Generated 2018-09-08T17:43:26 +0200

VIUA VM MANUAL

OVERVIEW OF CONTENTS

TABLE OF CONTENTS

---

[1] INTRODUCTION

Viua VM is a register based VM (what is a register is explained in section 1.1.1).

[1.1] PROGRAMMING MODEL {programming_model}

A program running on Viua is modelled as a set of communicating processes running in parallel. Each process runs a function, and a function is a sequence of instructions. Every instruction advances the execution of the program by manipulating values placed in registers.

[1.1.1] REGISTERS {registers}

All data in a Viua VM program is held in registers. Viua has no concept of "memory" in the traditional sense (as a byte-addressable array of bytes). This means that a value cannot be stored to, or loaded from, memory.

To cover a wide range of use cases Viua provides several different register sets. These sets vary in lifetime and visibility, and are described in section 1.1.1.2. With this architecture the need for explicit memory should be reduced. If a "store-and-load" functionality is needed it may be emulated using a combination of processes, global and static registers.

[1.1.1.1] A REGISTER

A register is a container for a single value of any type representable by a Viua VM program, either simple (e.g. an integer, a boolean) or compound (e.g. a vector, a struct).

An instruction can access registers to get its inputs and set its outputs. For example, the add instruction gets two inputs (`rhs` and `lhs`) and sets a single output (`output`):

```
add %output local %lhs local %rhs local
```

A register belongs to one of available register sets.

A register may be empty, or full – it never exists in an "uninitialised" state where it contains some unspecified value but is not empty. In Viua a register either contains a valid value, or nothing.

[1.1.1.2] REGISTER SETS {registers:register_sets}

Registers are grouped in register sets. Register sets are allocated per-process (i.e. no two processes may ever share a register set). Processes do not share any registers, and can only exchange values using message passing.

Some register sets are of the general purpose kind: local (described in section 1.1.1.2.1.3), static (described in section 1.1.1.2.1.2), and global (described in section 1.1.1.2.1.1); some are of kinda-sorta everyday use kind: arguments (described in section 1.1.1.2.1.4), and parameters (described in section 1.1.1.2.1.5); and some are special-purpose: message queue (described in section 1.1.1.2.2.1), and exception register (described in section 1.1.1.2.2.2);

There are 7 different register sets available:

```
local                 static                global                arguments
parameters            message-queue         exception-register
```

Three of them contain general purpose, read-write registers. User processes may freely read values from them, and write values to them. These are:

```
local     static    global
```

Two are used for performing function calls:

```
arguments     parameters
```

One is used for interprocess communication...

```
message-queue
```

...and one for error handling:

```
exception-register
```

[1.1.1.2.1] GENERAL PURPOSE REGISTER SETS

A register in a general purpose register set can be used directly by any instruction. An

instruction can freely mix the register sets it uses for its operands. As an example, the following instruction will add an integer from local register 1 to an integer from static register 2 and store the result in global register 3.

```
add %3 global %2 static %1 local
```

This manual assumes a certain style of referring to registers. Whenever "local register N" is written it means "N-th register of the local register set".

A register in a direct access register set is accessed using its index. A register index is an unsigned integer. Valid register indexes start with zero (0) and end with K, where K is the size of the register set to be accessed minus 1. For example, for a local register set with size 16 valid indexes span from 0 to 15.

[1.1.1.2.1.1] GLOBAL REGISTER SET {registers:register_sets:global}

Global register set is allocated per-process. It is accessible from any call frame inside a process. Global register set is shared between all call stacks inside a process.

[1.1.1.2.1.1.1] LIFETIME

Global register set is allocated when a process is spawned, before the process begins execution.

It is deallocated after the process finishes execution (either normally, or due to being killed by a runaway exception). Contents of the global register set are destructed in order, beginning with values in the lowest-indexed registers (i.e. the value in global register 1 is destructed before the value in global register 2, etc.).

[1.1.1.2.1.2] STATIC REGISTER SET {registers:register_sets:static}

Static register set is allocated per-function per-process. It is accessible only for the function that requested it (i.e. function f cannot access static register set of function g). Static register sets are shared between all call stacks inside a process.

[1.1.1.2.1.2.1] LIFETIME

Static register set is allocated when a function first requests it. Currently (2018-03-07), this means that it is allocated on first access to a register from the static register set.

It is deallocated after the process in which it was allocated finishes execution. Contents of a static register set are destructed in order, beginning with values in the lowest-indexed registers (i.e. the value in static register 1 is destructed before the value in static register 2, etc.). It is unspecified (as of 2018-03-07) whether static register sets are deallocated before or after the global register set.

[1.1.1.2.1.2.2] NOTES

Functions are responsible for setting up their static register sets. Since any call to a function may be the first one, access to static registers should be guarded by the isnull instruction (described in section 6.26).

[1.1.1.2.1.3] LOCAL REGISTER SET {registers:register_sets:local}

A local register set is accessible only inside a single call frame. This usually means that it is accessible only for the duration of a single function call, with the exception of local register sets allocated for closures.

Since closures can be called many times their local register sets must persist between calls. This makes them similar to static registers, but scoped only to a single closure instance.

Tail calls (facilitated by the tailcall instruction, described in section 6.43) do not "inherit" the contents of caller's local register set.

[1.1.1.2.1.3.1] LIFETIME

A local register set is allocated whenever:

- a function call is made (using the call instruction, described in section 6.6)
- a process is spawned (using the process instruction, described in section 6.35)
- a deferred call is scheduled (using the defer instruction, described in section 6.12)
- a closure is created (using the closure instruction, described in section 6.10)

Local register set is deallocated before the call frame with which it is associated is dropped (after the return instruction is executed, or during stack unwinding as a result of an uncaught exception), but after all of scheduled deferred calls are executed.
This facilitates simple implementation of deterministic, C++ style destructors. Consider this sample code, as an example:

```
.function: append_log_to_file/2
    .name: iota file_name
    .name: iota log_text
    .name: iota file_output_stream
    .name: iota file_output_stream_ptr

    move %file_name local %0 parameters
    move %log_text local %1 parameters

    frame %1
    move %0 arguments %file_name parameters
    call %file_output_stream local std::posix::io::open/1
    ptr %file_output_stream_ptr local %file_output_stream local

    frame %1
    copy %0 arguments %file_output_stream_ptr local
    defer std::posix::io::close/1

    frame %2
    copy %0 arguments %file_output_stream_ptr local
    move %0 arguments %log_text parameters
    call void std::posix::io::write_newline/2

    ; At this point, no instruction may access any local register set
    ; of the append_log_to_file/2 function.
    ; However, the deferred call to std::posix::io::close/1 may access
    ; the *value* contained in local register named file_output_stream,
    ; because it received the value by pointer - which means that it
    ; does not care where the value lives as it can access it in a
    ; different way than by direct register addressing.
    ;
    ; After the deferred call is executed, the frame is dropped, the
    ; values in it are deleted, and the execution of the rest of the
    ; program continues.
    return
.end
```

[1.1.1.2.1.4] ARGUMENTS REGISTER SET {registers:register_sets:arguments}

Arguments register set is used to pass arguments from the caller to the callee. This is a write-only register set accessible only by move and copy instructions. This register set may only appear in the destination register address, i.e. this is valid code:

```
move %0 arguments %2 local
move %1 arguments %3 local
```

...but this is not:

```
move %2 local %0 arguments
copy %3 local %1 arguments
```

[1.1.1.2.1.4.1] LIFETIME

Arguments register set is allocated by the frame instruction in the caller. It is deallocated by the callee, when the call frame of the callee is dropped.

From the point of view of the caller the register set lives from the moment just after the frame instruction, until the moment just after the next function call instruction

(e.g. call or process):

```
  frame %1                    ; "arguments" register set is allocated here, by the
                              ; caller of foo/1.
  move %0 arguments %1 local  ; Its registers are then filled by either "move" or
                              ; "copy" instructions.
  call void foo/1             ; The "call" instruction marks the point where the
                              ; ownership of the "arguments" register set is
                              ; passed from the caller to the callee.
                              ; After the call is finished, the register set is
                              ; deallocated.
```

Many arguments register sets may be allocated in a single function – one for each function call it makes.

[1.1.1.2.1.5] PARAMETERS REGISTER SET {registers:register_sets:parameters}

Parameters register set is used to hold parameters passed from the calles to the callee. This is a read-only register set accessible only by the move instruction (and may only appear in the source register address).

[1.1.1.2.1.5.1] LIFETIME

The register set is allocated for the callee by the caller. It is deallocated when the call frame for a function is dropped (either as a result of a return, or tail call, or during stack unwinding).

[1.1.1.2.2] SPECIAL-PURPOSE REGISTER SETS

A value placed in a special-purpose register is not directly accessible. It must be moved to (or from) a general purpose register set by a special instruction before it can be used further.

There are 4 instructions that may interact with these special-purpose registers are:

```
  throw      draw       send       receive
```

[1.1.1.2.2.1] MESSAGE QUEUE {registers:register_sets:message_queue}

Message queue is allocated per-process. It is accessible from all call frames, and all stacks of a process (similarly to a global register set).

[1.1.1.2.2.1.1] RECEIVING END

From the point of view of the process owning the message queue it is a read-only register set, accessible only by the receive instruction.

The message queue of a process contains an ordered sequence of the messages that a process received. The following example puts the first value in the queue in local register 1, or throws an exception if no value is available after 1 second:

```
  receive %1 local 1s
```

The size of the message queue is theoretically unbounded. When a message is delievered to a process it is put into its message queue (or: mailbox), increasing the size of the queue. When a message is received by the process it is popped from its mailbox, decreasing mailbox's size.

[1.1.1.2.2.1.2] SENDING END

From the point of view of a process sending a message to a message queue, it is a write-only register set, accessible only by the send instruction.

[1.1.1.2.2.1.3] LIFETIME

FIXME TODO

[1.1.1.2.2.2] EXCEPTION REGISTER {registers:register_sets:exception_register}

Exception register is allocated per-stack per-process. It is accessible from all call frames of a stack.

Exception register is set using the throw instruction. A value can be fetched from the exception register using the draw instruction.

[1.1.1.3] FETCH MODES {programming_model.registers.fetch_modes}

A value is fetched from a register using of three available modes:

- direct
- pointer dereference
- register indirect

The "fetch" name may be misleading in case of the direct mode. Direct fetch mode is also (always) used for specifying output register of an instruction.

[1.1.1.3.1] DIRECT

Direct, or plain, mode is identified by a '%' sign. A value is fetched from a register with the index specified after the '%' sign.

```
integer %2 local 42     ; create a value of type integer (valued 42) in
                        ; local register 2
iinc %2 local           ; increment a value in local register 2 by one
```

[1.1.1.3.2] POINTER DEREFERENCE

Pointer dereference mode is identified by a '*' sign. A value is fetched using a pointer located in register with the index specified after the '*' sign.

```
integer %2 local 42     ; create a value of type integer (valued 42) in
                        ; local register 2
ptr %3 local %2 local   ; create a pointer to an integer in local register 3
                        ; the pointer points to a value in local register 2
iinc *3 local           ; increment a value pointed to by the pointer in
                        ; local register 3
```

Pointers are described in detail in section 4.5.3.

[1.1.1.3.3] REGISTER INDIRECT

Register indirect mode (identified by '@' sign) will fetch a value from a register with index specified by the integer located in register with the index specified after the '@' sign. For example, below code will increment a value in local register 2:

```
integer %2 local 42
integer %3 local 2
iinc @3 local
```

[1.1.1.4] REGISTER ADDRESSING

To fetch or store a value in a register it needs to be addressed. A proper register address consists of a fetch mode, a register index, and a register set. For example:

```
%1 local
*1 local
@1 local

%1 static
*1 static
@1 static

%1 global
*1 global
@1 global
```

are all valid register addresses, conforming to the general syntax:
<fetch-mode> <index> <register-set-specifier>

Mixing fetch modes and register sets in single instructions is allowed. For exampe, this

code:

```
copy %1 static *1 local
```

will copy a value dereferenced from a pointer located in local register 1 into static register 1.

[2] VIUA VM ASSEMBLY LANGUAGE

This section is a reference for Viua VM assembly language. It discusses source code organisation and syntax. Reading this section may allow you to understand the SYNTAX and EXAMPLES sections in documentation for each instruction.

A listing of all instructions can be found in section 6. Instruction set architecture is described in section 5.

[2.1] PROGRAMMING

[2.1.1] CONTROL FLOW

Contents of this section are modelled after Chapter 6 "Control Flow" (Section 6.2 "Structured and Unstructred Flow") of Michael L. Scott's book Programming Language Pragmatics (ISBN 1-55860-442-1).

[2.1.1.1] SEQUENCING

[2.1.1.2] SELECTION

[2.1.1.3] ITERATION

[2.1.1.4] PROCEDURAL ABSTRACTION

[2.1.1.5] RECURSION

[2.1.1.6] CONCURRENCY

[2.1.1.7] NONDETERMINACY

[2.2] DIRECTIVES

[2.2.1] BLOCK

`.block:' directive begins a block definition. Block definitions must end with the `.end' directive.

A detailed description is provided in a later section.

[2.2.2] EXTERN BLOCK

`.extern_block:' declares that a block will be available at link time, even if it is an unresolved symbol in the current compilation unit.

Example:

```
.extern_block: a_block
```

[2.2.3] FUNCTION {assembly:directive:function}

`.function:' directive begins a function definition. Function definitions must end with the `.end' directive.

A detailed description is provided in section 2.3.1.1.

[2.2.4] EXTERN FUNCTION {assembly:directive:extern_function}

`.extern_function:' declares that a function will be available at link time, even if it is unresolved symbol in the current compilation unit.

The following code declares that symbol `foo/2' refers to a function name, and that the symbol will be available at a later time so the assembler should not reject the code:

```
.extern_function: some::module::foo/2
```

The symbol declared using this directive must be made available either during link time (if it is to be linked statically; static linking is described in section 3.1.2.1), or at runtime (if it is to be linked in dynamically; dynamic linking is described in section 3.2.1).

[2.2.5] CLOSURE {assembly:directive:closure}

`.closure:' directive begins a closure definition. Closure definitions must end with the `.end' directive.

[2.2.6] INFO

`.info:' directive allows adding metadata to the produced binary.

[2.2.7] IOTA

`.iota:' directive sets the next value to be used for the `iota' symbol substitution.

[2.2.8] IMPORT

`.import:' directive requests that a module be statically linked to the currently compiled executable.

[2.2.9] MARK

`.mark:' directive sets a name for a point in a block, closure, or function body. This name then may be used instead of an instruction index in branching instructions. Marks are a compile-time-only feature and disappear after the source code has been converted to bytecode. At runtime the VM uses only bytecode indexes, not marks.

NOTE FIXME
The assembler is currently unable to encode markers in bytecode, even as a debugging aid. This is a deficiency that will be fixed in the future.

[2.2.10] NAME

`.name:' directive sets a name for a register. Names are present only during compilation, and disappear after the source code has been converted to bytecode. At runtime the VM uses only indexes, not names.

NOTE FIXME
The assembler is currently unable to encode register names in bytecode, even as a debugging aid. This is a deficiency that will be fixed in the future.

[2.2.11] END {assembly:directive:end}

`.end' directive is used to signal an end of a previously opened directive that requires closing. An example of such a directive is `.function:'.

[2.3] FUNCTIONS {assembly:functions}

This section talks about functions. All code in Viua VM programs is contained in functions, therefore it is incredibly important to understand them.

[2.3.1] DEFINITIONS

This section talks about how to define functions.

[2.3.1.1] DELIMITING FUNCTION BODIES {assembly:blocks:functions:delimiting}

A function definition is delimited by the `.function:' and `.end' tokens. See their documentation in sections 2.2.3 and 2.2.11 respectively.

```
.function: main/0
    text %1 local "Hello World!"
    print %1 local

    izero %0 local
    return
.end
```

This example defines a function that prints "Hello World!" to the screen and returns 0. The first line

```
.function: main/0
```

contains the `.function:' token and the name of the function. In Viua assembly function names must have the following syntax: 1/ a sequence of alphanumeric ASCII characters, colons, and underscores (this is the root part of the name), 2/ a slash character, 3/ the arity of the function (a non-negative integer number).

For example, all following names are valid:

- main/0
- sqrt/1
- write/2
- vector::push_back/2

and all of these are invalid:

- main
  no arity
- foo/-1
  negative arity
- in/valid
  invalid character inside the name
- vector.push_back/2
  invalid character inside the name

The arity part may be empty. Functions with empty arity can be called with variable number of parameters. This is discouraged, however – passing a vector of values is the preferred way to pass variable number of parameters.

The last line

```
.end
```

must appear on its own. Note the lack of a colon at the end of the token.

[2.3.1.2] PARAMETERS

In Viua, functions do not declare the types of parameters they take; they only declare their own arity. The assembler will ensure that once extracted inside the function the parameters are used consistently with regard to the type (thanks to type inference). This means that types are not preserved across function boundaries. This is a deficiency that will be fixed somewhen in the future (FIXME).

[2.3.1.2.1] PASSING PARAMETERS

Parameters are passed into the function using either copy or move instruction with a register in the `arguments` register set as a destination.

[2.3.1.2.2] EXTRACTING ARGUMENTS

Arguments are extracted using the arg instruction.

[2.3.1.2.3] ARGUMENTS REGISTER SET

Parameters passed to a function become its arguments and are stored in a special register

set. The size of this register set depends on the function's arity – if the function takes 2 parameters, then the size of this register set will be 2, if the function takes 1 parameter it will be 1, and if the function takes no parameters it will be 0.

The lifetime of this register set begins just after the frame instruction finishes executing, and ends just after the return instruction causes the call frame containing it to be popped off the stack.

[2.3.1.3] RETURN VALUES

When a function executes the return instruction its call frame is popped off the stack, and the value contained by the local register 0 is automatically used as the return value. Viua allows functions to return only one value, but multiple return values can be emulated by returning a vector of values.

For more information refer to call instruction's documentation in section 6.6. Section 6.6.2.0.3 provides the detailed explanation of how to store return values.

[2.3.2] DECLARATIONS

How to declare that a function will be available during linking, and that the assembler should just trust us that it exists.

[2.4] REGISTER NAMING

How to name registers with human-readable names, e.g. `index' or `size', instead of bare indexes – `1', `4', etc.

[3] TOOLING

This section discusses tools provided in the standard distribution.

[3.1] ASSEMBLER

Source code must be compiled to bytecode before execution on Viua VM kernel. Assembler does the job of converting source files into the executable bytecode files (or libraries).

[3.1.1] COMPILE TIME VERIFICATION

Assembler includes a simple static analyser and type checker.

[3.1.1.1] TYPE CHECKER

The type checker will reject programs in which it can find errors at compile time. For example, the following code will clearly produce an error:

```
text %1 local "Hello World!"
integer %2 local 42
add %3 local %1 local %2 local
```

The third instruction (add) would cause a type error at runtime as it is not possible to add 42 to "Hello World!". The type checker will notice this and reject the program, producing an appropriate error message.

[3.1.1.1.1] TYPE INFERENCE

Viua VM assembly is a language with strong, mostly static typing, and type inference. Not all types have to be explicitly declared: assembler will infer types when they are not expressed in source code from the way the values defined by them are used. In other words – the assembler will enforce consistency of the assumptions about types of values expressed in the source code.

[3.1.1.1.2] TYPES OF FUNCTION PARAMETERS AND RETURNS (FIXME)

Functions in Viua VM assembly (as of 2018-01-17) do not declare types of their parameters (only the number of them), and do not declare the return type either. Type checker is thus unable to ensure type consistency across function boundaries. This deficiency will be

fixed in the future. FIXME

[3.1.1.2] STATIC ANALYSER

Static analyser will simulate how a piece of source code would be executed and check if any errors would be thrown. This mostly means following branches, and verifying that register accesses are correct.

[3.1.2] LINKER {tooling:linker}

Linker cannot be invoked explicitly, and is only usable via the assembler. Due to this, the first file passed as input must always contain source code and not bytecode. For example, this will compile and link correctly:

```
viua-asm main.asm module.viua
```

because the first file contains valid Viua assembly source code.
The following will most likely fail (unless you use non-standard file extensions and store Viua assembly source code in files that do not have the `.asm' extension):

```
viua-asm main.viua module.viua
```

In this case the assembler will most probably print the following error:

```
main.viua:1:1: error: expected a function or a block definition (or signature), or a newl↵
   ine

>>>> 1 VIUAL[illegible]f/0[illegible]Hello, World!d[illegible]
       ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

[3.1.2.1] STATIC LINKING {tooling:linker:static_linking}

Static linking is performed by the assembler as the last part of the compilation process. To link a module statically add it to the list of files the assembler takes as input. For example, the following command will compile `main.asm' module and statically link `modulea.viua' and `moduleb.viua' to it:

```
    viua-asm main.asm modulea.viua moduleb.viua
```

Viua supports both static (compile-time) linkage and dynamic (runtime) linkage. Dynamic linkage is implemented by the kernel, and is discussed in a section 3.2.1.

[3.1.2.1.1] SYMBOL VISIBILITY

All symbols available in a module are visible to all modules that will link it. There is currently no mechanism available to restrict the visibility of the symbols contained by the module.

[3.1.2.1.2] SYMBOL AVAILABILITY

When compiling a file the assembler must be able to resolve all symbols, or it will reject the code. For example, when the assembler encounters the following code:

```
  call void fn/0
```

it will attempt to resolve the symbol `fn/0' and throw an error if it fails to do so. There are several ways to make a symbol available to the assembler, or make it treat the symbol "as if" it was resolvable and to defer the resolving until link-time or runtime (described in section 3.2.1).

[3.1.2.1.2.1] IMPLEMENTING A SYMBOL

The first way to make the symbol resolvable is to implement the function it refers to in the same source code file. Section 2.3 describes how to write a function.

[3.1.2.1.2.2] DECLARING A SYMBOL

To tell the assembler that a symbol will be resolvable at either run- or linking-time

use the `.signature:' directive (described in section 2.2.4). This is most useful when:

- the symbol is in a module that will be linked statically to the current module but the current module is not the main file of the program
- the symbol is in the module that will be linked dynamically and thus will not be available until runtime

The linker will not complain if the declared symbol is not available during link time. If it cannot resolve the symbol, the resolving will be deferred until runtime (described in section 3.2.1).

[3.1.2.1.2.3] ADDING A MODULE CONTAINING THE SYMBOL TO ASSEMBLER'S INPUT FILE LIST

To make the symbol available without declaring it, add the module containing it to the list of modules the assembler should link to the current module.

```
viua-asm main.asm module.viua
```

The above command will automatically make all symbols in `module.viua' available to the code in `main.asm'.

[3.2] KERNEL

Kernel is a program that executes Viua VM bytecode. To run a program compiled into Viua VM bytecode you need both the kernel and the executable bytecode. Then, you can invoke the program like this:

```
viua-vm program.out --some flags
```

The kernel will the begin running the program, starting with the `main' function.

[3.2.1] DYNAMIC LINKING {tooling:kernel:dynamic_linking}

Symbols that were not resolved during compilation, but were declared, are resolved at runtime.

[3.2.1.1] WHEN SYMBOLS ARE RESOLVED

A symbol is resolved the first time it is accessed (which is a questionable design choice, and can lead to the program running for quite some time before the error is detected).

[3.2.1.2] WHEN MODULES ARE LINKED

Modules provide function implementations. To link a module at runtime use the import instruction described in section 6.24.

[3.2.2] ENVIRONMENT VARIABLES

There are environment variables that may be used to enable, disable, or otherwise affect certain kernel features. They are listed and discussed below.

[3.2.2.1] VIUA_ENABLE_TRACING

Setting this variable to `yes' (`VIUA_ENABLE_TRACING=yes') will make the kernel dump execution trace of the running program. Note that the trace is verbose, and a line is dumped to stderr for *every* executed instruction. This means that traces can quickly get long.
However, due to the verbosity and per-executed-instruction granularity they can be a great aid in debugging (of both programs running on the VM, and the VM itself).

```
$ viua-asm sample/asm/text/hello_world.asm
$ viua-vm a.out
Hello World!
$ VIUA_ENABLE_TRACING=yes viua-vm a.out
[ scheduler = 0x80e000000000, process = 0x913000000000, stack = 0x40b000000000, frame↵
    = 0x306000000780, jump_base = 0x605000c150e0, address = 0x605000d15104, depth = 1↵
    ] ress
[ scheduler = 0x80e000000000, process = 0x913000000000, stack = 0x40b000000000, frame↵
```

```
      = 0x306000000780, jump_base = 0x605000c150e0, address = 0x605000d15106, depth = 1↵
      ] frame
  [ scheduler = 0x80e000000000, process = 0x913000000000, stack = 0x40b000000000, frame↵
      = 0x306000000780, jump_base = 0x605000c150e0, address = 0x605000d15113, depth = 1↵
      ] call main/0
  [ scheduler = 0x80e000000000, process = 0x913000000000, stack = 0x40b000000000, frame↵
      = 0x306000003000, jump_base = 0x605000c150e0, address = 0x605000d150e0, depth = 2↵
      ] text
  [ scheduler = 0x80e000000000, process = 0x913000000000, stack = 0x40b000000000, frame↵
      = 0x306000003000, jump_base = 0x605000c150e0, address = 0x605000d150f5, depth = 2↵
      ] print
  Hello World!
  [ scheduler = 0x80e000000000, process = 0x913000000000, stack = 0x40b000000000, frame↵
      = 0x306000003000, jump_base = 0x605000c150e0, address = 0x605000d150fc, depth = 2↵
      ] izero
  [ scheduler = 0x80e000000000, process = 0x913000000000, stack = 0x40b000000000, frame↵
      = 0x306000003000, jump_base = 0x605000c150e0, address = 0x605000d15103, depth = 2↵
      ] return from main/0 with no deferred
  [ scheduler = 0x80e000000000, process = 0x913000000000, stack = 0x40b000000000, frame↵
      = 0x306000003000, jump_base = 0x605000c150e0, address = 0x605000d15103, depth = 2↵
      ] return from main/0 after deferred
  [ scheduler = 0x80e000000000, process = 0x913000000000, stack = 0x40b000000000, frame↵
      = 0x306000000780, jump_base = 0x605000c150e0, address = 0x605000d15121, depth = 1↵
      ] move
  [ scheduler = 0x80e000000000, process = 0x913000000000, stack = 0x40b000000000, frame↵
      = 0x306000000780, jump_base = 0x605000c150e0, address = 0x605000d1512e, depth = 1↵
      ] halt
```

[3.2.2.1.1] TRACE LINE EXPLAINED

Trace line contains several different pieces of information.

```
    [ scheduler = 0x80e000000000, process = 0x913000000000, stack = 0x40b000000000, fra↵
       me = 0x306000000780, jump_base = 0x605000c150e0, address = 0x605000d15104, depth↵
        = 1 ] ress
```

[3.2.2.1.1.1] SCHEDULER

The `scheduler =...' part identifies the scheduler that the process for which this line was emitted is running on. Depending on how many schedulers the VM is running there can be many different schedulers.

Also, a process does not need to run on the same scheduler it was spawned on. If the VM detects that the spawning scheduler is overloaded (i.e. that it runs more than its fair share of all running processes) the process will be migrated to a different scheduler (a one with lighter load).

[3.2.2.1.1.2] PROCESS

The `process =...' part identifies a process for which this line was generated.

[3.2.2.1.1.3] STACK

In Viua VM, a process may contain one active stack, and several suspended stacks. Suspended stacks may be added, for example, by defer instructions (then, the active stack is suspended when a function exits and stacks added by defer instructions are run).

The `stack =...' part identifies the stack that was active in the process when the trace line was emitted.

[3.2.2.1.1.4] FRAME

The `frame =...' part identifies the top-most frame on the stack that was active when the trace line was emitted. It is useful when you want to track a bug that occurs on the frame boundary (just after or before a call or a return).

[3.2.2.1.1.5] JUMP BASE

The `jump_base =...' part identifies a base that is used for resolving jumps during runtime. Jump bases are different for different modules, so you can use this to track module boundaries (from which module came the function that is executed in the current frame).

[3.2.2.1.1.6] ADDRESS

The `address =...' part identifies the address at which the currently executed instruction lives.

[3.2.2.1.1.7] DEPTH

The `depth =...' part indicates the depth of the stack in the moment the trace line was emitted.

[3.2.2.2] VIUA_VP_SCHEDULERS

Configures the number of virtual process schedulers the VM will launch at startup. By default the VM launches 2 virtual process schedulers.

```
$ ./build/bin/vm/kernel --info
0.9.0.969 [sched:ffi=2] [sched:vp=2]
$ VIUA_VP_SCHEDULERS=4 ./build/bin/vm/kernel --info
0.9.0.969 [sched:ffi=2] [sched:vp=4]
```

Increase the number of virtual process schedulers if your programs are CPU-intensive, and are mostly pure bytecode.

[3.2.2.3] VIUA_FFI_SCHEDULERS

Configures the number of FFI schedulers the VM will launch at startup. By default the VM launches 2 FFI schedulers.

```
$ ./build/bin/vm/kernel --info
0.9.0.969 [sched:ffi=2] [sched:vp=2]
$ VIUA_FFI_SCHEDULERS=4 ./build/bin/vm/kernel --info
0.9.0.969 [sched:ffi=4] [sched:vp=2]
```

Increase the number of FFI schedulers if your programs issue many FFI calls from many virtual processes. A FFI call blocks only the process that issued it and an FFI scheduler executes only a single call at once, so if you increase the number of schedulers you will be able to run more calls simultaneously.

[3.2.2.4] VIUAPRELINK [DEPRECATED]

A colon-separated list of bytecode modules to link before the kernel starts executing the `main' function.
This environment variable is DEPRECATED.

[3.2.2.5] VIUAPREIMPORT [DEPRECATED]

A colon-separated list of FFI modules to link before the kernel starts executing the `main' function.
This environment variable is DEPRECATED.

[3.3] DISASSEMBLER

Disassembler may be used to convert compiled bytecode into assembly source code.

It has little chance of producing exactly the source code that was used to compile the bytecode binary. The semantics *will* be exactly the same, but the disassembler will not be able to reproduce:

- register names
- attributes
- closures

[3.3.1] DISASSEMBLING CLOSURES (FIXME)

Closures will be disassembled as ordinary functions which will most likely cause them to be rejected if you tried to compile the disassembled code, as the static analyser would see that the function tries to access undefined registers – these registers will not be undefined at runtime, and with proper annotations (that are supplied by the programmer, by using `.closure:' instead of `.function:') the static analyser would accept them, but the information what function is a closure is lost during compilation.

The fact that the disassembler is unable to extract this information is a deficiency of the assembler which does not encode it in the binaries it produces.

[3.3.2] DISASSEMBLING METADATA (FIXME)

Metadata such as register names, attributes, etc. that is present in the original source code as needed for compilation is lost after the compilation is done and is not included in the resulting binaries. The reason for this is that such information is useless at runtime.

However, it may be useful during debugging or disassembling. The fact that the assembler does not encode such debugging information in binaries is a deficiency. It should be possible to request inclusion of debugging information in the resulting binary by using a flag (e.g. `-g') during compilation.

[4] THE ENVIRONMENT

This section describes the processes and procedures followed by the Viua VM assembly when it executes programs. It also describes program organisation from the VM's point of view, and talks about mechanisms that it makes available to programmers.

[4.1] RUNTIME

[4.2] PROGRAM ORGANISATION

Programs in Viua VM are organised into concurrently running processes. Each process runs a function.

Programs being execution from the main function, and with a single process.

[4.2.1] MAIN FUNCTION

There are several flavours of the main function. Just as C++ defines

```
int main();
int main(int args, char* argv[]);
```

as possible entry points, Viua VM defines

```
.function: main/0
.function: main/1
.function: main/2
```

Main function is required to return an integer exit code.

The `main/1' main function receives one argument: a vector of strings representing the parameters the program received on the command line with name with which the program was invoked inserted as the first entry. The `main/2' main function receives two arguments: 1/ a string containing the name of the program as it was called (which would be `argv[0]' in C++), and 2/ a vector of strings representing the parameters the program received on the command line.

Vectors are described in section 4.5.2.1, and strings in section 4.5.1.3.

[4.3] PROCESSES

[4.3.1] INITIALISATION {environment:processes:initialisation}

[4.3.2] EXECUTION

[4.3.2.1] NORMAL CALLS

Using the call instruction described in section 6.6.

[4.3.2.2] DEFERRED CALLS

Using the defer instruction described in section 6.12.

[4.3.2.3] SPAWNING NEW PROCESSES

Using the process instruction described in section 6.35.

[4.3.2.4] THE CALL STACK

[4.3.2.4.1] MULTIPLE STACKS

[4.3.3] INTERRUPTION {environment:processes:interruption}

A process may be interrupted by another process. To interrupt a process you must have access to its PID.

Interrupts are delievered asynchronously: the process that interrupts another one does not block until the interrupt is delievered. This is consistent with how messages are delievered.

An interrupt is handled by the receiving process as soon as it is delievered. This is in contrast with how messages are handled; when a message arrives it is put on a queue and the receiving process decides when to handle it, but when an interrupt arrives it must be handled immediately.

When an interrupt arrives two things can happen:
- if the process has registered a handler for the type of interrupt that arrived, a new stack is created for the handler, and the process switches execution to it (the interrupt is "serviced")
- if the process has not registered a handler for the type of interrupt that arrives, the process is cancelled (cancellation is described in section 4.3.3.3)

After the interrupt is serviced the process resumes executing the stack it was executing when the interrupt was delievered.

[4.3.3.1] INTERRUPT DELIVERY POINTS {environment:processes:interrupts:delivery_points}

Interrupts are delievered at well-defined points during program execution.

When a process is normally executing they are delievered between instructions. Every instruction is an atomic operation from the point of view of an interrupt.

When a process is blocked awaiting a message the wait is aborted, and process switches execution to the interrupt handler.

When a process is blocked awaiting I/O completion the wait is aborted, and process switches execution to the interrupt handler. If the I/O request has already begun processing then the operation is not aborted, and continues asynchronously. If the I/O request has been queued but did not begin processing then the operation is cancelled.

When a process is blocked by a foreign function call the call is not cancelled. The VM has no way of interrupting a foreign call while maintaining clean and correct semantics. The foreign call proceeds as if nothing happened, and its result is delivered normally. The process switches execution to the interrupt handler.

[4.3.3.2] ERRORS IN INTERRUPT HANDLERS {environment:processes:interrupts:errored}

If an exception escapes the stack of the interrupt handler the shutdown of the process is triggered. Shutdowns are described in section 4.3.4.

[4.3.3.3] CANCELLATION {environment:processes:cancellation}

Cancellation causes an immediate shutdown of a process. Shutdowns are described in section

4.3.4.

[4.3.4] SHUTDOWN {environment:processes:shutdown}

[4.3.5] ERROR HANDLING

[4.3.5.1] EXCEPTIONS {environment.processes.error_handling.exceptions}

[4.3.5.1.1] THROWING

[4.3.5.1.2] CATCHING

[4.3.5.1.3] STACK UNWINDING
{environment:processes:error_handling:exceptions:stack_unwinding}

[4.3.5.2] WATCHDOG

[4.3.6] COMMUNICATION

[4.3.6.1] MESSAGE PASSING {environment:processes:communication:message_passing}

[4.3.6.2] SIGNALS

Signals are not yet implemented.

[4.3.6.3] INTERRUPTS

Interrupts are not yet implemented.

[4.4] COMMUNICATION WITH THE OUTSIDE WORLD

[4.4.1] I/O

[4.4.1.1] STANDARD STREAMS

[4.4.1.1.1] STANDARD INPUT

[4.4.1.1.2] STANDARD OUTPUT

[4.4.1.1.3] STANDARD ERROR

[4.4.1.2] FILE I/O

[4.4.1.3] NETWORK I/O

[4.4.1.4] GENERIC I/O

[4.5] DATA TYPES {environment:data_types}

Viua provides several built-in data types. User-defined ones can be built on top of them.

[4.5.1] PRIMITIVE TYPES

[4.5.1.1] NUMERIC TYPES {environment:data_types:numeric}

Viua provides two basic numeric types: integers, and floating point numbers. Both these types are just thin wrappers around the platform-provided types. This means that they are quite efficient ("quite" because we still have to factor in the overhead of the VM), but not reliable - their semantics are not enforced by the VM so the behaviour of the program using them may vary depending on which platform it runs.

For floating point numbers there is no escape from platform-provided types (as of 03-01-2018), but for integers there is - Viua provides a bits type (described in section 4.5.1.2) and several instructions that operate on its bit patterns with well-defined semantics, implementing wrapping, saturating, and checked integer arithmetic for signed and unsigned integers.

To manipulate values of numeric types use instructions from the "arithmetic" group:

- add (6.1)
- sub (6.42)
- mul (6.33)
- div (6.13)

Values of numeric types can be compared using the following instructions:

- lt (6.30)
- lte (6.31)
- gt (6.20)
- gte (6.21)
- eq (6.33)

All of these instructions work on any combination of numeric types: the right-hand side operand's value is converted to the type of the left-hand side's operand before comparison.

[4.5.1.1.1] INTEGER {environment:data_types:numeric:integer}

Integer values are 64 bit wide, signed integers. They are implemented using primitive type provided by the platform so their semantics are not well-defined in the VM but they are fast. For integers with well-defined semantics see section 4.5.1.2.

To construct an integer value use izero (described in section 6.28), or integer (described in 6.25). To manipulate integer values use:

- iinc (6.23)
- idec (6.22)
- idec (6.22)

and instructions from the "arithmetic" group.

[4.5.1.1.2] FLOATING POINT {environment:data_types:numeric:floating_points}

Floating point values are 64 bit floating pointer numbers. They are implemented using platform-provided primitive type.

To construct a floating point value use float (described in section 6.16). To manipulate floating point values use instructions from the "arithmetic" group.

[4.5.1.2] BITS {environment:data_types:bits}

Bits provide the expected bit operators (shifts, rotates, and logic operations), and implement arbitrary precision signed- and unsigned integers.

On its own, the bits type is just that - a string of bits. However, there are instructions that treat bits values as integers and which implement saturating, checked, and wrapping arithmetic operations (for both signed and unsigned integers). Signed integers are encoded using two's complement representation.

[4.5.1.3] STRING {environment:data_types:string}

A string is just a string of bytes, without any particular structure. It can be used to represent ASCII text, but this usage is not recommended (there is a text type for representing text, described in section 4.5.1.4).

[4.5.1.4] TEXT {environment:data_types:text}

Text is a data type designed to hold text values. It is a sequence of UTF-8 encoded Unicode codepoints.

[4.5.1.5] ATOM {environment:data_types:atom}

An Atom is a value that represents itself. See description of atom instruction in section 6.4.

[4.5.2] AGGREGATE TYPES

[4.5.2.1] VECTOR {environment:data_types:vector}

A vector is a collection of values, indexed by a non-negative integer. Vectors can be nested.

Currently (as of 2018-02-01) vectors in Viua VM can have at most 2^64 items.

[4.5.2.2] STRUCT {environment:data_types:struct}

[4.5.3] POINTERS {environment:data_types:pointers}

Pointers can be created using the ptr instruction (see section 6.36).

[4.5.3.1] VALUE TRACKING

Pointers in Viua point to values, not to locations. This is in stark contrast to pointers in C or C++, where they point to memory locations. In other words, pointers in Viua "track" the values they point to: when a value is moved to a different register (even if it is in a different register set), or moved to a closure or a deferred function the pointer still points to that value.

```
integer %1 local 42     ; Create a value of type integer (valued 42) in
                        ; local register 1.

ptr %2 local %1 local   ; Create a pointer to a value in local register 1 in
                        ; local register 2.

iinc *2 local           ; Increment the value pointed to by the pointer in
                        ; local register 2. After this instruction local register 1
                        ; contains 43.

move %3 local %1 local  ; Move the value from local register 1 to local register 3.
                        ; Local register 1 becomes empty.

iinc *2 local           ; Increment the value pointed to by the pointer in
                        ; local register 2. After this instruction local register 3
                        ; contains 43.
```

[4.5.3.2] EXPIRATION

Another feature provided by pointers is that they expire. When the value a pointer points to is deleted the pointer is marked as "expired" (no longer valid), and any access through that pointer results in an exception being thrown by the VM. You can still write programs that will have dangling pointers, but you can recover if you accidentally try to use such a pointer to access a value. Moving (or otherwise manipulating) expired pointers does not throw an exception, only using the pointer to access a value does.

```
integer %1 local 42     ; Create a value of type integer (valued 42) in
                        ; local register 1.

ptr %2 local %1 local   ; Create a pointer to a value in local register 1 in
                        ; local register 2.

iinc *2 local           ; Increment the value pointed to by the pointer in
                        ; local register 2. After this instruction local register 1
                        ; contains 43.
```

```
delete %1 local         ; Delete a value in local register 1.
                        ; Local register 1 becomes empty.

iinc *2 local           ; This instruction throws an exception because the pointer it
                        ; uses to access the value has expired.
```

[4.5.3.3] INITIALISATION AND RESEATING

Pointers cannot be created uninitialised. A pointer is constructed from a value, and trying to construct it from an empty register will throw an exception.

```
ptr %2 local %1 local   ; Try to create a pointer to a value in local register 1 in
                        ; local register 2. This instruction throws an exception
                        ; because it is not legal to create a pointer that does not
                        ; point to a value.
```

Pointers cannot be "reseated". Once a pointer is constructed and starts pointing to some value it cannot be made to point to a different one, the binding established at construction time is immutable. This also means that pointers cannot be used to create cycles – since a value must exist before a pointer is taken to it, a pointer can only point to things that were constructed before the pointer itself; and since a pointer cannot be reseated it cannot be made to point to a value that was created after it.

No example is provided for a reseating attempt since Viua VM assembly languages lacks the way to express such a construction.

[4.5.3.4] NESTING

Pointers to pointers can be arbitrarily nested. Creating a pointer nested to n-th level requires using n ptr instructions. Dereferencing a pointer nested to n-th level also requires using n instructions. Only one level of nesting can be added or subtracted in one step.

```
integer %1 local 42     ; Create a value of type integer (valued 42) in
                        ; local register 1.

ptr %2 local %1 local   ; Try to create a pointer to a value in local register 1 in
                        ; local register 2. This instruction throws an exception
                        ; because it is not legal to create a pointer that does not
                        ; point to a value.

ptr %3 local %2 local   ; Create a pointer to a value in local register 2 in
                        ; local register 3. The pointer created in the local register 3
                        ; is a "pointer to a pointer to an integer".

move %4 local *3 local  ; Move a value dereferenced from the pointer in local register 3
                        ; to local register 4. Moving a dereferenced value copies the
                        ; value since there is no way to actually perform the move.

iinc *4 local           ; Increment the value pointed to by the pointer in
                        ; local register 3. After this instruction local register 1
                        ; contains 43.
```

[4.6] STANDARD LIBRARY

Viua VM provides a standard library. The library provides basic functionality that may be useful to a wide array of different programs and systems.

[5] INSTRUCTION SET ARCHITECTURE {isa}

Instructions in Viua vary in complexity from very simple (e.g. adding two integers) to complex and expensive ones (e.g. performing signed saturating multiplication).

[5.1] INSTRUCTION CATEGORIES

Instructions provided by Viua VM can be split into the general categories described below. This description is purely theoretical and does not manifest in the encoding of bytecode or execution of instructions.

[5.1.1] DATA MOVEMENT

This category includes instructions that move data between registers in the same process as well as instructions that move values across process boundaries (using message passing described in section 4.3.6.1).

[5.1.2] COMPUTATION

This category includes instructions that perform computations – those that "do stuff" to put it in less fancy terms, and because "computation" here is meant as a more general concept than simply arithmetic. For example, modifying text values or pushing new items onto a vector is also thrown under the "computation" umbrella.

[5.1.3] CONTROL TRANSFER

This category includes instructions that alter the control flow of the program. Alteration of the control flow includes performing jumps, calling functions, spawning processes, and throwing exceptions.

[5.2] OPERANDS

Operands can be supplied to Viua VM instructions either in registers or as immediates.

[5.2.1] REGISTER OPERANDS

[5.2.2] IMMEDIATES

Immediate operands come in several flavours. Some are symbols (e.g. function names to call), some are fixed-width data (e.g. integers and addresses), and some are variable-width data (e.g. text).

[5.2.2.1] SYMBOLS

Symbol operands are used with control-flow instructions, and encode function names. They are variable-width immediates, but are assigned given their own category for conceptual reasons – a piece of text is something entirely different (on the conceptual level, even if the encoding is largely similar) than a function name.

Symbol operands are also used in constructions of function (in section 6.19), closure (in section 6.10), and atom (in section 6.4) instructions.

[5.2.2.2] FIXED-WIDTH DATA

[5.2.2.3] VARIABLE-WIDTH DATA

The decision to include variable-width immediates as part of the instruction may seem a dubious design choice, but it works (as of 2018-02-05) and there are no plans to change it.

---

[6] INSTRUCTIONS {isa:instructions}

There are 137 instructions in the instruction set currently implemented by the VM:

add	allocate_registers	and
arg	argc	ashl
ashr	atom	atomeq
bitaeq	bitagt	bitagte
bitalt	bitalte	bitand
bitat	bitnot	bitor
bits	bitseq	bitset
bitsgt	bitsgte	bitslt
bitslte	bitswidth	bitxor
bool	call	capture
capturecopy	capturemove	catch
checkedsadd	checkedsdecrement	checkedsdiv
checkedsincrement	checkedsmul	checkedssub

```
  checkeduadd             checkedudecrement       checkedudiv
  checkeduincrement       checkedumul             checkedusub
  closure                 copy                    defer
  delete                  div                     draw
  echo                    enter                   eq
  float                   frame                   ftoi
  function                gt                      gte
  halt                    idec                    if
  iinc                    import                  integer
  isnull                  itof                    izero
  join                    jump                    leave
  lt                      lte                     move
  mul                     nop                     not
  or                      print                   process
  ptr                     ptrlive                 receive
  return                  rol                     ror
  saturatingsadd          saturatingsdecrement    saturatingsdiv
  saturatingsincrement    saturatingsmul          saturatingssub
  saturatinguadd          saturatingudecrement    saturatingudiv
  saturatinguincrement    saturatingumul          saturatingusub
  self                    send                    shl
  shr                     stof                    stoi
  streq                   string                  struct
  structinsert            structkeys              structremove
  sub                     swap                    tailcall
  text                    textat                  textcommonprefix
  textcommonsuffix        textconcat              texteq
  textlength              textsub                 throw
  try                     vat                     vector
  vinsert                 vlen                    vpop
  vpush                   watchdog                wrapadd
  wrapdecrement           wrapdiv                 wrapincrement
  wrapmul                 wrapsub
```

This section contains a reference for every one of them, with documented inputs, outputs, and exceptions.

It is loosely modelled after Appendix A to William Hohl's 2009 book "ARM Assembly Language: Fundamentals and Techniques" (ISBN 9781439806104).

[6.1] ADD {opcode:add}
    in group: arithmetic

  [6.1.1] SYNTAX

    (0)   add %output local %lhs local %rhs local

  [6.1.2] DESCRIPTION

    Adds two numerical values (either a float, or an integer).

    The result has the type of the left-hand side value. For example, if the left-hand side is an integer value the result will also be an integer value;
    if the left-hand is a float value the result will also be a float value.

  [6.1.3] EXCEPTIONS
    None.

  [6.1.4] EXAMPLES
    None.

  [6.1.5] REMARKS
    None.

[6.2] ALLOCATE_REGISTERS {opcode:allocate_registers}
    in group: register-management

  [6.2.1] SYNTAX

    (0)   allocate_registers %N local

  [6.2.2] DESCRIPTION

Allocates a set of N local registers. It is a compile-time error to access a register outside of allocated range.

This should always be the first instruction to appear in a normal function. Closures should not use this instruction, though, as their register set is externally allocated. It is a compile-time error to put this instruction as a not-first instruction in a function body.

[6.2.3] EXCEPTIONS
None.

[6.2.4] EXAMPLES
None.

[6.2.5] REMARKS
None.

[6.3] ARG {opcode:arg}
in group: arg

[6.3.1] SYNTAX

(0) arg %1 local %0

[6.3.2] DESCRIPTION

This instruction is used to extract arguments passed to the function and saved in the call frame. The second operand is the index of the argument saved in the call frame, and the first operand is the register index to save.
For example, the following code will extract the first argument and put it in local register 1:

```
arg %1 local %0
```

[6.3.3] EXCEPTIONS

[6.3.3.1] ARGUMENT REGISTER OUT OF BOUNDS

Thrown when the second operand of arg instruction references an index that is out of bounds for the current call frame. For example, indexes 2 and higher would cause this exception to be thrown on frames with two arguments saved (or less); remember that register indexes are counted from 0.

```
.function: main/0
    arg %1 local %0  ; This instruction will cause an exception to be thrown.

    izero %0 local
    return
.end
```

The above code would fail as the `main/0' function receives no arguments, and yet it would try to access some in its body.

[6.3.4] EXAMPLES
None.

[6.3.5] REMARKS
None.

[6.4] ATOM {opcode:atom}
in group: atom

[6.4.1] SYNTAX

(0) atom %1 local 'an_atom'

[6.4.2] DESCRIPTION

Atoms are values that stand for themselves. They are unique in the way that no two atoms are the same; however, there may be several instances of the same atom. As an example, there may

be two atoms with the value 'foo' and they will be indistinguishable from each other, but they will be radically different from a third atom with the value 'fooo'.

The only operation that atoms support is comparison for equality, using the atomeq instruction described in section 6.5.

[6.4.3] EXCEPTIONS
None.

[6.4.4] EXAMPLES
None.

[6.4.5] REMARKS
None.

[6.5] ATOMEQ {opcode:atomeq}
in group: atom

[6.5.1] SYNTAX

```
(0)    atomeq %Result local %Lhs local %Rhs local
```

[6.5.2] DESCRIPTION

Compare two atoms for equality.

[6.5.3] EXCEPTIONS
None.

[6.5.4] EXAMPLES
None.

[6.5.5] REMARKS
None.

[6.6] CALL {opcode:call}
in group: calls

[6.6.1] SYNTAX

```
(0)    call void f/0
(1)    call %0 local f/0
(2)    call void %1 local
(3)    call %0 local %1 local
```

[6.6.2] DESCRIPTION

Calls a function.

[6.6.2.0.1] STATICALLY DISPATCHED CALLS

The first syntax will call a function f/0 and discard its return value. The second syntax will call a function f/0 and store its return value in local register 0.

[6.6.2.0.2] DYNAMICALLY DISPATCHED CALLS

The third syntax will call a function that is referenced by the value stored in local register 1 and discard the return value of that call.
The fourth syntax will call a function that is referenced by the value stored in local register 1 and store the return value of that call in local register 0.

A function can be "stored" in a register using the function instruction described in section 6.19.

[6.6.2.0.3] STORING RETURN VALUES {storing_return_values}

To save the return value of a call use a register instead of void as the target operand of

the call instruction. For example:

  call void foo/0

will not save foo/0's return value, but

  call %2 local foo/0

will save it in local register 2.

Syntaxes `call void f/0' (0) and `call void %1 local' (2) discard the return value, and `call %0 local f/0' (1) and `call %0 local %1 local' (3) save it.

[6.6.3] EXCEPTIONS
None.

[6.6.4] EXAMPLES
None.

[6.6.5] REMARKS
None.

[6.7] CAPTURE {opcode:capture}
in group: closure

[6.7.1] SYNTAX

(0) capture %TargetClosure local %InClosureRegister %CapturedValue local

[6.7.2] DESCRIPTION

Capture a value by reference.

[6.7.3] EXCEPTIONS
None.

[6.7.4] EXAMPLES
None.

[6.7.5] REMARKS
None.

[6.8] CAPTURECOPY {opcode:capturecopy}
in group: closure

[6.8.1] SYNTAX

(0) capturecopy %TargetClosure local %InClosureRegister %CapturedValue local

[6.8.2] DESCRIPTION

Capture a value by copy.

[6.8.3] EXCEPTIONS
None.

[6.8.4] EXAMPLES
None.

[6.8.5] REMARKS
None.

[6.9] CAPTUREMOVE {opcode:capturemove}
in group: closure

[6.9.1] SYNTAX

```
    (0)    capturemove %TargetClosure local %InClosureRegister %CapturedValue local
```

[6.9.2] DESCRIPTION

Capture a value by move.

[6.9.3] EXCEPTIONS
None.

[6.9.4] EXAMPLES
None.

[6.9.5] REMARKS
None.

[6.10] CLOSURE {opcode:closure}
in group: functional

[6.10.1] SYNTAX

```
(0)    closure %0 local f/0
```

[6.10.2] DESCRIPTION

A closure is a function which bound some pieces of its environment.

In higher-level languages closures are usually created implicitly, when a function is defined inside another function and references some of the enclosing function's variables. In Viua VM assembly the process is explicit, as it requires the programmer to write a standalone function (which involves giving it a name), construct a closure from the function, capture required values in the closure. Every listed step requires an instruction, adding to the verbosity.

What is more, the functions that will be used to construct closures are specially marked – they begin with the `.closure:' directive (described in section 2.2.5).

Closure bodies are not statically analysed until the closure is actually constructed. This is because closures are defined with obvious "holes" (empty registers to be filled with captured values, but which are used in the closure body) whose values are not known until the closure constructor instruction is not executed. The static analyser will verify type correctness of each closure instantiation at points where closure instructions are executed.

[6.10.2.1] CAPTURING VALUES

Closure construction process, while explict, is very flexible. It allows capturing values by:

- copy – copying them into the closure
- reference – making the closure and its environment share the value
- move – moving values from the enclosing environment inside the closure

Due to the presence of by-copy and by-move captures and the fact that closures must be explicitly constructed the closure mechanism in Viua can be used to create specialised functions (for example, there can be a `min' function that is constructed with different values to create different "minimisers").

[6.10.2.1.1] BY REFERENCE

By reference captures are made using capture instruction (section 6.7). Captured value is shared between the closure and the enclosing function. Changes made be any of them are visible to the other.

This capture mode is provided to allow creating closures in the usual sense, but its use is not encouraged. It is better to use by copy or by move captures as they avoid sharing values between two different functions.

[6.10.2.1.2] BY COPY

By copy captures are made using capturecopy instruction (section 6.8). The closure gets a

fresh copy of the captured value. Enclosing function retains captured value.

[6.10.2.1.3] BY MOVE

By move captures are made using capturemove instruction (section 6.9). The closure becomes the owner of the value, and the enclosing function loses access to it.

[6.10.3] EXCEPTIONS
None.

[6.10.4] EXAMPLES
None.

[6.10.5] REMARKS
None.

[6.11] COPY {opcode:copy}
in groups: registers, calls

[6.11.1] SYNTAX

(0) copy {dst:register_access} {src:register_access}

[6.11.2] DESCRIPTION

Moves a value from one register (src) to another (dst). These registers may be in different register sets.

The source register must not be empty when this instruction is issued.

The copy of the value is made before the value in destination register is deleted. This ensures that in case of failure, the instruction will modify less state.

[6.11.2.1] PASSING ARGUMENTS

A special form of this instruction (where the destination register set is "arguments") is used to pass arguments by copy to a function:

```
text %1 local "Hello World!"
copy %0 arguments %1 local      ; pass the argument by copy
```

The arguments register set is described in section 1.1.1.2.1.4.

[6.11.3] EXCEPTIONS
None.

[6.11.4] EXAMPLES
None.

[6.11.5] REMARKS
None.

[6.12] DEFER {opcode:defer}
in group: calls

[6.12.1] SYNTAX

(0) defer function_name/0

[6.12.2] DESCRIPTION

The defer instruction schedules a call to be executed just *after* the frame in which this instruction is executed is returned from, but before the execution of the enclosing frame resumes.

For example, if a deferred call to function `foo' is scheduled then the sequence of events is as follows:

- execute a `defer foo/0' instruction
- schedule a call to `foo/0'
- encounter `return' instruction
- set return value in enclosing frame
- suspend current stack (let's call it S)
- execute all deferred calls (the process will execute all the deferred functions to completion, with the last deferred call being executed first)
- resume stack S
- pop the frame for `bar/0' function

Note that deferred calls can also contain defer instructions and schedule their own deferred calls. There is no limit on the nesting of deferred calls.

[6.12.3] EXCEPTIONS
None.

[6.12.4] EXAMPLES

```
.function: foo/0
    frame %0
    defer bar/0
    print (text %iota local "0. before deferred") local
    return
.end

.function: bar/0
    print (text %iota local "1. in deferred") local
    return
.end

.function: main/0
    frame %0
    call void foo/0
    print (text %iota local "2. after deferred") local

    izero %0 local
    return
.end
```

[6.12.5] REMARKS

This instruction may be used to implement a RAII approximation. You can create a value that needs to be finalised somehow (e.g. a socket needs to be closed) you can spawn the value, obtain a pointer to it, defer a call to a closing function and pass the value as a parameter by move.

The value is still available because the deferred call is scheduled after the current function ends so the pointer you leave for yourself is still valid.

Also, this ensures exception safety – deferred functions are called even if the frame is popped during stack unwinding after an exception is thrown.

[6.13] DIV {opcode:div}
in group: arithmetic

[6.13.1] SYNTAX

(0) div %output local %lhs local %rhs local

[6.13.2] DESCRIPTION

Divides two numerical values (either a float, or an integer). It models the following operation:

```
output = lhs / rhs
```

The result has the type of the left-hand side value. For example, if the left-hand side is an integer value the result will also be an integer value;
if the left-hand is a float value the result will also be a float value.

[6.13.3] EXCEPTIONS
None.

[6.13.4] EXAMPLES
None.

[6.13.5] REMARKS
None.

[6.14] DRAW {opcode:draw}
in groups: exceptions, error-handling

[6.14.1] SYNTAX

```
(0)    draw %target local
(1)    draw void
```

[6.14.2] DESCRIPTION

Draws caught value from exception register into a register in one of the main register sets.

The `draw void' discards caught value without fetching it into a register.

[6.14.3] EXCEPTIONS
NO CAUGHT OBJECT

No exception was caught. This exception is thrown when the draw instruction is executed but there is no value it could fetch.

[6.14.4] EXAMPLES
None.

[6.14.5] REMARKS
None.

[6.15] EQ {opcode:eq}
in groups: logic, arithmetic

[6.15.1] SYNTAX

```
(0)    eq %output local %lhs local %rhs local
```

[6.15.2] DESCRIPTION

Compares two numerical values (either a float, or an integer). If the left-hand side operand is an integer compares them as integers; if the left-hand side operand is a float compares them as floats.

The result is the value of the following operation:

```
result = lhs == rhs
```

[6.15.3] EXCEPTIONS
None.

[6.15.4] EXAMPLES
None.

[6.15.5] REMARKS
None.

[6.16] FLOAT {opcode:float}
in groups: ctors, float

[6.16.1] SYNTAX

```
(0)    float %1 local 0.0
(1)    float %1 local 3.14
(2)    float %1 local -3.14
```

(3) float %1 local default

[6.16.2] DESCRIPTION

Stores a floating point value in a specified register.
The `float %1 local default' syntax stores a 0.

[6.16.3] EXCEPTIONS
None.

[6.16.4] EXAMPLES
None.

[6.16.5] REMARKS
None.

[6.17] FRAME {opcode:frame}
in group: calls

[6.17.1] SYNTAX

(0) frame %0 %0

[6.17.2] DESCRIPTION

Spawns a new call frame and puts it in the "new call frame" slot. Spawned call frame contains two register sets: arguments register set and local register set.

Size of the arguments register set is specified by the first operand. Size of the local register set is specified by the second operand.

This instruction *MUST* be executed before any copy or move instructions with a register in "arguments" register set are executed.

[6.17.3] EXCEPTIONS
None.

[6.17.4] EXAMPLES
None.

[6.17.5] REMARKS
None.

[6.18] FTOI {opcode:ftoi}
in groups: cast, float, integer

[6.18.1] SYNTAX

(0) ftoi %output local %input local

[6.18.2] DESCRIPTION

Converts a float value in input register to an integer value, and puts result in output register.

[6.18.3] EXCEPTIONS
None.

[6.18.4] EXAMPLES
None.

[6.18.5] REMARKS

The conversion is done between native types of the CPU the VM is running on. Some precission loss may happen.

[6.19] FUNCTION {opcode:function}

in group: functional

[6.19.1] SYNTAX

(0) function %0 local f/0

[6.19.2] DESCRIPTION

Stores a function reference in a register.

Use this instruction to "store" a function in a register. The value stored can then be used in call instructions to decide which function to invoke at runtime, rather than during compilation.

[6.19.3] EXCEPTIONS
None.

[6.19.4] EXAMPLES
None.

[6.19.5] REMARKS
None.

[6.20] GT {opcode:gt}
in groups: logic, arithmetic

[6.20.1] SYNTAX

(0) gt %output local %lhs local %rhs local

[6.20.2] DESCRIPTION

Compares two numerical values (either a float, or an integer). If the left-hand side operand is an integer compares them as integers; if the left-hand side operand is a float compares them as floats.

The result is the value of the following operation:

    result = lhs > rhs

[6.20.3] EXCEPTIONS
None.

[6.20.4] EXAMPLES
None.

[6.20.5] REMARKS
None.

[6.21] GTE {opcode:gte}
in groups: logic, arithmetic

[6.21.1] SYNTAX

(0) gte %output local %lhs local %rhs local

[6.21.2] DESCRIPTION

Compares two numerical values (either a float, or an integer). If the left-hand side operand is an integer compares them as integers; if the left-hand side operand is a float compares them as floats.

The result is the value of the following operation:

    result = lhs >= rhs

[6.21.3] EXCEPTIONS
None.

[6.21.4] EXAMPLES

None.

[6.21.5] REMARKS
None.

[6.22] IDEC {opcode:idec}
in group: integer

[6.22.1] SYNTAX

(0) idec %1 local

[6.22.2] DESCRIPTION

Decrements an integer value at specified register.

[6.22.3] EXCEPTIONS
None.

[6.22.4] EXAMPLES

```
.function: main/0
    integer %1 local 43

    ; prints 43
    print %1 local

    ; increments integer at 1st local register
    idec %1 local

    ; prints 42
    print %1 local

    izero %0 local
    return
.end
```

[6.22.5] REMARKS

Same as for iinc.

[6.22.6] SEE ALSO
iinc

[6.23] IINC {opcode:iinc}
in group: integer

[6.23.1] SYNTAX

(0) iinc %1 local

[6.23.2] DESCRIPTION

Increments an integer value at specified register.

[6.23.3] EXCEPTIONS
None.

[6.23.4] EXAMPLES

```
.function: main/0
    integer %1 local 41

    ; prints 41
    print %1 local

    ; increments integer at 1st local register
    iinc %1 local

    ; prints 42
    print %1 local
```

```
        izero %0 local
        return
    .end
```

[6.23.5] REMARKS

This instruction operates on 64 bit signed integers. It is unreliable, in the sense that it uses the native int64_t type which incurs all the negative properties of no overflow checking, wraparounds, etc. However, it (the native type) is fast.

If you need safe, overflow-checked, version of this instruction look into checked arithmetic on fixed-width bit strings.

[6.23.6] SEE ALSO
idec

[6.24] IMPORT {opcode:import}
in group: import

[6.24.1] SYNTAX

```
(0)    import "a::module::to::import"
```

[6.24.2] DESCRIPTION

Import a module into the running program (perform dynamic linkage).

After a module is imported all symbols that it contains become immediately available to the running program.

[6.24.3] EXCEPTIONS

[6.24.3.1] IMPORT EXCEPTION

When the import fails an `ImportException' is thrown by the VM. The process that triggered the import can then service the exception appropriately, choosing from several different available solutions:

- crash
  This is the simplest thing to do, and requires no code to be written by the programmer. If the `ImportException' is not caught by the process it will cause the call stack to be unwound (stack unwinding is described in section 4.3.5.1.3), and the process will be shut down using the normal procedure (described in section 4.3.4).

[6.24.4] EXAMPLES
None.

[6.24.5] REMARKS
None.

[6.25] INTEGER {opcode:integer}
in groups: ctors, integer

[6.25.1] SYNTAX

```
(0)    integer %1 local 0
(1)    integer %1 local 42
(2)    integer %1 local default
```

[6.25.2] DESCRIPTION

Stores an integer value in a specified register.
The ` `integer %1 local default' ` syntax stores a 0.

[6.25.3] EXCEPTIONS
None.

[6.25.4] EXAMPLES
None.

[6.25.5] REMARKS
None.

[6.25.6] SEE ALSO
izero

[6.26] ISNULL {opcode:isnull}
in group: registers

[6.26.1] SYNTAX

(0) isnull %result local %target local

[6.26.2] DESCRIPTION

This instruction allows checking if a register is empty. It is most useful for setting up registers in static register set.

[6.26.2.1] RETURN VALUE

If the register selected as the target is empty, the instruction will store boolean true in register selected as the result destination. Otherwise, the instruction will store boolean false in register selected as the result destination.

[6.26.3] EXCEPTIONS
None.

[6.26.4] EXAMPLES
None.

[6.26.5] REMARKS
None.

[6.27] ITOF {opcode:itof}
in groups: cast, float, integer

[6.27.1] SYNTAX

(0) itof %output local %input local

[6.27.2] DESCRIPTION

Converts an integer value in input register to a float value, and puts result in output register.

[6.27.3] EXCEPTIONS
None.

[6.27.4] EXAMPLES
None.

[6.27.5] REMARKS

The conversion is done between native types of the CPU the VM is running on. Some precission loss may happen.

[6.28] IZERO {opcode:izero}
in groups: ctors, integer

[6.28.1] SYNTAX

(0) izero %1 local

[6.28.2] DESCRIPTION

The izero instruction stores an integer zero value in a specified register.

[6.28.3] EXCEPTIONS
None.

[6.28.4] EXAMPLES

```
.function: main/0
    ; return 0 from main function
    izero %0 local
    return
.end
```

[6.28.5] REMARKS

The instruction is equivalent to `integer register 0`, but is shorter.

[6.28.6] SEE ALSO
integer

[6.29] JOIN {opcode:join}
in group: join

[6.29.1] SYNTAX

```
(0)    join %0 local %1 local infinity
(1)    join %0 local %1 local 1ms
(2)    join %0 local %1 local 1s
(3)    join void %1 local infinity
(4)    join void %1 local 1ms
(5)    join void %1 local 1s
```

[6.29.2] DESCRIPTION

Used to join a different process.

This instruction causes the executing process to become suspended until the process that is the target of the instruction (whose PID is given in the second operand) finishes running.

After the target finishes running, this instruction will either transfer a return value of the joined process into the current process, or transfer the exception that killed the joined process into the current one (by rethrowing it).

Using `void` as the target register causes the value returned by the process to be dropped.

[6.29.3] EXCEPTIONS
None.

[6.29.4] EXAMPLES
None.

[6.29.5] REMARKS

It is impossible to specify all exception types that this instruction may throw due to the fact that it will rethrow any exception that killed its target.

[6.30] LT {opcode:lt}
in groups: logic, arithmetic

[6.30.1] SYNTAX

```
(0)    lt %output local %lhs local %rhs local
```

[6.30.2] DESCRIPTION

Compares two numerical values (either a float, or an integer). If the left-hand side operand is an integer compares them as integers; if the left-hand side operand is a float compares them as floats.

The result is the value of the following operation:

```
    result = lhs < rhs
```

[6.30.3] EXCEPTIONS
None.

[6.30.4] EXAMPLES

```
.function: main/0
    integer (.name: %iota three) local 3
    float (.name: %iota pi) local 3.14
    float (.name: %iota almost_pi) local 3.13

    .name: iota result

    ; Converts value taken from `%pi local' to an integer.
    lt %result local %three local %pi local
    print %result local

    ; Does not perform any conversions.
    ; Both operands are floats.
    lt %result local %almost_pi local %pi local
    print %result local

    ; Converts value taken from `%three local` to a float.
    lt %result local %pi local %three local
    print %result local

    izero %0 local
    return
.end
```

[6.30.5] REMARKS
None.

[6.31] LTE {opcode:lte}
in groups: logic, arithmetic

[6.31.1] SYNTAX

```
(0)    lte %output local %lhs local %rhs local
```

[6.31.2] DESCRIPTION

Compares two numerical values (either a float, or an integer). If the left-hand side operand is an integer compares them as integers; if the left-hand side operand is a float compares them as floats.

The result is the value of the following operation:

```
result = lhs <= rhs
```

[6.31.3] EXCEPTIONS
None.

[6.31.4] EXAMPLES

```
.function: main/0
    integer (.name: %iota three) local 3
    float (.name: %iota pi) local 3.14
    float (.name: %iota almost_pi) local 3.13

    .name: iota result

    ; Converts value taken from `%pi local' to an integer.
    lte %result local %three local %pi local
    print %result local

    ; Does not perform any conversions.
    ; Both operands are floats.
    ; The result is `true' because 3.13 is less than or
    ; equal to 3.14.
    lte %result local %almost_pi local %pi local
    print %result local

    ; Does not perform any conversions.
```

```
        ; Both operands are floats.
        ; The result is `false' because 3.14 is not less than or
        ; equal to 3.13.
        lte %result local %pi local %almost_pi local
        print %result local

        ; Converts value taken from `%three local` to a float.
        lte %result local %pi local %three local
        print %result local

        izero %0 local
        return
    .end
```

[6.31.5] REMARKS
None.

[6.32] MOVE {opcode:move}

in groups: registers, calls

[6.32.1] SYNTAX

```
(0)    move {dst:register_access} {src:register_access}
```

[6.32.2] DESCRIPTION

Moves a value from one register (src) to another (dst). These registers may be in different register sets.

After a value is moved it is no longer present in the source register, and the source register becomes empty. The source register must not be empty when this instruction is issued.

Being moved does not affect the value moved between registers in any way.

Any value that is present in the destination register when this instruction is executed is deleted before the move is performed.

[6.32.2.1] PASSING ARGUMENTS

A special form of this instruction (where the destination register set is "arguments") is used to pass arguments by move to a function:

```
text %1 local "Hello World!"
move %0 arguments %1 local      ; pass the argument by move
```

The arguments register set is described in section 1.1.1.2.1.4.

[6.32.2.2] OBTAINING PARAMETERS

Another special form of this instruction (where the source register set is "parameters") is used to extract parameters passed to a function:

```
move %1 local %0 parameters
```

The parameters register set is described in section 1.1.1.2.1.5.

[6.32.3] EXCEPTIONS

None.

[6.32.4] EXAMPLES

None.

[6.32.5] REMARKS

None.

[6.33] MUL {opcode:mul}

in group: arithmetic

[6.33.1] SYNTAX

(0) mul %output local %lhs local %rhs local

[6.33.2] DESCRIPTION

Multiplies two numerical values (either a float, or an integer). It models the following operation:

output = lhs * rhs

The result has the type of the left-hand side value. For example, if the left-hand side is an integer value the result will also be an integer value;
if the left-hand is a float value the result will also be a float value.

[6.33.3] EXCEPTIONS
None.

[6.33.4] EXAMPLES
None.

[6.33.5] REMARKS
None.

[6.34] NOP {opcode:nop}
in group: nop

[6.34.1] SYNTAX

(0) nop

[6.34.2] DESCRIPTION

Does nothing. This instruction causes the machine to execute an empty action.

[6.34.3] EXCEPTIONS
None.

[6.34.4] EXAMPLES
None.

[6.34.5] REMARKS
None.

[6.35] PROCESS {opcode:process}
in group: process

[6.35.1] SYNTAX

(0) process %0 local fn/0
(1) process void fn/0

[6.35.2] DESCRIPTION

Spawns a new process.

When a new process is spawned it begins executing the function that was passed as the second operand to the process instruction (full initialisation procedure of new processes is described in section 4.3.1).

Spawned process can be either "detached" or "joinable". Each of these two states is described in more details below.

[6.35.2.1] DETACHED PROCESSES

A detached process (spawned using the `process void fn/0' syntax) cannot be joined using the join instruction (described in section 6.29). Detached processes are independent of their parent processes and can run even after main function finishes.

[6.35.2.2] JOINABLE PROCESSES

A joinable process (spawned using the `process %0 local fn/0' syntax) can be joined to obtain the result of its execution (either a return value, or an exception).

[6.35.2.2.1] RELATION TO FUTURES AND PROMISES

A PID of a joinable process may be treated as a "future", and the process identified by it may treat its final return instruction as fulfilling a "promise".

[6.35.3] EXCEPTIONS
None.

[6.35.4] EXAMPLES
None.

[6.35.5] REMARKS
None.

[6.36] PTR {opcode:ptr}
in group: ptr

[6.36.1] SYNTAX

(0) ptr %Result local %Target local

[6.36.2] DESCRIPTION

Create a pointer to a value.

Pointers are described in detail in section 4.5.3.

[6.36.3] EXCEPTIONS
None.

[6.36.4] EXAMPLES
None.

[6.36.5] REMARKS
None.

[6.37] RECEIVE {opcode:receive}
in groups: concurrency, communication

[6.37.1] SYNTAX

(0) receive %target local 1ms
(1) receive %target local 1s
(2) receive %target local infinity
(3) receive void 1ms
(4) receive void 1s
(5) receive void infinity

[6.37.2] DESCRIPTION

Receive a message. Pops the first message in the message queue.

` `receive void 1ms' `, ` `receive void 1s' `, and ` `receive void infinity' ` will pop the first message in the queue but immediately discard it.

The timeouts (1s, 1ms, and infinity) specify for how long the VM should suspend the process if no message is available in the queue. These values are supplied by the programmer at compile time and are impossible to set dynamically (yet).

[6.37.3] EXCEPTIONS
NO MESSAGE RECEIVED

This exception is thrown by the receive instruction when either no timeout was given, or it has passed, but the message queue of the process executing the instruction is empty.

[6.37.4] EXAMPLES
None.

[6.37.5] REMARKS
None.

[6.38] RETURN {opcode:return}
in group: calls

[6.38.1] SYNTAX

(0) return

[6.38.2] DESCRIPTION

Pops the current call frame off the stack.

Before the frame is popped, all deferred calls scheduled by the defer instruction are executed till completion.

This instruction moves a value from `local register 0` (if any) to a "return value register". If the caller requested a return value then it is taken from this "return value register". If the caller did not request a return value, the value present in "return value register" is destroyed. Even if the caller did not request a return value, contents of `local register 0` are moved into the "return value register".

[6.38.3] EXCEPTIONS
None.

[6.38.4] EXAMPLES
None.

[6.38.5] REMARKS
None.

[6.39] SEND {opcode:send}
in groups: concurrency, communication

[6.39.1] SYNTAX

(0) send {pid:register-address} {src:register-address}

[6.39.2] DESCRIPTION

Send a message from {src} to the process with PID {pid}.

[6.39.3] EXCEPTIONS
None.

[6.39.4] EXAMPLES
None.

[6.39.5] REMARKS
None.

[6.40] STOF {opcode:stof}
in groups: cast, float, string

[6.40.1] SYNTAX

(0) stof %output local %input local

[6.40.2] DESCRIPTION

Converts a string value in input register to a float value, and puts result in output register.

[6.40.3] EXCEPTIONS
None.

[6.40.4] EXAMPLES
None.

[6.40.5] REMARKS
None.

[6.41] STOI {opcode:stoi}
in groups: cast, integer, string

[6.41.1] SYNTAX

(0) stoi %output local %input local

[6.41.2] DESCRIPTION

Converts a string value in input register to an integer value, and puts result in output register.

[6.41.3] EXCEPTIONS
None.

[6.41.4] EXAMPLES
None.

[6.41.5] REMARKS
None.

[6.42] SUB {opcode:sub}
in group: arithmetic

[6.42.1] SYNTAX

(0) sub %output local %lhs local %rhs local

[6.42.2] DESCRIPTION

Subtracts two numerical values (either a float, or an integer).

The result has the type of the left-hand side value. For example, if the left-hand side is an integer value the result will also be an integer value;
if the left-hand is a float value the result will also be a float value.

[6.42.3] EXCEPTIONS
None.

[6.42.4] EXAMPLES
None.

[6.42.5] REMARKS
None.

[6.43] TAILCALL {opcode:tailcall}
in group: calls

[6.43.1] SYNTAX

(0) tailcall Function_name

[6.43.2] DESCRIPTION

Makes a tail call to a function.

A tail call does not increase stack depth, it pops the caller's call frame before pushing its own.

[6.43.3] EXCEPTIONS
None.

[6.43.4] EXAMPLES
None.

[6.43.5] REMARKS
None.

[6.44] THROW {opcode:throw}
in groups: exceptions, error-handling

[6.44.1] SYNTAX

(0) throw %1 local

[6.44.2] DESCRIPTION

Throw a value. Thrown value is moved to the exception register (described in section 1.1.1.2.2.2).

Throwing a value triggers the exception handling mechanism, and may result in a stack being unwound. Exception handling is described in section 4.3.5.1. Stack unwinding is described in section 4.3.5.1.3.

[6.44.3] EXCEPTIONS
None.

[6.44.4] EXAMPLES
None.

[6.44.5] REMARKS
None.