

# 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

<b>1</b>	<b>Introduction</b>	<b>2</b>
1.1	Example virtual machine . . . . .	2
1.2	The Tools . . . . .	3
1.2.1	The front-end tools, GVMTC GVMTICand GVMTXC . .	3
1.2.2	The code-generators, gvmtas and gvmtec . . . . .	4
1.2.3	The linker, gvmmlink . . . . .	5
<b>2</b>	<b>The tools</b>	<b>6</b>
2.1	Core tools . . . . .	6
2.1.1	Interpreter description file . . . . .	7
2.1.2	The interpreter generator GVMTIC . . . . .	9
2.1.3	The C compiler GVMTC . . . . .	10
2.1.4	The GSC assembler GVMTAS . . . . .	11
2.1.5	The compiler generator GVMTEC . . . . .	11
2.1.6	The linker GVMTLINK . . . . .	11
2.1.7	lcc-gvmt . . . . .	12
2.2	Other tools . . . . .	12
2.2.1	The bytecode-processor generator GVMTXC . . . . .	12
2.2.2	The object layout tool . . . . .	13
<b>3</b>	<b>GVMT Abstract Machine</b>	<b>14</b>
3.1	Components . . . . .	14
3.2	Threads . . . . .	15
3.2.1	Execution Model . . . . .	15
3.2.2	Functions . . . . .	15
3.2.3	Interpreters . . . . .	15
3.2.4	Compiled Code . . . . .	16
3.2.5	Abstract Machine Instructions . . . . .	16
3.3	Data Types . . . . .	16
3.4	The Stacks . . . . .	17
3.4.1	The Data Stack . . . . .	17

3.4.2	GC Safe points . . . . .	17
3.4.3	The Control Stack . . . . .	18
3.4.4	The Native Parameter Stack . . . . .	18
3.4.5	The State Stack . . . . .	18
3.4.6	Raise and Transfer . . . . .	18
3.4.7	Thread-local variables . . . . .	18
3.5	The heap . . . . .	19
3.5.1	Shape . . . . .	19
3.5.2	Garbage collection . . . . .	19
3.5.3	Exception handling . . . . .	20
3.6	Concurrency . . . . .	21
3.7	Memory . . . . .	21
3.7.1	Sizes and Alignments . . . . .	22
<b>4</b>	<b>Building a VM using the Toolkit</b>	<b>23</b>
4.1	Before you start . . . . .	23
4.2	Defining the components . . . . .	23
4.3	Debugging . . . . .	24
4.4	Adding a compiler . . . . .	24
<b>5</b>	<b>User interface</b>	<b>25</b>
5.1	User provided code . . . . .	25
5.1.1	Garbage collection . . . . .	25
5.1.2	High-Performance Garbage Collection Interface . . . . .	26
5.1.3	The Marshalling Interface . . . . .	26
5.1.4	Debugging The Interpreter . . . . .	27
5.2	GVMT provided functions . . . . .	27
5.2.1	The data stack . . . . .	27
5.2.2	The garbage collector . . . . .	28
5.2.3	Exception handling . . . . .	29
5.2.4	Threading . . . . .	29
<b>A</b>	<b>Installing the GVMT</b>	<b>31</b>
<b>B</b>	<b>The Abstract Machine Instruction Set</b>	<b>32</b>
<b>C</b>	<b>Implementing and Porting the GVMT</b>	<b>68</b>
C.1	Front-end Tools . . . . .	68
C.2	Mapping the Abstract Machine to the Hardware . . . . .	68
C.2.1	Garbage Collector Interface . . . . .	68
C.2.2	The Data Stack . . . . .	69

C.2.3	The Control Stack . . . . .	69
C.2.4	The State Stack . . . . .	69
C.2.5	The Native Parameter Stack . . . . .	70
C.3	The IA-32 Implementation . . . . .	70
C.3.1	Calls and Returns . . . . .	70
C.3.2	Saving Machine State . . . . .	71

# 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,<sup>1</sup> 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 tools, `gvmtc`, `gvmtic` and `gvmtxc`

The three front-end tools are used to convert the VM source code into the intermediate representation of the GVMT, GVMT abstract machine code (GAMC). GVMTCC is the C compiler and compiles C files to GAMC files. GVMTIC is the interpreter generator, which takes an interpreter specification file and converts into a GAMC file. The interpreter specification is in the

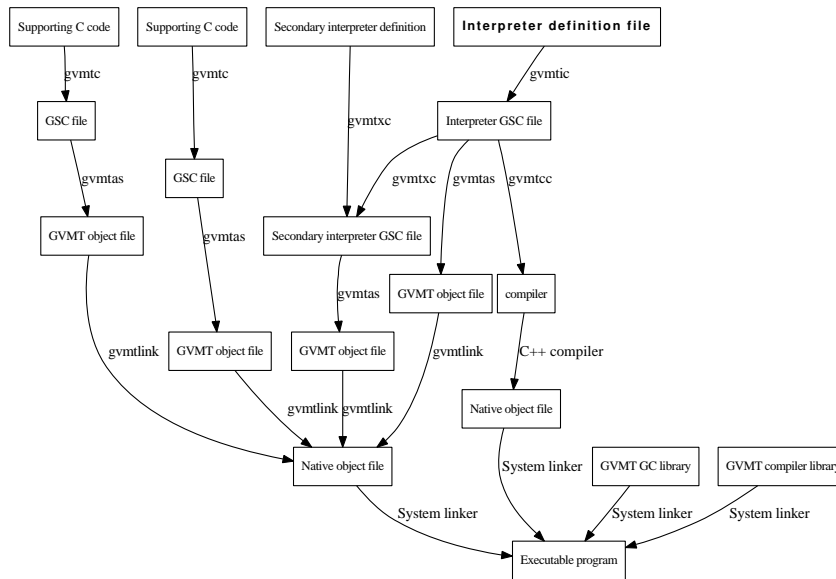
---

<sup>1</sup>JIT is short for Just-In-Time and is the standard term for runtime compilers

form of a list of bytecodes; each of which consists of a name, a stack effect comment and a C code (or GAMC) body. GVMTXC is the secondary interpreter generator, its takes a GAMC file containing a bytecode section representing the main interpreter, plus a secondary interpreter specification file produce a secondary interpreter 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.

All three 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 GAMC as input and create object files or executables.

GVMTAS compiles the GAMC into the GVMT object format, GSO. It works by converting GAMC 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 GAMC file containing a bytecode section and produces a bytecode to machine-code. Currently GVMTCC generates llvm-IR<sup>2</sup> and uses LLVM to generate machine-code at runtime (as a JIT compiler).

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.

### **1.2.3 The linker, gvmmlink**

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.

---

<sup>2</sup>See [llvm.org](http://llvm.org) for more details

# 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  $N^{\text{th}}$  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 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 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. A `CALL_X` instruction is then executed, where `X` depends on the type specified in the `gvmt_start_thread` function, which is then executed as follows.

### 3.2.2 Functions

A function in GVMT is defined as a linear sequence of instructions.

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 the current frame for accessing all temporary variables. The internal layout of this frame is not defined. Temporary variables in frames other than the current frame cannot be accessed. The first instruction in the function is then executed, proceeding to the next instruction and so on. The exceptions to this are flow control instructions such as `HOP` or `BRANCH` which may jump to a designated successor instruction.

### 3.2.3 Interpreters

An interpreter acts externally like a normal function; it can be called like any other. Internally, its behaviour is substantially different from that of a normal function.

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 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 terms<sup>1</sup>, 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.

## 3.3 Data Types

The GVMT supports 12 different types; 8 integer types (4 signed and 4 unsigned types), 2 floating point types and 2 pointer types. Integers can be 1,2,4 or 8 bytes in size. Floating point numbers can be 4 or 8 bytes in size.

---

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

Most instructions have a type suffix of the form **Xn**, where **X** is the first letter of the type and **n** is the size. The size is omitted for floating point and integer types. For example, the instruction to perform signed add of two 4-byte integers is **ADD\_I4**.

Signed and unsigned integer operations are interchangeable except for those operations such as multiplication and shifts where the sign makes a difference.

The GVMT does not specify alignments and byte-order, but implementations will match the underlying architecture.

## 3.4 The Stacks

### 3.4.1 The Data Stack

The data stack is where all arithmetic operations take 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.

### 3.4.2 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.<sup>2</sup>

The front-end tools **GVMT** and **GVMTIC** 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.

---

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

### 3.4.3 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.

### 3.4.4 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.

### 3.4.5 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.

### 3.4.6 Raise and Transfer

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

### 3.4.7 Thread-local variables

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

## 3.5 The heap

The GVMT Abstract Machine heap is a garbage collected heap. The toolkit user needs to provide a function `gvmt_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.

### 3.5.1 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 ]`.

### 3.5.2 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.

### 3.5.3 Exception handling

The GVMT exception handling model uses an explicit stack, rather than tables. These 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 re-raised.

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



## 3.6 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
```

## 3.7 Memory

The shared memory of the GVMT abstract machine is divided into N parts: Global references, the garbage collected heap and user-managed memory.

Global references are declared in the `.roots` section, the only valid data in the `.roots` section is an `address`. All `addresses` in the `.roots` section must refer to a label in the `.heap` section. All garbage collected object must reside in the `.heap` section. Each object begins with an `.object ID` directive and ends with an `' .end'` directive.

### 3.7.1 Sizes and Alignments

The sizes and alignments of data is determined by the underlying operating system and hardware. All data must conform to the hardware alignments. In order to ensure this is the case all objects must be align by the greatest alignment of any of the primitive data types, usually I8 or F8. This means that the front-end tools need to be aware of the target machine, in order to use the correct offsets.

# 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. GVMTIC 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 GVMTIC and GVMTTC 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 (some 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 using the native compiler, 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 3.5.1. 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 GAMC code, the user should also provide the `void gvmc_readable_name(GVMC_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 `GVMC_MAX_OBJECT_NAME_LENGTH` bytes in length.

`int GVMC_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 GVMC 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 GVMC 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 GVMC provided functions

GVMC 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:

- `gvmc_stack_top()` returns a pointer to the top-of-stack.
- `gvmc_insert(size_t n)` inserts `n` NULLs onto the stack and returns a pointer to them.
- `GVMC_PUSH(x)` can be used to push individual values on to the stack.
- `gvmc_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 GVMt-space 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@dcsgla.ac.uk](mailto:marks@dcsgla.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

**#+** ( $\text{---} \Rightarrow \text{---}$ )

*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.

**#-** ( $\text{---} \Rightarrow \text{---}$ )

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

*No operand bytes. Pushes 1 byte to instruction stream.*

Push 1 byte value to the front of the instruction stream.

**#2@** ( $\text{---} \Rightarrow \text{operand}$ )

*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@** ( $\text{---} \Rightarrow \text{operand}$ )

*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.

**#@** ( $\text{---} \Rightarrow \text{operand}$ )

*1 operand byte.*

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

**#[n]** ( $\text{---} \Rightarrow \text{---}$ )

*No operand bytes. Pushes 1 byte to instruction stream.*

Only valid in an interpreter definition. Peeks into the instruction stream and pushes the  $n^{\text{th}}$  byte in the stream to the front of the instruction stream.

**ADDR(name)** ( $\text{---} \Rightarrow \text{address}$ )

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

**ADD\_F4 (op1, op2  $\Rightarrow$  result)**

Binary operation: 32 bit floating point add.

result := op1 + op2.

**ADD\_F8 (op1, op2  $\Rightarrow$  result)**

Binary operation: 64 bit floating point add.

result := op1 + op2.

**ADD\_I4 (op1, op2  $\Rightarrow$  result)**

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

**ADD\_I8 (op1, op2  $\Rightarrow$  result)**

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

**ADD\_I4 (op1, op2  $\Rightarrow$  result)**

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

**ADD\_P (op1, op2  $\Rightarrow$  result)**

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

**ADD\_U4 (op1, op2  $\Rightarrow$  result)**

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

**ADD\_U8 (op1, op2  $\Rightarrow$  result)**

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

**ADD\_U4 (op1, op2  $\Rightarrow$  result)**

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

**ALLOC\_F4 (n  $\Rightarrow$  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.

**ALLOC\_F8 (n  $\Rightarrow$  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.

**ALLOC\_I1 (n  $\Rightarrow$  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 $\Rightarrow$ 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 $\Rightarrow$ 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 $\Rightarrow$ 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 $\Rightarrow$ 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 $\Rightarrow$ 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 $\Rightarrow$ 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  $\Rightarrow$  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  $\Rightarrow$  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  $\Rightarrow$  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  $\Rightarrow$  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  $\Rightarrow$  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  $\Rightarrow$  result)**

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

**AND\_I8 (op1, op2  $\Rightarrow$  result)**

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



**AND\_I4 (op1, op2  $\Rightarrow$  result)**

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

**AND\_U4 (op1, op2  $\Rightarrow$  result)**

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

**AND\_U8 (op1, op2  $\Rightarrow$  result)**

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

**AND\_U4 (op1, op2  $\Rightarrow$  result)**

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

**BRANCH\_F(n) (cond  $\Rightarrow$  —)**

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

**BRANCH\_T(n) (cond  $\Rightarrow$  —)**

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

**CALL\_F4 (—  $\Rightarrow$  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 (—  $\Rightarrow$  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 (—  $\Rightarrow$  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 (—  $\Rightarrow$  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  $\Rightarrow$  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  $\Rightarrow$  result)**

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

**DIV\_F8 (op1, op2  $\Rightarrow$  result)**

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

**DIV\_I4 (op1, op2  $\Rightarrow$  result)**

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

**DIV\_I8 (op1, op2  $\Rightarrow$  result)**

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

**DIV\_I4 (op1, op2  $\Rightarrow$  result)**

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

**DIV\_U4 (op1, op2  $\Rightarrow$  result)**

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

**DIV\_U8 (op1, op2  $\Rightarrow$  result)**

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

**DIV\_U4 (op1, op2  $\Rightarrow$  result)**

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

**DROP (top  $\Rightarrow$  —)**

Drops the top value from the stack.

**DROP\_N (n  $\Rightarrow$  —)**

*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  $\Rightarrow$  comp)**

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

**EQ\_F8 (op1, op2  $\Rightarrow$  comp)**

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

**EQ\_I4 (op1, op2  $\Rightarrow$  comp)**

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

**EQ\_I8 (op1, op2  $\Rightarrow$  comp)**

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

**EQ\_I4 (op1, op2  $\Rightarrow$  comp)**

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

**EQ\_P (op1, op2  $\Rightarrow$  comp)**

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

**EQ\_R (op1, op2  $\Rightarrow$  comp)**

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

**EQ\_U4 (op1, op2  $\Rightarrow$  comp)**

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

**EQ\_U8 (op1, op2  $\Rightarrow$  comp)**

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

**EQ\_U4 (op1, op2  $\Rightarrow$  comp)**

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

**EXT\_I1 (value  $\Rightarrow$  extended)**

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

**EXT\_I2 (value  $\Rightarrow$  extended)**

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

**EXT\_I4 (value  $\Rightarrow$  extended)**

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

**EXT\_I4 (value  $\Rightarrow$  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.

comp := op1  $\geq$  op2.

### **GE\_F8 (op1, op2 $\Rightarrow$ comp)**

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

comp := op1  $\geq$  op2.

### **GE\_I4 (op1, op2 $\Rightarrow$ comp)**

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

comp := op1  $\geq$  op2.

**GE\_I8 (op1, op2  $\Rightarrow$  comp)**

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

comp := op1  $\geq$  op2.

**GE\_I4 (op1, op2  $\Rightarrow$  comp)**

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

comp := op1  $\geq$  op2.

**GE\_P (op1, op2  $\Rightarrow$  comp)**

Comparison operation: pointer greater than or equals.

comp := op1  $\geq$  op2.

**GE\_U4 (op1, op2  $\Rightarrow$  comp)**

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

comp := op1  $\geq$  op2.

**GE\_U8 (op1, op2  $\Rightarrow$  comp)**

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

comp := op1  $\geq$  op2.

**GE\_U4 (op1, op2  $\Rightarrow$  comp)**

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

comp := op1  $\geq$  op2.

**GT\_F4 (op1, op2  $\Rightarrow$  comp)**

Comparison operation: 32 bit floating point greater than.

comp := op1  $>$  op2.

**GT\_F8 (op1, op2  $\Rightarrow$  comp)**

Comparison operation: 64 bit floating point greater than.

comp := op1  $>$  op2.

**GT\_I4 (op1, op2  $\Rightarrow$  comp)**

Comparison operation: 32 bit signed integer greater than.

comp := op1  $>$  op2.

**GT\_I8 (op1, op2  $\Rightarrow$  comp)**

Comparison operation: 64 bit signed integer greater than.

comp := op1  $>$  op2.

**GT\_I4 (op1, op2  $\Rightarrow$  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  $\Rightarrow$  value)**

Unary operation: 64 bit unsigned integer bitwise invert.

**INV\_U4 (op1  $\Rightarrow$  value)**

Unary operation: 32 bit unsigned integer bitwise invert.

**IP ( $\text{---} \Rightarrow$  instruction\_pointer)**

Pushes the current (interpreter) instruction pointer to TOS.

**JUMP ( $\text{---} \Rightarrow \text{---}$ )**

*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  $\Rightarrow$  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  $\Rightarrow$  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  $\Rightarrow$  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) ( $\text{---} \Rightarrow$  addr)**

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

**LE\_F4 (op1, op2  $\Rightarrow$  comp)**

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

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

**LE\_F8 (op1, op2  $\Rightarrow$  comp)**

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

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

**LE\_I4 (op1, op2  $\Rightarrow$  comp)**

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

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

**LE\_I8 (op1, op2  $\Rightarrow$  comp)**

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

comp := op1  $\leq$  op2.

**LE\_I4 (op1, op2  $\Rightarrow$  comp)**

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

comp := op1  $\leq$  op2.

**LE\_P (op1, op2  $\Rightarrow$  comp)**

Comparison operation: pointer less than or equals.

comp := op1  $\leq$  op2.

**LE\_U4 (op1, op2  $\Rightarrow$  comp)**

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

comp := op1  $\leq$  op2.

**LE\_U8 (op1, op2  $\Rightarrow$  comp)**

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

comp := op1  $\leq$  op2.

**LE\_U4 (op1, op2  $\Rightarrow$  comp)**

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

comp := op1  $\leq$  op2.

**LINE(n) ( $\text{---} \Rightarrow \text{---}$ )**

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

**LOCK (lock  $\Rightarrow \text{---}$ )**

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

**LOCK\_INTERNAL (offset, object  $\Rightarrow \text{---}$ )**

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

**LSH\_I4 (op1, op2  $\Rightarrow$  result)**

Binary operation: 32 bit signed integer left shift.

result := op1  $\ll$  op2.

**LSH\_I8 (op1, op2  $\Rightarrow$  result)**

Binary operation: 64 bit signed integer left shift.

result := op1  $\ll$  op2.

**LSH\_I4 (op1, op2  $\Rightarrow$  result)**

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

**LSH\_U4 (op1, op2  $\Rightarrow$  result)**

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

**LSH\_U8 (op1, op2  $\Rightarrow$  result)**

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

**LSH\_U4 (op1, op2  $\Rightarrow$  result)**

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

**LT\_F4 (op1, op2  $\Rightarrow$  comp)**

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

**LT\_F8 (op1, op2  $\Rightarrow$  comp)**

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

**LT\_I4 (op1, op2  $\Rightarrow$  comp)**

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

**LT\_I8 (op1, op2  $\Rightarrow$  comp)**

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

**LT\_I4 (op1, op2  $\Rightarrow$  comp)**

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

**LT\_P (op1, op2  $\Rightarrow$  comp)**

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

**LT\_U4 (op1, op2  $\Rightarrow$  comp)**

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

**LT\_U8 (op1, op2  $\Rightarrow$  comp)**

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

**LT\_U4 (op1, op2  $\Rightarrow$  comp)**

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

**MOD\_I4 (op1, op2  $\Rightarrow$  result)**

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

**MOD\_I8 (op1, op2  $\Rightarrow$  result)**

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

**MOD\_I4 (op1, op2  $\Rightarrow$  result)**

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

**MOD\_U4 (op1, op2  $\Rightarrow$  result)**

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

**MOD\_U8 (op1, op2  $\Rightarrow$  result)**

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

**MOD\_U4 (op1, op2  $\Rightarrow$  result)**

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

**MUL\_F4 (op1, op2  $\Rightarrow$  result)**

Binary operation: 32 bit floating point multiply.  
 result := op1  $\times$  op2.

**MUL\_F8 (op1, op2  $\Rightarrow$  result)**

Binary operation: 64 bit floating point multiply.  
 result := op1  $\times$  op2.

**MUL\_I4 (op1, op2  $\Rightarrow$  result)**

Binary operation: 32 bit signed integer multiply.  
 result := op1  $\times$  op2.

**MUL\_I8 (op1, op2  $\Rightarrow$  result)**

Binary operation: 64 bit signed integer multiply.  
 result := op1  $\times$  op2.

**MUL\_I4 (op1, op2  $\Rightarrow$  result)**

Binary operation: 32 bit signed integer multiply.  
 result := op1  $\times$  op2.

**MUL\_U4 (op1, op2  $\Rightarrow$  result)**

Binary operation: 32 bit unsigned integer multiply.  
 $\text{result} := \text{op1} \times \text{op2}$ .

**MUL\_U8 (op1, op2  $\Rightarrow$  result)**

Binary operation: 64 bit unsigned integer multiply.  
 $\text{result} := \text{op1} \times \text{op2}$ .

**MUL\_U4 (op1, op2  $\Rightarrow$  result)**

Binary operation: 32 bit unsigned integer multiply.  
 $\text{result} := \text{op1} \times \text{op2}$ .

**NAME(n,name) ( $\text{---} \Rightarrow \text{---}$ )**

Name the  $n^{\text{th}}$  temporary variable, for debugging purposes.

**NARG\_F4 (val  $\Rightarrow$   $\text{---}$ )**

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

**NARG\_F8 (val  $\Rightarrow$   $\text{---}$ )**

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

**NARG\_I4 (val  $\Rightarrow$   $\text{---}$ )**

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

**NARG\_I8 (val  $\Rightarrow$   $\text{---}$ )**

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

**NARG\_I4 (val  $\Rightarrow$   $\text{---}$ )**

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

**NARG\_P (val  $\Rightarrow$   $\text{---}$ )**

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

**NARG\_U4 (val  $\Rightarrow$   $\text{---}$ )**

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

**NARG\_U8 (val  $\Rightarrow$   $\text{---}$ )**

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.

⌘

**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) (—  $\Rightarrow$  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) (—  $\Rightarrow$  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) (—  $\Rightarrow$  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\_I8(n) (—  $\Rightarrow$  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) (—  $\Rightarrow$  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) (—  $\Rightarrow$  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) (—  $\Rightarrow$  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) (—  $\Rightarrow$  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) (—  $\Rightarrow$  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) (—  $\Rightarrow$  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) (—  $\Rightarrow$  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) (—  $\Rightarrow$  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) (—  $\Rightarrow$  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) (—  $\Rightarrow$  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) (—  $\Rightarrow$  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) (—  $\Rightarrow$  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 (—  $\Rightarrow$  opcode)**

Pushes the current opcode to TOS.

**OR\_I4 (op1, op2  $\Rightarrow$  result)**

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

**OR\_I8 (op1, op2  $\Rightarrow$  result)**

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

**OR\_I4 (op1, op2  $\Rightarrow$  result)**

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

**OR\_U4 (op1, op2  $\Rightarrow$  result)**

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

**OR\_U8 (op1, op2  $\Rightarrow$  result)**

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

96

**OR\_U4 (op1, op2  $\Rightarrow$  result)**

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

**PICK\_F4 ( $\text{---} \Rightarrow \mathbf{n^{\text{th}}}$ )**

*1 operand byte.*  
 Picks the  $\mathbf{n^{\text{th}}}$  item from the data stack(TOS is index 0)and  
 pushes it to TOS.

**PICK\_F8 ( $\text{---} \Rightarrow \mathbf{n^{\text{th}}}$ )**

*1 operand byte.*  
 Picks the  $\mathbf{n^{\text{th}}}$  item from the data stack(TOS is index 0)and  
 pushes it to TOS.

**PICK\_I4 ( $\text{---} \Rightarrow \mathbf{n^{\text{th}}}$ )**

*1 operand byte.*  
 Picks the  $\mathbf{n^{\text{th}}}$  item from the data stack(TOS is index 0)and  
 pushes it to TOS.

**PICK\_I8 ( $\text{---} \Rightarrow \mathbf{n^{\text{th}}}$ )**

*1 operand byte.*  
 Picks the  $\mathbf{n^{\text{th}}}$  item from the data stack(TOS is index 0)and  
 pushes it to TOS.

**PICK\_I4 ( $\text{---} \Rightarrow \mathbf{n^{\text{th}}}$ )**

*1 operand byte.*  
 Picks the  $\mathbf{n^{\text{th}}}$  item from the data stack(TOS is index 0)and  
 pushes it to TOS.

**PICK\_P ( $\text{---} \Rightarrow \mathbf{n^{\text{th}}}$ )**

*1 operand byte.*  
 Picks the  $\mathbf{n^{\text{th}}}$  item from the data stack(TOS is index 0)and  
 pushes it to TOS.

**PICK\_R ( $\text{---} \Rightarrow \mathbf{n^{\text{th}}}$ )**

*1 operand byte.*  
 Picks the  $\mathbf{n^{\text{th}}}$  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  $n^{\text{th}}$  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  $n^{\text{th}}$  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  $n^{\text{th}}$  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  $\Rightarrow$  value)**

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

**PLOAD\_P (addr  $\Rightarrow$  value)**

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

**PLOAD\_R (addr  $\Rightarrow$  value)**

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

**PLOAD\_U1 (addr  $\Rightarrow$  value)**

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

**PLOAD\_U2 (addr  $\Rightarrow$  value)**

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

**PLOAD\_U4 (addr  $\Rightarrow$  value)**

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

**PLOAD\_U8 (addr  $\Rightarrow$  value)**

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

**PLOAD\_U4 (addr  $\Rightarrow$  value)**

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

**POP\_STATE ( $\text{---} \Rightarrow$  value)**

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

**PSTORE\_F4 (value, array  $\Rightarrow$   $\text{---}$ )**

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

**PSTORE\_F8 (value, array  $\Rightarrow$   $\text{---}$ )**

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

**PSTORE\_I1 (value, array  $\Rightarrow$   $\text{---}$ )**

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

**PSTORE\_I2 (value, array  $\Rightarrow$  —)**

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

**PSTORE\_I4 (value, array  $\Rightarrow$  —)**

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

**PSTORE\_I8 (value, array  $\Rightarrow$  —)**

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

**PSTORE\_I4 (value, array  $\Rightarrow$  —)**

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

**PSTORE\_P (value, array  $\Rightarrow$  —)**

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

**PSTORE\_R (value, array  $\Rightarrow$  —)**

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

**PSTORE\_U1 (value, array  $\Rightarrow$  —)**

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

**PSTORE\_U2 (value, array  $\Rightarrow$  —)**

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

**PSTORE\_U4 (value, array  $\Rightarrow$  —)**

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

**PSTORE\_U8 (value, array  $\Rightarrow$  —)**

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

**PSTORE\_U4 (value, array  $\Rightarrow$  —)**

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

**PUSH\_CURRENT\_STATE (—  $\Rightarrow$  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  $\Rightarrow$  —)**

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  $\Rightarrow$  —)**

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

**RETURN\_F8 (value  $\Rightarrow$  —)**

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

**RETURN\_I4 (value  $\Rightarrow$  —)**

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

**RETURN\_I8 (value  $\Rightarrow$  —)**

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

**RETURN\_I4 (value  $\Rightarrow$  —)**

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

**RETURN\_P (value  $\Rightarrow$  —)**

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

**RETURN\_R (value  $\Rightarrow$  —)**

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

**RETURN\_U4 (value  $\Rightarrow$  —)**

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

**RETURN\_U8 (value  $\Rightarrow$  —)**

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

**RETURN\_U4 (value  $\Rightarrow$  —)**

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 $\Rightarrow$ 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 $\Rightarrow$ 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 $\Rightarrow$ 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 $\Rightarrow$ 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 $\Rightarrow$ 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 $\Rightarrow$ result)**

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

### **RSH\_I8 (op1, op2 $\Rightarrow$ result)**

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

### **RSH\_I4 (op1, op2 $\Rightarrow$ result)**

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

### **RSH\_U4 (op1, op2 $\Rightarrow$ result)**

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

### **RSH\_U8 (op1, op2 $\Rightarrow$ result)**

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

### **RSH\_U4 (op1, op2 $\Rightarrow$ result)**

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



**RSTORE\_F4 (value, object, offset  $\Rightarrow$  —)**

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  $\Rightarrow$  —)**

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  $\Rightarrow$  —)**

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  $\Rightarrow$  —)**

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  $\Rightarrow$  —)**

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  $\Rightarrow$  —)**

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  $\Rightarrow$  —)**

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  $\Rightarrow$  —)**

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  $\Rightarrow$  —)**

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  $\Rightarrow$  —)**

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  $\Rightarrow$  —)**

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  $\Rightarrow$  —)**

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  $\Rightarrow$  —)**

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  $\Rightarrow$  —)**

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  $\Rightarrow$  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 (—  $\Rightarrow$  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  $\Rightarrow$  result)**

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

**SUB\_F8 (op1, op2  $\Rightarrow$  result)**

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

**SUB\_I4 (op1, op2  $\Rightarrow$  result)**

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

**SUB\_I8 (op1, op2  $\Rightarrow$  result)**

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

**SUB\_I4 (op1, op2  $\Rightarrow$  result)**

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

**SUB\_P (op1, op2  $\Rightarrow$  result)**

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

**SUB\_U4 (op1, op2  $\Rightarrow$  result)**

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

**SUB\_U8 (op1, op2  $\Rightarrow$  result)**

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

**SUB\_U4 (op1, op2  $\Rightarrow$  result)**

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

**SYMBOL ( $\text{---} \Rightarrow$  address)**

*2 operand bytes.*  
 Push address of symbol to TOS

**TARGET(n) ( $\text{---} \Rightarrow \text{---}$ )**

Target for Jump and Branch.

**TLOAD\_F4(n) ( $\text{---} \Rightarrow$  value)**

Push the contents of the n<sup>th</sup> temporary variable as a 32 bit floating point

**TLOAD\_F8(n) ( $\text{---} \Rightarrow$  value)**

Push the contents of the n<sup>th</sup> temporary variable as a 64 bit floating point

**TLOAD\_I4(n) ( $\text{---} \Rightarrow$  value)**

Push the contents of the n<sup>th</sup> temporary variable as a 32 bit signed integer

**TLOAD\_I4(n) ( $\text{---} \Rightarrow$  value)**

Push the contents of the n<sup>th</sup> temporary variable as a 32 bit signed integer

**TLOAD\_I8(n) ( $\text{---} \Rightarrow$  value)**

Push the contents of the n<sup>th</sup> temporary variable as a 64 bit signed integer

**TLOAD\_P(n)** ( $\text{---} \Rightarrow \text{value}$ )

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

**TLOAD\_R(n)** ( $\text{---} \Rightarrow \text{value}$ )

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

**TLOAD\_U4(n)** ( $\text{---} \Rightarrow \text{value}$ )

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

**TLOAD\_U4(n)** ( $\text{---} \Rightarrow \text{value}$ )

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

**TLOAD\_U8(n)** ( $\text{---} \Rightarrow \text{value}$ )

Push the contents of the  $n^{\text{th}}$  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  $n^{\text{th}}$  temporary variable.

**TSTORE\_F8(n)** ( $\text{value} \Rightarrow \text{---}$ )

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

**TSTORE\_I4(n)** ( $\text{value} \Rightarrow \text{---}$ )

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

**TSTORE\_I4(n)** ( $\text{value} \Rightarrow \text{---}$ )

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

**TSTORE\_I8(n)** ( $\text{value} \Rightarrow \text{---}$ )

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

**TSTORE\_P(n)** ( $\text{value} \Rightarrow \text{---}$ )

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

**TSTORE\_R(n)** (**value**  $\Rightarrow$  —)

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

**TSTORE\_U4(n)** (**value**  $\Rightarrow$  —)

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

**TSTORE\_U4(n)** (**value**  $\Rightarrow$  —)

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

**TSTORE\_U8(n)** (**value**  $\Rightarrow$  —)

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

**TYPE\_NAME(n,name)** (—  $\Rightarrow$  —)

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

**UNLOCK** (**lock**  $\Rightarrow$  —)

Unlock the gvm-t-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** ( $\text{—} \Rightarrow \text{value}$ )

*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 64 bit signed integer.

### **V\_CALL\_I4** ( $\text{—} \Rightarrow \text{value}$ )

*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 signed integer.

### **V\_CALL\_P** ( $\text{—} \Rightarrow \text{value}$ )

*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 pointer.

### **V\_CALL\_R** ( $\text{—} \Rightarrow \text{value}$ )

*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 reference.

### **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\_U8** ( $\text{—} \Rightarrow \text{value}$ )

*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 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.

# Appendix C

## Implementing and Porting the GVMT

### C.1 Front-end Tools

The front-end tools are all C-compiler based. The GVMT abstract machine is a simple target for a compiler as it provides a complete, orthogonal set of arithmetic operators, as well as instructions for reading and writing to memory. The main complexity is the rigid separation of pointers and references. The front-end must be able to determine unambiguously which is which.

The current lcc-based implementation, walks the IR tree twice in order to minimise the number of possible interpretations of the AST. Given that mixing up pointers and references could be fatal, the compiler reports any ambiguities as an error.

### C.2 Mapping the Abstract Machine to the Hardware

The key part to efficiently implementing stack on normal hardware is ‘stack erasure’. Stack erasure is the process of converting some or all of the stack into discrete values.

#### C.2.1 Garbage Collector Interface

In order to use a portable GC written in systems language such C++, the remainder of the Abstract Machine must ensure that all roots to the heap reside in memory when the GC is invoked. Keeping all references, especially those near the top of the stack, would be rather inefficient. By performing liveness analysis on temporary variables, the back-ends can determine which can be freely handled by



GCC and LLVM. Any reference temporary which is live across a gc-safe point, an allocation or any call must be explicitly stored in the control-stack frame.

## C.2.2 The Data Stack

The data stack is where all expressions are evaluated. It is assumed that the majority of stack operations can be removed by stack erasure. The remaining operations require a stack to be implemented in memory.

### The In Memory Stck

The in-memory part of data stack is implemented as an array of values and a stack pointer, `SP`. Each value must be large enough to hold any of the GVMT data types. This means that it must hold 8 byte floats or integers. This is not a problem for 64 bit machines, but wastes up to half the stack space on 32 bits machines. This is acceptable because the size of the in memory part of the data stack is expected to be small.

## C.2.3 The Control Stack

In order that temporary variables can be accessed by the garbage collector, the live reference temporary variables are stored into a frame object. These frames are be linked together, so that the collector can scan each reference. The frame is implemented in C with the following frame:

```
struct _frame {
    struct _frame *previous;
    int count;
    GVMT_Object ref1;
    GVMT_Object ref2;
    ⋮
    GVMT_Object refN;
} frame;
```

The frame is declared as a local (`auto` in C) variable is allocated on the C stack. On entry to the function, the current frame pointer is stored into `frame.previous`. The current frame pointer is the C expression `&frame`, so does not need to explicitly maintained.

## C.2.4 The State Stack

State objects must retain the state of the underlying machine, i.e. its registers, and the state of the GVMT abstract machine, the data-stack and frame-stack pointers. The current implementation uses a single linked list of `gvmt_continuation` structs to implement the state stack.

```

struct gvmt_continuation {
    struct gvmt_registers registers;
    GVMT_StackItem* sp;
    uint8_t* ip; // INTERPRETER ip.
    struct gvmt_continuation* link;
    void *sentinel; // Used for debugging.
};

```

Where the `gvmt_registers` struct holds the machine registers.

When a `PROTECT` instruction is encountered the machine state is saved by pushing a `gvmt_continuation` structs to the state stack. The `gvmt_registers` struct is filled in using a custom assembly routine, as the `setjmp` C library function does not have the correct semantics.

The `gvmt_continuation` structs are cached to prevent excessive allocation and deallocation.

## C.2.5 The Native Parameter Stack

The native parameter stack must be empty at the end of any function or bytecode. It exists only during translation, to track parameters to native functions. It needs no physical representation.

## C.3 The IA-32 Implementation

### C.3.1 Calls and Returns

Use `fastcall` calling convention, which passes two integer or pointer values in registers. Pass `SP` and `FP` in registers, the stack is wholly in memory. `SP` is returned in a register.

This is implemented in GCC as follows:

```

__attribute__((fastcall)) StackItem* func(StackItem* SP, struct _frame* FP) {
    frame.previous = FP;
    frame.count = N;

    Body of function, which may modify SP.

    return SP;
}

```

`N` is the number of reference temporaries live across gc-safe points.

A call to `func` is implemented (in C) as `SP = func(SP, &_frame);`

### C.3.2 Saving Machine State

The IA-32 registers are saved in the follow struct:

```
struct gvm_t_registers {  
    void *eip;  
    void *ebp;  
    void *ebx;  
    void *edi;  
    void *esi;  
    void *esp;  
    void *sentinel; // Used for debugging.  
};
```