

The Glasgow Virtual Machine Toolkit Manual

Mark Shannon

Abstract

The GVMT is a toolkit for building virtual machines. The GVMT does not dictate the overall design of the virtual machine, but it does eliminate a lot of implementation details.

The developer needs to develop an interpreter and supporting code, as normal. The toolkit handles garbage collection transparently and can automatically create a just-in-time compiler from the interpreter source code.

This manual covers the components of the Glasgow Virtual Machine Toolkit, hereafter called the GVMT, and their use.

Conversion of source-code to bytecodes is not handled by the GVMT, that is up to the developer.

Contents

1	Introduction	1
1.1	Example virtual machine	1
1.2	The Tools	2
1.2.1	The front-end compilers, GVMTC and GVMTIC	2
1.2.2	The code-generators, gvmtas and gvmtcc	3
1.2.3	Other tools	3
1.3	GVMT input files	4
2	The tools	5
2.1	Core tools	5
2.1.1	Interpreter description file	6
2.1.2	The interpreter generator GVMTIC	8
2.1.3	The C compiler GVMTC	9
2.1.4	The GSC assembler GVMTAS	10
2.1.5	The compiler generator GVMTCC	10
2.1.6	The linker GVMTLINK	10
2.1.7	lcc-gvmt	11
2.2	Other tools	11
2.2.1	The bytecode-processor generator GVMTXC	11
2.2.2	The object layout tool	12
3	GVMT Abstract Machine	13
3.1	Components	13
3.2	Threads	14
3.2.1	Execution Model	14
3.2.2	Functions	14
3.2.3	Interpreters	14
3.2.4	Compiled Code	15
3.2.5	Abstract Machine Instructions	15
3.3	The Stacks	16
3.3.1	The Data Stack	16

3.3.2	GC Safe points	16
3.3.3	The Control Stack	16
3.3.4	The Native Parameter Stack	17
3.3.5	The State Stack	17
3.3.6	Raise and Transfer	17
3.3.7	Thread-local variables	17
3.4	The heap	17
3.4.1	Shape	17
3.4.2	Garbage collection	18
3.4.3	Exception handling	18
3.5	Concurrency	19
3.5.1	Memory model	20
4	Building a VM using the Toolkit	21
4.1	Before you start	21
4.2	Defining the components	21
4.3	Debugging	22
4.4	Adding a compiler	22
5	User interface	23
5.1	User provided code	23
5.1.1	Garbage collection	23
5.1.2	High-Performance Garbage Collection Interface	24
5.1.3	The Marshalling Interface	24
5.1.4	Debugging The Interpreter	25
5.2	GVMT provided functions	25
5.2.1	The data stack	25
5.2.2	The garbage collector	26
5.2.3	Exception handling	27
5.2.4	Threading	27
A	Installing the GVMT	29
B	The Abstract Machine Instruction Set	30

Chapter 1

Introduction

The Glasgow Virtual Machine Toolkit, GVMT, is a toolkit for constructing high-performance virtual machines.

The GVMT can be divided into three parts:

- Tools for the creation of interpreters, compilers and other virtual machine components.
- Library code. Primarily for Garbage Collection.
- An interface between the user-defined parts of the virtual machine and the toolkit library.

All the components operate on a common abstract machine model. It is not necessary to understand this model to use the GVMT, but a knowledge of the abstract machine is useful to get the best out of the GVMT. The abstract machine, which is described in detail in Chapter 3, is a stack-based machine supporting concurrency and garbage-collection. It has an extensible instruction set. The instruction set is abstract in that it cannot be directed executed, but is used to defined the semantics of bytecodes and other code.

The tools provided by the GVMT can be split into front-end tools which compile source code to the abstract machine model, and back-end tools which convert from the abstract machine code to real machine code.

1.1 Example virtual machine

In order to help understand the GVMT, a simple Scheme implementation is included as an example, it can be found in the `/example` directory. This example illustrates all the important aspects of the GVMT, while not being overly complex.

All scheme VMs are implemented by what is known as read-eval-print loop. It reads the input, evaluates the input and then prints the result, repeating until the interpreter exits. The ‘read’ part consists of reading text from a file or terminal and parsing that text to form a syntax tree.

The file `/example/parser.c` contains the parser, it is a standard C(89) source file which can be compiled with the GVMT C compiler, `GVMTCC`(see Section 2.1.3)

In some scheme implementations the ‘eval’ part is implemented by directly evaluating the syntax tree. This is not the case for the GVMT version, as all GVMT-based VM use bytecode interpreters. The GVMT scheme implementation first compiles the syntax tree to bytecode, then evaluates the resulting bytecode. The syntax-tree to bytecode compiler code can be found in `/example/compiler.c`.

The bytecode interpreter is defined in `/example/interpreter.vmc`

Although converting to bytecode is extra work it brings two benefits. The first is that the resulting bytecode can be optimised more easily than optimising the syntax tree directly. The second is that the GVMT can produce a bytecode to machine-code compiler automatically.

The Scheme optimiser is a series of passes, which use special interpreters created using `GVMTXC` (see Section 2.2.1). These passes are defined in the files of the form `/example/XXX.vmc` (except `/example/interpreter.vmc`).

The bytecode to machine-code, or JIT,¹ compiler is generated by the `GVMTCC` tool from the output of `GVMTIC`.

1.2 The Tools

The GVMT provides a number of tools, which are described in more detail in Chapter 2. These can be loosely grouped into front-end tools, and back-end tools. Front-end tools are responsible for compiling source code to the GVMT abstract machine code. Back-ends convert this abstract machine code into an executable virtual machine.

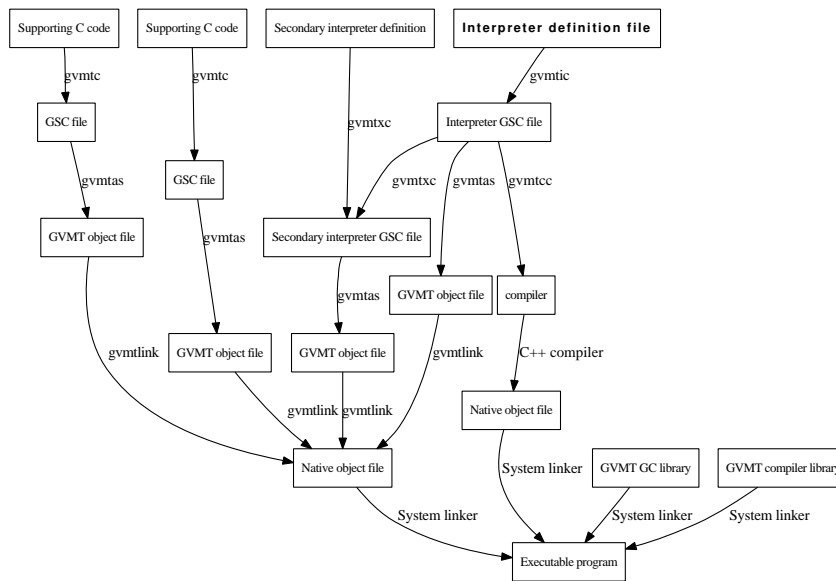
1.2.1 The front-end compilers, `gvmtc` and `gvmtic`

`GVMTCC` is a C compiler and compiles C files to GSC files. `GVMTIC` Takes an interpreter specification file and converts into a GSC file. The interpreter specification is in the form of a list of bytecodes; each of which consists of a name, a stack effect comment and a C code (or GSC) body.

¹JIT is short for Just-In-Time and is the standard term for runtime compilers

Both tools use a modified version of lcc (lcc-gvmt) to do the compilation. See section 2.1.7 for more details.

Figure 1.1: Building a Virtual Machine using GVMT



1.2.2 The code-generators, gvmtas and gvmtcc

The code-generators take GSC as input and create object files or executables.

GVMTAS compiles the GSC into the GVMT object format, GSO. It works by converting GSC back into C in such a way as to respect the semantics of the GVMT execution model, on a section-by-section basis. GVMTAS is also responsible for wrapping the bytecodes in an interpreter loop.

GVMTCC takes a GSC file containing a bytecode section and produces a bytecode to machine-code. Currently GVMTCC generates llvm-IR² and uses LLVM to generate machine-code at runtime (as a JIT compiler).

1.2.3 Other tools

GVMTXC takes a GSC file containing a bytecode section representing the main interpreter, plus a secondary file describing a secondary interpreter. It

²See llvm.org for more details

produces an interpreter (as a GSC file) which uses the bytecodes defined for the main interpreter, but with the semantics defined in the secondary file. GVMTXC can be used for creating bytecode verifiers and disassemblers, as well as optimisers and other abstract interpreters.

The bytecode-transformation engines produced by GVMTXC and the compilers produced by GVMTCC can be used independently or combined to form a specialised optimising compiler.

GVMTLINK links together all the GSO files produced by GVMTAS into a single object ready to be linked by the system linker. See Figure 1.1.

1.3 GVMT input files

GVMT takes a number of input files:

- Interpreter description file(s).
- Function definition file(s).

Chapter 2

The tools

GVMT is, as the name suggests, supplies a number of tools. Six of these form the core of the toolkit.

As mentioned in the previous chapter, the GVMT tools can be grouped into front-end and back-end tools. The front-end tools convert source code into code for the GVMT Abstract Machine. Back-end tools convert the abstract representation into real machine code.

2.1 Core tools

The six core tools are:

- Front end tools
 - GVMTIC
 - GVMTCC
 - GVMTXC
- Back end tools
 - GVMTAS
 - GVMTCC
 - GVMTLINK

GVMTIC and GVMTCC form the front-end and are used to translate the C-based interpreter description and standard C files, respectively, into GSC format.

GVMTAS converts these GSC files to object files and GVMTLINK links them together. GVMTCC takes the GSC file produced by GVMTIC and produces a compiler from it.

2.1.1 Interpreter description file

An interpreter description file consists of bytecode definitions plus the local variables shared by all bytecodes.

Each bytecode can be defined by its stack effect plus a snippet of C code or as a sequence of other bytecodes and predefined GVMT abstract stack machine codes.

Usually there is one primary interpreter description file that defines the bytecode format. Secondary interpreters (see Sect. 2.2.1) will be guaranteed to use the same bytecode format. However it is possible to have several independent primary interpreters. For example a virtual machine might use one bytecode format for its interpreter and compiler and an entirely different format for its regular-expression engine. Each primary interpreter may have secondary interpreters.

Interpreters are fully re-entrant and thread-safe. The interpreter description file defines not only the behaviour of the interpreter, but the behaviour of the GVMT-generated compiler as well.

Format

The interpreter description file consists of a locals block and any number of bytecode declarations in any order.

Interpreter-local variables

Interpreter-locals variables are declared in a locals section:

```
locals {  
    type name;  
  
:  
  
}
```

Interpreter-local variables are visible within all bytecode definitions, and can be accessed indirectly via the interpreter frame on the control stack. Interpreter-local variables (and the interpreter frame) have the same lifetime as the invocation of the interpreter.

Bytecode definitions

Bytecodes can be defined either in C or directly in GSC. Both types of definition can be used in the same interpreter.

C-bytecode definitions are in the form

```
name ('[' qualifier* ']')? '(' stack_comment ')' [ '=' <int> ] '{'
    Arbitrary C code
'}'
```

Legal qualifiers are:

private This bytecode is not visible at the interpreter level.

protected This bytecode is acceptable to the interpreter, but not to any verification code. It is only to be used internally as an optimisation. The meaning of 'protected' is largely conventional. GVMT will ignore it.

nocomp This bytecode will not appear in bytecode passed to the compiler. If this bytecode is seen by the compiler it will abort. Useful for reducing the size of the compiler.

componly This bytecode will only be passed to the compiler. It will cause the interpreter to abort. Useful for reducing the size of the interpreter.

The `stack_comment` is of the form:

```
type id (',' type, id)* -- type id (',' type, id)*
```

Where `type` is any legal C type and `id` is any legal C identifier, or a legal C identifier prefixed with `#`. Variables prefixed with `#` are fetched from the instruction stream, not popped from the stack. When referred to in the C code, the `#` prefix is dropped.

The integer at the end of the header line is the actual numerical value of this bytecode. If not supplied, GVMT will allocate bytecode numbers starting from 1.

Compound instructions are in the form:

```
name ('[' qualifier* ']')? '(' stack comment ')' [ '=' <int> ] ':'
    instruction*
';'
```

Where `instruction` is a legal GSC or user-defined instruction.

To allow user defined instructions to do anything useful, a large number of builtin instructions are provided. All are private, that is they must be used inside a compound instruction to be visible at the interpreter level.

Instruction stream manipulation operators are provided: `#0` Removes a byte from the instruction stream and pushes it to the data stack. `#N` where `N` is a one-byte integer pushes `N` into the front of the instruction stream. `#[N]`

Pushes a copy of the Nth item, starting at zero, in the instruction stream into the front of the instruction stream. **#+** or **#-** Adds or subtracts, respectively the first two values from the instruction stream and pushes the result back to the front of the instruction stream.

Appendix B contains a full list of all abstract machine instructions

There are four bytecode names which are treated specially by the GVMT, these are:

- `--preamble`
- `--postamble`
- `--enter`
- `--exit`
- `--default`

`--preamble` is executed when the interpreter is called, before any bytecodes are executed.

`--postamble` is executed when the interpreter reaches the end of the bytecodes. Interpreters with a `--postamble` instruction expect a second parameter, the end of the bytecode sequence. This is useful for ‘interpreters’ which analyse code rather than executing it.

`--enter` is executed before the main bytecode definition for every bytecode executed.

`--exit` is executed after the main bytecode definition for every bytecode executed.

`--default` is used in secondary interpreter definitions. When no explicit definition exists for a bytecode, the `--default` is used instead. `--default` must not modify the instruction stream.

Examples

TO DO – Put a couple of examples here. 1 C, 1 vmc.

2.1.2 The interpreter generator `gvmtic`

GVMTIC comiles an interpreter description file, as described above, into a GSC file.

GVMTIC takes the following options:

-D symbol Define symbol in preprocessor

- I file** Add file to list of #include file for lcc sub-process, can be used multiple times
- a** Warn about non-ANSI C.
- b bytecode-header-file** Specify bytecode header file
- h** Print this help and exit
- n name** Name of generated interpreter
- o outfile-name** Specify output file name
- v** Echo sub-processes invoked
- z** Do not put gc-safe-points on backward edges

2.1.3 The C compiler gvmc

GVMTC takes a standard C89 file as its input and outputs a GSC file.

GVMTC takes the following options:

- D symbol** Define symbol in preprocessor
- I file** Add file to list of #include file for lcc sub-process, can be used multiple times
- L** Directory to find lcc executables
- a** Warn about non-ANSI C.
- h** Print this help and exit
- o outfile-name** Specify output file name
- v** Echo sub-processes invoked
- x** Output dot file of IR (For debugging compiler)
- z** Do not put gc-safe-points on backward edges

2.1.4 The GSC assembler `gvmtas`

GVMTAS takes a GSC file and assembles it to a GVM T object file (.gso is the expected file extension)

GVMTAS takes the following options:

- H dir** Look for gvmt internal headers in dir.
- O** Optimise. Levels 0 to 3
- T** Use token-threading dispatch
- g** Debug
- h** Print this help and exit
- l** Output GSO suitable for library code, no bytecode, root or heap sections allowed
- m memory_manager** Memory manager (garbage collector) used
- o outfile-name** Specify output file name

2.1.5 The compiler generator `gvmtcc`

GVMTCC Takes a GSC file produced by GVMTIC and produces a compiler. The generated compiler is in C++ and relies on LLVM (<http://llvm.org/>). It will need to be compiled with the system C++ compiler.

It may be possible to generate compilers using other code-generator back-ends in the future, but there is currently no plan to do so.

GVMTCC takes the following options:

- h** Print this help and exit
- m memory_manager** Memory manager (garbage collector) used
- o outfile-name** Specify output file name

2.1.6 The linker `gvmtlink`

GVMTLINK takes any number of GVM T object (.gso) files and links them to form a single native object file. This object file can be linked with the GC object file and compiler object file (if required) using the system linker to form an executable.

GVMTLINK takes the following options:

- h** Print this help and exit
- l** Output a library (.dll or .so)
- n** No-heap, error if heap or roots section exist
- o outfile-name** Specify output file name
- v** verbose

2.1.7 lcc-gvmt

Both GVMTC and GVMTC use a modified version of lcc as their C-to-GSC compiler. This compiler uses the standard lcc front-end and a custom back-end. The back-end operates in four passes. The first pass labels the IR forests with the C type for that expression, as much as can be derived without the explicit casts present in the source code. The second pass is a bottom pass which labels the trees with the set of possible GSC types it may take. The third pass is a top-down pass which attempts to find the exact GSC type of each node. The fourth pass emits the GSC.

2.2 Other tools

2.2.1 The bytecode-processor generator gvmtxc

GVMTXC Takes a (partial) interpreter description file and a master GSC file (produced by GVMTC) and produces a GSC file representing an interpreter. This interpreter is guaranteed to accept the same bytecode format as the master interpreter. For a partial interpreter description file, the generate interpreter skips over the omitted bytecodes.

GVMTXC takes the following options:

- D symbol** Define symbol in preprocessor
- I file** Add file to list of #include file for lcc sub-process, can be used multiple times
- a** Warn about non-ANSI C.
- b bytecode-header-file** Specify bytecode header file
- e** Explicit: All bytecodes must be explicitly defined
- h** Print this help and exit

- n name** Name of generated interpreter
- o outfile-name** Specify output file name
- v** Echo sub-processes invoked
- z** Do not put gc-safe-points on backward edges

2.2.2 The object layout tool

The object layout tool is an optional tool to help manage heap object layout. It can create C structs, shape vectors and marshalling to GSC files for objects all from a single specification. This tool is designed as an extensible Python script rather than a stand-alone tool, although it can be used as such. When used as a standalone tool it has takes the following options:

- h** Print this help and exit
- m** Output marshalling code for all kinds
- o outfile-name** Specify output file name
- p** Define a pattern to use for typedefs
- s** Output header with stuct definitions for C code
- t** Output header with typedefs for C code

Chapter 3

GVMT Abstract Machine

The GVMT defines an abstract stack-based machine (The GVMT Abstract Machine) that has well defined semantics, but is abstract in the sense that it cannot execute any programs. It is designed to both assist in implementing stack-based virtual machine and allow translation to efficient machine-code.

NOTE: the term ‘bytecode’ is used below to refer to a virtual-machine instruction and the term ‘instruction’ is used to refer to an abstract-machine instruction. Bytecodes (virtual-machine instructions) are defined by sequences of instructions (abstract-machine instructions).

3.1 Components

The GVMT Abstract Machine consists of main (shared) memory and zero or more threads of execution.

Each thread consists of four stacks, a transfer register, and a list of thread-local variables. The stacks are:

1. Data stack
2. Control stack
3. State stack
4. Native parameter stack

The main memory is divided into two parts: a user-managed region and a garbage-collected heap.

Upon initialisation, the GVMT Abstract Machine has zero threads; no code is executing. Threads can be added and started by using the `gvmt_start_thread` function.

3.2 Threads

Each thread is independent, the number of concurrently executing threads is limited by the underlying operating system and hardware. Threads share the heap, but have their own stacks and thread-local variables. GVMT threads are implemented as operating system threads.

3.2.1 Execution Model

Execution of a thread starts by creating a new set of stacks for that thread. Initially all stacks are empty. The arguments passed to the `gvmt_start_thread` function are pushed to the data stack, followed by the address of the start function. The `CALL_X` is then executed, where `X` depends on the type specified in the `gvmt_start_thread` function.

Execution of a function proceeds as follows: A frame containing all the temporary variables necessary for the function is pushed to the control stack. This frame becomes current frame for accessing all temporary variables. The layout and management of this frame is not defined. Temporary variables in previously pushed stacks cannot be accessed. The first instruction in the function is then executed, proceeding to the next instruction and so on. The exception to this are flow control instruction such as `HOP` or `BRANCH` which may jump to a designated successor instruction.

3.2.2 Functions

A function in GVMT is defined as a linear sequence of instructions. Execution of a function starts by pushing a frame to the control stack. This frame will have sufficient space to store all the temporary variables required by the function. Execution then starts with the first instruction in the sequence and proceeds to the next instruction and so on. The exception to this are flow control instruction such as `HOP` or `BRANCH` which may jump to a designated successor instruction.

3.2.3 Interpreters

The interpreter acts externally like a normal function; it can be called like any other. Its behaviour is substantially different from that of normal functions. The interpreter commences execution, like a normal function, by pushing a frame to the control stack. This frame will have sufficient space to store all the temporary variables of the bytecodes of the interpreter plus any interpreter-

scope variables. The interpreter definition specifies the names and types of these variables.

Each activation of an interpreter contains a virtual-machine-level instruction pointer which tells it which bytecode to execute. The start-point of the interpreter is passed in as a parameter and popped from the data-stack on entry.

Execution of bytecodes (bytecodes) proceeds in a linear fashion, unless a `JUMP` or `FAR_JUMP` abstract-machine instruction is encountered.

The execution of individual bytecodes proceeds as follows: The abstract-machine instructions that make up that bytecode are executed in the same way as for a normal function. Should the end of the bytecode be reached (as it will be for most bytecodes) then the instruction pointer is updated to point at the next instruction and that instruction is then executed.

3.2.4 Compiled Code

The output of the compiler is a function and can be called like any other. Its behaviour, in GVMT abstract-machine terms¹, is exactly the same as if the interpreter were called with the same input (bytecodes) as passed to the compiler when it generated the compiled function, provided the bytecodes are not modified.

3.2.5 Abstract Machine Instructions

GVMT abstract machine instructions. can be grouped into three categories:

Data Instructions that take values on the data stack and place results there, such as addition.

Flow control Instructions that alter the flow of control, both between instructions and between bytecodes.

Stack manipulation Instructions that explicitly manipulate or describe the status of the data, control and state stacks.

Appendix B contains the full abstract machine instruction set.

¹Its real-world behaviour may differ; it should be faster, and it may implement the top of the data-stack differently.

3.3 The Stacks

3.3.1 The Data Stack

The data stack is where all arithmetic operations takes place. The instruction set is designed so that the data stack can be treated as an in-memory array, yet allow back-ends some freedom to store the top few items in registers. See the `STACK` instruction for more details on accessing the stack as an array. The data stack grows downward. The stack supports six distinct types: `I4`, `I8`, `F4`, `F8`, `p`, `R`. The `U4` and `U8` types are also supported, but these are synonymous with the `I4` and `I8` types. Barring `I4/U4` and `I8/U8` pairs, pushing one type to the stack and then popping a different type is an error.

3.3.2 GC Safe points

The GC safe instruction declares that the executing thread can be interrupted for garbage collection. In order for this to be safe, the following restrictions must be observed:

1. No non-reference values are on the data stack.
2. All heap objects reachable from this thread are in a scannable state.²

The front-end tools `GVMTC` and `GVMTC` automatically ensure that condition 1 is true for all C code. The toolkit ensures that all virtual stack items are reachable by the garbage-collector. Condition 2 can be ensured simply by initialising all objects immediately after allocation.

3.3.3 The Control Stack

The control stack is an opaque structure used to handle procedure calls and returns, and storing of temporary variables. Interpreter frames are also allocated on the control stack. Memory can be allocated on the call-stack, but subsequent allocation are not guaranteed to be contiguous. The control stack will usually map directly onto the native C stack.

Temporary variables are accessed by the `TLOAD_X(n)` and `TSTORE_X(n)` instructions. Temporaries have no address. Temporary variables have the same types as data stack elements, with the same restrictions on mixing types.

²Usually this means that the `header(class)` of any newly created object will have been initialised.

3.3.4 The Native Parameter Stack

Parameters are also passed to native code using the native parameter stack. Values are pushed to the native parameter stack by the `NARG_X` instructions and popped by the `N_CALL` instruction. All functions and bytecodes must have pop exactly as many values as they push to the native value stack.

3.3.5 The State Stack

Each state object consists of the current underlying machine state (in order to resume execution from the same point), the current data-stack depth, the current control stack depth, and the current interpreter instruction pointer (if in the interpreter).

The state stack depth does not need to be recorded as it must be unchanged.

3.3.6 Raise and Transfer

The `RAISE` and `TRANSFER` instructions are primarily designed to implement exception handling, but can be used to implement coroutines and continuations.

3.3.7 Thread-local variables

Each thread has thread-locals variables, the number and type of these variables are determined by the developer.

3.4 The heap

The GVMT Abstract Machine heap is a garbage collected heap. The toolkit user needs to provide a function `gvm_t_shape` which supplies the garbage collector with information for scanning of objects. See section 5.1.1. Static roots of the heap can either be defined when the VM is built or added at runtime, see section 5.2.2.

3.4.1 Shape

During tracing the garbage collector needs to know both the extent of an object and which fields are references to other objects, and which are not. The developer communicates this information to the toolkit via a ‘shape’ vector

(array). This shape is a zero-terminated vector of integers. Each integer represents either N successive references or N successive words of non-references. References are represented by positive numbers, non-references by negative numbers (zero is the terminator). For example the shape [2, -2, 1, 0] represents an object of total extent 5 words, the first 2 and last of which are references. The following C declaration (taken from the example Scheme VM):

```
GVMT_OBJECT(cons) {
    struct type *type;    // Opaque (non-GC) pointer
    GVMT_Object car;     // Reference to heap object
    GVMT_Object cdr;     // Reference to heap object
};
```

defines `cons` objects. These `cons` objects would have a shape of [-1, 2, 0].

3.4.2 Garbage collection

No garbage collection algorithm is specified for the GVMT. However, since the GVMT can support both read and write barriers as well as the declaration of GC safe points, a wide range of collectors should be feasible. Currently both generational and non-generational collectors are available, although only ‘stop-the-world’ collectors have been implemented. The GVMT garbage collectors also support tagging, which allows non-references to be stored in reference slots by setting one or more ‘tag’ bits. Read and write barriers are implemented via the `RLOAD_R` and `RSTORE_R` instructions respectively.

3.4.3 Exception handling

The GVMT exception handling model uses an explicit stack, rather than tables. This means that using exceptions has a runtime cost even when the exceptions are not used as a means of flow control transfer, but that exception handling itself is much faster than for table-based approaches. The GVMT exception handling mechanism is similar to the C `setjump-longjump` mechanism.

The four instructions involved are (See section 5.2.3 for the matching intrinsics):

PUSH_CURRENT_STATE Pushes the current execution state onto the state stack. This state-object holds the following information: The current execution point (immediately after the `PUSH_CURRENT_STATE`

instruction), the stack depth, and the current control stack frame. Finally it pushes NULL onto the data stack.

POP_STATE Pops and discards the state-object from the state stack.

RAISE Pops the value from the TOS and stores it. The state-object on top of the state stack is examined and the thread execution state (data and control stack pointers) restored to the values held in the handler. The stored value is pushed onto the data stack. Finally, execution resumes from the location stored in the state-object.

TRANSFER Pops the value from the TOS and stores it. The state-object on top of the state stack is examined and the thread execution state, except the data-stack (control stack pointer only) restored to the value held in the handler. The stored value is pushed onto the data stack. Finally, execution resumes from the location stored in the state-object.

It is an error (resulting in a probable crash) to leave a state object on the state stack after the function in which it was pushed has returned. There is no concept of handling only some sorts of exceptions in the GVMT. All exceptions are caught; exceptions that should be handled elsewhere must be explicitly reraised.

Exceptions work as expected in interpreted (and compiled) code, as the interpreter instruction pointer is stored along with the machine instruction pointer.

3.5 Concurrency

The GVMT is concurrency friendly, all generated code and library code is thread-safe. The GVMT does not provide a thread library, so the developer will have to use the underlying operating system library. Concurrency is at the thread level, the GVMT is unaware of processes.

In order that the toolkit can operate correctly, it must be informed of the creation and destruction of threads. The toolkit provides functions for this purpose, see Section 5.2.4 for more details. The developer should also be aware that heap allocation may block, waiting for a collection, so locks should not be held across an allocation.

The GVMT also provides fast locks for synchronisation in the case where contention is expected to be rare. This functionality is provided by the `LOCK`, `UNLOCK`, `LOCK_INTERNAL` and `UNLOCK_INTERNAL` abstract machine instructions.

The intrinsic functions `gvmt_lock(gvmt_lock_t *lock)`, `gvmt_unlock(gvmt_lock_t *lock)`, `gvmt_unlock_internal(GVMT_Object object, size_t offset)` and `gvmt_unlock_internal(GVMT_Object object, size_t offset, gvmt_lock_t *lock)` are translated to the `LOCK`, `UNLOCK`, `LOCK_INTERNAL` and `UNLOCK_INTERNAL` instructions, respectively.

Warning: GVMT locks are not automatically unlocked by the `RAISE` instruction. The developer must ensure that either a `RAISE` cannot occur between a `LOCK` and `UNLOCK`, or that the `LOCK-UNLOCK` pair are `PROTECTED`. A safe lock/unlock sequence is (in C):

```
gvmt_lock_t lock = GVMT_LOCK_INITIALIZER;
GVMT_Object ex;
gvmt_lock(&lock)
GVMT_BEGIN_TRY(ex)

:

gvmt_unlock(&lock)
GVMT_CATCH
gvmt_unlock(&lock)
gvmt_raise(ex)
GVMT_END_TRY
```

3.5.1 Memory model

As all threads share the same heap, they may access heap objects concurrently. The following guarantees are made about the state of the heap and roots seen from one thread when modified by another thread:

All writes to pointers(P), references(R) and integers(I) not larger than a pointer are atomic.

In between synchronisation points, the delay between a write being made by one thread and a write being seen by another thread is indefinite.

Chapter 4

Building a VM using the Toolkit

4.1 Before you start

Before starting a virtual machine, you will need to generate some bytecode programs to execute. If you are targetting a pre-existing VM format, this is straightforward. If you are creating a new language you will need a parser. GVMTC can produce a C header file containing definitions for all the opcodes: the C macro for the opcode for instruction ‘xxx’ of interpreter ‘int’ is `GVMT_OPCODE(int, xxx)`.

4.2 Defining the components

For GVMT to build a VM, you will need to define:

- An interpreter. See section 2.1.2
- Any other support code that your interpreter calls. See section 2.1.3

You will also need to define a small number of functions required by the GVMT. The header file for these can be found in `/include/user.h` and sample implementations can be found in `/example/native.c` and `/example/support.c`. It may also be useful to specify:

- The object layout for heap objects. See section 2.2.2

Once these are defined, the VM is built as follows, see figure 1.1: The `.gsc` files for the interpreter and other support code is created by running GVMTC and GVMTC respectively. These `.gsc` files are converted to `.gso`

with GVMTAS. All the `.gso` files are then linked together using GVMTLINK to form a single object file, which can be linked with any native object files to form an executable using the standard linker.

4.3 Debugging

Although the toolkit maintains symbolic information from the source to the executable, some variables may get renamed. The signatures of functions may appear different in the debugging, although their names should be unaltered. The execution stack can be accessed via the stack pointer, the name of which is `gvmt_sp`.

4.4 Adding a compiler

It is probably best to have a fully working interpreter before adding the compiler. To generate the compiler simply run GVMTCC on the output file from GVMTIC. The resulting C++ file should be compiled and linked using the system C++ compiler. See section 2.1.5 for more details.

Chapter 5

User interface

The API of the GVMT, consists of two sets of functions and macros. The first is the set of functions and macros that the VM developer must implement. These are used by the GVMT in order to correctly implement the garbage collector, marshalling and debugging support. The second set of functions (soem of which may be implemented as macros) is provided by the GVMT to allow the VM developer to access features of the GVMT abstract machine from C.

For examples of API usage, see the Scheme interpreter in the `/example` subdirectory which provides the necessary functions, and uses the API extensively.

5.1 User provided code

To create a complete VM, the various components need to interact and to do so correctly. In order that the GVMT can correctly build the VM, the VM developer is required to implement a number of functions.

The file `/include/gvmt/user.h` lists all the functions which the developer needs to supply, with the exception of the `gvmt_shape` function which must be compiled natively, not by GVMTC.

5.1.1 Garbage collection

In order for the toolkit to provide precise garbage collection, the layout of all heap objects must be known. The developer must provide three functions and one integer value to allow GVMT to perform *precise* garbage collection.

int GVMT_MAX_SHAPE_SIZE The maximum number of entries required to represent any shape (including terminating zero).

long* gvmnt_shape(GVMT_Object object, int* buf) This function returns a vector of integers, representing the shape of the object, as defined in section 3.4.1. The int array buf can be used as temporary storage for the shape vector if required. The buffer, buf is guaranteed to include at least GVMT_MAX_SHAPE_SIZE entries.

size_t gvmnt_length(GVMT_Object obj) This function returns the length of the object in bytes.

void user_finalize_object(GVMT_Object o) This function may be called by the garbage collector for any object that is declared as requiring finalization and is unreachable. Exceptions raised in this function, or any callee, terminate finalisation of the object, but do not propagate. The function is called by the finalizer thread and will be called asynchronously. This function may be called on any or all object declared as finalizable, in any order.

5.1.2 High-Performance Garbage Collection Interface

The GVMT provides the means to integrate the GC more closely with the VM, providing better performance. It is strongly recommended that, this interface is only used after the VM has been well tested with the standard interface, since the gvmnt_shape function will still need to be implemented. To use the high-performance interface, the GVMT will need to be rebuilt with the macro GVMT_CUSTOM_GC_INTERFACE defined and the following functions implemented.

template <class Collection> inline Address scan_object(Address addr)

```
This function should execute the sequence
if (Collection::wants(*item)) *item = Collection::apply(*item);
for each reference in the object pointed to by address. item should be
a pointer to the field (of type GVMT_Object*). This function should
return the address of the end of the object,
addr.plus_bytes(gvmnt_user_length(addr.as_object())).
```

inline size_t gvmnt_object_length(GVMT_Object) Inlined version of gvmnt_length(GVMT_Object obj).

See include/gvmt/internal/gc_custom.hpp for more details.

5.1.3 The Marshalling Interface

In order to support marshalling the user must provide a function gvmnt_write_func get_marshall_for_object(GVMT_Object object) to

give a write function for each object. The toolkit can create a write function for any object declaration, but it is unable to determine which function should be applied to which object at runtime.

To assist marshalling to GSC code, the user should also provide the `void gvmt_readable_name(GVMT_Object object, char *buffer)` function to provide meaningful names for objects. This function should store an ASCII string into the provided buffer. The buffer is guaranteed to be of at least `GVMT_MAX_OBJECT_NAME_LENGTH` bytes in length.

`int GVMT_MAX_OBJECT_NAME_LENGTH` should also be defined.

If marshalling support is not required, these functions still need to be implemented, but will not be called so can just be empty.

5.1.4 Debugging The Interpreter

The GVMT can insert arbitrary code at the start and end of each bytecode, by defining one or both of the `__enter` and `__exit` pseudo bytecode definitions in the interpreter definition.

Since GVMT knows nothing about the semantics of the VM, the developer must provide the actual code. A very simple tracing function is provided in the example.

5.2 GVMT provided functions

GVMT provides a wide range of functions to interact with the underlying abstract machine, as well as a number of intrinsic functions for code written in C.

5.2.1 The data stack

The data stack can be examined and modified in C code, using intrinsics:

- `gvmt_stack_top()` returns a pointer to the top-of-stack.
- `gvmt_insert(size_t n)` inserts `n` NULLs onto the stack and returns a pointer to them.
- `GVMT_PUSH(x)` can be used to push individual values on to the stack.
- `gvmt_drop(size_t n)` discards `n` items from the top of the stack.

Care should be taken with these functions as they manipulate the data stack, which is used to evaluate expression. It is best to keep any C statement using these intrinsics as simple as possible.

5.2.2 The garbage collector

The following intrinsic allocates object (the abstract machine instruction is GC_MALLOC).

- `GVMT_Object gvmt_malloc(size_t s)` Allocates a new object. This object should be initialised immediately, and *must* be initialised before the next GC-SAFE point.

The garbage collector can automatically find all objects from the stacks and all global variables. Sometimes it is necessary to have some control over the garbage collector. GVMT provides the following functions to interact with the garbage collector.

- `gvmt_gc_finalizable(GVMT_Object obj)` Declares that obj will need finalization.
- `void* gvmt_gc_weak_reference(void)` Returns a weak reference to obj.
- `GVMT_Object gvmt_gc_read_weak_reference(void* w)` Reads a weak reference, the reference returned will either be the last value written to this weak-reference or NULL.
- `gvmt_gc_write_weak_reference(void* w, GVMT_Object o)` Writes to a weak reference
- `void gvmt_gc_free_weak_reference(void* w)` Frees a weak reference if no longer required.
- `void* gvmt_gc_add_root(void)` Returns a new root to the heap, initialised to obj.
- `GVMT_Object gvmt_gc_read_root(void* root)` Returns the value referred to by the root.
- `void gvmt_gc_write_root(void* root, GVMT_Object obj)` Writes a new object to a root.
- `void gvmt_gc_free_root(void* root)` Deletes this root, the object referred to may now be garbage collected.
- `void* gvmt_pin(GVMT_Object obj)` Pins the object referred to by obj. The garbage collector will not move it. Be aware that collection and pinning are independent. The garbage collector may collect

pinned objects. To pass an object to native code, or to use an internal pointer will require both pinning the object *and* retaining a reference to it. Once pinned, objects remained pinned until they are collected.

5.2.3 Exception handling

Three intrinsics are provided:

- `GVMT_Object gvmt_protect(void)` Intrinsic for the PROTECT instruction. Returns NULL when initially called. When `gvmt_raise(ex)` or `gvmt_transfer(ex)` is called then `gvmt_protect()` returns `ex`.
- `void gvmt_unprotect(void)` Intrinsic for the UNPROTECT instruction. Removes previous `gvmt_protect()`.
- `void gvmt_raise(GVMT_Object ex)` Intrinsic for the RAISE instruction. Restores the machine state to that of the of the previous call to `gvmt_protect()`, then pushes `ex`.
- `void gvmt_transfer(GVMT_Object ex)` Intrinsic for the TRANSFER instruction. Restores the machine state to that of the of the previous call to `gvmt_protect()`, but does not modify the data stack.

Rather than use `GVMT_Object gvmt_protect()` and `void gvmt_unprotect()` directly developers can use the macros: `GVMT_TRY`, `GVMT_CATCH` and `GVMT_END_TRY` as follows:

```
GVMT_TRY(exception_variable)
    // code which may raise exception
    // exception_variable is NULL
GVMT_CATCH
    // code to execute if an exception is raised
    // exception_variable is set to the argument of gvmt_raise() or gvmt_call().
GVMT_END_TRY
```

5.2.4 Threading

Since the GVMT relies on the operating system to provide threading support, it must be informed when a new thread is created and used. To start running GVMT code in a new thread, call:

```
int gvmt_enter(uintptr_t stack_space, gvmt_func_ptr func, int pcount, ...);
```

Where:

- `stack_space` is the amount of items required on the data stack,
- `gvmt_func_ptr` is the GSC function to execute, and
- `pcount` is the number of parameters following.

If reentering GVMT from native code that was running in an already initialised thread, then use

```
int gvmt_reenter(gvmt_func_ptr func, int pcount, ...);
```

to avoid reinitialising the running thread.

When a thread has finished call `gvmt_finished(void)`.

Appendix A

Installing the GVMT

Currently the GVMT has only been tested on the x86-linux platform, but it should work on any posix-compliant x86 platform. GVMT can be installed in the following steps:

1. Download GVMT from <http://code.google.com/p/gvmt/>
2. Unpack into a clean directory, `$(GVMT)`
3. Install llvm (version2.5) from <http://llvm.org/releases/download.html#2.5>
4. Type `make`

The build process will fetch lcc using wget. If you do not have wget installed or wget fails then you will have to download lcc from <http://sites.google.com/site/lccretargetablecompiler/downloads/4.2.tar.gz> and save it as `$(GVMT)/lcc.tar.gz`

5. Type `sudo make install` (You will need write privileges to `/usr/local` in order to install the tools)

If it doesn't work, then email me: marks@dcs.gla.ac.uk

Appendix B

The Abstract Machine Instruction Set

Introduction

This appendix lists all 367 instructions of the GVMT abstract machine instruction set. The instruction set is not as large as it first appears. Many of these are multiple versions of the form `OP_X` where `X` can be any or all of the twelve different types. These types are `I1`, `I2`, `I4`, `I8`, `U1`, `U2`, `U4`, `U8`, `F4`, `F8`, `P`, `R`.

`IX`, `UX` and `FX` refer to a signed integer, unsigned integer and floating point real of size (in bytes) `X`. `P` is a pointer and `R` is a reference. `P` pointers cannot point into the GC heap. `R` references are pointers that can *only* point into the GC heap.

For all instructions where the type is a pointer sized integer, `I4` and `U4` for 32-bit machines or `I8` and `U8` for 64-bit machines, there is an alias for each instruction of the form `OP_IPTR` or `OP_UPTR`. E.g. on a 32-bit machine the instruction `ADD_I4` has an alias `ADD_IPTR`.

`TOS` is an abbreviation for top-of-stack and `NOS` is an abbreviation for next-on-stack.

Each instruction is listed below in the form:

Name (inputs \Rightarrow outputs)

Instruction stream effect

Description of the instruction

#+ (— ⇒ —)

2 operand bytes. Pushes 1 byte to instruction stream.
Fetches the first two values in the instruction stream, adds them and pushes the result back to the stream.

#- (— ⇒ —)

2 operand bytes. Pushes 1 byte to instruction stream.
Fetches the first two values in the instruction stream, subtracts them and pushes the result back to the stream.

#n (— ⇒ —)

No operand bytes. Pushes 1 byte to instruction stream.
Push 1 byte value to the front of the instruction stream.

#2@ (— ⇒ **operand**)

2 operand bytes.
Fetches the next 2 bytes from the instruction stream. Combine into an integer, first byte is most significant. Push onto the data stack.

#4@ (— ⇒ **operand**)

4 operand bytes.
Fetches the next 4 bytes from the instruction stream. Combine into an integer, first byte is most significant. Push onto the data stack.

#@ (— ⇒ **operand**)

1 operand byte.
Fetches the next byte from the instruction stream. Push onto the data stack.

#[n] (— ⇒ —)

No operand bytes. Pushes 1 byte to instruction stream.
Only valid in an interpreter definition. Peeks into the instruction stream and pushes the n^{th} byte in the stream to the front of the instruction stream.

ADDR(name) (— ⇒ **address**)

Pushes the address of the global variable name to the stack (as a pointer).

ADD_F4 (op1, op2 ⇒ result)

Binary operation: 32 bit floating point add.
result := op1 + op2.

ADD_F8 (op1, op2 ⇒ result)

Binary operation: 64 bit floating point add.
result := op1 + op2.

ADD_I4 (op1, op2 ⇒ result)

Binary operation: 32 bit signed integer add.
 result := op1 + op2.

ADD_I8 (op1, op2 ⇒ result)

Binary operation: 64 bit signed integer add.
 result := op1 + op2.

ADD_I4 (op1, op2 ⇒ result)

Binary operation: 32 bit signed integer add.
 result := op1 + op2.

ADD_P (op1, op2 ⇒ result)

Binary operation: pointer add.
 result := op1 + op2.

ADD_U4 (op1, op2 ⇒ result)

Binary operation: 32 bit unsigned integer add.
 result := op1 + op2.

ADD_U8 (op1, op2 ⇒ result)

Binary operation: 64 bit unsigned integer add.
 result := op1 + op2.

ADD_U4 (op1, op2 ⇒ result)

Binary operation: 32 bit unsigned integer add.
 result := op1 + op2.

ALLOCA_F4 (n ⇒ ptr)

Allocates space for n 32 bit floating points in the current control stack frame, leaving pointer to allocated space in TOS. All memory allocated after a PUSH_CURRENT_STATE is invalidated immediately by a RAISE, but not necessarily immediately reclaimed. All memory allocated is invalidated and reclaimed by a RETURN instruction.

ALLOCA_F8 (n ⇒ ptr)

Allocates space for n 64 bit floating points in the current control stack frame, leaving pointer to allocated space in TOS. All memory allocated after a PUSH_CURRENT_STATE is invalidated immediately by a RAISE, but not necessarily immediately reclaimed. All memory allocated is invalidated and reclaimed by a RETURN instruction.

ALLOCA_I1 (n ⇒ ptr)

Allocates space for n 8 bit signed integers in the current control stack frame, leaving pointer to allocated space in TOS. All memory allocated after a PUSH_CURRENT_STATE is invalidated immediately by a RAISE, but not necessarily im-

mediately reclaimed. All memory allocated is invalidated and reclaimed by a RETURN instruction.

ALLOCA_I2 (n ⇒ ptr)

Allocates space for n 16 bit signed integers in the current control stack frame, leaving pointer to allocated space in TOS. All memory allocated after a PUSH_CURRENT_STATE is invalidated immediately by a RAISE, but not necessarily immediately reclaimed. All memory allocated is invalidated and reclaimed by a RETURN instruction.

ALLOCA_I4 (n ⇒ ptr)

Allocates space for n 32 bit signed integers in the current control stack frame, leaving pointer to allocated space in TOS. All memory allocated after a PUSH_CURRENT_STATE is invalidated immediately by a RAISE, but not necessarily immediately reclaimed. All memory allocated is invalidated and reclaimed by a RETURN instruction.

ALLOCA_I8 (n ⇒ ptr)

Allocates space for n 64 bit signed integers in the current control stack frame, leaving pointer to allocated space in TOS. All memory allocated after a PUSH_CURRENT_STATE is invalidated immediately by a RAISE, but not necessarily immediately reclaimed. All memory allocated is invalidated and reclaimed by a RETURN instruction.

ALLOCA_I4 (n ⇒ ptr)

Allocates space for n 32 bit signed integers in the current control stack frame, leaving pointer to allocated space in TOS. All memory allocated after a PUSH_CURRENT_STATE is invalidated immediately by a RAISE, but not necessarily immediately reclaimed. All memory allocated is invalidated and reclaimed by a RETURN instruction.

ALLOCA_P (n ⇒ ptr)

Allocates space for n pointers in the current control stack frame, leaving pointer to allocated space in TOS. All memory allocated after a PUSH_CURRENT_STATE is invalidated immediately by a RAISE, but not necessarily immediately reclaimed. All memory allocated is invalidated and reclaimed by a RETURN instruction.

ALLOCA_R (n ⇒ ptr)

Allocates space for n references in the current control stack frame, leaving pointer to allocated space in TOS. All memory allocated after a PUSH_CURRENT_STATE is invalidated immediately by a RAISE, but not necessarily immediately reclaimed. All memory allocated is invalidated and reclaimed by a RETURN instruction. ALLOCA_R cannot be used after the first HOP, BRANCH, TARGET, JUMP or FAR_JUMP instruction.

ALLOCA_U1 (n ⇒ ptr)

Allocates space for n 8 bit unsigned integers in the current control stack frame, leaving pointer to allocated space in TOS. All memory allocated after a PUSH_CURRENT_STATE is invalidated immediately by a RAISE, but not necessarily immediately reclaimed. All memory allocated is invalidated and reclaimed by a RETURN instruction.

ALLOCA_U2 (n ⇒ ptr)

Allocates space for n 16 bit unsigned integers in the current control stack frame, leaving pointer to allocated space in TOS. All memory allocated after a PUSH_CURRENT_STATE is invalidated immediately by a RAISE, but not necessarily immediately reclaimed. All memory allocated is invalidated and reclaimed by a RETURN instruction.

ALLOCA_U4 (n ⇒ ptr)

Allocates space for n 32 bit unsigned integers in the current control stack frame, leaving pointer to allocated space in TOS. All memory allocated after a PUSH_CURRENT_STATE is invalidated immediately by a RAISE, but not necessarily immediately reclaimed. All memory allocated is invalidated and reclaimed by a RETURN instruction.

ALLOCA_U8 (n ⇒ ptr)

Allocates space for n 64 bit unsigned integers in the current control stack frame, leaving pointer to allocated space in TOS. All memory allocated after a PUSH_CURRENT_STATE is invalidated immediately by a RAISE, but not necessarily immediately reclaimed. All memory allocated is invalidated and reclaimed by a RETURN instruction.

ALLOCA_U4 (n ⇒ ptr)

Allocates space for n 32 bit unsigned integers in the current control stack frame, leaving pointer to allocated space in TOS. All memory allocated after a PUSH_CURRENT_STATE is invalidated immediately by a RAISE, but not necessarily immediately reclaimed. All memory allocated is invalidated and reclaimed by a RETURN instruction.

AND_I4 (op1, op2 ⇒ result)

Binary operation: 32 bit signed integer bitwise and.
result := op1 & op2.

AND_I8 (op1, op2 ⇒ result)

Binary operation: 64 bit signed integer bitwise and.
result := op1 & op2.

AND_I4 (op1, op2 ⇒ result)

Binary operation: 32 bit signed integer bitwise and.
 result := op1 & op2.

AND_U4 (op1, op2 ⇒ result)

Binary operation: 32 bit unsigned integer bitwise and.
 result := op1 & op2.

AND_U8 (op1, op2 ⇒ result)

Binary operation: 64 bit unsigned integer bitwise and.
 result := op1 & op2.

AND_U4 (op1, op2 ⇒ result)

Binary operation: 32 bit unsigned integer bitwise and.
 result := op1 & op2.

BRANCH_F(n) (cond ⇒ —)

Branch if TOS is zero to Target(n). TOS must be an integer.

BRANCH_T(n) (cond ⇒ —)

Branch if TOS is non-zero to Target(n). TOS must be an integer.

CALL_F4 (— ⇒ value)

Calls the function whose address is TOS. TOS must be a pointer. Removal parameters from the stack is the callee's responsibility. The function called must return a 32 bit floating point.

CALL_F8 (— ⇒ value)

Calls the function whose address is TOS. TOS must be a pointer. Removal parameters from the stack is the callee's responsibility. The function called must return a 64 bit floating point.

CALL_I4 (— ⇒ value)

Calls the function whose address is TOS. TOS must be a pointer. Removal parameters from the stack is the callee's responsibility. The function called must return a 32 bit signed integer.

CALL_I8 (— ⇒ value)

Calls the function whose address is TOS. TOS must be a pointer. Removal parameters from the stack is the callee's responsibility. The function called must return a 64 bit signed integer.

CALL_I4 ($\text{---} \Rightarrow \text{value}$)

Calls the function whose address is TOS. TOS must be a pointer. Removal parameters from the stack is the callee's responsibility. The function called must return a 32 bit signed integer.

CALL_P ($\text{---} \Rightarrow \text{value}$)

Calls the function whose address is TOS. TOS must be a pointer. Removal parameters from the stack is the callee's responsibility. The function called must return a pointer.

CALL_R ($\text{---} \Rightarrow \text{value}$)

Calls the function whose address is TOS. TOS must be a pointer. Removal parameters from the stack is the callee's responsibility. The function called must return a reference.

CALL_U4 ($\text{---} \Rightarrow \text{value}$)

Calls the function whose address is TOS. TOS must be a pointer. Removal parameters from the stack is the callee's responsibility. The function called must return a 32 bit unsigned integer.

CALL_U8 ($\text{---} \Rightarrow \text{value}$)

Calls the function whose address is TOS. TOS must be a pointer. Removal parameters from the stack is the callee's

responsibility. The function called must return a 64 bit unsigned integer.

CALL_U4 ($\text{---} \Rightarrow \text{value}$)

Calls the function whose address is TOS. TOS must be a pointer. Removal parameters from the stack is the callee's responsibility. The function called must return a 32 bit unsigned integer.

CALL_V ($\text{---} \Rightarrow \text{value}$)

Calls the function whose address is TOS. TOS must be a pointer. Removal parameters from the stack is the callee's responsibility. The function called must return void.

D2F ($\text{val} \Rightarrow \text{result}$)

Converts 64 bit floating point to 32 bit floating point. This is a conversion, not a cast. It is the value that remains the same, not the bit-pattern.

D2I ($\text{val} \Rightarrow \text{result}$)

Converts 64 bit floating point to 32 bit signed integer. This is a conversion, not a cast. It is the value that remains the same, not the bit-pattern.

D2L (val ⇒ result)

Converts 64 bit floating point to 64 bit signed integer. This is a conversion, not a cast. It is the value that remains the same, not the bit-pattern.

DIV_F4 (op1, op2 ⇒ result)

Binary operation: 32 bit floating point divide.
result := op1 / op2. Rounds towards zero.

DIV_F8 (op1, op2 ⇒ result)

Binary operation: 64 bit floating point divide.
result := op1 / op2. Rounds towards zero.

DIV_I4 (op1, op2 ⇒ result)

Binary operation: 32 bit signed integer divide.
result := op1 / op2. Rounds towards zero.

DIV_I8 (op1, op2 ⇒ result)

Binary operation: 64 bit signed integer divide.
result := op1 / op2. Rounds towards zero.

DIV_I4 (op1, op2 ⇒ result)

Binary operation: 32 bit signed integer divide.
result := op1 / op2. Rounds towards zero.

DIV_U4 (op1, op2 ⇒ result)

Binary operation: 32 bit unsigned integer divide.
result := op1 / op2. Rounds towards zero.

DIV_U8 (op1, op2 ⇒ result)

Binary operation: 64 bit unsigned integer divide.
result := op1 / op2. Rounds towards zero.

DIV_U4 (op1, op2 ⇒ result)

Binary operation: 32 bit unsigned integer divide.
result := op1 / op2. Rounds towards zero.

DROP (top ⇒ —)

Drops the top value from the stack.

DROP_N (n ⇒ —)

1 operand byte.

Drops n values from the stack at offset fetched from stream. E.g. for offset=1 and n=2, TOS would be untouched, but NOS and 3OS would be discarded

EQ_F4 (op1, op2 ⇒ comp)

Comparison operation: 32 bit floating point equals.
comp := op1 = op2.

EQ_F8 (op1, op2 ⇒ comp)

Comparison operation: 64 bit floating point equals.
 comp := op1 = op2.

EQ_I4 (op1, op2 ⇒ comp)

Comparison operation: 32 bit signed integer equals.
 comp := op1 = op2.

EQ_I8 (op1, op2 ⇒ comp)

Comparison operation: 64 bit signed integer equals.
 comp := op1 = op2.

EQ_I4 (op1, op2 ⇒ comp)

Comparison operation: 32 bit signed integer equals.
 comp := op1 = op2.

EQ_P (op1, op2 ⇒ comp)

Comparison operation: pointer equals.
 comp := op1 = op2.

EQ_R (op1, op2 ⇒ comp)

Comparison operation: reference equals.
 comp := op1 = op2.

EQ_U4 (op1, op2 ⇒ comp)

Comparison operation: 32 bit unsigned integer equals.
 comp := op1 = op2.

EQ_U8 (op1, op2 ⇒ comp)

Comparison operation: 64 bit unsigned integer equals.
 comp := op1 = op2.

EQ_U4 (op1, op2 ⇒ comp)

Comparison operation: 32 bit unsigned integer equals.
 comp := op1 = op2.

EXT_I1 (value ⇒ extended)

Sign extends TOS from to a I1 to a pointer-sized integer.

EXT_I2 (value ⇒ extended)

Sign extends TOS from to a I2 to a pointer-sized integer.

EXT_I4 (value ⇒ extended)

Sign extends TOS from to a I4 to a pointer-sized integer.

EXT_I4 (value ⇒ extended)

Sign extends TOS from to a I4 to a pointer-sized integer.

EXT_U1 (value \Rightarrow extended)

Zero extends TOS from to a U1 to a pointer-sized integer.

EXT_U2 (value \Rightarrow extended)

Zero extends TOS from to a U2 to a pointer-sized integer.

EXT_U4 (value \Rightarrow extended)

Zero extends TOS from to a U4 to a pointer-sized integer.

EXT_U4 (value \Rightarrow extended)

Zero extends TOS from to a U4 to a pointer-sized integer.

F2D (val \Rightarrow result)

Converts 32 bit floating point to 64 bit floating point. This is a conversion, not a cast. It is the value that remains the same, not the bit-pattern.

F2I (val \Rightarrow result)

Converts 32 bit floating point to 32 bit signed integer. This is a conversion, not a cast. It is the value that remains the same, not the bit-pattern.

F2L (val \Rightarrow result)

Converts 32 bit floating point to 64 bit signed integer. This is a conversion, not a cast. It is the value that remains the same, not the bit-pattern.

FAR_JUMP (ip \Rightarrow —)

Continue interpretation, with the current abstract machine state, at the IP popped from the stack. FAR_JUMP is intended for unusual flow control in code processors and the like. Warning: This instruction is not supported in compiled code, in order to use jumps in compiled code use JUMP instead.

FIELD_IS_NOT_NULL (object, offset \Rightarrow value)

Tests whether an object field is null. Equivalent to RLOAD_X 0 EQ_X where X is a R, P or a pointer sized integer.

FIELD_IS_NULL (object, offset \Rightarrow value)

Tests whether an object field is null. Equivalent to RLOAD_X 0 EQ_X where X is a R, P or a pointer sized integer.

FILE(name) (— \Rightarrow —)

Declares the source file for this code. Informational only, like #FILE in C.

FULLY_INITIALIZED (*object* \Rightarrow —)

Declare TOS object to be fully-initialised. This allows optimisations to be made by the toolkit. Drops TOS as a side effect. TOS must be a reference, it is a (serious) error if TOS object has *any* uninitialised reference fields

GC_MALLOC (*size* \Rightarrow *ref*)

Allocates *size* bytes in the heap leaving reference to allocated space in TOS. GC pass may replace with a faster inline version. Defaults to GC_MALLOC_CALL.

GC_MALLOC_CALL (*size* \Rightarrow *ref*)

Allocates *size* bytes, via a call to the GC collector. Generally users should use GC_MALLOC and allow the toolkit to substitute appropriate inline code. Safe to use, but front-ends should use GC_MALLOC instead.

GC_MALLOC_FAST (*size* \Rightarrow *ref*)

Fast allocates *size* bytes, *ref* is 0 if cannot allocate fast. Generally users should use GC_MALLOC and allow the toolkit to substitute appropriate inline code. For internal toolkit use only.

GC_SAFE (— \Rightarrow —)

Declares this point to be a safe point for garbage collection to occur at. GC pass should replace with a custom version. Defaults to GC_SAFE_CALL.

GC_SAFE_CALL (— \Rightarrow —)

Calls GC to inform it that calling thread is safe for garbage collection. Generally users should use GC_SAFE and allow the toolkit to substitute appropriate inline code.

GE_F4 (*op1*, *op2* \Rightarrow *comp*)

Comparison operation: 32 bit floating point greater than or equals.

$\text{comp} := \text{op1} \geq \text{op2}$.

GE_F8 (*op1*, *op2* \Rightarrow *comp*)

Comparison operation: 64 bit floating point greater than or equals.

$\text{comp} := \text{op1} \geq \text{op2}$.

GE_I4 (*op1*, *op2* \Rightarrow *comp*)

Comparison operation: 32 bit signed integer greater than or equals.

$\text{comp} := \text{op1} \geq \text{op2}$.

GE_I8 (op1, op2 ⇒ comp)

Comparison operation: 64 bit signed integer greater than or equals.

comp := op1 ≥ op2.

GE_I4 (op1, op2 ⇒ comp)

Comparison operation: 32 bit signed integer greater than or equals.

comp := op1 ≥ op2.

GE_P (op1, op2 ⇒ comp)

Comparison operation: pointer greater than or equals.

comp := op1 ≥ op2.

GE_U4 (op1, op2 ⇒ comp)

Comparison operation: 32 bit unsigned integer greater than or equals.

comp := op1 ≥ op2.

GE_U8 (op1, op2 ⇒ comp)

Comparison operation: 64 bit unsigned integer greater than or equals.

comp := op1 ≥ op2.

GE_U4 (op1, op2 ⇒ comp)

Comparison operation: 32 bit unsigned integer greater than or equals.

comp := op1 ≥ op2.

GT_F4 (op1, op2 ⇒ comp)

Comparison operation: 32 bit floating point greater than.

comp := op1 > op2.

GT_F8 (op1, op2 ⇒ comp)

Comparison operation: 64 bit floating point greater than.

comp := op1 > op2.

GT_I4 (op1, op2 ⇒ comp)

Comparison operation: 32 bit signed integer greater than.

comp := op1 > op2.

GT_I8 (op1, op2 ⇒ comp)

Comparison operation: 64 bit signed integer greater than.

comp := op1 > op2.

GT_I4 (op1, op2 ⇒ comp)

Comparison operation: 32 bit signed integer greater than.

comp := op1 > op2.

GT_P (op1, op2 ⇒ comp)

Comparison operation: pointer greater than.
 comp := op1 > op2.

GT_U4 (op1, op2 ⇒ comp)

Comparison operation: 32 bit unsigned integer greater than.
 comp := op1 > op2.

GT_U8 (op1, op2 ⇒ comp)

Comparison operation: 64 bit unsigned integer greater than.
 comp := op1 > op2.

GT_U4 (op1, op2 ⇒ comp)

Comparison operation: 32 bit unsigned integer greater than.
 comp := op1 > op2.

HOP(n) (— ⇒ —)

Jump (unconditionally) to TARGET(n)

I2D (val ⇒ result)

Converts 32 bit signed integer to 64 bit floating point. This is a conversion, not a cast. It is the value that remains the same, not the bit-pattern.

I2F (val ⇒ result)

Converts 32 bit signed integer to 32 bit floating point. This is a conversion, not a cast. It is the value that remains the same, not the bit-pattern.

INSERT (n ⇒ address)

1 operand byte.

Pops count off the stack. Inserts n NULLs into the stack at offset fetched from the instruction stream. Ensures that all inserted values are flushed to memory. Pushes the address of first inserted slot to the stack.

INV_I4 (op1 ⇒ value)

Unary operation: 32 bit signed integer bitwise invert.

INV_I8 (op1 ⇒ value)

Unary operation: 64 bit signed integer bitwise invert.

INV_I4 (op1 ⇒ value)

Unary operation: 32 bit signed integer bitwise invert.

INV_U4 (op1 ⇒ value)

Unary operation: 32 bit unsigned integer bitwise invert.

INV_U8 (op1 ⇒ value)

Unary operation: 64 bit unsigned integer bitwise invert.

INV_U4 (op1 ⇒ value)

Unary operation: 32 bit unsigned integer bitwise invert.

IP (— ⇒ instruction_pointer)

Pushes the current (interpreter) instruction pointer to TOS.

JUMP (— ⇒ —)

2 operand bytes.

Only valid in bytecode context. Performs VM jump. Jumps by N bytes, where N is the next two-byte value in the instruction stream.

L2D (val ⇒ result)

Converts 64 bit signed integer to 64 bit floating point. This is a conversion, not a cast. It is the value that remains the same, not the bit-pattern.

L2F (val ⇒ result)

Converts 64 bit signed integer to 32 bit floating point. This is a conversion, not a cast. It is the value that remains the same, not the bit-pattern.

L2I (val ⇒ result)

Converts 64 bit signed integer to 32 bit signed integer. This is a conversion, not a cast. It is the value that remains the same, not the bit-pattern.

LADDR(name) (— ⇒ addr)

Pushes the address of the local variable 'name' to TOS.

LE_F4 (op1, op2 ⇒ comp)

Comparison operation: 32 bit floating point less than or equals.

comp := op1 ≤ op2.

LE_F8 (op1, op2 ⇒ comp)

Comparison operation: 64 bit floating point less than or equals.

comp := op1 ≤ op2.

LE_I4 (op1, op2 ⇒ comp)

Comparison operation: 32 bit signed integer less than or equals.

comp := op1 ≤ op2.

LE_I8 (op1, op2 ⇒ comp)

Comparison operation: 64 bit signed integer less than or equals.

comp := op1 ≤ op2.

LE_I4 (op1, op2 ⇒ comp)

Comparison operation: 32 bit signed integer less than or equals.

comp := op1 ≤ op2.

LE_P (op1, op2 ⇒ comp)

Comparison operation: pointer less than or equals.

comp := op1 ≤ op2.

LE_U4 (op1, op2 ⇒ comp)

Comparison operation: 32 bit unsigned integer less than or equals.

comp := op1 ≤ op2.

LE_U8 (op1, op2 ⇒ comp)

Comparison operation: 64 bit unsigned integer less than or equals.

comp := op1 ≤ op2.

LE_U4 (op1, op2 ⇒ comp)

Comparison operation: 32 bit unsigned integer less than or equals.

comp := op1 ≤ op2.

LINE(n) (— ⇒ —)

Set the source code line number of the source code. Informational only, like #LINE in C.

LOCK (lock ⇒ —)

Lock the gvmt-lock pointed to by TOS. Pop TOS.

LOCK_INTERNAL (offset, object ⇒ —)

Lock the gvmt-lock in object referred to by TOS at offset NOS. Pop both reference and offset from stack.

LSH_I4 (op1, op2 ⇒ result)

Binary operation: 32 bit signed integer left shift.

result := op1 << op2.

LSH_I8 (op1, op2 ⇒ result)

Binary operation: 64 bit signed integer left shift.

result := op1 << op2.

LSH_I4 (op1, op2 ⇒ result)

Binary operation: 32 bit signed integer left shift.
 result := op1 \ll op2.

LSH_U4 (op1, op2 ⇒ result)

Binary operation: 32 bit unsigned integer left shift.
 result := op1 \ll op2.

LSH_U8 (op1, op2 ⇒ result)

Binary operation: 64 bit unsigned integer left shift.
 result := op1 \ll op2.

LSH_U4 (op1, op2 ⇒ result)

Binary operation: 32 bit unsigned integer left shift.
 result := op1 \ll op2.

LT_F4 (op1, op2 ⇒ comp)

Comparison operation: 32 bit floating point less than.
 comp := op1 < op2.

LT_F8 (op1, op2 ⇒ comp)

Comparison operation: 64 bit floating point less than.
 comp := op1 < op2.

LT_I4 (op1, op2 ⇒ comp)

Comparison operation: 32 bit signed integer less than.
 comp := op1 < op2.

LT_I8 (op1, op2 ⇒ comp)

Comparison operation: 64 bit signed integer less than.
 comp := op1 < op2.

LT_I4 (op1, op2 ⇒ comp)

Comparison operation: 32 bit signed integer less than.
 comp := op1 < op2.

LT_P (op1, op2 ⇒ comp)

Comparison operation: pointer less than.
 comp := op1 < op2.

LT_U4 (op1, op2 ⇒ comp)

Comparison operation: 32 bit unsigned integer less than.
 comp := op1 < op2.

LT_U8 (op1, op2 ⇒ comp)

Comparison operation: 64 bit unsigned integer less than.
 comp := op1 < op2.

LT_U4 (op1, op2 ⇒ comp)

Comparison operation: 32 bit unsigned integer less than.
 comp := op1 < op2.

MOD_I4 (op1, op2 ⇒ result)

Binary operation: 32 bit signed integer modulo.
 result := op1

MOD_I8 (op1, op2 ⇒ result)

Binary operation: 64 bit signed integer modulo.
 result := op1

MOD_I4 (op1, op2 ⇒ result)

Binary operation: 32 bit signed integer modulo.
 result := op1

MOD_U4 (op1, op2 ⇒ result)

Binary operation: 32 bit unsigned integer modulo.
 result := op1

MOD_U8 (op1, op2 ⇒ result)

Binary operation: 64 bit unsigned integer modulo.
 result := op1

MOD_U4 (op1, op2 ⇒ result)

Binary operation: 32 bit unsigned integer modulo.
 result := op1

MUL_F4 (op1, op2 ⇒ result)

Binary operation: 32 bit floating point multiply.
 result := op1 × op2.

MUL_F8 (op1, op2 ⇒ result)

Binary operation: 64 bit floating point multiply.
 result := op1 × op2.

MUL_I4 (op1, op2 ⇒ result)

Binary operation: 32 bit signed integer multiply.
 result := op1 × op2.

MUL_I8 (op1, op2 ⇒ result)

Binary operation: 64 bit signed integer multiply.
 result := op1 × op2.

MUL_I4 (op1, op2 ⇒ result)

Binary operation: 32 bit signed integer multiply.
 result := op1 × op2.

MUL_U4 (op1, op2 ⇒ result)

Binary operation: 32 bit unsigned integer multiply.
result := op1 × op2.

MUL_U8 (op1, op2 ⇒ result)

Binary operation: 64 bit unsigned integer multiply.
result := op1 × op2.

MUL_U4 (op1, op2 ⇒ result)

Binary operation: 32 bit unsigned integer multiply.
result := op1 × op2.

NAME(n,name) (— ⇒ —)

Name the nth temporary variable, for debugging purposes.

NARG_F4 (val ⇒ —)

Native argument of type 32 bit floating point. TOS is pushed to the native argument stack.

NARG_F8 (val ⇒ —)

Native argument of type 64 bit floating point. TOS is pushed to the native argument stack.

NARG_I4 (val ⇒ —)

Native argument of type 32 bit signed integer. TOS is pushed to the native argument stack.

NARG_I8 (val ⇒ —)

Native argument of type 64 bit signed integer. TOS is pushed to the native argument stack.

NARG_I4 (val ⇒ —)

Native argument of type 32 bit signed integer. TOS is pushed to the native argument stack.

NARG_P (val ⇒ —)

Native argument of type pointer. TOS is pushed to the native argument stack.

NARG_U4 (val ⇒ —)

Native argument of type 32 bit unsigned integer. TOS is pushed to the native argument stack.

NARG_U8 (val ⇒ —)

Native argument of type 64 bit unsigned integer. TOS is pushed to the native argument stack.

NARG_U4 (val \Rightarrow —)

Native argument of type 32 bit unsigned integer. TOS is pushed to the native argument stack.

NEG_F4 (op1 \Rightarrow value)

Unary operation: 32 bit floating point negate.

NEG_F8 (op1 \Rightarrow value)

Unary operation: 64 bit floating point negate.

NEG_I4 (op1 \Rightarrow value)

Unary operation: 32 bit signed integer negate.

NEG_I8 (op1 \Rightarrow value)

Unary operation: 64 bit signed integer negate.

NEG_I4 (op1 \Rightarrow value)

Unary operation: 32 bit signed integer negate.

NEXT_IP (— \Rightarrow instruction_pointer)

Pushes the (interpreter) instruction pointer for the *next* instruction to TOS. This is equal to IP plus the length of the current bytecode

NE_F4 (op1, op2 \Rightarrow comp)

Comparison operation: 32 bit floating point not equals.
comp := op1 *eq* op2.

NE_F8 (op1, op2 \Rightarrow comp)

Comparison operation: 64 bit floating point not equals.
comp := op1 *eq* op2.

NE_I4 (op1, op2 \Rightarrow comp)

Comparison operation: 32 bit signed integer not equals.
comp := op1 *eq* op2.

NE_I8 (op1, op2 \Rightarrow comp)

Comparison operation: 64 bit signed integer not equals.
comp := op1 *eq* op2.

NE_I4 (op1, op2 \Rightarrow comp)

Comparison operation: 32 bit signed integer not equals.
comp := op1 *eq* op2.

NE_P (op1, op2 \Rightarrow comp)

Comparison operation: pointer not equals.
comp := op1 *eq* op2.

NE_R (op1, op2 ⇒ comp)

Comparison operation: reference not equals.

comp := op1 *eq* op2.

NE_U4 (op1, op2 ⇒ comp)

Comparison operation: 32 bit unsigned integer not equals.

comp := op1 *eq* op2.

NE_U8 (op1, op2 ⇒ comp)

Comparison operation: 64 bit unsigned integer not equals.

comp := op1 *eq* op2.

49

NE_U4 (op1, op2 ⇒ comp)

Comparison operation: 32 bit unsigned integer not equals.

comp := op1 *eq* op2.

N_CALL_F4(n) (— ⇒ value)

Calls the function whose address is TOS. Uses the native calling convention for this platform with 0 parameters which are popped from the native argument stack. Pushes the return value which must be a 32 bit floating point.

N_CALL_F8(n) (— ⇒ value)

Calls the function whose address is TOS. Uses the native calling convention for this platform with 0 parameters which are

popped from the native argument stack. Pushes the return value which must be a 64 bit floating point.

N_CALL_I4(n) (— ⇒ value)

Calls the function whose address is TOS. Uses the native calling convention for this platform with 0 parameters which are popped from the native argument stack. Pushes the return value which must be a 32 bit signed integer.

N_CALL_I4(n) (— ⇒ value)

Calls the function whose address is TOS. Uses the native calling convention for this platform with 0 parameters which are popped from the native argument stack. Pushes the return value which must be a 32 bit signed integer.

N_CALL_I8(n) (— ⇒ value)

Calls the function whose address is TOS. Uses the native calling convention for this platform with 0 parameters which are popped from the native argument stack. Pushes the return value which must be a 64 bit signed integer.

N_CALL_NO_GC_F4(n) (— ⇒ value)

As N_CALL_F4(n). Garbage collection is suspended during this call. Only use the NO_GC variant for calls which cannot block. If unsure use N_CALL.

N_CALL_NO_GC_F8(n) ($\text{---} \Rightarrow \text{value}$)

As N_CALL_F8(n). Garbage collection is suspended during this call. Only use the NO_GC variant for calls which cannot block. If unsure use N_CALL.

N_CALL_NO_GC_I4(n) ($\text{---} \Rightarrow \text{value}$)

As N_CALL_I4(n). Garbage collection is suspended during this call. Only use the NO_GC variant for calls which cannot block. If unsure use N_CALL.

N_CALL_NO_GC_I4(n) ($\text{---} \Rightarrow \text{value}$)

50

As N_CALL_I4(n). Garbage collection is suspended during this call. Only use the NO_GC variant for calls which cannot block. If unsure use N_CALL.

N_CALL_NO_GC_I8(n) ($\text{---} \Rightarrow \text{value}$)

As N_CALL_I8(n). Garbage collection is suspended during this call. Only use the NO_GC variant for calls which cannot block. If unsure use N_CALL.

N_CALL_NO_GC_P(n) ($\text{---} \Rightarrow \text{value}$)

As N_CALL_P(n). Garbage collection is suspended during this call. Only use the NO_GC variant for calls which cannot block. If unsure use N_CALL.

N_CALL_NO_GC_R(n) ($\text{---} \Rightarrow \text{value}$)

As N_CALL_R(n). Garbage collection is suspended during this call. Only use the NO_GC variant for calls which cannot block. If unsure use N_CALL.

N_CALL_NO_GC_U4(n) ($\text{---} \Rightarrow \text{value}$)

As N_CALL_U4(n). Garbage collection is suspended during this call. Only use the NO_GC variant for calls which cannot block. If unsure use N_CALL.

N_CALL_NO_GC_U4(n) ($\text{---} \Rightarrow \text{value}$)

As N_CALL_U4(n). Garbage collection is suspended during this call. Only use the NO_GC variant for calls which cannot block. If unsure use N_CALL.

N_CALL_NO_GC_U8(n) ($\text{---} \Rightarrow \text{value}$)

As N_CALL_U8(n). Garbage collection is suspended during this call. Only use the NO_GC variant for calls which cannot block. If unsure use N_CALL.

N_CALL_NO_GC_V(n) ($\text{---} \Rightarrow \text{value}$)

As N_CALL_V(n). Garbage collection is suspended during this call. Only use the NO_GC variant for calls which cannot block. If unsure use N_CALL.

N_CALL_P(n) ($\text{---} \Rightarrow \text{value}$)

Calls the function whose address is TOS. Uses the native calling convention for this platform with 0 parameters which are popped from the native argument stack. Pushes the return value which must be a pointer.

N_CALL_R(n) ($\text{---} \Rightarrow \text{value}$)

Calls the function whose address is TOS. Uses the native calling convention for this platform with 0 parameters which are popped from the native argument stack. Pushes the return value which must be a reference.

N_CALL_U4(n) ($\text{---} \Rightarrow \text{value}$)

Calls the function whose address is TOS. Uses the native calling convention for this platform with 0 parameters which are popped from the native argument stack. Pushes the return value which must be a 32 bit unsigned integer.

N_CALL_U4(n) ($\text{---} \Rightarrow \text{value}$)

Calls the function whose address is TOS. Uses the native calling convention for this platform with 0 parameters which are popped from the native argument stack. Pushes the return value which must be a 32 bit unsigned integer.

N_CALL_U8(n) ($\text{---} \Rightarrow \text{value}$)

Calls the function whose address is TOS. Uses the native calling convention for this platform with 0 parameters which are popped from the native argument stack. Pushes the return value which must be a 64 bit unsigned integer.

N_CALL_V(n) ($\text{---} \Rightarrow \text{value}$)

Calls the function whose address is TOS. Uses the native calling convention for this platform with 0 parameters which are popped from the native argument stack. Pushes the return value which must be a void.

OPCODE ($\text{---} \Rightarrow \text{opcode}$)

Pushes the current opcode to TOS.

OR_I4 (**op1, op2** \Rightarrow **result**)

Binary operation: 32 bit signed integer bitwise or.
 $\text{result} := \text{op1} \mid \text{op2}$.

OR_I8 (**op1, op2** \Rightarrow **result**)

Binary operation: 64 bit signed integer bitwise or.
 $\text{result} := \text{op1} \mid \text{op2}$.

OR_I4 (op1, op2 ⇒ result)

Binary operation: 32 bit signed integer bitwise or.
result := op1 | op2.

OR_U4 (op1, op2 ⇒ result)

Binary operation: 32 bit unsigned integer bitwise or.
result := op1 | op2.

OR_U8 (op1, op2 ⇒ result)

Binary operation: 64 bit unsigned integer bitwise or.
result := op1 | op2.

52

OR_U4 (op1, op2 ⇒ result)

Binary operation: 32 bit unsigned integer bitwise or.
result := op1 | op2.

PICK_F4 (— ⇒ nth)

1 operand byte.
Picks the nth item from the data stack(TOS is index 0)and pushes it to TOS.

PICK_F8 (— ⇒ nth)

1 operand byte.
Picks the nth item from the data stack(TOS is index 0)and pushes it to TOS.

PICK_I4 (— ⇒ nth)

1 operand byte.
Picks the nth item from the data stack(TOS is index 0)and pushes it to TOS.

PICK_I8 (— ⇒ nth)

1 operand byte.
Picks the nth item from the data stack(TOS is index 0)and pushes it to TOS.

PICK_I4 (— ⇒ nth)

1 operand byte.
Picks the nth item from the data stack(TOS is index 0)and pushes it to TOS.

PICK_P (— ⇒ nth)

1 operand byte.
Picks the nth item from the data stack(TOS is index 0)and pushes it to TOS.

PICK_R (— ⇒ nth)

1 operand byte.
Picks the nth item from the data stack(TOS is index 0)and pushes it to TOS.

PICK_U4 ($\text{---} \Rightarrow \mathbf{n^{\text{th}}}$)

1 operand byte.

Picks the n^{th} item from the data stack(TOS is index 0)and pushes it to TOS.

PICK_U8 ($\text{---} \Rightarrow \mathbf{n^{\text{th}}}$)

1 operand byte.

Picks the n^{th} item from the data stack(TOS is index 0)and pushes it to TOS.

PICK_U4 ($\text{---} \Rightarrow \mathbf{n^{\text{th}}}$)

1 operand byte.

Picks the n^{th} item from the data stack(TOS is index 0)and pushes it to TOS.

PIN (**object** \Rightarrow **pinned**)

Pins the object on TOS. Changes type of TOS from a reference to a pointer.

PINNED_OBJECT (**pointer** \Rightarrow **object**)

Declares that pointer is in fact a reference to a pinned object. Changes type of TOS from a pointer to a reference. It is an error if the pointer is not a reference to a pinned object. Incorrect use of this instruction can be difficult to detect. Use with care.

PLOAD_F4 (**addr** \Rightarrow **value**)

Load from memory. Push 32 bit floating point value loaded from address in TOS (which must be a pointer).

PLOAD_F8 (**addr** \Rightarrow **value**)

Load from memory. Push 64 bit floating point value loaded from address in TOS (which must be a pointer).

PLOAD_I1 (**addr** \Rightarrow **value**)

Load from memory. Push 8 bit signed integer value loaded from address in TOS (which must be a pointer).

PLOAD_I2 (**addr** \Rightarrow **value**)

Load from memory. Push 16 bit signed integer value loaded from address in TOS (which must be a pointer).

PLOAD_I4 (**addr** \Rightarrow **value**)

Load from memory. Push 32 bit signed integer value loaded from address in TOS (which must be a pointer).

PLOAD_I8 (**addr** \Rightarrow **value**)

Load from memory. Push 64 bit signed integer value loaded from address in TOS (which must be a pointer).

PLOAD_I4 (addr ⇒ value)

Load from memory. Push 32 bit signed integer value loaded from address in TOS (which must be a pointer).

PLOAD_P (addr ⇒ value)

Load from memory. Push pointer value loaded from address in TOS (which must be a pointer).

PLOAD_R (addr ⇒ value)

Load from memory. Push reference value loaded from address in TOS (which must be a pointer).

PLOAD_U1 (addr ⇒ value)

Load from memory. Push 8 bit unsigned integer value loaded from address in TOS (which must be a pointer).

PLOAD_U2 (addr ⇒ value)

Load from memory. Push 16 bit unsigned integer value loaded from address in TOS (which must be a pointer).

PLOAD_U4 (addr ⇒ value)

Load from memory. Push 32 bit unsigned integer value loaded from address in TOS (which must be a pointer).

PLOAD_U8 (addr ⇒ value)

Load from memory. Push 64 bit unsigned integer value loaded from address in TOS (which must be a pointer).

PLOAD_U4 (addr ⇒ value)

Load from memory. Push 32 bit unsigned integer value loaded from address in TOS (which must be a pointer).

POP_STATE (— ⇒ value)

Pops and discards the state-object on top of the state stack.

PSTORE_F4 (value, array ⇒ —)

Store to memory. Store 32 bit floating point value in NOS to address in TOS. (TOS must be a pointer)

PSTORE_F8 (value, array ⇒ —)

Store to memory. Store 64 bit floating point value in NOS to address in TOS. (TOS must be a pointer)

PSTORE_I1 (value, array ⇒ —)

Store to memory. Store 8 bit signed integer value in NOS to address in TOS. (TOS must be a pointer)

PSTORE_I2 (value, array ⇒ —)

Store to memory. Store 16 bit signed integer value in NOS to address in TOS. (TOS must be a pointer)

PSTORE_I4 (value, array ⇒ —)

Store to memory. Store 32 bit signed integer value in NOS to address in TOS. (TOS must be a pointer)

PSTORE_I8 (value, array ⇒ —)

Store to memory. Store 64 bit signed integer value in NOS to address in TOS. (TOS must be a pointer)

PSTORE_I4 (value, array ⇒ —)

Store to memory. Store 32 bit signed integer value in NOS to address in TOS. (TOS must be a pointer)

PSTORE_P (value, array ⇒ —)

Store to memory. Store pointer value in NOS to address in TOS. (TOS must be a pointer)

PSTORE_R (value, array ⇒ —)

Store to memory. Store reference value in NOS to address in TOS. (TOS must be a pointer)

PSTORE_U1 (value, array ⇒ —)

Store to memory. Store 8 bit unsigned integer value in NOS to address in TOS. (TOS must be a pointer)

PSTORE_U2 (value, array ⇒ —)

Store to memory. Store 16 bit unsigned integer value in NOS to address in TOS. (TOS must be a pointer)

PSTORE_U4 (value, array ⇒ —)

Store to memory. Store 32 bit unsigned integer value in NOS to address in TOS. (TOS must be a pointer)

PSTORE_U8 (value, array ⇒ —)

Store to memory. Store 64 bit unsigned integer value in NOS to address in TOS. (TOS must be a pointer)

PSTORE_U4 (value, array ⇒ —)

Store to memory. Store 32 bit unsigned integer value in NOS to address in TOS. (TOS must be a pointer)

PUSH_CURRENT_STATE (— ⇒ value)

Pushes a new state-object to the state stack and pushes 0 to TOS, when initially executed. When execution resumes after a RAISE or TRANSFER, then the value in the transfer register is pushed to TOS.

RAISE (value ⇒ —)

Pop TOS, which must be a reference, and place in the transfer register. Examine the state object on top of state stack. Pop values from the data-stack to the depth recorded. Resume execution from the PUSH_CURRENT_STATE instruction that stored the state object on the state stack.

RETURN_F4 (value ⇒ —)

Returns from the current function. Type must match that of CALL instruction.

RETURN_F8 (value ⇒ —)

Returns from the current function. Type must match that of CALL instruction.

RETURN_I4 (value ⇒ —)

Returns from the current function. Type must match that of CALL instruction.

RETURN_I8 (value ⇒ —)

Returns from the current function. Type must match that of CALL instruction.

RETURN_I4 (value ⇒ —)

Returns from the current function. Type must match that of CALL instruction.

RETURN_P (value ⇒ —)

Returns from the current function. Type must match that of CALL instruction.

RETURN_R (value ⇒ —)

Returns from the current function. Type must match that of CALL instruction.

RETURN_U4 (value ⇒ —)

Returns from the current function. Type must match that of CALL instruction.

RETURN_U8 (value ⇒ —)

Returns from the current function. Type must match that of CALL instruction.

RETURN_U4 (value ⇒ —)

Returns from the current function. Type must match that of CALL instruction.

RETURN_V (value \Rightarrow —)

Returns from the current function. Type must match that of CALL instruction.

RLOAD_F4 (object, offset \Rightarrow value)

Load from object. Load 32 bit floating point value from object NOS at offset TOS. (NOS must be a reference and TOS must be an integer)

RLOAD_F8 (object, offset \Rightarrow value)

Load from object. Load 64 bit floating point value from object NOS at offset TOS. (NOS must be a reference and TOS must be an integer)

RLOAD_I1 (object, offset \Rightarrow value)

Load from object. Load 8 bit signed integer value from object NOS at offset TOS. (NOS must be a reference and TOS must be an integer)

RLOAD_I2 (object, offset \Rightarrow value)

Load from object. Load 16 bit signed integer value from object NOS at offset TOS. (NOS must be a reference and TOS must be an integer)

RLOAD_I4 (object, offset \Rightarrow value)

Load from object. Load 32 bit signed integer value from object NOS at offset TOS. (NOS must be a reference and TOS must be an integer)

RLOAD_I8 (object, offset \Rightarrow value)

Load from object. Load 64 bit signed integer value from object NOS at offset TOS. (NOS must be a reference and TOS must be an integer)

RLOAD_I4 (object, offset \Rightarrow value)

Load from object. Load 32 bit signed integer value from object NOS at offset TOS. (NOS must be a reference and TOS must be an integer)

RLOAD_P (object, offset \Rightarrow value)

Load from object. Load pointer value from object NOS at offset TOS. (NOS must be a reference and TOS must be an integer)

RLOAD_R (object, offset \Rightarrow value)

Load from object. Load reference value from object NOS at offset TOS. (NOS must be a reference and TOS must be an integer)Any read-barriers required by the garbage collector are performed.

RLOAD_U1 (object, offset ⇒ value)

Load from object. Load 8 bit unsigned integer value from object NOS at offset TOS. (NOS must be a reference and TOS must be an integer)

RLOAD_U2 (object, offset ⇒ value)

Load from object. Load 16 bit unsigned integer value from object NOS at offset TOS. (NOS must be a reference and TOS must be an integer)

RLOAD_U4 (object, offset ⇒ value)

∞

Load from object. Load 32 bit unsigned integer value from object NOS at offset TOS. (NOS must be a reference and TOS must be an integer)

RLOAD_U8 (object, offset ⇒ value)

Load from object. Load 64 bit unsigned integer value from object NOS at offset TOS. (NOS must be a reference and TOS must be an integer)

RLOAD_U4 (object, offset ⇒ value)

Load from object. Load 32 bit unsigned integer value from object NOS at offset TOS. (NOS must be a reference and TOS must be an integer)

RSH_I4 (op1, op2 ⇒ result)

Binary operation: 32 bit signed integer arithmetic right shift.
result := op1 \gg op2.

RSH_I8 (op1, op2 ⇒ result)

Binary operation: 64 bit signed integer arithmetic right shift.
result := op1 \gg op2.

RSH_I4 (op1, op2 ⇒ result)

Binary operation: 32 bit signed integer arithmetic right shift.
result := op1 \gg op2.

RSH_U4 (op1, op2 ⇒ result)

Binary operation: 32 bit unsigned integer logical right shift.
result := op1 \gg op2.

RSH_U8 (op1, op2 ⇒ result)

Binary operation: 64 bit unsigned integer logical right shift.
result := op1 \gg op2.

RSH_U4 (op1, op2 ⇒ result)

Binary operation: 32 bit unsigned integer logical right shift.
result := op1 \gg op2.

RSTORE_F4 (value, object, offset ⇒ —)

Store into object. Store 32 bit floating point value at 3OS into object NOS, offset TOS. (NOS must be a reference and TOS must be an integer)

RSTORE_F8 (value, object, offset ⇒ —)

Store into object. Store 64 bit floating point value at 3OS into object NOS, offset TOS. (NOS must be a reference and TOS must be an integer)

RSTORE_I1 (value, object, offset ⇒ —)

69 Store into object. Store 8 bit signed integer value at 3OS into object NOS, offset TOS. (NOS must be a reference and TOS must be an integer)

RSTORE_I2 (value, object, offset ⇒ —)

Store into object. Store 16 bit signed integer value at 3OS into object NOS, offset TOS. (NOS must be a reference and TOS must be an integer)

RSTORE_I4 (value, object, offset ⇒ —)

Store into object. Store 32 bit signed integer value at 3OS into object NOS, offset TOS. (NOS must be a reference and TOS must be an integer)

RSTORE_I8 (value, object, offset ⇒ —)

Store into object. Store 64 bit signed integer value at 3OS into object NOS, offset TOS. (NOS must be a reference and TOS must be an integer)

RSTORE_I4 (value, object, offset ⇒ —)

Store into object. Store 32 bit signed integer value at 3OS into object NOS, offset TOS. (NOS must be a reference and TOS must be an integer)

RSTORE_P (value, object, offset ⇒ —)

Store into object. Store pointer value at 3OS into object NOS, offset TOS. (NOS must be a reference and TOS must be an integer)

RSTORE_R (value, object, offset ⇒ —)

Store into object. Store reference value at 3OS into object NOS, offset TOS. (NOS must be a reference and TOS must be an integer) Any write-barriers required by the garbage collector are performed.

RSTORE_U1 (value, object, offset ⇒ —)

Store into object. Store 8 bit unsigned integer value at 3OS into object NOS, offset TOS. (NOS must be a reference and TOS must be an integer)

RSTORE_U2 (value, object, offset ⇒ —)

Store into object. Store 16 bit unsigned integer value at 3OS into object NOS, offset TOS. (NOS must be a reference and TOS must be an integer)

RSTORE_U4 (value, object, offset ⇒ —)

Store into object. Store 32 bit unsigned integer value at 3OS into object NOS, offset TOS. (NOS must be a reference and TOS must be an integer)

RSTORE_U8 (value, object, offset ⇒ —)

Store into object. Store 64 bit unsigned integer value at 3OS into object NOS, offset TOS. (NOS must be a reference and TOS must be an integer)

RSTORE_U4 (value, object, offset ⇒ —)

Store into object. Store 32 bit unsigned integer value at 3OS into object NOS, offset TOS. (NOS must be a reference and TOS must be an integer)

SIGN (val ⇒ extended)

On a 32 bit machine, sign extend TOS from a 32 bit value to a 64 bit value. This is a no-op for 64bit machines.

STACK (— ⇒ sp)

Pushes the data-stack stack-pointer to TOS. The data stack grows downwards, so stack items will be at non-negative offsets from sp. Values subsequently pushed on to the stack are not visible. Attempting to access values at negative offsets is an error. As soon as a net positive number of values are popped from the stack, sp becomes invalid and should *not* be used.

SUB_F4 (op1, op2 ⇒ result)

Binary operation: 32 bit floating point subtract.
result := op1 - op2.

SUB_F8 (op1, op2 ⇒ result)

Binary operation: 64 bit floating point subtract.
result := op1 - op2.

SUB_I4 (op1, op2 ⇒ result)

Binary operation: 32 bit signed integer subtract.
result := op1 - op2.

SUB_I8 (op1, op2 ⇒ result)

Binary operation: 64 bit signed integer subtract.
result := op1 - op2.

SUB_I4 (op1, op2 ⇒ result)

Binary operation: 32 bit signed integer subtract.
 result := op1 - op2.

SUB_P (op1, op2 ⇒ result)

Binary operation: pointer subtract.
 result := op1 - op2.

SUB_U4 (op1, op2 ⇒ result)

Binary operation: 32 bit unsigned integer subtract.
 result := op1 - op2.

SUB_U8 (op1, op2 ⇒ result)

Binary operation: 64 bit unsigned integer subtract.
 result := op1 - op2.

SUB_U4 (op1, op2 ⇒ result)

Binary operation: 32 bit unsigned integer subtract.
 result := op1 - op2.

SYMBOL (— ⇒ address)

2 operand bytes.
 Push address of symbol to TOS

TARGET(n) (— ⇒ —)

Target for Jump and Branch.

TLOAD_F4(n) (— ⇒ value)

Push the contents of the nth temporary variable as a 32 bit floating point

TLOAD_F8(n) (— ⇒ value)

Push the contents of the nth temporary variable as a 64 bit floating point

TLOAD_I4(n) (— ⇒ value)

Push the contents of the nth temporary variable as a 32 bit signed integer

TLOAD_I4(n) (— ⇒ value)

Push the contents of the nth temporary variable as a 32 bit signed integer

TLOAD_I8(n) (— ⇒ value)

Push the contents of the nth temporary variable as a 64 bit signed integer

TLOAD_P(n) ($\text{---} \Rightarrow \text{value}$)

Push the contents of the n^{th} temporary variable as a pointer

TLOAD_R(n) ($\text{---} \Rightarrow \text{value}$)

Push the contents of the n^{th} temporary variable as a reference

TLOAD_U4(n) ($\text{---} \Rightarrow \text{value}$)

Push the contents of the n^{th} temporary variable as a 32 bit unsigned integer

TLOAD_U4(n) ($\text{---} \Rightarrow \text{value}$)

Push the contents of the n^{th} temporary variable as a 32 bit unsigned integer

TLOAD_U8(n) ($\text{---} \Rightarrow \text{value}$)

Push the contents of the n^{th} temporary variable as a 64 bit unsigned integer

TRANSFER ($\text{---} \Rightarrow \text{---}$)

Pop TOS, which must be a reference, and place in the transfer register. Resume execution from the PUSH_CURRENT_STATE instruction that stored the state object on the state stack. Unlike RAISE, TRANSFER does not modify the data stack.

TSTORE_F4(n) ($\text{value} \Rightarrow \text{---}$)

Pop a 32 bit floating point from the stack and store in the n^{th} temporary variable.

TSTORE_F8(n) ($\text{value} \Rightarrow \text{---}$)

Pop a 64 bit floating point from the stack and store in the n^{th} temporary variable.

TSTORE_I4(n) ($\text{value} \Rightarrow \text{---}$)

Pop a 32 bit signed integer from the stack and store in the n^{th} temporary variable.

TSTORE_I4(n) ($\text{value} \Rightarrow \text{---}$)

Pop a 32 bit signed integer from the stack and store in the n^{th} temporary variable.

TSTORE_I8(n) ($\text{value} \Rightarrow \text{---}$)

Pop a 64 bit signed integer from the stack and store in the n^{th} temporary variable.

TSTORE_P(n) ($\text{value} \Rightarrow \text{---}$)

Pop a pointer from the stack and store in the n^{th} temporary variable.

TSTORE_R(n) (value \Rightarrow —)

Pop a reference from the stack and store in the n^{th} temporary variable.

TSTORE_U4(n) (value \Rightarrow —)

Pop a 32 bit unsigned integer from the stack and store in the n^{th} temporary variable.

TSTORE_U4(n) (value \Rightarrow —)

Pop a 32 bit unsigned integer from the stack and store in the n^{th} temporary variable.

TSTORE_U8(n) (value \Rightarrow —)

Pop a 64 bit unsigned integer from the stack and store in the n^{th} temporary variable.

TYPE_NAME(n,name) (— \Rightarrow —)

Name the (reference) type of the n^{th} temporary variable, for debugging purposes.

UNLOCK (lock \Rightarrow —)

Unlock the gvmt-lock pointed to by TOS. Pop TOS.

UNLOCK_INTERNAL (offset, object \Rightarrow —)

Unlock the fast-lock in object referred to by TOS at offset NOS. Pop both reference and offset from stack.

V_CALL_F4 (— \Rightarrow value)

1 operand byte.

Variadic call. The number of parameters, n , is the next byte in the instruction stream (which is consumed). Calls the function whose address is TOS. Upon return removes the n parameters are from the data stack. The function called must return a 32 bit floating point.

V_CALL_F8 (— \Rightarrow value)

1 operand byte.

Variadic call. The number of parameters, n , is the next byte in the instruction stream (which is consumed). Calls the function whose address is TOS. Upon return removes the n parameters are from the data stack. The function called must return a 64 bit floating point.

V_CALL_I4 (— \Rightarrow value)

1 operand byte.

Variadic call. The number of parameters, n , is the next byte in the instruction stream (which is consumed). Calls the function whose address is TOS. Upon return removes the n pa-

rameters are from the data stack. The function called must return a 32 bit signed integer.

V_CALL_I8 (— ⇒ **value**)

1 operand byte.

Variadic call. The number of parameters, *n*, is the next byte in the instruction stream (which is consumed). Calls the function whose address is TOS. Upon return removes the *n* parameters are from the data stack. The function called must return a 64 bit signed integer.

V_CALL_I4 (— ⇒ **value**)

1 operand byte.

Variadic call. The number of parameters, *n*, is the next byte in the instruction stream (which is consumed). Calls the function whose address is TOS. Upon return removes the *n* parameters are from the data stack. The function called must return a 32 bit signed integer.

V_CALL_P (— ⇒ **value**)

1 operand byte.

Variadic call. The number of parameters, *n*, is the next byte in the instruction stream (which is consumed). Calls the function whose address is TOS. Upon return removes the *n* parameters are from the data stack. The function called must return a pointer.

V_CALL_R (— ⇒ **value**)

1 operand byte.

Variadic call. The number of parameters, *n*, is the next byte in the instruction stream (which is consumed). Calls the function whose address is TOS. Upon return removes the *n* parameters are from the data stack. The function called must return a reference.

V_CALL_U4 (— ⇒ **value**)

1 operand byte.

Variadic call. The number of parameters, *n*, is the next byte in the instruction stream (which is consumed). Calls the function whose address is TOS. Upon return removes the *n* parameters are from the data stack. The function called must return a 32 bit unsigned integer.

V_CALL_U8 (— ⇒ **value**)

1 operand byte.

Variadic call. The number of parameters, *n*, is the next byte in the instruction stream (which is consumed). Calls the function whose address is TOS. Upon return removes the *n* parameters are from the data stack. The function called must return a 64 bit unsigned integer.

V_CALL_U4 ($\text{—} \Rightarrow \text{value}$)

1 operand byte.

Variadic call. The number of parameters, n , is the next byte in the instruction stream (which is consumed). Calls the function whose address is TOS. Upon return removes the n parameters are from the data stack. The function called must return a 32 bit unsigned integer.

V_CALL_V ($\text{—} \Rightarrow \text{value}$)

1 operand byte.

Variadic call. The number of parameters, n , is the next byte in the instruction stream (which is consumed). Calls the function whose address is TOS. Upon return removes the n parameters are from the data stack. The function called must return void.

XOR_I4 (**op1, op2** \Rightarrow **result**)

Binary operation: 32 bit signed integer bitwise exclusive or.
 $\text{result} := \text{op1} \oplus \text{op2}$.

XOR_I8 (**op1, op2** \Rightarrow **result**)

Binary operation: 64 bit signed integer bitwise exclusive or.
 $\text{result} := \text{op1} \oplus \text{op2}$.

XOR_I4 (**op1, op2** \Rightarrow **result**)

Binary operation: 32 bit signed integer bitwise exclusive or.
 $\text{result} := \text{op1} \oplus \text{op2}$.

XOR_U4 (**op1, op2** \Rightarrow **result**)

Binary operation: 32 bit unsigned integer bitwise exclusive or.
 $\text{result} := \text{op1} \oplus \text{op2}$.

XOR_U8 (**op1, op2** \Rightarrow **result**)

Binary operation: 64 bit unsigned integer bitwise exclusive or.
 $\text{result} := \text{op1} \oplus \text{op2}$.

XOR_U4 (**op1, op2** \Rightarrow **result**)

Binary operation: 32 bit unsigned integer bitwise exclusive or.
 $\text{result} := \text{op1} \oplus \text{op2}$.

ZERO (**val** \Rightarrow **extended**)

On a 32 bit machine, zero extend TOS from a 32 bit value to a 64 bit value. This is a no-op for 64bit machines.