Delft University of Technology

DelftVM

November 25, 2013

Maurice Bos

Preface

The document you are currently reading is the result of my Bachelor's thesis for the BSc *Computer Science* at the Delft University of Technology.

I would like to thank Stefan Dulman, the first supervisor of this project, for introducing me to the subject of spatial computing, and giving me the opportunity and inspiration to work on this project.

Furthermore, I want to thank Koen Langendoen, the second supervisor of this project, for his useful feedback while writing this report.

Delft, July 2013
Maurice Bos

Contents

1 Introduction

The Delft VM is a Virtual Machine for Spatial Computing. First, this chapter explains what spatial computing is using a definition and a few examples. Then, it shows how virtual machines can be very useful in spatial computing. And finally, the design requirements for the Delft VM are discussed.

1.1 Spatial Computing

The term 'Spatial Computing' is a very abstract, vague and broad term. Lots of fields with active research from robotics to biology can fall under Spatial Computing. A good way to define the term is as follows:

> "[In a spatial computer,] a program runs on a collection of devices spread through space whose ability to interact is strongly dependent on their geometry." — *J. Bachrach, J. Beal*

1.1.1 Examples

The following three examples should give an idea of what a 'spatial computer' can be.

Wireless Sensor Network A wireless sensor network is a network of small devices with sensors, which can communicate with eachother wirelessly. Such a network can be seen as a single spatial computer.

For example, wireless sensor nodes with rain sensors could be spread over a big area of farmland. The sensors could detect that some parts of the field are getting more rain than other parts by communicating with eachother. The farmer (or an automated irrigation system) could then be informed to fix the sitation.

Quadcopter Swarm Imagine a group of (small) quadcopters flying in formations such that the whole group behaves like a 'swarm.'[1] The swarm as a whole can be seen as a single 'spatial computer' capable of movement. We would like to be able to handle this spatial computer as a single entity. Instead of telling the individual quadcopters how to fly to their next goal while avoiding eachother, we would tell the swarm to go to some other location.

Interactive Floor This one was actually built on a small scale at the Delft University of Technology. At the architecture faculty, there is a room with an 'interactive floor': Each tile of the floor has a small device underneath it with a pressure sensor (such that it can detect things standing on the tile) and a big bright RGB LED which is clearly visible through a small window in the tile. Each device can communicate with the devices of the four adjacent tiles. It is part of a project to make architecture less static.

[1] At the time of writing, http://youtube.com/ has very interesting videos of small quadcopters flying in formations. Simply search for "quadcopter swarm."

The floor could with lighting patterns show paths or directions to other people in the room. It might detect that one interesting part of the room is visited less often than other parts, and direct people there with light effects.

1.1.2 Scalability

In all three examples (the sensor network, quadcopters, and the floor), the nodes could simply give their data (sensor values, location, etc.) to a central node, and all the processing could happen there. However, in spatial computing, it is common to not centralize anything. The key point here is scalability. With a centralized system, it might still work if we double or triple the area, but the central node and all communication paths would certainly need a big upgrade if we for example make the sensor network observe an entire country, or expand the interactive floor to cover a whole city. With a decentralized system common in spatial computing, no single entity knows all information of the entire network; only local information (its own information and the information of close neighbours) is needed and used by the nodes. This means that only the node density, but not the total area, is relevant for the complexity of the spatial computer and the program running on it.

1.1.3 Extreme Examples

All of the three examples given before were fairly realistic; I've seen working implementations of all three of them. However, although all of those implementations were set up in a decentralized way the networks were small enough so they could've been easily implemented in a centralized way.

This section however, explains a few extreme cases of what can still be called a spatial computer, in which a centralized way would be unthinkable due to the unimaginable amount of devices in such spatial computers.

Claytronics Imagine you have trillions of robots smaller than the smallest sand grains, and with the ability to move and attach to each other. Put all these robots together, and you have 'programmable matter.' The spatial computer can physically take virtually any form by rearranging its nodes.[2]

Imagine the spatial computer has the task to transform into the shape of a car. Instead of calculating the right position of each actuator of each of the trillions of robots centrally and somehow getting all that information to the right nodes, a perfect and brilliant decentralized program for the spatial computer would somehow let the calculations for the geometry of the wheels happen in the nodes that will be wheels, such that that information will not have to travel through the entire network.

The Human Body Although the human body is not made from small robots or sensor nodes running software, it is a very good example of a spatial computer in which the biological cells can be seen as the individual devices. The cells can communicate with each other using chemicals (like hormones) and electrical pulses. The ability of two cells

[2] At the time of writing, http://youtube.com/ has a video demonstrating this idea. Search for "Claytronics."

to communicate is dependent on their location in the body. Cells directly connected or inside the nerve system can communicate much more easily and quickly with other cells than the blood cells in veins for example.

No single cell in a body knows all the information, or controls the entire body. Although the brain controls most of the significant movement of the body, you don't have to think about widening some veins when a body part temporarily needs more oxygen. Those processes (widening your veins, most digestive processes, etc.) are all local processes. It would be impossible to manage such a system of about 50 trillion devices if everything was centralized.

1.2 Programming Spatial Computers

Although programming programmable matter and human bodies would be way more spectacular, we focus on programming the kind of spatial computers that are feasible at the present day, like the three examples given in section 1.1.1. In order to program a spatial computer, we need some kind of programming language, and all the tools to somehow get a program expressed in that language to actually run on a spatial computer.

One approach is to create a virtual machine which runs on all devices and can execute programs with a specifically for the language designed instruction set. This approach has three main advantages over directly compiling programs to run natively on the devices:

- The virtual machine could relatively easily allow for (virally) updating the program, such that as long as the virtual machine is running, the program can still be updated. This would mean that a faulty program can never render a device unusable.
- The program binaries can be a lot smaller, since the instruction set can be optimized for spatial computing. For example, single instructions for common operations like sending a message or calculating a dot product can exist. (Which usually take tens or even hundreds of instructions in the native instruction set of the devices.)
- The same program binary can run on each of the nodes of a heterogeneous spatial computer, like one in which some of the devices are laptops, some are mobile phones, etc.

1.2.1 Proto

Proto is a non-pure functional programming language for spatial computers based on Lisp, and designed at MIT by J. Bachrach and J. Beal. It has a compiler, which gives 'Proto bytecode' as output, which can be executed by the 'Proto Virtual Machine' that runs on the nodes. A simulator also exists, which runs the virtual machine locally for every simulated node. The original virtual machine for Proto has been superseded by the DelftProto VM, which I made in 2011.

Every value in Proto is a floating point value, or a vector of values. Since such vectors may contain other vectors, calling them tuples might be more appropriate. The Proto

virtual machine supports these types directly. Instructions exist to operate on vectors as a whole (some even recursively).

A Proto program has state variables, and running the program once will advance this state to the next state, based on the state of the neighbouring nodes and sensor values. The virtual machine will run the program repeatedly, taking care of broadcasting the current state and processing any incoming data from the other nodes. The program itself has no control over when what messages are sent or when incoming messages are processed.

1.2.2 Delft VM

The Delft VM is a virtual machine for spatial computers, but is not specifically designed for one programming language. The goal is to provide a single platform for spatial computers which can be programmed using Proto, NetLogo, and any other (future) programming language which targets spatial computers like the three examples given in section 1.1.1. Of course, compilers will have to be made for the Delft VM architecture before any of these languages can be used. However, since a virtual machine without a compiler is still a lot more useful than a compiler for a platform that doesn't exist yet, a compiler would be the next step after this virtual machine is done. The virtual machine alone for example can still be programmed by writing the instructions by hand, which is not very hard when compared to doing the same for architectures like x86 and ARM, since the instructions of the Delft VM are all very high level.

1.3 Design Requirements for the Delft VM

In this section, we list the requirements for the Delft VM in order to reach its goal of being a useful generic virtual machine for spatial computers, as described in Section 1.2.2.

First, for the **functionality** of the VM:

- The instruction set should be complete. That is, any reasonable program for the VM should be expressable using the instruction set.
- The instruction set should be optimized to express programs in as few bytes as possible. This is to allow for virally updating the program, without having to send lots of data through the entire network. Also, since some microcontrollers on which the virtual machine will run have only a few kibibytes of memory, the space taken by the program itself should be kept to a minimum.
- Data types that occur frequently in programs for spatial computers should be natively supported by the VM. That is, there should be instructions to handle common types like integers, floating point values, and possibly vectors.
- The instruction set should be simple, such that programs can be written by hand, and such that making a compiler for it will be relatively easy.

Furthermore, for the **implementation** of the VM:

- It must be written in a programming language that can be used for virtually all embedded platforms, such as C or C++, and not rely on (non-standard) language features that might not be available on all platforms.
- The implementation of all platform-dependent functionality, such as sending messages, should be left open to support all platforms. Whether the VM is ported to a platform with wireless communication, or one in a fixed wired network, the source code of the VM itself should not have to be changed.
- The virtual machine implementation should be small and lightweight, as most target platforms have microcontrollers with very little program memory (ROM).
- Memory (RAM) should be used efficiently, due to possible memory limitations of the platform.

2 Design

As we have seen in the last section, the requirements for a virtual machine for spatial computing are quite different than those for any other common virtual machine. Overall, the Delft VM is a very 'high level' virtual machine in the sense that it supports functions usually only found in language support libraries as single instructions, such as calculating the dot product of two vectors and sending messages.

This section describes the architecture of the Delft VM from the perspective of the writer of code that will be run in the virtual machine (which may be the compiler).

2.1 Primitive Types

Since mathematical calculations (mostly floating point) are very common in spatial computing, the primitive type in the Delft VM is a 'number', which can hold either a floating point number or a signed integer. (See section 2.1.1.)

Because of the 'spatial' aspect, vectors can be found in virtually every spatial algorithm. Therefore, in addition to numbers, the stack can also hold vectors of numbers (of any length), and most instructions can operate on entire vectors at once. (See section 2.1.2.)

Unlike in most architectures, types are dynamic: From only the value one can tell which type it is (floating point, integer, vector or a reference).

2.1.1 Numbers

A number value is either a (32-bit) floating point value or a (16-bit, signed) integer. Integers will be converted implicitly to floating point numbers when used in combination with floating point numbers. Although a floating point value will suffice for most values, integers are a better choice for values that represent counters, intexes and truth values (booleans).

2.1.2 Vectors

In addition to numbers, the stack can contain vectors of any number of numbers. In existing architectures with fundamental support for vectors, the contents of vectors are usually placed in a managed memory area. Different management strategies have been tried, but all of them have quite some memory overhead, not even mentioning the time overhead. Also, a memory manager might increase the program size of the virtual machine itself.

In the Delft VM, vectors are not stored as a pointer or reference to their data in a managed memory area; the contents of vectors are stored entirely on the stack. For multiple copies of the same vector, only the contents for the lowest (first pushed) vector are stored, the other 'vectors' are simply references to this last vector.

So there are two ways to represent a vector on the stack:

Vector Headers The first way is to use a special 'vector header' value followed directly by its actual contents (the numbers):

There are special 'vector header' values, which contain an integer: The size of the vector. The first value after the vector header is the last element of the vector. This document refers to these vector headers as: VN, where N is the size of the vector. (e.g. V5)

For example, when the stack looks like this:

(*top*) V3 1 2 3 4 V0 V1 5 6 (*bottom*)

Then it contains the vector [3 2 1], the number 4, an empty vector, a vector with the single element 5, and the number 6.

Vector References To avoid having to copy all the contents when a vector is needed in two different places on the stack, the other way to represent a vector is by a reference to an already existing one.

Similar to the vector headers, there are another set of special values called 'vector references', which also contain an integer: The offset. The offset is the relative distance (counted in number of elementary values) from the reference value to the vector header value of the vector it refers to. These references can only refer to vector headers lower on the stack (i.e. vector headers that have been pushed before this reference is pushed). That way, the reference is always popped before the value it refers to is popped, so there is no need for any memory management such as reference counting.

This document refers to these vector references as: RN, where N is distance to the vector header. (e.g. R5)

For example, when the stack looks like this:

(*top*) R3 1 2 V3 4 5 6 (*bottom*)

Then it contains (a reference to) the vector [6 5 4], the number 1, the number 2, and again the vector [6 5 4].

Note that vector references always refer directly to vector headers, never to other vector references.

When instructions that would modify a vector are applied to a reference to a vector, the original vector is not modified. Instead, the reference will be replaced by a new vector containing the modified results. For some programs this can result in slightly more time spent copying the contents of vectors, but in almost all cases this outweighs the overhead (both in time and memory) of memory management techniques.

2.2 Stack

The previous section already described the type of values that can be stored on the stack and how they are stored. This section describes the available operations to work with the stack.

The stack is designed to be very flexible. It has push and pop operations, which both take vectors (as described in section 2.1.2) into account. However, the separate elements of a vector (references, its header, and its contents) can also be handled separately, for fine control over the stack.

Furthermore, popped elements can be awaken from death again with an 'unpop' operation, if they are not overwritten yet in memory by any other elements pushed on the stack. The idea behind it (apart from simply allowing fine control over the stack) is as follows: Most instructions pop their operands, because that is usually what is wanted. If however an operand is used by two different instructions right after each other, the operand can be used again simply by 'unpopping' it.

Following are brief descriptions of the most fundamental stack operations. Each of these operations has a corresponding instruction.

Push Push a single number on the stack.

Make Vector Push a vector header on the stack with a specified 'size' property N. This effectively turns the top N elements into a vector. Since the contents of a vector are stored on the stack in reversed order, the last pushed number will be the last element of the vector.

So pushing 1, 2 and 3, in that order, and then doing a 'make vector 3' operation, will leave the stack with a single vector: [1 2 3].

Pop Pop the top value of the stack. If it is a vector header, all of the contents of the vector represented are popped as well.

So if the stack looks like:

$$(top) \quad \text{V3} \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad (bottom)$$

(Such that it contains the vector [3 2 1], the number 4, and the number 5.) Then after popping twice, the vector and the number 4 have been removed, leaving only the number 5 on the stack.

Pop Elements Pop a one or more values from the stack. Unlike the regular 'pop' operation, vectors are not treated specially.

So if the stack looks like:

$$(top) \quad \text{V2} \quad 1 \quad 2 \quad \text{V1} \quad 3 \quad 4 \quad (bottom)$$

(Such that it contains the vector [2 1], the vector [3] and the number 4.) Then after popping five elements, the stack will contain only one number: 4.

This can be used to pop multiple values from the stack in $O(1)$ time, because unlike the regular 'pop' operation, 'pop elements' does not have to check each popped element whether it is a vector header or not.

Unpop Element(s) Revives values back from the death: When values are popped, another value gets to be called the 'top of the stack', but the values 'above the top' are

not destroyed. The 'unpop' operation will simply put back any popped elements on the stack, given that they were not overwritten by any intermediate push operations.

So on a sufficiently filled stack, after popping 5 elements and then 'unpopping' 5 elements, nothing has happened.

Copy Pushes a copy of a value that is already on the stack, given its distance from the top of the stack in number of values. (So, 0 refers to the last pushed value.)

If the source is a vector, only a reference to the vector header is pushed. So in all cases, this operation only pushes a single value.

For example, if the stack looks like:

(*top*) 7 V3 1 2 3 (*bottom*)

(Such that it contains the number 7, and the vector [3 2 1].) Then after a 'copy 1' operation, it will look like:

(*top*) R2 7 V3 1 2 3 (*bottom*)

(Such that it contains the vector [3 2 1], the number 7, and again the vector [3 2 1].)

2.3 Instruction Set

The most important requirements for the instruction set were that it should be complete, and that it should be size efficient (everything reasonable should be possible, and with a minimal number of instructions).

Since the Delft VM is a stack machine, all operands to the instructions are implicit. That is, you do not have to (or can) specify on what values or memory locations an instruction should operate. Except for a few special stack instructions like `copy`, instructions always operate on the value(s) at the top of the stack.

Furthermore, the instructions are polymorphic. This means that the exact behaviour of an instruction depends on the type of values it gets as operands. For example, the result of the `add` instruction can be an integer, a floating point value, or a vector, depending on the operands.

2.3.1 Instruction Codes

Each instruction has a single byte representation: Its instruction code. The instruction codes are not assigned randomly or consecutively. The space of 256 instruction codes is divided in 4 groups of 8 subgroups of 8 instructions, such that when the instruction code is written in octal with three digits, the first digit is the group number, the second digit is the subgroup number within that group, and the third digit is the instruction within that subgroup.

For example, group 1 contains all the mathematical instructions, subgroup 7 of which contains the 'vector math' instructions, of which instruction 3 is the `dot` instruction that

calculates the dot product of two vectors. The instruction code for `dot` is therefore 173 in octal, 01111011 in binary, and 123 in decimal.

See appendix A for a full overview of all instruction codes.

2.3.2 Instruction Parameters

Some instructions, like the jump instruction, take a parameter. (In this case the (relative) address to jump to.) Such a parameter is encoded in up to five bytes after the instruction code itself. For each byte of the parameter, the most significant bits shows whether another byte follows (bit set), or not (bit cleared). This way, values less than 128 only take a single byte. (See Section 3.3 for the exact details of this encoding.)

There are 'shortcut instructions' available for most commonly used instruction-parameter combinations: For example, instruction 200 is the `copy` instruction that takes a single parameter, but instruction 203 is the `copy 3` instruction, which has the same effect as the `copy` instruction with the parameter 3.

2.4 Numeric Operations

All the instructions performing calculations take and pop their operands from the stack, and push the result back. Note that this means that the first operand will be overwritten by the result, but other operands (if any) can be 'unpopped', as explained in section 2.2.

The following numeric instructions are available:

> `add`, `sub`, `mul`, `div`, `mod`, `neg`, `abs`, `mux`, `trunc`, `exp`, `log`, `pow`, `sqrt`, `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `atan2`, `min`, `max`

(Please refer to appendix B for their exact definition.)

Non-commutative binary operations (such as `sub`) also have an instruction that takes the two operands in reverse order (such as `rsub`):

> `rsub`, `rdiv`, `rmod`, `rpow`, `ratan2`

For the comparison of values, the following instructions are provided:

> `neq`, `eq`, `lt`, `lte`, `gt`, `gte`

These instructions push one of the integers 0 and 1, depending on the result of the comparision.

All of the above instructions not only work on numbers, but on vectors as well. For vectors, the operation will be applied pointwise to the operands. If not all operands are vectors of the same length, the result is as if the shorter one has sufficient zeros appended. If one operand is a vector and the other a single number, the single number will be handled as if it was a vector of one element.

To push literal numbers on the stack, the instructions `lit` and `litf` are available. The former pushes an integer, the latter a floating point value. Both take one parameter: the value to push.

When comparing two vectors with the comparison instructions given above, the result will be a vector containing the pointwise result of the comparison. For example, applying the less-than instruction `lt` to the vectors [2 1 3] and [1 2 3] will give [0 1 0]. If you want to compare the vectors as a whole instead of pointwise, they can be compared lexicographically using the following instructions:

`lneq`, `leq`, `llt`, `llte`, `lgt`, `lgte`

The result of these instructions will always be a single integer (0 or 1).

And finally, specially for vectors, the following instructions are availeble:

`scale`, `len`, `dot`, `cross`, `cross-z`

These instructions respectively scale a vector with a scalar (number), calculate the spatial length (i.e. norm) of a vector, the dot product of two vectors, the cross product of two vectors, and the z-component of the cross product of two vectors. Again, when a number is given where a vector is expected, it will be seen as a vector with a single element. Also, where a too short vector is given, it will be interpreted as if it has zeros appended to it.

2.5 Local Memory

Apart from storing temporary values on the stack, values can be stored for long term in an area that is simply called 'the memory.' This is used for state variables, like in feedback loops. For example, a function that integrates some values over time should keep its sum in this memory, so the stack can freely be used for the temporary values required to do the calculations.

The memory cannot store vector headers or references. However, apart from numbers it does have a special 'not set' value (all bits set), indicating that a value has not (yet) been initialized. Of course, three consecutive numbers could represent the three elements of a three dimensional vector, but the virtual machine will not keep track of that information.

There are instructions to copy values from the stack into the memory, and the other way around. Values in memory cannot directly be used as operands to other instructions; all other instructions only work with values on the stack. The four instructions to use the memory are: `get`, `set`, `isset`, and `unset`.

The `get` instruction takes a size and an index as parameter, and will put a vector on the stack of the specified size containing values from the memory starting from the specified index. If the size is equal to 1, a single number (not a vector with a single element) is pushed.

The `set` instruction takes the same parameters as the `get` instruction (a size and an index), and will store the vector or number at the top of the stack in the memory starting at the given index. If the vector has more elements than the specified size, it will be truncated. If it has less, or when only a single number (not a vector) is given, the rest will be filled with zeros.

The `isset` instruction takes only an index, and will push one of the integers 0 and 1 on the stack, depending on whether the specified value in memory contains the 'not set' value (0), or not (1). The `unset` instruction can be used to reset values to this special 'not set' value. It takes the same parameters as the `set` instruction: the sIze and index.

2.6 Function Calling

The Delft VM supports functions by providing the `call` and `ret` (return) instructions. The `call` instruction takes the address (relative to its own address) of the first instruction of the function. After executing the `call` instruction, execution will continue inside the called function. When the function executes the `ret` instruction (which takes no parameters), execution will continue right after the original `call` instruction. This works properly for nested functions as well (i.e. a called function may call functions, including itself).

2.6.1 Separate Call Stack

In virtually all architectures, the information required by the return instruction (such as the address to return to), is pushed on the stack by the call function. This effectively divides the stack in two portions: The part that was already there before the call (below the return information), and the part that is used inside the function (above the return information). This means that a function cannot simply pop values that were pushed on the stack by the caller.

The Delft VM however, uses a different approach. A separate stack is used to store the return information. This separate stack is called the 'call stack.'[3] The 'regular' stack remains unmodified when calling a function. This greatly simplifies taking parameters and returning (possibly multiple) values.

For example, a function taking three numbers and returning the sum of them looks like this:

```
add, add, ret
```

Nothing special needs to be done to retrieve the parameters of the function, since the `add` instructions can just access the top of the stack as it was in the caller.

As can be seen from this example, inlining (the operation of replacing a function call with equivalent instructions based on the instructions inside the called function) is trivial: A call to the example function above can be replaced with `add, add`. Also, turning a sequence of instructions into a function call to a (newly generated) function is trivial as well. This can be beneficial for the overall program size when that sequence occurs multiple times in the program.

[3] The programming language 'Forth' also uses a separate stack for return information. In Forth, this is called the 'return stack', usually abbreviated as 'rstack.'

2.6.2 Local Memory Offset

Let's say we have a function that integrates a value over time, such that when it is called repeatedly with the speed of the device, it will return an estimation of the travelled distance. When called, it will add the value on top of the stack multiplied by some time-dependent factor to a (state) value kept in memory.

Now if we want to integrate yet another value (for example, the result of a gyroscopic sensor to estimate the rotation), we cannot re-use this exact same function because that would simply use the same (state) value and add the speed values and the gyroscopic values together.

A solution could be to modify the integration function such that it also takes a memory address (index) as parameter, such that we can tell it to use a different memory value for the position and rotation. However, a cleaner solution is available.

When calling a function, a 'local memory offset' can be specified. (By using the `calll` instruction instead `call`.) Inside the called function, the memory appears to start at this memory offset, instead of at 0. For example, when a function is called with a memory offset of 5, and the called function reads from the memory using index 2, index 7 is effectively used.

The idea is that all functions can be written as if they use the memory from address 0 and up. It is the responsibility of the caller to shift the view of the memory for the function such that the function will use the right part of the memory. This way, the same function can be re-used with different state variables, like what we wanted with the integration function.

The active memory offset can be thought of as like the 'this' pointer in C++, or to the 'self' parameter in Python, when we see functions as member functions of an 'object' stored at the memory offset.

2.7 Communication

Most algorithms for spatial computers require some form of communication between the nodes. For example, an algorithm that synchronizes a periodic clock on all nodes in the network should be able to know the current time value of the clocks of the neighbour nodes. In this case, the 'current time' value is thus somehow made available to the neighbour nodes; it is a public state variable. State variables that are not public are called local state variables.

In some programming languages, the program has to form the messages to be broadcast and process received messages itself. In some other programming languages (for example, Proto) however, the platform will handle most of this automatically. The Proto virtual machine will broadcast the public values and process any incoming messages periodically. Special instructions can then be used to inspect the received public values of the neighbours. Although the second way is usually convenient and sufficient, the first way gives the program(mer) more freedom and flexibility.

The Delft VM implements a hybrid solution. State variables stored in memory can be either 'local' or 'public.' All, or a part of, the public state variables can be broadcast with the `send` instruction. Incoming messages will be placed in a queue and can be retrieved individually by the program itself with the `receive` instruction, or be processed automatically by the virtual machine with the `import` instruction. The `import` instruction makes the received data available in a structure called the 'neighbour list,' which is described in section 2.7.2.

All sent messages contain a small header that includes the unique id of the node and the start index (in the public memory) and size of the data that follows.

2.7.1 Public Memory

There is a separate memory area called 'the public memory' identical to the local memory (as described by section 2.5) to store all public state variables. For each of the four instructions to access the local memory, the same instruction exists for the public memory. Also, identical to the 'local memory offset' (2.6.2), there is a 'public memory offset.'

In most applications, each node keeps a copy of each of the public state variables of each of its neighbours. Because all neighbours have their public state variables put together in their public memory, a single contiguous memory area (just like the public memory) can be used to store the neighbours public state. This is the main reason for choosing a separate memory area for public state over having a flag for each value in the (local) memory to mark the value as a local or public state variable.

2.7.2 Neighbour List

Apart from the stack and the local and public memory, there is a fourth data structure available for the program running in the virtual machine: the neighbour list. At all times, exactly one of the neighbours in this list is called the 'current neighbour.' With the `nbr-get` and `nbr-isset` instructions, values from the local copy of the public memory of the current neighbour can be read.

Properties about the neighbours can be read with the `nbr-prop` instruction. The first property (index 0) is always the unique ID of the node. The second property (index 1) is always the time at which the last message from this node was received. The other properties are platform specific. Examples include the estimated distance (for example, using the signal strength of the received messages), and the exact coordinates (for example, in a fixed network where all positions are known).

Since in most algorithms nodes are considered neighbours of themselves, the first element in the neighbour list represents the local node itself. This way, the neighbour list is never empty and there is always a current neighbour.

The `nbr-next` instruction marks the next neighbour in the list as the current neighbour. The list wraps around, so after the last neighbour, the first one (representing the local node) is selected again. The instruction takes a (relative) instruction address as parameter, and jumps to this address unless the first element of the neighbour list is selected again.

This way, the `nbr-next` instruction can be used to loop once over all neighbours (including the local node).

For example, the following code gives the sum of the first public state variable of all neighbours and the local node:

```
    lit 0
┌─→ nbr-get 1 0
│   add
└── nbr-next -5
```

The `nbr-skip` instruction is the same as `nbr-next`, except that it only takes the jump when the local node is selected as neighbour again. So, `nbr-skip` can be used to skip the first neighbour (the local node). The jump is then taken if there are no (other) neighbours.

For example, the following code gives the sum of the first public state variable of all neighbours not including the local node:

```
    lit 0
┌─┌─→ nbr-skip 5
│ │   nbr-get 1 0
│ │   add
│ └── nbr-next -5
└──→
```

The neighbour list is automatically updated using the incomming messages by the `import` instruction. In order to allow manual processing of the messages using the `receive` instruction, the neighbour list can be modified directly using the `nbr-clear`, `nbr-add`, `nbr-del`, `nbr-set`, `nbr-unset` instructions.

Note that when using the `receive` instruction, the memory area in the neighbour list reserved for a copy of the public memory of a neighbour doesn't necessarily have to be used for this purpose. It could be used to store any kind of useful information about the neighbour.

2.8 Debug Features

Eight instruction codes are reserved to be used by a debugger. One of them is a simple breakpoint: A debugger can overwrite an instruction in the program with this 'breakpoint' instruction. When executed, the virtual machine will be paused such that the state of the machine can be inspected from the debugger. Before continuing, the debugger can overwrite it again with the original instruction, so the machine can go on as if nothing happened.

A second instruction is the 'global breakpoint.' It has the same behaviour as the breakpoint instruction, except that it will broadcast a message requesting all receiving virtual machines to act as if they executed the same instruction as well. This 'breakpoint' message will thus spread virally through the network, pausing every virtual machine, ready to be inspected by a debugger.

The other six debugger instructions are reserved, and do nothing by default. The platform-specific code can provide the behaviour for those, such that platform-specific debug features can be implemented.

Both stacks, the location of the current instruction and the memory of any virtual machine can be inspected from anywhere in the network. Such an 'inspection' message contains the unique ID of a virtual machine from which the data should be read, will be sent from the debugger, and spread virally through the network. Only the targetted virtual machine will send a response, which will also travel virally. Obviously, this is not the most efficient way, but since it is only for debugging, the simplicity and robusteness of it count more than its efficiency.

3 Implementation

In this section, we take a look at how the Delft VM is implemented. First, we discuss some general details about the implementation, and then some interesting details about the implementation of a few specific features of the VM.

3.1 Generic Details

The Delft VM is written in C++, as defined by the ISO standard from 2003, but is written such that it is compatible with the ISO C++ standard from 2011 and the expected next standard that will be published in 2014. [4]

Since the Delft VM will run on embedded devices where a full implementation of the C++ standard library is not available, Delft VM only assumes a freestanding C++ environment. (The exact definition of a 'freestanding environent' is defined in the ISO C++ standard.) Also, certain features of the language that are not commonly supported on embedded platforms, or simply cannot be implemented efficiently on simple microcontrollers, are avoided. These features include exceptions, virtual member functions, and virtual inheritance.

The source code of the Delft VM is licensed under the GNU General Public License, version 3. All code is documented using comments that can be parsed by Doxygen to generate a reference manual.

3.2 Dynamic Typed Values

As described in Section 2.1, values on the stack can have different types. They are either a floating point value, an integer, a vector header, or a vector reference. All values in the Delft VM (including those on the stack) are 32 bits, which already includes their (dynamic) type information.

Floating point values are represented by their IEEE754 32-bit representation: 1 sign bit, 8 bits for the exponent, and the other 23 bits for the mantissa. In this representation, values with the exponent part set to its extreme (all bits set) are reserved for values other than regular numbers. Two of those are used for positive and negative infinity (mantissa zero, and sign bit clear or set respectively). The other $2^{24} - 2$ are reserved for the 'undefined results', such as the result of $0/0$ or $\arcsin 2$, and are usually referred to as `NaN`: Not a Number.

Although all these different `NaN` values could be used to represent different types of calculation errors (such that we can tell whether the source was a $0/0$ or $\arcsin 2$), in the Delft VM they are put to use in a different way:

[4] None of the planned changes for the future ISO C++ standard of 2017 seem to break anything of the Delft VM either, but that is still subject to change.

Binary Representation				Meaning
Byte 3	Byte 2	Byte 1	Byte 0	
SEEEEEEE	EMMMMMMM	MMMMMMMM	MMMMMMMM	floating point number
01111111	10000000	00000000	00000000	$+\infty$
11111111	10000000	00000000	00000000	$-\infty$
01111111	10000000	00000000	00000001	NaN
01111111	11000000	XXXXXXXX	XXXXXXXX	integer
11111111	11000000	XXXXXXXX	XXXXXXXX	vector header
11111111	11100000	XXXXXXXX	XXXXXXXX	vector reference

Table 1: Binary representations of values in the Delft VM. The S, E and M bits represent the sign, exponent and mantissa, respectively. The X bits contain a 16 bit integer.

One of these values is still used to represent an 'undefined result.' Any reference later in this document to NaN refers to this value, not to all NaN values of the IEEE754 representation.

Furthermore, 2^{16} of these values are used to represent all of the 16-bit 2's complement signed integers. Although all these integers can also be represented in the floating point format, these integer values are a better choice for values that represent counters, indexes, and boolean values.

Also, of the IEEE754 NaNs, another 2^{16} values are used as 'vector headers': The headers for vectors of size 0 till $2^{16} - 1$. Another 2^{16} values are reserved to represent the 'vector references': The references to vectors with an offset of 0 till $2^{16} - 1$.

The binary representations of the values are chosen such that only the two most significant bytes of the value have to be inspected in order to determine the type. Also, in the case of an integer, a vector header or a vector reference, the value, size or offset respectively are stored in the two least significant bytes, such that no bit shifting or other calculations are required when accessing these values. Table 1 shows the exact binary representation of all possible values.

3.3 Parameter Encoding

For instructions that take a parameter, such as the jump instruction (that takes an address), the parameter is encoded in as a variable length integer. For each byte of the parameter, the most significant bits indicates whether another byte follows (set) or not (cleared), and the other seven are part of the encoded 32-bit value. The parts of the value that are not given are assumed to be zero. The least significant seven bits of the encoded value are stored in the first byte, such that for (unsigned) values less than 128, only one byte is needed.

Using two's complement encoding for signed values would not work very well with this encoding, as a small and common value like -1 has all 32 bits set and whould thus require five bytes in the encoding. For signed parameters, the value is encoded as follows: The absolute value times two, minus one for negative values, is stored as an unsigned value as

described above. This way, the values between -64 and 63 (inclusive) can be stored in a single byte. The least significant bit of the encoded value can be seen as the 'sign bit'.

To clarify:

Encoded (unsigned) value	0	1	2	3	4	5	6	7	⋯
Meaning as signed value	0	−1	+1	−2	+2	−3	+3	−4	⋯

3.4 Neighbour List

The neighbour list is implemented as a circular linked list structure. Each node contains the properties of the corresponding neighbour (such as its unique ID), and has space to contain a copy of the public memory of that neighbour. The nodes are dynamically allocated, to allow for insertion and removal of nodes when the layout of the network changes. (For example, in a wireless network.)

The node in the neighbour list representing the local virtual machine has a type that uses inheritance to also store the local memory, in addition to what makes it a neighbour node. The 'current neighbour' is kept track of by a pointer to a node in the neighbour list. Usually, it points at the node representing the local machine (which always exists).

3.5 Platform Specific Functionality

In order to specify the platform specific implementations of functions like sending a message or reading out a sensor value to the platform implementation, a (constant) object of the type 'PlatformSpecifics' should be filled in. This structure contains function pointers like 'send' (to send a message), and 'in' (for reading out a sensor value), which are called by the VM when the related instructions are executed in the VM. This way, the unmodified virtual machine can be used on platforms with very different ways of for example communication.

3.6 Testing

All classes and functions are automatically tested using unit tests. The Google C++ Testing Framework is used to do so. Apart from elementary unit testing, the VM is also tested as a whole by running predefined programs on it and checking the results.

Not only single VMs are tested, there is also a simulator which can run hundreds of VMs at once and simulate their communication. Simple algorithms like a gradient effect are run on the entire network, and the result is checked.

4 Evaluation

In this section, we reflect upon the design requirements from section 1.3. Comparisons are made against the Proto virtual machine, because that is (as far as I know) the virtual machine with goals closest to those of the Delft VM.

4.1 Functionality

Completeness All programs which can be expressed in Proto or in Proto Bytecode can be expressed in Delft VM instructions. The main differences are the way one can loop over the neighbour list, and whether vectors can be nested.

In the Proto virtual machine, there exists a single instruction to loop over the neighbour list. It requires a function (pointer) as parameter, which it will call for every neighbour to get the data from that neighbour. Another function, which was also given as parameter, will be called repeatedly to combine two results into one. Such a function could for example sum the results. The instruction then returns the final result.

In the Delft VM, one can loop 'manually' over the neighbour list using the `nbr-` instructions. In the loop, two functions can be called to simulate the behaviour of the Proto instruction. However, since those functions are usually as simple as a single instruction (such as getting a public neighbour value with the `nbr-mem` instruction, and summing it with the `add` instruction), they can be inlined, which saves code compared to having two separate functions and function calls.

In Proto, vectors can be nested as if they were tuples. The Proto virtual machine natively supports this, which makes it relatively easy for the compiler. The Delft VM does not support nested vectors, but that does not mean that such programs cannot be expressed in the Delft VM instruction set. The compiler could 'flatten' the vectors. That is, it could turn a 'vector' containing two vectors of three elements into a single vector with six elements. This operation is certainly not trivial, especially when the size of the contained vectors changes at run time, but it is possible. So by not supporting nested vectors, the virtual machine implementation can be simpler in exchange for a more complex compiler.

Small Programs Most Proto programs can be expressed in a shorter way in Delft VM instructions than in Proto virtual machine instructions. (Which is interesting, since the Proto virtual machine is specifically designed for Proto, whereas the Delft VM is not.) This is due to the following differences:

- As already mentioned earlier in this section, the Proto virtual machine has instructions that take a function (pointer) as parameter. All the functionality of those instructions are split over multiple instructions in the Delft VM, like the `nbr-` instructions. This usually saves space, since no separate functions have to be defined and called.
- Function definitions are simpler and shorter. In the Proto virtual machine, a function has to be defined by a 'DEF FUN' instruction, which takes the length of the function body in bytes as parameter. It basically stores the function (pointer) on the global

list of functions. It can then later be referred to by its index in that list. In the Delft VM, all one has to do is add a `ret` instruction to a list of instructions to turn it into a function. It can be referred to from the rest of the program by using its (byte) address relative to the first instruction of the program.

– The Proto virtual machine has two stacks that can hold regular values: The execution stack and the environment stack. Values can me moved from the execution stack to the environment stack, and copied back. When multiple copies of a value are needed, a value is moved to the environment stack, such that whenever a copy is needed it can be copied back. The Delft VM does not have or require such an 'environment stack', since it has more advanced instructions for manipulating the stack, including ones to copy data from other places on the same stack. This usually results in less copying data back and forth, and thus less instructions.

Table 2 shows implementations in both Delft VM and Proto instructions of the same simple program for comparison.

4.2 Implementation

Portability The virtual machine has successfully been ported to the ARM Cortex M0, Atmel AVR, x86, and AMD64 architectures.

The Delft VM uses very little of the standard library. What is used, is the C math support library (for functions like `sin` and `log`), `operator new`, and `operator delete`. The latter two are not provided by all platforms, as some only have a C support library (such as most AVR toolchains). In those cases, they can be defined trivially by forwarding to the C `malloc` and `free` functions.

The Delft VM is used as a library. To run it, you need to provide your own `main` function, which creates a `Vm::Machine` object. This object has member functions to load a program, run it, and inspect its state. On creation, it requires a `PlatformSpecifics` object, which contains function pointers to the platform specific implementations of functions like sending a message, reading out a sensor value, etc.

To run the Delft VM on the AVR platform, no code had to be changed of the Delft VM. Apart from the platform specific functionality and the `main` function, only `operator new` and `operator delete` needed to be defined.

Program Memory Although the size of the virtual machines are hard to compare, since it depends a lot on the target platform and compiler settings, I think a fair comparison can be made against the DelftProto VM since it is ported to the same platforms, and is written in the same language in a similar style as the Delft VM (since I wrote both).

For the AVR platform, the complete DelftProto VM (but without any advanced platform-specific code like communication drivers), takes about 32KiB. The DelftVM, for the same platform, results in 20KiB. Most of the space is taken by code to handle floating point values since this platform has, like most embedded platforms, no native support for floating point numbers.

Table 2: Implementaions of a simple program that calculates the distance to the closest node with a non-zero value for sensor 1. The Proto program defines a few helper functions, as some instructions (like `init-feedback`) take a function as argument. Also, values are copied to and from the environment stack (that is what the `let` and `ref` instructions do, respectively) in order to use them multiple times and access the arguments in a function.

(a) The program in Proto instructions (60 bytes)

```
def-fun 2
    inf
    ret
def-fun 6
    ref 1
    ref 0
    min
    ret
def-fun 14
    nbr-range
    lit 0
    eq
    if 6
    ref 0
    nbr-range
    add
    jmp 1
    inf
    ret
def-fun 21
    glo-ref 0
    init-feedback 0
    let 1
    ref 0
    lit 1
    sense
    lit 0
    glo-ref 1
    glo-ref 2
    ref 0
    fold-hood-plus 0
    mux
    feedback 0
    let 1
    ref 0
    pop-let 2
    ret
```

(b) The program in Delft VM instructions (23 bytes)

```
in 1
cjmp 14
lit inf
skip-nbr 11
nbr-get 1 0
nbr-prop 1
add
min
next-nbr -4
jmp 2
lit 0
dup
set 1 0
import
send
jmp -23
```

Memory Usage Memory is used more efficiently in the Delft VM than in the Proto virtual machine. This is because of the following reasons:

- Each value is 32 bits in the Delft VM, including its type information. In the Proto virtual machine, values are 32 bits as well, but the type information is stored in an additional byte, making it at least 40 bits per value. (Not taking any alignment into account, which would be another 8 bits for the ARM Cortex platform.)
- Vectors are stored entirely on the stack in the Delft VM, taking only 32 bits more than the sum of its contents. In the Proto virtual machine, there is also the overhead of the memory management, since all vectors are allocated on the heap.
- With the more advanced stack manipulation instructions, no separate 'environment stack' is required (as explained above). This results in not having to keep extra copies of values around on the environment stack.
- In the Proto virtual machine, values that are used for state ('feedback variables') and values used in communication ('exports') are separated. However, it is very common for a state variable to also be used in communication. In the Delft VM, the public memory can be used for state variables that are also used for communication. This avoids having to keep unnecessary copies.

5 Conclusions

As seen in Section 4, all the design goals have been met closely. It is easily portable, allows for a great variety of spatial programs, and has both an efficient design and implementation. The VM has been tested in perfect simulated environments, and seems to work perfect. However, it's usability in the real world must still be proven. Although it is likely that quite some changes and additions have to be made to adjust the VM to the requirements of real world applications, the VM has a well designed and implemented solid base to build from.

6 Future Work

Currently, a simulator exists which runs the virtual machine locally for every simulated node. However, the simulation is nowhere near realistic. All nodes run at exactly the same speed, and all communication is perfect. This is fine to test if the VM works properly, but is not that useful for testing spatial algorithms. A simulator which can realistically simulate (wireless) non-perfect communication would be of great use for spatial algorithm designers.

Another very important and obvious next step is the creation of a compiler. Such a compiler could be made for an already existing language, such as Proto. Then, already existing and tested algorithms can directly be compiled for and used on Delft VM spatial computers.

However, before such a compiler is made, it might be worth it to start with a simpler tool: An assembler. It is relatively easy (compared to most machine languages) to write Delft VM instructions by hand. However, it gets rather annoying and error-prone if those instructions also have to be encoded to raw bytes by hand. A simple assembler that supports using the instruction names, and labels for jumps, would already be very useful.

A Instruction Codes

The next page has an overview of all instruction codes.

All instruction codes are one byte. The 256 values are divided in 8 groups which are divided in 8 subgroups, such that when the value of the byte is written in octal, the first digit is the group number, the second is the subgroup number, and the third is the instruction number inside the subgroup.

All instruction codes are shown in the four tables below. Each group has its own table. The group number is shown in the top left corner of each table. The topmost row shows the subgroup number, and the leftmost column shows the instruction number. (So each column of instruction codes is a subgroup.)

For example: The instruction code for `tan` is 132 in octal.

0--	-0-	-1-	-2-	-3-	-4-	-5-	-6-	-7-
--0	nop	setup	---	send	nbr-reset	receive	in	---
--1	cjmp	---	---	send 1	nbr-skip	nbr-clear	in 1	---
--2	jmp	---	---	---	nbr-next	nbr-add	in 2	---
--3	ret	---	---	---	num-nbrs	nbr-del	in 3	---
--4	call	---	---	import	nbr-get 1	nbr-set	out	---
--5	calll	---	---	---	nbr-get	nbr-unset	out 1	---
--6	callp	---	---	---	nbr-isset	---	out 2	---
--7	calllp	---	---	---	nbr-prop	---	out 3	---

1--	-0-	-1-	-2-	-3-	-4-	-5-	-6-	-7-
--0	add	neg	exp	sin	neq	lneq	lit	scale
--1	sub	abs	log	cos	eq	leq	lit 0	len
--2	rsub	mux	pow	tan	lt	llt	lit 1	dot
--3	mul	---	rpow	asin	lte	llte	lit 2	cross
--4	div	trunc	sqrt	acos	gt	lgt	litf	cross-z
--5	rdiv	---	---	atan	gte	lgte	litf 0	---
--6	mod	---	---	atan2	min	lmin	litf 1	---
--7	rmod	---	---	ratan2	max	lmax	litf inf	---

2--	-0-	-1-	-2-	-3-	-4-	-5-	-6-	-7-
--0	copy	dup	pope	unpope	copy-down	vec	elt	pop
--1	copy 0	dup 1	pope 1	unpope 1	copy-down 0	vec 0	elt 0	spop
--2	copy 1	dup 2	pope 2	unpope 2	copy-down 1	vec 1	elt 1	concat
--3	copy 2	dup 3	pope 3	unpope 3	copy-down 2	vec 2	elt 2	---
--4	copy 3	dup 4	pope 4	unpope 4	copy-down 3	vec 3	elt 3	---
--5	copy 4	dup 5	pope 5	unpope 5	copy-down 4	vec 4	elt 4	---
--6	copy 5	dup 6	pope 6	unpope 6	copy-down 5	vec 5	elt 5	---
--7	copy 6	dup 7	pope 7	unpope 7	copy-down 6	vec 6	elt 6	---

3--	-0-	-1-	-2-	-3-	-4-	-5-	-6-	-7-
--0	get	get 1 0	set	set 1 0	unset	unset 1 0	isset	debug-a
--1	get 1	get 1 1	set 1	set 1 1	unset 1	unset 1 1	isset 0	debug-b
--2	get 2	get 1 2	set 2	set 1 2	unset 2	unset 1 2	isset 1	debug-c
--3	get 3	get 1 3	set 3	set 1 3	unset 3	unset 1 3	isset 2	debug-d
--4	gets	gets 1 0	sets	sets 1 0	unsets	unsets 1 0	issets	debug-e
--5	gets 1	gets 1 1	sets 1	sets 1 1	unsets 1	unsets 1 1	issets 0	debug-f
--6	gets 2	gets 1 2	sets 2	sets 1 2	unsets 2	unsets 1 2	issets 1	break
--7	gets 3	gets 1 3	sets 3	sets 1 3	unsets 3	unsets 1 3	issets 2	gbreak

B Instructions

This appendix describes all instructions.

In all cases when a vector is expected but only a single number is given, the number will be interpreted as a vector with a single element. Also, in all cases when a vector of a different length is expected than the one given, it is interpreted as truncated or filled with a sufficient number of zeros.

B.1 Control Flow

`nop` does nothing.

`cjmp` a takes a number from the stack and adds a to the instruction pointer if the number is not zero.

`jmp` a adds a to the instruction pointer.

`ret` pops an element from the call stack, and (re)sets the instruction pointer and memory offsets to the values in that element.

`call` a pushes an element on the call stack containing the value of the instruction pointer and the memory offsets, and then adds a to the instruction pointer.

`calll` l a does the same as `call` a, but adds l to the local memory offset afterwards.

`callp` p a does the same as `call` a, but adds p to the public memory offset afterwards.

`calllp` l p a does the same as `calll` l a, but adds p to the public memory offset afterwards.

B.2 Virtual Machine Setup

`setup` s c l p resets the virtual machine, and makes sure the stack has space for s elements, the call stack for c elements, the local memory for l elements, and the public memory for p elements.

B.3 Time

`time` pushes the time (in seconds) (since the last reset) on the stack.

B.4 Neighbours

`nbr-reset` resets the neighbour pointer to point to the 'self' node again.

`nbr-skip` a advances the neighbour pointer to the next neighbour, and adds a to the instruction pointer if this results in it pointing to the 'self' node again.

`nbr-next` a advances the neighbour pointer to the next neighbour, and adds a to the instruction pointer if this does not result in it pointing to the 'self' node again.

`num-nbrs` pushes the number of neighbours in the current neighbour list on the stack as an integer.

`nbr-get` n i reads n elements starting from index i (taking the public memory offset into account) from the public memory of the current neighbour. The result is pushed on the stack as a single number if $n = 1$, and as a vector of length n in all other cases.

`nbr-prop` n i reads n elements starting from index i from the properties of the current neighbour. The result is pushed on the stack as a single number if $n = 1$, and as a vector of length n in all other cases.

B.5 Communication

`send` n i broadcasts a message containing n elements starting from index i (taking the memory offset into account) of the public memory.

`import` Processes all messages in the incoming queue and updates the neighbour list accordingly.

B.6 Input and Output

`in` n i reads input(s) $i, \ldots, i + n - 1$. The result is pushed on the stack as a single number if $n = 1$, and as a vector of length n in all other cases.

`out` n i writes to output(s) $i, \ldots, i + n - 1$. The values to write are taken from the vector of length n popped from the stack.

B.7 Mathematical Operations

All of the instructions in this section pop their operands from the stack, and push the result back. When applied to vectors, the operation is applied pointwise. (For example, adding [1 2 3] and [2 1 1] gives [3 3 4] as result.)

`add` adds two numbers.

`sub` subtracts the first number from the second one.

`rsub` subtracts the second number from the first one.

`mul` multiplies two numbers.

`div` divides the first number by the second one.

`rdiv` divides the second number by the first one.

`mod` calculates the remainder of the first number divided by the second one.

`rmod` calculates the remainder of the second number divided by the first one.

`neg` negates a number.

`abs` gives the absolute value of a number.

`trunc` truncates the floating point number. (Rounding towards zero.)

`exp` calculates e^x for a number x.

`log` calculates the natural logarithm of a number.

`pow` raises the first number to the power given by the second number.

`rpow` raises the second number to the power given by the first number.

`sqrt` gives the square root of a number.

`sin` gives the sine of a number.

`cos` gives the cosine of a number.

`tan` gives the tangent of a number.

`asin` gives the arc sine of a number.

`acos` gives the arc cosine of a number.

`atan` gives the arc tangent of a number.

`atan2` gives the arc tangent of the ratio of two numbers representing the y and x coordinates respectively. The result is corrected to be in the right quadrant.

`ratan2` does the same as `atan2`, but takes the operands in reverse order.

`max` gives the maximum of two numbers.

`min` gives the minimum of two numbers.

B.7.1 Comparison

All comparison instructions push the integer 1 on the stack if the comparision equation was true, or the integer 0 if it was not.

`neq` checks if two numbers are unequal.

`eq` checks if two numbers are equal.

`lt` checks if a number is less than another.

`lte` checks if a number is less than or equal to another

`gt` checks if a number is greater than another.

`gte` checks if a number is greater than or equal to another

B.8 Lexicographical Comparison

The following comparison instructions pop two vectors from the stack, and push the integer 1 back if the comparision equation was true, or the integer 0 if it was not.

`lneq` checks if two vectors are unequal.

`leq` checks if two vectors are equal.

`llt` checks if a vectors is lexicograhically less than another.

`llte` checks if a vector is lexicograhically less than or equal to another

`lgt` checks if a vectors is lexicograhically greater than another.

`lgte` checks if a vector is lexicograhically greater than or equal to another

The next two instructions use lexicographical comparison to determine their result.

`max` gives the maximum of two vectors.

`min` gives the minimum of two vectors.

B.9 Literal Numbers

`lit` v pushes v on the stack as an integer.

`litf` v pushes v on the stack as a floating point number. The parameter to this function must already be encoded as a IEEE754 binary32 floating point value.

B.10 Vector Math

`scale` gives a vector with the contents of the given vector multiplied by the given number.

`len` gives the length of the given vector.

`dot` calculates the dot product of two vectors.

`cross` gives the cross product of two three dimensional vectors.

`cross-z` gives the third component of the cross product of two three dimensional vectors.

B.11 Stack Operations

`copy` n pushes a copy of the element with distance n from the top of the stack. If the element represents (a referen to) a vector, only a vector reference is pushed.

`dup` n pushes copies of the top n values (taking vectors into account) on the top of the stack.

`pope` n pops n elements from the stack. (Effectively subtracts n from the stack pointer.)

`unpope` n 'unpops' (Effectively adds n to the stack pointer.)

`copy-down` n pushes a copy of the value n elements above the stack top. If the value is a (reference to) a vector (above the stack top), the vector including its contents will be placed on top of the stack. (If it is a reference to a vector below the stack top, only a reference is pushed.)

`vec` n pushes a 'vector header' with value n on top of the stack. Effectively turning the top n elements into a vector of length n.

`elt` n pops (refernce to) a vector from the stack, and pushes element n from it back.

`pop` pops a value (taking vectors into account) from the stack.

`spop` pops the second value (taking vectors into account) from the stack. The first value is moved down so it is still at the top of the stack.

`concat` turns the two vectors at the top of the stack into one by concatenating them. Note that since single numbers will be seen as vectors with a single element, this can be used to append a number to a vector to make it grow.

B.12 Memory

`get(p)` n i gets n elements starting from index i (taking the memory offset into account) from the local (public) memory. The result is pushed on the stack as a single number if $n = 1$, and as a vector of length n in all other cases.

`set(p)` n i sets n elements starting from index i (taking the memory offset into account) in the local (public) memory. The values to write are taken from the vector of length n popped from the stack.

`unset(p)` n i resets n elements starting from index i (taking the memory offset into account) in the local (public) memory to the special 'not set' value.

`isset(p)` i pushes the integer 0 on the stack when the local (public) memory at index i (taking the memory offset into account) is set to the special 'not set' value, or the integer 1 otherwise.