

# The Ngaro Virtual Machine

## Abstract

Ngaro is a minimalistic virtual machine emulating a dual stack computer with a simple instruction set and a few basic I/O devices.

At present, there are full implementations in C, Go, Python, and Ruby, and partial implementations in C#, Forth, Common Lisp, Lua, JavaScript, and Java.

## Memory Access

Ngaro provides a memory space consisting of 32-bit, signed integer values. The first address is mapped to zero, with subsequent values following in a strictly linear fashion.

Each addressable 32-bit unit is called a *cell*. There is no support in the instruction set for accessing values larger or smaller than a single cell.

## Registers

Ngaro exposes no registers directly.

Internally, it may have registers for the *instruction pointer*, *data stack pointer*, and *address stack pointer*.

## Stacks

Ngaro emulates two LIFO stacks. The primary one is the *data stack*, and serves as a place for holding data, and passing data between instructions.

The other is the *address stack*, which is used to hold return addresses from calls, and can be used to temporarily hold small amounts of data.

The current depth of a stack can be determined by using the I/O ports to query the VM.

A standard implementation will provide at least 128 cells for data stack, and 1024 for the address stack.

## Image Files

On startup, Ngaro will load an *image file*. This is a flat, linear array of signed integer values. On disk, images are stored in little endian format, but the VM may convert them to big endian internally.

The first value loaded is mapped to address zero, and subsequent values are loaded to sequential addresses in the Ngaro memory space.

An image file does not contain copies of the stacks or any internal registers used by the VM.

## I/O Ports

Interfacing with the underlying OS and hardware devices is done by reading and writing I/O ports.

Ports 0 - 12 are reserved for current and future standard I/O devices. A VM may provide non-standard devices on port numbers above or below this range.

Port	Used For
0	Triggering I/O events
1	Keyboard
2	Text Output

3	Force Video Update
4	File I/O
5	Querying the VM

## Port 0: Wait for Hardware Event

This is used to determine if Ngaro should enter a wait for hardware event loop. It should be set to 0, then use the WAIT instruction.

Usage Example:

```
#0 #0 out, wait,
```

## Port 1: Read from the Keyboard

To read a value from the keyboard, set port 1 to 1, wait for an I/O event, then read the key value from port 1.

Usage Example:

```
#1 #1 out,  
#0 #0 out, wait,  
#1 in,
```

## Port 2: Character Generator

Ngaro provides a hardware character generator. This takes data off the stack, so you should make sure the character value is *on the stack* before using it. To use this, write the value 1 to port 2 and wait for an I/O event.

The VM is expected to clear the screen when a negative value is passed.

Usage Example:

```
#98 #1 #2 out,  
#0 #0 out, wait,
```

## Port 3: Force Video Update

To help improve performance, an Ngaro implementation is permitted to cache output and update the display periodically. For implementations that do this, setting this port will force an immediate update.

An implementation that caches output must check this routinely, and respond immediately. No waiting for an I/O event is necessary for this.

Example:

```
#0 #3 out,
```

## Port 4: File Operations

If your Ngaro implementation allows saving images, you can use this port to do so. To save, set port 4 to 1 and *wait*.

Other operations can be done using negative values, if the VM supports this. To use these, setup the stack, write the operation code to port 4, *wait*, then read the value back in from port 4.

Op	Takes	Returns	Notes
-1	filename, mode	handle	Open a file
-2	handle	flag	Read a byte from a file
-3	character, handle	flag	Write a byte to a file
-4	handle	flag	Close a file
-5	handle	offset	Return current location in file
-6	offset, handle	flag	Seek a new location in file
-7	handle	size	Return the size of a file
-8	filename	flag	Delete a file.

Valid modes for opening files are:

Value	Used For	Create if not existing?
0	Open file for reading	No
1	Open file for writing	Yes
2	Open file for append	Yes
3	Open file for modification	No

Reading and modification should *not* create a file if none exist. Writing and append modes *should* create a file if it does not exist.

The write mode should create a new file, removing the contents of existing files with the same name.

The append mode should set the file read/write position to the end of the file.

If opening fails, the returned handle should be zero. Any non-zero handle is considered valid.

When closing a valid handle, *close* should return zero.

The *write* operation should return a value of 1. Any other value indicates an error.

The *delete* operation should return -1 if the file is deleted, or 0 if the deletion fails.

## Port 5: Queries Into the VM Devices

Set port 5 to one of the following values; wait; then read the result back.

value	returns
-1	Memory Size
-2	Does a Canvas device exist?
-3	Canvas Width
-4	Canvas Height
-5	Data Stack Depth
-6	Address Stack Depth
-7	Does a Mouse device exist?
-8	Current time (in seconds, Unix-style)
-9	Exit the VM
-10	Query for an environment variable

-11	Console Width
-12	Console Height
-13	Number of bits per cell
-14	0 for little endian, 1 for big endian

At a minimum, an implementation must provide support for -1, -5, -6, -8, and -9.

For -10, the application must provide a buffer address on the stack, and a pointer to a string. The VM should search the system environment for the string and copy its value to the application memory, starting at the provided buffer address. If an environment variable is not found, the VM should store a value of zero in the provided buffer address.

For -13, if the returned value is zero, the image can assume a 32-bit environment.

For -14, if the VM is using big endian internally, this should return a value of 1.

## Port 6: Canvas

Some Ngaro implementations allow for drawing to a *canvas* device. Setup the data stack as shown in the table, and write the appropriate values to port 6.

*This device is optional.*

value	stack	action performed
1	n-	set color for drawing operations
2	xy-	draw a pixel at coordinates x, y
3	xyhw-	draw a rectangle of specified width (w) and height (h). The top corner is denoted by the x, y pair
4	xyhw-	draw a filled rectangle of specified width (w) and height (h). The top corner is denoted by the x, y pair
5	xyh-	draw a vertical line of height (h) starting at x, y
6	xyw-	draw a horizontal line of width (w) starting at x, y
7	xyw-	draw a circle of width (w) starting at x, y
8	xyw-	draw a filled circle of width (w) starting at x, y

For setting colors, the following values are guaranteed safe:

code	name
0	black
1	dark blue
2	dark green
3	dark cyan
4	dark red
5	purple
6	brown
7	dark gray
8	gray
9	blue

10	green
11	cyan
12	red
13	magenta
14	yellow
15	white

Additional colors may be supported, but are not guaranteed to exist.

## Port 7: Mouse

Set port 7 to one of the following values and wait. The results will be pushed to the data stack.

*This device is optional.*

value	returns
1	Mouse X and Y coordinates. Y will be on TOS when done. X will be NOS.
2	Is mouse button pressed? 0 = false, non-zero is true. True values <i>may</i> indicate the button being pressed, but this is not required.

## Instruction Set

One instruction per memory location. Instructions with an x in the A column take an additional value in the following memory location.

All opcode numbers are listed in decimal. Stack diagrams are for the data stack.

opcode	name	assembler	A	stack
0	NOP	nop,		-
1	LIT	lit,	x	-n
2	DUP	dup,		n-nn
3	DROP	drop,		n-
4	SWAP	swap,		xy-yx
5	PUSH	push,		n-
6	POP	pop,		-n
7	LOOP	loop,	x	n-n
8	JUMP	jump,	x	-
9	RETURN	;;		-
10	LT_JUMP	<jump,	x	xy-
11	GT_JUMP	>jump,	x	xy-
12	NE_JUMP	!jump,	x	xy-
13	EQ_JUMP	=jump,	x	xy-
14	FETCH	@,		a-n
15	STORE	!,		na-
16	ADD	+,		xy-z

17	SUBTRACT	-,		xy-z
18	MULTIPLY	*,		xy-z
19	DIVMOD	/mod,		xy-rq
20	AND	and,		xy-z
21	OR	or,		xy-z
22	XOR	xor,		xy-z
23	SHL	<<,		xy-z
24	SHR	>>,		xy-z
25	ZERO_EXIT	0;		n-?
26	INC	1+,		x-y
27	DEC	1-,		x-y
28	IN	in,		p-n
29	OUT	out,		np-
30	WAIT	wait,		-

## Instruction Processing

The instruction pointer is incremented, then the opcode at the current address is handled. Execution ends when the instruction pointer is greater than the end of the simulated memory space.

A psuedocode in Retro:

```
-1 !ip
[ ip ++ processOpcode @ip 1000000 < ] while
```

And in Lua:

```
ip = 0
while ip < 1000000 do
  processOpcode()
  ip = ip + 1
end
```

## The Instructions

### **Opcode 0: NOP**

Does nothing.

### **Opcode 1: LIT**

Push the value in the following memory location to the data stack. Advances the instruction pointer by one.

In memory this might appear as:

```
0000 LIT
0001 101
```

After LIT executes, the IP would be set at 0001, and the top item on the data stack would be 101.

**Opcode 2: DUP**

Make a duplicate copy of the top item on the data stack and push the copy to the data stack.

before	after
1	1
2	1
3	2
	3

**Opcode 3: DROP**

Remove the top item from the data stack.

before	after
1	2
2	3
3	

**Opcode 4: SWAP**

Remove the top two items from the stack, and push them back in the reverse order.

before	after
1	2
2	1
3	3

**Opcode 5: PUSH**

Remove the top item from the data stack, and push it to the address stack.

before	after
1	2
2	3
3	

**Opcode 6: POP**

Remove the top item from the address stack, and push it to the data stack.

before	after
--------	-------

2	1
3	2
	3

### **Opcode 7: LOOP**

Decrement the top value on the stack and advance the instruction pointer. If the top item on the stack is greater than zero, jump to the address following the LOOP instruction, otherwise discard the top item on the stack and continue execution normally.

### **Opcode 8: JUMP**

Set the instruction pointer to the address in the cell following the JUMP instruction.

This instruction needs to decrement the requested address by one to account for the increment of the instruction pointer by the opcode process cycle. E.g., if the jump target is 1234, JUMP needs to set the instruction pointer to 1233.

To improve performance, this instruction may skip leading NOP's at the destination address.

### **Opcode 9: RETURN**

Return from a call to a subroutine. This will pop the return address from the address stack, and set the instruction pointer to it.

### **Opcode 10: LT\_JUMP**

Increment the instruction pointer.

Pop the top two values from the stack. If the first stack item is less than the second item, set the instruction pointer to the address stored at the memory location following this instruction. If not, continue execution.

This instruction needs to decrement the requested address by one to account for the increment of the instruction pointer by the opcode process cycle. E.g., if the jump target is 1234, the instruction pointer should be set to 1233.

In memory, this will be stored as:

```
0000 LT_JUMP
0001 destination
```

before	after
1	
2	

### **Opcode 11: GT\_JUMP**

Increment the instruction pointer.

Pop the top two values from the stack. If the first stack item is greater than the second item, set the instruction pointer to the address stored at the memory location following this instruction. If not, continue execution.

This instruction needs to decrement the requested address by one to account for the increment of the instruction pointer by the opcode process cycle. E.g., if the jump target is 1234, the instruction pointer

should be set to 1233.

In memory, this will be stored as:

```
0000 GT_JUMP
0001 destination
```

before	after
1	
2	

### Opcode 12: NE\_JUMP

Increment the instruction pointer.

Pop the top two values from the stack. If the first stack item is not equal to the second item, set the instruction pointer to the address stored at the memory location following this instruction. If not, continue execution.

This instruction needs to decrement the requested address by one to account for the increment of the instruction pointer by the opcode process cycle. E.g., if the jump target is 1234, the instruction pointer should be set to 1233.

In memory, this will be stored as:

```
0000 NE_JUMP
0001 destination
```

before	after
1	
2	

### Opcode 13: EQ\_JUMP

Increment the instruction pointer.

Pop the top two values from the stack. If the first stack item is equal to the second item, set the instruction pointer to the address stored at the memory location following this instruction. If not, continue execution.

This instruction needs to decrement the requested address by one to account for the increment of the instruction pointer by the opcode process cycle. E.g., if the jump target is 1234, the instruction pointer should be set to 1233.

In memory, this will be stored as:

```
0000 EQ_JUMP
0001 destination
```

before	after
1	
2	

### **Opcode 14: FETCH**

Remove the top item from the data stack. Lookup the value stored in the memory address this value points to, and push the value read to the data stack.

Assuming that memory at 1234 contains 45:

before	after
1234	45

### **Opcode 15: STORE**

Take two values from the stack. The top item will be a destination address, and the second will be a value. Modify the contents of the specified memory address to be equal to the value.

before	after
1	
2	

In this, 1 would be the address, and 2 would be the value to store there.

### **Opcode 16: ADD**

Take two values from the data stack, add them together, and push the results to the data stack.

before	after
1	3
2	

### **Opcode 17: SUBTRACT**

Take two values from the data stack. Subtract the top item from the second item, and push the results back to the data stack.

before	after
4	5
9	

### **Opcode 18: MULTIPLY**

Take two values from the data stack. Multiply them and push the results back to the data stack.

before	after
2	6
3	

### Opcode 19: DIVMOD

Take two values from the data stack. The top item is the divisor, and the second item is the dividend. Perform the division, and push the quotient and remainder to the stack. After execution the quotient should be on top, with the remainder below it.

*Division is symmetric, not floored.*

before	after
2	2
5	1

### Opcode 20: AND

Remove the top two items on the data stack. Perform a bitwise AND operation and push the result back to the data stack.

before	after
-1	-1
-1	

before	after
0	0
-1	

before	after
0	0
0	

### Opcode 21: OR

Remove the top two items on the data stack. Perform a bitwise OR operation and push the result back to the data stack.

before	after
-1	-1
-1	

before	after
0	-1
-1	

before	after
--------	-------

0	0
0	

**Opcode 22: XOR**

Remove the top two items on the data stack. Perform a bitwise XOR operation and push the result back to the data stack.

before	after
-1	0
-1	

before	after
0	-1
-1	

before	after
0	0
0	

**Opcode 23: SHL**

Take two values from the data stack. Perform a bitwise left shift on the second item by the number of bits specified by the top item. Push the results back to the data stack.

The values in these tables are in binary.

before	after
11	111000111000
111000111	

The results of a negative shift are implementation specific.

**Opcode 24: SHR**

Take two values from the data stack. Perform a bitwise right shift on the second item by the number of bits specified by the top item. Push the results back to the data stack.

The values in these tables are in binary.

before	after
11	111000111
111000111000	

The results of a negative shift are implementation specific.

### Opcode 25: ZERO\_EXIT

If the top item on the stack is zero, remove and discard it, then pop the top item from the address stack and set the instruction pointer to it.

If the top item is not zero, leave it alone and do nothing.

before	after
1	1
0	

### Opcode 26: INC

Increase the value on the top of the stack by one.

before	after
2	3

### Opcode 27: DEC

Decrease the value on the top of the stack by one.

before	after
2	1

### Opcode 28: IN

Take a value from the data stack. Read the value stored in the I/O port corresponding to the value read from the stack, and push this value to the data stack.

After reading, set the value of the port read to zero.

before	after
1	?

The value returned will vary depending on the I/O device subsystem and specific port requested.

### Opcode 29: OUT

Take two values off the data stack. The top value will be an I/O port number, and the second will be a value. Store the value in the I/O port specified.

before	after
2	
1	

With the values in this table, port 2 would be set to a value of 1.

### ***Opcode 30: WAIT***

Run the simulated device handler. Before calling this, the code being run should set I/O port 0 to 0 to ensure that a request is actually handled. If no requests are pending (based on the values written to the ports previously), continue execution as normal.

### ***Opcodes Above 30***

Any opcode above 30 is treated as an *implicit call*.

The VM will push the current value of the instruction pointer to the address stack and set the instruction pointer to the value of the opcode. Note that like jumps, the VM must decrement this value by one to account for the increment that will happen before the next instruction is processed.