1,m,p); report_error(err_buf); }
```

```
⟨ Subroutines 28 ⟩ +≡
  void report_error ARGS((char *));
  void report_error(message)
      char *message;
  { if (¬filename[cur_file]) filename[cur_file] = "(nofile)";
    if (message[0] ≡ '*')
      fprintf(stderr,"\"%s\", line %d warning: %s\n",filename[cur_file],line_no,message+1);
    else if (message[0] ≡ '!')
      fprintf(stderr,"\"%s\", line %d fatal error: %s\n",filename[cur_file],line_no,message+1);
    else
    { fprintf(stderr,"\"%s\", line %d: %s!\n",filename[cur_file],line_no,message);
      err_count++;
    }
    if (listing_file)
    { if (¬line_listed) flush_listing_line("****************** ");
      if (message[0] ≡ '*') fprintf(listing_file,"********** warning: %s\n",message+1);
      else if (message[0] ≡ '!') fprintf(listing_file,"******** fatal error: %s!\n",message+1);
      else fprintf(listing_file,"********** error: %s!\n",message);
    }
    if (message[0] ≡ '!') exit(-2);
  }
```

46.

```
⟨ Global variables 27 ⟩ +≡
  int err_count;      /* this many errors were found */
```

47. Output to the binary *obj_file* occurs four bytes at a time. The bytes are assembled in small buffers, not output as single tetrabytes, because we want the output to be big-endian even when the assembler is running on a little-endian machine.

```
#define mmo_write(buf) if (fwrite(buf,1,4,obj_file)≠4) dpanic("Can't write on %s",obj_file_name)
⟨ Subroutines 28 ⟩ +≡
  void mmo_clear ARGS((void));
  void mmo_out ARGS((void));
  unsigned char lop_quote_command[4] = { mm,lop_quote,0,1} ;
  void mmo_clear( )      /* clears hold_buf, when held_bits ≠ 0 */
  { if (hold_buf[0] ≡ mm) mmo_write(lop_quote_command);
    mmo_write(hold_buf);
    if (listing_file ∧ listing_bits) listing_clear( );
    held_bits = 0;
    hold_buf[0] = hold_buf[1] = hold_buf[2] = hold_buf[3] = 0;
    mmo_cur_loc = incr(mmo_cur_loc,4); mmo_cur_loc.l &= −4;
    if (mmo_line_no) mmo_line_no++;
  }
  unsigned char mmo_buf[4];
  int mmo_ptr;
  void mmo_out( )      /* output the contents of mmo_buf */
  { if (held_bits) mmo_clear( );
    mmo_write(mmo_buf);
  }
```

48. ⟨ Subroutines 28 ⟩ +≡

```
void mmo_tetra ARGS((tetra));
void mmo_byte ARGS((unsigned char));
void mmo_lop ARGS((char, unsigned char, unsigned char));
void mmo_lopp ARGS((char, unsigned short));
void mmo_tetra(t)      /* output a tetrabyte */
    tetra t;
{ mmo_buf[0] = t >> 24; mmo_buf[1] = (t >> 16) & #ff;
  mmo_buf[2] = (t >> 8) & #ff; mmo_buf[3] = t & #ff;
  mmo_out( );
}
void mmo_byte(b)
    unsigned char b;
{ mmo_buf[(mmo_ptr++) & 3] = b;
  if (¬(mmo_ptr & 3)) mmo_out( );
}
void mmo_lop(x, y, z)      /* output a loader operation */
    char x;
    unsigned char y, z;
{ mmo_buf[0] = mm; mmo_buf[1] = x; mmo_buf[2] = y; mmo_buf[3] = z;
  mmo_out( );
}
void mmo_lopp(x, yz)      /* output a loader operation with two-byte operand */
    char x;
    unsigned short yz;
{ mmo_buf[0] = mm; mmo_buf[1] = x; mmo_buf[2] = yz >> 8; mmo_buf[3] = yz & #ff;
  mmo_out( );
}
```

49. The *mmo_loc* subroutine makes the current location in the object file equal to *cur_loc*.

⟨ Subroutines 28 ⟩ +≡

```
void mmo_loc ARGS((void));
void mmo_loc( )
{ octa o;
  if (held_bits) mmo_clear( );
  o = ominus(cur_loc, mmo_cur_loc);
  if (o.h ≡ 0 ∧ o.l < #10000)
  { if (o.l) mmo_lopp(lop_skip, o.l);
  } else
  { if (cur_loc.h & #ffffff)
    { mmo_lop(lop_loc, 0, 2);
      mmo_tetra(cur_loc.h);
    } else mmo_lop(lop_loc, cur_loc.h >> 24, 1);
    mmo_tetra(cur_loc.l);
  }
  mmo_cur_loc = cur_loc;
}
```

50. Similarly, the *mmo_sync* subroutine makes sure that the current file and line number in the output file agree with *cur_file* and *line_no*.

```
⟨ Subroutines 28 ⟩ +≡
  void mmo_sync ARGS((void));
  void mmo_sync( )
  { register int j;
    register unsigned char *p;
    if (cur_file ≠ mmo_cur_file)
    { if (filename_passed[cur_file]) mmo_lop(lop_file, cur_file, 0);
      else
      { mmo_lop(lop_file, cur_file, (strlen(filename[cur_file]) + 3) ≫ 2);
        for (j = 0, p = filename[cur_file]; *p; p++, j = (j + 1) & 3)
        { mmo_buf[j] = *p;
          if (j ≡ 3) mmo_out( );
        }
        if (j)
        { for ( ; j < 4; j++) mmo_buf[j] = 0;
          mmo_out( );
        }
        filename_passed[cur_file] = 1;
      }
      mmo_cur_file = cur_file;
      mmo_line_no = 0;
    }
    if (line_no ≠ mmo_line_no)
    { if (line_no ≥ #10000) panic("I␣can't␣deal␣with␣line␣numbers␣exceeding␣65535");
      mmo_lopp(lop_line, line_no);
      mmo_line_no = line_no;
    }
  }
```

51.
```
⟨ Global variables 27 ⟩ +≡
  octa mmo_cur_loc;      /* current location in the object file */
  int mmo_line_no;       /* current line number in the mmo output so far */
  int mmo_cur_file;      /* index of the current file in the mmo output so far */
  char filename_passed[256];      /* has a filename been recorded in the output? */
```

52. Here is a basic subroutine that assembles k bytes starting at *cur_loc*. The value of k should be 1, 2, or 4, and *cur_loc* should be a multiple of k. The *x_bits* parameter tells which bytes, if any, are part of a future reference.

```
⟨ Subroutines 28 ⟩ +≡
  void assemble ARGS((char, tetra, unsigned char));
  void assemble(k, dat, x_bits)
      char k;
      tetra dat;
      unsigned char x_bits;
  { register int j, jj, l;
    if (spec_mode) l = spec_mode_loc;
    else
    { l = cur_loc.l;
      ⟨ Make sure cur_loc and mmo_cur_loc refer to the same tetrabyte 53 ⟩;
      if (¬held_bits ∧ ¬(cur_loc.h & #e0000000)) mmo_sync( );
    }
    for (j = 0; j < k; j++)
    { jj = (l + j) & 3;
      hold_buf[jj] = (dat ≫ (8 * (k − 1 − j))) & #ff;
      held_bits |= 1 ≪ jj;
      listing_bits |= 1 ≪ jj;
    }
    listing_bits |= x_bits;
    if (((l + k) & 3) ≡ 0)
    { if (listing_file) listing_clear( );
      mmo_clear( );
    }
    if (spec_mode) spec_mode_loc += k;
    else cur_loc = incr(cur_loc, k);
  }
```

53.

```
⟨ Make sure cur_loc and mmo_cur_loc refer to the same tetrabyte 53 ⟩ ≡
  if (cur_loc.h ≠ mmo_cur_loc.h ∨ ((cur_loc.l ⊕ mmo_cur_loc.l) & #fffffffc)) mmo_loc( );
```

This code is used in section 52.

54. The symbol table. Symbols are stored and retrieved by means of a *ternary search trie*, following ideas of Bentley and Sedgewick. (See *ACM–SIAM Symp. on Discrete Algorithms* **8** (1997), 360–369; R. Sedgewick, *Algorithms in C* (Reading, Mass.: Addison–Wesley, 1998), §15.4.) Each trie node stores a character, and there are branches to subtries for the cases where a given character is less than, equal to, or greater than the character in the trie. There also is a pointer to a symbol table entry if a symbol ends at the current node.

```
⟨ Type definitions 26 ⟩ +≡
  typedef struct ternary_trie_struct {
    unsigned short ch;      /* the (possibly wyde) character stored here */
    struct ternary_trie_struct *left, *mid, *right;      /* downward in the ternary trie */
    struct sym_tab_struct *sym;      /* equivalents of symbols */
  } trie_node;
```

55. We allocate trie nodes in chunks of 1000 at a time.

```
⟨ Subroutines 28 ⟩ +≡
  trie_node *new_trie_node ARGS((void));
  trie_node *new_trie_node()
  { register trie_node *t = next_trie_node;
    if (t ≡ last_trie_node)
    { t = (trie_node *) calloc(1000, sizeof(trie_node));
      if (¬t) panic("Capacity␣exceeded:␣Out␣of␣trie␣memory");
      last_trie_node = t + 1000;
    }
    next_trie_node = t + 1;
    return t;
  }
```

56.
```
⟨ Global variables 27 ⟩ +≡
  trie_node *trie_root;      /* root of the trie */
  trie_node *op_root;      /* root of subtrie for opcodes */
  trie_node *next_trie_node, *last_trie_node;      /* allocation control */
  trie_node *cur_prefix;      /* root of subtrie for unqualified symbols */
```

57. The *trie_search* subroutine starts at a given node of the trie and finds a given string in its middle subtrie, inserting new nodes if necessary. The string ends with the first nonletter or nondigit; the location of the terminating character is stored in global variable *terminator*.

```
#define isletter(c) (isalpha(c) ∨ c ≡ '_' ∨ c ≡ ':' ∨ (unsigned int)(c) > 126)
⟨ Subroutines 28 ⟩ +≡
  trie_node *trie_search ARGS((trie_node *, Char *));
  Char *terminator;      /* where the search ended */
  trie_node *trie_search(t, s)
      trie_node *t;
      Char *s;
  { register trie_node *tt = t;
    register Char *p = s;
    while (1)
    { if (¬isletter(*p) ∧ ¬isdigit(*p))
      { terminator = p; return tt;
      }
      if (tt→mid)
      { tt = tt→mid;
        while (*p ≠ tt→ch)
        { if (*p < tt→ch)
          { if (tt→left) tt = tt→left;
            else
            { tt→left = new_trie_node( ); tt = tt→left; goto store_new_char;
            }
          } else
          { if (tt→right) tt = tt→right;
            else
            { tt→right = new_trie_node( ); tt = tt→right; goto store_new_char;
            }
          }
        }
        p++;
      } else
      { tt→mid = new_trie_node( ); tt = tt→mid;
      store_new_char: tt→ch = *p++;
      }
    }
  }
```

58. Symbol table nodes hold the serial numbers and equivalents of defined symbols. They also hold "fixup information" for undefined symbols; this will allow the loader to correct any previously assembled instructions that refer to such symbols when they are eventually defined.

In the symbol table node for a defined symbol, the *link* field has one of the special codes `DEFINED` or `REGISTER` or `PREDEFINED`, and the *equiv* field holds the defined value. The *serial* number is a unique identifier for all user-defined symbols.

In the symbol table node for an undefined symbol, the *equiv* field is ignored. The *link* field points to the first node of fixup information; that node is, in turn, a symbol table node that might link to other fixups. The *serial* number in a fixup node is either 0 or 1 or 2, meaning respectively "fixup the octabyte pointed to by *equiv*" or "fixup the relative address in the YZ field of the instruction pointed to by *equiv*" or "fixup the relative address in the XYZ field of the instruction pointed to by *equiv*."

```
#define DEFINED (sym_node *) 1 /* code value for octabyte equivalents */
#define REGISTER (sym_node *) 2 /* code value for register-number equivalents */
#define PREDEFINED (sym_node *) 3 /* code value for not-yet-used equivalents */
#define fix_o 0 /* serial code for octabyte fixup */
#define fix_yz 1 /* serial code for relative fixup */
#define fix_xyz 2 /* serial code for JMP fixup */
```

⟨ Type definitions 26 ⟩ +≡

```
typedef struct sym_tab_struct {
  int serial; /* serial number of symbol; type number for fixups */
  struct sym_tab_struct *link; /* DEFINED status or link to fixup */
  octa equiv; /* the equivalent value */
} sym_node;
```

59. The allocation of new symbol table nodes proceeds in chunks, like the allocation of trie nodes. But in this case we also have the possibility of reusing old fixup nodes that are no longer needed.

```
#define recycle_fixup(pp) pp→link = sym_avail, sym_avail = pp
```

⟨ Subroutines 28 ⟩ +≡

```
sym_node *new_sym_node ARGS((bool));
sym_node *new_sym_node(serialize)
    bool serialize; /* should the new node receive a unique serial number? */
{ register sym_node *p = sym_avail;
  if (p)
  { sym_avail = p→link; p→link = Λ; p→serial = 0; p→equiv = zero_octa;
  } else
  { p = next_sym_node;
    if (p ≡ last_sym_node)
    { p = (sym_node *) calloc(1000, sizeof(sym_node));
      if (¬p) panic("Capacity␣exceeded:␣Out␣of␣symbol␣memory");
      last_sym_node = p + 1000;
    }
    next_sym_node = p + 1;
  }
  if (serialize) p→serial = ++serial_number;
  return p;
}
```

60. ⟨ Global variables 27 ⟩ +≡
```
int serial_number;
sym_node *sym_root;  /* root of the sym */
sym_node *next_sym_node, *last_sym_node;  /* allocation control */
sym_node *sym_avail;  /* stack of recycled symbol table nodes */
```

61. We initialize the trie by inserting all the predefined symbols. Opcodes are given the prefix `^`, to distinguish them from ordinary symbols; this character nicely divides uppercase letters from lowercase letters.

⟨ Initialize everything 29 ⟩ +≡
```
trie_root = new_trie_node();
cur_prefix = trie_root;
op_root = new_trie_node();
trie_root→mid = op_root;
trie_root→ch = ':';
op_root→ch = '^';
⟨ Put the MMIX opcodes and MMIXAL pseudo-ops into the trie 64 ⟩;
⟨ Put the special register names into the trie 66 ⟩;
⟨ Put other predefined symbols into the trie 70 ⟩;
```

62. Most of the assembly work can be table driven, based on bits that are stored as the "equivalents" of opcode symbols like `^ADD`.
```
#define rel_addr_bit #1  /* is YZ or XYZ relative? */
#define immed_bit #2  /* should opcode be immediate if Z or YZ not register? */
#define zar_bit #4  /* should register status of Z be ignored? */
#define zr_bit #8  /* must Z be a register? */
#define yar_bit #10  /* should register status of Y be ignored? */
#define yr_bit #20  /* must Y be a register? */
#define xar_bit #40  /* should register status of X be ignored? */
#define xr_bit #80  /* must X be a register? */
#define yzar_bit #100  /* should register status of YZ be ignored? */
#define yzr_bit #200  /* must YZ be a register? */
#define xyzar_bit #400  /* should register status of XYZ be ignored? */
#define xyzr_bit #800  /* must XYZ be a register? */
#define one_arg_bit #1000  /* is it OK to have zero or one operand? */
#define two_arg_bit #2000  /* is it OK to have exactly two operands? */
#define three_arg_bit #4000  /* is it OK to have exactly three operands? */
#define many_arg_bit #8000  /* is it OK to have more than three operands? */
#define align_bits #30000  /* how much alignment: byte, wyde, tetra, or octa? */
#define no_label_bit #40000  /* should the label be blank? */
#define mem_bit #80000  /* must YZ be a memory reference? */
#define spec_bit #100000  /* is this opcode allowed in SPEC mode? */
```
⟨ Type definitions 26 ⟩ +≡
```
typedef struct {
  Char *name;  /* symbolic opcode */
  short code;  /* numeric opcode */
  int bits;  /* treatment of operands */
} op_spec;
typedef enum {
  SET = #100, IS, LOC, PREFIX, BSPEC, ESPEC, GREG, LOCAL,
  BYTE, WYDE, TETRA, OCTA
} pseudo_op;
```

63. ⟨ Global variables 27 ⟩ +≡

```
op_spec op_init_table[] = {
{ "TRAP",#00,#27554} ,{ "FCMP",#01,#240a8} ,{ "FUN",#02,#240a8} ,{ "FEQL",#03,#240a8} ,
{ "FADD",#04,#240a8} ,{ "FIX",#05,#26288} ,{ "FSUB",#06,#240a8} ,{ "FIXU",#07,#26288} ,
{ "FLOT",#08,#26282} ,{ "FLOTU",#0a,#26282} ,{ "SFLOT",#0c,#26282} ,{ "SFLOTU",#0e,
    #26282} ,
{ "FMUL",#10,#240a8} ,{ "FCMPE",#11,#240a8} ,{ "FUNE",#12,#240a8} ,{ "FEQLE",#13,#240a8} ,
{ "FDIV",#14,#240a8} ,{ "FSQRT",#15,#26288} ,{ "FREM",#16,#240a8} ,{ "FINT",#17,#26288} ,
{ "MUL",#18,#240a2} ,{ "MULU",#1a,#240a2} ,{ "DIV",#1c,#240a2} ,{ "DIVU",#1e,#240a2} ,
{ "ADD",#20,#240a2} ,{ "ADDU",#22,#240a2} ,{ "SUB",#24,#240a2} ,{ "SUBU",#26,#240a2} ,
{ "2ADDU",#28,#240a2} ,{ "4ADDU",#2a,#240a2} ,{ "8ADDU",#2c,#240a2} ,{ "16ADDU",#2e,
    #240a2} ,
{ "CMP",#30,#240a2} ,{ "CMPU",#32,#240a2} ,{ "NEG",#34,#26082} ,{ "NEGU",#36,#26082} ,
{ "SL",#38,#240a2} ,{ "SLU",#3a,#240a2} ,{ "SR",#3c,#240a2} ,{ "SRU",#3e,#240a2} ,
{ "BN",#40,#22081} ,{ "BZ",#42,#22081} ,{ "BP",#44,#22081} ,{ "BOD",#46,#22081} ,
{ "BNN",#48,#22081} ,{ "BNZ",#4a,#22081} ,{ "BNP",#4c,#22081} ,{ "BEV",#4e,#22081} ,
{ "PBN",#50,#22081} ,{ "PBZ",#52,#22081} ,{ "PBP",#54,#22081} ,{ "PBOD",#56,#22081} ,
{ "PBNN",#58,#22081} ,{ "PBNZ",#5a,#22081} ,{ "PBNP",#5c,#22081} ,{ "PBEV",#5e,#22081} ,
{ "CSN",#60,#240a2} ,{ "CSZ",#62,#240a2} ,{ "CSP",#64,#240a2} ,{ "CSOD",#66,#240a2} ,
{ "CSNN",#68,#240a2} ,{ "CSNZ",#6a,#240a2} ,{ "CSNP",#6c,#240a2} ,{ "CSEV",#6e,#240a2} ,
{ "ZSN",#70,#240a2} ,{ "ZSZ",#72,#240a2} ,{ "ZSP",#74,#240a2} ,{ "ZSOD",#76,#240a2} ,
{ "ZSNN",#78,#240a2} ,{ "ZSNZ",#7a,#240a2} ,{ "ZSNP",#7c,#240a2} ,{ "ZSEV",#7e,#240a2} ,
{ "LDB",#80,#a60a2} ,{ "LDBU",#82,#a60a2} ,{ "LDW",#84,#a60a2} ,{ "LDWU",#86,#a60a2} ,
{ "LDT",#88,#a60a2} ,{ "LDTU",#8a,#a60a2} ,{ "LDO",#8c,#a60a2} ,{ "LDOU",#8e,#a60a2} ,
{ "LDSF",#90,#a60a2} ,{ "LDHT",#92,#a60a2} ,{ "CSWAP",#94,#a60a2} ,{ "LDUNC",#96,#a60a2} ,
{ "LDVTS",#98,#a60a2} ,{ "PRELD",#9a,#a6022} ,{ "PREGO",#9c,#a6022} ,{ "GO",#9e,#a60a2} ,
{ "STB",#a0,#a60a2} ,{ "STBU",#a2,#a60a2} ,{ "STW",#a4,#a60a2} ,{ "STWU",#a6,#a60a2} ,
{ "STT",#a8,#a60a2} ,{ "STTU",#aa,#a60a2} ,{ "STO",#ac,#a60a2} ,{ "STOU",#ae,#a60a2} ,
{ "STSF",#b0,#a60a2} ,{ "STHT",#b2,#a60a2} ,{ "STCO",#b4,#a6022} ,{ "STUNC",#b6,#a60a2} ,
{ "SYNCD",#b8,#a6022} ,{ "PREST",#ba,#a6022} ,{ "SYNCID",#bc,#a6022} ,{ "PUSHGO",#be,
    #a6062} ,
{ "OR",#c0,#240a2} ,{ "ORN",#c2,#240a2} ,{ "NOR",#c4,#240a2} ,{ "XOR",#c6,#240a2} ,
{ "AND",#c8,#240a2} ,{ "ANDN",#ca,#240a2} ,{ "NAND",#cc,#240a2} ,{ "NXOR",#ce,#240a2} ,
{ "BDIF",#d0,#240a2} ,{ "WDIF",#d2,#240a2} ,{ "TDIF",#d4,#240a2} ,{ "ODIF",#d6,#240a2} ,
{ "MUX",#d8,#240a2} ,{ "SADD",#da,#240a2} ,{ "MOR",#dc,#240a2} ,{ "MXOR",#de,#240a2} ,
{ "SETH",#e0,#22080} ,{ "SETMH",#e1,#22080} ,{ "SETML",#e2,#22080} ,{ "SETL",#e3,#22080} ,
{ "INCH",#e4,#22080} ,{ "INCMH",#e5,#22080} ,{ "INCML",#e6,#22080} ,{ "INCL",#e7,#22080} ,
{ "ORH",#e8,#22080} ,{ "ORMH",#e9,#22080} ,{ "ORML",#ea,#22080} ,{ "ORL",#eb,#22080} ,
{ "ANDNH",#ec,#22080} ,{ "ANDNMH",#ed,#22080} ,{ "ANDNML",#ee,#22080} ,{ "ANDNL",#ef,
    #22080} ,
{ "JMP",#f0,#21001} ,{ "PUSHJ",#f2,#22041} ,{ "GETA",#f4,#22081} ,{ "PUT",#f6,#22002} ,
{ "POP",#f8,#23000} ,{ "RESUME",#f9,#21000} ,{ "SAVE",#fa,#22080} ,{ "UNSAVE",#fb,
    #23a00} ,
{ "SYNC",#fc,#21000} ,{ "SWYM",#fd,#27554} ,{ "GET",#fe,#22080} ,{ "TRIP",#ff,#27554} ,
{ "SET",SET,#22180} ,{ "LDA",#22,#a60a2} ,
{ "IS",IS,#101400} ,{ "LOC",LOC,#1400} ,{ "PREFIX",PREFIX,#141000} ,
{ "BYTE",BYTE,#10f000} ,{ "WYDE",WYDE,#11f000} ,{ "TETRA",TETRA,#12f000} ,{ "OCTA",OCTA,
    #13f000} ,
{ "BSPEC",BSPEC,#41400} ,{ "ESPEC",ESPEC,#141000} ,
{ "GREG",GREG,#101000} ,{ "LOCAL",LOCAL,#141800} } ;
int op_init_size;      /* the number of items in op_init_table */
```

64. ⟨ Put the `MMIX` opcodes and `MMIXAL` pseudo-ops into the trie 64 ⟩ ≡

```
op_init_size = (sizeof op_init_table)/sizeof(op_spec);
for (j = 0; j < op_init_size; j++)
{ tt = trie_search(op_root, op_init_table[j].name);
  pp = tt→sym = new_sym_node(false);
  pp→link = PREDEFINED;
  pp→equiv.h = op_init_table[j].code, pp→equiv.l = op_init_table[j].bits;
}
```

This code is used in section 61.

65. ⟨ Local variables 40 ⟩ +≡

```
register trie_node *tt;
register sym_node *pp, *qq;
```

66. ⟨ Put the special register names into the trie 66 ⟩ ≡

```
for (j = 0; j < 32; j++)
{ tt = trie_search(trie_root, special_name[j]);
  pp = tt→sym = new_sym_node(false);
  pp→link = PREDEFINED;
  pp→equiv.l = j;
}
```

This code is used in section 61.

67. ⟨ Global variables 27 ⟩ +≡

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

68. ⟨ Type definitions 26 ⟩ +≡

```
typedef struct {
  Char *name;
  tetra h, l;
}  predef_spec;
```

69. ⟨ Global variables 27 ⟩ +≡

```
predef_spec predefs[] = { { "ROUND_CURRENT",0,0} ,{ "ROUND_OFF",0,1} ,{ "ROUND_UP",0,2} ,
    { "ROUND_DOWN",0,3} ,{ "ROUND_NEAR",0,4} ,
{ "Inf",#7ff00000,0} ,
{ "Data_Segment",#20000000,0} ,{ "Pool_Segment",#40000000,0} ,{ "Stack_Segment",
    #60000000,0} ,
{ "D_BIT",0,#80} ,{ "V_BIT",0,#40} ,{ "W_BIT",0,#20} ,{ "I_BIT",0,#10} ,{ "O_BIT",0,#08} ,
    { "U_BIT",0,#04} ,{ "Z_BIT",0,#02} ,{ "X_BIT",0,#01} ,
{ "D_Handler",0,#10} ,{ "V_Handler",0,#20} ,{ "W_Handler",0,#30} ,{ "I_Handler",0,#40} ,
    { "O_Handler",0,#50} ,{ "U_Handler",0,#60} ,{ "Z_Handler",0,#70} ,{ "X_Handler",0,
    #80} ,
{ "StdIn",0,0} ,{ "StdOut",0,1} ,{ "StdErr",0,2} ,
{ "TextRead",0,0} ,{ "TextWrite",0,1} ,{ "BinaryRead",0,2} ,{ "BinaryWrite",0,3} ,
    { "BinaryReadWrite",0,4} ,
{ "Halt",0,0} ,{ "Fopen",0,1} ,{ "Fclose",0,2} ,{ "Fread",0,3} ,{ "Fgets",0,4} ,{ "Fgetws",0,
    5} ,{ "Fwrite",0,6} ,{ "Fputs",0,7} ,{ "Fputws",0,8} ,{ "Fseek",0,9} ,{ "Ftell",0,10} } ;
int predef_size;
```

70. ⟨ Put other predefined symbols into the trie 70 ⟩ ≡
```
predef_size = (sizeof predefs)/sizeof(predef_spec);
for (j = 0; j < predef_size; j++)
{ tt = trie_search(trie_root, predefs[j].name);
  pp = tt→sym = new_sym_node(false);
  pp→link = PREDEFINED;
  pp→equiv.h = predefs[j].h, pp→equiv.l = predefs[j].l;
}
```
This code is used in section 61.

71. We place `Main` into the trie at the beginning of assembly, so that it will show up as an undefined symbol if the user specifies no starting point.

⟨ Initialize everything 29 ⟩ +≡
```
trie_search(trie_root, "Main")→sym = new_sym_node(true);
```

72. At the end of assembly we traverse the entire symbol table, visiting each symbol in lexicographic order and transmitting the trie structure to the output file. We detect any undefined future references at this time.

The order of traversal has a simple recursive pattern: To traverse the subtrie rooted at t, we

traverse t→*left*, if the left subtrie is nonempty;
visit t→*sym*, if this symbol table entry is present;
traverse t→*mid*, if the middle subtrie is nonempty;
traverse t→*right*, if the right subtrie is nonempty.

This pattern leads to a compact representation in the `mmo` file, usually requiring fewer than two bytes per trie node plus the bytes needed to encode the equivalents and serial numbers. Each node of the trie is encoded as a "master byte" followed by the encodings of the left subtrie, character, equivalent, middle subtrie, and right subtrie. The master byte is the sum of

#80, if the character occupies two bytes instead of one;
#40, if the left subtrie is nonempty;
#20, if the middle subtrie is nonempty;
#10, if the right subtrie is nonempty;
#01 to #08, if the symbol's equivalent is one to eight bytes long;
#09 to #0e, if the symbol's equivalent is 2^{61} plus one to six bytes;
#0f, if the symbol's equivalent is $0 plus one byte;

the character is omitted if the middle subtrie and the equivalent are both empty. The "equivalent" of an undefined symbol is zero, but stated as two bytes long. Symbol equivalents are followed by the serial number, represented as a sequence of one or more bytes in radix 128; the final byte of the serial number is tagged by adding 128. (Thus, serial number $2^{14} - 1$ is encoded as #7fff; serial number 2^{14} is #010080.)

73. First we prune the trie by removing all predefined symbols that the user did not redefine.

```
⟨ Subroutines 28 ⟩ +≡
  trie_node *prune ARGS((trie_node *));
  trie_node *prune(t)
      trie_node *t;
  { register int useful = 0;
    if (t→sym)
    { if (t→sym→serial) useful = 1;
      else t→sym = Λ;
    }
    if (t→left)
    { t→left = prune(t→left);
      if (t→left) useful = 1;
    }
    if (t→mid)
    { t→mid = prune(t→mid);
      if (t→mid) useful = 1;
    }
    if (t→right)
    { t→right = prune(t→right);
      if (t→right) useful = 1;
    }
    if (useful) return t;
    else return Λ;
  }
```

74. Then we output the trie by following the recursive traversal pattern.

```
⟨ Subroutines 28 ⟩ +≡
  void out_stab ARGS((trie_node *));
  void out_stab(t)
      trie_node *t;
  { register int m = 0, j;
    register sym_node *pp;
    if (t→ch > #ff) m += #80;
    if (t→left) m += #40;
    if (t→mid) m += #20;
    if (t→right) m += #10;
    if (t→sym)
    { if (t→sym→link ≡ REGISTER) m += #f;
      else if (t→sym→link ≡ DEFINED) ⟨ Encode the length of t→sym→equiv 76 ⟩
      else if (t→sym→link ∨ t→sym→serial ≡ 1) ⟨ Report an undefined symbol 79 ⟩;
    }
    mmo_byte(m);
    if (t→left) out_stab(t→left);
    if (m & #2f) ⟨ Visit t and traverse t→mid 75 ⟩;
    if (t→right) out_stab(t→right);
  }
```

75. A global variable called *sym_buf* holds all characters on middle branches to the current trie node; *sym_ptr* is the first currently unused character in *sym_buf*.

```
⟨ Visit t and traverse t→mid 75 ⟩ ≡
  { if (m & #80) mmo_byte(t→ch ≫ 8);
    mmo_byte(t→ch & #ff);
    *sym_ptr++ = (m & #80 ? '?' : t→ch);      /* Unicode? not yet */
    m &= #f; if (m ∧ t→sym→link)
    { if (listing_file) ⟨ Print symbol sym_buf and its equivalent 78 ⟩;
      if (m ≡ 15) m = 1;
      else if (m > 8) m −= 8;
      for ( ; m > 0; m−−)
        if (m > 4) mmo_byte((t→sym→equiv.h ≫ (8 * (m − 5))) & #ff);
        else mmo_byte((t→sym→equiv.l ≫ (8 * (m − 1))) & #ff);
      for (m = 0; m < 4; m++)
        if (t→sym→serial < (1 ≪ (7 * (m + 1)))) break;
      for ( ; m ≥ 0; m−−) mmo_byte(((t→sym→serial ≫ (7 * m)) & #7f) + (m ? 0 : #80));
    }
    if (t→mid) out_stab(t→mid);
    sym_ptr−−;
  }
```

This code is used in section 74.

76.

```
⟨ Encode the length of t→sym→equiv 76 ⟩ ≡
  { register tetra x;
    if ((t→sym→equiv.h & #ffff0000) ≡ #20000000) m += 8, x = t→sym→equiv.h − #20000000;
        /* data segment */
    else x = t→sym→equiv.h;
    if (x) m += 4; else x = t→sym→equiv.l;
    for (j = 1; j < 4; j++)
      if (x < (1 ≪ (8 * j))) break;
    m += j;
  }
```

This code is used in section 74.

77. We make room for symbols up to 999 bytes long. Strictly speaking, the program should check if this limit is exceeded; but really!

```
⟨ Global variables 27 ⟩ +≡
  Char sym_buf[1000];
  Char *sym_ptr;
```

78. The initial '`:`' of each fully qualified symbol is omitted here, since most users of `MMIXAL` will probably not need the `PREFIX` feature. One consequence of this omission is that the one-character symbol '`:`' itself, which is allowed by the rules of `MMIXAL`, is printed as the null string.

```
⟨ Print symbol sym_buf and its equivalent 78 ⟩ ≡
  { *sym_ptr = '\0';
    fprintf(listing_file, "␣%s␣=␣", sym_buf + 1);
    pp = t→sym;
    if (pp→link ≡ DEFINED) fprintf(listing_file, "#%08x%08x", pp→equiv.h, pp→equiv.l);
    else if (pp→link ≡ REGISTER) fprintf(listing_file, "$%03d", pp→equiv.l);
    else fprintf(listing_file, "?");
    fprintf(listing_file, "␣(%d)\n", pp→serial);
  }
```

This code is used in section 75.

79.

```
⟨ Report an undefined symbol 79 ⟩ ≡
  { *sym_ptr = (m & #80 ? '?' : t→ch);      /* Unicode? not yet */
    *(sym_ptr + 1) = '\0';
    fprintf(stderr, "undefined␣symbol:␣%s\n", sym_buf + 1);
    err_count++;
    m += 2;
  }
```

This code is used in section 74.

80.

```
⟨ Check and output the trie 80 ⟩ ≡
  op_root→mid = Λ;      /* annihilate all the opcodes */
  prune(trie_root);
  sym_ptr = sym_buf;
  if (listing_file) fprintf(listing_file, "\nSymbol␣table:\n");
  mmo_lop(lop_stab, 0, 0);
  out_stab(trie_root);
  while (mmo_ptr & 3) mmo_byte(0);
  mmo_lopp(lop_end, mmo_ptr ≫ 2);
```

This code is used in section 142.

81. Expressions. The most intricate part of the assembly process is the task of scanning and evaluating expressions in the operand field. Fortunately, `MMIXAL`'s expressions have a simple structure that can be handled easily with a stack-based approach.

Two stacks hold pending data as the operand field is scanned and evaluated. The *op_stack* contains operators that have not yet been performed; the *val_stack* contains values that have not yet been used. After an entire operand list has been scanned, the *op_stack* will be empty and the *val_stack* will hold the operand values needed to assemble the current instruction.

82. Entries on *op_stack* have one of the constant values defined here, and they have one of the precedence levels defined here.

Entries on *val_stack* have *equiv*, *link*, and *status* fields; the *link* points to a trie node if the expression is a symbol that has not yet been subjected to any operations.

```
⟨ Type definitions 26 ⟩ +≡
  typedef enum {
    negate, serialize, complement, registerize, inner_lp,
        plus, minus, times, over, frac, mod, shl, shr, and, or, xor,
        outer_lp, outer_rp, inner_rp
  } stack_op;
  typedef enum {
    zero, weak, strong, unary
  } prec;
  typedef enum {
    pure, reg_val, undefined
  } stat;
  typedef struct {
    octa equiv;      /* current value */
    trie_node *link;      /* trie reference for symbol */
    stat status;      /* pure, reg_val, or undefined */
  } val_node;
```

83.
```
#define top_op op_stack[op_ptr − 1]      /* top entry on the operator stack */
#define top_val val_stack[val_ptr − 1]      /* top entry on the value stack */
#define next_val val_stack[val_ptr − 2]      /* next-to-top entry of the value stack */
⟨ Global variables 27 ⟩ +≡
  stack_op *op_stack;      /* stack for pending operators */
  int op_ptr;      /* number of items on op_stack */
  val_node *val_stack;      /* stack for pending operands */
  int val_ptr;      /* number of items on val_stack */
  prec precedence[] = { unary, unary, unary, unary, zero,
      weak, weak, strong, strong, strong, strong, strong, strong, weak, weak,
      zero, zero, zero} ;      /* precedences of the respective stack_op values */
  stack_op rt_op;      /* newly scanned operator */
  octa acc;      /* temporary accumulator */
```

84.
```
⟨ Initialize everything 29 ⟩ +≡
  op_stack = (stack_op *) calloc(buf_size, sizeof(stack_op));
  val_stack = (val_node *) calloc(buf_size, sizeof(val_node));
  if (¬op_stack ∨ ¬val_stack) panic("No␣room␣for␣the␣stacks");
```

85. The operand field of an instruction will have been copied into a separate **Char** array called *operand_list* when we reach this part of the program.

```
⟨ Scan the operand field 85 ⟩ ≡
  p = operand_list;
  val_ptr = 0;      /* val_stack is empty */
  op_stack[0] = outer_lp, op_ptr = 1;      /* op_stack contains an "outer left parenthesis" */
  while (1)
  { ⟨ Scan opening tokens until putting something on val_stack 86 ⟩;
  scan_close: ⟨ Scan a binary operator or closing token, rt_op 97 ⟩;
    while (precedence[top_op] ≥ precedence[rt_op]) ⟨ Perform the top operation on op_stack 98 ⟩;
  hold_op: op_stack[op_ptr++] = rt_op;
  }
operands_done:
```

This code is used in section 102.

86. A comment that follows an empty operand list needs to be detected here.

```
⟨ Scan opening tokens until putting something on val_stack 86 ⟩ ≡
scan_open: if (isletter(*p)) ⟨ Scan a symbol 87 ⟩
  else if (isdigit(*p))
  { if (*(p + 1) ≡ 'F') ⟨ Scan a forward local 88 ⟩
    else if (*(p + 1) ≡ 'B') ⟨ Scan a backward local 89 ⟩
    else ⟨ Scan a decimal constant 94 ⟩;
  }  else switch (*p++)
    { case '#': ⟨ Scan a hexadecimal constant 95 ⟩; break;
    case '\'': ⟨ Scan a character constant 92 ⟩; break;
    case '\"': ⟨ Scan a string constant 93 ⟩; break;
    case '@': ⟨ Scan the current location 96 ⟩; break;
    case '-': op_stack[op_ptr++] = negate;
    case '+': goto scan_open;
    case '&': op_stack[op_ptr++] = serialize; goto scan_open;
    case '~': op_stack[op_ptr++] = complement; goto scan_open;
    case '$': op_stack[op_ptr++] = registerize; goto scan_open;
    case '(': op_stack[op_ptr++] = inner_lp; goto scan_open;
    default:
      if (p ≡ operand_list + 1)
      {      /* treat operand list as empty */
        operand_list[0] = '0', operand_list[1] = '\0', p = operand_list;
        goto scan_open;
      }
      if (*(p − 1)) derr("syntax␣error␣at␣character␣'%c'", *(p − 1));
      derr("syntax␣error␣after␣character␣'%c'", *(p − 2));
    }
```

This code is used in section 85.

87. ⟨ Scan a symbol 87 ⟩ ≡

```
{ if (*p ≡ ':') tt = trie_search(trie_root, p + 1);
  else tt = trie_search(cur_prefix, p);
  p = terminator;
symbol_found: val_ptr ++;
  pp = tt→sym;
  if (¬pp) pp = tt→sym = new_sym_node(true);
  top_val.link = tt, top_val.equiv = pp→equiv;
  if (pp→link ≡ PREDEFINED) pp→link = DEFINED;
  top_val.status = (pp→link ≡ DEFINED ? pure : pp→link ≡ REGISTER ? reg_val : undefined);
}
```

This code is used in section 86.

88. ⟨ Scan a forward local 88 ⟩ ≡

```
{ tt = &forward_local_host[*p − '0']; p += 2; goto symbol_found;
}
```

This code is used in section 86.

89. ⟨ Scan a backward local 89 ⟩ ≡

```
{ tt = &backward_local_host[*p − '0']; p += 2; goto symbol_found;
}
```

This code is used in section 86.

90. Statically allocated variables *forward_local_host*[*j*] and *backward_local_host*[*j*] masquerade as nodes of the trie.

⟨ Global variables 27 ⟩ +≡

```
trie_node forward_local_host[10], backward_local_host[10];
sym_node forward_local[10], backward_local[10];
```

91. Initially `0H`, `1H`, ..., `9H` are defined to be zero.

⟨ Initialize everything 29 ⟩ +≡

```
for (j = 0; j < 10; j++)
{ forward_local_host[j].sym = &forward_local[j];
  backward_local_host[j].sym = &backward_local[j];
  backward_local[j].link = DEFINED;
}
```

92. We have already checked to make sure that the character constant is legal.

⟨ Scan a character constant 92 ⟩ ≡

```
acc.h = 0, acc.l = *p;
p += 2;
goto constant_found;
```

This code is used in section 86.

93. ⟨ Scan a string constant 93 ⟩ ≡

```
acc.h = 0, acc.l = *p;
if (*p ≡ '\"')
{ p++;
  acc.l = 0;
  err("*null␣string␣is␣treated␣as␣zero");
} else if (*(p + 1) ≡ '\"') p += 2;
else *p = '\"', *--p = ',';
goto constant_found;
```

This code is used in section 86.

94. ⟨ Scan a decimal constant 94 ⟩ ≡

```
acc.h = 0, acc.l = *p − '0';
for (p++; isdigit(*p); p++)
{ acc = oplus(acc, shift_left(acc, 2));
  acc = incr(shift_left(acc, 1), *p − '0');
}
constant_found: val_ptr++;
top_val.link = Λ;
top_val.equiv = acc;
top_val.status = pure;
```

This code is used in section 86.

95. ⟨ Scan a hexadecimal constant 95 ⟩ ≡

```
if (¬isxdigit(*p)) err("illegal␣hexadecimal␣constant");
acc.h = acc.l = 0;
for ( ; isxdigit(*p); p++)
{ acc = incr(shift_left(acc, 4), *p − '0');
  if (*p ≥ 'a') acc = incr(acc, '0' − 'a' + 10);
  else if (*p ≥ 'A') acc = incr(acc, '0' − 'A' + 10);
}
goto constant_found;
```

This code is used in section 86.

96. ⟨ Scan the current location 96 ⟩ ≡

```
acc = cur_loc;
goto constant_found;
```

This code is used in section 86.

97. ⟨ Scan a binary operator or closing token, *rt_op* 97 ⟩ ≡

```
switch (*p++)
{ case '+': rt_op = plus; break;
case '-': rt_op = minus; break;
case '*': rt_op = times; break;
case '/': if (*p ≠ '/') rt_op = over;
  else p++, rt_op = frac; break;
case '%': rt_op = mod; break;
case '<': rt_op = shl; goto sh_check;
case '>': rt_op = shr;
sh_check: p++; if (*(p − 1) ≡ *(p − 2)) break;
  derr("syntax␣error␣at␣‘%c’", *(p − 2));
case '&': rt_op = and; break;
case '|': rt_op = or; break;
case '^': rt_op = xor; break;
case ')': rt_op = inner_rp; break;
case '\0': case ',': rt_op = outer_rp; break;
default: derr("syntax␣error␣at␣‘%c’", *(p − 1));
}
```

This code is used in section 85.

98. ⟨ Perform the top operation on *op_stack* 98 ⟩ ≡

```
switch (op_stack[−−op_ptr])
{ case inner_lp: if (rt_op ≡ inner_rp) goto scan_close;
  err("*missing␣right␣parenthesis"); break;
case outer_lp: if (rt_op ≡ outer_rp)
  { if (top_val.status ≡ reg_val ∧ (top_val.equiv.l > #ff ∨ top_val.equiv.h))
    { err("*register␣number␣too␣large,␣will␣be␣reduced␣mod␣256");
      top_val.equiv.h = 0, top_val.equiv.l &= #ff;
    }
    if (¬*(p − 1)) goto operands_done;
    else rt_op = outer_lp; goto hold_op;      /* comma */
  } else
  { op_ptr++;
    err("*missing␣left␣parenthesis");
    goto scan_close;
  }
⟨ Cases for unary operators 100 ⟩
⟨ Cases for binary operators 99 ⟩
}
```

This code is used in section 85.

99. Now we come to the part where equivalents are changed by unary or binary operators found in the expression being scanned.

The most typical operator, and in some ways the fussiest one to deal with, is binary addition. Once we've written the code for this case, the other cases almost take care of themselves.

```
⟨ Cases for binary operators 99 ⟩ ≡
case plus: if (top_val.status ≡ undefined) err("cannot␣add␣an␣undefined␣quantity");
  if (next_val.status ≡ undefined) err("cannot␣add␣to␣an␣undefined␣quantity");
  if (top_val.status ≡ reg_val ∧ next_val.status ≡ reg_val) err("cannot␣add␣two␣register␣numbers");
  next_val.equiv = oplus(next_val.equiv, top_val.equiv);
fin_bin: next_val.status = (top_val.status ≡ next_val.status ? pure : reg_val);
  val_ptr −−;
delink: top_val.link = Λ; break;
```

See also section 101.

This code is used in section 98.

100. `#define` *unary_check*(*verb*) **if** (*top_val.status* ≠ *pure*) *derr*("can␣%s␣pure␣values␣only", *verb*)

```
⟨ Cases for unary operators 100 ⟩ ≡
case negate: unary_check("negate");
  top_val.equiv = ominus(zero_octa, top_val.equiv); goto delink;
case complement: unary_check("complement");
  top_val.equiv.h = ∼top_val.equiv.h, top_val.equiv.l = ∼top_val.equiv.l;
  goto delink;
case registerize: unary_check("registerize");
  top_val.status = reg_val; goto delink;
case serialize: if (¬top_val.link) err("can␣take␣serial␣number␣of␣symbol␣only");
  top_val.equiv.h = 0, top_val.equiv.l = top_val.link→sym→serial;
  top_val.status = pure; goto delink;
```

This code is used in section 98.

101. **#define** *binary_check*(*verb*)
if (*top_val*.*status* ≠ *pure* ∨ *next_val*.*status* ≠ *pure*) *derr*("can␣%s␣pure␣values␣only", *verb*)

⟨ Cases for binary operators 99 ⟩ +≡
case *minus*: **if** (*top_val*.*status* ≡ *undefined*) *err*("cannot␣subtract␣an␣undefined␣quantity");
if (*next_val*.*status* ≡ *undefined*) *err*("cannot␣subtract␣from␣an␣undefined␣quantity");
if (*top_val*.*status* ≡ *reg_val* ∧ *next_val*.*status* ≠ *reg_val*)
err("cannot␣subtract␣register␣number␣from␣pure␣value");
next_val.*equiv* = *ominus*(*next_val*.*equiv*, *top_val*.*equiv*); **goto** *fin_bin*;
case *times*: *binary_check*("multiply");
next_val.*equiv* = *omult*(*next_val*.*equiv*, *top_val*.*equiv*); **goto** *fin_bin*;
case *over*: **case** *mod*: *binary_check*("divide");
if (*top_val*.*equiv*.*l* ≡ 0 ∧ *top_val*.*equiv*.*h* ≡ 0) *err*("*division␣by␣zero");
next_val.*equiv* = *odiv*(*zero_octa*, *next_val*.*equiv*, *top_val*.*equiv*);
if (*op_stack*[*op_ptr*] ≡ *mod*) *next_val*.*equiv* = *aux*;
goto *fin_bin*;
case *frac*: *binary_check*("compute␣a␣ratio␣of");
if (*next_val*.*equiv*.*h* ≥ *top_val*.*equiv*.*h* ∧ (*next_val*.*equiv*.*l* ≥ *top_val*.*equiv*.*l* ∨ *next_val*.*equiv*.*h* >
top_val.*equiv*.*h*)) *err*("*illegal␣fraction");
next_val.*equiv* = *odiv*(*next_val*.*equiv*, *zero_octa*, *top_val*.*equiv*); **goto** *fin_bin*;
case *shl*: **case** *shr*: *binary_check*("compute␣a␣bitwise␣shift␣of");
if (*top_val*.*equiv*.*h* ∨ *top_val*.*equiv*.*l* > 63) *next_val*.*equiv* = *zero_octa*;
else if (*op_stack*[*op_ptr*] ≡ *shl*) *next_val*.*equiv* = *shift_left*(*next_val*.*equiv*, *top_val*.*equiv*.*l*);
else *next_val*.*equiv* = *shift_right*(*next_val*.*equiv*, *top_val*.*equiv*.*l*, 1);
goto *fin_bin*;
case *and*: *binary_check*("compute␣bitwise␣and␣of");
next_val.*equiv*.*h* &= *top_val*.*equiv*.*h*, *next_val*.*equiv*.*l* &= *top_val*.*equiv*.*l*;
goto *fin_bin*;
case *or*: *binary_check*("compute␣bitwise␣or␣of");
next_val.*equiv*.*h* |= *top_val*.*equiv*.*h*, *next_val*.*equiv*.*l* |= *top_val*.*equiv*.*l*;
goto *fin_bin*;
case *xor*: *binary_check*("compute␣bitwise␣xor␣of");
next_val.*equiv*.*h* ⊕= *top_val*.*equiv*.*h*, *next_val*.*equiv*.*l* ⊕= *top_val*.*equiv*.*l*;
goto *fin_bin*;

102. Assembling an instruction. Now let's move up from the expression level to the instruction level. We get to this part of the program at the beginning of a line, or after a semicolon at the end of an instruction earlier on the current line. Our current position in the buffer is the value of *buf_ptr*.

```
⟨ Process the next MMIXAL instruction or comment 102 ⟩ ≡
  p = buf_ptr; buf_ptr = "";
  ⟨ Scan the label field; goto bypass if there is none 103 ⟩;
  ⟨ Scan the opcode field; goto bypass if there is none 104 ⟩;
  ⟨ Copy the operand field 106 ⟩;
  buf_ptr = p;
  if (spec_mode ∧ ¬(op_bits & spec_bit)) derr("cannot␣use␣‘%s’␣in␣special␣mode", op_field);
  if ((op_bits & no_label_bit) ∧ lab_field[0])
  { derr("*label␣field␣of␣‘%s’␣instruction␣is␣ignored", op_field);
    lab_field[0] = '\0';
  }
  if (op_bits & align_bits) ⟨ Align the location pointer 107 ⟩;
  ⟨ Scan the operand field 85 ⟩;
  if (opcode ≡ GREG) ⟨ Allocate a global register 108 ⟩;
  if (lab_field[0]) ⟨ Define the label 109 ⟩;
  ⟨ Do the operation 116 ⟩;
bypass:
```

This code is used in section 136.

103.

```
⟨ Scan the label field; goto bypass if there is none 103 ⟩ ≡
  if (¬*p) goto bypass;
  q = lab_field;
  if (¬isspace(*p))
  { if (¬isdigit(*p) ∧ ¬isletter(*p)) goto bypass;      /* comment */
    for (*q++ = *p++; isdigit(*p) ∨ isletter(*p); p++, q++) *q = *p;
    if (*p ∧ ¬isspace(*p)) derr("label␣syntax␣error␣at␣‘%c’", *p);
  }
  *q = '\0';
  if (isdigit(lab_field[0]) ∧ (lab_field[1] ≠ 'H' ∨ lab_field[2]))
    derr("improper␣local␣label␣‘%s’", lab_field);
  for (p++; isspace(*p); p++) ;
```

This code is used in section 102.

104. We copy the opcode field to a special buffer because we might want to refer to the symbolic opcode in error messages.

```
⟨ Scan the opcode field; goto bypass if there is none 104 ⟩ ≡
  q = op_field; while (isletter(*p) ∨ isdigit(*p)) *q++ = *p++;
  *q = '\0';
  if (¬isspace(*p) ∧ *p ∧ op_field[0]) derr("opcode␣syntax␣error␣at␣‘%c’", *p);
  pp = trie_search(op_root, op_field)→sym;
  if (¬pp)
  { if (op_field[0]) derr("unknown␣operation␣code␣‘%s’", op_field);
    if (lab_field[0]) derr("*no␣opcode;␣label␣‘%s’␣will␣be␣ignored", lab_field);
    goto bypass;
  }
  opcode = pp→equiv.h, op_bits = pp→equiv.l;
  while (isspace(*p)) p++;
```

This code is used in section 102.

105. ⟨ Global variables 27 ⟩ +≡

```
tetra opcode;    /* numeric code for MMIX operation or MMIXAL pseudo-op */
tetra op_bits;   /* flags describing an operator's special characteristics */
```

106. We copy the operand field to a special buffer so that we can change string constants while scanning them later.

⟨ Copy the operand field 106 ⟩ ≡

```
q = operand_list;
while (*p)
{ if (*p ≡ ';') break;
  if (*p ≡ '\'')
  { *q++ = *p++;
    if (¬*p) err("incomplete␣character␣constant");
    *q++ = *p++;
    if (*p ≠ '\'') err("illegal␣character␣constant");
  } else if (*p ≡ '\"')
  { for (*q++ = *p++; *p ∧ *p ≠ '\"'; p++, q++) *q = *p;
    if (¬*p) err("incomplete␣string␣constant");
  }
  *q++ = *p++;
  if (isspace(*p)) break;
}
while (isspace(*p)) p++;
if (*p ≡ ';') p++;
else p = "";      /* if not followed by semicolon, rest of the line is a comment */
if (q ≡ operand_list) *q++ = '0';      /* change empty operand field to '0' */
*q = '\0';
```

This code is used in section 102.

107. It is important to do the alignment in this step before defining the label or evaluating the operand field.

⟨ Align the location pointer 107 ⟩ ≡

```
{ j = (op_bits & align_bits) ≫ 16;
  acc.h = −1, acc.l = −(1 ≪ j);
  cur_loc = oand(incr(cur_loc, (1 ≪ j) − 1), acc);
}
```

This code is used in section 102.

108. ⟨ Allocate a global register 108 ⟩ ≡

```
{ if (val_stack[0].equiv.l ∨ val_stack[0].equiv.h)
  { for (j = greg; j < 255; j++)
      if (greg_val[j].l ≡ val_stack[0].equiv.l ∧ greg_val[j].h ≡ val_stack[0].equiv.h)
      { cur_greg = j;
        goto got_greg;
      }
  }
  if (greg ≡ 32) err("too many global registers");
  greg--;
  greg_val[greg] = val_stack[0].equiv; cur_greg = greg;
got_greg: ;
}
```

This code is used in section 102.

109. If the label is, say `2H`, we will already have used the old value of `2B` when evaluating the operands. Furthermore, an operand of `2F` will have been treated as undefined, which it still is.

Symbols can be defined more than once, but only if each definition gives them the same equivalent value.

A warning message is given when a predefined symbol is being redefined, if its predefined value has already been used.

⟨ Define the label 109 ⟩ ≡

```
{ sym_node *new_link = DEFINED;
  acc = cur_loc;
  if (opcode ≡ IS)
  { cur_loc = val_stack[0].equiv;
    if (val_stack[0].status ≡ reg_val) new_link = REGISTER;
  } else if (opcode ≡ GREG) cur_loc.h = 0, cur_loc.l = cur_greg, new_link = REGISTER;
  ⟨ Find the symbol table node, pp 111 ⟩;
  if (pp→link ≡ DEFINED ∨ pp→link ≡ REGISTER)
  { if (pp→equiv.l ≠ cur_loc.l ∨ pp→equiv.h ≠ cur_loc.h ∨ pp→link ≠ new_link)
    { if (pp→serial) derr("symbol '%s' is already defined", lab_field);
      pp→serial = ++serial_number;
      derr("*redefinition of predefined symbol '%s'", lab_field);
    }
  } else if (pp→link ≡ PREDEFINED) pp→serial = ++serial_number;
  else if (pp→link)
  { if (new_link ≡ REGISTER) err("future reference cannot be to a register");
    do ⟨ Fix prior references to this label 112 ⟩ while (pp→link);
  }
  if (isdigit(lab_field[0])) pp = &backward_local[lab_field[0] - '0'];
  pp→equiv = cur_loc; pp→link = new_link;
  ⟨ Fix references that might be in the val_stack 110 ⟩;
  if (listing_file ∧ (opcode ≡ IS ∨ opcode ≡ LOC)) ⟨ Make special listing to show the label equivalent 115 ⟩;
  cur_loc = acc;
}
```

This code is used in section 102.

110. ⟨ Fix references that might be in the *val_stack* 110 ⟩ ≡

```
if (¬isdigit(lab_field[0]))
  for (j = 0; j < val_ptr; j++)
    if (val_stack[j].status ≡ undefined ∧ val_stack[j].link→sym ≡ pp)
    { val_stack[j].status = (new_link ≡ REGISTER ? reg_val : pure);
      val_stack[j].equiv = cur_loc;
    }
```

This code is used in section 109.

111. ⟨ Find the symbol table node, *pp* 111 ⟩ ≡

```
if (isdigit(lab_field[0])) pp = &forward_local[lab_field[0] − '0'];
else
{ if (lab_field[0] ≡ ':') tt = trie_search(trie_root, lab_field + 1);
  else tt = trie_search(cur_prefix, lab_field);
  pp = tt→sym;
  if (¬pp) pp = tt→sym = new_sym_node(true);
}
```

This code is used in section 109.

112. ⟨ Fix prior references to this label 112 ⟩ ≡

```
{ qq = pp→link;
  pp→link = qq→link;
  mmo_loc( );
  if (qq→serial ≡ fix_o) ⟨ Fix a future reference from an octabyte 113 ⟩
  else ⟨ Fix a future reference from a relative address 114 ⟩;
  recycle_fixup(qq);
}
```

This code is used in section 109.

113. ⟨ Fix a future reference from an octabyte 113 ⟩ ≡

```
{ if (qq→equiv.h & #ffffff)
  { mmo_lop(lop_fixo, 0, 2);
    mmo_tetra(qq→equiv.h);
  } else mmo_lop(lop_fixo, qq→equiv.h ≫ 24, 1);
  mmo_tetra(qq→equiv.l);
}
```

This code is used in section 112.

114. ⟨ Fix a future reference from a relative address 114 ⟩ ≡

```
{ octa o;
  o = ominus(cur_loc, qq→equiv);
  if (o.l & 3) dderr("*relative␣address␣in␣location␣#%08x%08x␣not␣divisible␣by␣4", qq→equiv.h,
        qq→equiv.l);
  o = shift_right(o, 2, 0); k = 0;
  if (o.h ≡ 0)
    if (o.l < #10000) mmo_lopp(lop_fixr, o.l);
    else if (qq→serial ≡ fix_xyz ∧ o.l < #1000000)
    { mmo_lop(lop_fixrx, 0, 24); mmo_tetra(o.l);
    } else k = 1;
  else if (o.h ≡ #ffffffff)
    if (qq→serial ≡ fix_xyz ∧ o.l ≥ #ff000000)
    { mmo_lop(lop_fixrx, 0, 24); mmo_tetra(o.l & #1ffffff);
    } else if (qq→serial ≡ fix_yz ∧ o.l ≥ #ffff0000)
    { mmo_lop(lop_fixrx, 0, 16); mmo_tetra(o.l & #100ffff);
    } else k = 1;
  else k = 1;
  if (k)
    dderr("relative␣address␣in␣location␣#%08x%08x␣is␣too␣far␣away", qq→equiv.h, qq→equiv.l);
}
```

This code is used in section 112.

115. ⟨ Make special listing to show the label equivalent 115 ⟩ ≡

```
if (new_link ≡ DEFINED)
{ fprintf(listing_file, "(%08x%08x)", cur_loc.h, cur_loc.l);
  flush_listing_line("␣");
} else
{ fprintf(listing_file, "($%03d)", cur_loc.l & #ff);
  flush_listing_line("␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣");
}
```

This code is used in section 109.

116. ⟨ Do the operation 116 ⟩ ≡

```
future_bits = 0;
if (op_bits & many_arg_bit) ⟨ Do a many-operand operation 117 ⟩
else switch (val_ptr)
  { case 1: if (¬(op_bits & one_arg_bit))
      derr("opcode␣'%s'␣needs␣more␣than␣one␣operand", op_field);
    ⟨ Do a one-operand operation 129 ⟩;
  case 2: if (¬(op_bits & two_arg_bit))
      if (op_bits & one_arg_bit) derr("opcode␣'%s'␣must␣not␣have␣two␣operands", op_field)
      else derr("opcode␣'%s'␣must␣have␣more␣than␣two␣operands", op_field);
    if ((op_bits & (three_arg_bit + mem_bit)) ≡ three_arg_bit) goto make_two_three;
    ⟨ Do a two-operand operation 124 ⟩;
  make_two_three: val_stack[2] = val_stack[1], val_ptr = 3;
    val_stack[1].equiv = zero_octa, val_stack[1].link = Λ, val_stack[1].status = pure;
      /* insert 0 as the second operand */
  case 3: if (¬(op_bits & three_arg_bit))
      derr("opcode␣'%s'␣must␣not␣have␣three␣operands", op_field);
    ⟨ Do a three-operand operation 119 ⟩;
  default: derr("too␣many␣operands␣for␣opcode␣'%s'", op_field);
  }
```

This code is used in section 102.

117. The many-operand operators are `BYTE`, `WYDE`, `TETRA`, and `OCTA`.

⟨ Do a many-operand operation 117 ⟩ ≡

```
for (j = 0; j < val_ptr; j++)
{ ⟨ Deal with cases where val_stack[j] is impure 118 ⟩;
  k = 1 ≪ (opcode − BYTE);
  if ((val_stack[j].equiv.h ∧ opcode < OCTA) ∨
      (val_stack[j].equiv.l > #ffff ∧ opcode < TETRA) ∨
      (val_stack[j].equiv.l > #ff ∧ opcode < WYDE))
    if (k ≡ 1) err("*constant␣doesn't␣fit␣in␣one␣byte")
    else derr("*constant␣doesn't␣fit␣in␣%d␣bytes", k);
  if (k < 8) assemble(k, val_stack[j].equiv.l, 0);
  else if (val_stack[j].status ≡ undefined) assemble(4, 0, #f0), assemble(4, 0, #f0);
  else assemble(4, val_stack[j].equiv.h, 0), assemble(4, val_stack[j].equiv.l, 0);
}
```

This code is used in section 116.

118. ⟨ Deal with cases where val_stack[j] is impure 118 ⟩ ≡

```
if (val_stack[j].status ≡ reg_val) err("*register␣number␣used␣as␣a␣constant")
else if (val_stack[j].status ≡ undefined)
{ if (opcode ≠ OCTA) err("undefined␣constant");
  pp = val_stack[j].link→sym;
  qq = new_sym_node(false);
  qq→link = pp→link;
  pp→link = qq;
  qq→serial = fix_o;
  qq→equiv = cur_loc;
}
```

This code is used in section 117.

119. ⟨ Do a three-operand operation 119 ⟩ ≡
⟨ Do the Z field 121 ⟩;
⟨ Do the Y field 122 ⟩;
assemble_X: ⟨ Do the X field 123 ⟩;
assemble_inst: *assemble*(4, (*opcode* ≪ 24) + *xyz*, *future_bits*);
break;
This code is used in section 116.

120. Individual fields of an instruction are placed into global variables *z*, *y*, *x*, *yz*, and/or *xyz*.

⟨ Global variables 27 ⟩ +≡
tetra *z*, *y*, *x*, *yz*, *xyz*; /* pieces for assembly */
int *future_bits*; /* places where there are future references */

121. ⟨ Do the Z field 121 ⟩ ≡
if (*val_stack*[2].*status* ≡ *undefined*) *err*("Z␣field␣is␣undefined");
if (*val_stack*[2].*status* ≡ *reg_val*)
{ **if** (¬(*op_bits* & (*immed_bit* + *zr_bit* + *zar_bit*)))
derr("*Z␣field␣of␣‘%s’␣should␣not␣be␣a␣register␣number", *op_field*);
} **else if** (*op_bits* & *immed_bit*) *opcode*++; /* immediate */
else if (*op_bits* & *zr_bit*) *derr*("*Z␣field␣of␣‘%s’␣should␣be␣a␣register␣number", *op_field*);
if (*val_stack*[2].*equiv*.*h* ∨ *val_stack*[2].*equiv*.*l* > #ff) *err*("*Z␣field␣doesn't␣fit␣in␣one␣byte");
z = *val_stack*[2].*equiv*.*l* & #ff;
This code is used in section 119.

122. ⟨ Do the Y field 122 ⟩ ≡
if (*val_stack*[1].*status* ≡ *undefined*) *err*("Y␣field␣is␣undefined");
if (*val_stack*[1].*status* ≡ *reg_val*)
{ **if** (¬(*op_bits* & (*yr_bit* + *yar_bit*)))
derr("*Y␣field␣of␣‘%s’␣should␣not␣be␣a␣register␣number", *op_field*);
} **else if** (*op_bits* & *yr_bit*) *derr*("*Y␣field␣of␣‘%s’␣should␣be␣a␣register␣number", *op_field*);
if (*val_stack*[1].*equiv*.*h* ∨ *val_stack*[1].*equiv*.*l* > #ff) *err*("*Y␣field␣doesn't␣fit␣in␣one␣byte");
y = *val_stack*[1].*equiv*.*l* & #ff; *yz* = (*y* ≪ 8) + *z*;
This code is used in section 119.

123. ⟨ Do the X field 123 ⟩ ≡
if (*val_stack*[0].*status* ≡ *undefined*) *err*("X␣field␣is␣undefined");
if (*val_stack*[0].*status* ≡ *reg_val*)
{ **if** (¬(*op_bits* & (*xr_bit* + *xar_bit*)))
derr("*X␣field␣of␣‘%s’␣should␣not␣be␣a␣register␣number", *op_field*);
} **else if** (*op_bits* & *xr_bit*) *derr*("*X␣field␣of␣‘%s’␣should␣be␣a␣register␣number", *op_field*);
if (*val_stack*[0].*equiv*.*h* ∨ *val_stack*[0].*equiv*.*l* > #ff) *err*("*X␣field␣doesn't␣fit␣in␣one␣byte");
x = *val_stack*[0].*equiv*.*l* & #ff; *xyz* = (*x* ≪ 16) + *yz*;
This code is used in section 119.

124. ⟨ Do a two-operand operation 124 ⟩ ≡

```
if (val_stack[1].status ≡ undefined)
{ if (op_bits & rel_addr_bit) ⟨ Assemble YZ as a future reference and goto assemble_X 125 ⟩
  else err("YZ field is undefined");
} else if (val_stack[1].status ≡ reg_val)
{ if (¬(op_bits & (immed_bit + yzr_bit + yzar_bit)))
    derr("*YZ field of '%s' should not be a register number", op_field);
  if (opcode ≡ SET) val_stack[1].equiv.l ≪= 8, opcode = #c1;      /* change to OR */
  else if (op_bits & mem_bit) val_stack[1].equiv.l ≪= 8, opcode++;      /* silently append ,0 */
} else
{     /* val_stack[1].status ≡ pure */
  if (op_bits & mem_bit) ⟨ Assemble YZ as a memory address and goto assemble_X 127 ⟩;
  if (opcode ≡ SET) opcode = #e3;      /* change to SETL */
  else if (op_bits & immed_bit) opcode++;      /* immediate */
  else if (op_bits & yzr_bit)
  { derr("*YZ field of '%s' should be a register number", op_field);
  }
  if (op_bits & rel_addr_bit) ⟨ Assemble YZ as a relative address and goto assemble_X 126 ⟩;
}
if (val_stack[1].equiv.h ∨ val_stack[1].equiv.l > #ffff) err("*YZ field doesn't fit in two bytes");
yz = val_stack[1].equiv.l & #ffff;
goto assemble_X;
```

This code is used in section 116.

125. ⟨ Assemble YZ as a future reference and goto *assemble_X* 125 ⟩ ≡

```
{ pp = val_stack[1].link→sym;
  qq = new_sym_node(false);
  qq→link = pp→link;
  pp→link = qq;
  qq→serial = fix_yz;
  qq→equiv = cur_loc;
  yz = 0;
  future_bits = #c0;
  goto assemble_X;
}
```

This code is used in section 124.

126. ⟨ Assemble YZ as a relative address and **goto** *assemble_X* 126 ⟩ ≡

```
{ octa source, dest;
  if (val_stack[1].equiv.l & 3) err("*relative␣address␣is␣not␣divisible␣by␣4");
  source = shift_right(cur_loc, 2, 0);
  dest = shift_right(val_stack[1].equiv, 2, 0);
  acc = ominus(dest, source);
  if (¬(acc.h & #80000000))
  { if (acc.l > #ffff ∨ acc.h) err("relative␣address␣is␣more␣than␣#ffff␣tetrabytes␣forward");
  } else
  { acc = incr(acc, #10000);
    opcode++;
    if (acc.l > #ffff ∨ acc.h)
      err("relative␣address␣is␣more␣than␣#10000␣tetrabytes␣backward");
  }
  yz = acc.l;
  goto assemble_X;
}
```

This code is used in section 124.

127. ⟨ Assemble YZ as a memory address and **goto** *assemble_X* 127 ⟩ ≡

```
{ octa o;
  o = val_stack[1].equiv, k = 0;
  for (j = greg; j < 255; j++)
    if (greg_val[j].h ∨ greg_val[j].l)
    { acc = ominus(val_stack[1].equiv, greg_val[j]);
      if (acc.h ≤ o.h ∧ (acc.l ≤ o.l ∨ acc.h < o.h)) o = acc, k = j;
    }
  if (o.l ≤ #ff ∧ ¬o.h ∧ k) yz = (k ≪ 8) + o.l, opcode++;
  else if (¬expanding) err("no␣base␣address␣is␣close␣enough␣to␣the␣address␣A")
  else ⟨ Assemble instructions to put supplementary data in $255 128 ⟩;
  goto assemble_X;
}
```

This code is used in section 124.

128. **#define** SETH #e0
#define SETL #e3
#define ORH #e8
#define ORL #eb

⟨ Assemble instructions to put supplementary data in $255 128 ⟩ ≡

```
{ for (j = SETH; j ≤ ORL; j++)
  { switch (j & 3)
    { case 0: yz = o.h ≫ 16; break;      /* SETH */
    case 1: yz = o.h & #ffff; break;      /* SETMH or ORMH */
    case 2: yz = o.l ≫ 16; break;      /* SETML or ORML */
    case 3: yz = o.l & #ffff; break;      /* SETL or ORL */
    }
    if (yz ∨ j ≡ SETL)
    { assemble(4, (j ≪ 24) + (255 ≪ 16) + yz, 0);
      j |= ORH;
    }
  }
  if (k) yz = (k ≪ 8) + 255;      /* Y = $k, Z = $255 */
  else yz = 255 ≪ 8, opcode++;      /* Y = $255, Z = 0 */
}
```

This code is used in section 127.

129. ⟨ Do a one-operand operation 129 ⟩ ≡

```
if (val_stack[0].status ≡ undefined)
{ if (op_bits & rel_addr_bit) ⟨ Assemble XYZ as a future reference and goto assemble_inst 130 ⟩
  else if (opcode ≠ PREFIX) err("the␣operand␣is␣undefined");
} else if (val_stack[0].status ≡ reg_val)
{ if (¬(op_bits & (xyzr_bit + xyzar_bit)))
    derr("*operand␣of␣'%s'␣should␣not␣be␣a␣register␣number", op_field);
} else
{     /* val_stack[0].status ≡ pure */
  if (op_bits & xyzr_bit) derr("*operand␣of␣'%s'␣should␣be␣a␣register␣number", op_field);
  if (op_bits & rel_addr_bit) ⟨ Assemble XYZ as a relative address and goto assemble_inst 131 ⟩;
}
if (opcode > #ff) ⟨ Do a pseudo-operation and goto bypass 132 ⟩;
if (val_stack[0].equiv.h ∨ val_stack[0].equiv.l > #ffffff)
  err("*XYZ␣field␣doesn't␣fit␣in␣three␣bytes");
xyz = val_stack[0].equiv.l & #ffffff;
goto assemble_inst;
```

This code is used in section 116.

130. ⟨ Assemble XYZ as a future reference and **goto** *assemble_inst* 130 ⟩ ≡

```
{ pp = val_stack[0].link→sym;
  qq = new_sym_node(false);
  qq→link = pp→link;
  pp→link = qq;
  qq→serial = fix_xyz;
  qq→equiv = cur_loc;
  xyz = 0;
  future_bits = #e0;
  goto assemble_inst;
}
```

This code is used in section 129.

131. ⟨ Assemble XYZ as a relative address and **goto** *assemble_inst* 131 ⟩ ≡

```
{ octa source, dest;
  if (val_stack[0].equiv.l & 3) err("*relative␣address␣is␣not␣divisible␣by␣4");
  source = shift_right(cur_loc, 2, 0);
  dest = shift_right(val_stack[0].equiv, 2, 0);
  acc = ominus(dest, source);
  if (¬(acc.h & #80000000))
  { if (acc.l > #ffffff ∨ acc.h)
      err("relative␣address␣is␣more␣than␣#ffffff␣tetrabytes␣forward");
  } else
  { acc = incr(acc, #1000000);
    opcode++;
    if (acc.l > #ffffff ∨ acc.h)
      err("relative␣address␣is␣more␣than␣#1000000␣tetrabytes␣backward");
  }
  xyz = acc.l;
  goto assemble_inst;
}
```

This code is used in section 129.

132. ⟨ Do a pseudo-operation and **goto** *bypass* 132 ⟩ ≡

```
switch (opcode)
{ case LOC: cur_loc = val_stack[0].equiv;
case IS: goto bypass;
case PREFIX: if (¬val_stack[0].link) err("not a valid prefix");
  cur_prefix = val_stack[0].link; goto bypass;
case GREG: if (listing_file) ⟨ Make listing for GREG 134 ⟩;
  goto bypass;
case LOCAL: if (val_stack[0].equiv.l > lreg) lreg = val_stack[0].equiv.l;
  if (listing_file)
  { fprintf(listing_file, "($%03d)", val_stack[0].equiv.l);
    flush_listing_line("              ");
  }
  goto bypass;
case BSPEC: if (val_stack[0].equiv.l > #ffff ∨ val_stack[0].equiv.h)
    err("*operand of 'BSPEC' doesn't fit in two bytes");
  mmo_loc( ); mmo_sync( );
  mmo_lopp(lop_spec, val_stack[0].equiv.l);
  spec_mode = true; spec_mode_loc = 0; goto bypass;
case ESPEC: spec_mode = false; goto bypass;
}
```

This code is used in section 129.

133. ⟨ Global variables 27 ⟩ +≡

```
octa greg_val[256];      /* initial values of global registers */
```

134. ⟨ Make listing for GREG 134 ⟩ ≡

```
if (val_stack[0].equiv.l ∨ val_stack[0].equiv.h)
{ fprintf(listing_file, "($%03d=#%08x", cur_greg, val_stack[0].equiv.h);
  flush_listing_line("    ");
  fprintf(listing_file, "         %08x)", val_stack[0].equiv.l);
  flush_listing_line(" ");
} else
{ fprintf(listing_file, "($%03d)", cur_greg);
  flush_listing_line("              ");
}
```

This code is used in section 132.

135. Running the program. On a UNIX-like system, the command

```
mmixal [options] sourcefilename
```

will assemble the `MMIXAL` program in file `sourcefilename`, writing any error messages on the standard error file. (Nothing is written to the standard output.) The options, which may appear in any order, are:

- `-o objectfilename` Send the output to a binary file called `objectfilename`. If no `-o` specification is given, the object file name is obtained from the input file name by changing the final letter from '`s`' to '`o`', or by appending '`.mmo`' if `sourcefilename` doesn't end with `s`.
- `-l listingname` Output a listing of the assembled input and output to a text file called `listingname`.
- `-x` Expand memory-oriented commands that cannot be assembled as single instructions, by assembling auxiliary instructions that make temporary use of global register $255.
- `-b bufsize` Allow up to `bufsize` characters per line of input.

136. Here, finally, is the overall structure of this program.

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>
#include <time.h>
  ⟨ Preprocessor definitions 31 ⟩
  ⟨ Type definitions 26 ⟩
  ⟨ Global variables 27 ⟩
  ⟨ Subroutines 28 ⟩
  int main(argc, argv)
      int argc; char *argv[];
  { register int j, k;      /* all-purpose integers */
    ⟨ Local variables 40 ⟩;
    ⟨ Process the command line 137 ⟩;
    ⟨ Initialize everything 29 ⟩;
    while (1)
    { ⟨ Get the next line of input text, or break if the input has ended 34 ⟩;
      while (1)
      { ⟨ Process the next MMIXAL instruction or comment 102 ⟩;
        if (¬*buf_ptr) break;
      }
      if (listing_file)
      { if (listing_bits) listing_clear( );
        else if (¬line_listed) flush_listing_line("␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣");
      }
    }
    ⟨ Finish the assembly 142 ⟩;
  }
```

137. The space after `"-b"` is optional, because MMIX-SIM does not use a space in this context.

```
⟨ Process the command line 137 ⟩ ≡
  for (j = 1; j < argc − 1 ∧ argv[j][0] ≡ '-'; j++)
    if (¬argv[j][2])
    { if (argv[j][1] ≡ 'x') expanding = 1;
      else if (argv[j][1] ≡ 'o') j++, strcpy(obj_file_name, argv[j]);
      else if (argv[j][1] ≡ 'l') j++, strcpy(listing_name, argv[j]);
      else if (argv[j][1] ≡ 'b' ∧ sscanf(argv[j + 1], "%d", &buf_size) ≡ 1) j++;
      else break;
    } else if (argv[j][1] ≠ 'b' ∨ sscanf(argv[j] + 2, "%d", &buf_size) ≠ 1) break;
  if (j ≠ argc − 1)
  { fprintf(stderr, "Usage: %s %s sourcefilename\n", argv[0],
        "[-x] [-l listingname] [-b buffersize] [-o objectfilename]");
    exit(−1);
  }
  src_file_name = argv[j];
```

This code is used in section 136.

138.

```
⟨ Open the files 138 ⟩ ≡
  src_file = fopen(src_file_name, "r");
  if (¬src_file) dpanic("Can't open the source file %s", src_file_name);
  if (¬obj_file_name[0])
  { j = strlen(src_file_name);
    if (src_file_name[j − 1] ≡ 's')
    { strcpy(obj_file_name, src_file_name); obj_file_name[j − 1] = 'o';
    }
    else sprintf(obj_file_name, "%s.mmo", src_file_name);
  }
  obj_file = fopen(obj_file_name, "wb");
  if (¬obj_file) dpanic("Can't open the object file %s", obj_file_name);
  if (listing_name[0])
  { listing_file = fopen(listing_name, "w");
    if (¬listing_file) dpanic("Can't open the listing file %s", listing_name);
  }
```

This code is used in section 140.

139.

```
⟨ Global variables 27 ⟩ +≡
  char *src_file_name;      /* name of the MMIXAL input file */
  char obj_file_name[FILENAME_MAX + 1];      /* name of the binary output file */
  char listing_name[FILENAME_MAX + 1];      /* name of the optional listing file */
  FILE *src_file, *obj_file, *listing_file;
  int expanding;      /* are we expanding instructions when base address fail? */
  int buf_size;      /* maximum number of characters per line of input */
```

140.

```
⟨ Initialize everything 29 ⟩ +≡
  ⟨ Open the files 138 ⟩;
  filename[0] = src_file_name;
  filename_count = 1;
  ⟨ Output the preamble 141 ⟩;
```

141. ⟨ Output the preamble 141 ⟩ ≡
```
mmo_lop(lop_pre, 1, 1);
mmo_tetra(time(Λ));
mmo_cur_file = −1;
```
This code is used in section 140.

142. ⟨ Finish the assembly 142 ⟩ ≡
```
if (lreg ≥ greg) dpanic("Danger: Must reduce the number of GREGs by %d", lreg − greg + 1);
⟨ Output the postamble 144 ⟩;
⟨ Check and output the trie 80 ⟩;
⟨ Report any undefined local symbols 145 ⟩;
if (err_count)
{ if (err_count > 1) fprintf(stderr, "(%d errors were found.)\n", err_count);
  else fprintf(stderr, "(One error was found.)\n");
}
exit(err_count);
```
This code is used in section 136.

143. ⟨ Global variables 27 ⟩ +≡
```
int greg = 255;      /* global register allocator */
int cur_greg;        /* global register just allocated */
int lreg = 32;       /* local register allocator */
```

144. ⟨ Output the postamble 144 ⟩ ≡
```
mmo_lop(lop_post, 0, greg);
greg_val[255] = trie_search(trie_root, "Main")→sym→equiv;
for (j = greg; j < 256; j++)
{ mmo_tetra(greg_val[j].h);
  mmo_tetra(greg_val[j].l);
}
```
This code is used in section 142.

145. ⟨ Report any undefined local symbols 145 ⟩ ≡
```
for (j = 0; j < 10; j++)
  if (forward_local[j].link) err_count++, fprintf(stderr, "undefined local symbol %dF\n", j);
```
This code is used in section 142.

146. Index.

December 12, 2011 at 17:41

MMIXAL