EZVM SPEC

Eric Buss

A specification for the Eazy Virtual Machine. This spec **is not** binding, complete, or correct, it exists to assist development, not hinder it.

Table of Contents

[Architecture 3](#_Toc45639638)

[Tables 3](#_Toc45639639)

[Stacks 3](#_Toc45639640)

[Registers 3](#_Toc45639641)

[Statuses 3](#_Toc45639642)

[Variables 3](#_Toc45639643)

[Execution 5](#_Toc45639644)

[Import and Export 6](#_Toc45639645)

[Runtime Patching 6](#_Toc45639646)

[Extensions 6](#_Toc45639647)

# Introduction

Adsf

# Intermediate Representation

The Eazy Intermediate Representation (EZIR) is a data scheme for representing an executable Eazy program such that

* Transformation into byte-code is straightforward
* Data manipulation using host language tools is straightforward (eg. no raw bytes)
* Interpretation pre-compilation is viable

EZIR represents code as a list of simple operations, very much like opcodes, with operands. It differs from Bytecode in that these structures explicitly exist as actual lists with arbitrarily sized arguments. Here is a simple EZIR sample:

|  |
| --- |
| [  [ MODULE, “0.0.1”, “src/sample.ez”, 1, [  [ EXPORT, V[0] ],  ],  ] |

Several notes on this format:

* This format uses JavaScript as its base syntax
* Operations like **MODULE** and **EXPORT** are integer constants corresponding to opcodes.
* Operands like **V[0]** are integer constants corresponding to variable offsets
* This format is not explicitly serializable, ie. it may contain types that are not JSON compatible

## Types

Operands of EZIR are limited to specific types. These types may or may not also exist in the Easy language itself. The following table defines these types:

|  |  |  |
| --- | --- | --- |
| Type | EZ Type (Y/N) | Description |
| NothingType | Y | A type whose only valid value is **Nothing** |
| Boolean | Y | A type whose only valid values are **True** and **False** |
| Number | Y | A numeric type (implemented as a 64bit float) |
| String | Y | S string data type |
| List | Y | A type representing an ordered collection of elements |
| Map | Y | A type representing an unordered set of associated elements |
| Function | Y | A type representing a function |
| Type | Y | A type representing the other types, eg. List is of type Type |
| Reference | N | A type that provides a mutable reference to another type |
| Variable | N | A type representing a variable, eg. **V[0],** **ARGS**, or **CAPS.** The notation Variable[Type] indicates that the instruction expects an instance of Type at that variable. |
| Collection | N | A String, List, or Map |
| Value | N | A value type, one of NothingType, Boolean, Number, Sttring, List, Type, Map, or Reference |

## Structure

There are two structural operations in EZIR, modules and functions. All other constructs are flattened (eg. conditionals, and loops). At the top level EZIR must be contained in a module operation. Both modules and functions take their internal code as their final element.

## Operations

EZIR defines the following operations.

### Control Operations

[**MODULE**, version: String, path: String, locals: Number operations: List]

Defines a module from **path**. The number of available local variables is defined by **locals.** All operations of the module are contained in **operations**.

[**HALT**]

Indicates the completion of execution.

[**LABEL,** id: Number]

Does nothing but marks a location in the code that can be jumped to by referencing **id**.

[**GOTO,** id: Number]

Unconditional jump to the operation after the label marked with **id**.

[**GOIF,** condition: Variable[Boolean], id: Number]

Jump to the operation after the label marked with **id** if the value at **condition** is **True.** Throw a **TypeError** if the value at **condition** is not a Boolean.

[**IMPORT,** path: Variable[String], dst: Variable],

Import the module at **path’s** export value into **dst.**

[**EXPORT,** src: Variable],

Export the value at **src** from this module. This effectively terminates module execution.

### Arithmetic Operations

[**NEG,** src: Variable[Number], dst: Variable]

Negate the numeric value at **src** and store the result in **dst**. Throw a **TypeError** if the value at **src** is not a Number.

[**ADD,** left: Variable[Number], right: Variable[Number], dst: Variable]

Add the value at **left** to the value at **right** and store in **dst**. Throw a **TypeError** if either **left** or **right** are not a Number.

[**SUB,** left: Variable[Number], right: Variable[Number], dst: Variable]

Subtract the value at **left** to the value at **right** and store in **dst**. Throw a **TypeError** if either **left** or **right** are not a Number.

[**MUL,** left: Variable[Number], right: Variable[Number], dst: Variable]

Multiply the value at **left** to the value at **right** and store in **dst**. Throw a **TypeError** if either **left** or **right** are not a Number.

[**DIV,** left: Variable[Number], right: Variable[Number], dst: Variable]

Divide the value at **left** to the value at **right** and store in **dst**. Throw a **TypeError** if either **left** or **right** are not a Number.

[**MOD**, left: Variable[Number], right: Variable[Number], dst: Variable]

Store the remainder of dividing **left** by **right** in **dst**. Throw a **TypeError** if either **left** or **right** are not a Number.

[**POW,** left: Variable[Number], right: Variable[Number], dst: Variable]

Store **left** to the power of **right** in **dst**. Throw a **TypeError** if either **left** or **right** are not a Number.

[**FLOOR,** src: Variable[Number], dst: Variable]

Store the floor of **src** in **dst.** Throw a **TypeError** if **src** is not a Number.

[**CEIL,** src: Variable[Number], dst: Variable]

Store the ceiling of **src** in **dst.** Throw a **TypeError** if **src** is not a Number.

### Bit-Operations

[**BAND,** left: Variable[Number], right: Variable[Number], dst: Variable]

Bitwise and the value at **left** to the value at **right** and store in **dst**. Throw a **TypeError** if either **left** or **right** are not a Number.

[**BOR,** left: Variable[Number], right: Variable[Number], dst: Variable]

Bitwise or the value at **left** to the value at **right** and store in **dst**. Throw a **TypeError** if either **left** or **right** are not a Number.

[**BXOR,** left: Variable[Number], right: Variable[Number], dst: Variable]

Bitwise exclusive-or the value at **left** to the value at **right** and store in **dst**. Throw a **TypeError** if either **left** or **right** are not a Number.

[**BNOT,** src: Variable[Number], dst: Variable]

Bitwise not the value in **src** and store in **dst**. Throw a **TypeError** if **src** is not a Number.

[**SHL,** src: Variable[Number], shift: Variable[Number], dst: Variable]

Shift the value at **src** left by the value at **shift** and store in **dst.** Throw a **TypeError** if either **src** or **shift** are not a Number.

[**ASHR,** src: Variable[Number], shift: Variable[Number], dst: Variable]

Arithmetically shift the value at **src** right by the value at **shift** and store in **dst.** Throw a **TypeError** if either **src** or **shift** are not a Number.

[**LSHR,** src: Variable[Number], shift: Variable[Number], dst: Variable]

Logically shift the value at **src** right by the value at **shift** and store in **dst.** Throw a **TypeError** if either **src** or **shift** are not a Number.

### Boolean and Conditional Operations

[**AND,** left: Variable[Boolean], right: Variable[Boolean], dst: Variable]

[**OR,** left: Variable[Boolean], right: Variable[Boolean], dst: Variable]

[**NOT,** src: Variable[Number], dst: Variable]

[**EQ,** left: Variable, right: Variable, dst: Variable]

[**IS,** left: Variable, right: Variable, dst: Variable]

[**IN,** val: Variable, col: Variable[Collection], dst: Variable]

[**LT**, left: Variable[Number], right: Variable[Number], dst: Variable]

[**LTE**, left: Variable[Number], right: Variable[Number], dst: Variable]

[**GT,** left: Variable[Number], right: Variable[Number], dst: Variable]

[**GTE,** left: Variable[Number], right: Variable[Number], dst: Variable]

### Variable Operations

[**MOVE,** src: Variable, dst: Variable]

[**CONST,** src: Value, dst: Variable]

### Type Operations

[**TYPE,** src: Variable, dst: Variable]

### Exception Operations

[**THROW,** src: Variable ]

[**TRY,** id: Number, dst: Variable]

[**SUCCEED**]

[**CAUGHT**]

### Collection Operations

[**COPY,** src: Variable, dst: Variable]

[**MERGE,** left: Variable[Collection], right: Variable[Collection], dst: Variable]

[**UPDATE,** left: Variable[Collection], right: Variable[Collection], dst: Variable]

[**GET,** key: Variable, col: Variable[Collection], dst: Variable]

[**SET,** key: Variable, val: Variable, col: Variable[Collection]]

[**COUNT,** col: Collection, dst: Variable]

[**SLICE,** start: Variable[Number], stop: Variable[Number], col: Variable[Collection], dst: Variable]

[**INDEX,** val: Variable, col: Variable[Collection], dst: Variable]

[**KEYS,** col: Variable[Collection], dst: Variable]

[**VALUES,** col: Variable[Collection], dst: Variable]

[**FREEZE,** col: Variable[Collection]]

### String Operations

[**UNICODE,** idx: Variable[Number], str: Variable[String], dst: Variable]

[**MATCH,** pat: Variable[String], str: Variable[String], dst: Variable]

[**SPLIT,** pat: Variable[String], str: Variable[String], dst: Variable]

[**JOIN,** pat: Variable[String], str: Variable[String], dst: Variable]

### Function Operations

[**FUNCTION,** id: Number, locals: Number, operations: List]

[**CLOSURE,** id: Number, caps: Variable, dst: Variabrle]

[**CALL,** closure: Variable[Function], args: Variable, dst: Variable]

[**TAILCALL,** closure: Variable[Function], args: Variable]

[**RETURN,** src: Variable]

### Extension Operations

[**EXTENSION,** name: String, dst: Variable]

### Debug Operations

[**BREAK**]

[**TRACE,** dst: Variable]

[**META,** src: Variable, dst: Variable]

# Architecture

## Stores

EZVM maintains several data stores while running. These stores have unique access patterns or unique update patterns. These stores are distinct from stacks which are typically used for computation, stores are used primarily for data lookup.

**Note:** max sizes indicated as blank are at the implementers distraction so long as they are “reasonably large”.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Store | Full Name | Type | Max Size | Description |
| CODE | Code | Byte Array |  | A table of memory containing the byte code of the program. |
| CONST | Constant Table | Variable List |  | An array of constant values that can be loaded into the program. |
| DLINE | Debug Line Table | Map |  | A map from code offset to line of source code. Details in **5.1** |
| DFUNC | Debug Function Table | Map |  | A map from code offset to function metadata. Details in **5.2** |
| SOURCE | Source Code | String |  | The program’s original source code |

## Tables

EZVM stores constant data in table. This data will only never grow or shrink during imports. Data from tables is only ever accessed by integer offset.

**Note:** max sizes indicated as blank are at the implementers discretion so long as they are “reasonably large”.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Table | Full Name | Type | Max Size | Description |
| CODE | Code | Int32 List |  | A table of memory containing the byte code of the program. |
| CONST | Constant Table | Variable List |  | An array of constant values that can be loaded into the program. |
| DLINE | Debug Line Table | Map |  | A map from code offset to line of source code. Details in **5.1** |
| DFUNC | Debug Function Table | Map |  | A map from code offset to function metadata. Details in **5.2** |
| SOURCE | Source Code | String |  | The program’s original source code |

## Stacks

EZVM stores dynamic data in stacks. This data will grow and shrink as the program executes. Data in stacks is only ever accessed by integer offset.

**Note:** max sizes indicated as blank are at the implementers discretion so long as they are “reasonably large”.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Stack | Full Name | Type | Max Size | Description |
| VARS | Variable Stack | Variable List |  | An array used to store variables. |
| CALLS | Call Stack | Int32 List |  | An array that stores function offsets, return offsets, and variables offsets when functions are called. |
| CATS | Catch Stack | Variable List |  | An array of error catchers that can respond to errors. Catchers are tuples of (address, **CSR**, **VSR**, count **VARS**) |

## Registers

EZVM has the following registers

|  |  |  |
| --- | --- | --- |
| Register | Full Name | Description |
| ISR | Instruction Register | The current offset into **CODE** where **CODE[ISR - 1]** is the currently executing instruction |
| VSR | Variable Stack Register | The current offset into **VARS** such that variable **V[0]** refers to **VARS[VSR + 0]** |
| CSR | Call Stack Register | The current offset into **CALLS** such that **CALLS[CSR – 1]** is the current function offset, **CALLS[CSR – 2]** is the return **ISR**, and **CALLS[CSR – 3]** is the return **VSR** |
| STR | Status Register | Indicates the status of the VM, details in **3.4** |

## Statuses

**STR** can be in any of the following states

|  |  |
| --- | --- |
| Status | Description |
| OK | Execution is proceeding as normal |
| HALTED | Execution has halted |
| WAIT | Execution is currently waiting on an asynchronous process |
| PAUSED | Execution has hit a breakpoint |
| ERROR | An error has been thrown and not been caught |

## Variables

All variables in EZVM are kept on **VARS**. They function similarly to both general purpose registers in register-based machine and the stack in a stack-based machine. Instructions encode **VARS** offsets which are used to operate on variables. Because every variable offset is encoded in a single byte the maximum registers that can be used is 256. By default functions are only allocated 2 special purpose variables. A function must expand **VARS** if more variables are needed. Any operations on unallocated variables is **undefined behaviour**.

|  |  |  |
| --- | --- | --- |
| Variable | Name | Description |
| V0 | ARGS | Within a function execution **ARGS** contains the list of arguments passed to that function. [1] |
| V1 | CAPS | Within a closure execution **CAPS** contains a list of captured values. |
| V2-255 | EXP0-254 | **Exp**anded or **exp**ression variables used for whatever the function needs. Limited by the size of **VARS**. A function can expand **VARS** with the **EXPAND** instruction. If a function is a Closure that captures variables from its environment then they will appear here in order of capture (**EXP0** contains the first capture, **EXP1** contains the second capture, etc.). **VARS** will be automatically extended to fit captured values. |

[1] Note **ARG** and **ARGS** are distinct constructs   
Instructions

Instructions are all 4 byte (32 bit) integers. The least significant byte (LSB) of all instructions are their opcode. The remaining 3 bytes are (potentially) operands. An instruction can take either

* A 2 byte **L** operand and/or a 1 byte **X** operand
* A 1 byte **X** operand and/or a 1 byte **Y** operand and/or a 1 byte **Z** operand

There are several limits on memory sizes due to operand sizes. Programs can be addressed by **K** allowing for a maximum code size of 16777216 instructions or 67108864 bytes (67 MB). Constants can be addressed by **L** allowing for a maximum of 65536 constants. Variables can addressed by **X, Y,** or **Z** allowing for a maximum of 256 variables per function.

When defining the operation of an instruction the following notation is used to refer to values

* **V[#]** refers to **VARS[VSR + #]**
* **C[#]** refers to **CONST[#]**
* **X** ← **Y** means to store the value of **Y** in the location of **X**

The following table lists the 81 instructions of the EZVM.

|  |  |  |  |
| --- | --- | --- | --- |
| Mnemonic | Operation | Instruction (0x) | Notes |
| **Control** | | | |
| HALT | **STR** ← **HALTED** | 00 00 00 00 |  |
| JUMP | **ISR** ← **ISR** +signed **L** | LL LL 00 01 |  |
| JUMP\_IF | if **V[X]** then **ISR** ← **ISR** + **L** | LL LL XX 02 |  |
| IMPORT | **V[Y]** ←import **V[X]** | 00 YY XX 0C |  |
| EXPORT | export **V[X]** | 00 00 XX 0D |  |
| **Arithmetic** | | | |
| ADD | **V[Z]** ← **V[X]** + **V[Y]** | ZZ YY XX 10 |  |
| SUB | **V[Z]** ← **V[X]** – **V[Y]** | ZZ YY XX 11 |  |
| MUL | **V[Z]** ← **V[X]** \* **V[Y]** | ZZ YY XX 12 |  |
| DIV | **V[Z]** ← **V[X]** / **V[Y]** | ZZ YY XX 13 |  |
| MOD | **V[Z]** ← **V[X]** % **V[Y]** | ZZ YY XX 14 |  |
| POW | **V[Z]** ← **V[X]** pow **V[Y]** | ZZ YY XX 15 |  |
| NEG | **V[Y]** ←-**V[X]** | 00 YY XX 16 |  |
| FLOOR | **V[Y]** ←floor **V[X]** | 00 YY XX 17 |  |
| CEIL | **V[Y]** ←ceil **V[X]** | 00 YY XX 18 |  |
| **Booleans and Conditions** | | | |
| AND | **V[Z]** ← **V[X]** and **V[Y]** | ZZ YY XX 20 |  |
| OR | **V[Z]** ← **V[X]** or **V[Y]** | ZZ YY XX 21 |  |
| NOT | **V[Y]** ←not **V[X]** | 00 YY XX 22 |  |
| EQ | **V[Z]** ← **V[X]** = **V[Y]** | ZZ YY XX 23 |  |
| IS | **V[Z]** ← **V[X]** is **V[Y]** | ZZ YY XX 24 |  |
| LT | **V[Z]** ← **V[X]** < **V[Y]** | ZZ YY XX 25 |  |
| GTE | **V[Z]** ← **V[X]** >= **V[Y]** | ZZ YY XX 26 |  |
| **Bit-oriented** | | | |
| BIT\_AND | **V[Z]** ← **V[X]** & **V[Y]** | ZZ YY XX 30 |  |
| BIT\_OR | **V[Z]** ← **V[X]** | **V[Y]** | ZZ YY XX 31 |  |
| BIT\_XOR | **V[Z]** ← **V[X]** ^ **V[Y]** | ZZ YY XX 32 |  |
| BIT\_NOT | **V[Y]** ← ~**V[X]** | 00 YY XX 33 |  |
| BIT\_SHL | **V[Z]** ← **V[X]** << **V[Y]** | ZZ YY XX 34 |  |
| BIT\_ASHR | **V[Z]** ← **V[X]** >> **V[Y]** | ZZ YY XX 35 |  |
| BIT\_LSHR | **V[Z]** ← **V[X]** >>> **V[Y]** | ZZ YY XX 36 |  |
| **Variable** | | | |
| MOVE | **V[Y]** ← **V[X]** | 00 YY XX 40 |  |
| COPY | **V[Y]** ← copy **V[X]** | 00 YY XX 42 |  |
| EXPAND | count **VARS** ← count **VARS** + **X** | 00 00 XX 43 |  |
| CONST | **V[X]** ← **C[L]** | LL LL XX 44 |  |
| **Type** | | | |
| TYPE | **V[Y]** ← type **V[X]** | 00 YY XX 45 |  |
| **Exceptions** | | | |
| THROW | If not Nothing **ERR** then  **ERR** ← wrap **ERR V[X]**  else  **ERR** ← **V[X]**  if not empty **TRYS** then  handler ← pop **TRYS**  **ISR** ← handler[0]  **CSR** ← handler[1]  **VSR** ← handler[2]  count **VARS** ← handler[3]  **V[**handler[4]**]** ← **ERR**  else  **STR** ← **ERROR** | 00 00 XX 50 |  |
| TRY | push (**ISR** + signed **L**, **CSR**, **VSR**, count **VARS, X**) **TRYS** | LL LL XX 52 |  |
| CAUGHT | **ERR** ← Nothing | 00 00 00 53 |  |
| **Collections** | | | |
| MERGE | **V[Z]** ← merge[EA[ **V[X] V[Y]** | ZZ YY XX 60 |  |
| UPDATE | **V[Z]** ← update **V[X] V[Y]** | ZZ YY XX 61 |  |
| GET | **V[Z]** ← **V[Y][V[X]]** | ZZ YY XX 62 |  |
| SET | **V[Y][V[X]]** ← **V[Z]** | ZZ YY XX 63 |  |
| PUSH | **V[Y][**count **V[Y]]** ← **V[X]** | 00 YY XX 64 |  |
| COUNT | **V[Y]** ← count **V[X]** | 00 YY XX 65 |  |
| TAKE | **V[Z]** ← **V[Y][:V[X]]** | ZZ YY XX 66 |  |
| DROP | **V[Z]** ← **V[Y][V[X]:]** | ZZ YY XX 67 |  |
| FILL | **V[Y][:]** ← **V[X]** | 00 YY XX 68 |  |
| REPEAT | **V[Z]** ← repeat **V[Y] V[X]** times | ZZ YY XX 69 |  |
| INDEX | **V[Z]** ← index **V[X]** in **V[Y]** | ZZ YY XX 6A |  |
| REVERSE | **V[Y]** ← reverse **V[X]** | 00 YY XX 6B |  |
| ENTRIES | **V[Y]** ← entries **V[X]** | 00 YY XX 6C |  |
| KEYS | **V[Y]** ← keys **V[X]** | 00 YY XX 6D |  |
| VALUES | **V[Y]** ← values **V[X]** | 00 YY XX 6E |  |
| FREEZE | freeze **V[X]** | 00 00 XX 6F |  |
| **Strings** | | | |
| UNICODE | **V[Z]** ← Unicode **V[Y][V[X]]** | ZZ YY XX 70 |  |
| MATCH | **V[Z]** ← matches of **V[Y]** in **V[X]** | ZZ YY XX 71 |  |
| SPLIT | **V[Z]** ← split **V[Y]** on **V[X]** | ZZ YY XX 72 |  |
| JOIN | **V[Z]** ← join **V[Y]** with **V[X]** | ZZ YY XX 73 |  |
| **Functions** | | | |
|  | tmp← **V[X]**  count **VARS** ← **VSR**  **ISR** ← **CALLS[CSR** ← **CSR** - 1**]**  **VSR** ← **CALLS[CSR**← **CSR** - 1**]**  **V[CSR** ← **CSR** – 1**]** ← tmp | 00 00 XX 81 |  |
| FUNCTION | **V[X]** ← **FUNCT[L]** | 00 YY XX 82 |  |
| EXTENSION | **V[Y]** ← **EXT[V[X]]** | 00 YY XX 82 |  |
| CLOSURE | **V[Z]** ← (**V[X][0], V[Y]**) | ZZ YY XX 83 |  |
| CALL | push **Z CALLS**  push **VSR CALLS**  push **ISR CALLS**  **VSR** ←count **VARS**  **ISR** ←**V[X][0]**  count **VARS** ← count **VARS** + 2  **ARGS** ← **V[Y]**  **CAPS** ← **V[X][1]** | ZZ YY XX 84 |  |
| TAIL\_CALL | **ARGS** ← **V[Y]**  count **VARS** ← **VSR** + 2  **ISR** ←**V[X][0]**  **CAPS** ← **V[X][1]** | 00 YY XX 85 |  |
| **Debug** | | | |
| BREAK | **STR** ← **PAUSED** | 00 00 00 F0 |  |
| STACK | **V[X]** ← **CALLS** | 00 00 XX F1 |  |
| META | **V[Y]** ← metadata **V[X]** | 00 YY XX F2 |  |

# Execution

## Import and Export

Asdf

Jump and const rewrite on export

## Runtime Patching

Asdf

## Extensions

Asdf

# Debug

## Line Numbers

Asdf

## Function Names

Asdf

## Variables

Asdf

# Source Format

Asdf

# Example Codegen

|  |  |
| --- | --- |
| Source | Code |
| var x = 1 + 1 | CONST 1 ACC  ADD ACC ACC ACC  HALT |
| if x > 0 then {  x = “positive”  } else {  x = “negative”  } | CONST 0 ARG  SKIP\_GT ACC ARG  JUMP .THEN  .ELSE  CONST “negative” ACC  JUMP .JOIN  .THEN  CONST “positive” ACC  .JOIN  HALT |

# Huge Problems (so far)

* The contagious closure, if I important a function and it’s a closure, how can I call it from another function without making that other function also a closure?
* Dynamic linking/importing how can I efficiently share code between two modules not compiled together/patched in at runtime (global variables may assist in this problem, but they are not a full on solution as imports within functions will still face the same problem). Simplifying solution: make all functions closures (uck -> how the hell would hot reloading work then???)
* Hot reloading – how can I hot reload a function, probably doable if the other problems are solved using DFUNC. Possibly add a layer of indirection to function via a function table? Would make hot reloading much easier – at least consider

Per module Function table

VM = init\_ezvm(options)

Module\_id = load\_module(VM, code)

Patch\_module(VM, module\_id, code\_patch)

Extension\_id = Register\_extension(VM, extension\_handlers)

Message sending as the core way to interact with a running program (program is essentially always a server)