Dis Virtual Machine Specification

Lucent Technologies Inc
30 September 1999

Extensively revised by Vita Nuova Limited
5 June 2000, 9 January 2003

1. Introduction

The Dis virtual machine provides the execution environment for programs running under the Inferno operating system. The virtual machine models a CISC-like, three operand, memory-to-memory architecture. Code can either be interpreted by a C library or compiled on-the-fly into machine code for the target architecture.

This paper defines the virtual machine informally. A separate paper by Winterbottom and Pike[2] discusses its design. The Dis object file format is also defined here. Literals and keywords are in `typewriter` typeface.

2. Addressing Modes

Operand Size

Operand sizes are defined as follows: a byte is 8 bits, a word or pointer is 32 bits, a float is 64 bits, a big integer is 64 bits. The operand size of each instruction is encoded explicitly by the operand code. The operand size and type are specified by the last character of the instruction mnemonic:

`W`	word, 32-bit two's complement
`B`	byte, 8-bit unsigned
`F`	float, 64-bit IEEE format
`L`	big, 64-bit two's complement
`P`	pointer
`C`	Unicode string encoded in UTF-8
`M`	memory
`MP`	memory containing pointers

Two more operand types are defined to provide 'short' types for use by languages other than Limbo: signed 16-bit integers, called 'short word' here, and 32-bit IEEE format floating-point numbers, called 'short float' or 'short real' here. Support for them is limited to conversion to and from words or floats respectively; the instructions are marked below with a dagger (†).

Memory Organization

Memory for a thread is divided into several separate regions. The code segment stores either a decoded virtual machine instruction stream suitable for execution by the interpreter or flash compiled native machine code for the host CPU. Neither type of code segment is addressable from the instruction set. At the object code level, PC values are offsets, counted in instructions, from the beginning of the code space.

Data memory is a linear array of bytes, addressed using 32-bit pointers. Words are stored in the native representation of the host CPU. Data types larger than a byte must be stored at addresses aligned to a multiple of the data size. A thread executing a module has access to two regions of addressable data memory. A module pointer (`mp` register) defines a region of global storage for a particular module, a frame pointer (`fp` register) defines the current activation record or frame for the thread. Frames are allocated dynamically from a stack by function call and return instructions. The stack is extended automatically from the heap.

The `mp` and `fp` registers cannot be addressed directly, and therefore, can be modified only by call and return instructions.

Effective Addresses

Each instruction can potentially address three operands. The source and destination operands are general, but the middle operand can use any address mode except double indirect. If the middle operand of a three address instruction is omitted, it is assumed to be the same as the destination operand.

The general operands generate an effective address from three basic modes: immediate, indirect and double indirect. The assembler syntax for each mode is:

`10(fp)`	30- bitsigned indirect from fp
`20(mp)`	30- bitsigned indirect from mp
`$0x123`	30- bitsigned immediate value
`10(20(fp))`	two 16- bitunsigned offsets double indirect from fp
`10(20(mp))`	two 16- bitunsigned offsets double indirect from mp

Garbage Collection

The Dis machine performs both reference counted and real time mark and sweep garbage collection. This hyrbrid approach allows code to be generated in several styles: pure reference counted, mark and sweep, or a hybrid of the two approaches. Compiler writers have the freedom to choose how specific types are handled by the machine to optimize code for performance or language implementation. Instruction selection determines which algorithm will be applied to specific types.

When using reference counting, pointers are a special operand type and should only be manipulated using the pointer instructions in order to ensure the correct functioning of the garbage collector. Every memory location that stores a pointer must be known to the interpreter so that it can be initialized and deallocated correctly. The information is transmitted in the form of type descriptors in the object module. Each type descriptor contains a bit vector for a particular type where each bit corresponds to a word in memory. Type descriptors are generated automatically by the Limbo compiler. The assembler syntax for a type descriptor is:

```
desc    $10, 132, "001F"
```

The first parameter is the descriptor number, the second is the size in bytes, and the third a pointer map. The map contains a list of hex bytes where each byte maps eight 32 bit words. The most significant bit represents the lowest memory address. A one bit indicates a pointer in memory. The map need not have an entry for every byte and unspecified bytes are assumed zero.

Throughout this description, the symbolic constant `H` refers to a nil pointer.

3. Instruction Set

add*x* – Add

```
Syntax:     addb    src1, src2, dst
        addf    src1, src2, dst
        addw    src1, src2, dst
        addl    src1, src2, dst
Function:   dst = src1 + src2
```

The `add` instructions compute the sum of the operands addressed by `src1` and `src2` and stores the result in the `dst` operand. For `addb` the result is truncated to eight bits.

addc – Add strings

```
Syntax:     addc    src1, src2, dst
Function:   dst = src1 + src2
```

The `addc` instruction concatenates the two UTF strings pointed to by `src1` and `src2`; the result is placed in the pointer addressed by `dst`. If both pointers are `H` the result will be a zero length string rather than `H`.

alt – Alternate between communications

```
Syntax:     alt src, dst
```

The `alt` instruction selects between a set of channels ready to communicate. The `src` argument is the address of a structure of the following form:

```
struct Alt {
    int nsend;      /* Number of senders */
    int nrecv;      /* Number of receivers */
    struct {
        Channel* c;     /* Channel */
        void*   val;    /* Address of lval/rval */
    } entry[];
};
```

The vector is divided into two sections; the first lists the channels ready to send values, the second lists channels either ready to receive or an array of channels each of which may be ready to receive. The counts of the sender and receiver channels are stored as the first and second words addressed by `src`. An `alt` instruction proceeds by testing each channel for readiness to communicate. A ready channel is added to a list. If the list is empty after each channel has been considered, the thread blocks at the `alt` instruction waiting for a channel to become ready; otherwise, a channel is picked at random from the ready set.

The `alt` instruction then uses the selected channel to perform the communication using the `val` address as either a source for send or a destination for receive. The numeric index of the selected vector element is placed in `dst`.

and*x* – **Logical AND**

```
Syntax:     andb    src1, src2, dst
        andw    src1, src2, dst
        andl    src1, src2, dst
Function:   dst = src1 & src2
```

The instructions compute the bitwise AND of the two operands addressed by `src1` and `src2` and stores the result in the `dst` operand.

beq*x* – **Branch equal**

```
Syntax:     beqb    src1, src2, dst
        beqc    src1, src2, dst
        beqf    src1, src2, dst
        beqw    src1, src2, dst
        beql    src1, src2, dst
Function:   if src1 == src2 then pc = dst
```

If the `src1` operand is equal to the `src2` operand, then control is transferred to the program counter specified by the `dst` operand.

bge*x* – **Branch greater or equal**

```
Syntax:     bgeb    src1, src2, dst
        bgec    src1, src2, dst
        bgef    src1, src2, dst
        bgew    src1, src2, dst
        bgel    src1, src2, dst
Function:   if src1 >= src2 then pc = dst
```

If the `src1` operand is greater than or equal to the `src2` operand, then control is transferred to program counter specified by the `dst` operand. This instruction performs a signed comparison.

bgt*x* – **Branch greater**

```
Syntax:     bgtb    src1, src2, dst
        bgtc    src1, src2, dst
        bgtf    src1, src2, dst
        bgtw    src1, src2, dst
        bgtl    src1, src2, dst
Function:   if src1 > src2 then pc = dst
```

If the `src1` operand is greater than the `src2` operand, then control is transferred to the program counter specified by the `dst` operand. This instruction performs a signed comparison.

ble*x* – Branch less than or equal

```
Syntax:     bleb    src1, src2, dst
        blec    src1, src2, dst
        blef    src1, src2, dst
        blew    src1, src2, dst
        blel    src1, src2, dst
Function:   if src1 <= src2 then pc = dst
```

If the `src1` operand is less than or equal to the `src2` operand, then control is transferred to the program counter specified by the `dst` operand. This instruction performs a signed comparison.

blt*x* – Branch less than

```
Syntax:     bltb    src1, src2, dst
        bltc    src1, src2, dst
        bltf    src1, src2, dst
        bltw    src1, src2, dst
        bltl    src1, src2, dst
Function:   if src1 < src2 then pc = dst
```

If the `src1` operand is less than the `src2` operand, then control is transferred to the program counter specified by the `dst` operand.

bne*x* – Branch not equal

```
Syntax:     bneb    src1, src2, dst
        bnec    src1, src2, dst
        bnef    src1, src2, dst
        bnew    src1, src2, dst
        bnel    src1, src2, dst
Function:   if src1 != src2 then pc = dst
```

If the `src1` operand is not equal to the `src2` operand, then control is transferred to the program counter specified by the `dst` operand.

call – Call local function

```
Syntax:     call    src, dst
Function:   link(src) = pc
        frame(src) = fp
        mod(src) = 0
        fp = src
        pc = dst
```

The `call` instruction performs a function call to a routine in the same module. The `src` argument specifies a frame created by `new`. The current value of `pc` is stored in link(src), the current value of `fp` is stored in frame(src) and the module link register is set to 0. The value of `fp` is then set to `src` and control is transferred to the program counter specified by `dst`.

case – Case compare integer and branch

```
Syntax:     case    src, dst
Function:   pc = 0..i: dst[i].pc where
          dst[i].lo >= src && dst[i].hi < src
```

The `case` instruction jumps to a new location specified by a range of values. The `dst` operand points to a table in memory containing a table of `i` values. Each value is three words long: the first word specifies a low value, the second word specifies a high value, and the third word specifies a program counter. The first word of the table gives the number of entries. The `case` instruction searches the table for the first matching value where the `src` operand is greater than or equal to the low word and less than the high word. Control is transferred to the program counter stored in the first word of the matching entry.

casec – Case compare string and branch

```
Syntax:     casec   src, dst
Function:   pc = 0..i: dst[i].pc where
           dst[i].lo >= src && dst[i].hi < src
```

The `casec` instruction jumps to a new location specified by a range of string constants. The table is the same as described for the `case` instruction.

cons*x* – Allocate new list element

```
Syntax:     consb   src, dst
        consc   src, dst
        consf   src, dst
        consl   src, dst
        consm   src, dst
        consmp  src, dst
        consp   src, dst
        consw   src, dst
Function:   p = new(src, dst)
        dst = p
```

The `cons` instructions add a new element to the head of a list. A new list element is composed from the `src` operand and a pointer to the head of an extant list specified by `dst`. The resulting element is stored back into `dst`.

cvtac – Convert byte array to string

```
Syntax:     cvtac   src, dst
Function:   dst = string(src)
```

The `src` operand must be an array of bytes, which is converted into a character string and stored in `dst`. The new string is a copy of the bytes in `src`.

cvtbw – Convert byte to word

```
Syntax:     cvtbw   src, dst
Function:   dst = src & 0xff
```

A byte is fetched from the `src` operand extended to the size of a word and then stored into `dst`.

cvtca – Convert string to byte array

```
Syntax:     cvtca   src, dst
Function:   dst = array(src)
```

The `src` operand must be a string which is converted into an array of bytes and stored in `dst`. The new array is a copy of the characters in src.

cvtcf – Convert string to real

```
Syntax:     cvtcf   src, dst
Function:   dst = (float)src
```

The string addressed by the `src` operand is converted to a floating point value and stored in the `dst` operand. Initial white space is ignored; conversion ceases at the first character in the string that is not part of the representation of the floating point value.

cvtcl – Convert string to big

```
Syntax:     cvtcl   src, dst
Function:   dst = (big)src
```

The string addressed by the `src` operand is converted to a big integer and stored in the `dst` operand. Initial white space is ignored; conversion ceases at the first non- digitin the string.

cvtcw – Convert string to word

```
Syntax:     cvtcw   src, dst
Function:   dst = (int)src
```

The string addressed by the `src` operand is converted to a word and stored in the `dst` operand. Initial white space is ignored; after a possible sign, conversion ceases at the first non- digitin the string.

cvtfc – Convert real to string

```
Syntax:     cvtfc   src, dst
Function:   dst = string(src)
```

The floating point value addressed by the `src` operand is converted to a string and stored in the `dst` operand. The string is a floating point representation of the value.

cvtfw – Convert real to word

```
Syntax:     cvtfw   src, dst
Function:   dst = (int)src
```

The floating point value addressed by `src` is converted into a word and stored into `dst`. The floating point value is rounded to the nearest integer.

cvtfl – Convert real to big

```
Syntax:     cvtfl   src, dst
Function:   dst = (big)src
```

The floating point value addressed by `src` is converted into a big integer and stored into `dst`. The floating point value is rounded to the nearest integer.

cvtfr – Convert real to short real†

```
Syntax:     cvtfr   src, dst
Function:   dst = (short float)src
```

The floating point value addressed by `src` is converted to a short (32- bit)floating point value and stored into `dst`. The floating point value is rounded to the nearest integer.

cvtlc – Convert big to string

```
Syntax:     cvtlc   src, dst
Function:   dst = string(src)
```

The big integer addressed by the `src` operand is converted to a string and stored in the `dst` operand. The string is the decimal representation of the big integer.

cvtlw – Convert big to word

```
Syntax:     cvtlw   src, dst
Function:   dst = (int)src
```

The big integer addressed by the `src` operand is converted to a word and stored in the `dst` operand.

cvtsw – Convert short word to word†

```
Syntax:     cvtsw   src, dst
Function:   dst = (int)src
```

The short word addressed by the `src` operand is converted to a word and stored in the `dst` operand.

cvtwb – Convert word to byte

```
Syntax:     cvtwb   src, dst
Function:   dst = (byte)src;
```

The `src` operand is converted to a byte and stored in the `dst` operand.

cvtwc – Convert word to string

```
Syntax:     cvtwc   src, dst
Function:   dst = string(src)
```

The word addressed by the `src` operand is converted to a string and stored in the `dst` operand. The string is the decimal representation of the word.

cvtwl – Convert word to big

```
Syntax:     cvtwl   src, dst
Function:   dst = (big)src;
```

The word addressed by the `src` operand is converted to a big integer and stored in the `dst` operand.

cvtwf – Convert word to real

```
Syntax:     cvtwf   src, dst
Function:   dst = (float)src;
```

The word addressed by the `src` operand is converted to a floating point value and stored in the `dst` operand.

cvtws – Convert word to short word†

```
Syntax:     cvtws   src, dst
Function:   dst = (short)src;
```

The word addressed by the `src` operand is converted to a short word and stored in the `dst` operand.

cvtlf – Convert big to real

```
Syntax:     cvtlf   src, dst
Function:   dst = (float)src;
```

The big integer addressed by the `src` operand is converted to a floating point value and stored in the `dst` operand.

cvtrf – Convert short real to real†

```
Syntax:     cvtrf   src, dst
Function:   dst = (float)src;
```

The short (32 bit) floating point value addressed by the `src` operand is converted to a 64- bitfloating point value and stored in the `dst` operand.

div*x* – Divide

```
Syntax:     divb    src1, src2, dst
        divf    src1, src2, dst
        divw    src1, src2, dst
        divl    src1, src2, dst
Function:   dst = src2/src1
```

The `src2` operand is divided by the `src1` operand and the quotient is stored in the `dst` operand. Division by zero causes the thread to terminate.

eclr – Clear exception stack

```
Syntax:     eclr
```

Exception handling is implemented by system primitives outside the scope of this description. The `eclr` instruction is reserved for internal use by the implementation, to cancel exception handlers on return from a function; it does not appear in the instruction stream.

exit – Terminate thread

```
Syntax:     exit
Function:   exit()
```

The executing thread terminates. All resources held in the stack are deallocated.

frame – Allocate frame for local call

```
Syntax:     frame   src1, src2
Function:   src2 = fp + src1->size
        initmem(src2, src1);
```

The frame instruction creates a new stack frame for a call to a function in the same module. The frame is initialized according to the type descriptor supplied as the `src1` operand. A pointer to the newly created frame is stored in the `src2` operand.

goto – Computed goto

```
Syntax:     goto    src, dst
Function:   pc = dst[src]
```

The `goto` instruction performs a computed goto. The `src` operand must be an integer index into a table of PC values specified by the `dst` operand.

head*x* – **Head of list**

```
Syntax:     headb   src, dst
        headf   src, dst
        headm   src, dst
        headmp  src, dst
        headp   src, dst
        headw   src, dst
        headl   src, dst
Function:   dst = hd src
```

The `head` instructions make a copy of the first data item stored in a list. The `src` operand must be a list of the correct type. The first item is copied into the `dst` operand. The list is not modified.

indc – Index by character

```
Syntax:     indc    src1, src2, dst
Function:   dst = src1[src2]
```

The `indc` instruction indexes Unicode strings. The `src1` instruction must be a string. The `src2` operand must be an integer specifying the origin-0 index in `src1` of the (Unicode) character to store in the `dst` operand.

indx – Array index

```
Syntax:     indx    src1, dst, src2
Function:   dst = &src1[src2]
```

The `indx` instruction computes the effective address of an array element. The `src1` operand must be an array created by the `newa` instruction. The `src2` operand must be an integer. The effective address of the `src2` element of the array is stored in the `dst` operand.

ind*x* – **Index by type**

```
Syntax:     indb    src1, dst, src2
        indw    src1, dst, src2
        indf    src1, dst, src2
        indl    src1, dst, src2
Function:   dst = src1[src2]
```

The indb, indw, indf and indl instructions index arrays of the basic types. The src1 operand must be an array created by the newa instruction. The src2 operand must be a non- negativeinteger index less than the array size. The value of the element at the index is loaded into the dst operand.

insc – Insert character into string

```
Syntax:     insc    src1, src2, dst
Function:   src1[src2] = dst
```

The insc instruction inserts a character into an existing string. The index in src2 must be a non- negative integer less than the length of the string plus one. (The character will be appended to the string if the index is equal to the string's length.) The src1 operand must be a string (or nil). The character to insert must be a valid 16- bitunicode value represented as a word.

jmp – Branch always

```
Syntax:     jmp dst
Function:   pc = dst
```

Control is transferred to the location specified by the dst operand.

lea – Load effective address

```
Syntax:     lea src, dst
Function:   dst = &src
```

The lea instruction computes the effective address of the src operand and stores it in the dst operand.

lena – Length of array

```
Syntax:     lena    src, dst
Function:   dst = nelem(src)
```

The lena instruction computes the length of the array specified by the src operand and stores it in the dst operand.

lenc – Length of string

```
Syntax:     lenc    src, dst
Function:   dst = utflen(src)
```

The lenc instruction computes the number of characters in the UTF string addressed by the src operand and stores it in the dst operand.

lenl – Length of list

```
Syntax:     lenl    src, dst
Function:   dst = 0;
        for(l = src; l; l = tl l)
            dst++;
```

The lenl instruction computes the number of elements in the list addressed by the src operand and stores the result in the dst operand.

load – Load module

```
Syntax:     load    src1, src2, dst
Function:   dst = load src2 src1
```

The load instruction loads a new module into the heap. The module might optionally be compiled into machine code depending on the module header. The src1 operand is a pathname to the file containing the object code for the module. The src2 operand specifies the address of a linkage descriptor for the module (see below). A reference to the newly loaded module is stored in the dst operand. If the module could not be loaded for any reason, then dst will be set to H.

The linkage descriptor referenced by the src2 operand is a table in data space that lists the functions imported by the current module from the module to be loaded. It has the following layout:

```
int nentries;
struct {    /* word aligned */
    int sig;
    byte    name[]; /* UTF encoded name, 0-terminated */
} entry[];
```

The `nentries` value gives the number of entries in the table and can be zero. It is followed by that many linkage entries. Each entry is aligned on a word boundary; there can therefore be padding before each structure. The entry names the imported function in the UTF- encodedstring in `name`, which is terminated by a byte containing zero. The MD5 hash of the function's type signature is given in the value `sig`. For each entry, `load` instruction checks that a function with the same name in the newly loaded exists, with the same signature. Otherwise the load will fail and `dst` will be set to `H`.

The entries in the linkage descriptor form an array of linkage records (internal to the virtual machine) associated with the module pointer returned in `dst`, that is indexed by operators `mframe`, `mcall` and `mspawn` to refer to functions in that module. The linkage scheme provides a level of indirection that allows a module to be loaded using any module declaration that is a valid subset of the implementation module's declaration, and allows entry points to be added to modules without invalidating calling modules.

lsr*x* – Logical shift right

```
Syntax:     lsrw    src1, src2, dst
        lsrl    src1, src2, dst
Function:   dst = (unsigned)src2 >> src1
```

The `lsr` instructions shift the `src2` operand right by the number of bits specified by the `src1` operand, replacing the vacated bits by 0, and store the result in the `dst` operand. Shift counts less than 0 or greater than the number of bits in the object have undefined results. This instruction is included for support of languages other than Limbo, and is not used by the Limbo compiler.

mcall – Inter-module call

```
Syntax:     mcall   src1, src2, src3
Function:   link(src1) = pc
        frame(src1) = fp
        mod(src1) = current_moduleptr
        current_moduleptr = src3->moduleptr
        fp = src1
        pc = src3->links[src2]->pc
```

The `mcall` instruction calls a function in another module. The first argument specifies a new frame for the called procedure and must have been built using the `mframe` instruction. The `src3` operand is a module reference generated by a successful `load` instruction. The `src2` operand specifies the index for the called function in the array of linkage records associated with that module reference (see the `load` instruction).

mframe – Allocate inter-module frame

```
Syntax:     mframe  src1, src2, dst
Function:   dst = fp + src1->links[src2]->t->size
        initmem(dst, src1->links[src2])
```

The `mframe` instruction allocates a new frame for a procedure call into another module. The `src1` operand specifies the location of a module pointer created as the result of a successful load instruction. The `src2` operand specifies the index for the called function in the array of linkage records associated with that module pointer (see the `load` instruction). A pointer to the initialized frame is stored in `dst`. The `src2` operand specifies the linkage number of the function to be called in the module specified by `src1`.

mnewz – Allocate object given type from another module

```
Syntax:     mnewz   src1, src2, dst
Function:   dst = malloc(src1->types[src2]->size)
        initmem(dst, src1->types[src2]->map)
```

The `mnewz` instruction allocates and initializes storage to a new area of memory. The `src1` operand specifies the location of a module pointer created as the result of a successful load instruction. The size of the

new memory area and the location of pointers within it are specified by the `src2` operand, which gives a type descriptor number within that module. Space not occupied by pointers is initialized to zero. A pointer to the initialized object is stored in `dst`. This instruction is not used by Limbo; it was added to implement other languages.

mod*x* – Modulus

```
Syntax:     modb    src1, src2, dst
        modw    src1, src2, dst
        modl    src1, src2, dst
Function:   dst = src2 % src1
```

The modulus instructions compute the remainder of the `src2` operand divided by the `src1` operand and store the result in `dst`. The operator preserves the condition that the absolute value of a%b is less than the absolute value of `b`; `(a/b)*b + a%b` is always equal to `a`.

mov*x* – Move scalar

```
Syntax:     movb    src, dst
        movw    src, dst
        movf    src, dst
        movl    src, dst
Function:   dst = src
```

The move operators perform assignment. The value specified by the `src` operand is copied to the `dst` operand.

movm – Move memory

```
Syntax:     movm    src1, src2, dst
Function:   memmove(&dst, &src1, src2)
```

The `movm` instruction copies memory from the `src1` operand to the `dst` operand for `src2` bytes. The `src1` and `dst` operands specify the effective address of the memory rather than a pointer to the memory.

movmp – Move memory and update reference counts

```
Syntax:     movmp   src1, src2, dst
Function:   decmem(&dst, src2)
        memmove(&dst, &src1, src2->size)
        incmem(&src, src2)
```

The `movmp` instructions performs the same function as the `movm` instruction but increments the reference count of pointers contained in the data type. For each pointer specified by the `src2` type descriptor, the corresponding pointer reference count in the destination is decremented. The `movmp` instruction then copies memory from the `src1` operand to the `dst` operand for the number of bytes described by the type descriptor. For each pointer specified by the type descriptor the corresponding pointer reference count in the source is incremented.

movp – Move pointer

```
Syntax:     movp    src, dst
Function:   destroy(dst)
        dst = src
        incref(src)
```

The `movp` instruction copies a pointer adjusting the reference counts to reflect the new pointers.

movpc – Move program counter

```
Syntax:     movpc   src, dst
Function:   dst = PC(src);
```

The `movpc` instruction computes the actual address of an immediate PC value. The `dst` operand is set to the actual machine address of the instruction addressed by the `src` operand. This instruction must be used to calculate PC values for computed branches.

mspawn – Module spawn function

```
Syntax:     mspawn  src1, src2, src3
Function:   fork();
        if(child){
            link(src1) = 0
            frame(src1) = 0
            mod(src1) = src3->moduleptr
            current_moduleptr = src3->moduleptr
            fp = src1
            pc = src3->links[src2]->pc
        }
```

The `mspawn` instruction creates a new thread, which starts executing a function in another module. The first argument specifies a new frame for the called procedure and must have been built using the `mframe` instruction. The `src3` operand is a module reference generated by a successful `load` instruction. The `src2` operand specifies the index for the called function in the array of linkage records associated with that module reference (see the `load` instruction above).

mul*x* - Multiply

```
Syntax:     mulb    src1, src2, dst
        mulw    src1, src2, dst
        mulf    src1, src2, dst
        mull    src1, src2, dst
Function:   dst = src1 * src2
```

The `src1` operand is multiplied by the `src2` operand and the product is stored in the `dst` operand.

nbalt – Non blocking alternate

```
Syntax:     nbalt   src, dst
```

The `nbalt` instruction has the same operands and function as `alt` , except that if no channel is ready to communicate, the instruction does not block. When no channels are ready, control is transferred to the PC in the last element of the table addressed by `dst`.

negf – Negate real

```
Syntax:     negf    src, dst
Function:   dst = -src
```

The floating point value addressed by the `src` operand is negated and stored in the `dst` operand.

new, newz – Allocate object

```
Syntax:     new src, dst
        newz    src, dst
Function:   dst = malloc(src->size);
        initmem(dst, src->map);
```

The `new` instruction allocates and initializes storage to a new area of memory. The size and locations of pointers are specified by the type descriptor number given as the `src` operand. A pointer to the newly allocated object is placed in `dst`. Any space not occupied by pointers has undefined value.

The `newz` instruction additionally guarantees that all non- pointervalues are set to zero. It is not used by Limbo.

newa, newaz – Allocate array

```
Syntax:     newa    src1, src2, dst
        newaz   src1, src2, dst
Function:   dst = malloc(src2->size * src1);
        for(i = 0; i < src1; i++)
            initmem(dst + i*src2->size, src2->map);
```

The newa instruction allocates and initializes an array. The number of elements is specified by the src1 operand. The type of each element is specified by the type descriptor number given as the src2 operand. Space not occupied by pointers has undefined value. The newaz instruction additionally guarantees that all non- pointervalues are set to zero; it is not used by Limbo.

newc*x* – Allocate channel

```
Syntax:     newcw   dst
        newcb   dst
        newcl   dst
        newcf   dst
        newcp   dst
        newcm   src, dst
        newcmp  src, dst
Function:   dst = new(Channel)
```

The newc instruction allocates a new channel of the specified type and stores a reference to the channel in dst. For the newcm instruction the source specifies the number of bytes of memory used by values sent on the channel (see the movm instruction above). For the newcmp instruction the first operand specifies a type descriptor giving the length of the structure and the location of pointers within the structure (see the movmp instruction above).

or*x* – Logical OR

```
Syntax:     orb src1, src2, dst
        orw src1, src2, dst
        orl src1, src2, dst
Function:   dst = src1 | src
```

These instructions compute the bitwise OR of the two operands addressed by src1 and src2 and store the result in the dst operand.

recv – Receive from channel

```
Syntax:     recv    src, dst
Function:   dst = <-src
```

The recv instruction receives a value from some other thread on the channel specified by the src operand. Communication is synchronous, so the calling thread will block until a corresponding send or alt is performed on the channel. The type of the received value is determined by the channel type and the dst operand specifies where to place the received value.

ret – Return from function

```
Syntax:     ret
Function:   npc = link(fp)
        mod = mod(fp)
        fp = frame(fp)
        pc = npc
```

The ret instruction returns control to the instruction after the call of the current function.

send – Send to channel

```
Syntax:     send    src, dst
Function:   dst <-= src
```

The send instruction sends a value from this thread to some other thread on the channel specified by the dst operand. Communication is synchronous so the calling thread will block until a corresponding recv or alt is performed on the channel. The type of the sent value is determined by the channel type and the dst operand specifies where to retrieve the sent value.

shl*x* – Shift left arithmetic

```
Syntax:     shlb    src1, src2, dst
        shlw    src1, src2, dst
        shll    src1, src2, dst
Function:   dst = src2 << src1
```

The `shl` instructions shift the `src2` operand left by the number of bits specified by the `src1` operand and store the result in the `dst` operand. Shift counts less than 0 or greater than the number of bits in the object have undefined results.

shr*x* – Shift right arithmetic

```
Syntax:     shrb    src1, src2, dst
        shrw    src1, src2, dst
        shrl    src1, src2, dst
Function:   dst = src2 >> src1
```

The `shr` instructions shift the `src2` operand right by the number of bits specified by the `src1` operand and store the result in the `dst` operand. Shift counts less than 0 or greater than the number of bits in the object have undefined results.

slicea – Slice array

```
Syntax:     slicea  src1, src2, dst
Function:   dst = dst[src1:src2]
```

The `slicea` instruction creates a new array, which contains the elements from the index at `src1` to the index `src2-1`. The new array is a reference array which points at the elements in the initial array. The initial array will remain allocated until both arrays are no longer referenced.

slicec – Slice string

```
Syntax:     slicec  src1, src2, dst
Function:   dst = dst[src1:src2]
```

The `slicec` instruction creates a new string, which contains characters from the index at `src1` to the index `src2-1`. Unlike `slicea` , the new string is a copy of the elements from the initial string.

slicela – Assign to array slice

```
Syntax:     slicela   src1, src2, dst
Function:   dst[src2:] = src1
```

The `src1` and `dst` operands must be arrays of equal types. The `src2` operand is a non- negative integer index. The `src1` array is assigned to the array slice `dst[src2:]`; `src2 + nelem(src1)` must not exceed `nelem(dst)`.

spawn – Spawn function

```
Syntax:     spawn   src, dst
Function:   fork();
        if(child)
            dst(src);
```

The `spawn` instruction creates a new thread and calls the function specified by the `dst` operand. The argument frame passed to the thread function is specified by the `src` operand and should have been created by the `frame` instruction.

sub*x* – Subtract

```
Syntax:     subb    src1, src2, dst
        subf    src1, src2, dst
        subw    src1, src2, dst
        subl    src1, src2, dst
Function:   dst = src2 - src1
```

The `sub` instructions subtract the operands addressed by `src1` and `src2` and stores the result in the `dst` operand. For `subb`, the result is truncated to eight bits.

tail – Tail of list

```
Syntax:     tail    src, dst
Function:   dst = src->next
```

The `tail` instruction takes the list specified by the `src` operand and creates a reference to a new list with the head removed, which is stored in the `dst` operand.

tcmp – Compare types

```
Syntax:     tcmp    src, dst
Function:   if(typeof(src) != typeof(dst))
            error("typecheck");
```

The `tcmp` instruction compares the types of the two pointers supplied by the `src` and `dst` operands. The comparison will succeed if the two pointers were created from the same type descriptor or the `src` operand is `nil`; otherwise, the program will error. The `dst` operand must be a valid pointer.

xor*x* – Exclusive OR

```
Syntax:     xorb    src1, src2, dst
        xorw    src1, src2, dst
        xorl    src1, src2, dst
Function:   dst = src1 ^ src2
```

These instructions compute the bitwise exclusive- ORof the two operands addressed by `src1` and `src2` and store the result in the `dst` operand.

4. Object File Format

An object file defines a single module. The file has the following structure:

```
Objfile
{
    Header;
    Code_section;
    Type_section;
    Data_section;
    Module_name;
    Link_section;
};
```

The following data types are used in the description of the file encoding:

`OP`	encoded integer operand, encoding selected by the two most significant bits as follows: 00 signed 7 bits, 1 byte 10 signed 14 bits, 2 bytes 11 signed 30 bits, 4 bytes
`B`	unsigned byte
`W`	32 bit signed integer
`F`	canonicalized 64- bitIEEE754 floating point value
`SO`	16 bit unsigned small offset from register
`SI`	16 bit signed immediate value
`LO`	30 bit signed large offset from register

All binary values are encoded in two's complement format, most significant byte first.

The Header Section

```
Header
{
    OP: magic_number;
    Signature;
    OP: runtime_flag;
    OP: stack_extent;
    OP: code_size;
    OP: data_size;
    OP: type_size;
    OP: link_size;
    OP: entry_pc;
    OP: entry_type;
};
```

The magic number is defined as 819248 (symbolically `XMAGIC`), for modules that have not been signed cryptographically, and 923426 (symbolically `SMAGIC`), for modules that contain a signature. On the Inferno system, the symbolic names `XMAGIC` and `SMAGIC` are defined by the C include file `/include/isa.h` and the Limbo module `/module/dis.m`.

The signature field is only present if the magic number is `SMAGIC`. It has the form:

```
Signature
{
    OP: length;
    array[length] of byte: signature;
};
```

A digital signature is defined by a length, followed by an array of untyped bytes. Data within the signature should identify the signing authority, algorithm, and data to be signed.

The `runtime_flag` is a bit mask that defines various execution options for a Dis module. The flags currently defined are:

```
MUSTCOMPILE = 1<<0
DONTCOMPILE = 1<<1
SHAREMP     = 1<<2
```

The `MUSTCOMPILE` flag indicates that a `load` instruction should draw an error if the implementation is unable to compile the module into native instructions using a just- in- time compiler.

The `DONTCOMPILE` flag indicates that the module should not be compiled into native instructions, even though it is the default for the runtime environment. This flag may be set to allow debugging or to save memory.

The `SHAREMP` flag indicates that each instance of the module should use the same module data for all instances of the module. There is no implicit synchronization between threads using the shared data.

The `stack_extent` value indicates the number of bytes by which the thread stack of this module should be extended in the event that procedure calls exhaust the allocated stack. While stack extension is transparent to programs, increasing this value may improve the efficiency of execution at the expense of using more memory.

The `code_size` is a count of the number of instructions stored in the Code_section.

The `data_size` gives the size in bytes of the module's global data, which is initialized by evaluating the contents of the data section.

The `type_size` is a count of the number of type descriptors stored in the Type_section.

The `link_size` is a count of the number of external linkage directives stored in the Link_section.

The `entry_pc` is an integer index into the instruction stream that is the default entry point for this module. The `entry_pc` should point to the first instruction of a function. Instructions are numbered from a program counter value of zero.

The `entry_type` is the index of the type descriptor that corresponds to the function entry point set by `entry_pc`.

The Code Section

The code section describes a sequence of instructions for the virtual machine. An instruction is encoded as follows:

```
Instruction
{
    B: opcode;
    B: address_mode;
    Middle_data;
    Source_data;
    Dest_data;
};
```

The `opcode` specifies the instruction to execute, encoded as follows:

00 nop	20 headb	40 mulw	60 blew	80 shrl
01 alt	21 headw	41 mulf	61 bgtw	81 bnel
02 nbalt	22 headp	42 divb	62 bgew	82 bltl
03 goto	23 headf	43 divw	63 beqf	83 blel
04 call	24 headm	44 divf	64 bnef	84 bgtl
05 frame	25 headmp	45 modw	65 bltf	85 bgel
06 spawn	26 tail	46 modb	66 blef	86 beql
07 runt	27 lea	47 andb	67 bgtf	87 cvtlf
08 load	28 indx	48 andw	68 bgef	88 cvtfl
09 mcall	29 movp	49 orb	69 beqc	89 cvtlw
0A mspawn	2A movm	4A orw	6A bnec	8A cvtwl
0B mframe	2B movmp	4B xorb	6B bltc	8B cvtlc
0C ret	2C movb	4C xorw	6C blec	8C cvtcl
0D jmp	2D movw	4D shlb	6D bgtc	8D headl
0E case	2E movf	4E shlw	6E bgec	8E consl
0F exit	2F cvtbw	4F shrb	6F slicea	8F newcl
10 new	30 cvtwb	50 shrw	70 slicela	90 casec
11 newa	31 cvtfw	51 insc	71 slicec	91 indl
12 newcb	32 cvtwf	52 indc	72 indw	92 movpc
13 newcw	33 cvtca	53 addc	73 indf	93 tcmp
14 newcf	34 cvtac	54 lenc	74 indb	94 mnewz
15 newcp	35 cvtwc	55 lena	75 negf	95 cvtrf
16 newcm	36 cvtcw	56 lenl	76 movl	96 cvtfr
17 newcmp	37 cvtfc	57 beqb	77 addl	97 cvtws
18 send	38 cvtcf	58 bneb	78 subl	98 cvtsw
19 recv	39 addb	59 bltb	79 divl	99 lsrw
1A consb	3A addw	5A bleb	7A modl	9A lsrl
1B consw	3B addf	5B bgtb	7B mull	9B eclr
1C consp	3C subb	5C bgeb	7C andl	9C newz
1D consf	3D subw	5D beqw	7D orl	9D newaz
1E consm	3E subf	5E bnew	7E xorl	
1F consmp	3F mulb	5F bltw	7F shll	

The `address_mode` byte specifies the addressing mode of each of the three operands: middle, source and destination. The source and destination operands are encoded by three bits and the middle operand by two bits. The bits are packed as follows:

```
bit  7  6  5  4  3  2  1  0
    m1 m0 s2 s1 s0 d2 d1 d0
```

The middle operand is encoded as follows:

`00`	*none*	no middle operand
`01`	`$SI`	small immediate
`10`	`SO(FP)`	small offset indirect from FP
`11`	`SO(MP)`	small offset indirect from MP

The source and destination operands are encoded as follows:

000	LO(MP)	offset indirect from MP
001	LO(FP)	offset indirect from FP
010	$OP	30 bit immediate
011	*none*	no operand
100	SO(SO(MP))	double indirect from MP
101	SO(SO(FP))	double indirect from FP
110	*reserved*	
111	*reserved*	

The `middle_data` field is only present if the middle operand specifier of the address_mode is not 'none'. If the field is present it is encoded as an `OP`.

The `source_data` and `dest_data` fields are present only if the corresponding `address_mode` field is not 'none'. For offset indirect and immediate modes the field contains a single `OP`. For double indirect modes the values are encoded as two `OP` values: the first value is the register indirect offset, and the second value is the final indirect offset. The offsets for double indirect addressing cannot be larger than 16 bits.

The Type Section

The type section contains type descriptors describing the layout of pointers within data types. The format of each descriptor is:

```
Type_descriptor
{
    OP: desc_number;
    OP: size;
    OP: number_ptrs;
    array[number_ptrs] of B: map;
};
```

The `desc_number` is a small integer index used to identify the descriptor to instructions such as `new`.

The `size` field is the size in bytes of the memory described by this type.

The `number_ptrs` field gives the size in bytes of the `map` array.

The `map` array is a bit vector where each bit corresponds to a word in memory. The most significant bit corresponds to the lowest address. For each bit in the map, the word at the corresponding offset in the type is a pointer iff the bit is set to 1.

The Data Section

The data section encodes the contents of the `MP` data for the module. The section contains a sequence of items; each item contains a control byte and an offset into the section, followed by one or more data items. A control byte of zero marks the end of the data section. Otherwise, it gives the type of data to be loaded and selects between two representations of an item:

```
Short_item
{
    B: code;
    OP: offset;
    array[code & 16rF] of type[code>>4]: data;
};
Long_item
{
    B: code;
    OP: count;
    OP: offset;
    array[ndata] of type[code>>4]: data;
};
```

A `Short_item` is generated for 15 or fewer items, otherwise a `Long_item` is generated. In a `Long_item` the count field (bottom 4 bits of code) is set to zero and the count follows as an `OP`. The top 4 bits of code determine the type of the datum. The defined values are:

`0001`	8 bit bytes
`0010`	32 bit words
`0011`	utf encoded string
`0100`	real value IEEE754 canonical representation
`0101`	Array
`0110`	Set array address
`0111`	Restore load address
`1000`	64 bit big

The byte, word, real and big operands are encoded as sequences of bytes (of appropriate length) in big-endian form, converted to native format before being stored in the data space. The 'string' code takes a UTF-encoded sequence of `count` bytes, which is converted to an array of 16-bit Unicode values stored in an implementation-dependent structure on the heap; a 4-byte pointer to the string descriptor is stored in the data space. The 'array' code takes two 4-byte operands: the first is the index of the array's type descriptor in the type section; the second is the length of the array to be created. The result in memory is a 4-byte pointer to an implementation-dependent array descriptor in the heap.

Each item's data is stored at the address formed by adding the `offset` in that item to a base address maintained by the loader. Initially that address is the base of the data space of the module instance. A new base for loading subsequent items can be set or restored by the following operations, used to initialize arrays. The 'set array index' item must appear immediately following an 'array' item. Its operand is a 4-byte big-endian integer that gives an index into that array, at which address subsequent data should be loaded; the previous load address is stacked internally. Subsequent data will be loaded at offsets from the new base address. The 'restore load address' item has no operands; it pops a load address from the internal address stack and makes that the new base address.

The Module Name

The module name immediately follows the data section. It contains the name of the implementation module, in UTF encoding, terminated by a zero byte.

The Link Section

The link section contains an array of external linkage items: the list of functions exported by this module. Each item describes one exported function in the following form:

```
Linkage_item
{
    OP: pc;
    OP: desc_number;
    W: sig;
    array[] of byte: name;
};
```

The `pc` is the instruction number of the function's entry point. The `desc_number` is the index, in the type section, of the type descriptor for the function's stack frame. The `sig` word is a 32-bit hash of the function's type signature. Finally, the name of the function is stored as a variable length array of bytes in UTF-8 encoding, with the end of the array marked by a zero byte. The names of member functions of an exported adt are qualified by the name of the adt. The next linkage item, if any, follows immediately.

5. Symbol Table File Format

The object file format does not include type information for debuggers. The Limbo compiler can optionally produce a separate symbol table file. Its format is defined in the entry *sbl*(6) of [1].

6. References

1. *Inferno Programmer's Manual* (Third Edition), Volume 1 ('the manual'), Vita Nuova Holdings Limited, June 2000.
2. P Winterbottom and R Pike, ''The Design of the Inferno Virtual Machine'', reprinted in this volume.