Glulx

A 32-Bit Virtual Machine for IF

VM specification version 3.1.2

Andrew Plotkin <erkyrath@eblong.com>

The virtual machine *described* by this document is an idea, not an expression of an idea, and is therefore not copyrightable. Anyone is free to write programs that run on the Glulx VM or make use of it, including compilers, interpreters, debuggers, and so on.

This document and further Glulx information can be found at:<http://eblong.com/zarf/glulx/>

0. Introduction

Glulx is a simple solution to a fairly trivial problem. We want a virtual machine which the Inform compiler can compile to, without the increasingly annoying restrictions of the Z-machine.

Glulx does this, without much fuss. All arithmetic is 32-bit (although there are opcodes to handle 8-bit and 16-bit memory access.) Input and output are handled through the Glk API (which chops out half the Z-machine opcodes, and most of the complexity of a Z-code interpreter.) Some care has been taken to make the bytecode small, but simplicity and elbow room are considered more important — bytecode is not a majority of the bulk in current Inform games.

0.1. Why Bother?

We're buried in IF VMs already, not to mention general VMs like Java, not to mention other interpreters or bytecode systems like Perl. Do we need another one?

Well, maybe not, but Glulx is simple enough that it was easier to design and implement it than to use something else. Really.

The Inform compiler already does most of the work of translating a high-level language to bytecode. It has long since outgrown many of the IF-specific features of the Z-machine (such as the object structure.) So it makes sense to remove those features, leaving a generic VM. Furthermore, there are enough other constraints (Inform's assumption of a flat memory model, the desire to have a lightweight VM suitable for PDAs) that no existing system is really ideal. So it seems worthwhile to design a new one.

Indeed, most of the effort that has gone into this system has been modifying Inform. Glulx itself is nearly an afterthought.

0.2. Glulx and Other IF Systems

Glulx grew out of the desire to extend Inform. However, it may well be suitable as a VM for other IF systems.

Or maybe not. Since Glulx *is* so lightweight, a compiler has to be fairly complex to compile to it.

Many IF systems take the approach of a simple compiler, and a complex, high-level, IF-specific interpreter. Glulx is not suitable for this.

However, if a system wants to use a simple runtime format with 32-bit data, Glulx may be a good choice.

Note that this is entirely separate from question of the I/O layer. Glulx uses the Glk I/O API, for the sake of simplicity and portability. Any IF system can do the same. One can use Glk I/O without using the Glulx game-file format.

On the obverse, one could also extend the Glulx VM to use a different I/O system instead of Glk. One such extension is FyreVM, a commercial IF system developed by Textfyre. FyreVM is described at <http://textfyre.com/fyrevm/>.

Other extension projects, not yet solidified, are being developed by Dannii Willis. See <http://curiousdannii.github.com/if/>.

This specification does not cover FyreVM and the other projects, except to note opcodes, gestalt selectors, and iosys values that are specific to them.

0.3. Credits

Graham Nelson gets pretty much all of it. Without Inform, there would be no reason for any of this. The entirety of Glulx is fallout from my attempt to deconstruct Inform and rebuild its code generator in my own image, with Graham’s support.

1. The Machine

The Glulx machine consists of main memory, the stack, and a few registers (the program counter, the stack pointer, and the call-frame pointer.)

Main memory is a simple array of bytes, numbered from zero up. When accessing multibyte values, the most significant byte is stored first (big-endian). Multibyte values are not necessarily aligned in memory.

The stack is an array of values. It is not a part of main memory; the terp maintains it separately. The format of the stack is technically up to the implementation. However, the needs of the machine (especially the game-save format) leave about one good option. (See section 1.8, "The Save-Game Format".) One important point: the stack can be kept in either byte ordering. The program should make no assumptions about endianness on the stack. (In fact, programs should never need to care.) Values on the stack always have their natural alignment (16-bit values at even addresses, 32-bit values at multiples of four).

The stack consists of a set of call frames, one for each function in the current chain. When a function is called, a new stack frame is pushed, containing the function’s local variables. The function can then push or pull 32-bit values on top of that, to store intermediate computations.

All values are treated as unsigned integers, unless otherwise noted. Signed integers are handled with the usual two’s-complement notation. Arithmetic overflows and underflows are truncated, also as usual.

1.1. Input and Output

No input/output facilities are built into the Glulx machine itself. Instead, the machine has one or more opcodes which dispatch calls to an I/O library.

At the moment, that means Glk. All Glulx interpreters support the Glk I/O facility (via the glk opcode), and no other I/O facilities exist. However, other I/O libraries may be adapted to Glk in the future. For best behavior, a program should test for the presence of an I/O facility before using it, using the IOSystem gestalt selector (see section 2.18, "Miscellaneous").

One I/O system is set as current at any given time. This does not mean that the others are unavailable. (If the interpreter supports Glk, for example, the glk opcode will always function.) However, the basic Glulx output opcodes — streamchar, streamnum, and streamstr — always print using the current I/O system.

Every Glulx interpreter supports at least one normal I/O facility (such as Glk), and also two special facilities.

The "null" I/O system does nothing. If this is selected, all Glulx output is simply discarded. *[Silly, perhaps, but I like simple base cases.]* When the Glulx machine starts up, the null system is the current system. You must select a different one before using the streamchar, streamnum, or streamstr opcodes.

The "filter" I/O system allows the Glulx program itself to handle output. The program specifies a function when selecting this I/O system. That function is then called for every single character of output that the machine generates (via streamchar, streamnum, or streamstr). The function can output its character directly via the glk opcode (or one of the other output opcodes).

[This may all seem rather baroque, but in practice most authors can ignore it. Most programs will want to test for the Glk facility, set it to be the current output system immediately, and then leave the I/O system alone for the rest of the game. All output will then automatically be handled through Glk.]

1.2. The Memory Map

Memory is divided into several segments. The sizes of the segments are determined by constant values in the game-file header.

```
    Segment    Address (hex)

  +---------+  00000000
  | Header  |
  | - - - - |  00000024
  |         |
  |   ROM   |
  |         |
  +---------+  RAMSTART
  |         |
  |   RAM   |
  |         |
  | - - - - |  EXTSTART
  |         |
  |         |
  +---------+  ENDMEM
```

As you might expect, the section marked ROM never changes during execution; it is illegal to write there. Executable code and constant data are usually (but not necessarily) kept in ROM. Note that unlike the Z-machine, the Glulx machine’s ROM comes before RAM; the 36-byte header is part of ROM.

The boundary marked EXTSTART is a trivial gimmick for making game-files smaller. A Glulx game-file only stores the data from 0 to EXTSTART. When the terp loads it in, it allocates memory up to ENDMEM; everything above EXTSTART is initialized to zeroes. Once execution starts, there is no difference between the memory above and below EXTSTART.

For the convenience of paging interpreters, the three boundaries RAMSTART, EXTSTART, and ENDMEM must be aligned on 256-byte boundaries.

Any of the segments of memory can be zero-length, except that ROM must be at least 256 bytes long (so that the header fits in it).

1.3. The Stack

The stack pointer starts at zero, and the stack grows upward. The maximum size of the stack is determined by a constant value in the game-file header. For convenience, this must be a multiple of 256.

The stack pointer counts in bytes. If you push a 32-bit value on the stack, the pointer increases by four.

1.3.1. The Call Frame

A call frame looks like this:

```
+------------+  FramePtr
| Frame Len  |    (4 bytes)
| Locals Pos |    (4 bytes)
|            |
| Format of  |    (2*n bytes)
|     Locals |
|            |
| Padding    |    (0 or 2 bytes)
+------------+  FramePtr+LocalsPos
| Locals     |    (1, 2, or 4 bytes each)
|            |
| Padding    |    (0 to 3 bytes)
+------------+  FramePtr+FrameLen
| Values     |    (4 bytes each)
|      ....  |
+------------+  StackPtr
```

When a function begins executing, the last segment is empty (StackPtr equals FramePtr+FrameLen.) Computation can push and pull 32-bit values on the stack. It is illegal to pop back beyond the original FramePtr+FrameLen boundary.

The "locals" are a list of values which the function uses as local variables. These also include function arguments. (The first N locals can be used as the arguments to an N-argument function.) Locals can be 8, 16, or 32-bit values. They are not necessarily contiguous; padding is inserted wherever necessary to bring a value to its natural alignment (16-bit values at even addresses, 32-bit values at multiples of four).

The "format of locals" is a series of bytes, describing the arrangement of the "locals" section of the frame (from LocalsPos up to FrameLen). This information is copied directly from the header of the function being called. (See section 1.6.2, "Functions".)

Each field in this section is two bytes:

- LocalType: 1, 2, or 4, indicating a set of locals which are that many bytes each.
- LocalCount: 1 to 255, indicating how many locals of LocalType to declare.

The section is terminated by a pair of zero bytes. Another pair of zeroes is added if necessary to reach a four-byte boundary.

(Example: if a function has three 8-bit locals followed by six 16-bit locals, the format segment would contain eight bytes: (1, 3, 2, 6, 0, 0, 0, 0). The locals segment would then be 16 bytes long, with a padding byte after the third local.)

The "format of locals" information is needed by the terp in two places: when calling a function (to write in function arguments), and when saving the game (to fix byte-ordering of the locals.) The formatting is *not* enforced by the terp while a function is executing. The program is not prevented from accessing locations whose size and position don't match the formatting, or locations that overlap, or even locations in the padding between locals. However, if a program does this, the results are undefined, because the byte-ordering of locals is up to the terp. The save-game algorithm will fail, if nothing else.

[In fact, the call frame may not exist as a byte sequence during function execution. The terp is free to maintain a more structured form, as long as it generates valid save-game files, and correctly handles accesses to valid (according to the format) locals.]

[NOTE: 8-bit and 16-bit locals have never been in common use, and this spec has not been unambiguous in describing their handling. (By which I mean, what I implemented in the reference interpreter didn't match the spec.) Therefore, 8-bit and 16-bit locals are deprecated. Use of the copyb and copys opcodes with a local-variable operand is also deprecated.]

1.3.2. Call Stubs

Several different Glulx operations require the ability to jump back to a previously-saved execution state. (For example: function call/return, game-state save/restore, and exception catch/throw.)

For simplicity, all these operations store the execution state the same way — as a "call stub" on the stack. This is a block of four 32-bit values. It encodes the PC and FramePtr, and also a location to store a single 32-bit value at jump-back time. (For example, the function return value, or the game-restore success flag.)

The values are pushed on the stack in the following order (FramePtr pushed last):

```
+-----------+
| DestType  |  (4 bytes)
| DestAddr  |  (4 bytes)
| PC        |  (4 bytes)
| FramePtr  |  (4 bytes)
+-----------+
```

FramePtr is the current value of FramePtr — the stack position of the call frame of the function during which the call stub was generated.

PC is the current value of the program counter. This is the address of the instruction *after* the one which caused the call stub to be generated. (For example, for a function call, the call stub contains the address of the first instruction to execute after the function returns.)

DestType and DestAddr describe a location in which to store a result. This will occur after the operation is completed (function returned, game restored, etc). It happens after the PC and FramePtr are reloaded from the call stub, and the call stub is removed from the stack.

DestType is one of the following values:

- 0: Do not store. The result value is discarded. DestAddr should be zero.
- 1: Store in main memory. The result value is stored in the main-memory address given by DestAddr.
- 2: Store in local variable. The result value is stored in the call frame at position ((FramePtr+LocalsPos) + DestAddr). See section 1.5, "Instruction Format".
- 3: Push on stack. The result value is pushed on the stack. DestAddr should be zero.

The string-decoding mechanism complicates matters a little, since it is possible for a function to be called from inside a string, instead of another function. (See section 1.3.4, "Calling and Returning Within Strings".) The following DestType values allow this:

- 10: Resume printing a compressed (E1) string. The PC value contains the address of the byte (within the string) to continue printing in. The DestAddr value contains the bit number (0 to 7) within that byte.
- 11: Resume executing function code after a string completes. The PC value contains the program counter as usual, but the FramePtr field is ignored, since the string is printed in the same call frame as the function that executed it. DestAddr should be zero.
- 12: Resume printing a signed decimal integer. The PC value contains the integer itself. The DestAddr value contains the position of the digit to print next. (0 indicates the first digit, or the minus sign for negative integers; and so on.)
- 13: Resume printing a C-style (E0) string. The PC value contains the address of the character to print next. The DestAddr value should be zero.
- 14: Resume printing a Unicode (E2) string. The PC value contains the address of the (four-byte) character to print next. The DestAddr value should be zero.

1.3.3. Calling and Returning

When a function is called, the terp pushes a four-value call stub. (This includes the return-value destination, the PC, and the FramePtr; see section 1.3.2, "Call Stubs".) The terp then sets the FramePtr to the StackPtr, and builds a new call frame. (See section 1.3.1, "The Call Frame".) The PC moves to the first instruction of the function, and execution continues.

Function arguments can be stored in the locals of the new call frame, or pushed on the stack above the new call frame. This is determined by the type of the function; see section 1.6.2, "Functions".

When a function returns, the process is reversed. First StackPtr is set back to FramePtr, throwing away the current call frame (and any pushed values). The FramePtr and PC are popped off the stack, and then the return-value destination. The function’s return value is stored where the destination says it should be. Then execution continues at the restored PC.

(But note that a function can also return to a suspended string, as well as a suspended caller function. See section 1.3.4, "Calling and Returning Within Strings" and section 1.3.5, "Calling and Returning During Output Filtering".)

1.3.4. Calling and Returning Within Strings

Glulx uses a Huffman string-compression scheme. This allows bit sequences in strings to decode to large strings, or even function invocations which generate output. This means the streamstr opcode can invoke function calls, and we must therefore be able to represent this situation on the stack.

When the terp begins printing a string, it pushes a type-11 call stub. (This includes only the current PC. The FramePtr is included, for consistency's sake, but it will be ignored when the call stub is read back off.) The terp then starts decoding the string data. The PC now indicates the position within the string data.

If, during string decoding, the terp encounters an indirect reference to a string or function, it pushes a type-10 call stub. This includes the string-decoding PC, and the bit number within that address. It also includes the current FramePtr, which has not changed since string-printing began.

If the indirect reference is to another string, the decoding continues at the new location after the type-10 stub is pushed. However, if the reference is to a function, the usual call frame is pushed on top of the type-10 stub, and the terp returns to normal function execution.

When a string completes printing, the terp pops a call stub. This will necessarily be either a type-10 or type-11. If the former, the terp resumes string decoding at the PC address/bit number in the stub. If the latter, the topmost string is finished, and the terp resumes function execution at the stub's PC.

When a function returns, it must check to see if it was called from within a string, instead of from another function. This is the case if the call stub it pops is type-10. (The call stub cannot be type-11.) If so, the FramePtr is taken from the stub as usual; but the stub's PC is taken to refer to a string data address, with the DestAddr value being the bit number within that address. (The function's return value is discarded.) String decoding resumes from there.

[It may seem wasteful for the terp to push and pop a call stub every time a string is printed. Fortunately, in the most common case — printing a string with no indirect references at all — this can easily be optimized out. (No VM code is executed between the push and pop, so it is safe to skip them.) Similarly, when printing an unencoded (E0) string, there can be no indirect references, so it is safe to optimize away the call stub push/pop.]

1.3.5. Calling and Returning During Output Filtering

The "filter" I/O system allows the terp to call a Glulx function for each character that is printed via streamchar, streamnum, or streamstr. We must be able to represent this situation on the call stack as well.

If filtering is the current I/O system, then when the terp executes streamchar, it pushes a normal function call stub and begins executing the output function. Nothing else is required; when the function returns, execution will resume after the streamchar opcode. (A type-0 call stub is used, so the function's return value is discarded.)

The other output opcodes are more complex. When the terp executes streamnum, it pushes a type-11 call stub. As before, this records the current PC. The terp then pushes a type-12 call stub, which contains the integer being printed and the position of the next character to be printed (namely 1). It then executes the output function.

When the output function returns, the terp pops the type-12 stub and realizes that it should continue printing the integer contained therein. It pushes another type-12 stub back on the stack, indicating that the next position to print is 2, and calls the output function again.

This process continues until there are no more characters in the decimal representation of the integer. The terp then pops the type-11 stub, restores the PC, and resumes execution after the streamnum opcode.

The streamstr opcode works on the same principle, except that instead of type-12 stubs, the terp uses type-10 stubs (when interrupting an encoded string) and type-13/14 stubs (when interruping a C-style, null-terminated string of bytes/Unicode chars). Type-13 and type-14 stubs look like the others, except that they contain only the address of the next character to print; no other position or bit number is necessary.

The interaction between the filter I/O system and indirect string/function calls within encoded strings is left to the reader's imagination. *[Because I couldn't explain it if I tried. Follow the rules; they work.]*

1.4. The Header

The header is the first 36 bytes of memory. It is always in ROM, so its contents cannot change during execution. The header is organized as nine 32-bit values. (Recall that values in memory are always big-endian.)

```
+---------------+  address 0
| Magic Number  |  (4 bytes)
| Glulx Version |  (4 bytes)
| RAMSTART      |  (4 bytes)
| EXTSTART      |  (4 bytes)
| ENDMEM        |  (4 bytes)
| Stack Size    |  (4 bytes)
| Start Func    |  (4 bytes)
| Decoding Tbl  |  (4 bytes)
| Checksum      |  (4 bytes)
+---------------+
```

- Magic number: 47 6C 75 6C, which is to say ASCII 'Glul'.
- Glulx version number: The upper 16 bits stores the major version number; the next 8 bits stores the minor version number; the low 8 bits stores an even more minor version number, if any. This specification is version 3.1.2, so a game file generated to this spec would contain 00030102.
- RAMSTART: The first address which the program can write to.
- EXTSTART: The end of the game-file's stored initial memory (and therefore the length of the game file.)
- ENDMEM: The end of the program's memory map.
- Stack size: The size of the stack needed by the program.
- Address of function to execute: Execution commences by calling this function.
- Address of string-decoding table: This table is used to decode compressed strings. See section 1.6.1.3, "Compressed strings". This may be zero, indicating that no compressed strings are to be decoded. *[Note that the game can change which table the terp is using, with the setstringtbl opcode. See section 2.11, "Output".]*
- Checksum: A simple sum of the entire initial contents of memory, considered as an array of big-endian 32-bit integers. The checksum should be computed with this field set to zero.

The interpreter should validate the magic number and the Glulx version number. An interpreter which is written to version X.Y.Z of this specification should accept game files whose Glulx version between X.0.0 and X.Y.*. (That is, the major version number should match; the minor version number should be less than or equal to Y; the subminor version number does not matter.)

EXCEPTION: A version 3.* interpreter should accept version 2.0 game files. The only difference between spec 2.0 and spec 3.0 is that 2.0 lacks Unicode functionality. Therefore, an interpreter written to this version of the spec (3.1.2) should accept game files whose version is between 2.0.0 and 3.1.* (0x00020000 and 0x000301FF inclusive).

[These rules mean, in the vernacular, that minor version changes are backwards compatible, and subminor version changes are backwards and forwards compatible. If I add a feature which I expect every terp to implement (e.g. mzero and mcopy), then I bump the minor version number, and your game can use that feature without worrying about availability. If I add a feature which not all terps will implement (e.g. floating point), then I bump the subminor version number, and your game should only use the feature after doing a gestalt test for availability.]

[The header is conventionally followed by a 32-bit word which describes the layout of data in the rest of the file. This value is not *a part of the Glulx specification; it is the first ROM word after the header, not a part of the header. It is an option that compilers can insert, when generating Glulx files, to aid debuggers and decompilers.*

For Inform-generated Glulx files, this descriptive value is 49 6E 66 6F, which is to say ASCII 'Info'. There then follow several more bytes of data relevant to the Inform compiler. See the Glulx chapter of the Inform Technical Manual.]

[Note that version 2.0 (pre-Unicode) has been obsolete since 2006. There are still 2.0 game files out there, so interpreters should still support them. However, there are no 2.0-only interpreters left; so compilers may freely target 3..]*

1.5. Instruction Format

There are 2^28 Glulx opcodes, numbered from 0 to 0FFFFFFF. If this proves insufficient, more may be added in the future.

An instruction is encoded as follows:

```
+--------------+
| Opcode Num   |  (1 to 4 bytes)
|              |
| Operand      |  (two per byte)
|   Addr Modes |
|              |
| Operand Data |  (as defined by
|        ....  |      addr modes)
+--------------+
```

The opcode number OP, which can be anything up to 0FFFFFFF, may be packed into fewer than four bytes:

- 00..7F: One byte, OP
- 0000..3FFF: Two bytes, OP+8000
- 00000000..0FFFFFFF: Four bytes, OP+C0000000

Note that the length of this field can be decoded by looking at the top two bits of the first byte. Also note that, for example, 01 and 8001 and C0000001 all represent the same opcode.

The operand addressing modes are a list of fields which tell where opcode arguments are read from or written to. Each is four bits long, and they are packed two to a byte. (They occur in the same order as the arguments, low bits first. If there are an odd number, the high bits of the last byte are left zero.)

Since each addressing mode is a four-bit number, there are sixteen addressing modes. Each is associated with a fixed number of bytes in the "operand data" segment of the instruction. These bytes appear after the addressing modes, in the same order. (There is no alignment padding.)

- 0: Constant zero. (Zero bytes)
- 1: Constant, -80 to 7F. (One byte)
- 2: Constant, -8000 to 7FFF. (Two bytes)
- 3: Constant, any value. (Four bytes)
- 4: (Unused)
- 5: Contents of address 00 to FF. (One byte)
- 6: Contents of address 0000 to FFFF. (Two bytes)
- 7: Contents of any address. (Four bytes)
- 8: Value popped off stack. (Zero bytes)
- 9: Call frame local at address 00 to FF. (One byte)
- A: Call frame local at address 0000 to FFFF. (Two bytes)
- B: Call frame local at any address. (Four bytes)
- C: (Unused)
- D: Contents of RAM address 00 to FF. (One byte)
- E: Contents of RAM address 0000 to FFFF. (Two bytes)
- F: Contents of RAM, any address. (Four bytes)

Things to note:

The "constant" modes sign-extend their data into a 32-bit value; the other modes do not. This is just because negative constants occur more frequently than negative addresses.

The indirect modes (all except "constant") access 32-bit fields, either in the stack or in memory. This means four bytes starting at the given address. A few opcodes are exceptions: copyb and copys (copy byte and copy short) access 8-bit and 16-bit fields (one or two bytes starting at the given address.)

The "call frame local" modes access a field on the stack, starting at byte ((FramePtr+LocalsPos) + address). As described in section 1.3.1, "The Call Frame", this must be aligned with (and the same size as) one of the fields described in the function’s locals format. It must not point outside the range of the current function’s locals segment.

The "contents of address" modes access a field in main memory, starting at byte (addr). The "contents of RAM" modes access a field in main memory, starting at byte (RAMSTART + addr). Since the byte-ordering of main memory is well-defined, these need not have any particular alignment or position.

All address addition is truncated to 32 bits, and addresses are unsigned. So, for example, "contents of RAM" address FFFFFFFC (RAMSTART + FFFFFFFC) accesses the last 32-bit value in ROM, since it effectively subtracts 4 from RAMSTART. "Contents of address" FFFFFFFC would access the very last 32-bit value in main memory, assuming you can find a terp which handles four-gigabyte games. "Call frame local" FFFFFFFC is illegal; whether you interpret it as a negative number or a large positive number, it’s outside the current call frame’s locals segment.

Some opcodes store values as well as reading them in. Store operands use the same addressing modes, with a few exceptions:

- 8: The value is pushed into the stack, instead of being popped off.
- 3, 2, 1: These modes cannot be used, since it makes no sense to store to a constant. *[We delicately elide the subject of Fortran. And rule-based property algebras.]*
- 0: This mode means "throw the value away"; it is not stored at all.

Operands are evaluated from left to right. (This is important if there are several push/pop operands.)

1.6. Typable Objects

It is convenient for a program to store object references as 32-bit pointers, and still determine the type of a reference at run-time.

To facilitate this, structured objects in Glulx main memory follow a simple convention: the first byte indicates the type of the object.

At the moment, there are only two kinds of Glulx objects: functions and strings. A program (or compiler, or library) may declare more, but the Glulx VM does not have to know about them.

Of course, not every byte in memory is the start of the legitimate object. It is the program's responsibility to keep track of which values validly refer to typable objects.

1.6.1. Strings

Strings have a type byte of E0 (for unencoded, C-style strings), E2 (for unencoded strings of Unicode values), or E1 (for compressed strings.) Types E3 to FF are reserved for future expansion of string types.

1.6.1.1. Unencoded strings

An unencoded string consists of an E0 byte, followed by all the bytes of the string, followed by a zero byte.

1.6.1.2. Unencoded Unicode strings

An unencoded Unicode string consists of an E2 byte, followed by three padding 0 bytes, followed by the Unicode character values (each one being a four-byte integer). Finally, there is a terminating value (four 0 bytes).

```
Unencoded Unicode string
+----------------+
| Type: E2       |  (1 byte)
| Padding: 00    |  (3 bytes)
| Characters.... |  (any length, multiple of 4)
| NUL: 00000000  |  (4 bytes)
+----------------+
```

Note that the character data is not encoded in UTF-8, UTF-16, or any other peculiar encoding. It is treated as an array of 32-bit integers (which are, as always in Glulx, stored big-endian). Each integer is a Unicode code point.

1.6.1.3. Compressed strings

A compressed string consists of an E1 byte, followed by a block of Huffman-encoded data. This should be read as a stream of bits, starting with the low bit (the 1 bit) of the first byte after the E1, proceeding through the high bit (the 128 bit), and so on with succeeding bytes.

Decoding compressed strings requires looking up data in a Huffman table. The address of this table is normally found in the header. However, the program can select a different decompression table at run-time; see section 2.11, "Output".

The Huffman table is logically a binary tree. Internal nodes are branch points; leaf nodes represent printable entities. To decode a string, begin at the root node. Read one bit from the bit stream, and go to the left or right child depending on its value. Continue reading bits and branching left or right, until you reach a leaf node. Print that entity. Then jump back to the root, and repeat the process. One particular leaf node indicates the end of the string (rather than any printable entity), and when the bit stream leads you to that node, you stop.

[This is a fairly slow process, with VM memory reads and a conditional test for every bit *of the string. A terp can speed it up considerably by reading the Huffman table all at once, and caching it as native data structures. A binary tree is the obvious choice, but one can do even better (at the cost of some space) by looking up four-bit chunks at a time in a 16-branching tree.]*

[Note that decompression tables are not necessarily in ROM. This is particularly important for tables that are generated and selected at run-time. Furthermore, it is technically legal for a table in RAM to be altered at runtime — possibly even when it is the currently-selected table. Therefore, an interpreter that caches or preloads this decompression data must be careful. If it caches data from RAM, it must watch for writes to that RAM space, and invalidate its cache upon seeing such a write.]

1.6.1.4. The String-Decoding Table

The decoding table has the following format:

```
+-----------------+
| Table Length    |  (4 bytes)
| Number of Nodes |  (4 bytes)
| Root Node Addr  |  (4 bytes)
| Node Data ....  |  (table length - 12 bytes)
+-----------------+
```

The table length is measured in bytes, from the beginning of the table to the end of the last node. The node count includes both branch and leaf nodes. *[There will, of course, be an odd number of nodes, and (N+1)/2 of them will be leaves.]* The root address indicates which node is the root of the tree; it is not necessarily the first node. This is an absolute address, not an offset from the beginning of the table.

There then follow all the nodes, with no extra data before, between, or after them. They need not be in any particular order. There are several possible types of nodes, distinguished by their first byte.

```
Branch (non-leaf node)
+----------------+
| Type: 00       |  (1 byte)
| Left  (0) Node |  (4 bytes)
```

```
| Right (1) Node |  (4 bytes)
+----------------+
```

The left and right node fields are addresses (again, absolute addresses) of the nodes to go to given a 0 or 1 bit from the bit stream.

```
String terminator
+----------------+
| Type: 01       |  (1 byte)
+----------------+
```

This ends the string-decoding process.

```
Single character
+----------------+
| Type: 02       |  (1 byte)
| Character      |  (1 byte)
+----------------+
```

This prints a single character. *[The encoding scheme is the business of the I/O system; in Glk, it will be the Latin-1 character set.]*

```
C-style string
+----------------+
| Type: 03       |  (1 byte)
| Characters.... |  (any length)
| NUL: 00        |  (1 byte)
+----------------+
```

This prints an array of characters. Note that the array cannot contain a zero byte, since that is reserved to terminate the array. *[A zero byte can be printed using the single-character node type.]*

```
Single Unicode character
+----------------+
| Type: 04       |  (1 byte)
| Character      |  (4 bytes)
+----------------+
```

This prints a single Unicode character. *[To be precise, it prints a 32-bit character, which will be interpreted as Unicode if the I/O system is Glk.]*

```
C-style Unicode string
+----------------+
| Type: 05       |  (1 byte)
| Characters.... |  (any length, multiple of 4)
| NUL: 00000000  |  (4 bytes)
+----------------+
```

This prints an array of Unicode characters. Note that the array cannot contain a zero word, since that is reserved to terminate the array. Also note that, unlike an E2-encoded string object, there is no padding.

[If the Glk library is unable to handle Unicode, node types 04 and 05 are still legal. However, characters beyond FF will be printed as 3F ("?").]

```
Indirect reference
+----------------+
| Type: 08       |  (1 byte)
| Address        |  (4 bytes)
+----------------+
```

This prints a string or calls a function, which is not actually part of the decoding table. The address may refer to a location anywhere in memory (including RAM.) It must be a valid Glulx string (see section 1.6.1, "Strings") or function (see section 1.6.2, "Functions"). If it is a string, it is printed. If a function, it is called (with no arguments) and the result is discarded.

The management of the stack during an indirect string/function call is a bit tricky. See section 1.3.4, "Calling and Returning Within Strings".

```
Double-indirect reference
+----------------+
| Type: 09       |  (1 byte)
| Address        |  (4 bytes)
+----------------+
```

This is similar to the indirect-reference node, but the address refers to a four-byte field in memory, and *that* contains the address of a string or function. The extra layer of indirection can be useful. For example, if the four-byte field is in RAM, its contents can be changed during execution, pointing to a new typable object, without modifying the decoding table itself.

```
Indirect reference with arguments
+----------------+
| Type: 0A       |  (1 byte)
| Address        |  (4 bytes)
| Argument Count |  (4 bytes)
| Arguments....  |  (4*N bytes)
+----------------+

Double-indirect reference with arguments
+----------------+
| Type: 0B       |  (1 byte)
| Address        |  (4 bytes)
| Argument Count |  (4 bytes)
| Arguments....  |  (4*N bytes)
+----------------+
```

These work the same as the indirect and double-indirect nodes, but if the object found is a function, it will be called with the given argument list. If the object is a string, the arguments are ignored.

1.6.2. Functions

Functions have a type byte of C0 (for stack-argument functions) or C1 (for local-argument functions). Types C2 to DF are reserved for future expansion of function types.

A Glulx function always takes a list of 32-bit arguments, and returns exactly one 32-bit value. (If you want a function which returns no value, discard or ignore it. Store operand mode zero is convenient.)

If the type is C0, the arguments are passed on the stack, and are made available on the stack. After the function's call frame is constructed, all the argument values are pushed — last argument pushed first, first argument topmost. Then the number of arguments is pushed on top of that. All locals in the call frame itself are initialized to zero.

If the type is C1, the arguments are passed on the stack, and are written into the locals according to the "format of locals" list of the function. Arguments passed into 8-bit or 16-bit locals are truncated. It is legitimate for there to be too many or too few arguments. Extras are discarded silently; any locals left unfilled are initialized to zero.

A function has the following structure:

```
+------------+
|  C0 or C1  |  Type (1 byte)
+------------+
| Format of  |    (2*n bytes)
|     Locals |
+------------+
|  Opcodes   |
|      ....  |
+------------+
```

The locals-format list is encoded the same way it is on the stack; see section 1.3.1, "The Call Frame". This is a list of LocalType/LocalCount byte pairs, terminated by a zero/zero pair. (There is, however, no extra padding to reach four-byte alignment.)

Note that although a LocalType/LocalCount pair can only describe up to 255 locals, there is no restriction on how many locals the function can have. It is legitimate to encode several pairs in a row with the same LocalType.

Immediately following the two zero bytes, the instructions start. There is no explicit terminator for the function.

1.6.3. Other Glulx Objects

There are no other Glulx objects at this time, but type 80 to BF are reserved for future expansion. Type 00 is also reserved; it indicates "no object", and should not be used by any typable object. A null reference's type would be considered 00. (Even though byte 00000000 of main memory is not in fact 00.)

1.6.4. User-Defined Objects

Types 01 to 7F are available for use by the compiler, the library, or the program. Glulx will not use them.

[Inform uses 60 for dictionary words, and 70 for objects and classes. It reserves types 40 to 7F. Types 01 to 3F remain available for use by Inform programmers.]

1.7. Floating-Point Numbers

Glulx values are 32-bit integers, big-endian when stored in memory. To handle floating-point math, we must be able to encode float values as 32-bit values. Unsurprisingly, Glulx uses the big-endian, single-precision IEEE-754 encoding. (See <http://www.psc.edu/general/software/packages/ieee/ieee.php>.) This allows floats to be stored in memory, on the stack, in local variables, and in any other place that a 32-bit value appears.

However, float values and integer values are *not* interchangable. You cannot pass floats to the normal arithmetic opcodes, or vice versa, and expect to get meaningful answers. Always pass floats to the float opcodes and integers to the int opcodes, with the appropriate conversion opcodes to convert back and forth. (See section 2.12, "Floating-Point Math".)

Floats have limited precision; they cannot represent all real values exactly. They can't even represent all integers exactly. (Integers between -1000000 and 1000000 (hex) have exact representations. Beyond that, the rounding error can be greater than 1. But when you get into fractions, errors are possible anywhere: 1/3 cannot be stored exactly.)

Therefore, you must be careful when comparing results. A series of float operations may produce a result fractionally different from what you expect. When comparing float values, you will most often want to use the jfeq opcode, which tests whether two values are *near* each other (within a specified range).

A float value has three fields in its 32 bits, from highest (the sign bit) to lowest:

```
+---------------+
| Sign Bit (S)  |  (1 bit)
| Exponent (E)  |  (8 bits)
| Mantissa (M)  |  (23 bits)
+---------------+
```

The interpretation of the value depends on the exponent value:

- If E is FF and M is zero, the value is positive or negative infinity, depending on S. Infinite values represent overflows. (+Inf is 7F800000; -Inf is FF800000.)
- If E is FF and M is nonzero, the value is a positive or negative NaN ("not a number"), depending on S. NaN values represent arithmetic failures. (+NaN values are in the range 7F800001 to 7FFFFFFF; -NaN are FF800001 to FFFFFFFF.)
- If E is 00 and M is zero, the value is a positive or negative zero, depending on S. Zero values represent underflows, and also, you know, zero. (+0 is 00000000; -0 is 80000000.)
- If E is 00 and M is nonzero, the value is a "denormalized" number, very close to zero: plus or minus 2^(-149)*M.
- If E is anything else, the value is a "normalized" number: plus or minus 2^(E-150)*(800000+M).

[I'm using decimal exponents there amid all the hex constants. -149 is hex -95; -150 is hex -96. Sorry about that.]

The numeric formulas may look more familiar if you write them as 2^(-126)*(0.MMMM...) and 2^(E-127)*(1.MMMM...), where "0.MMMM..." is a fraction between zero and one (23 mantissa bits after the binal point) and "1.MMMM...." is a fraction beween one and two.

Some example values:

- 0.0   = 00000000 (S=0, E=00, M=0)
- 1.0   = 3F800000 (S=0, E=7F, M=0)
- -2.0 = C0000000 (S=1, E=80, M=0)
- 100.0 = 42C80000 (S=0, E=85, M=480000)
- pi   = 40490FDB (S=0, E=80, M=490FDB)
- 2*pi = 40C90FDB (S=0, E=81, M=490FDB)
- e   = 402DF854 (S=0, E=80, M=2DF854)

To give you an idea of the behavior of the special values:

- 1 / 0   = +Inf
- -1 / 0   = -Inf
- 1 / Inf = 0
- 1 / -Inf = -0
- 0 / 0   = NaN
- 2 * 0   = 0
- 2 * -0   = -0
- +Inf * 0 = NaN
- +Inf * 1 = +Inf
- +Inf + +Inf = +Inf
- +Inf * +Inf = +Inf
- +Inf - +Inf = NaN
- +Inf / +Inf = NaN

NaN is sticky; almost *any* mathematical operation involving a NaN produces NaN. (There are a few exceptions.)

However, Glulx does not guarantee *which* NaN value you will get from such operations. The underlying platform may try to encode information about what operation failed in the mantissa field of the NaN. Or, contrariwise, it may return the same value for every NaN. The sign bit, similarly, is never guaranteed. (The sign may be preserved if that’s meaningful for the failed operation, but it may not be.) You should not test for NaN by comparing to a fixed encoded value; instead, use the jisnan opcode.

1.8. The Save-Game Format

(Or, if you like, "serializing the machine state".)

This is a variant of Quetzal, the standard Z-machine save file format. (See <http://ifarchive.org/if-archive/infocom/interpreters/specification/savefile_14.txt>.)

Everything in the Quetzal specification applies, with the following exceptions:

1.8.1. Contents of Dynamic Memory

In both compressed and uncompressed form, the memory chunk (’CMem’ or ’UMem’) starts with a four-byte value, which is the current size of memory. The memory data then follows. During a restore, the size of memory is changed to this position.

The memory area to be saved does not start at address zero, but at RAMSTART. It continues to the current end of memory (which may not be the ENDMEM value in the header.) When generating or reading compressed data ('CMem' chunk), the data above EXTSTART is handled as if the game file were extended with as many zeroes as necessary.

1.8.2. Contents of the Stack

Before the stack is written out, a four-value call stub is pushed on — result destination, PC, and FramePtr. (See section 1.3.2, "Call Stubs".) Then the entire stack can be written out, with all of its values (of whatever size) transformed to big-endian. (Padding is not skipped; it's written out as the appropriate number of zero bytes.)

When the game-state is loaded back in — or, for that matter, when continuing after a game-save — the four values are read back off the stack, a result code for the operation is stored in the appropriate destination, and execution continues.

[Remember that in a call stub, the PC contains the address of the instruction after *the one being executed.]*

1.8.3. Memory Allocation Heap

If the heap is active (see section 2.9, "Memory Allocation Heap"), an allocation heap chunk is written ('MAll'). This chunk contains two four-byte values, plus two more for each extant memory block:

- Heap start address
- Number of extant blocks
- Address of first block
- Length of first block
- Address of second block
- Length of second block
- ...

The blocks need not be listed in any particular order.

If the heap is not active, the 'MAll' chunk can contain 0,0 or it may be omitted.

1.8.4. Associated Story File

The contents of the game-file identifier ('IFhd' chunk) are simply the first 128 bytes of memory. This is within ROM (since RAMSTART is at least 256), so it does not vary during play. It includes the story file length and checksum, as well as any compiler-specific information that may be stored immediately after the header.

1.8.5. State Not Saved

Some aspects of Glulx execution are not part of the save process, and therefore are not changed during a restart, restore, or restoreundo operation. The program is responsible for checking these values after a restore to see if they have (from the program's point of view) changed unexpectedly.

Examples of information which is not saved:

- Glk library state. This includes Glk opaque objects (windows, filerefs, streams). It also includes I/O state such as the current output stream, contents of windows, and cursor

positions. Accounting for Glk object changes after restore/restoreundo is tricky, but absolutely necessary.
- The protected-memory range (position, length, and whether it exists at all). Note that the *contents* of the range (if it exists) are not treated specially during saving, and are therefore saved normally.
- The random number generator's internal state.
- The I/O system mode and current string-decoding table address.

2. Dictionary of Opcodes

Opcodes are written here in the format:

```
opname L1 L2 S1
```

...where "L1" and "L2" are operands using the load addressing modes, and "S1" is an operand using the store addressing modes. (See section 1.5, "Instruction Format".)

The table of opcodes:

- 0x00: nop
- 0x10: add
- 0x11: sub
- 0x12: mul
- 0x13: div
- 0x14: mod
- 0x15: neg
- 0x18: bitand
- 0x19: bitor
- 0x1A: bitxor
- 0x1B: bitnot
- 0x1C: shiftl
- 0x1D: sshiftr
- 0x1E: ushiftr
- 0x20: jump
- 0x22: jz
- 0x23: jnz
- 0x24: jeq
- 0x25: jne
- 0x26: jlt
- 0x27: jge
- 0x28: jgt
- 0x29: jle
- 0x2A: jltu
- 0x2B: jgeu
- 0x2C: jgtu
- 0x2D: jleu
- 0x30: call
- 0x31: return
- 0x32: catch
- 0x33: throw
- 0x34: tailcall
- 0x40: copy
- 0x41: copys
- 0x42: copyb

- 0x44: sexs
- 0x45: sexb
- 0x48: aload
- 0x49: aloads
- 0x4A: aloadb
- 0x4B: aloadbit
- 0x4C: astore
- 0x4D: astores
- 0x4E: astoreb
- 0x4F: astorebit
- 0x50: stkcount
- 0x51: stkpeek
- 0x52: stkswap
- 0x53: stkroll
- 0x54: stkcopy
- 0x70: streamchar
- 0x71: streamnum
- 0x72: streamstr
- 0x73: streamunichar
- 0x100: gestalt
- 0x101: debugtrap
- 0x102: getmemsize
- 0x103: setmemsize
- 0x104: jumpabs
- 0x110: random
- 0x111: setrandom
- 0x120: quit
- 0x121: verify
- 0x122: restart
- 0x123: save
- 0x124: restore
- 0x125: saveundo
- 0x126: restoreundo
- 0x127: protect
- 0x130: glk
- 0x140: getstringtbl
- 0x141: setstringtbl
- 0x148: getiosys
- 0x149: setiosys
- 0x150: linearsearch
- 0x151: binarysearch
- 0x152: linkedsearch
- 0x160: callf
- 0x161: callfi
- 0x162: callfii
- 0x163: callfiii
- 0x170: mzero
- 0x171: mcopy
- 0x178: malloc
- 0x179: mfree
- 0x180: accelfunc
- 0x181: accelparam
- 0x190: numtof
- 0x191: ftonumz

- 0x192: ftonumn
- 0x198: ceil
- 0x199: floor
- 0x1A0: fadd
- 0x1A1: fsub
- 0x1A2: fmul
- 0x1A3: fdiv
- 0x1A4: fmod
- 0x1A8: sqrt
- 0x1A9: exp
- 0x1AA: log
- 0x1AB: pow
- 0x1B0: sin
- 0x1B1: cos
- 0x1B2: tan
- 0x1B3: asin
- 0x1B4: acos
- 0x1B5: atan
- 0x1B6: atan2
- 0x1C0: jfeq
- 0x1C1: jfne
- 0x1C2: jflt
- 0x1C3: jfle
- 0x1C4: jfgt
- 0x1C5: jfge
- 0x1C8: jisnan
- 0x1C9: jisinf

Opcodes 0x1000 to 0x10FF are reserved for use by FyreVM. Opcodes 0x1100 to 0x11FF are reserved for extension projects by Dannii Willis. Opcodes 0x1200 to 0x12FF are reserved for iOS extension features by Andrew Plotkin. These are not documented here. Opcodes 0x7900 to 0x79FF are (apparently) reserved for experimental features in the Git interpreter. See section 0.2, "Glulx and Other IF Systems".

2.1. Integer Math

```
add L1 L2 S1
```

Add L1 and L2, using standard 32-bit addition. Truncate the result to 32 bits if necessary. Store the result in S1.

```
sub L1 L2 S1
```

Compute (L1 - L2), and store the result in S1.

```
mul L1 L2 S1
```

Compute (L1 * L2), and store the result in S1. Truncate the result to 32 bits if necessary.

```
div L1 L2 S1
```

Compute (L1 / L2), and store the result in S1. This is signed integer division.

```
mod L1 L2 S1
```

Compute (L1 % L2), and store the result in S1. This is the remainder from signed integer division.

In division and remainer, signs are annoying. Rounding is towards zero. The sign of a remainder equals the sign of the dividend. It is always true that (A / B) * B + (A % B) == A. Some examples (in decimal):

```
 11 /  2 =  5
-11 /  2 = -5
 11 / -2 = -5
-11 / -2 =  5
 13 %  5 =  3
-13 %  5 = -3
 13 % -5 =  3
-13 % -5 = -3
```

```
neg L1 S1
```

Compute the negative of L1.

```
bitand L1 L2 S1
```

Compute the bitwise AND of L1 and L2.

```
bitor L1 L2 S1
```

Compute the bitwise OR of L1 and L2.

```
bitxor L1 L2 S1
```

Compute the bitwise XOR of L1 and L2.

```
bitnot L1 S1
```

Compute the bitwise negation of L1.

```
shiftl L1 L2 S1
```

Shift the bits of L1 to the left (towards more significant bits) by L2 places. The bottom L2 bits are filled in with zeroes. If L2 is 32 or more, the result is always zero.

```
ushiftr L1 L2 S1
```

Shift the bits of L1 to the right by L2 places. The top L2 bits are filled in with zeroes. If L2 is 32 or more, the result is always zero.

```
sshiftr L1 L2 S1
```

Shift the bits of L1 to the right by L2 places. The top L2 bits are filled in with copies of the top bit of L1. If L2 is 32 or more, the result is always zero or FFFFFFFF, depending on the top bit of L1.

Notes on the shift opcodes: If L2 is zero, the result is always equal to L1. L2 is considered unsigned, so 80000000 or greater is "more than 32".

2.2. Branches

All branches (except jumpabs) specify their destinations with an offset value. The actual destination address of the branch is computed as (Addr + Offset - 2), where Addr is the address of the instruction *after* the branch opcode, and offset is the branch's operand. The special offset values 0 and 1 are interpreted as "return 0" and "return 1" respectively. *[This odd hiccup is inherited from the Z-machine. Inform uses it heavily for code optimization.]*

It is legal to branch to code that is in another function. *[Indeed, there is no well-defined notion of where a function ends.]* However, this does not affect the current stack frame; that remains set up according to the same function call as before the branch. Similarly, it is legal to branch to code which is not associated with any function — e.g., code compiled on the fly in RAM.

```
jump L1
```

Branch unconditionally to offset L1.

```
jz L1 L2
```

If L1 is equal to zero, branch to L2.

```
jnz L1 L2
```

If L1 is not equal to zero, branch to L2.

```
jeq L1 L2 L3
```

If L1 is equal to L2, branch to L3.

```
jne L1 L2 L3
```

If L1 is not equal to L2, branch to L3.

```
jlt L1 L2 L3
jle L1 L2 L3
jgt L1 L2 L3
jge L1 L2 L3
```

Branch is L1 is less than, less than or equal to, greater than, greater than or equal to L2. The values are compared as signed 32-bit values.

```
jltu L1 L2 L3
jleu L1 L2 L3
jgtu L1 L2 L3
jgeu L1 L2 L3
```

The same, except that the values are compared as unsigned 32-bit values.

[Since the address space can span the full 32-bit range, it is wiser to compare addresses with the unsigned comparison operators.]

```
jumpabs L1
```

Branch unconditionally to address L1. Unlike the other branch opcodes, this takes an absolute address, not an offset. The special cases 0 and 1 (for returning) do not apply; jumpabs 0 would branch to memory address 0, if that were ever a good idea, which it isn't.

2.3. Moving Data

```
copy L1 S1
```

Read L1 and store it at S1, without change.

```
copys L1 S1
```

Read a 16-bit value from L1 and store it at S1.

```
copyb L1 S1
```

Read an 8-bit value from L1 and store it at S1.

Since copys and copyb can access chunks smaller than the usual four bytes, they require some comment. When reading from main memory or the call-frame locals, they access two or one bytes, instead of four. However, when popping or pushing values on the stack, these opcodes pull or push a full 32-bit value.

Therefore, if copyb (for example) copies a byte from main memory to the stack, a 32-bit value will be pushed, whose value will be from 0 to 255. Sign-extension *does not* occur. Conversely, if copyb copies a byte from the stack to memory, a 32-bit value is popped, and the bottom 8 bits are written at the given address. The upper 24 bits are lost. Constant values are truncated as well.

If copys or copyb are used with both L1 and S1 in pop/push mode, the 32-bit value is popped, truncated, and pushed.

[NOTE: Since a call frame has no specified endianness, it is unwise to use these opcodes to pull out one or two bytes from a four-byte local variable. The result will be implementation-dependent. Therefore, use of the copyb and copys opcodes with a local-variable operand of different size is deprecated. Since locals of less than four bytes are also *deprecated, you should not use copyb or copys with local-variable operands at all.]*

```
sexs L1 S1
```

Sign-extend a value, considered as a 16-bit value. If the value's 8000 bit is set, the upper 16 bits are all set; otherwise, the upper 16 bits are all cleared.

```
sexb L1 S1
```

Sign-extend a value, considered as an 8-bit value. If the value's 80 bit is set, the upper 24 bits are all set; otherwise, the upper 24 bits are all cleared.

Note that these opcodes, like most, work on 32-bit values. Although (for example) sexb is commonly used in conjunction with copyb, it does not share copyb's behavior of reading a single byte from memory or the locals.

Also note that the upper bits, 16 or 24 of them, are entirely ignored and overwritten with ones or zeroes.

2.4. Array Data

```
astore L1 L2 L3
```

Store L3 into the 32-bit field at main memory address (L1+4*L2).

```
aload L1 L2 S1
```

Load a 32-bit value from main memory address (L1+4*L2), and store it in S1.

```
astores L1 L2 L3
```

Store L3 into the 16-bit field at main memory address (L1+2*L2).

```
aloads L1 L2 S1
```

Load an 16-bit value from main memory address (L1+2*L2), and store it in S1.

```
astoreb L1 L2 L3
```

Store L3 into the 8-bit field at main memory address (L1+L2).

```
aloadb L1 L2 S1
```

Load an 8-bit value from main memory address (L1+L2), and store it in S1.

Note that these opcodes cannot access call-frame locals, or the stack. (Not with the L1 and L2 opcodes, that is.) L1 and L2 provide a main-memory address. Be not confused by the fact that L1 and L2 can be any addressing mode, including call-frame or stack-pop modes. That controls where the values come from which are used to *compute* the main-memory address.

The other end of the transfer (S1 or L3) is always a 32-bit value. The "store" opcodes truncate L3 to 8 or 16 bits if necessary. The "load" opcodes expand 8-bit or 16-bit values *without* sign extension. (If signed values are appropriate, you can follow aloads/aloadb with sexs/sexb.)

L2 is considered signed, so you can access addresses before L1 as well as after.

```
astorebit L1 L2 L3
```

Set or clear a single bit. This is bit number (L2 mod 8) of memory address (L1+L2/8). It is cleared if L3 is zero, set if nonzero.

```
aloadbit L1 L2 S1
```

Test a single bit, similarly. If it is set, 1 is stored at S1; if clear, 0 is stored.

For these two opcodes, bits are effectively numbered sequentially, starting with the least significant bit of address L1. L2 is considered signed, so this numbering extends both positively and negatively. For example:

```
astorebit  1002  0  1:  Set bit 0 of address 1002. (The 1's place.)
astorebit  1002  7  1:  Set bit 7 of address 1002. (The 128's place.)
astorebit  1002  8  1:  Set bit 0 of address 1003.
astorebit  1002  9  1:  Set bit 1 of address 1003.
astorebit  1002 -1  1:  Set bit 7 of address 1001.
astorebit  1002 -3  1:  Set bit 5 of address 1001.
astorebit  1002 -8  1:  Set bit 0 of address 1001.
astorebit  1002 -9  1:  Set bit 7 of address 1000.
```

Like the other aload and astore opcodes, these opcodes cannot access call-frame locals, or the stack.

2.5. The Stack

```
stkcount S1
```

Store a count of the number of values on the stack. This counts only values above the current call-frame. In other words, it is always zero when a C1 function starts executing, and (numargs+1) when a C0 function starts executing. It then increases and decreases thereafter as values are pushed and popped; it is always the number of values that can be popped legally. (If S1 uses the stack push mode, the count is done before the result is pushed.)

```
stkpeek L1 S1
```

Peek at the L1'th value on the stack, without actually popping anything. If L1 is zero, this is the top value; if one, it's the value below that; etc. L1 must be less than the current stack-count. (If L1 or S1 use the stack pop/push modes, the peek is counted after L1 is popped, but before the result is pushed.)

```
stkswap
```

Swap the top two values on the stack. The current stack-count must be at least two.

```
stkcopy L1
```

Peek at the top L1 values in the stack, and push duplicates onto the stack in the same order. If L1 is zero, nothing happens. L1 must not be greater than the current stack-count. (If L1 uses the stack pop mode, the stkcopy is counted after L1 is popped.)

An example of stkcopy, starting with six values on the stack:

```
5 4 3 2 1 0 <top>
stkcopy 3
5 4 3 2 1 0 2 1 0 <top>
```

```
stkroll L1 L2
```

Rotate the top L1 values on the stack. They are rotated up or down L2 places, with positive values meaning up and negative meaning down. The current stack-count must be at least L1. If either L1 or L2 is zero, nothing happens. (If L1 and/or L2 use the stack pop mode, the roll occurs after they are popped.)

An example of two stkrolls, starting with nine values on the stack:

```
8 7 6 5 4 3 2 1 0 <top>
stkroll 5 1
8 7 6 5 0 4 3 2 1 <top>
stkroll 9 -3
5 0 4 3 2 1 8 7 6 <top>
```

Note that stkswap is equivalent to stkroll 2 1, or for that matter stkroll 2 -1. Also, stkcopy 1 is equivalent to stkpeek 0 sp.

These opcodes can only access the values pushed on the stack above the current call-frame. It is illegal to stkswap, stkpeek, stkcopy, or stkroll values below that — i.e, the locals segment or any previous function call frames.

2.6. Functions

```
call L1 L2 S1
```

Call function whose address is L1, passing in L2 arguments, and store the return result at S1.

The arguments are taken from the stack. Before you execute the call opcode, you must push the arguments on, in backward order (last argument pushed first, first argument topmost on the stack.) The L2 arguments are removed before the new function's call frame is constructed. (If L1, L2, or S1 use the stack pop/push modes, the arguments are taken after L1 or L2 is popped, but before the result is pushed.)

Recall that all functions in Glulx have a single 32-bit return value. If you do not care about the return value, you can use operand mode 0 ("discard value") for operand S1.

```
callf L1 S1
callfi L1 L2 S1
callfii L1 L2 L3 S1
callfiii L1 L2 L3 L4 S1
```

Call function whose address is L1, passing zero, one, two, or three arguments. Store the return result at S1.

These opcodes behave the same as call, except that the arguments are given in the usual opcode format instead of being found on the stack. (If L2, L3, etc. all use the stack pop mode, then the behavior is exactly the same as call.)

```
return L1
```

Return from the current function, with the given return value. If this is the top-level function, Glulx execution is over.

Note that all the branch opcodes (jump, jz, jeq, and so on) have an option to return 0 or 1 instead of branching. These behave exactly as if the return opcode had been executed.

```
tailcall L1 L2
```

Call function whose address is L1, passing in L2 arguments, and pass the return result out to whoever called the current function.

This destroys the current call-frame, as if a return had been executed, but does not touch the call stub below that. It then immediately calls L1, creating a new call-frame. The effect is the same as a call immediately followed by a return, but takes less stack space.

It is legal to use tailcall from the top-level function. L1 becomes the top-level function.

[This opcode can be used to implement tail recursion, without forcing the stack to grow with every call.]

2.7. Continuations

```
catch S1 L1
```

Generates a "catch token", which can be used to jump back to this execution point from a throw opcode. The token is stored in S1, and then execution branches to offset L1. If execution is proceeding from this point because of a throw, the thrown value is stored instead, and the branch is ignored.

Remember if the branch value is not 0 or 1, the branch is to to (Addr + L1 - 2), where Addr is the address of the instruction *after* the catch. If the value *is* 0 or 1, the function returns immediately, invalidating the catch token.

If S1 or L1 uses the stack push/pop modes, note that the precise order of execution is: evaluate L1 (popping if appropriate); generate a call stub and compute the token; store S1 (pushing if appropriate).

```
throw L1 L2
```

Jump back to a previously-executed catch opcode, and store the value L1. L2 must be a valid catch token.

The exact catch/throw procedure is as follows:

When catch is executed, a four-value call stub is pushed on the stack — result destination, PC, and FramePtr. (See section 1.3.2, "Call Stubs". The PC is the address of the next instruction after the catch.) The catch token is the value of the stack pointer after these are pushed. The token value is stored in the result destination, and execution proceeds, branching to L1.

When throw is executed, the stack is popped down until the stack pointer equals the given token. Then the four values are read back off the stack, the thrown value is stored in the destination, and execution proceeds with the instruction after the catch.

If the call stub (or any part of it) is removed from the stack, the catch token becomes invalid, and must not be used. This will certainly occur when you return from the function containing the catch opcode. It will also occur if you pop too many values from the stack after executing the catch. (You may wish to do this to "cancel" the catch; if you pop and discard those four values, the token is invalidated, and it is as if you had never executed the catch at all.) The catch token is also invalidated if any part of the call stub is overwritten (e.g. with stkswap or stkroll).

[Why is the catch branch taken at catch time, and ignored after a throw? Because it's easier to write the interpreter that way, that's why. If it had to branch after a throw, either the call stub would have to contain the branch offset, or the terp would have to re-parse the catch instruction. Both are ugly.]

2.8. Memory Map

```
getmemsize S1
```

Store the current size of the memory map. This is originally the ENDMEM value from the header, but you can change it with the setmemsize opcode. (The malloc and mfree opcodes may also cause this value to change; see section 2.9, "Memory Allocation Heap".) It will always be greater than or equal to ENDMEM, and will always be a multiple of 256.

```
setmemsize L1 S1
```

Set the current size of the memory map. The new value must be a multiple of 256, like all memory boundaries in Glulx. It must be greater than or equal to ENDMEM (the initial memory-size value which is stored in the header.) It does not have to be greater than the previous memory size. The memory size may grow and shrink over time, as long as it never gets smaller than the initial size.

When the memory size grows, the new space is filled with zeroes. When it shrinks, the contents of the old space are lost.

If the allocation heap is active (see section 2.9, "Memory Allocation Heap") you may not use setmemsize -- the memory map is under the control of the heap system. If you free all heap objects, the heap will then no longer be active, and you can use setmemsize.

Since memory allocation is never guaranteed, you must be prepared for the possibility that setmemsize will fail. The opcode stores the value zero if it succeeded, and 1 if it failed. If it failed, the memory size is unchanged.

Some interpreters do not have the capability to resize memory at all. On such interpreters, setmemsize will *always* fail. You can check this in advance with the ResizeMem gestalt selector.

Note that the memory size is considered part of the game state. If you restore a saved game, the current memory size is changed to the size that was in effect when the game was saved. If you restart, the current memory size is reset to its initial value.

2.9. Memory Allocation Heap

Manage the memory allocation heap.

Glulx is able to maintain a list of dynamically-allocated memory objects. These objects exist in the memory map, above ENDMEM. The malloc and mfree opcodes allow the game to request the allocation and destruction of these objects.

Some interpreters do not have the capability to manage an allocation heap. On such interpreters, malloc will always fail. You can check this in advance with the MAlloc gestalt selector.

When you first allocate a block of memory, the heap becomes active. The current end of memory -- that is, the current getmemsize value -- becomes the beginning address of the heap. The memory map is then extended to accomodate the memory block.

Subsequent memory allocations and deallocations are done within the heap. The interpreter may extend or reduce the memory map, as needed, when allocations and deallocations occur. While the heap is active, you may not manually resize the memory map with setmemsize; the heap system is responsible for doing that.

When you free the last extant memory block, the heap becomes inactive. The interpreter will reduce the memory map size down to the heap-start address. (That is, the getmemsize value returns to what it was before you allocated the first block.) Thereafter, it is legal to call setmemsize again.

It is legitimate to read or write any memory address in the heap range (from ENDMEM to the end of the memory map). You are not restricted to extant blocks. *[The VM's heap state is not stored in its own memory map. So, unlike the familiar C heap, you cannot damage it by writing outside valid blocks.]*

The heap state (whether it is active, its starting address, and the addresses and sizes of all extant blocks) *is* part of the saved game state.

These opcodes were added in Glulx version 3.1.

```
malloc L1 S1
```

Allocate a memory block of L1 bytes. (L1 must be positive.) This stores the address of the new memory block, which will be within the heap and will not overlap any other extant block. The interpreter may have to extend the memory map (see section 2.8, "Memory Map") to accomodate the new block.

This operation does not change the contents of the memory block (or, indeed, the contents of the memory map at all). If you want the memory block to be initialized, you must do it yourself.

If the allocation fails, this stores zero.

```
mfree L1
```

Free the memory block at address L1. This *must* be the address of an extant block -- that is, a value returned by malloc and not previously freed.

This operation does not change the contents of the memory block (or, indeed, the contents of the memory map at all).

2.10. Game State

```
quit
```

Shut down the terp and exit. This is equivalent to returning from the top-level function, or for that matter calling glk_exit().

Note that (in the Glk I/O system) Glk is responsible for any "hit any key to exit" prompt. It is safe for you to print a bunch of final text and then exit immediately.

```
restart
```

Restore the VM to its initial state (memory, stack, and registers). Note that the current memory size is reset, as well as the contents of memory.

```
save L1 S1
```

Save the VM state to the output stream L1. It is your responsibility to prompt the player for a filespec, open the stream, and then destroy these objects afterward. S1 is set to zero if the operation succeeded, 1 if it failed, and -1 if the VM has just been restored and is continuing from this instruction.

(In the Glk I/O system, L1 should be the ID of a writable Glk stream. In other I/O systems, it will mean something different. In the "filter" and "null" I/O systems, the save opcode is illegal, as the interpreter has nowhere to write the state.)

```
restore L1 S1
```

Restore the VM state from the input stream L1. S1 is set to 1 if the operation failed. If it succeeded, of course, this instruction never returns a value.

```
saveundo S1
```

Save the VM state in a temporary location. The terp will choose a location appropriate for rapid access, so this may be called once per turn. S1 is set to zero if the operation succeeded, 1 if it failed, and -1 if the VM state has just been restored.

```
restoreundo S1
```

Restore the VM state from temporary storage. S1 is set to 1 if the operation failed.

```
protect L1 L2
```

Protect a range of memory from restart, restore, restoreundo. The protected range starts at address L1 and has a length of L2 bytes. This memory is silently unaffected by the state-restoring operations. (However, if the result-storage S1 is directed into the protected range, that is not blocked.)

When the VM starts up, there is no protection range. Only one range can be protected at a time. Calling protect cancels any previous range. To turn off protection, call protect with L1 and L2 set to zero.

It is important to note that the protection range itself (its existence, location, and length) is *not* part of the saved game state! If you save a game, move the protection range to a new location, and then restore that game, it is the new range that will be protected, and the range will remain there afterwards.

```
verify S1
```

Perform sanity checks on the game file, using its length and checksum. S1 is set to zero if everything looks good, 1 if there seems to be a problem. (Many interpreters will do this automatically, before the game starts executing. This opcode is provided mostly for slower interpreters, where auto-verify might cause an unacceptable delay.)

Notes:

All the save and restore opcodes can generate diagnostic information on the current output stream.

A terp may support several levels of temporary storage. You should not make any assumptions about how many times restoreundo can be called. If the player so requests, you should keep calling it until it fails.

Glk opaque objects (windows, streams, filespecs) are not part of the saved game state. Therefore, when you restore a game, all the object IDs you have in Glulx memory must be considered invalid. (This includes both IDs in main memory and on the stack.) You must use the Glk iteration calls to go through all the opaque objects in existence, and recognize them by their rocks.

The same applies after restoreundo, to a lesser extent. Since saveundo/restoreundo only operate within a single play session, you can rely on the IDs of objects created before the first saveundo. However, if you have created any objects since then, you must iterate and recognize them.

The restart opcode is a similar case. You must do an iteration as soon as your program starts, to find objects created in an earlier incarnation. Alternatively, you can be careful to close all opaque objects before invoking restart.

[Another approach is to use the protect opcode, to preserve global variables containing your object IDs. This will work within a play session — that is, with saveundo, restoreundo, and restart. You must still deal with save and restore.]

2.11. Output

```
getiosys S1 S2
```

Return the current I/O system mode and rock.

Due to a long-standing bug in the reference interpreter, the two store operands must be of the same general type: both main-memory/global stores, both local variable stores, or both stack pushes.

```
setiosys L1 L2
```

Set the I/O system mode and rock. If the system L1 is not supported by the interpreter, it will default to the "null" system (0).

These systems are currently defined:

- 0: The null system. All output is discarded. (When the Glulx machine starts up, this is the current system.)
- 1: The filtering system. The rock (L2) value should be the address of a Glulx function. This function will be called for every character output (with the character value as its sole argument). The function’s return value is ignored.
- 2: The Glk system. All output will be handled through Glk function calls, sent to the current Glk stream.
- 20: The FyreVM channel system. See section 0.2, "Glulx and Other IF Systems".

It is important to recall that when Glulx starts up, the Glk I/O system is *not* set. And when Glk starts up, there are no windows and no current output stream. To make anything appear to the user, you must first do three things: select the Glk I/O system, open a Glk window, and set its stream as the current one. (It is illegal in Glk to send output when there is no stream set. Sending output to Glulx’s "null" I/O system is legal, but pointless.)

```
streamchar L1
```

Send L1 to the current stream. This sends a single character; the value L1 is truncated to eight bits.

```
streamunichar L1
```

Send L1 to the current stream. This sends a single (32-bit) character.

This opcode was added in Glulx version 3.0.

```
streamnum L1
```

Send L1 to the current stream, represented as a signed decimal number in ASCII.

```
streamstr L1
```

Send a string object to the current stream. L1 must be the address of a Glulx string object (type E0, E1, or E2.) The string is decoded and sent as a sequence of characters.

When the Glk I/O system is set, these opcodes are implemented using the Glk API. You can bypass them and directly call glk_put_char(), glk_put_buffer(), and so on. Remember, however, that glk_put_string() only accepts unencoded string (E0) objects; glk_put_string_uni() only accepts unencoded Unicode (E2) objects.

Note that it is illegal to decode a compressed string (E1) if there is no string-decoding table set.

```
getstringtbl S1
```

Return the address the terp is currently using for its string-decoding table. If there is no table, set, this returns zero.

```
setstringtbl L1
```

Change the address the terp is using for its string-decoding table. This may be zero, indicating that there is no table (in which case it is illegal to print any compressed string). Otherwise, it must be the address of a *valid* string-decoding table.

[This does not change the value in the header field at address 001C. The header is in ROM, and never changes. To determine the current table address, use the getstringtbl opcode.]

A string-decoding table may be in RAM or ROM, but there may be speed penalties if it is in RAM. See section 1.6.1.4, "The String-Decoding Table".

2.12. Floating-Point Math

Recall that floating-point values are encoded as single-precision (32-bit) IEEE-754 values (see section 1.7, "Floating-Point Numbers"). The interpreter must convert values (from memory or the stack) before performing a floating-point operation, and unconvert them afterwards.

[In other words, passing a float value to an integer arithmetic opcode will operate on the IEEE-754-encoded 32-bit value. Such an operation would be deterministic, albeit mathematically meaningless. The same is true for passing an integer to a float opcode.]

Float operations which produce inexact results are not guaranteed to be identical on every platform. That is, 1.0 plus 1.0 will always be 2.0, because that can be represented exactly. But acos(-1.0), which should be pi, may generate either 40490FDA (3.14159250...) or 40490FDB (3.14159274...). Both are approximations of the correct result, but which one you get depends on the interpreter’s underlying math library.

If any argument to a float operation is a NaN ("not a number") value, the result will be a NaN value. (Except for the pow opcode, which has some special cases.)

[Speaking of special cases: I have tried to describe all the important ones for these operations. However, you should also consult the Glulxercise unit test (available on the Glulx web site). Consider it definitive if this document is unclear.]

These opcodes were added in Glulx version 3.1.2. However, not all interpreters may support them. You can test for their availability with the Float gestalt selector.

```
numtof L1 S1
```

Convert an integer value to the closest equivalent float. (That is, if L1 is 1, then 3F800000 -- the float encoding of 1.0 -- will be stored in S1.) Integer zero is converted to (positive) float zero.

If the value is less than -1000000 or greater than 1000000 (hex), the conversion may not be exact. (More specifically, it may round to a nearby multiple of a power of 2.)

```
ftonumz L1 S1
```

Convert a float value to an integer, rounding towards zero (i.e., truncating the fractional part). If the value is outside the 32-bit integer range, or is NaN or infinity, the result will be 7FFFFFFF (for positive values) or 80000000 (for negative values).

```
ftonumn L1 S1
```

Convert a float value to an integer, rounding towards the nearest integer. Again, overflows become 7FFFFFFF or 80000000.

```
fadd L1 L2 S1
fsub L1 L2 S1
fmul L1 L2 S1
fdiv L1 L2 S1
```

Perform floating-point arithmetic. Overflows produce infinite values (with the appropriate sign); underflows produce zero values (ditto). 0/0 is NaN. Inf/Inf, or Inf-Inf, is NaN. Any finite number added to infinity is infinity. Any nonzero number divided by an infinity, or multiplied by zero, is a zero. Any nonzero number multiplied by an infinity, or divided by zero, is an infinity.

```
fmod L1 L2 S1 S2
```

Perform a floating-point modulo operation. S1 is the remainder (or modulus); S2 is the quotient.

S2 is L1/L2, rounded (towards zero) to an integral value. S1 is L1-(S2*L2). Note that S1 always has the same sign as L1; S2 has the appropriate sign for L1/L2.

If L2 is 1, this gives you the fractional and integer parts of L1. If L1 is zero, both results are zero. If L2 is infinite, S1 is L1 and S2 is zero. If L1 is infinite or L2 is zero, both results are NaN.

```
ceil L1 S1
floor L1 S1
```

Round L1 up (towards +Inf) or down (towards -Inf) to the nearest integral value. (The result is still in float format, however.) These opcodes are idempotent.

The result keeps the sign of L1; in particular, floor(0.5) is 0 and ceil(-0.5) is -0. Rounding -0 up or down gives -0. Rounding an infinite value gives infinity.

```
sqrt L1 S1
exp L1 S1
log L1 S1
```

Compute the square root of L1, e^L1, and log of L1 (base e).

sqrt(-0) is -0. sqrt returns NaN for all other negative values. exp(+0) and exp(-0) are 1; exp(-Inf) is +0. log(+0) and log(-0) are -Inf. log returns NaN for all other negative values.

```
pow L1 L2 S1
```

Compute L1 raised to the L2 power.

The special cases are breathtaking. The following is quoted (almost) directly from the libc man page:

- pow(±0, y) returns ±Inf for y an odd integer < 0.
- pow(±0, y) returns +Inf for y < 0 and not an odd integer.
- pow(±0, y) returns ±0 for y an odd integer > 0.
- pow(±0, y) returns +0 for y > 0 and not an odd integer.
- pow(-1, ±Inf) returns 1.
- pow(1, y) returns 1 for any y, even a NaN.
- pow(x, ±0) returns 1 for any x, even a NaN.
- pow(x, y) returns a NaN for finite x < 0 and finite non-integer y.
- pow(x, -Inf) returns +Inf for |x| < 1.
- pow(x, -Inf) returns +0 for |x| > 1.
- pow(x, +Inf) returns +0 for |x| < 1.
- pow(x, +Inf) returns +Inf for |x| > 1.
- pow(-Inf, y) returns -0 for y an odd integer < 0.
- pow(-Inf, y) returns +0 for y < 0 and not an odd integer.
- pow(-Inf, y) returns -Inf for y an odd integer > 0.
- pow(-Inf, y) returns +Inf for y > 0 and not an odd integer.
- pow(+Inf, y) returns +0 for y < 0.
- pow(+Inf, y) returns +Inf for y > 0.
- pow(x, y) returns NaN if x is negative and y is not an integer (both finite).

```
sin L1 S1
cos L1 S1
tan L1 S1
acos L1 S1
asin L1 S1
atan L1 S1
```

Compute the standard trigonometric functions.

sin and cos return values in the range -1 to 1. sin, cos, and tan of infinity are NaN.

asin is always in the range -pi/2 to pi/2; acos is always in the range 0 to pi. asin and acos of values greater than 1, or less than -1, are NaN. atan(±Inf) is ±pi/2.

```
atan2 L1 L2 S1
```

Computes the arctangent of L1/L2, using the signs of both arguments to determine the quadrant of the return value. (Note that the Y argument is first and the X argument is second.)

Again with the special cases:

- atan2(±0, -0) returns ±pi.
- atan2(±0, +0) returns ±0.
- atan2(±0, x) returns ±pi for x < 0.
- atan2(±0, x) returns ±0 for x > 0.
- atan2(y, ±0) returns +pi/2 for y > 0.
- atan2(y, ±0) returns -pi/2 for y < 0.

- atan2(±y, -Inf) returns ±pi for finite y.
- atan2(±y, +Inf) returns ±0 for finite y.
- atan2(±Inf, x) returns ±pi/2 for finite x.
- atan2(±Inf, -Inf) returns ±3*pi/4.
- atan2(±Inf, +Inf) returns ±pi/4.

2.13. Floating-Point Comparisons

All these branch opcodes specify their destinations with an offset value. See section 2.2, "Branches".

Most of these opcodes never branch if any argument is NaN. (Exceptions are jisnan and jfne.) In particular, NaN is neither less than, greater than, nor equal to NaN.

These opcodes were added in Glulx version 3.1.2. However, not all interpreters may support them. You can test for their availability with the Float gestalt selector.

```
jisnan L1 L2
```

Branch to L2 if the floating-point value L1 is a NaN value. (See section 1.7, "Floating-Point Numbers".)

```
jisinf L1 L2
```

Branch to L2 if the floating-point value L1 is an infinity (7F800000 or FF800000).

```
jfeq L1 L2 L3 L4
```

Branch to L4 if the difference between L1 and L2 is less than or equal to (plus or minus) L3. The sign of L3 is ignored.

If any of the arguments are NaN, this will not branch. If L3 is infinite, this will always branch -- unless L1 and L2 are opposite infinities. (Opposite infinities are never equal, regardless of L3. Infinities of the same sign are always equal.)

If L3 is (plus or minus) zero, this tests for exact equality. Note that +0 is considered exactly equal to -0.

```
jfne L1 L2 L3 L4
```

The reverse of jfeq. This *will* branch if *any* of the arguments is NaN.

```
jflt L1 L2 L3
jfle L1 L2 L3
jfgt L1 L2 L3
jfge L1 L2 L3
```

Branch to L3 if L1 is less than (less than or equal to, greater than, greater than or equal to) L2.

+0 and -0 behave identically in comparisons. In particular, +0 is considered equal to -0, not greater than -0.

2.14. Random Number Generator

```
random L1 S1
```

Return a random number in the range 0 to (L1-1); or, if L1 is negative, the range (L1+1) to 0. If L1 is zero, return a random number in the full 32-bit integer range. (Remember that this may be either positive or negative.)

```
setrandom L1
```

Seed the random-number generator with the value L1. If L1 is zero, subsequent random numbers will be as genuinely unpredictable as the terp can provide; it may include timing data or other random sources in its generation. If L1 is nonzero, subsequent random numbers will follow a deterministic sequence, always the same for a given nonzero seed.

The terp starts up in the "nondeterministic" mode (as if setrandom 0 had been invoked.)

The random-number generator is not part of the saved-game state.

2.15. Block Copy and Clear

```
mzero L1 L2
```

Write L1 zero bytes, starting at address L2. This is exactly equivalent to:

```
for (ix=0: ix<L1: ix++) L2->ix = 0;
```

```
mcopy L1 L2 L3
```

Copy L1 bytes from address L2 to address L3. It is safe to copy a block to an overlapping block. This is exactly equivalent to:

```
if (L3 < L2)
  for (ix=0: ix<L1: ix++) L3->ix = L2->ix;
else
  for (ix=L1-1: ix>=0: ix--) L3->ix = L2->ix;
```

For both of these opcodes, L1 may be zero, in which case the opcodes do nothing. The operands are considered unsigned, so a "negative" L1 is a very large number (and almost certainly a mistake).

These opcodes were added in Glulx version 3.1. You can test for their availability with the MemCopy gestalt selector.

2.16. Searching

Perform a generic linear, binary, or linked-list search.

[These are outrageously CISC for an hardware CPU, but easy enough to add to a software terp; and taking advantage of them can speed up a program considerably. Advent, under the Inform library, runs 15-20% faster when property-table lookup is handled with a binary-search opcode instead of Inform code. A similar change in the dictionary lookup trims another percent or so.]

All three of these opcodes operate on a collection of fixed-size data structures in memory. A key, which is a fixed-length array of bytes, is found at a known position within each data structure.

The opcodes search the collection of structures, and find one whose key matches a given key.

The following flags may be set in the Options argument. Note that not all flags can be used with all types of searches.

- KeyIndirect (0x01): This flag indicates that the Key argument passed to the opcode is the address of the actual key. If this flag is not used, the Key argument is the key value itself. (In this case, the KeySize *must* be 1, 2, or 4 -- the native sizes of Glulx values. If the KeySize is 1 or 2, the lower bytes of the Key are used and the upper bytes ignored.)
- ZeroKeyTerminates (0x02): This flag indicates that the search should stop (and return failure) if it encounters a structure whose key is all zeroes. If the searched-for key happens to also be all zeroes, the success takes precedence.
- ReturnIndex (0x04): This flag indicates that search should return the array index of the structure that it finds, or -1 (0xFFFFFFFF) for failure. If this flag is not used, the search returns the address of the structure that it finds, or 0 for failure.

```
linearsearch L1 L2 L3 L4 L5 L6 L7 S1
```

- L1: Key
- L2: KeySize
- L3: Start
- L4: StructSize
- L5: NumStructs
- L6: KeyOffset
- L7: Options
- S1: Result

An array of data structures is stored in memory, beginning at Start, each structure being StructSize bytes. Within each struct, there is a key value KeySize bytes long, starting at position KeyOffset (from the start of the structure.) Search through these in order. If one is found whose key matches, return it. If NumStructs are searched with no result, the search fails.

NumStructs may be -1 (0xFFFFFFFF) to indicate no upper limit to the number of structures to search. The search will continue until a match is found, or (if ZeroKeyTerminates is used) a zero key.

The KeyIndirect, ZeroKeyTerminates, and ReturnIndex options may be used.

```
binarysearch L1 L2 L3 L4 L5 L6 L7 S1
```

- L1: Key
- L2: KeySize
- L3: Start
- L4: StructSize
- L5: NumStructs
- L6: KeyOffset
- L7: Options
- S1: Result

An array of data structures is in memory, as above. However, the structs must be stored in forward order of their keys (taking each key to be a big-endian unsigned integer.) There can be no duplicate keys. NumStructs must indicate the exact length of the array; it cannot be -1.

The KeyIndirect and ReturnIndex options may be used.

```
linkedsearch L1 L2 L3 L4 L5 L6 S1
```

- L1: Key
- L2: KeySize
- L3: Start
- L4: KeyOffset
- L5: NextOffset
- L6: Options
- S1: Result

The structures need not be consecutive; they may be anywhere in memory, in any order. They are linked by a four-byte address field, which is found in each struct at position NextOffset. If this field contains zero, it indicates the end of the linked list.

The KeyIndirect and ZeroKeyTerminates options may be used.

2.17. Accelerated Functions

To improve performance, Glulx incorporates some complex functions which replicate code in the Inform library. *[Yes, this is even more outrageously CISC than the search opcodes.]*

Rather than allocating a new opcode for each function, Glulx offers an expandable function acceleration system. Two functions are defined below. The game may request that a particular address — the address of a VM function — be replaced by one of the available functions. This does not alter memory; but any subsequent call to that address might invoke the terp's built-in version of the function, instead of the VM code at that address.

(A "call" includes any function invocation of that address, including the call, tailcall, and callf (etc.) opcodes. It also includes invocation via the filter I/O system, and function nodes in the string-decoding table. Branches to the address are *not* affected; neither are returns, throws, or other ways the terp might reach it.)

A terp may implement any, all, or none of the functions on the list. If the game requests an accelerated function which is not available, the request is ignored. Therefore, the game *must* be sure that it only requests an accelerated function at an address which actually matches the requested function.

Some functions may require values (or addresses) which are compiled into the game file, or otherwise stored by the game. The interpreter maintains a table of these parameters — whichever ones are needed by the functions it supports. All parameters in the table are initially zero; the game may supply values as needed.

The set of active acceleration requests, and the values in the parameter table, are *not* part of the saved-game state.

The behavior of an accelerated function is somewhat limited. The state of the VM during the function is not defined, so there is no way for an accelerated function to call a normal VM function. The normal printing mechanism (as in the streamchar opcode, etc) is not available, since that can call VM functions via the filter I/O system. *[Not that I/O functions are likely to be worth accelerating in any case.]*

Errors encountered during an accelerated function will be displayed to the user by some convenient means. For example, an interpreter may send the error message to the current Glk output stream. However, the terp may have no recourse but to invoke a *fatal* error. (For example, if there is no current Glk output stream.) Therefore, accelerated functions are defined with no error conditions that must be recoverable.

These opcodes were added in Glulx version 3.1.1. Since a 3.1.1 game file ought to run in a 3.1.0 interpreter, you *may not* use these opcodes without first testing the Acceleration gestalt selector. If it returns zero, your game is running on a 3.1.0 terp (or earlier), and it is your responsibility to avoid executing these opcodes. *[Of course, the way the opcodes are defined should ensure that skipping them does not affect the behavior of your game.]*

```
accelfunc L1 L2
```

Request that the VM function with address L2 be replaced by the accelerated function whose number is L1. If L1 is zero, the acceleration for address L2 is cancelled.

If the terp does not offer accelerated function L1, this does nothing.

If you request acceleration at an address which is already accelerated, the previous request is cancelled before the new one is considered. If you cancel at an unaccelerated address, nothing happens.

A given accelerated function L1 may replace several VM functions (at different addresses) at the same time. Each request is considered separate, and must be cancelled separately.

```
accelparam L1 L2
```

Store the value L2 in the parameter table at position L1. If the terp does not know about parameter L1, this does nothing.

The list of accelerated functions is as follows. They are defined as if in Inform 6 source code. (Consider Inform's "strict" mode to be off, for the purposes of operators such as .& and -->.) ERROR() represents code which displays an error, as described above.

(Functions may be added to this list in future versions of the Glulx spec. Existing functions will not be removed or altered. Functions and parameters numbered 0x1100 to 0x11FF are reserved for extension projects by Dannii Willis. These are not documented here. See section 0.2, "Glulx and Other IF Systems".)

Note that functions 2 through 7 are deprecated; they behave badly if the Inform 6 NUM_ATTR_BYTES option (parameter 7) is changed from its default value (7). They will not be removed, but new games should use functions 8 through 13 instead.

```
Constant PARAM_0_classes_table = #classes_table;
Constant PARAM_1_indiv_prop_start = INDIV_PROP_START;
Constant PARAM_2_class_metaclass = Class;
Constant PARAM_3_object_metaclass = Object;
Constant PARAM_4_routine_metaclass = Routine;
Constant PARAM_5_string_metaclass = String;
Constant PARAM_6_self = #globals_array + WORDSIZE * #g$self;
Constant PARAM_7_num_attr_bytes = NUM_ATTR_BYTES;
Constant PARAM_8_cpv__start = #cpv__start;

! OBJ_IN_CLASS: utility function; implements "obj in Class".
[ OBJ_IN_CLASS obj;
  return ((obj + 13 + PARAM_7_num_attr_bytes)-->0
    == PARAM_2_class_metaclass);
```

```
];

! FUNC_1_Z__Region: implements Z__Region() as of Inform 6.31.
[ FUNC_1_Z__Region addr
  tb endmem; ! locals
  if (addr<36) rfalse;
  @getmemsize endmem;
  @jgeu addr endmem?outrange;  ! branch if addr >= endmem (unsigned)
  tb=addr->0;
  if (tb >= $E0) return 3;
  if (tb >= $C0) return 2;
  if (tb >= $70 && tb <= $7F && addr >= (0-->2)) return 1;
.outrange;
  rfalse;
];

! FUNC_2_CP__Tab: implements CP__Tab() as of Inform 6.31.
[ FUNC_2_CP__Tab obj id
  otab max res; ! locals
  if (FUNC_1_Z__Region(obj)~=1) {
    ERROR("[** Programming error: tried to find the ~.~ of (something)
   **]");
    rfalse;
  }
  otab = obj-->4;
  if (otab == 0) return 0;
  max = otab-->0;
  otab = otab+4;
  @binarysearch id 2 otab 10 max 0 0 res;
  return res;
];

! FUNC_3_RA__Pr: implements RA__Pr() as of Inform 6.31.
[ FUNC_3_RA__Pr obj id
  cla prop ix; ! locals
  if (id & $FFFF0000) {
   cla = PARAM_0_classes_table-->(id & $FFFF);
   if (~~FUNC_5_OC__Cl(obj, cla)) return 0;
   @ushiftr id 16 id;
   obj = cla;
  }
  prop = FUNC_2_CP__Tab(obj, id);
  if (prop==0) return 0;
  if (OBJ_IN_CLASS(obj) && cla == 0) {
   if (id < PARAM_1_indiv_prop_start
       || id >= PARAM_1_indiv_prop_start+8)
     return 0;
  }
  if (PARAM_6_self-->0 ~= obj) {
   @aloadbit prop 72 ix;
   if (ix) return 0;
  }
  return prop-->1;
];

! FUNC_4_RL__Pr: implements RL__Pr() as of Inform 6.31.
[ FUNC_4_RL__Pr obj id
  cla prop ix; ! locals
  if (id & $FFFF0000) {
   cla = PARAM_0_classes_table-->(id & $FFFF);
   if (~~FUNC_5_OC__Cl(obj, cla)) return 0;
   @ushiftr id 16 id;
   obj = cla;
  }
```

```
  prop = FUNC_2_CP__Tab(obj, id);
  if (prop==0) return 0;
  if (OBJ_IN_CLASS(obj) && cla == 0) {
   if (id < PARAM_1_indiv_prop_start
       || id >= PARAM_1_indiv_prop_start+8)
     return 0;
  }
  if (PARAM_6_self-->0 ~= obj) {
   @aloadbit prop 72 ix;
   if (ix) return 0;
  }
  @aloads prop 1 ix;
  return WORDSIZE * ix;
];

! FUNC_5_OC__Cl: implements OC__Cl() as of Inform 6.31.
[ FUNC_5_OC__Cl obj cla
  zr jx inlist inlistlen; ! locals
  zr = FUNC_1_Z__Region(obj);
  if (zr == 3) {
   if (cla == PARAM_5_string_metaclass) rtrue;
   rfalse;
  }
  if (zr == 2) {
   if (cla == PARAM_4_routine_metaclass) rtrue;
   rfalse;
  }
  if (zr ~= 1) rfalse;
  if (cla == PARAM_2_class_metaclass) {
   if (OBJ_IN_CLASS(obj)
     || obj == PARAM_2_class_metaclass or PARAM_5_string_metaclass
        or PARAM_4_routine_metaclass or PARAM_3_object_metaclass)
     rtrue;
   rfalse;
  }
  if (cla == PARAM_3_object_metaclass) {
   if (OBJ_IN_CLASS(obj)
     || obj == PARAM_2_class_metaclass or PARAM_5_string_metaclass
        or PARAM_4_routine_metaclass or PARAM_3_object_metaclass)
     rfalse;
   rtrue;
  }
  if (cla == PARAM_5_string_metaclass or PARAM_4_routine_metaclass)
    rfalse;
  if (~~OBJ_IN_CLASS(cla)) {
   ERROR("[** Programming error: tried to apply ’ofclass’ with non-class
   **]");
   rfalse;
  }
  inlist = FUNC_3_RA__Pr(obj, 2);
  if (inlist == 0) rfalse;
  inlistlen = FUNC_4_RL__Pr(obj, 2) / WORDSIZE;
  for (jx=0 : jx<inlistlen : jx++) {
   if (inlist-->jx == cla) rtrue;
  }
  rfalse;
];

! FUNC_6_RV__Pr: implements RV__Pr() as of Inform 6.31.
[ FUNC_6_RV__Pr obj id
  addr; ! locals
  addr = FUNC_3_RA__Pr(obj, id);
  if (addr == 0) {
   if (id > 0 && id < PARAM_1_indiv_prop_start) {
```

```
      return PARAM_8_cpv__start-->id;
    }
    ERROR("[** Programming error: tried to read (something) **]");
    return 0;
  }
  return addr-->0;
];

! FUNC_7_OP__Pr: implements OP__Pr() as of Inform 6.31.
[ FUNC_7_OP__Pr obj id
  zr; ! locals
  zr = FUNC_1_Z__Region(obj);
  if (zr == 3) {
   if (id == print or print_to_array) rtrue;
   rfalse;
  }
  if (zr == 2) {
   if (id == call) rtrue;
   rfalse;
  }
  if (zr ~= 1) rfalse;
  if (id >= PARAM_1_indiv_prop_start
      && id < PARAM_1_indiv_prop_start+8) {
   if (OBJ_IN_CLASS(obj)) rtrue;
  }
  if (FUNC_3_RA__Pr(obj, id) ~= 0)
   rtrue;
  rfalse;
];

! FUNC_8_CP__Tab: implements CP__Tab() as of Inform 6.33.
[ FUNC_8_CP__Tab obj id
  otab max res; ! locals
  if (FUNC_1_Z__Region(obj)~=1) {
    ERROR("[** Programming error: tried to find the ~.~ of (something)
   **]");
    rfalse;
  }
  otab = obj-->(3+(PARAM_7_num_attr_bytes/4));
  if (otab == 0) return 0;
  max = otab-->0;
  otab = otab+4;
  @binarysearch id 2 otab 10 max 0 0 res;
  return res;
];

! FUNC_9_RA__Pr: implements RA__Pr() as of Inform 6.33.
[ FUNC_9_RA__Pr obj id
  cla prop ix; ! locals
  if (id & $FFFF0000) {
   cla = PARAM_0_classes_table-->(id & $FFFF);
   if (~~FUNC_11_OC__Cl(obj, cla)) return 0;
   @ushiftr id 16 id;
   obj = cla;
  }
  prop = FUNC_8_CP__Tab(obj, id);
  if (prop==0) return 0;
  if (OBJ_IN_CLASS(obj) && cla == 0) {
   if (id < PARAM_1_indiv_prop_start
       || id >= PARAM_1_indiv_prop_start+8)
      return 0;
  }
  if (PARAM_6_self-->0 ~= obj) {
   @aloadbit prop 72 ix;
```

```
    if (ix) return 0;
  }
  return prop-->1;
];

! FUNC_10_RL__Pr: implements RL__Pr() as of Inform 6.33.
[ FUNC_10_RL__Pr obj id
  cla prop ix; ! locals
  if (id & $FFFF0000) {
    cla = PARAM_0_classes_table-->(id & $FFFF);
    if (~~FUNC_11_OC__Cl(obj, cla)) return 0;
    @ushiftr id 16 id;
    obj = cla;
  }
  prop = FUNC_8_CP__Tab(obj, id);
  if (prop==0) return 0;
  if (OBJ_IN_CLASS(obj) && cla == 0) {
    if (id < PARAM_1_indiv_prop_start
        || id >= PARAM_1_indiv_prop_start+8)
      return 0;
  }
  if (PARAM_6_self-->0 ~= obj) {
    @aloadbit prop 72 ix;
    if (ix) return 0;
  }
  @aloads prop 1 ix;
  return WORDSIZE * ix;
];

! FUNC_11_OC__Cl: implements OC__Cl() as of Inform 6.33.
[ FUNC_11_OC__Cl obj cla
  zr jx inlist inlistlen; ! locals
  zr = FUNC_1_Z__Region(obj);
  if (zr == 3) {
    if (cla == PARAM_5_string_metaclass) rtrue;
    rfalse;
  }
  if (zr == 2) {
    if (cla == PARAM_4_routine_metaclass) rtrue;
    rfalse;
  }
  if (zr ~= 1) rfalse;
  if (cla == PARAM_2_class_metaclass) {
    if (OBJ_IN_CLASS(obj)
      || obj == PARAM_2_class_metaclass or PARAM_5_string_metaclass
         or PARAM_4_routine_metaclass or PARAM_3_object_metaclass)
      rtrue;
    rfalse;
  }
  if (cla == PARAM_3_object_metaclass) {
    if (OBJ_IN_CLASS(obj)
      || obj == PARAM_2_class_metaclass or PARAM_5_string_metaclass
         or PARAM_4_routine_metaclass or PARAM_3_object_metaclass)
      rfalse;
    rtrue;
  }
  if (cla == PARAM_5_string_metaclass or PARAM_4_routine_metaclass)
     rfalse;
  if (~~OBJ_IN_CLASS(cla)) {
    ERROR("[** Programming error: tried to apply ’ofclass’ with non-class
    **]");
    rfalse;
  }
  inlist = FUNC_9_RA__Pr(obj, 2);
```

```
  if (inlist == 0) rfalse;
  inlistlen = FUNC_10_RL__Pr(obj, 2) / WORDSIZE;
  for (jx=0 : jx<inlistlen : jx++) {
   if (inlist-->jx == cla) rtrue;
  }
  rfalse;
];

! FUNC_12_RV__Pr: implements RV__Pr() as of Inform 6.33.
[ FUNC_12_RV__Pr obj id
  addr; ! locals
  addr = FUNC_9_RA__Pr(obj, id);
  if (addr == 0) {
   if (id > 0 && id < PARAM_1_indiv_prop_start) {
     return PARAM_8_cpv__start-->id;
   }
   ERROR("[** Programming error: tried to read (something) **]");
   return 0;
  }
  return addr-->0;
];

! FUNC_13_OP__Pr: implements OP__Pr() as of Inform 6.33.
[ FUNC_13_OP__Pr obj id
  zr; ! locals
  zr = FUNC_1_Z__Region(obj);
  if (zr == 3) {
   if (id == print or print_to_array) rtrue;
   rfalse;
  }
  if (zr == 2) {
   if (id == call) rtrue;
   rfalse;
  }
  if (zr ~= 1) rfalse;
  if (id >= PARAM_1_indiv_prop_start
      && id < PARAM_1_indiv_prop_start+8) {
   if (OBJ_IN_CLASS(obj)) rtrue;
  }
  if (FUNC_9_RA__Pr(obj, id) ~= 0)
   rtrue;
  rfalse;
];
```

2.18. Miscellaneous

```
nop
```

Do nothing.

```
gestalt L1 L2 S1
```

Test the Gestalt selector number L1, with optional extra argument L2, and store the result in S1. If the selector is not known, store zero.

The reasoning behind the design of a Gestalt system is, I hope, too obvious to explain.

[This list of Gestalt selectors has nothing to do with the list in the Glk library.]

The list of L1 selectors is as follows. Note that if a selector does not mention L2, you should always set that argument to zero. *[This will ensure future compatibility, in case the selector definition is extended.]*

- GlulxVersion (0): Returns the version of the Glulx spec which the interpreter implements. The upper 16 bits of the value contain a major version number; the next 8 bits contain a minor version number; and the lowest 8 bits contain an even more minor version number, if any. This specification is version 3.1.2, so a terp implementing it would return 0x00030102. I will try to maintain the convention that minor version changes are backwards compatible, and subminor version changes are backwards and forwards compatible.
- TerpVersion (1): Returns the version of the interpreter. The format is the same as the GlulxVersion. *[Each interpreter has its own version numbering system, defined by its author, so this information is not terribly useful. But it is convenient for the game to be able to display it, in case the player is capturing version information for a bug report.]*
- ResizeMem (2): Returns 1 if the terp has the potential to resize the memory map, with the setmemsize opcode. If this returns 0, setmemsize will always fail. *[But remember that setmemsize might fail in any case.]*
- Undo (3): Returns 1 if the terp has the potential to undo. If this returns 0, saveundo and restoreundo will always fail.
- IOSystem (4): Returns 1 if the terp supports the I/O system given in L2. (The constants are the same as for the setiosys opcode: 0 for null, 1 for filter, 2 for Glk, 20 for FyreVM. 0 and 1 will always succeed.)
- Unicode (5): Returns 1 if the terp supports Unicode operations. These are: the E2 Unicode string type; the 04 and 05 string node types (in compressed strings); the streamunichar opcode; the type-14 call stub. If the Unicode selector returns 0, encountering any of these will cause a fatal interpreter error.
- MemCopy (6): Returns 1 if the interpreter supports the mzero and mcopy opcodes. (This must true for any terp supporting Glulx 3.1.)
- MAlloc (7): Returns 1 if the interpreter supports the malloc and mfree opcodes. (If this is true, MemCopy and ResizeMem must also both be true, so there is no need to check all three.)
- MAllocHeap (8): Returns the start address of the heap. This is the value that getmemsize had when the first memory block was allocated. If the heap is not active (no blocks are extant), this returns zero.
- Acceleration (9): Returns 1 if the interpreter supports the accelfunc and accelparam opcodes. (This must true for any terp supporting Glulx 3.1.1.)
- AccelFunc (10): Returns 1 if the terp implements the accelerated function given in L2.
- Float (11): Returns 1 if the interpreter supports the floating-point arithmetic opcodes.

Selectors 0x1000 to 0x10FF are reserved for use by FyreVM. Selectors 0x1100 to 0x11FF are reserved for extension projects by Dannii Willis. Selectors 0x1200 to 0x12FF are reserved for iOS extension features by Andrew Plotkin. These are not documented here. See section 0.2, "Glulx and Other IF Systems".

[The Unicode selector is slightly redundant. Since the Unicode operations exist in Glulx spec 3.0 and higher, you can get the same information by testing GlulxVersion against 0x00030000. However, it's clearer to have a separate selector. Similarly, the MemCopy selector is true exactly when GlulxVersion is 0x00030100 or higher.]

[The Unicode selector does not *guarantee that your Glk library supports Unicode. For that, you must check the Glk gestalt selector gestalt_Unicode. If the Glk library is non-Unicode, the Glulx Unicode operations are still legal; however, Unicode characters (beyond FF) will be printed as 3F ("?").]*

```
debugtrap L1
```

Interrupt execution to do something interpreter-specific with L1. If the interpreter has nothing in mind, it should halt with a visible error message.

[This is intended for use by debugging interpreters. The program might be sprinkled with consistency tests, set to call debugtrap if an assertion failed. The interpreter could then be set to halt, display a warning, or ignore the debugtrap.]

This should *not* be used as an arbitrary interpreter trap-door in a finished (non-debugging) program. If you really want to add interpreter functionality to your program, and you're willing to support an alternate interpreter to run it, you should add an entirely new opcode. There are still 2^28 of them available, give or take.

```
glk L1 L2 S1
```

Call the Glk API function whose identifier is L1, passing in L2 arguments. The return value is stored at S1. (If the Glk function has no return value, zero is stored at S1.)

The arguments are passed on the stack, last argument pushed first, just as for the call opcode.

Arguments should be represented in the obvious way. Integers and character are passed as integers. Glk opaque objects are passed as integer identifiers, with zero representing NULL. Strings and Unicode strings are passed as the addresses of Glulx string objects (see section 1.6.1, "Strings".) References to values are passed by their addresses. Arrays are passed by their addresses; note that an array argument, unlike a string argument, is always followed by an array length argument.

Reference arguments require more explanation. A reference to an integer or opaque object is the address of a 32-bit value (which, being in main memory, does not have to be aligned, but must be big-endian.) Alternatively, the value -1 (FFFFFFFF) may be passed; this is a special case, which means that the value is read from or written to the stack. Arguments are always evaluated left to right, which means that input arguments are popped from the stack first-topmost, but output arguments are pushed on last-topmost.

A reference to a Glk structure is the address of an array of 32-bit values in main memory. Again, -1 means that all the values are written to the stack. Also again, an input structure is popped off first-topmost, and an output structure is pushed on last-topmost.

All stack input references (-1 addresses) are popped after the Glk argument list is popped. *[This should be obvious, since the -1 occurs* in *the Glk argument list.]* Stack output references are pushed after the Glk call, but before the S1 result value is stored.

[The difference between strings and character arrays is somewhat confusing. These are the same type in the C Glk API, but different in Glulx. Calls such as glk_put_buffer() and glk_request_line_event() take character arrays; this is the address of a byte array containing character values, followed by an integer array length. The byte array itself has neither a length field or a terminator. In contrast, calls such as glk_put_string() and glk_fileref_create_by_name() take string arguments, which must be unencoded Glulx string objects. An unencoded Glulx string object is nearly a byte array, but not quite; it has an E0 byte at the beginning and a zero byte at the end. Similarly, calls such as glk_put_string_uni() take unencoded (E2) Unicode objects.]

[Previous versions of this spec said that string arguments could be unencoded or encoded *string objects. This use of encoded strings has never been supported, however, and it is withdrawn from the spec.]*

[The convention that "address" -1 refers to the stack is a feature of the Glk invocation mechanism; it applies only to Glk arguments. It is not *part of the general Glulx definition. When instruction operands are being evaluated, -1 has no special meaning. This includes the L1, L2, and S1 arguments of the glk opcode.]*

2.19. Assembly Language

The format used by Inform is acceptable for now:

```
@opcode [ op op op ... ] ;
```

Where each "op" is a constant, the name of a local variable, the name of a global variable, or "sp" (for stack push/pop modes).

[It would be convenient to have a one-line form for the opcodes that pass arguments on the stack (call and glk).]

To make life a little easier for cross-platform I6 code, Inform accepts the macro "@push val" for "@copy val sp", and "@pull val" for "@copy sp val". Supporting these forms is recommended.

You can synthesize opcodes that the compiler does not know about:

```
@"FlagsCount:Code" [ op op op ... ] ;
```

The optional Flags can include "S" if the last operand is a store; "SS" if the last two operands are stores; "B" for branch format; "R" if execution never continues after the opcode. The Count is the number of arguments (0 to 9). The Code is a decimal integer representing the opcode number. So these two lines generate the same code:

```
@add x 1 y;
@"S3:16" x 1 y;
```

...because the @add opcode has number 16 (decimal), and has format "@add L1 L2 S1".