WEBASSEMBLY illustrated

exploring some mental models and implementations

Takenobu T.

WIP

Rev. 0.00.0

NOTE

- Please refer to the official documents in detail.
- This information is based on "WebAssembly Specification Release 1.0 (Draft, last updated Oct 31, 2018)".
- This information is current as of Nov, 2018. Still work in progress.

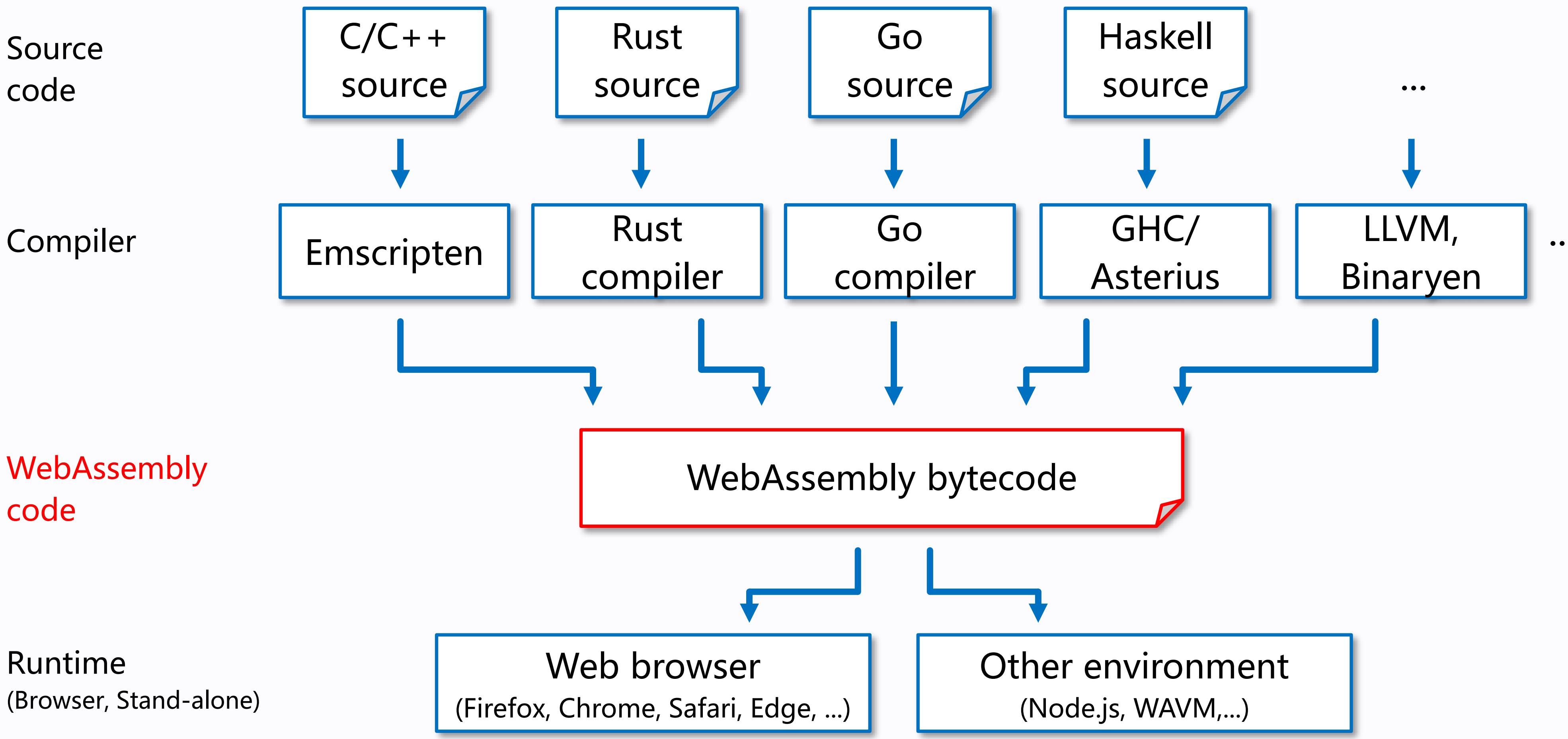
Contents

1. Introduction

1. Introduction

Overview

WebAssembly is a code format



WebAssembly is a safe, portable, low-level code format.

References : [1] Ch.1.1

WebAssembly code

Text format

syntactic sugar

```
(module
  (func (export "add7")
    (param $x i64)
    (result i64)
    (i64.add
      (get_local $x)
      (i64.const 7))))
```

core syntax

```
(module
  (type
    (func (param i64) (result i64)))
  (func (type 0)
    (param i64) (result i64)
    get_local 0
    i64.const 7
    i64.add)
  (export "add7" (func 0)))
```

Binary format

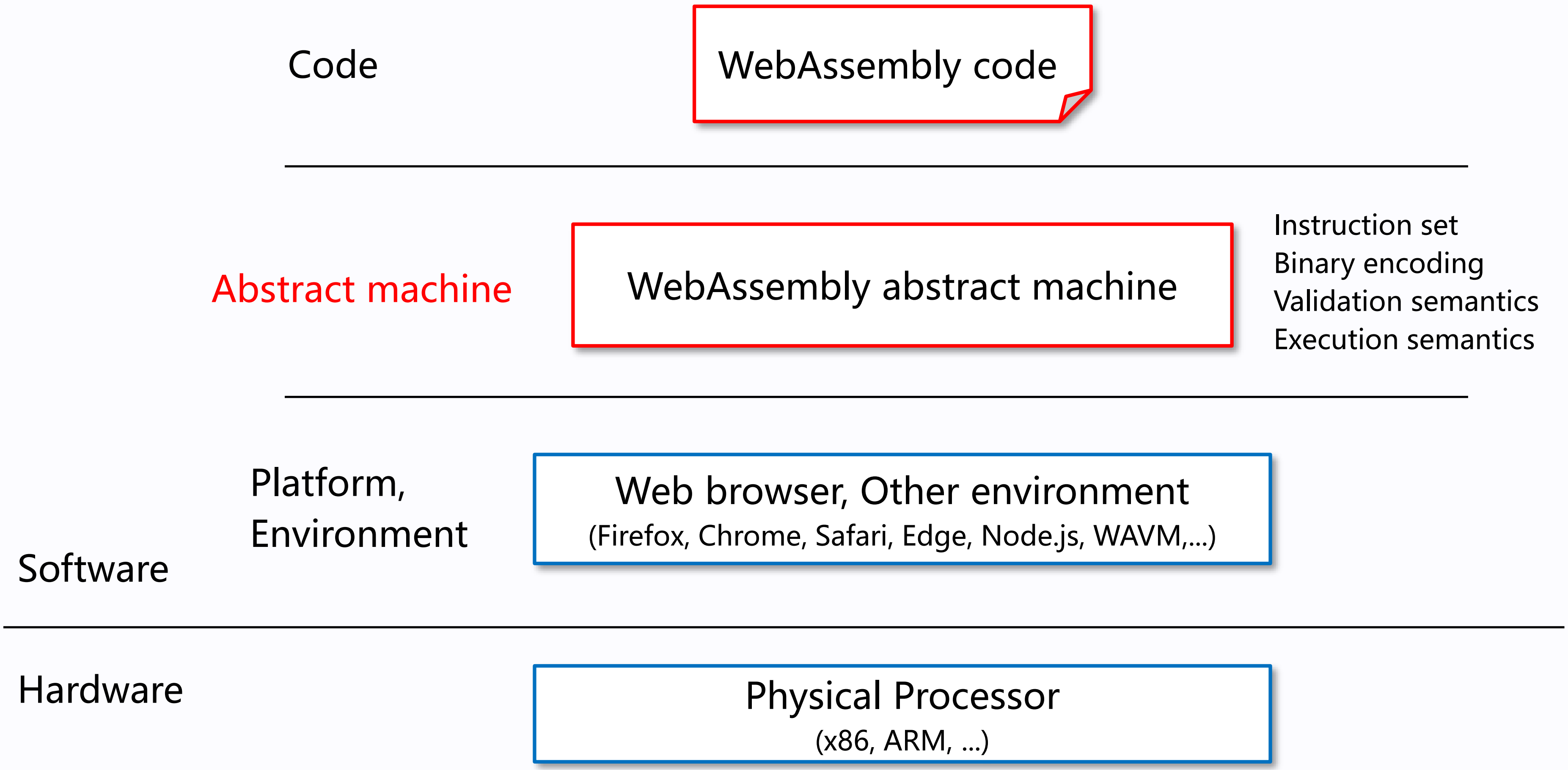
```
0x0061736d010000 ...
```

WebAssembly encodes a low-level, assembly-like programming language.

WebAssembly has multiple concrete representations.
(its text format and the binary format.)

References : [1] Ch.2, Ch.5, Ch.6

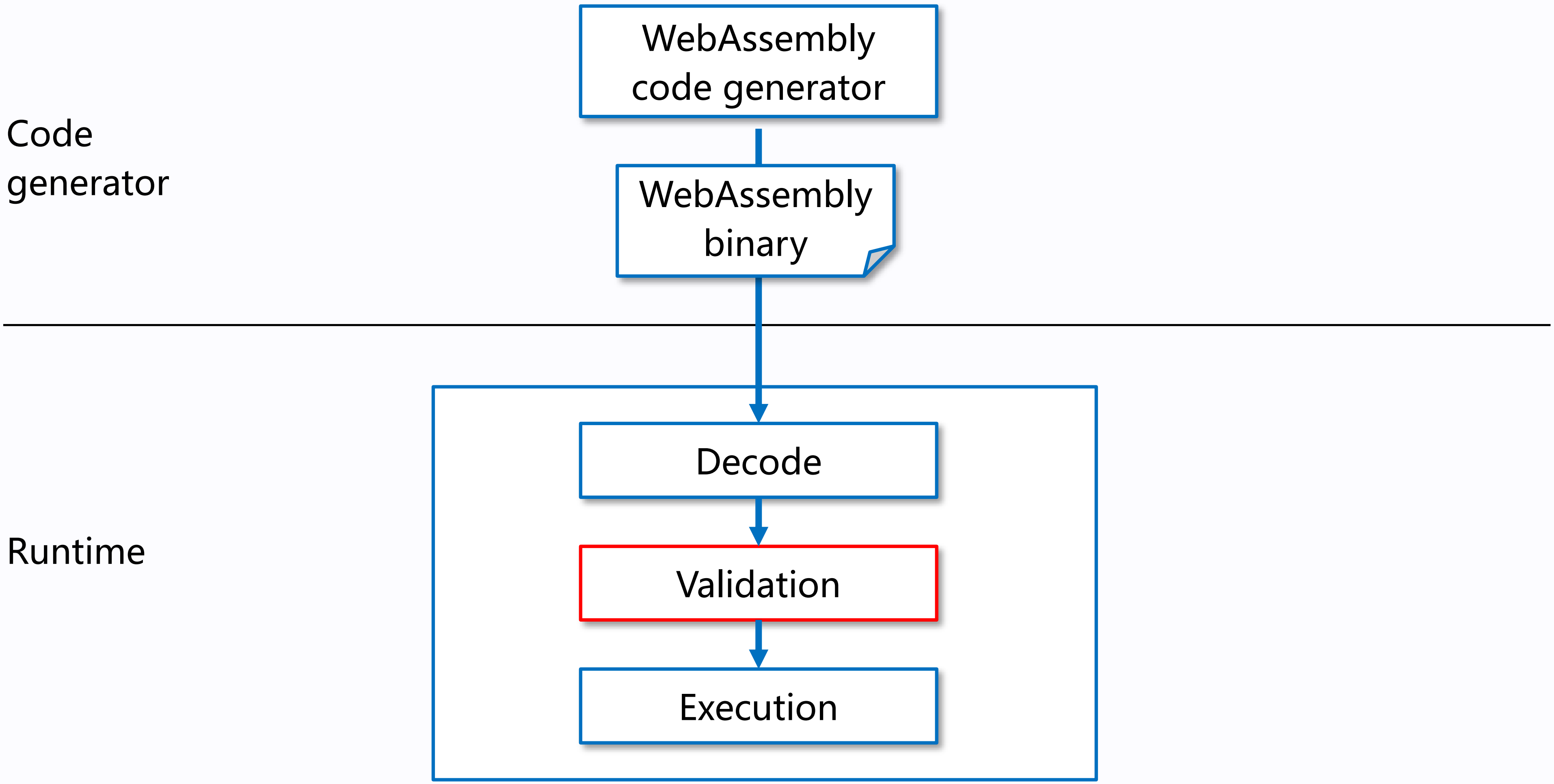
Abstract machine is defined



WebAssembly is a virtual instruction set architecture (virtual ISA).

References : [1] Ch.1.1

Validation



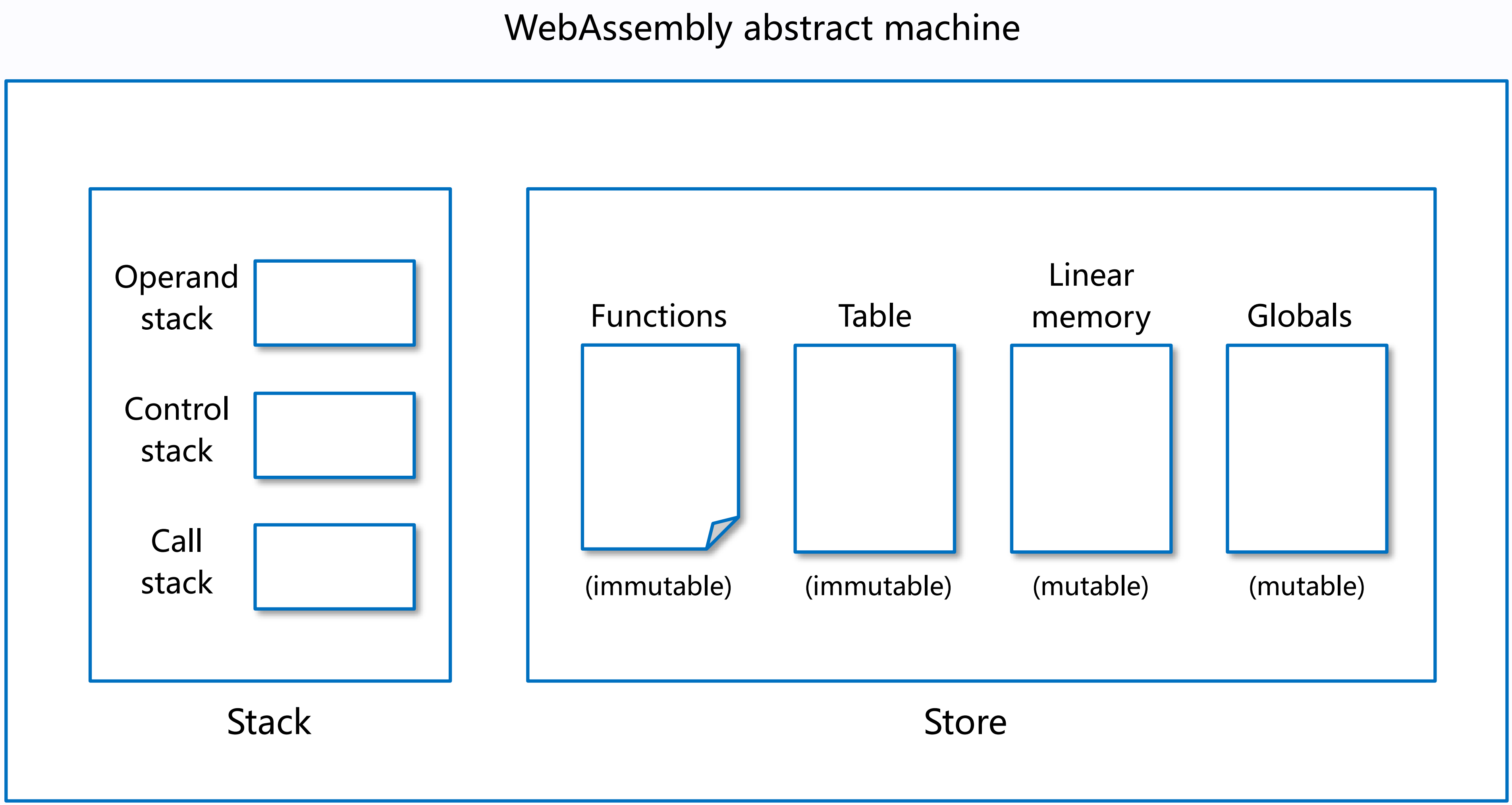
Validation checks that a WebAssembly module is well-formed.
Only valid modules can be instantiated.

References : [1] Ch.1

2. WebAssembly abstract machine

Abstract machine

WebAssembly abstract machine

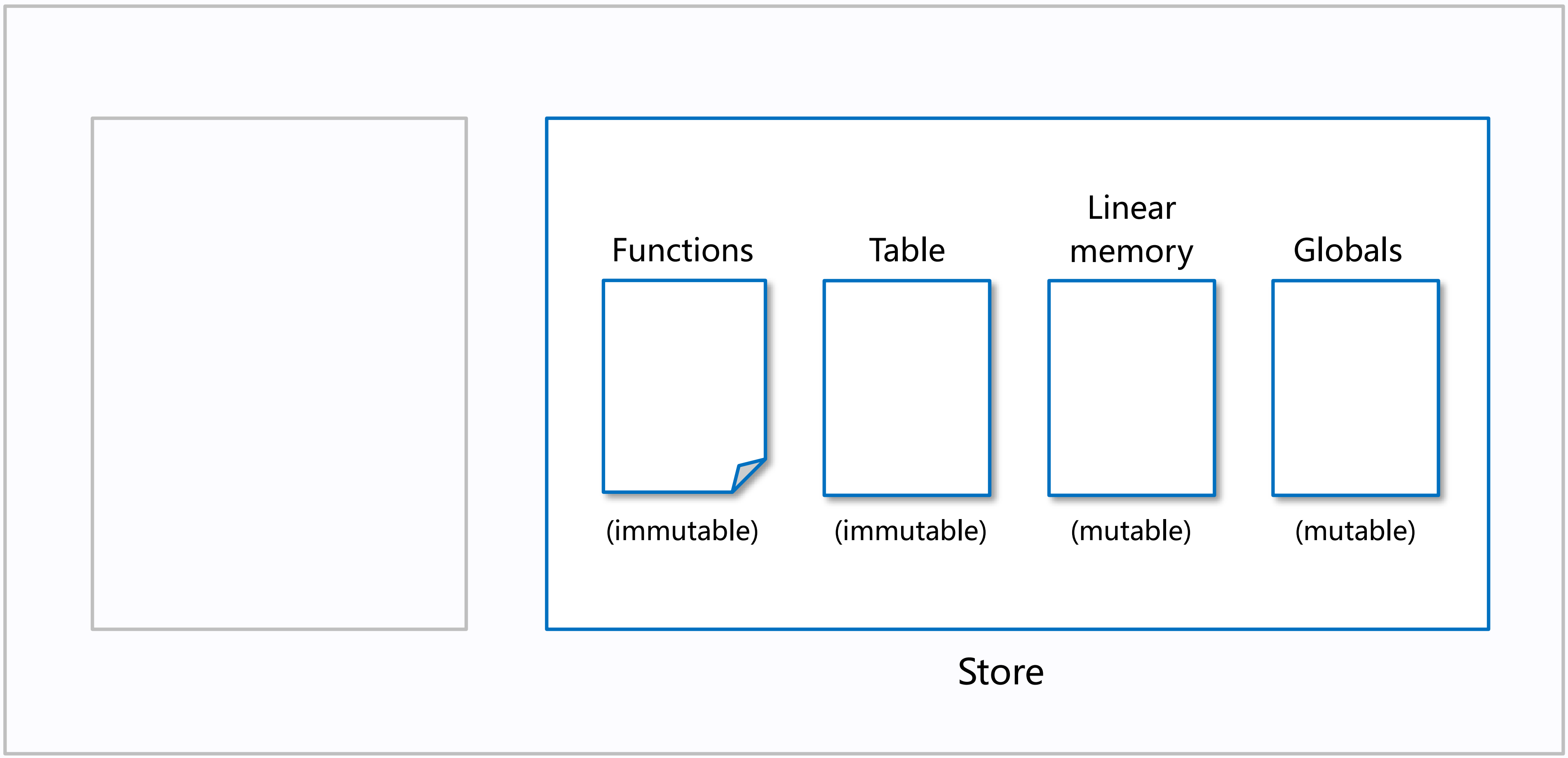


WebAssembly abstract machine is based on a stack machine.
The abstract machine includes a store and an implicit stack.

References : [1] Ch.4

Store

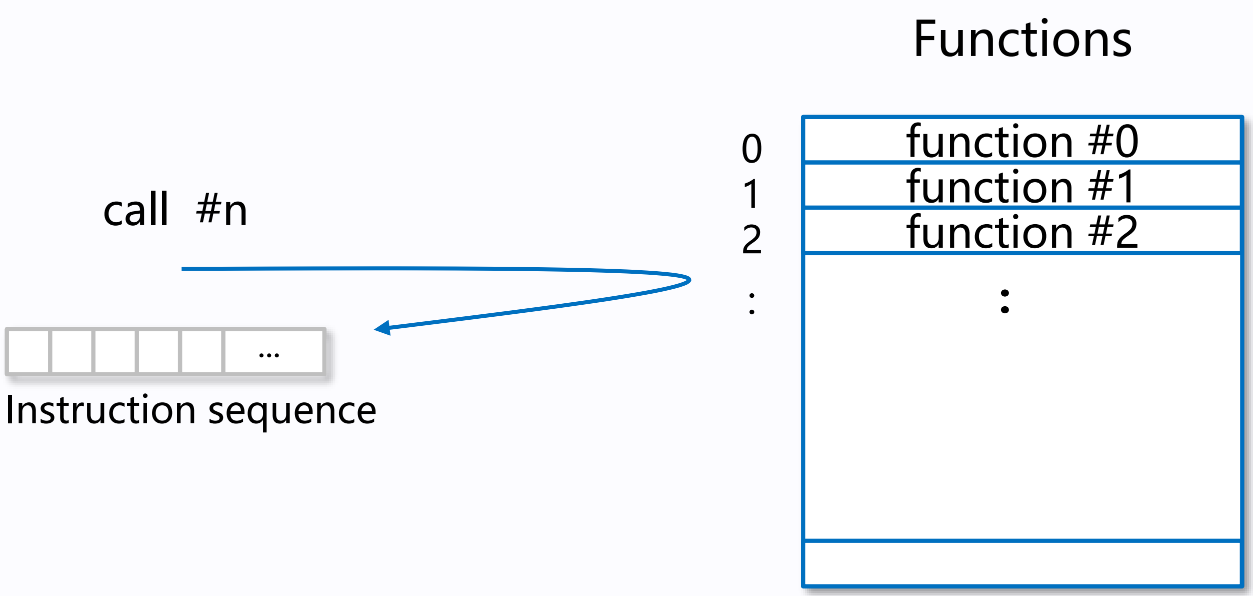
Store



The store represents all global state.
The store have been allocated during the life time of the abstract machine.

References : [1] Ch.4

Functions

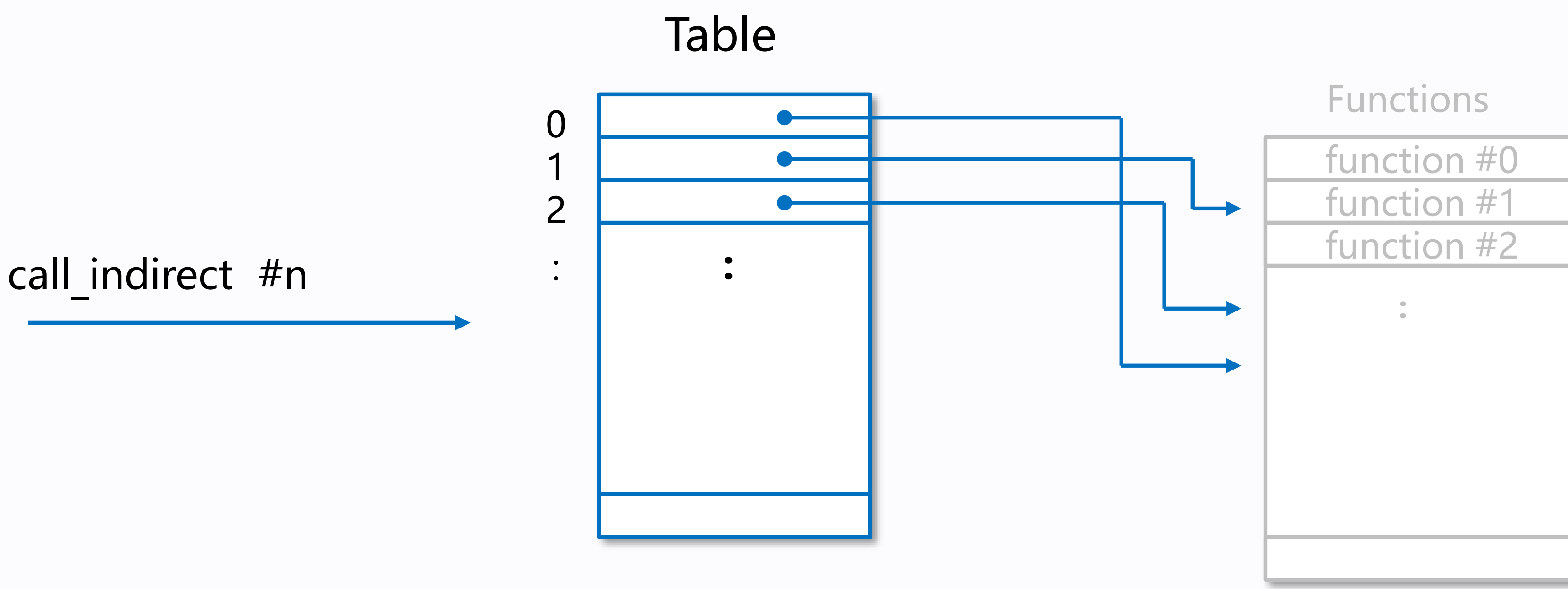


The function component of a module defines a vector of functions.
Functions are referenced through function indices.

sequence of code

References : [1] Ch.2, Ch.4

Table

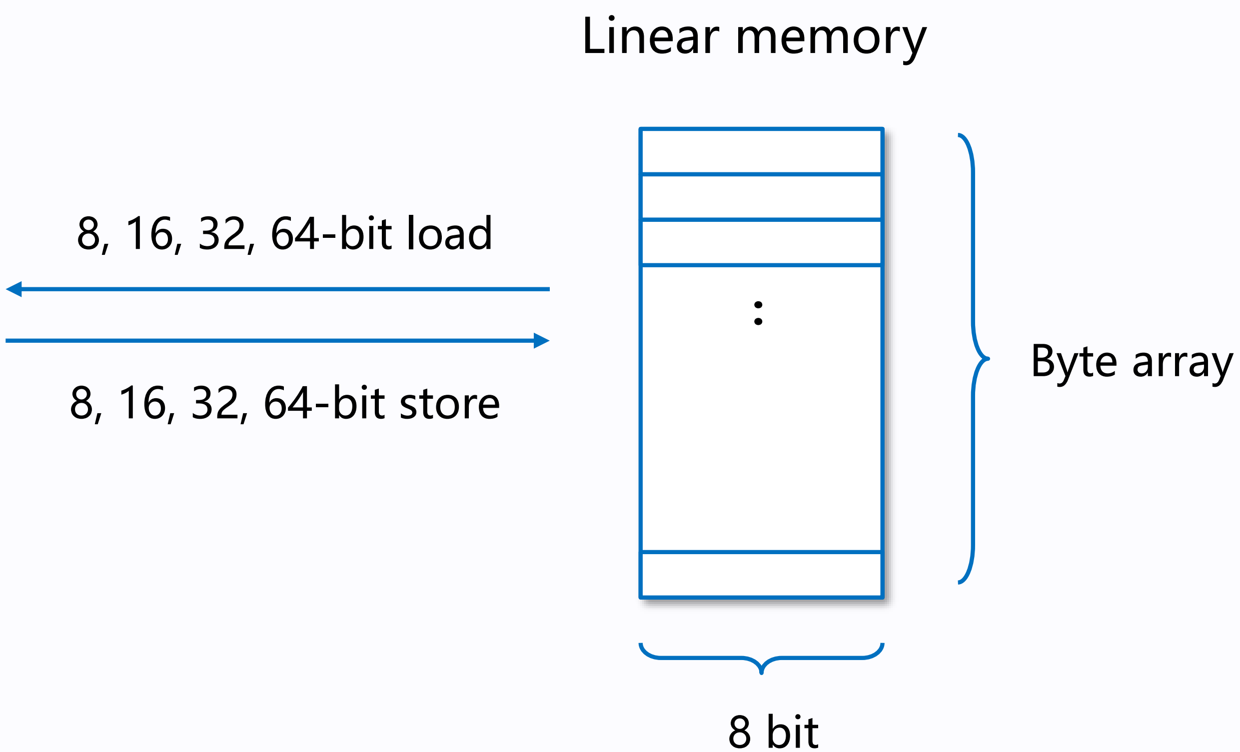


[spec, 1.2.1]

The table is an array of opaque values of a particular element type.
Currently, the only available element type is an untyped function reference.
This allows emulating function pointers by way of table indices.
Tables are referenced through table indices

References : [1] Ch.1, Ch.2, Ch.4

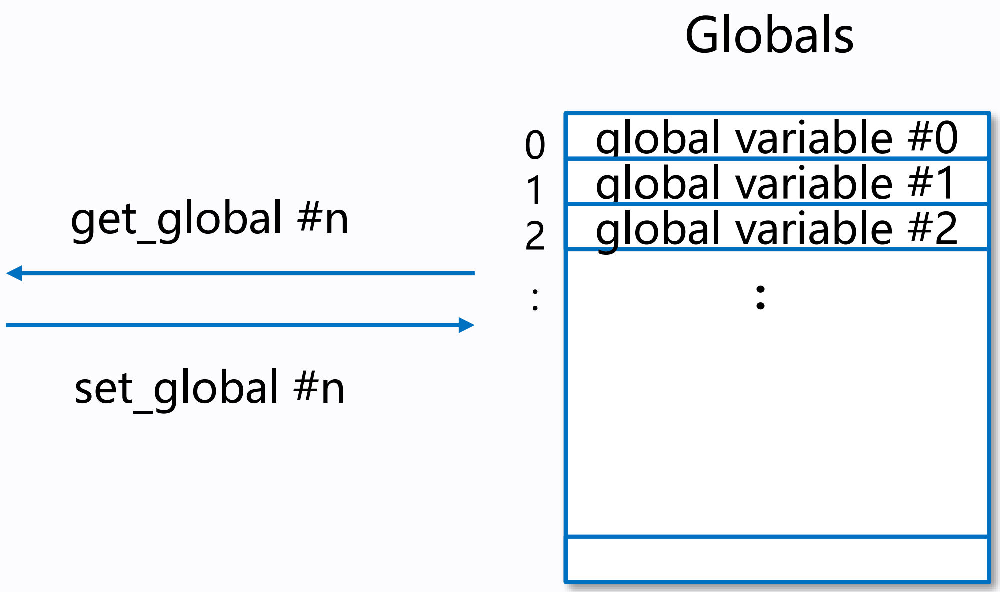
Linear memory



The linear memory is a contiguous, mutable array of raw bytes.
The linear memory can be addressed at byte level (including unaligned).
The size of the memory is a multiple of the WebAssembly page size.

References : [1] Ch.1, Ch.2, Ch.4

Globals



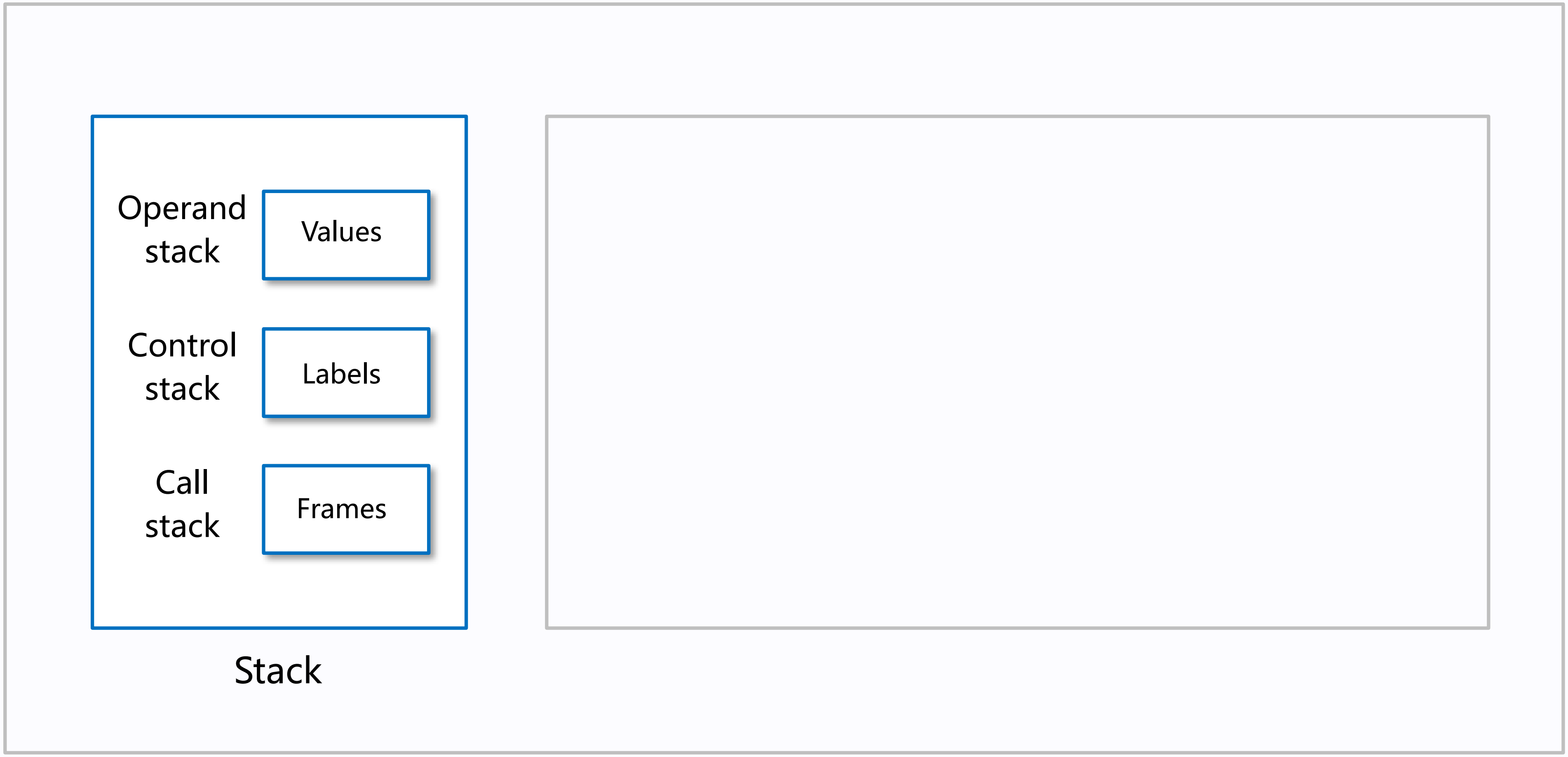
The globals component defines a vector of global variables.
The globals are referenced through global indices.
The global variables hold a value and can either be mutable or immutable.

References : [1] Ch.2, Ch.4

2. WebAssembly abstract machine

Stack

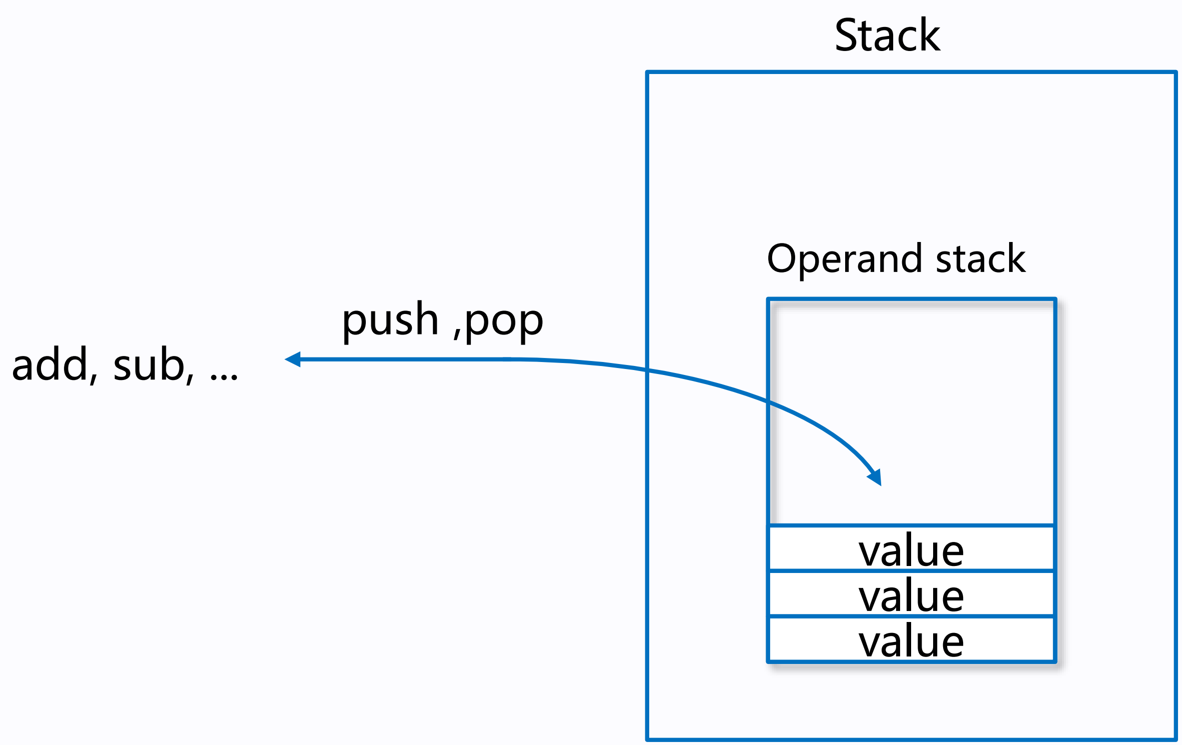
Stack



Most instructions interact with the implicit stack.
The stack contains values, labels and frames(activations).

References : [1] Ch.4

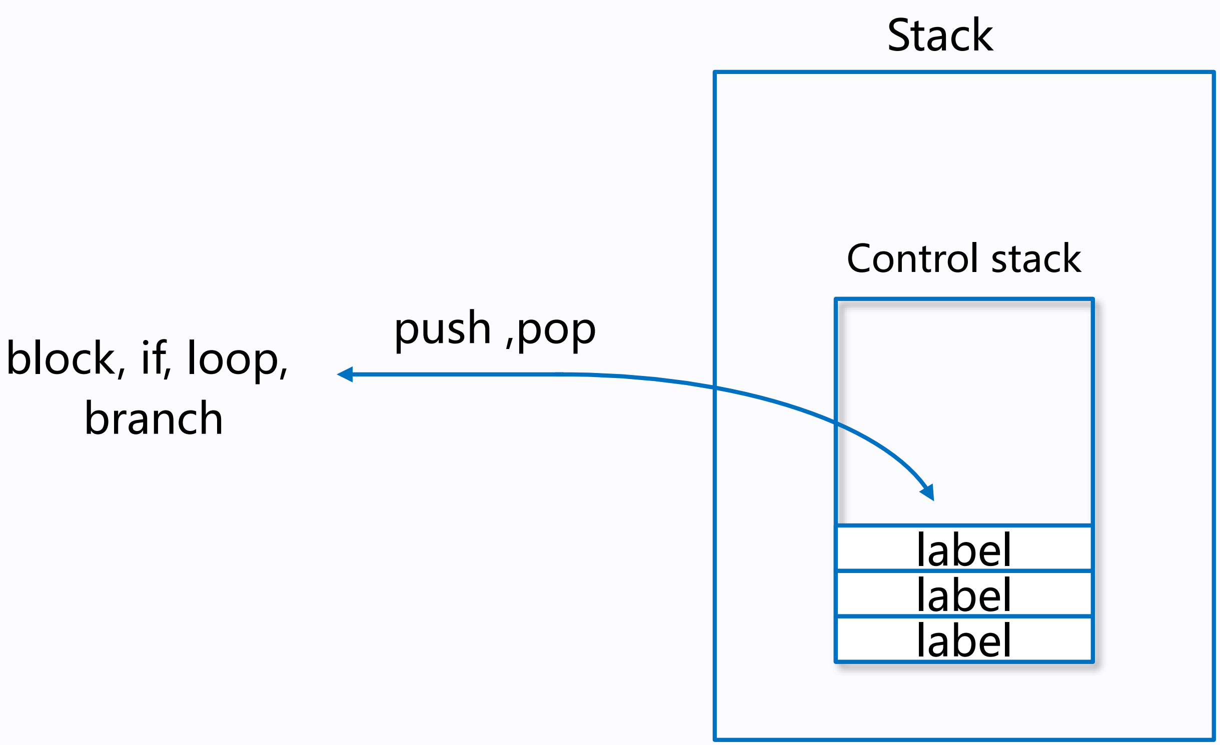
Operand stack



Instructions manipulate values on an implicit operand stack.
The layout of the operand stack can be statically determined at any point in the code.

References : [1] Ch.2, Ch.4, Ch.7, [2]

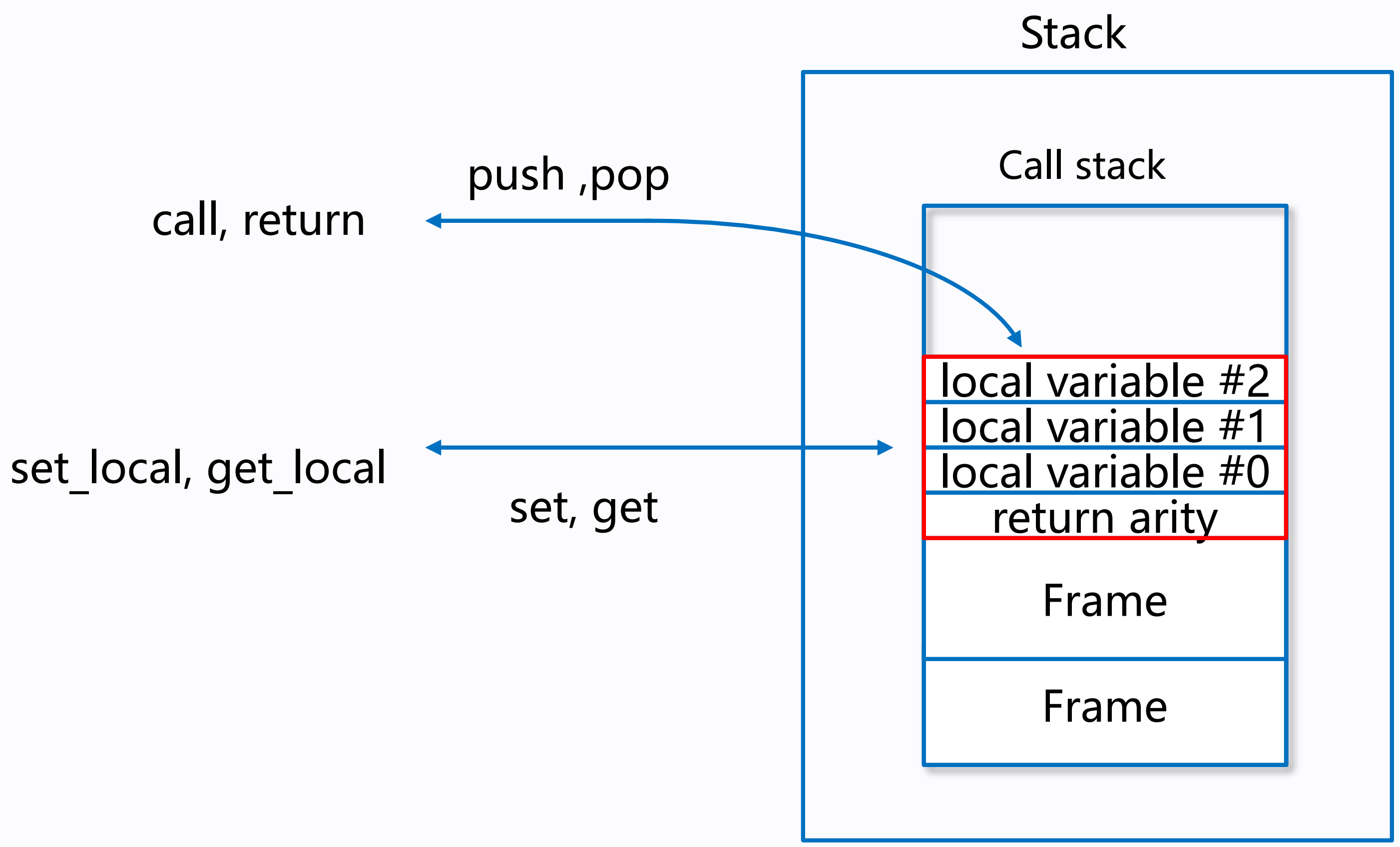
Control stack



Each structured control instruction introduces an implicit label.
Labels are targets for branch instructions that reference them with label indices.

References : [1] Ch.2, Ch.4, Ch.7, [2]

Call stack

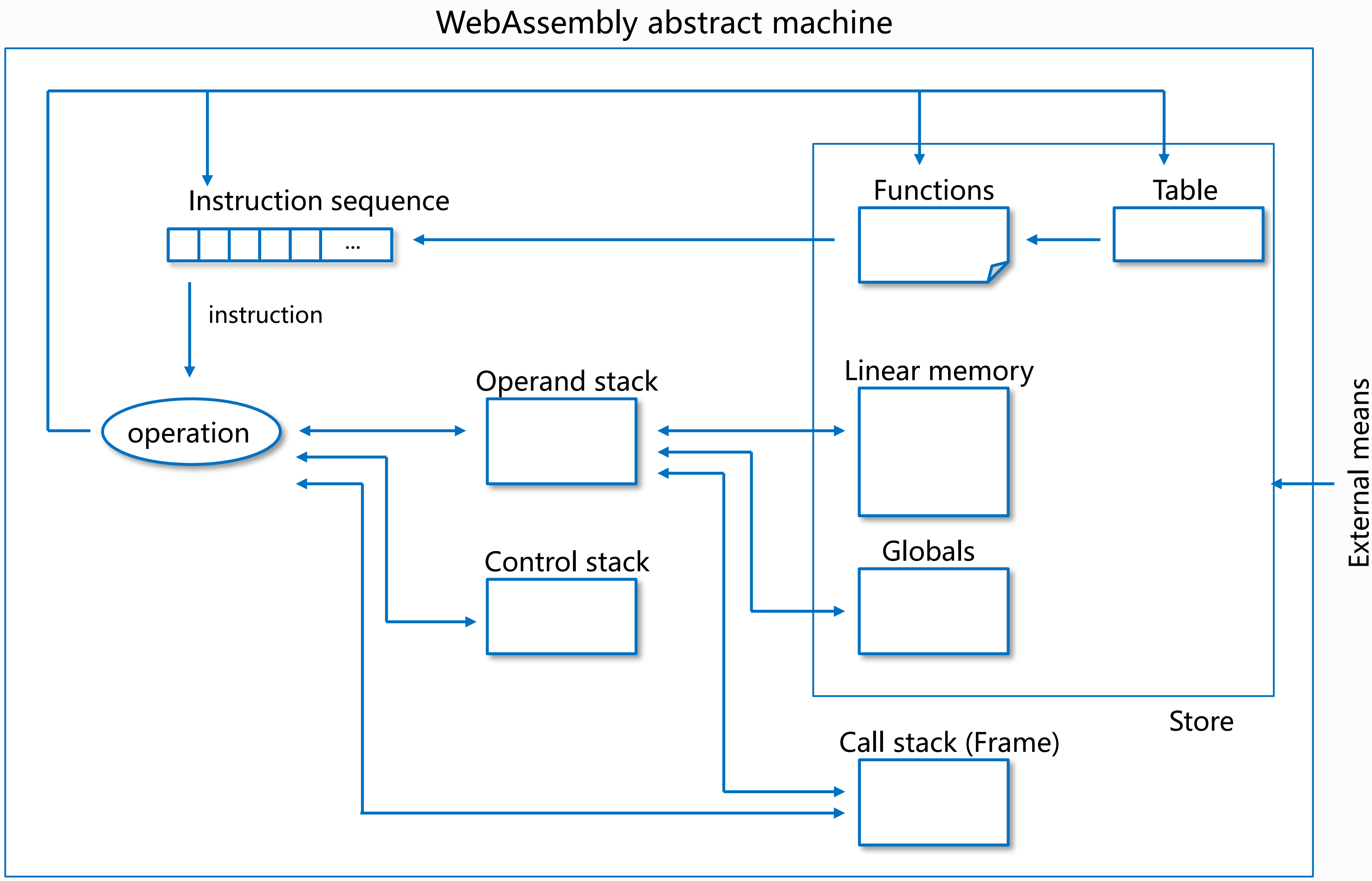


Frames hold the values of its local variables (including arguments).
Frames also carry the return arity of the respective function.

References : [1] Ch.2, Ch.4, Ch.7, [2]

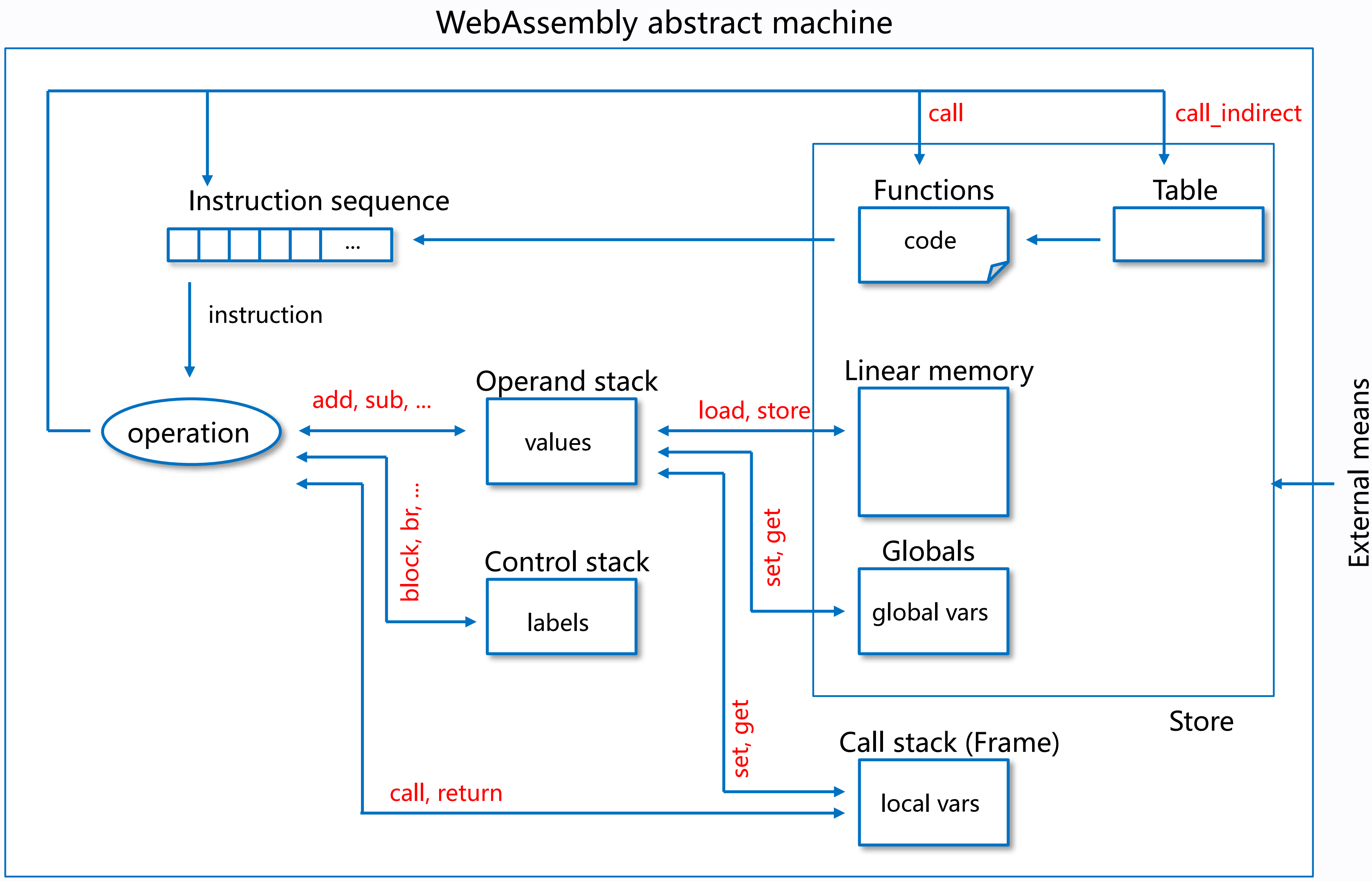
Computational model

Computational model



References : [1] Ch.1, Ch.4

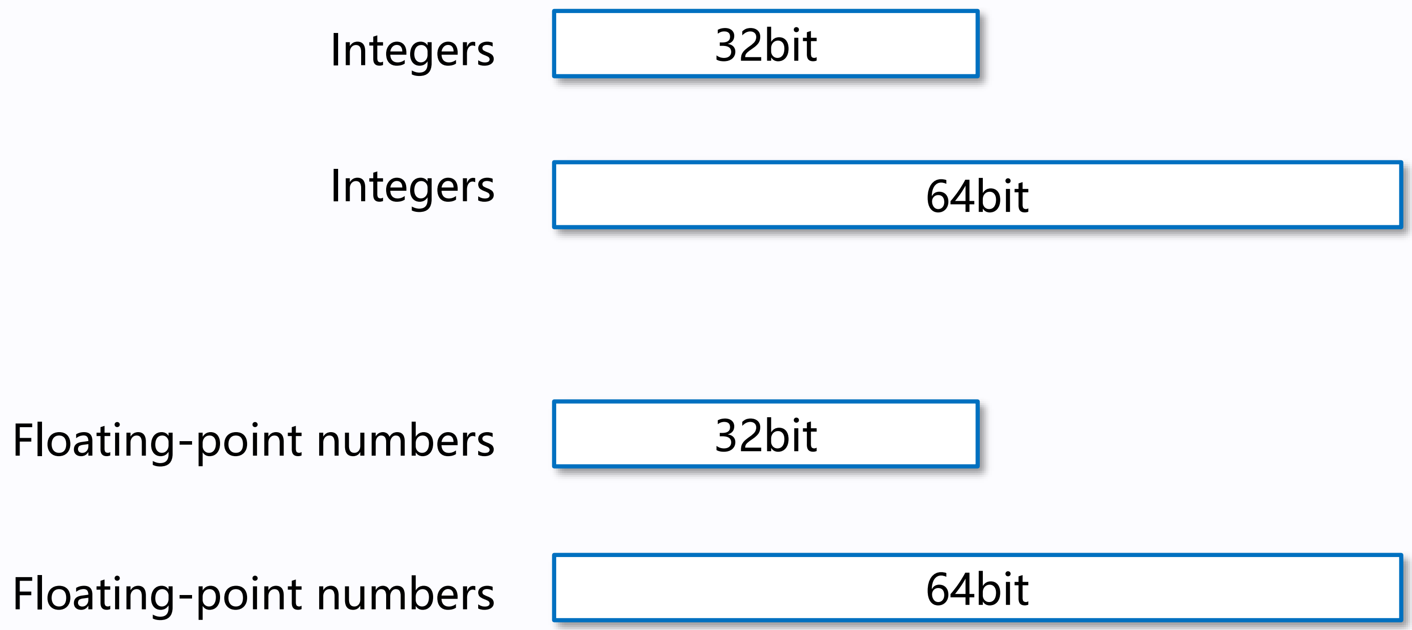
Computational model



References : [1] Ch.1, Ch.4

Type

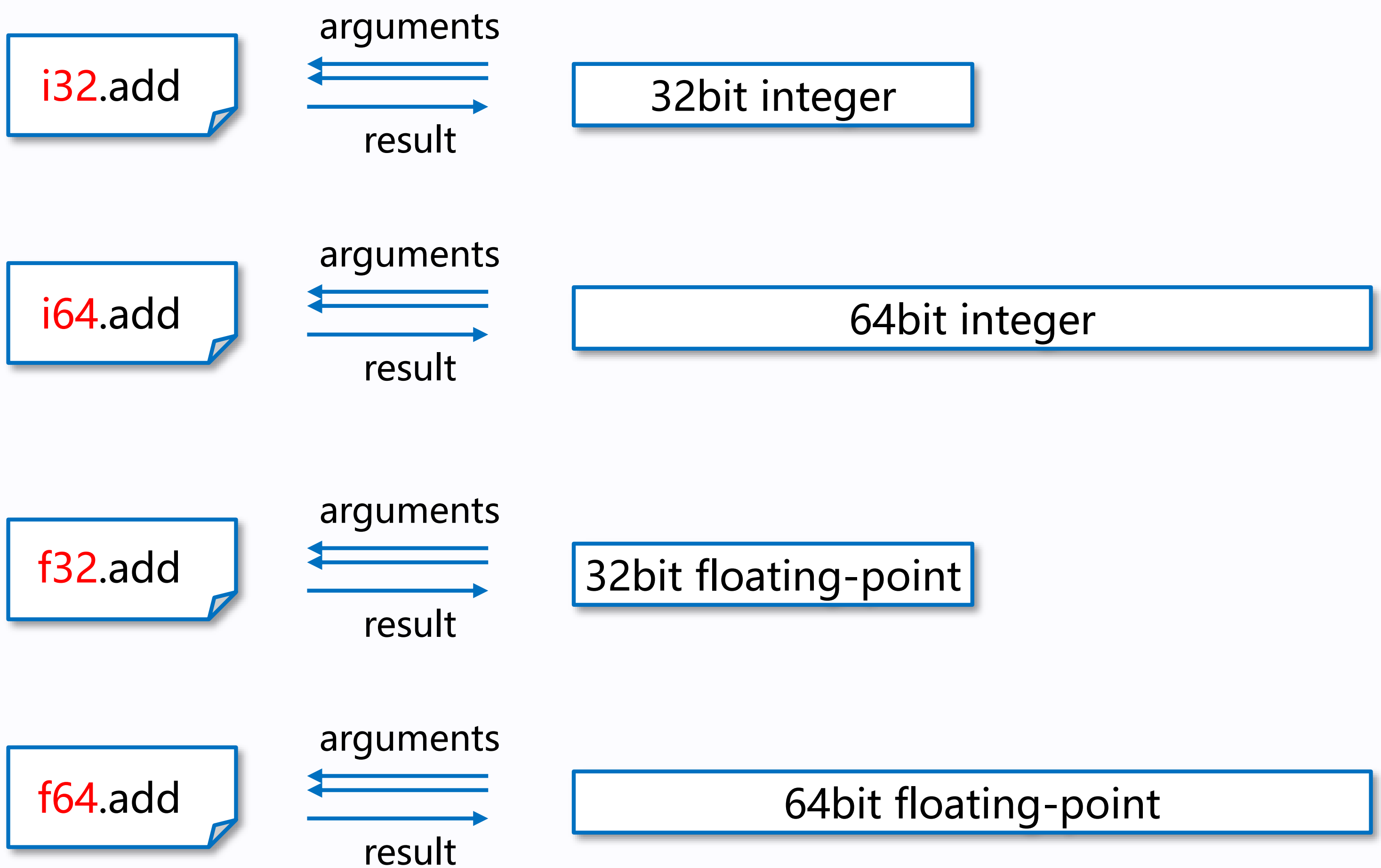
Value types



WebAssembly provides only four basic value types.
32 bit integers also serve as Booleans and as memory addresses.

References : [1] Ch.1, Ch.4

Instruction has type annotation

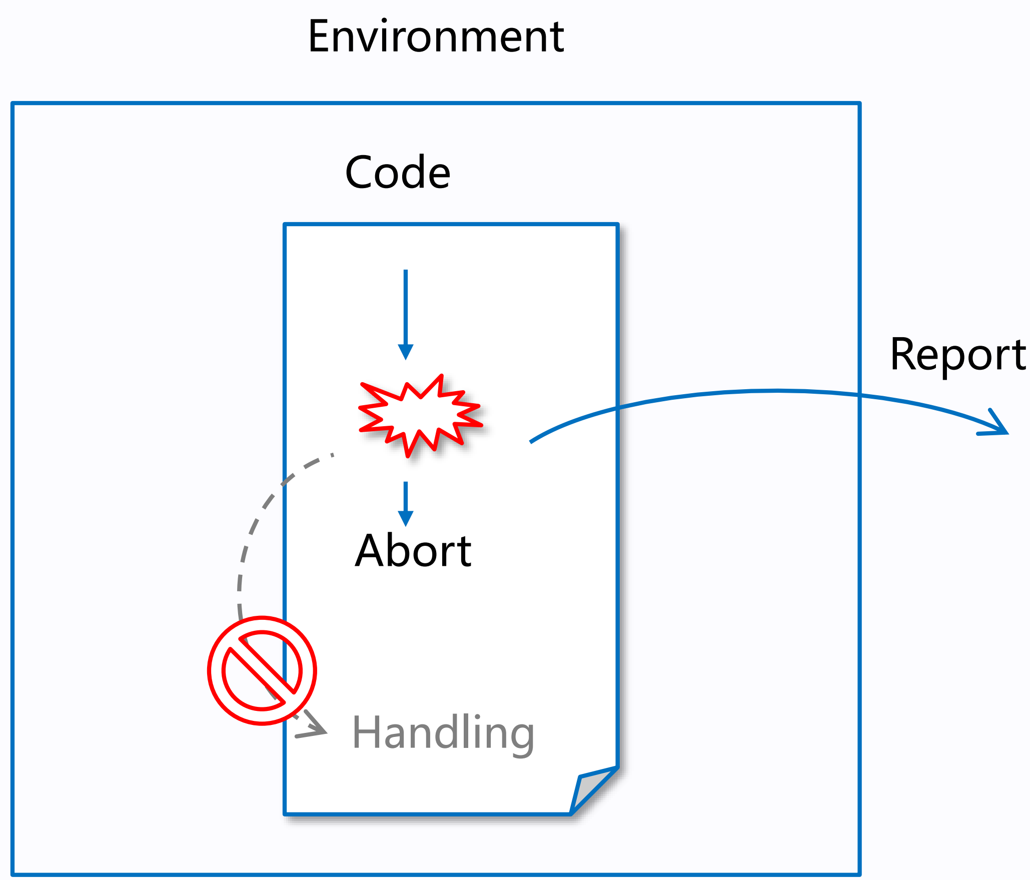


Some instructions have type annotations.
For example, the instruction i32.add has type [i32 i32] → [i32],
consuming two i32 values and producing one.

References : [1] Ch.2, Ch.3, Ch.4

Trap

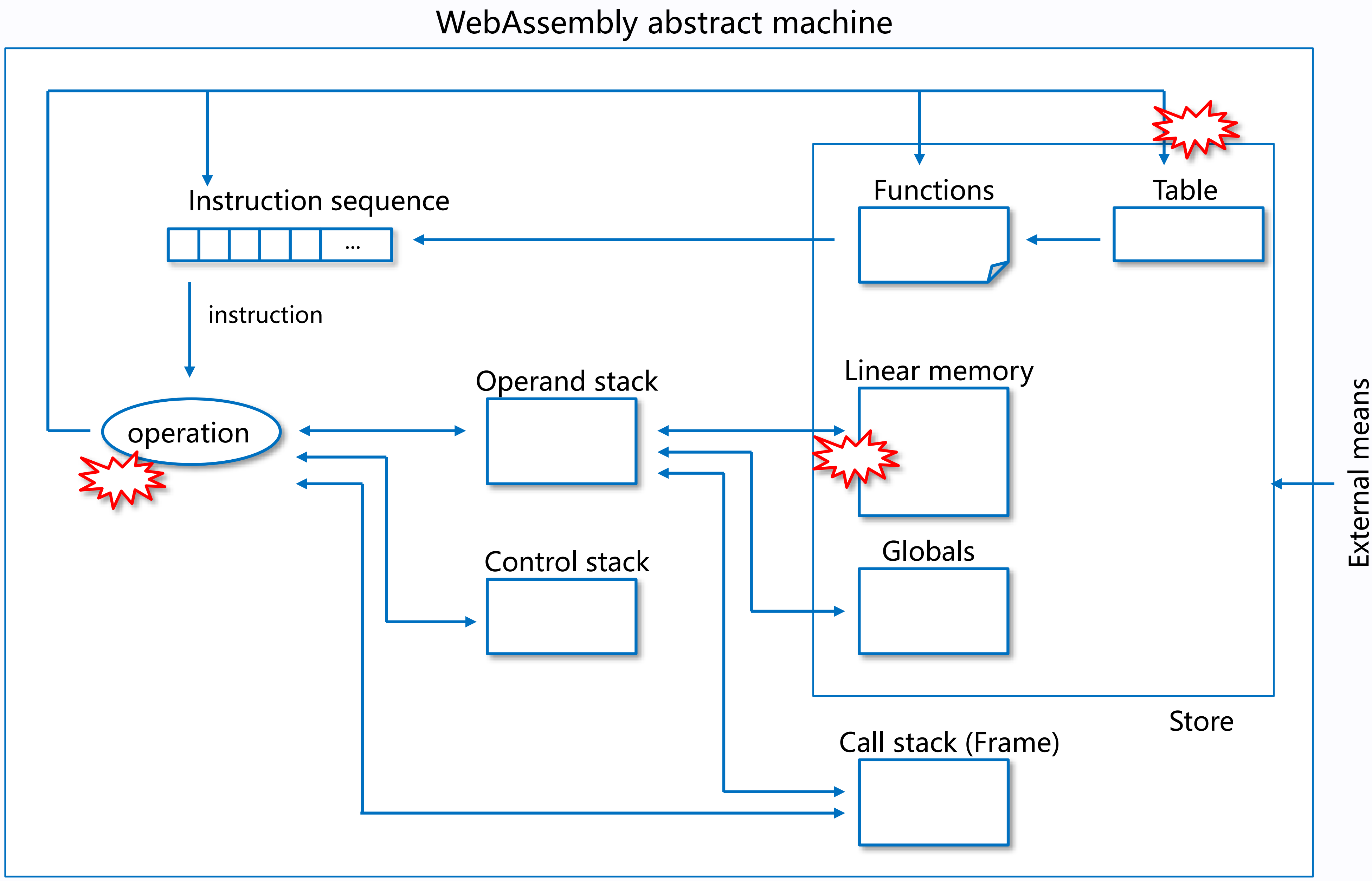
Trap



Certain instructions may produce a trap, which immediately aborts execution.
Traps cannot be handled by WebAssembly code,
but are reported to the outside environment, where they typically can be caught.

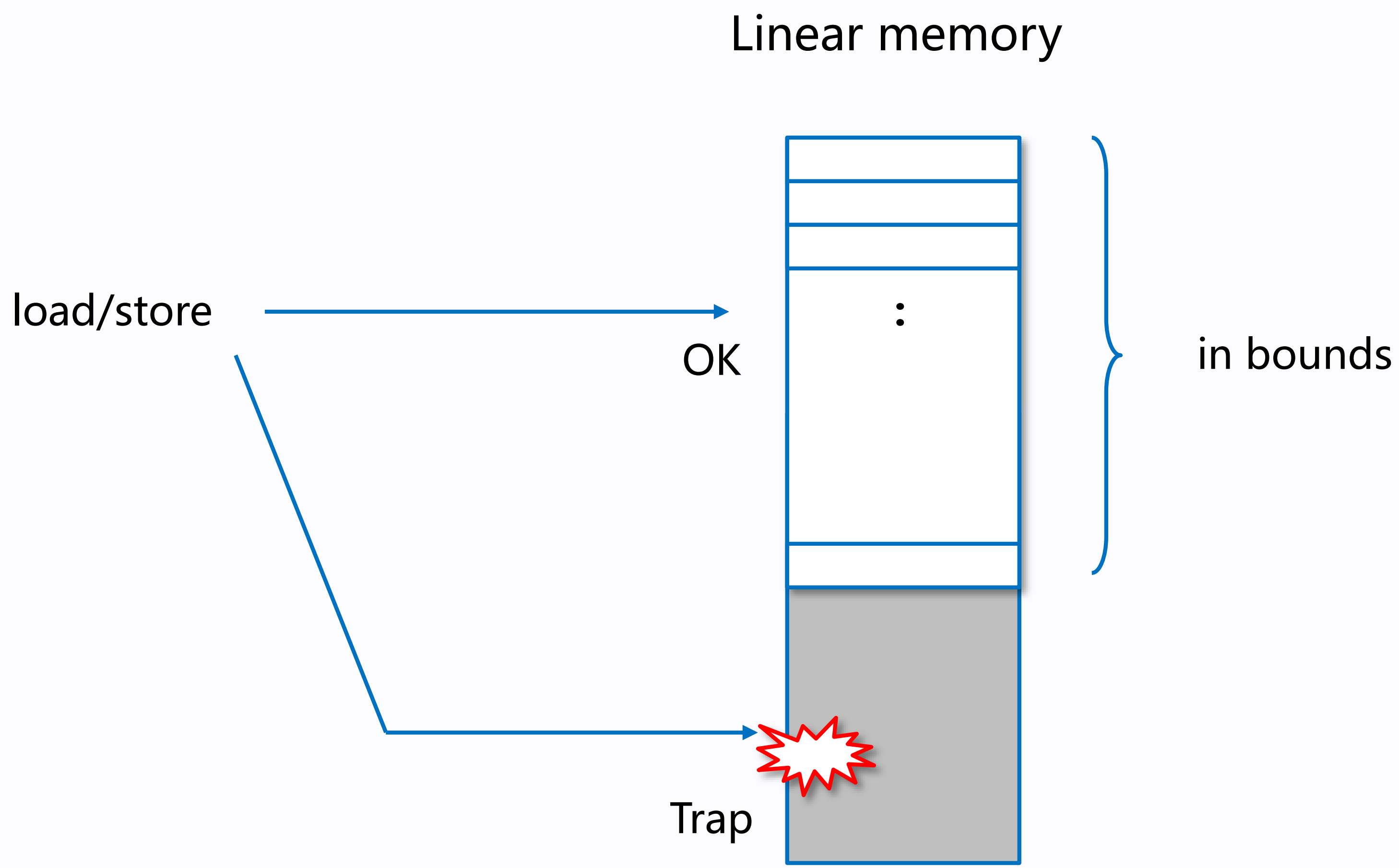
References : [1] Ch.1, Ch.4

Trap



References : [1] Ch.4, [2]

Linear memory

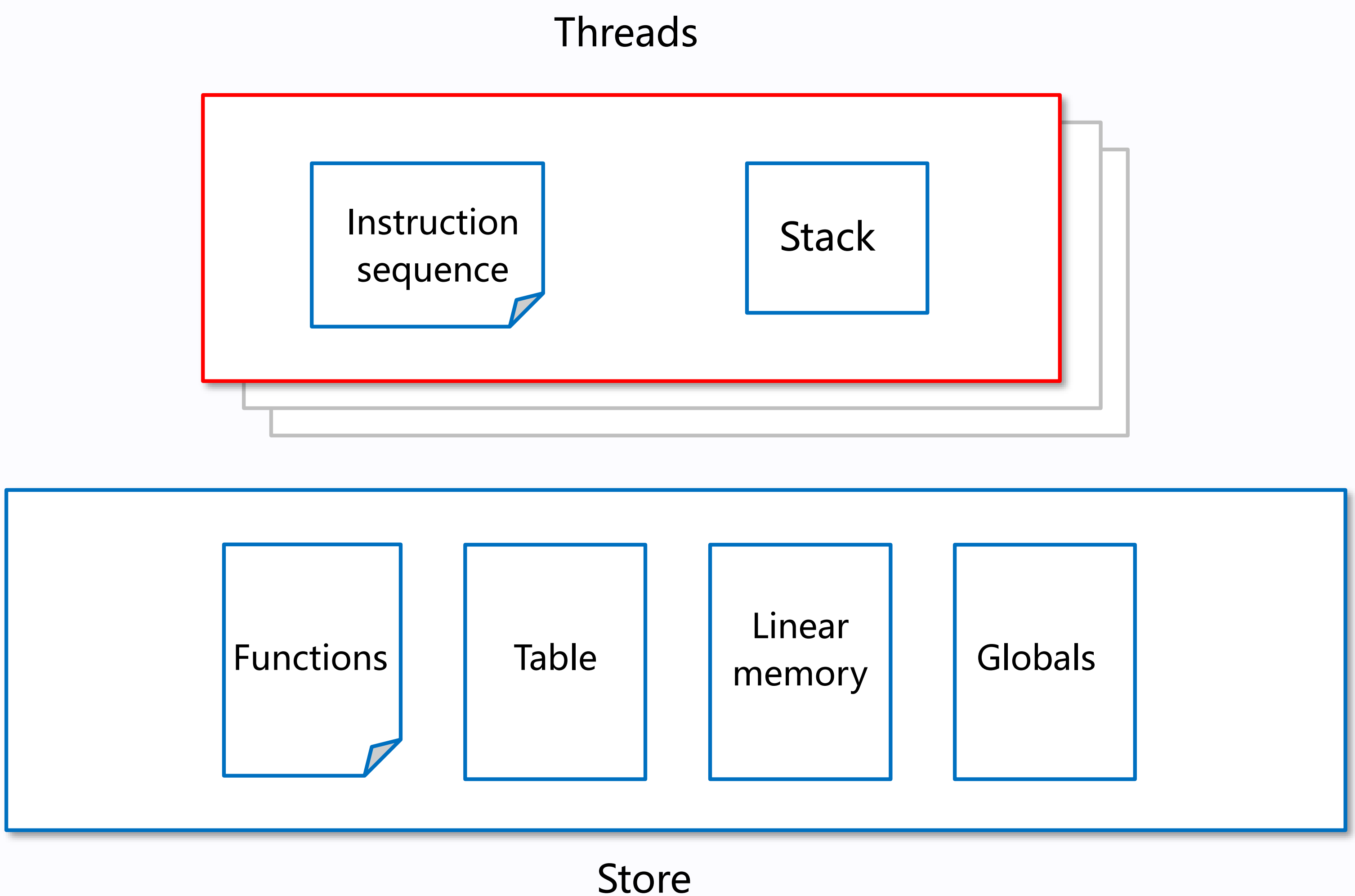


A trap occurs if an access is not within the bounds of the current memory size.

References : [1] Ch.1, Ch.4

Thread

Thread



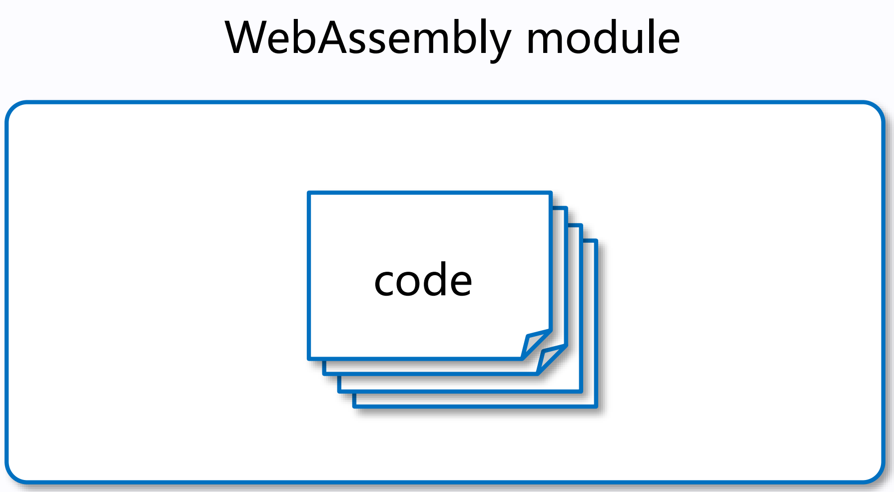
The current version of WebAssembly is single-threaded,
but configurations with multiple threads may be supported in the future.

References : [1] Ch.4

3. WebAssembly module

Module

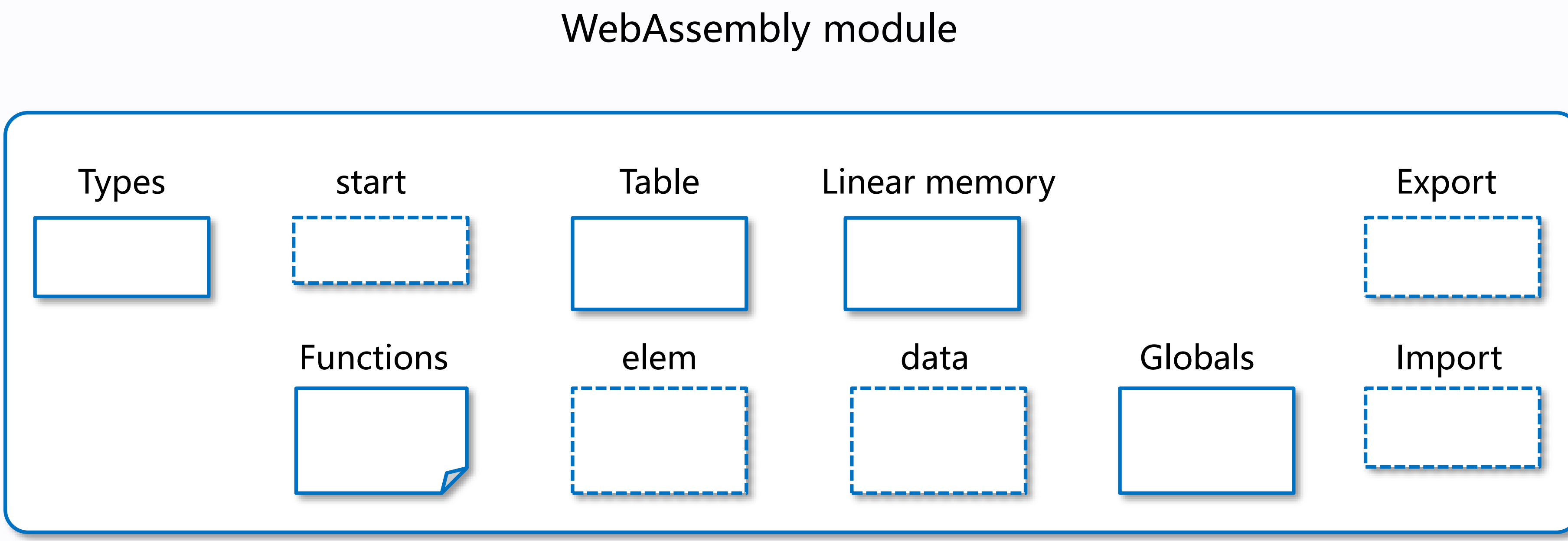
WebAssembly module



WebAssembly programs are organized into modules.
Modules are the distributable, loadable, and executable unit of code.
WebAssembly modules are distributed in a binary format.

References : [1] Ch.1, Ch.4, [5]

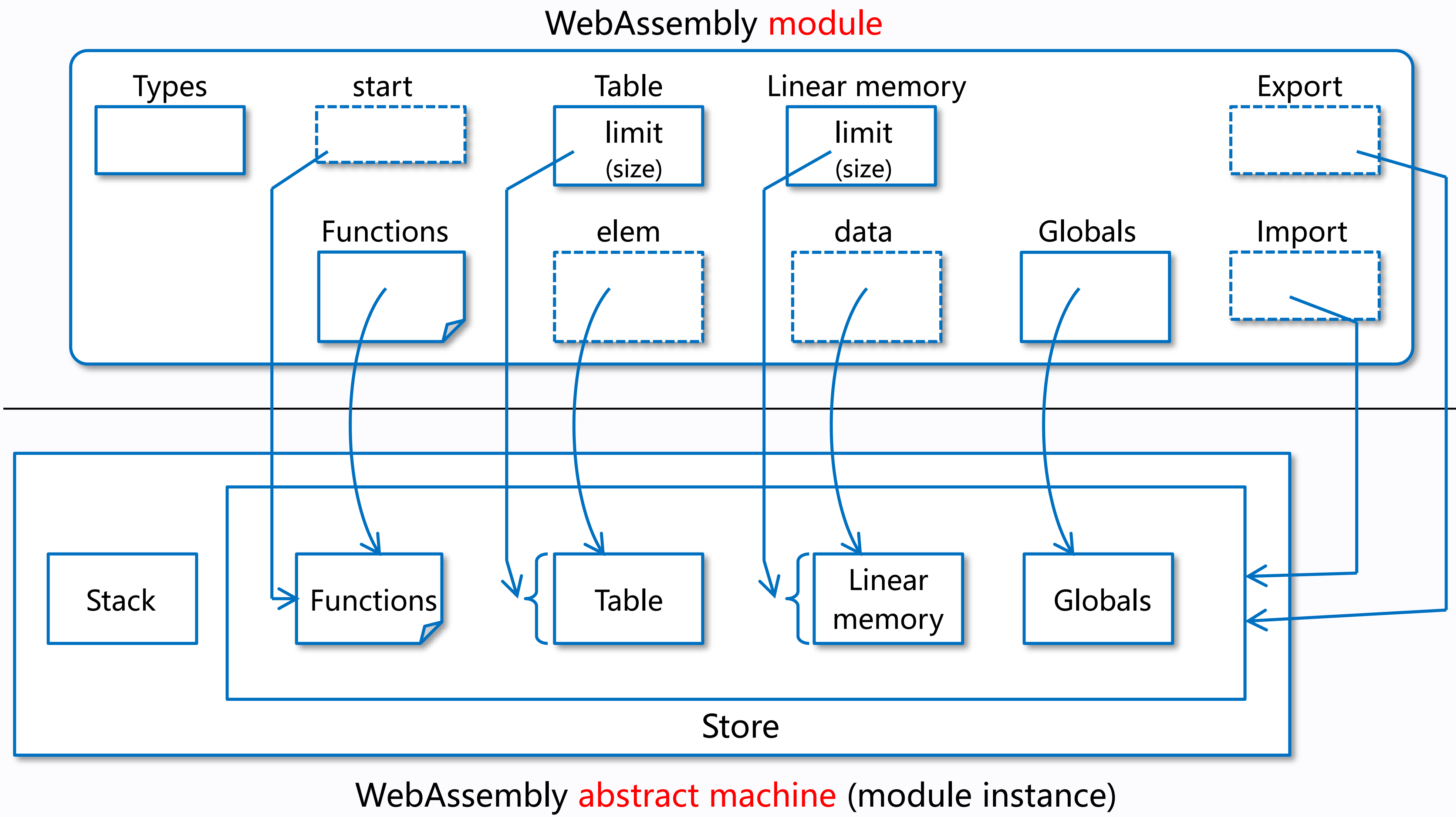
WebAssembly module



A module collects definitions for types, functions, table, memory, and globals. In addition, it can declare imports and exports and provide initialization logic in the form of data and element segments or a start function.

References : [1] Ch.1, Ch.2, Ch.4

WebAssembly module and abstract machine

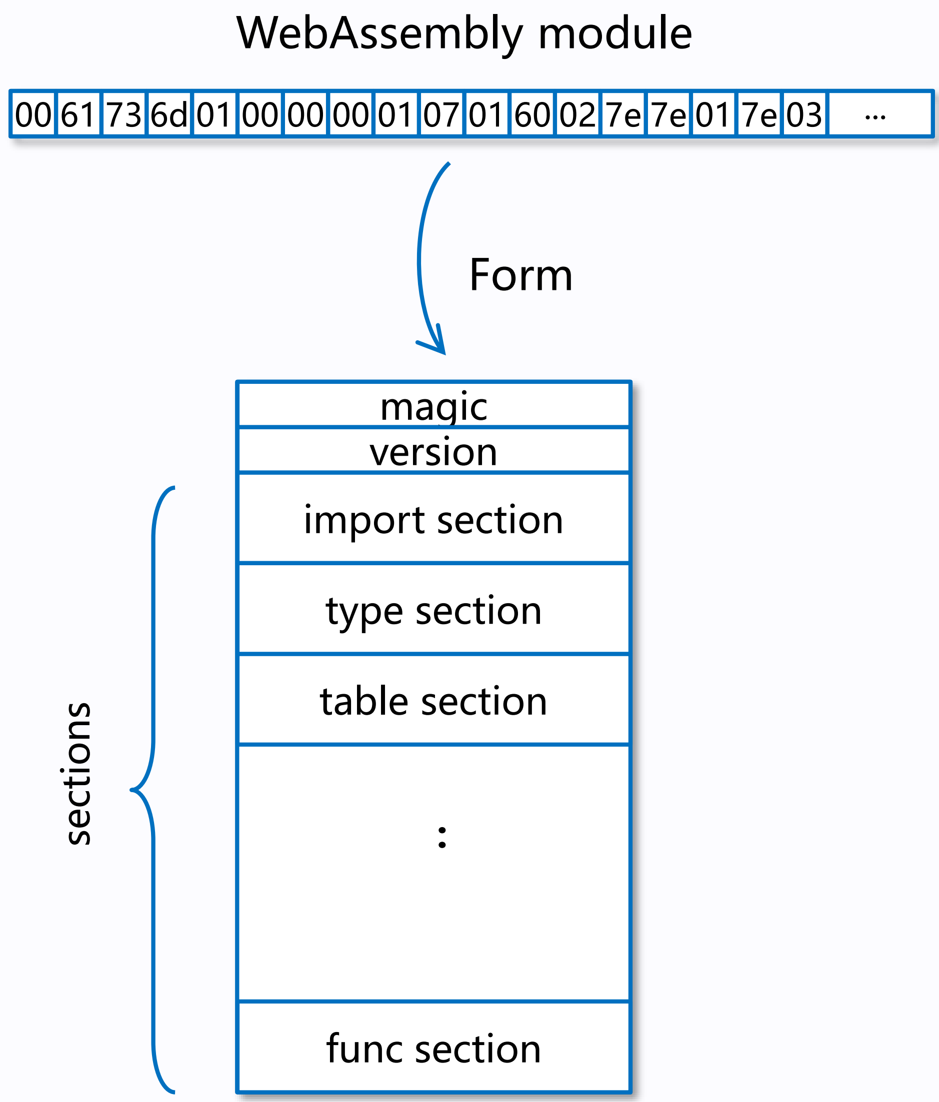


A module corresponds to the static representation of a program.
A module instance corresponds to a dynamic representation.

References : [1] Ch.1, Ch.2, Ch.4

Binary encoding

Binary encoding of modules



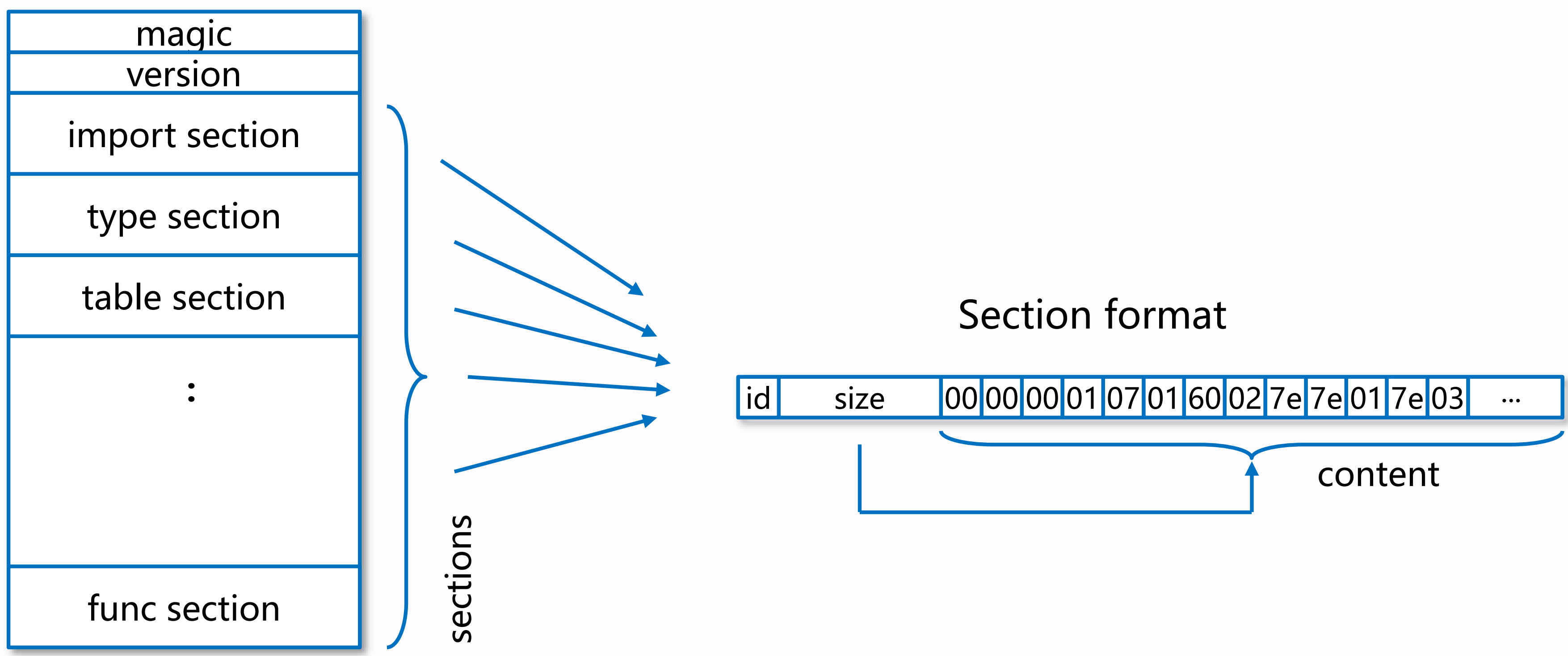
Binary encoding of modules

The binary encoding of modules is organized into sections.

References : [1] Ch.5

Sections

Binary encoding of modules



Each section consists of

- a one-byte section id,
- the u32 size of the contents, in bytes,
- the actual contents, whose structure is depended on the section id.

References : [1] Ch.5

Example of WebAssembly module

[text format]

```
(module
  (func (export "foo" (result i32)
    i32.const 7))
```

(`wat2wasm –v` command)

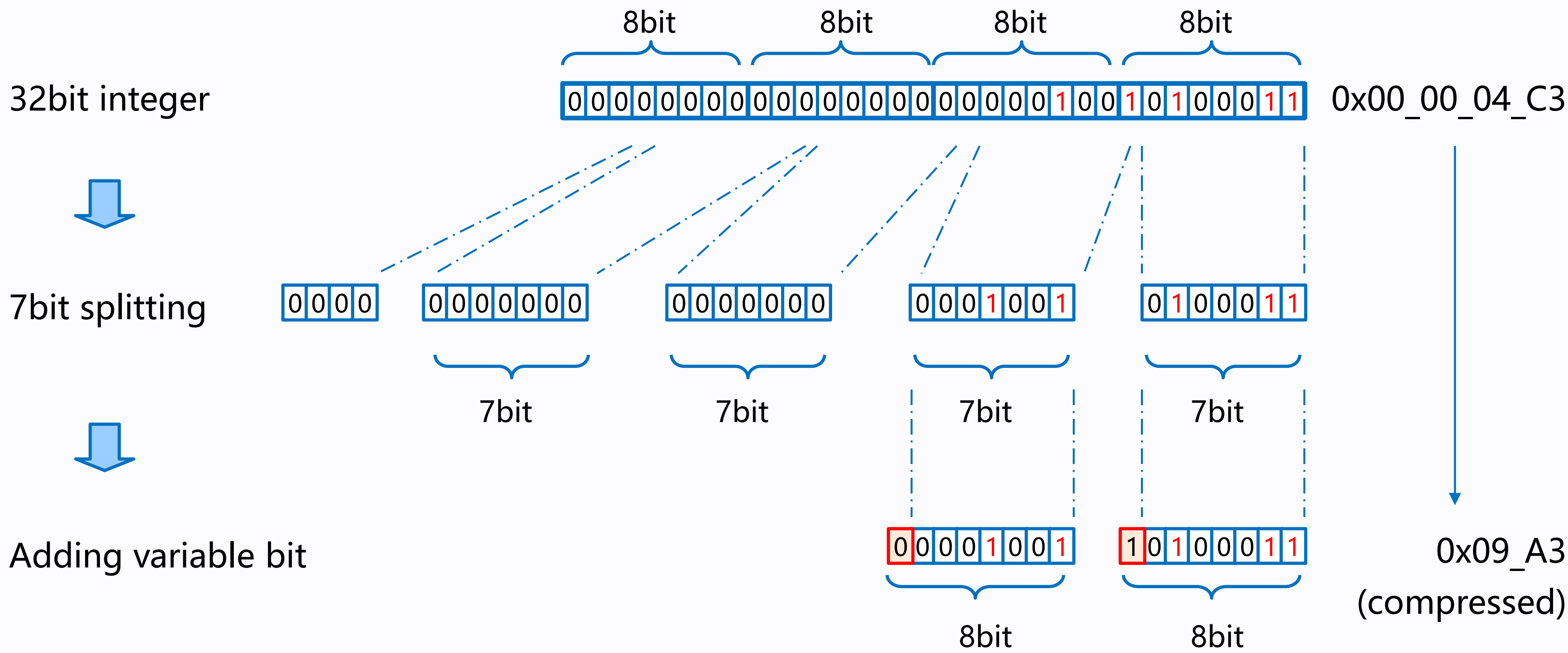
[binary format]

```
0000000: 0061 736d          ; WASM_BINARY_MAGIC
0000004: 0100 0000          ; WASM_BINARY_VERSION
; section "Type" (1)
0000008: 01                 ; section code
0000009: 05                 ; section size
000000a: 01                 ; num types
; type 0
000000b: 60                 ; func
000000c: 00                 ; num params
000000d: 01                 ; num results
000000e: 7f                 ; i32
; section "Function" (3)
000000f: 03                 ; section code
0000010: 02                 ; section size
0000011: 01                 ; num functions
0000012: 00                 ; function 0 signature
                            ; index
```

```
; section "Export" (7)
0000013: 07                 ; section code
0000014: 07                 ; section size
0000015: 01                 ; num exports
0000016: 03                 ; string length
0000017: 666f 6f            ; foo  ; export name
000001a: 00                 ; export kind
000001b: 00                 ; export func index
; section "Code" (10)
000001c: 0a                 ; section code
000001d: 06                 ; section size
000001e: 01                 ; num functions
; function body 0
000001f: 04                 ; func body size
0000020: 00                 ; local decl count
0000021: 41                 ; i32.const
0000022: 07                 ; i32 literal
0000023: 0b                 ; end
```

References : [1] Ch.5

Integer encoding with LEB128



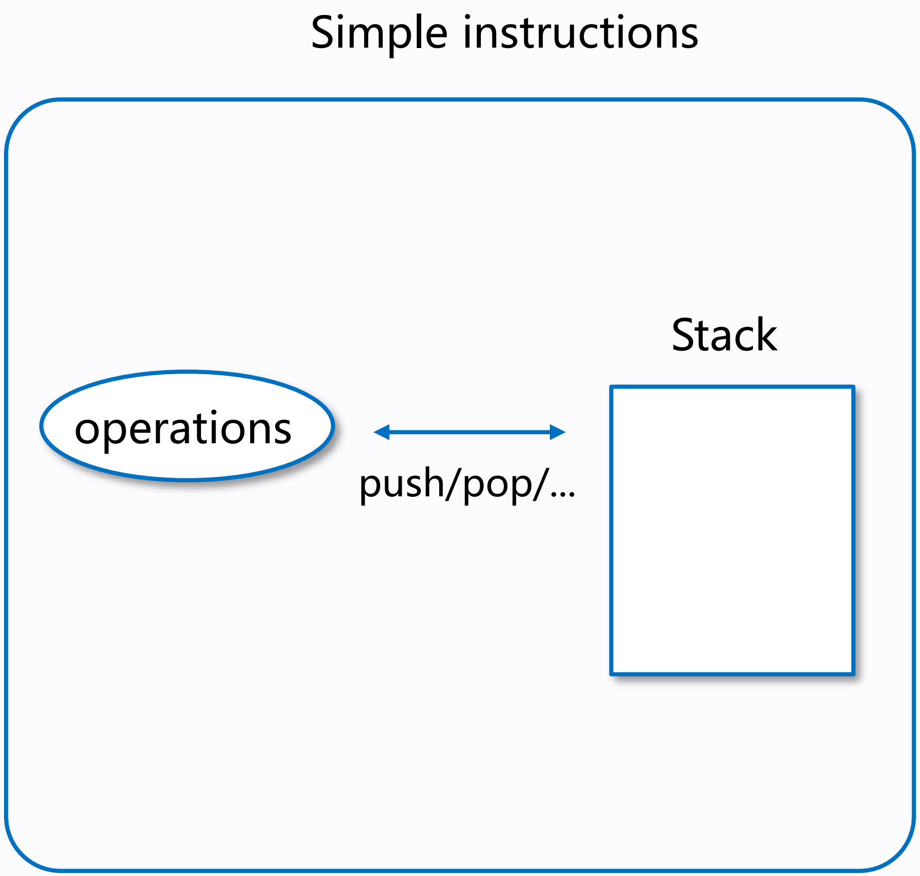
All integers are encoded using the LEB128 variable-length integer encoding.

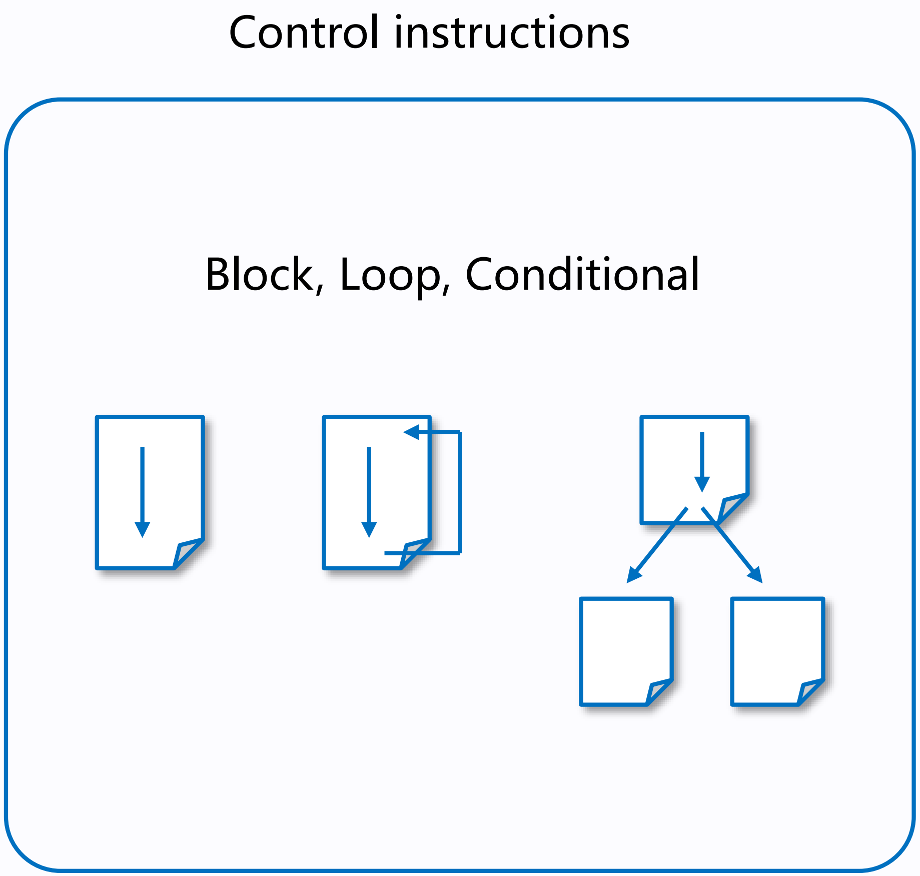
References : [8], [1] Ch.5, [2]

4. WebAssembly instructions

Instructions

Instructions



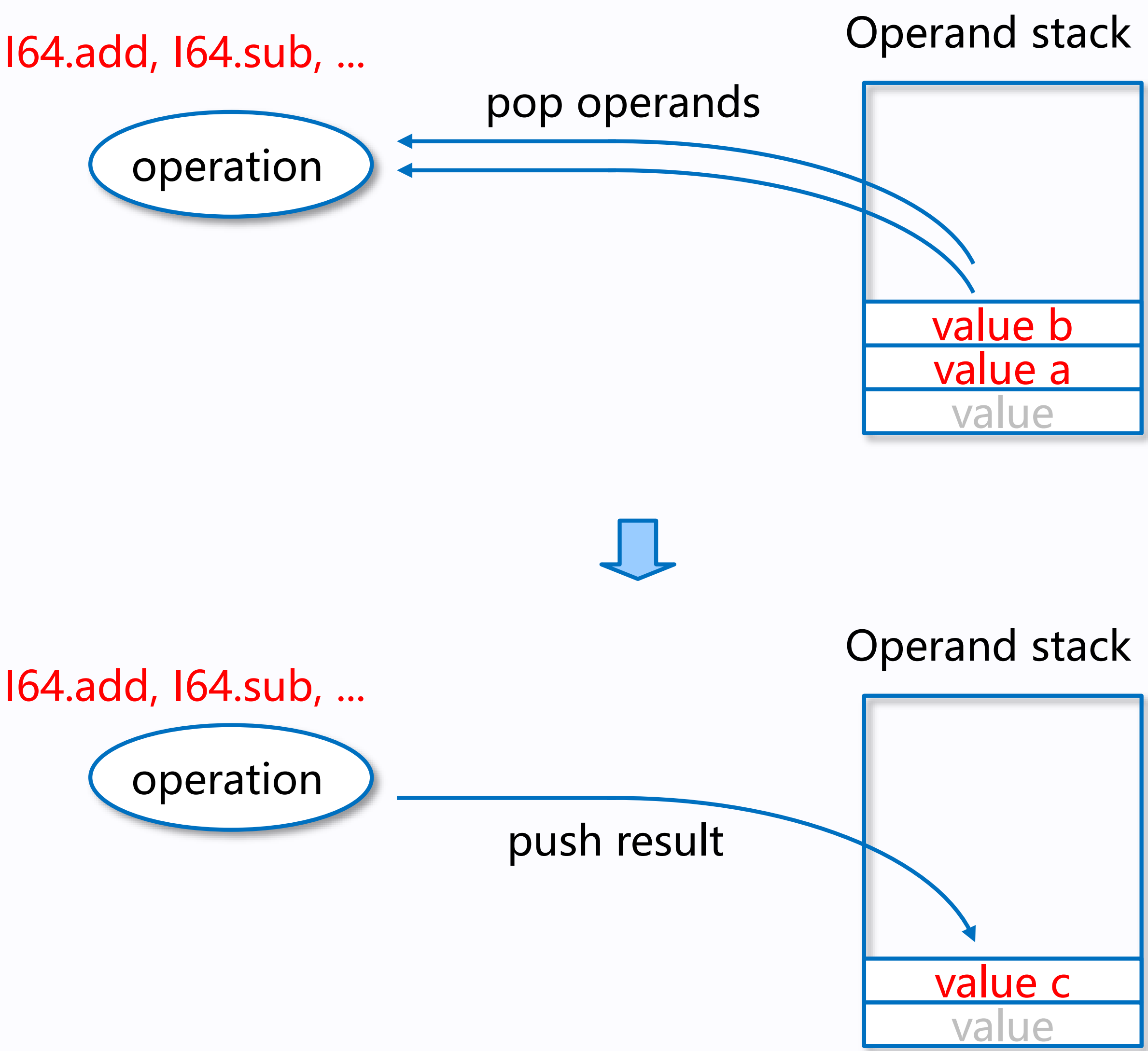


Instructions fall into two main categories.
Simple instructions perform basic operations on data.
Control instructions alter control flow.

References : [1] Ch.1., [2]

Simple instructions

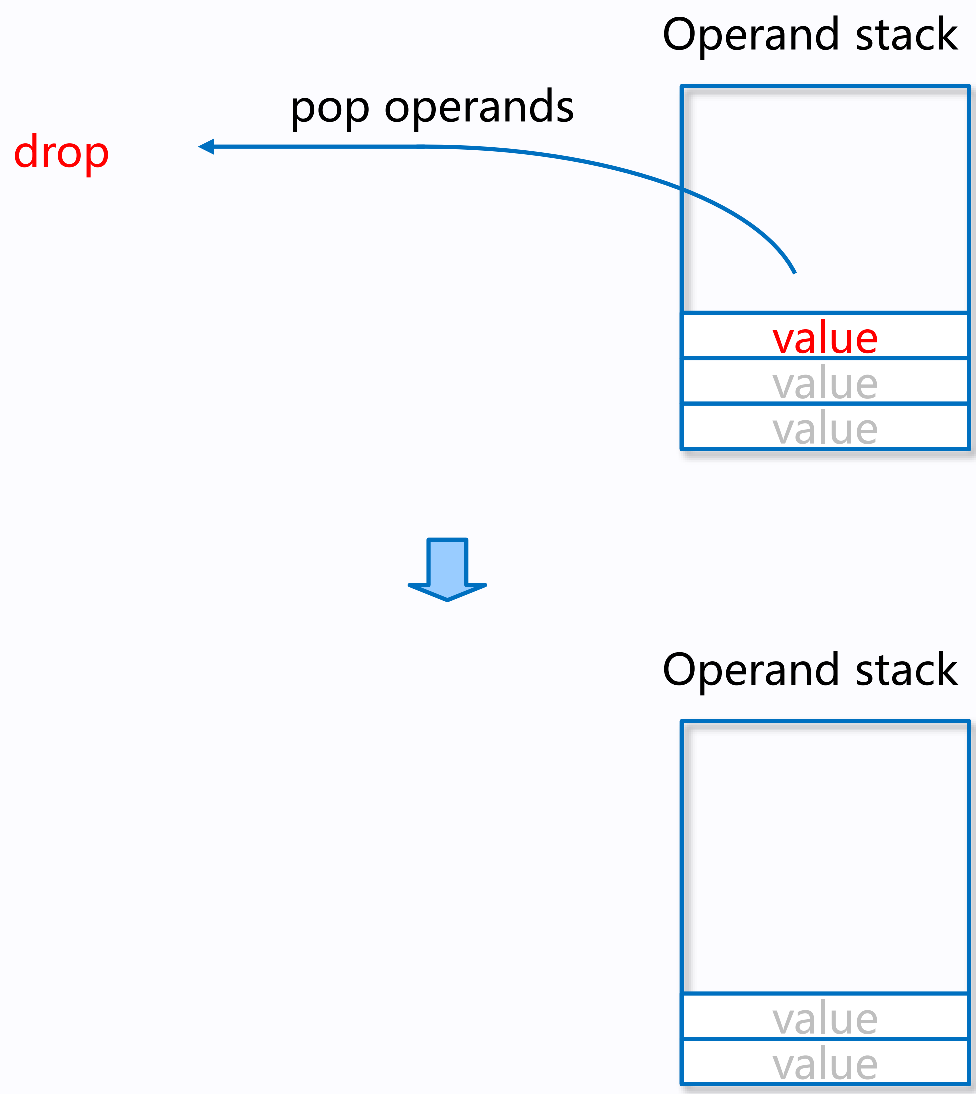
Numeric instructions



Numeric instructions pop arguments from the operand stack and push results back to it.

References : [1] Ch.2, Ch.4

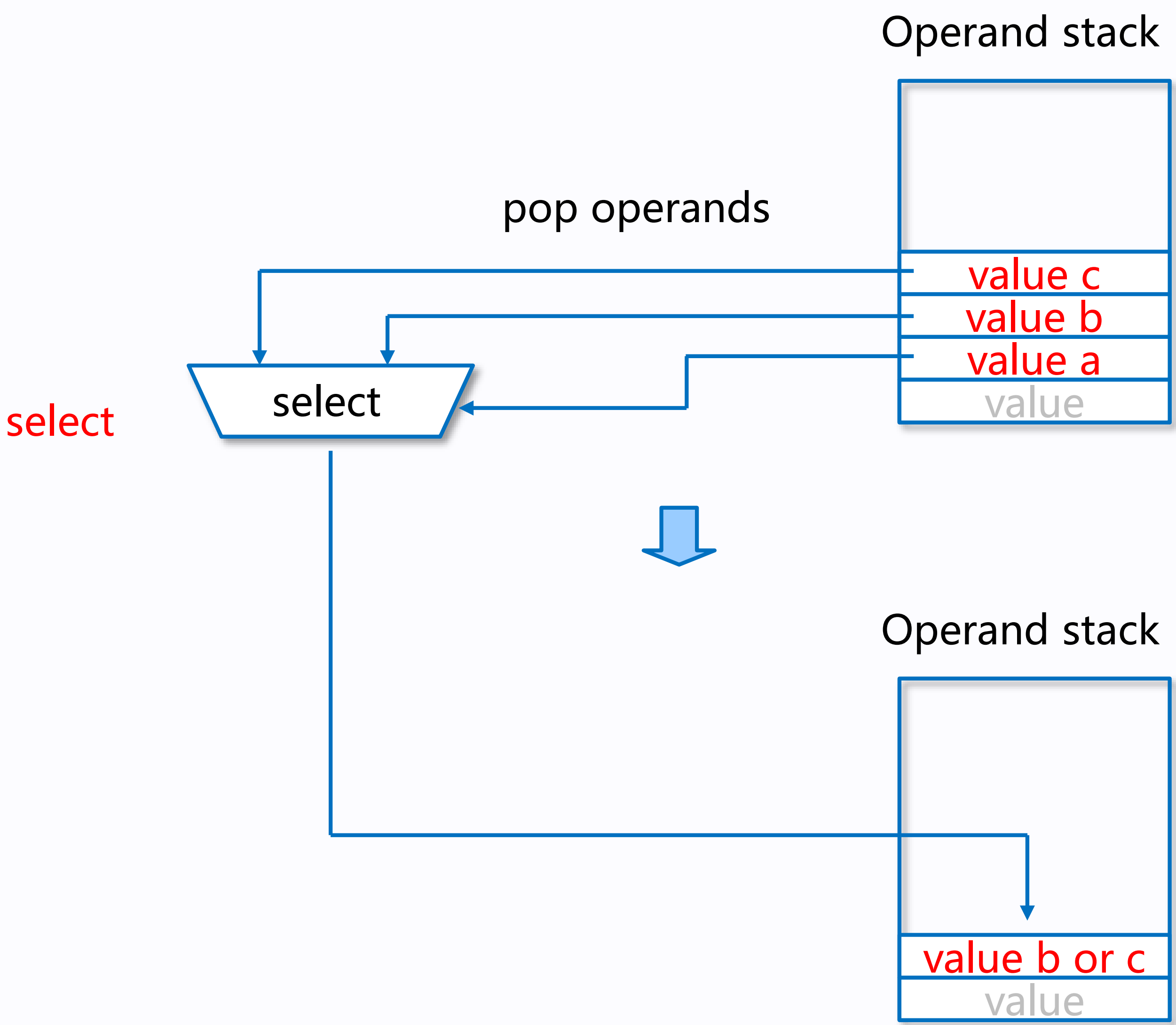
Parametric instructions : drop



The drop instruction simply throws away a single operand.

References : [1] Ch.2, Ch.4

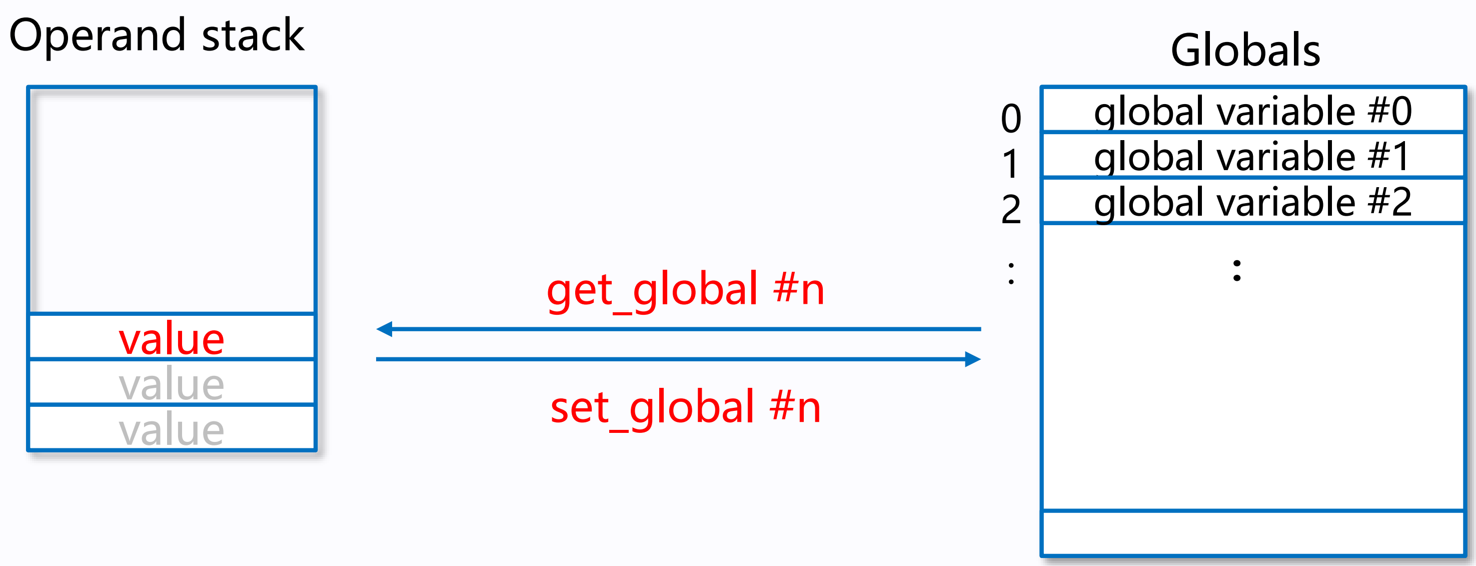
Parametric instructions : select



The select instruction selects one of its first two operands based on whether its third operand is zero or not.

References : [1] Ch.2, Ch.4

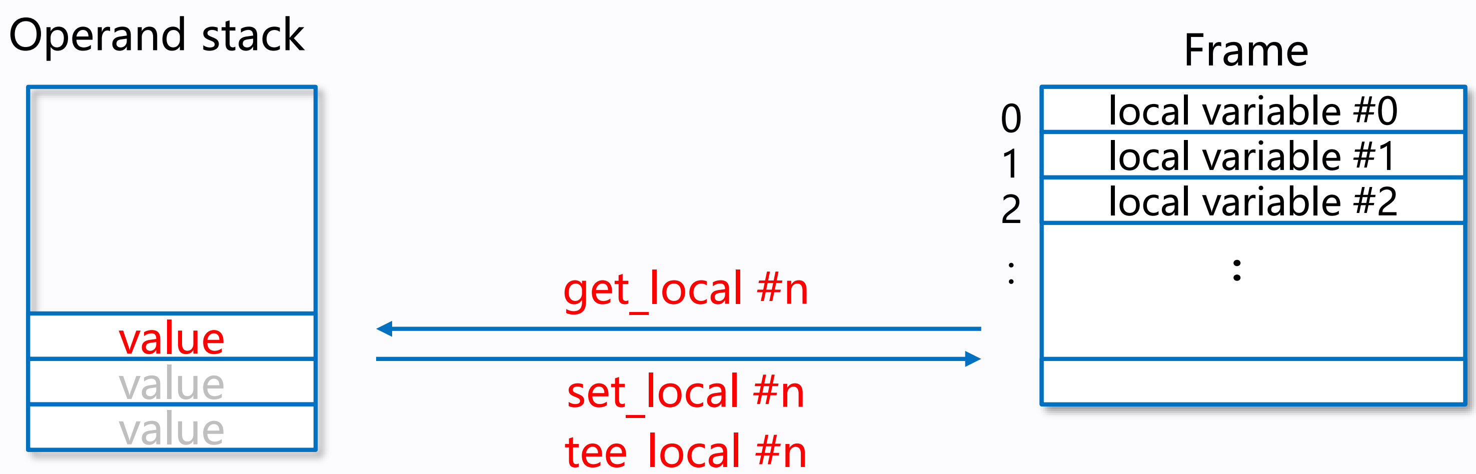
Global variable instructions



Global variable instructions get or set the values of variables.

References : [1] Ch.2, Ch.4

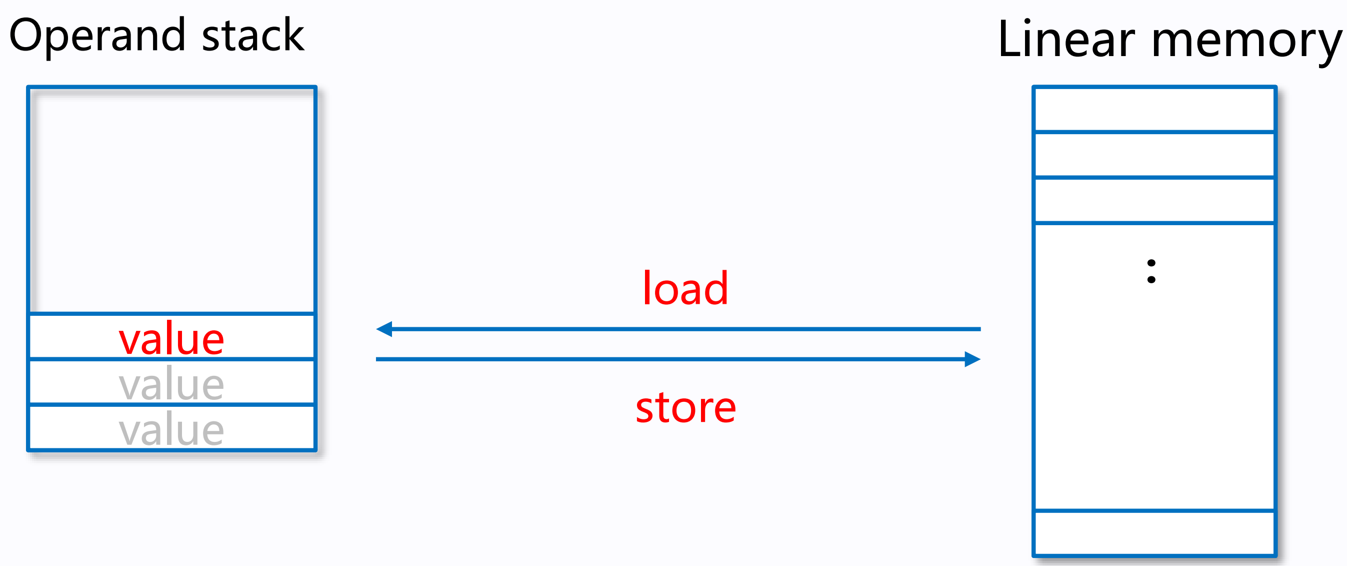
Local variable instructions



Local variable instructions get or set the values of variables.
(including function arguments)

References : [1] Ch.2, Ch.4

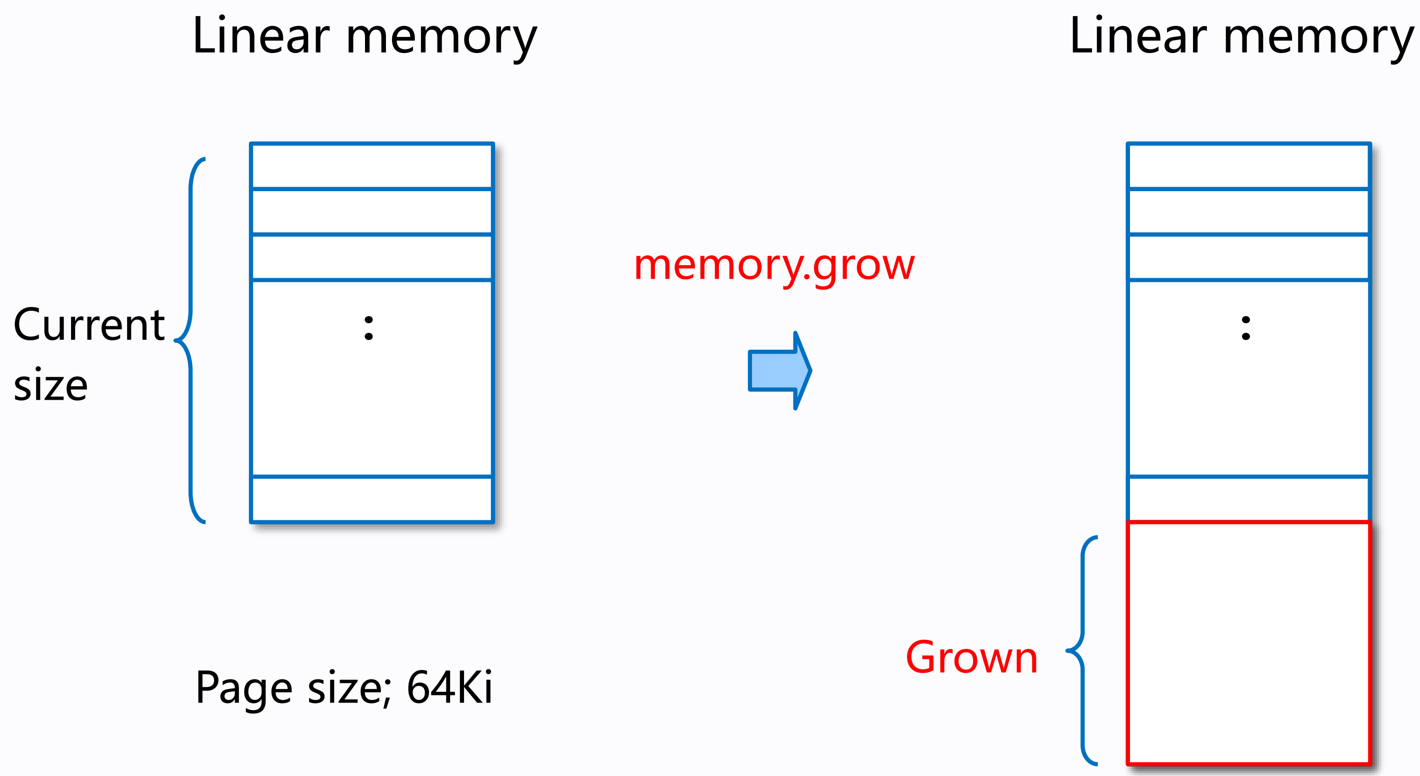
Memory instructions : load, store



Memory is accessed with load and store instructions for the different value types.

References : [1] Ch.2, Ch.4

Memory instructions : memory.grow

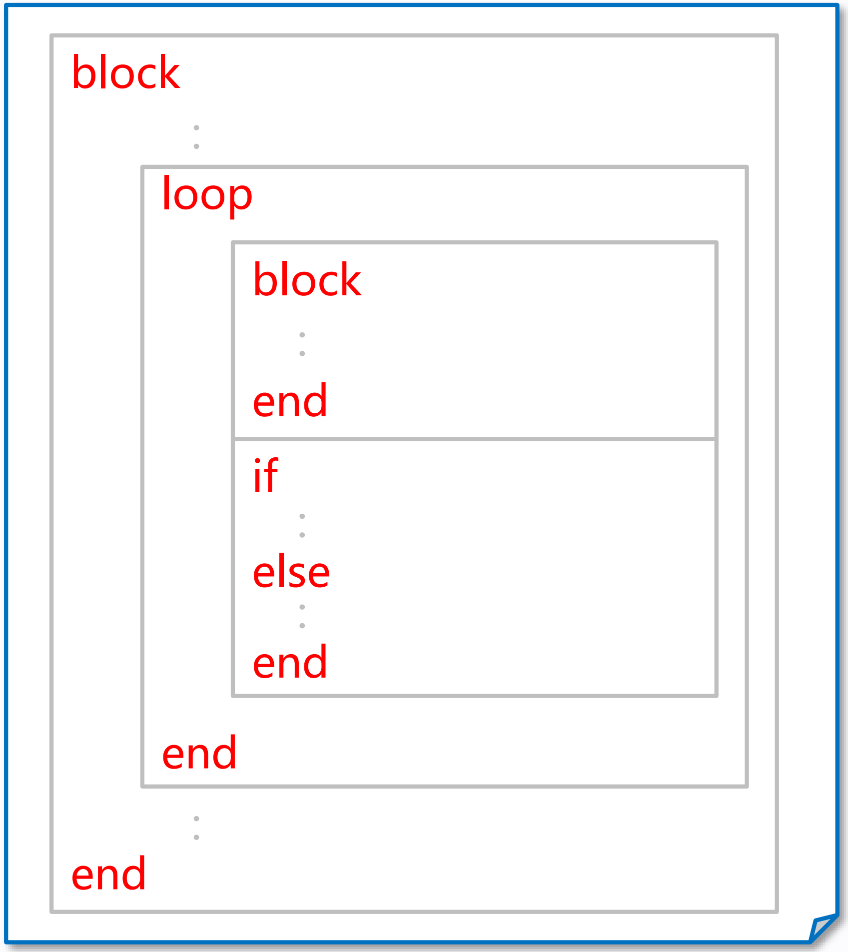


The memory.grow instruction grows memory by a given delta.
The memory.grow instruction operate in units of page size (64Ki).

References : [1] Ch.2, Ch.4

Control instructions

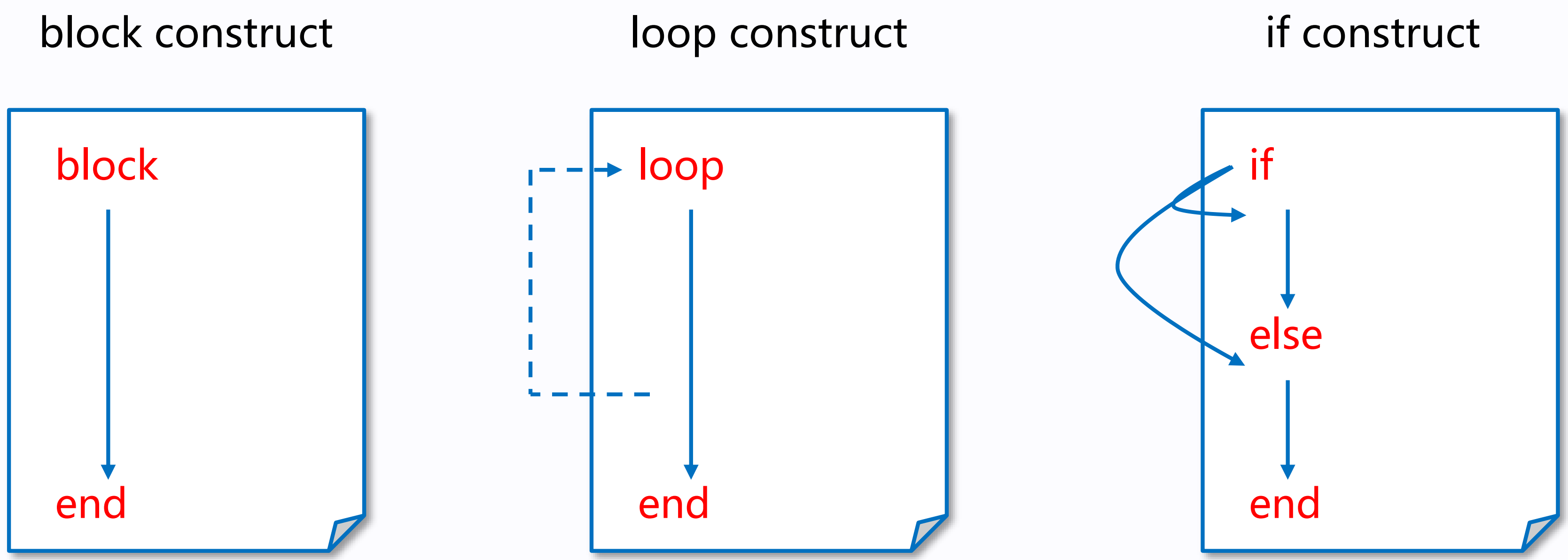
Control flow is structured



Control flow is expressed with well-nested constructs such as blocks, loops, and conditionals (if-else).
Structured control flow allows simpler and more efficient verification.

References : [1] Ch.2, Ch.4, [2]

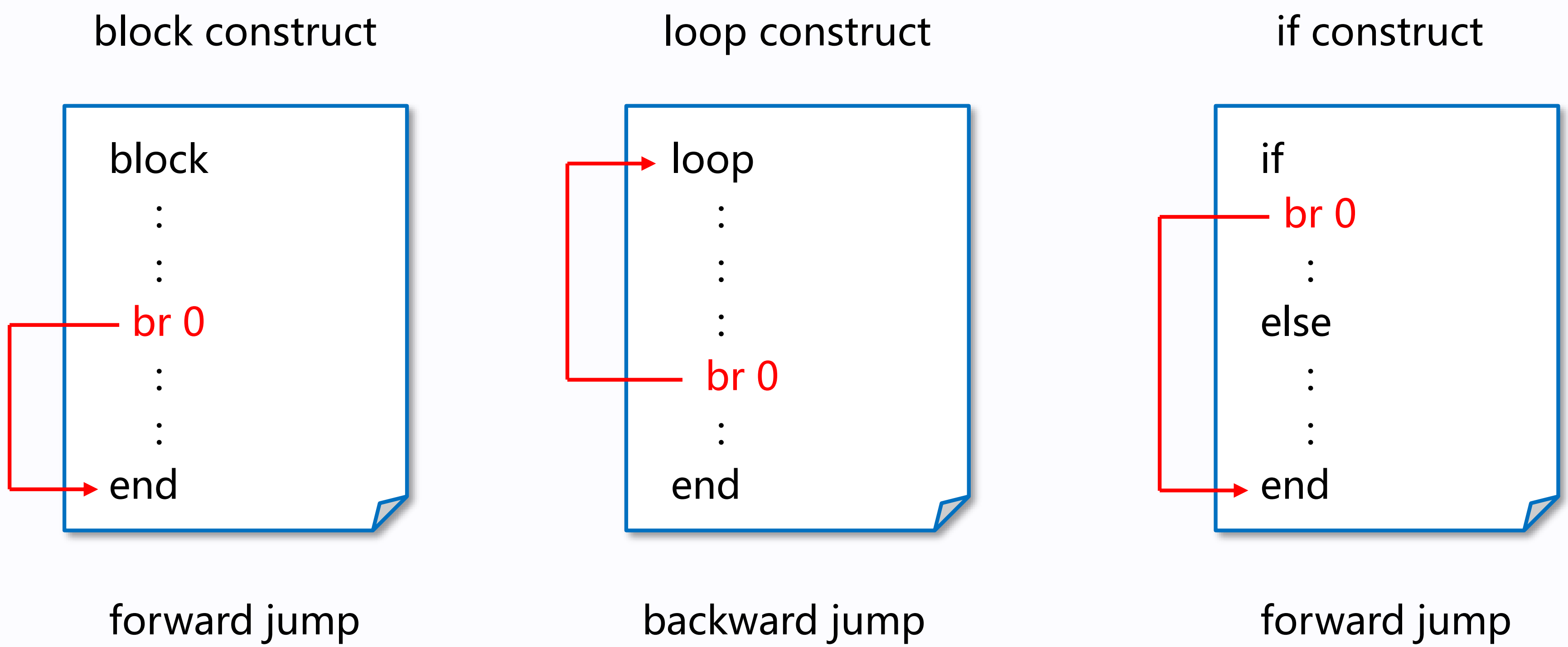
Structured control instructions



The block, loop and if instructions are structured control instructions.

References : [1] Ch.2, Ch.4, [2]

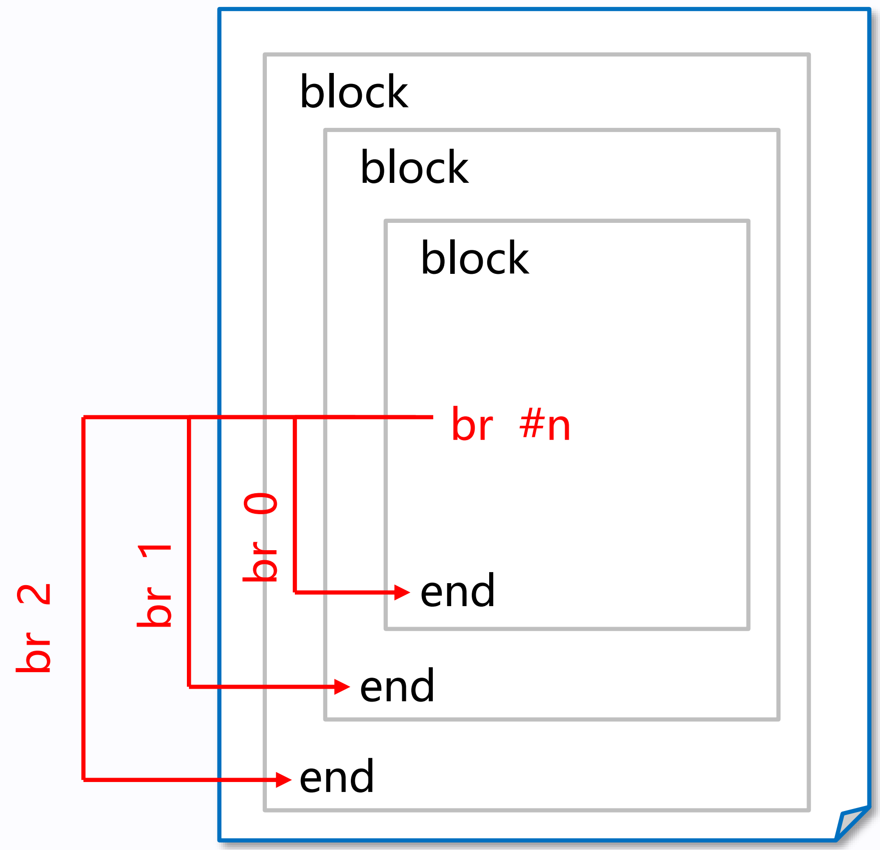
Control constructs and branch instruction



Branches can only target control constructs.
Intuitively, a branch targeting a block or if behaves like a break statement, while a branch targeting a loop behaves like a continue statement.

References : [1] Ch.2, Ch.4, [2]

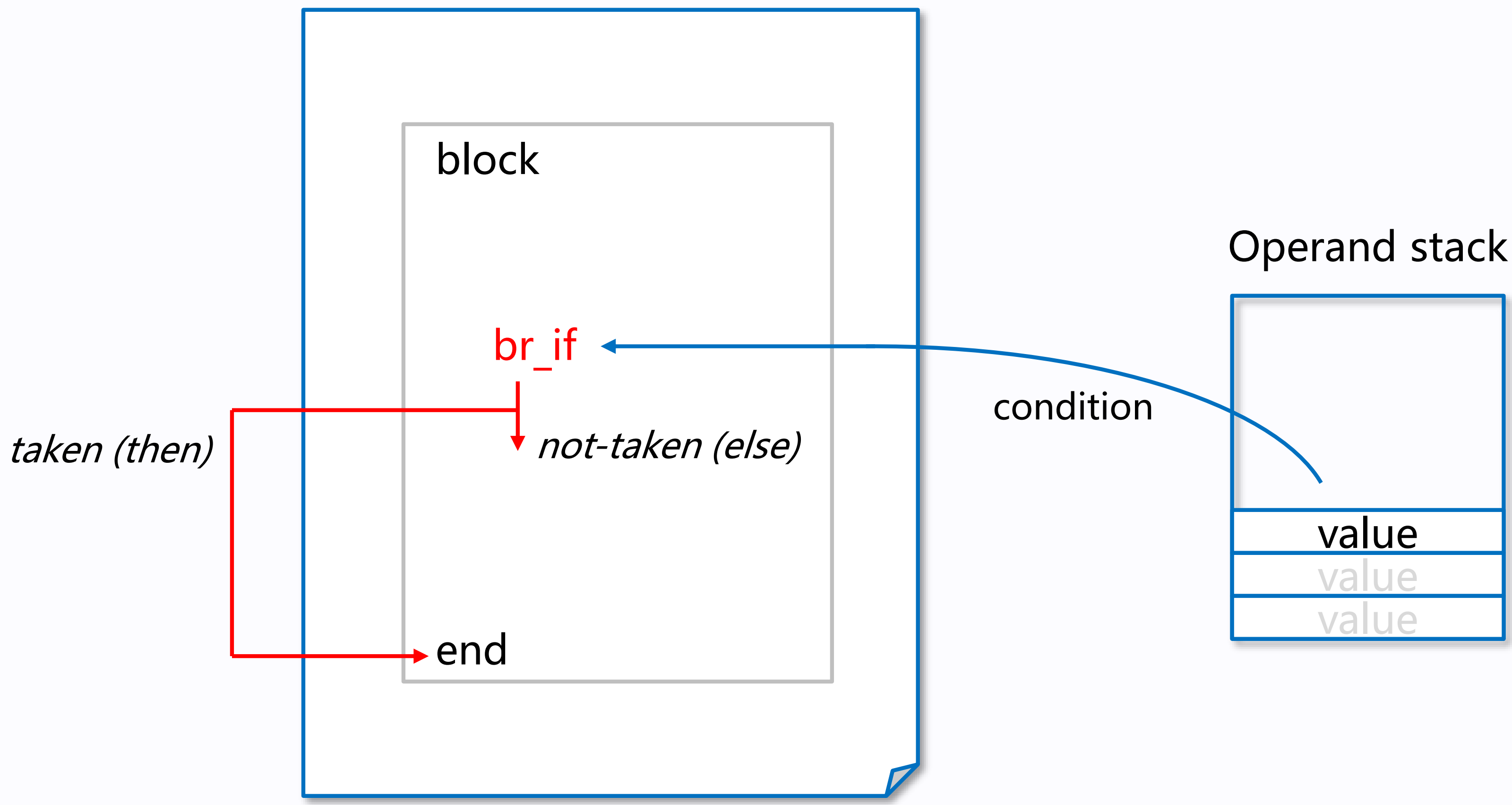
Nested constructs and branch instruction



Branches have "label" immediates.
It do not reference program positions in the instruction stream
but instead reference outer control constructs by relative nesting depth.

References : [1] Ch.2, Ch.4, [2]

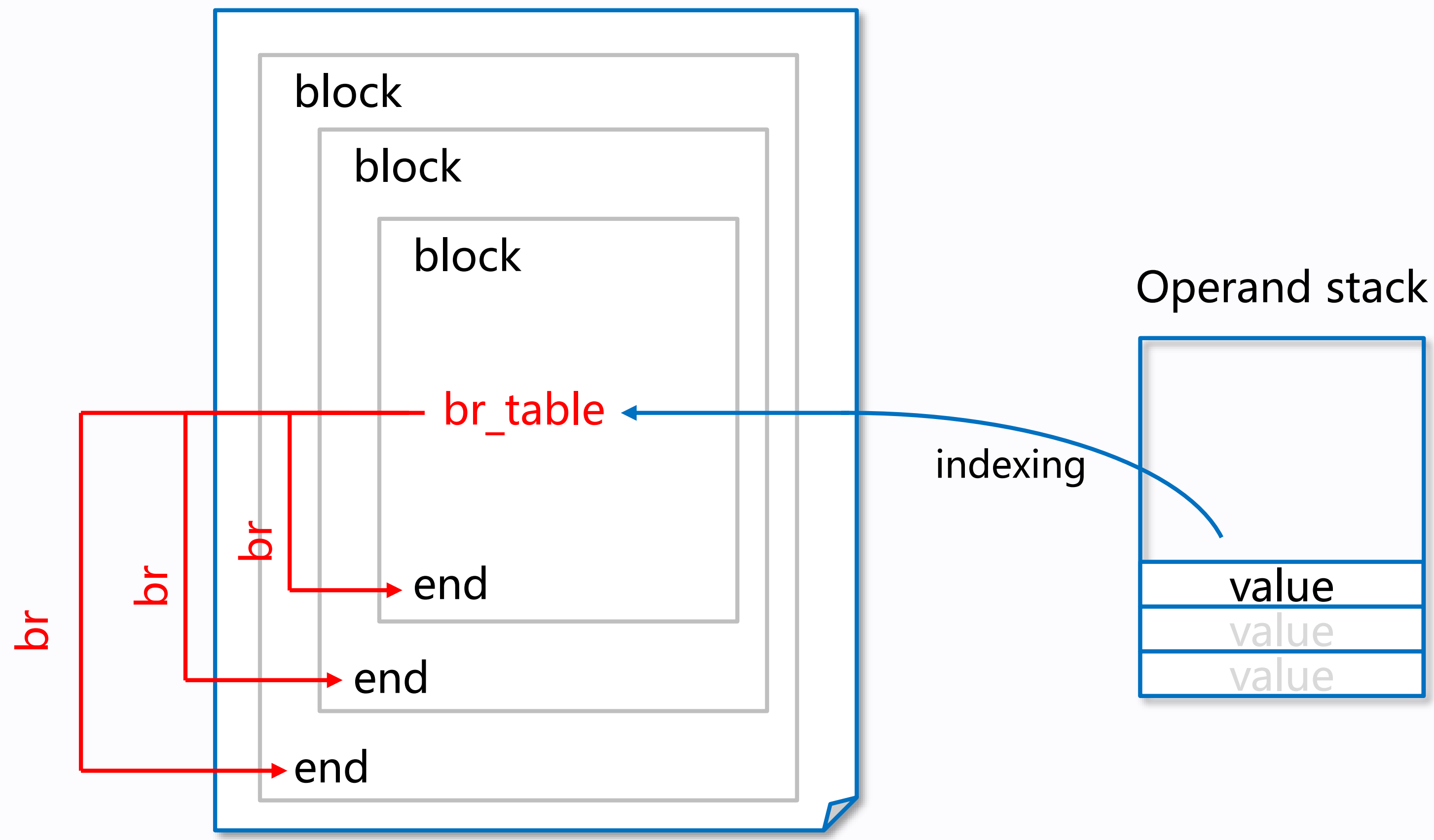
Conditional branch instruction



The br_if instruction performs a conditional branch.

References : [1] Ch.2, Ch.4, [2]

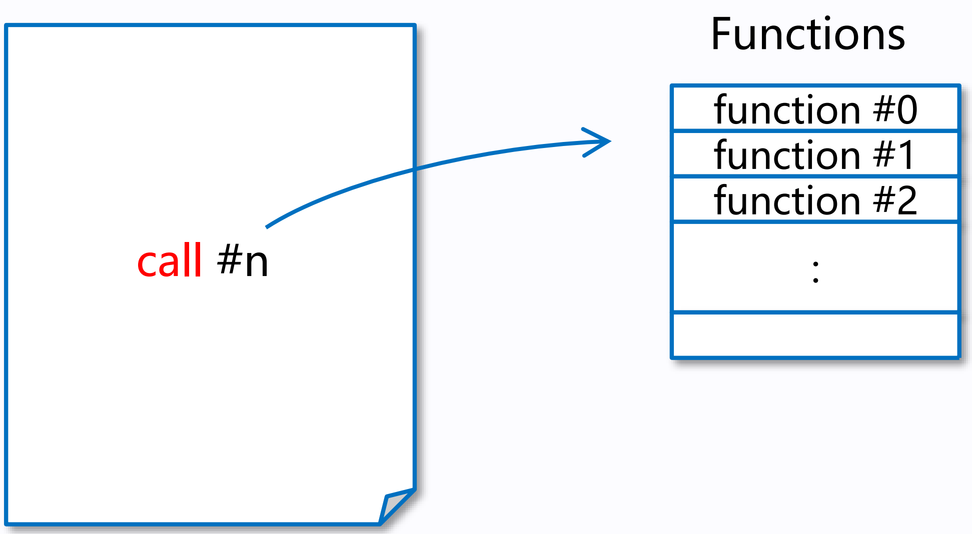
Table branch instruction



The br_table performs an indirect branch through an operand indexing into the label vector.

References : [1] Ch.2, Ch.4, [2]

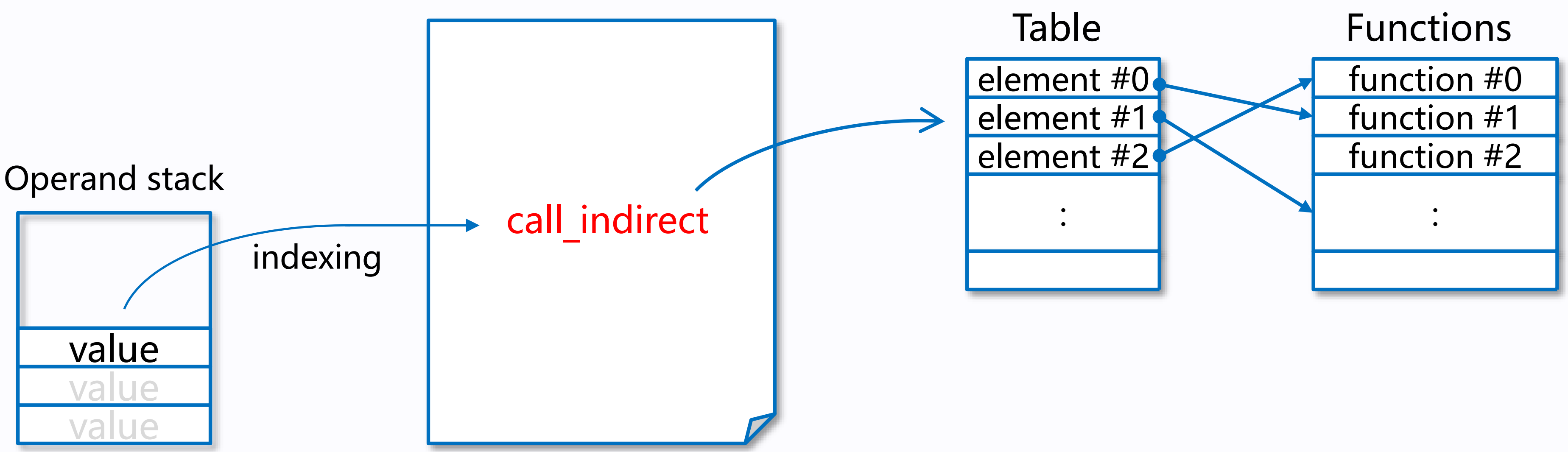
Call instruction



The call instruction invokes another function,
consuming the necessary arguments from the stack and returning the result values of the call.

References : [1] Ch.2, Ch.4, [2]

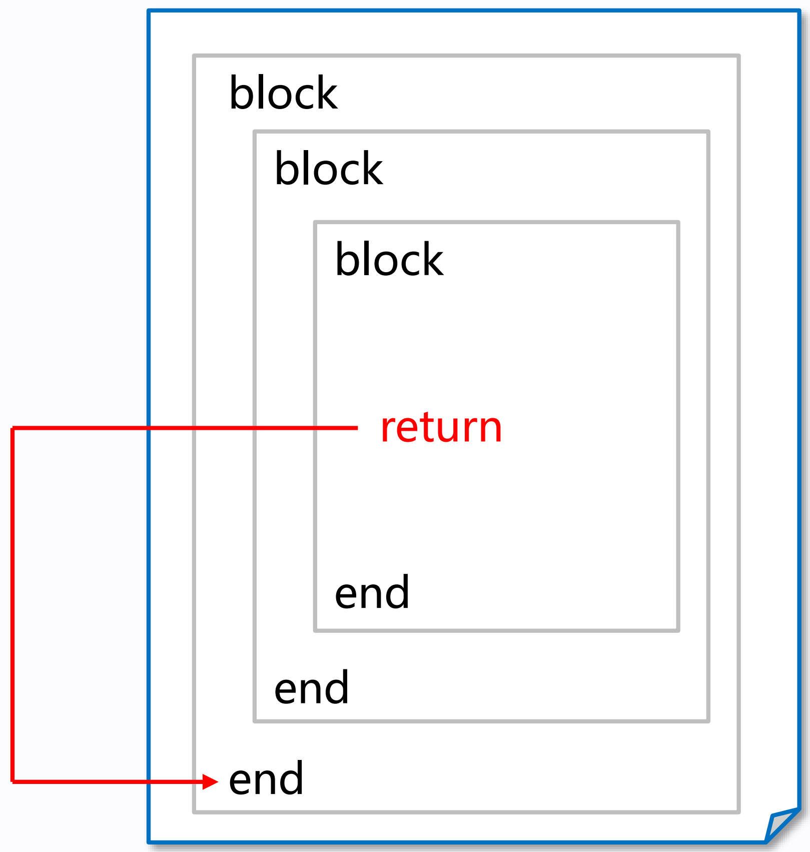
Indirect call instruction



The call_indirect instruction calls a function indirectly through an operand indexing into a table.

References : [1] Ch.2, Ch.4, [2]

Return instruction

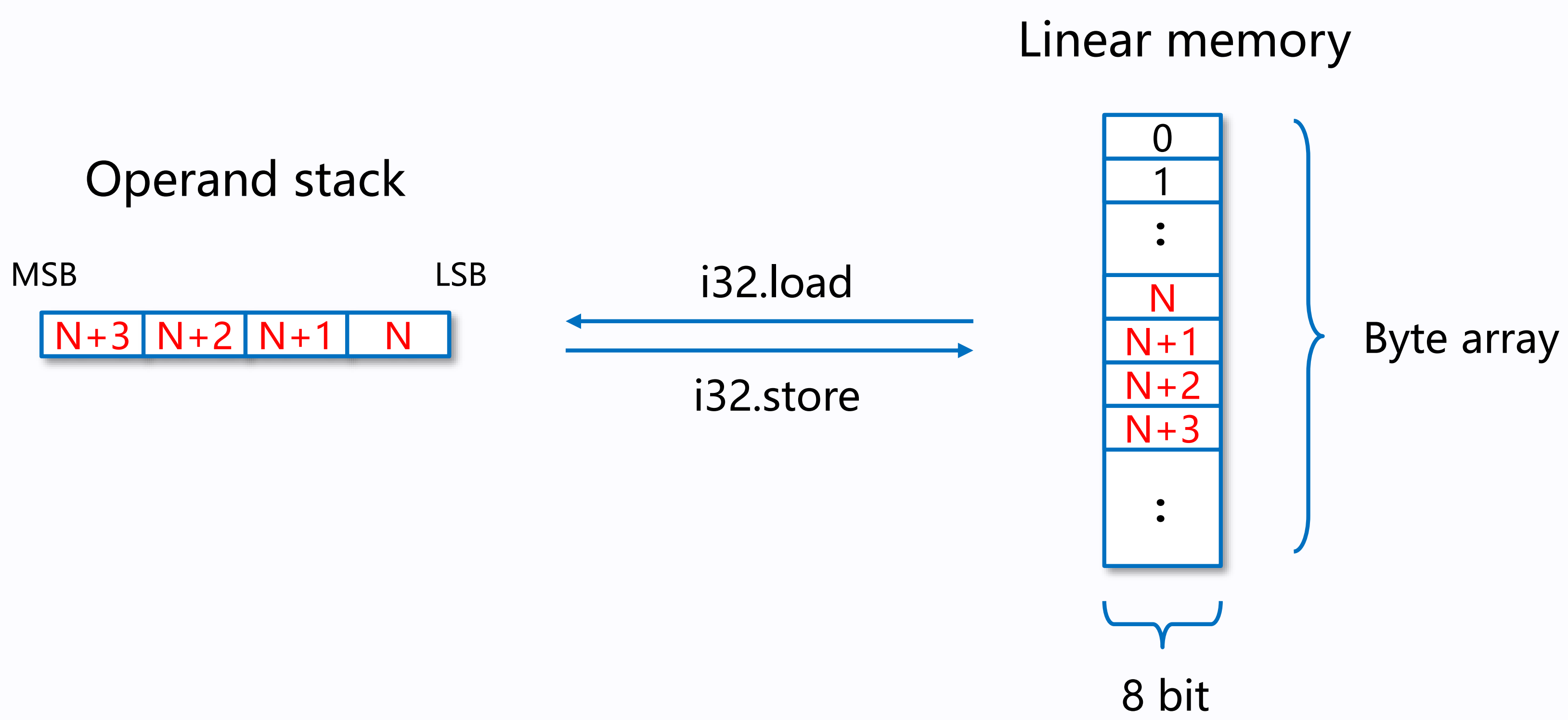


The return instruction is an unconditional branch to the outermost block, which implicitly is the body of the current function.

References : [1] Ch.2, Ch.4, [2]

Byte order

Endian



WebAssembly abstract machine is little endian byte order.
When a number is stored into memory, it is converted into a sequence of bytes in little endian byte order.

References : [1] Ch. 4, [2]

Appendix A

Appendix A

Operational semantics

Operational semantics

[spec, 4.1-]

@@@ see tegaki, No.11-12, A1

[spec, 4.1]

Execution behavior is defined in terms of an abstract machine that models the program state. It includes a stack,
which records operand values and control constructs, and an abstract store containing global state.

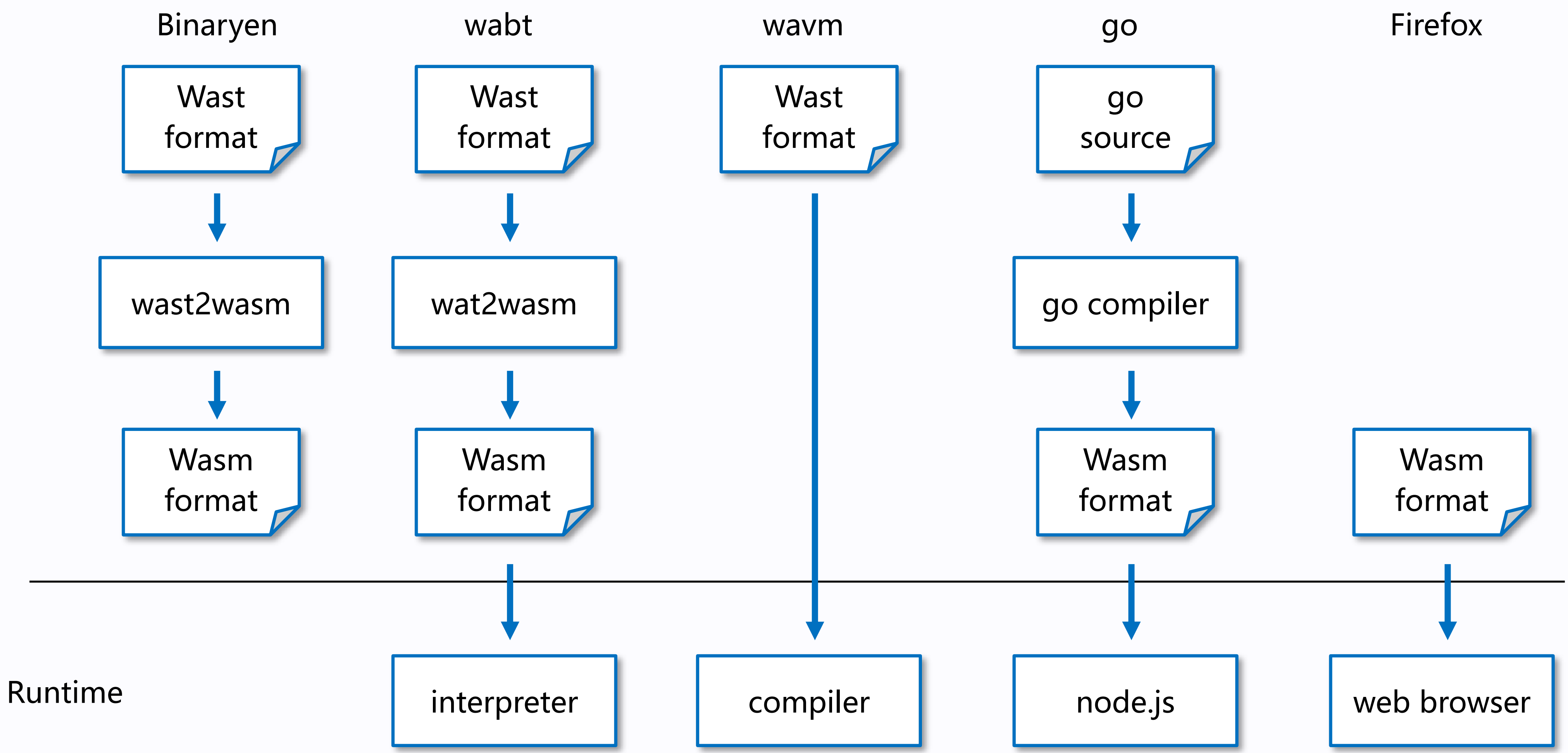
References : [1] Ch.4.1

Appendix B

Appendix B

Implementations

Implementations



References : [C1], [C2], [C3], [C4], [C5]

Rererence interpreter : spec

https://github.com/WebAssembly/spec
[interpreter/exec/eval.ml]

```
let rec step (c : config) : config =
  let {frame; code = vs, es; _} = c in
  let e = List.hd es in
  let vs', es' =
    match e.it, vs with
    | Plain e', vs ->
      (match e', vs with
      | Unreachable, vs ->
        vs, [Trapping "unreachable executed" @@ e.at]

      | Nop, vs ->
        vs, []

      | Block (ts, es'), vs ->
        vs, [Label (List.length ts, [], ([], List.map plain es')) @@ e.at]

      | Loop (ts, es'), vs ->
        vs, [Label (0, [e' @@ e.at], ([], List.map plain es')) @@ e.at]

      | If (ts, es1, es2), I32 0l :: vs' ->
        vs', [Plain (Block (ts, es2)) @@ e.at]
```

References : [C1]

Interpreter : wabt

(@@@)

https://github.com/WebAssembly/spec
[src/interp/interp.cc]

```
Result Thread::Run(int num_instructions) {
  Result result = Result::Ok;

  const uint8_t* istream = GetIstream();
  const uint8_t* pc = &istream[pc_];
  for (int i = 0; i < num_instructions; ++i) {
    Opcode opcode = ReadOpcode(&pc);
    assert(!opcode.IsInvalid());
    switch (opcode) {
      case Opcode::Select: {
        uint32_t cond = Pop<uint32_t>();
        Value false_ = Pop();
        Value true_ = Pop();
        CHECK_TRAP(Push(cond ? true_ : false_));
        break;
      }

      case Opcode::Br:
        GOTO(ReadU32(&pc));
        break;

      case Opcode::BrIf: {
```

References : [C3]

Stand-alone VM : wavm

(@@@)

[@@@]

```
@@@
```

References : [C4]

Web browser : Firefox

(@@@)

[@@@]

```
@@@
```

References : [C5]

Appendix B

CLI development utilities

Assemble wast to wasm

Binaryen : Assemble Wast(Wat) text code to Wasm binary

```
$ wasm-as sample.wast
```

wabt : Assemble Wast(Wat) text code to Wasm binary

```
$ wat2wasm sample.wast
```

```
$ wat2wasm -v sample.wast
```

References : [C2], [C3]

Disassemble from wasm

Binaryen : Disassemble from Wasm binary

```
$ wasm-dis sample.wasm
```

wabt : Disassemble from Wasm binary

```
$ wasm2wat sample.wasm
```

```
$ wasm-objdump -d sample.wasm
```

References : [C2], [C3]

Desugar wast

wabt : Desugar from wast to wast

```
$ wat-desugar sample.wast
```

References : [C3]

Module information

wabt : Dump module infromation

```
$ wasm-objdump -s sample.wasm
```

```
$ wasm-objdump -x sample.wasm
```

References : [C3]

Run wasm and wast

wabt : Run wasm with trace

```
$ wasm-interp --run-all-exports --trace sample.wasm
```

wavm : Run wast

```
$ wavm-run sample.wast
```

References : [C3], [c4]

Appendix B

Test suites

Test suites and wast examples

https://github.com/WebAssembly/spec
[test/core]

```
README.md              fac.wast                    names.wast
address.wast           float_exprs.wast            nop.wast
align.wast             float_literals.wast         return.wast
binary.wast            float_memory.wast           run.py*
block.wast             float_misc.wast             select.wast
br.wast                forward.wast                set_local.wast
br_if.wast             func.wast                   skip-stack-guard-page.wast
br_table.wast          func_ptrs.wast              stack.wast
break-drop.wast        get_local.wast              start.wast
call.wast              globals.wast                store_retval.wast
call_indirect.wast     i32.wast                    switch.wast
comments.wast          i64.wast                    tee_local.wast
const.wast             if.wast                     token.wast
conversions.wast       imports.wast                traps.wast
custom.wast            inline-module.wast          type.wast
data.wast              int_exprs.wast              typecheck.wast
elem.wast              int_literals.wast           unreachable.wast
endianness.wast        labels.wast                 unreached-invalid.wast
exports.wast           left-to-right.wast          unwind.wast
f32.wast               linking.wast                utf8-custom-section-id.wast
f32_bitwise.wast       loop.wast                   utf8-import-field.wast
f32_cmp.wast           memory.wast                 utf8-import-module.wast
f64.wast               memory_grow.wast            utf8-invalid-encoding.wast
f64_bitwise.wast       memory_redundancy.wast
f64_cmp.wast           memory_trap.wast
```

References : [C1]

Appendix B

Desugar examples

Appendix C

Future

Future directions

* zero-cost exception, threads, SIMD

* tail call, stack switching, coroutines

* garbage collectors

References : [2], [3]

References

References

[1] WebAssembly Specification Release 1.0 (Draft, last updated Oct 31, 2018)
https://webassembly.github.io/spec/core/

[2] Bringing the Web up to Speed with WebAssembly
https://github.com/WebAssembly/spec/blob/master/papers/pldi2017.pdf

[3] WebAssembly High-Level Goals
https://webassembly.org/docs/high-level-goals/

[4] Design Rationale
https://webassembly.org/docs/rationale/

[5] Modules
https://webassembly.org/docs/modules/

[6] WebAssembly Concepts
https://developer.mozilla.org/en-US/docs/WebAssembly/Concepts

[7] Understanding WebAssembly text format
https://developer.mozilla.org/en-US/docs/WebAssembly/Understanding_the_text_format

[8] LEB128
https://en.wikipedia.org/wiki/LEB128

References

[C1] Spec: WebAssembly Interpreter
https://github.com/WebAssembly/spec/tree/master/interpreter

[C2] Binaryen
https://github.com/WebAssembly/binaryen

[C3] WABT: The WebAssembly Binary Toolkit
https://github.com/WebAssembly/wabt

[C4] WebAssembly Virtual Machine (WAVM)
https://github.com/WAVM/WAVM

[C5] mozilla/gecko-dev (Firefox)
https://github.com/mozilla/gecko-dev

Drop figures

Architecture layer

Abstract machine
(Virtual machine)

WebAssembly Virtual ISA

* Instruction set
* Binary encoding
* Validation
* Execution semantics

Physical processor

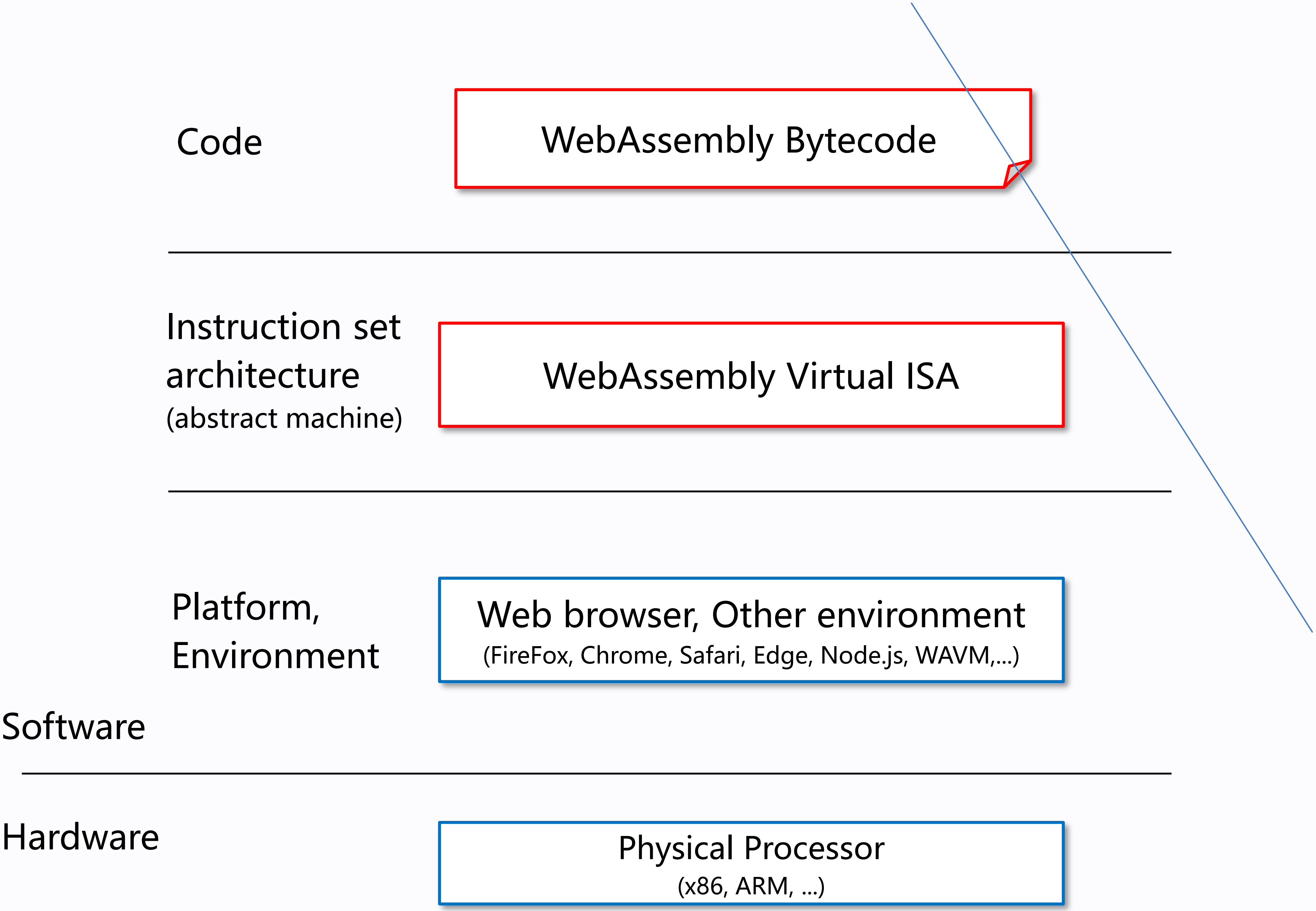
x86 ISA, ARM ISA, ...

[spec, 1.1.2]

WebAssembly is a virtual instruction set architecture (virtual ISA).

References : [1] Ch.1.1

WebAssembly is a virtual instruction set architecture



References : [1] Ch.1.1

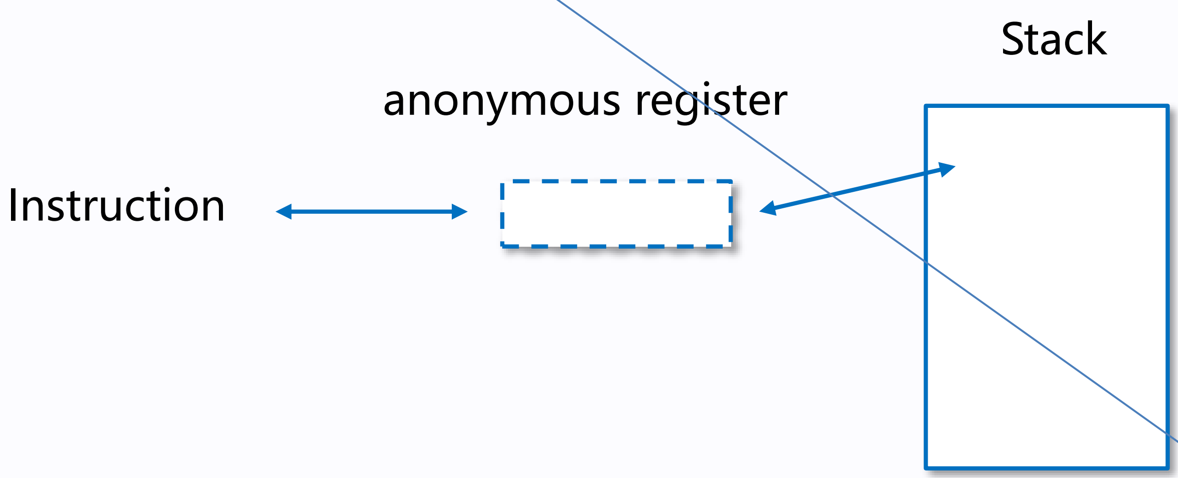
WebAssembly Spec of Core and API

Core ISA	WebAssembly Core	ISA Binary encoding Validation Exec
Environment	WebAssembly API	invoke interact

References : [1] Ch.1.1

Stack machine

The computational model of WebAssembly is based on a stack machine.
Code ha meirei no sequence. inorder.
Anmokuno operand stack manipulate.
Stack, anonymous register

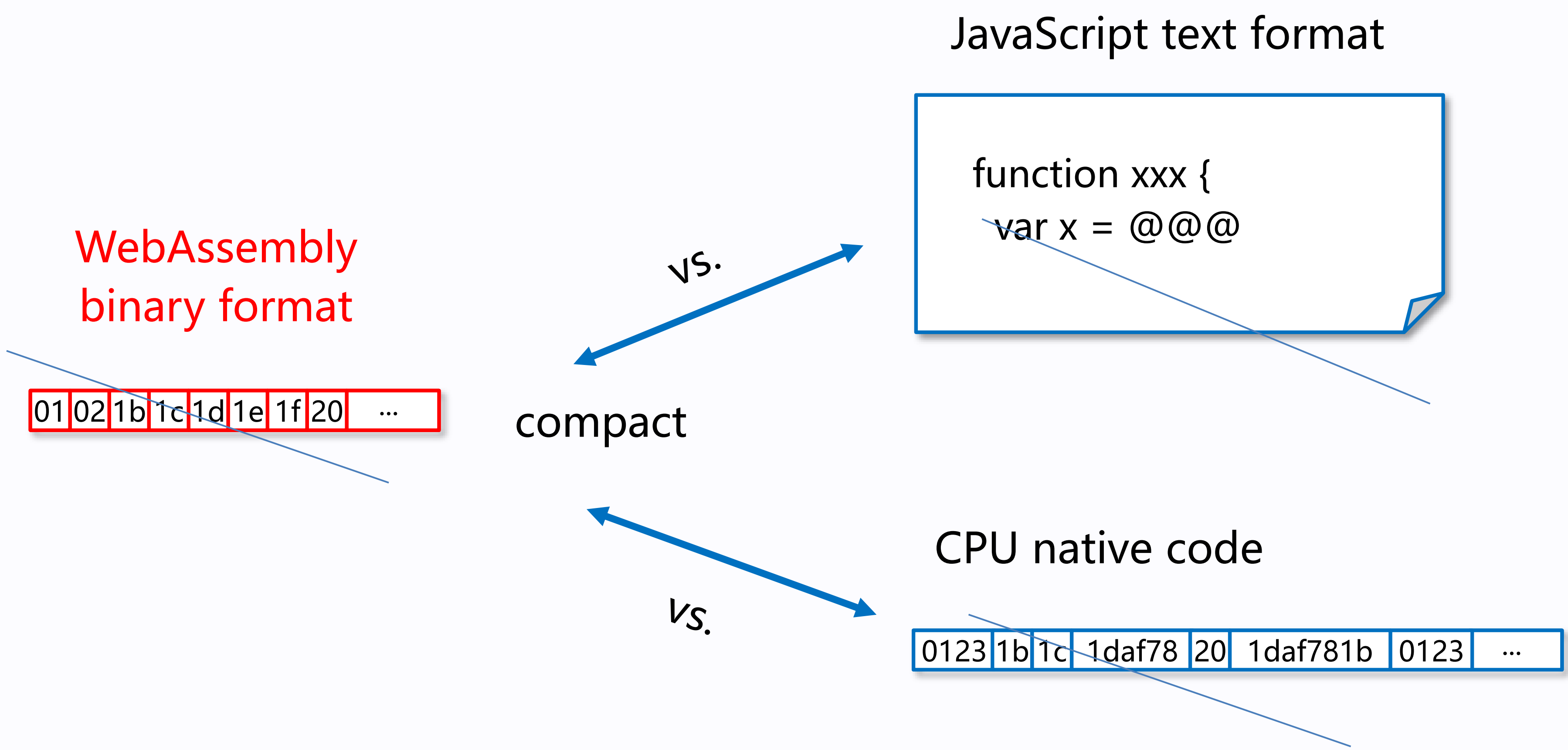


[spec, 1.2.1]

The computational model of WebAssembly is based on a stack machine.

References : [1] Ch.1.2

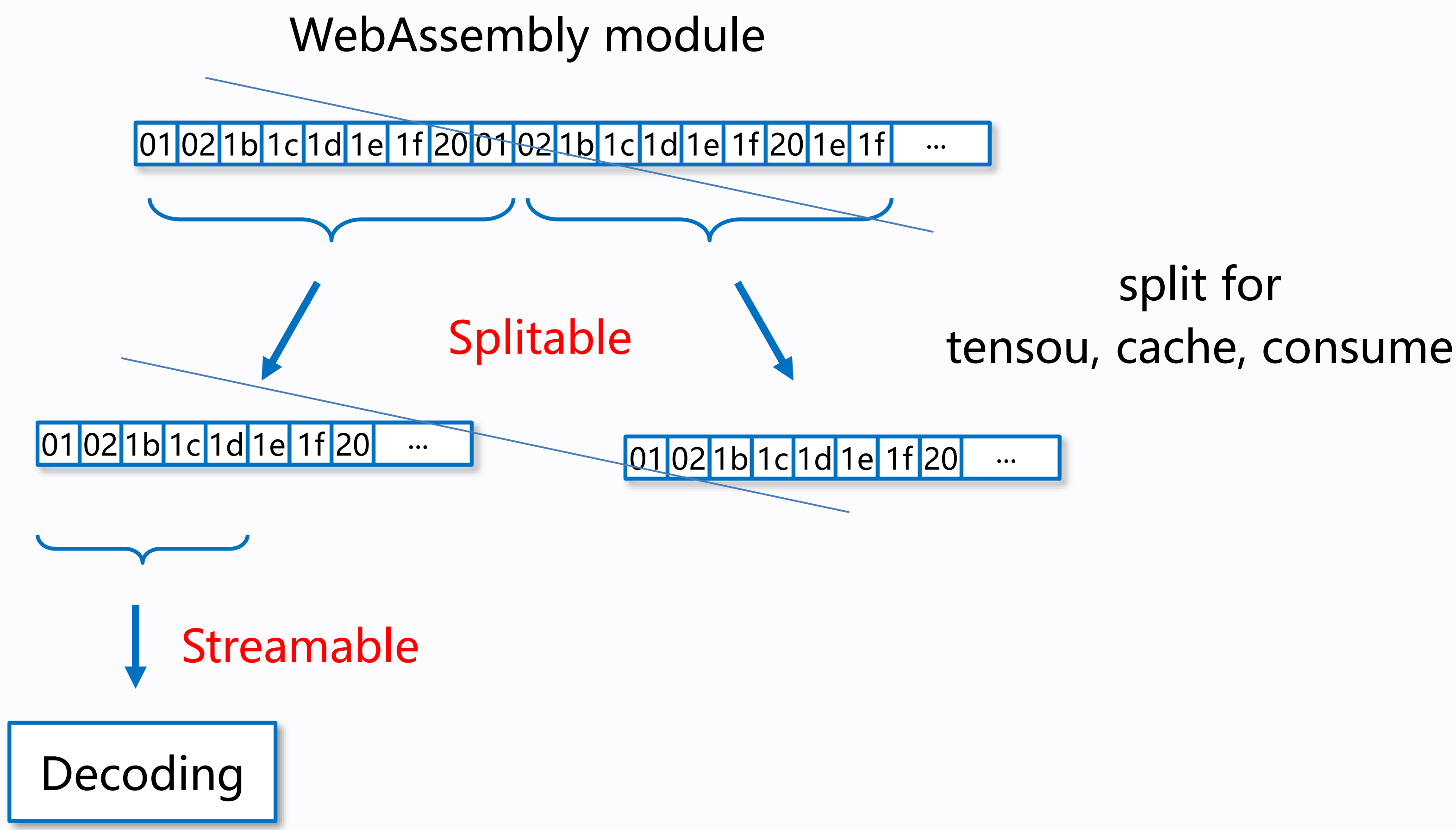
WebAssembly code is compact



@@@

References : [1] Ch.1.1

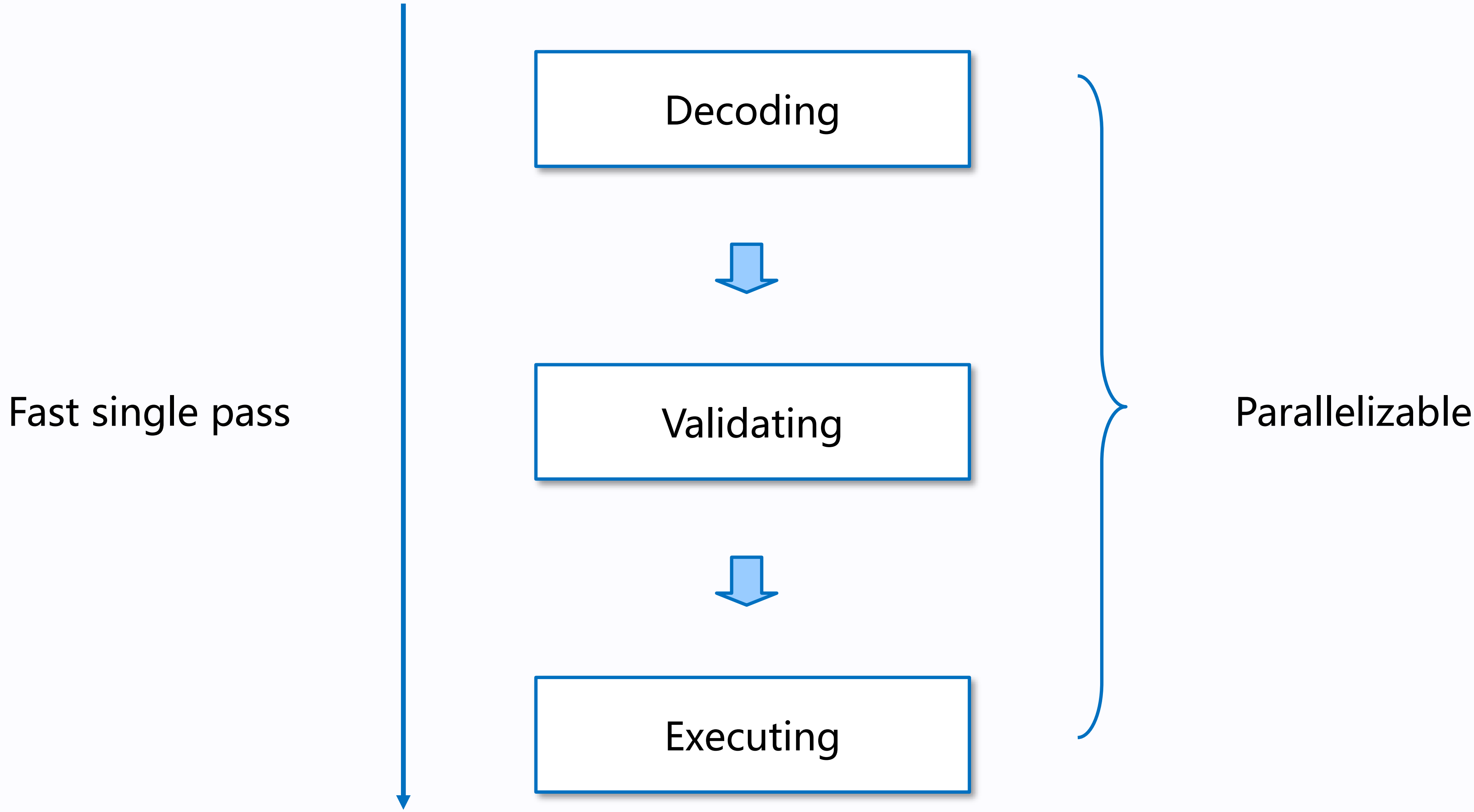
Split and streamable



@@@

References : [1] Ch.1.1

Efficient semantics

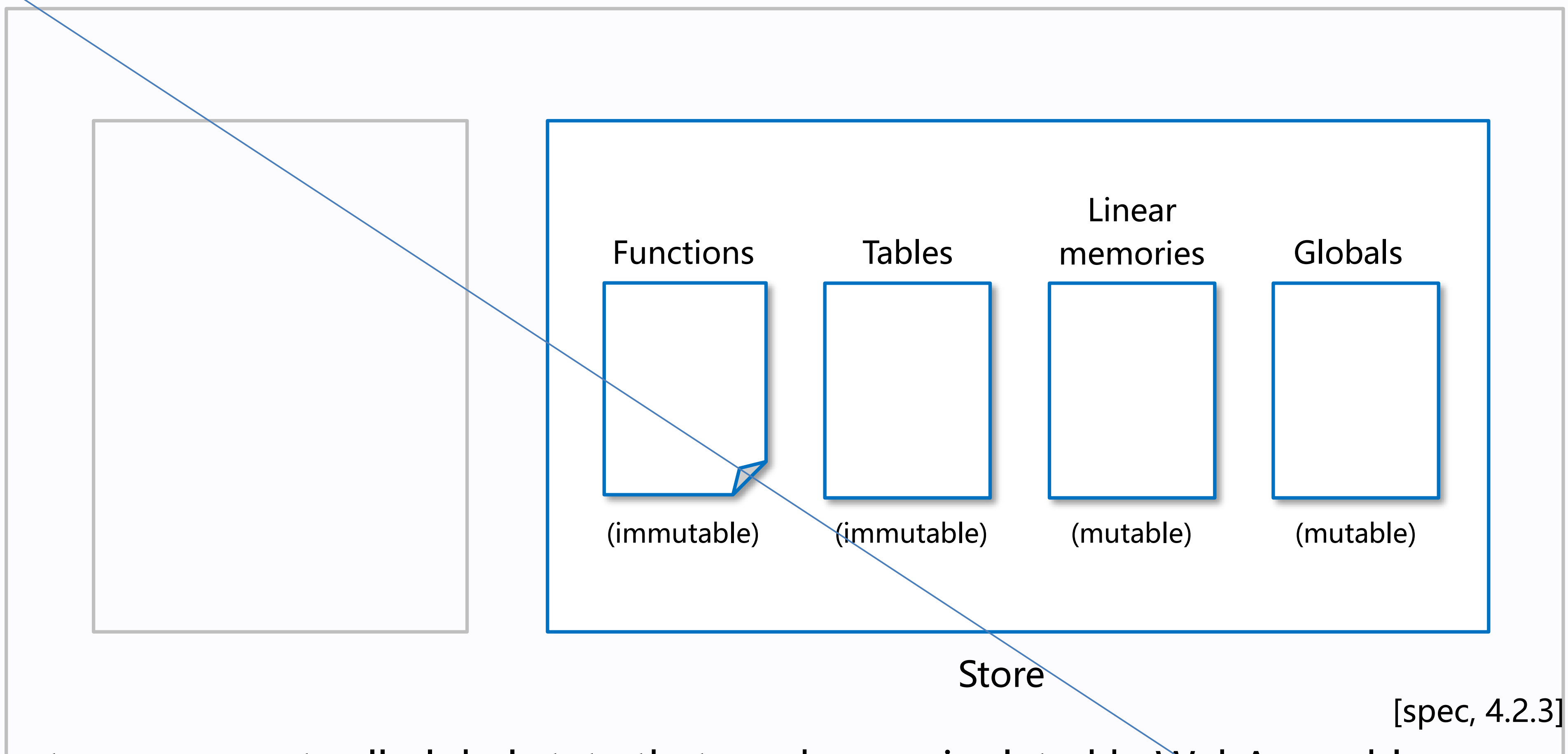


[spec, 1.2.2]

Conceptually, the semantics of WebAssembly is divided into three phases.

References : [1] Ch.1.1

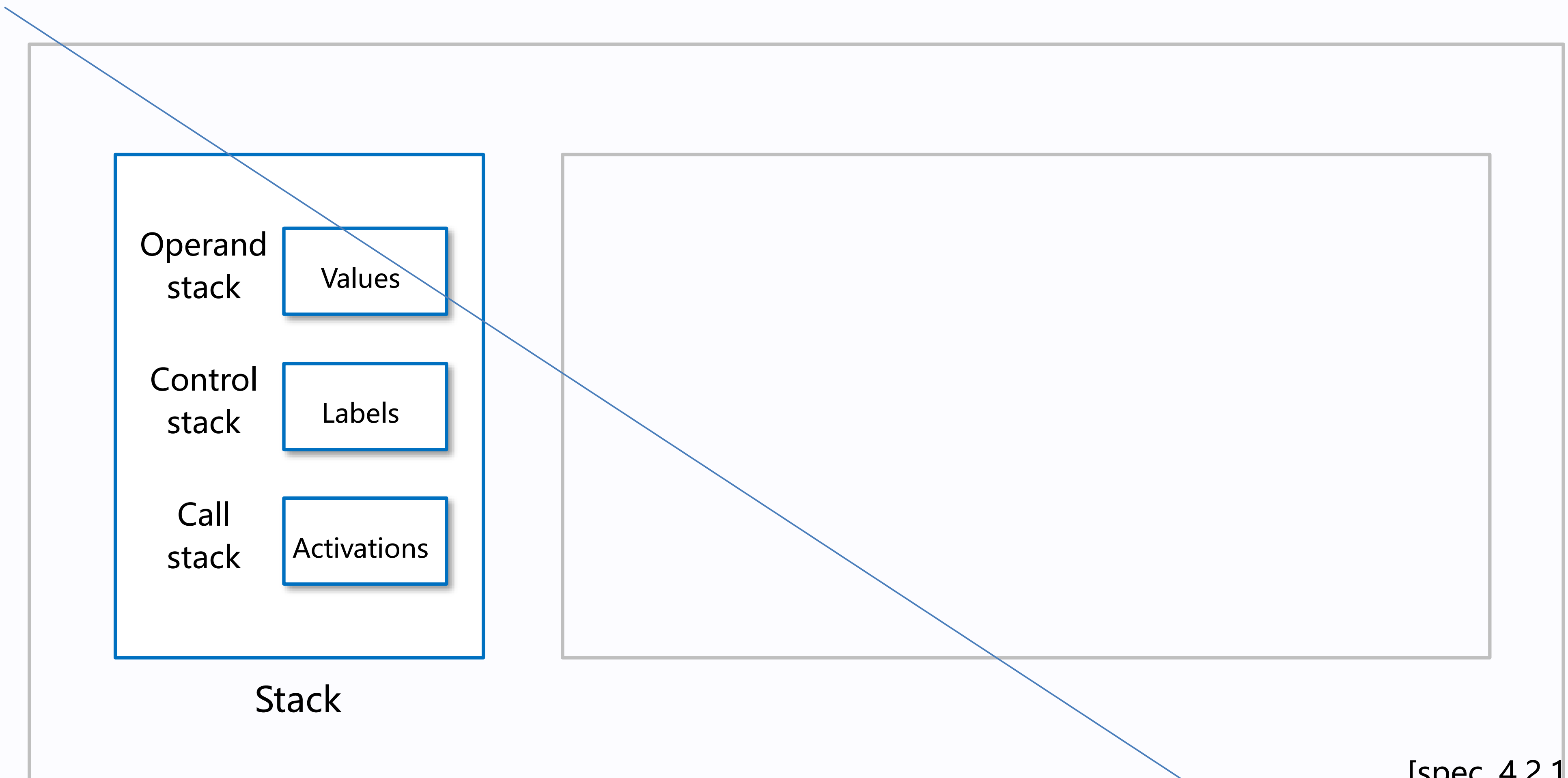
Store



The store represents all global state that can be manipulated byWebAssembly programs.
It consists of the runtime representation of all instances of functions, tables, memories, and globals that have been allocated during the life time of the abstract machine.

References : [1] Ch.1.2, Ch1.4

Stack



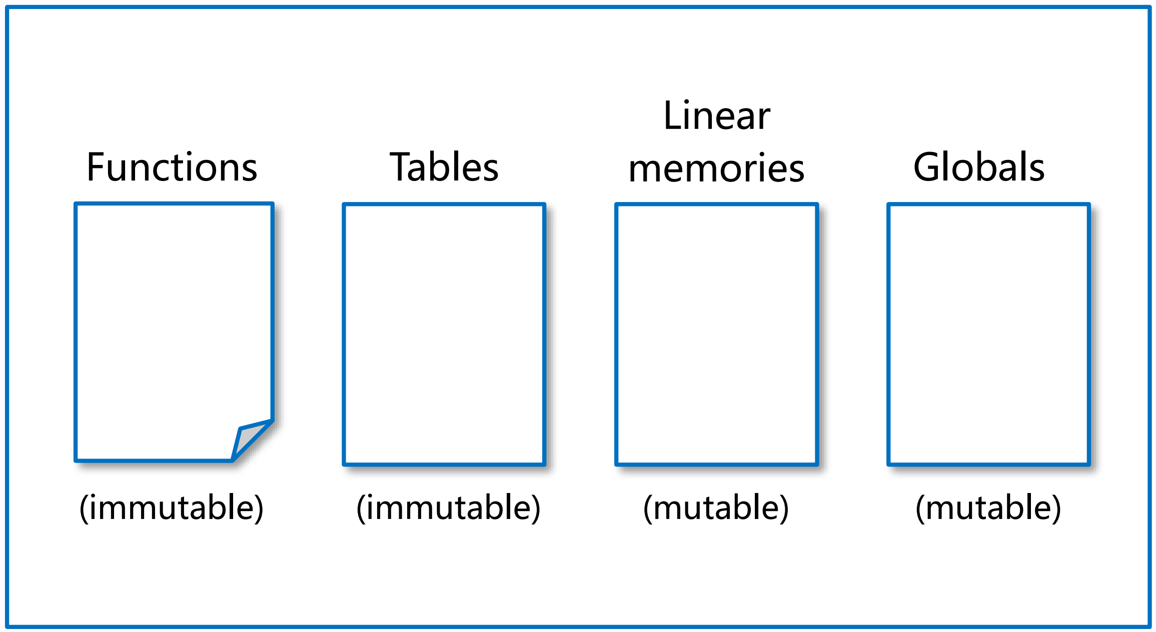
[spec, 4.2.12]

Besides the store, most instructions interact with an implicit stack. The stack contains three kinds of entries:

- Values: the operands of instructions.
- Labels: active structured control instructions that can be targeted by branches.
- Activations: the call frames of active function calls.

References : [1] Ch.1.2, Ch1.4

Index spaces



Store

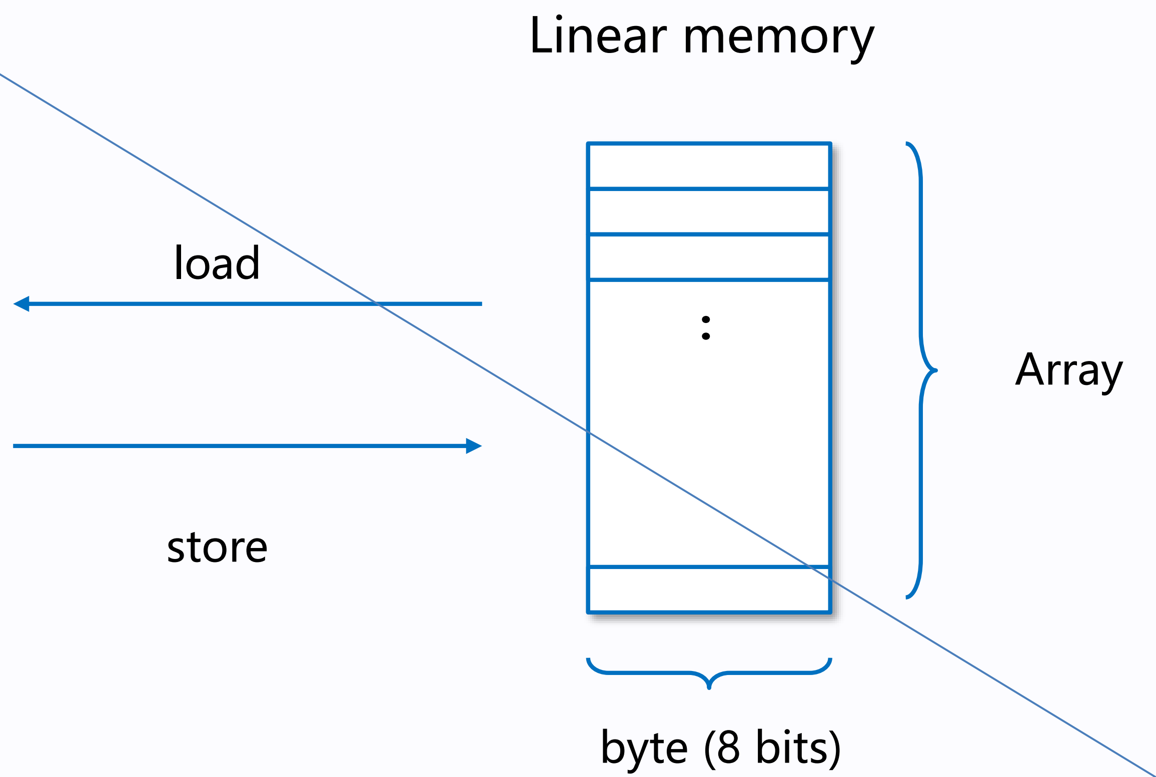
[spec, 2.5.1]

@@@ see tegaki, No.10

References : [1] Ch.1.2, Ch1.4

Linear memory

[spec, 4.2.8]



The len
which i

65536 -
memor

this pag

The byt
data se

provide

[spec, 1.2.1]

A linear memory is a contiguous, mutable array of raw bytes.

A program can load and store values from/to a linear memory at any byte address (including unaligned).
Integer loads and stores can specify a storage size which is smaller than the size of the respective value type.

References : [1] Ch.1.2

Linear memory

[spec, 4.2.8]

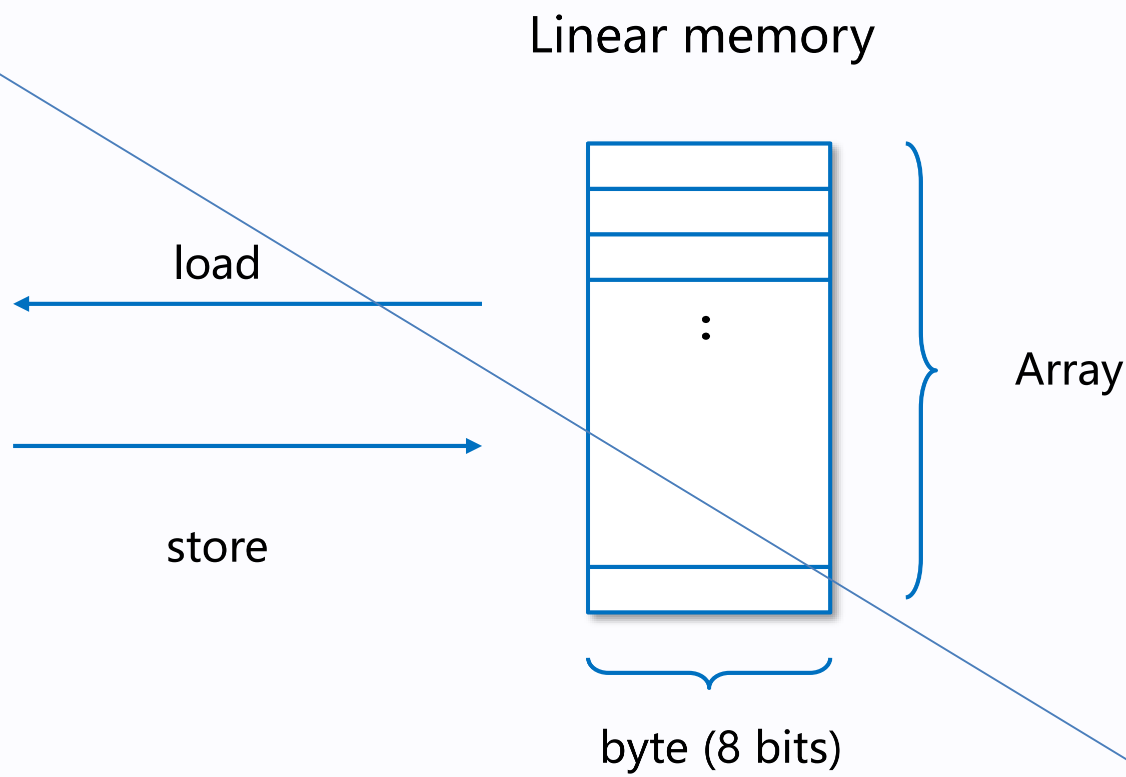
The len
which i

65536 -
memor

this pag

The byt
data se

provide



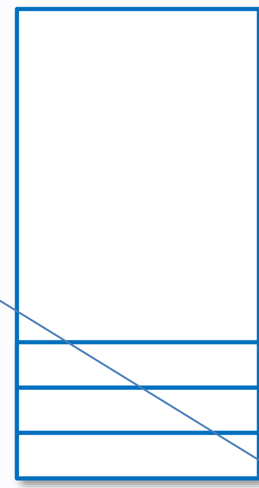
[spec, 1.2.1]

A linear memory is a contiguous, mutable array of raw bytes.

A program can load and store values from/to a linear memory at any byte address (including unaligned).
Integer loads and stores can specify a storage size which is smaller than the size of the respective value type.

References : [1] Ch.1.2

Functions



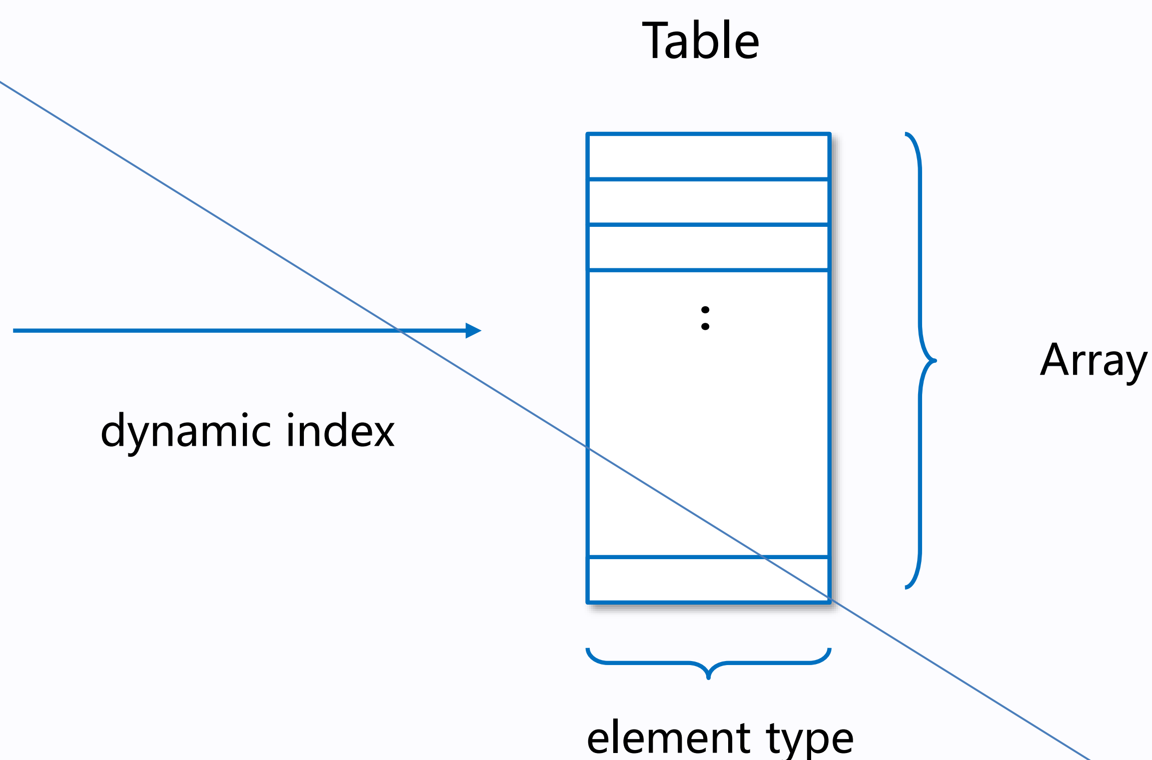
[spec, 1.2.1]

Functions can call each other, including recursively, resulting in an implicit call stack that cannot be accessed directly.

The funcs component of a module defines a vector of functions

References : [1] Ch.1.2

Table



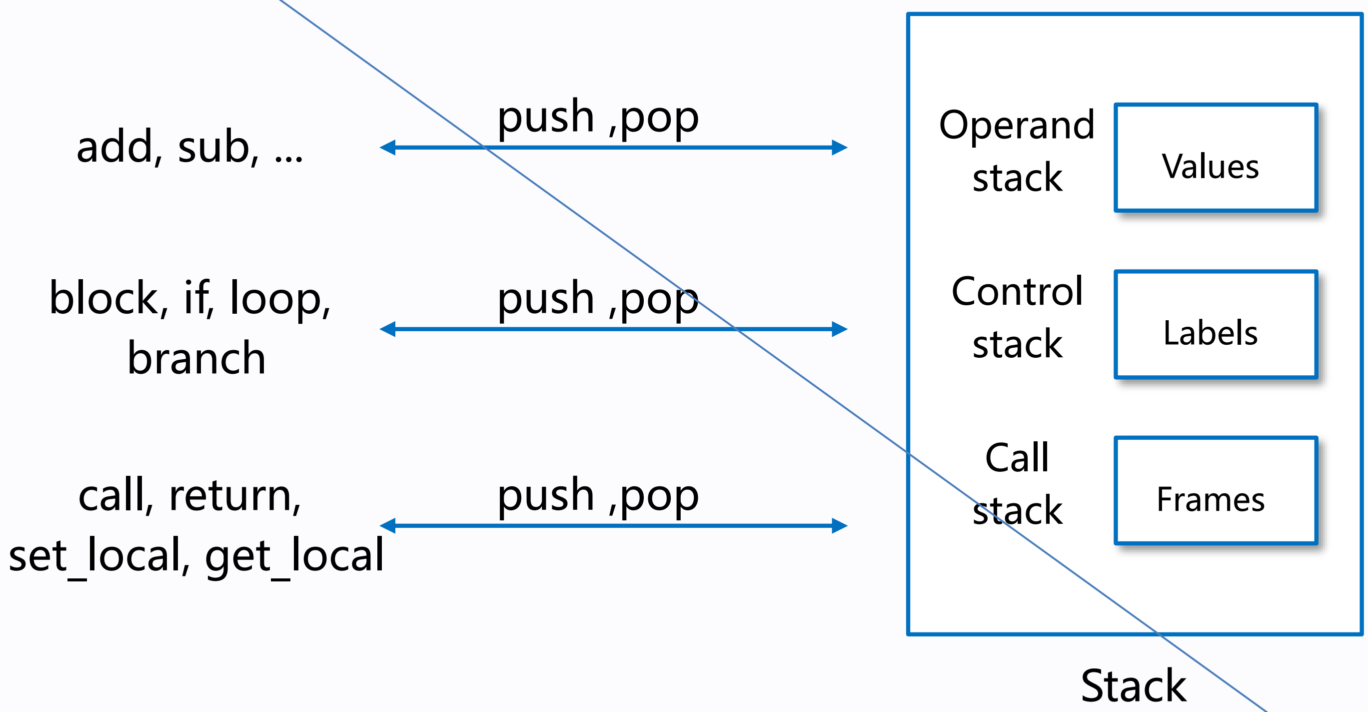
[spec, 1.2.1]

A table is an array of opaque values of a particular element type.

This allows emulating function pointers by way of table indices.

Stack

[spec, 4.2.12]



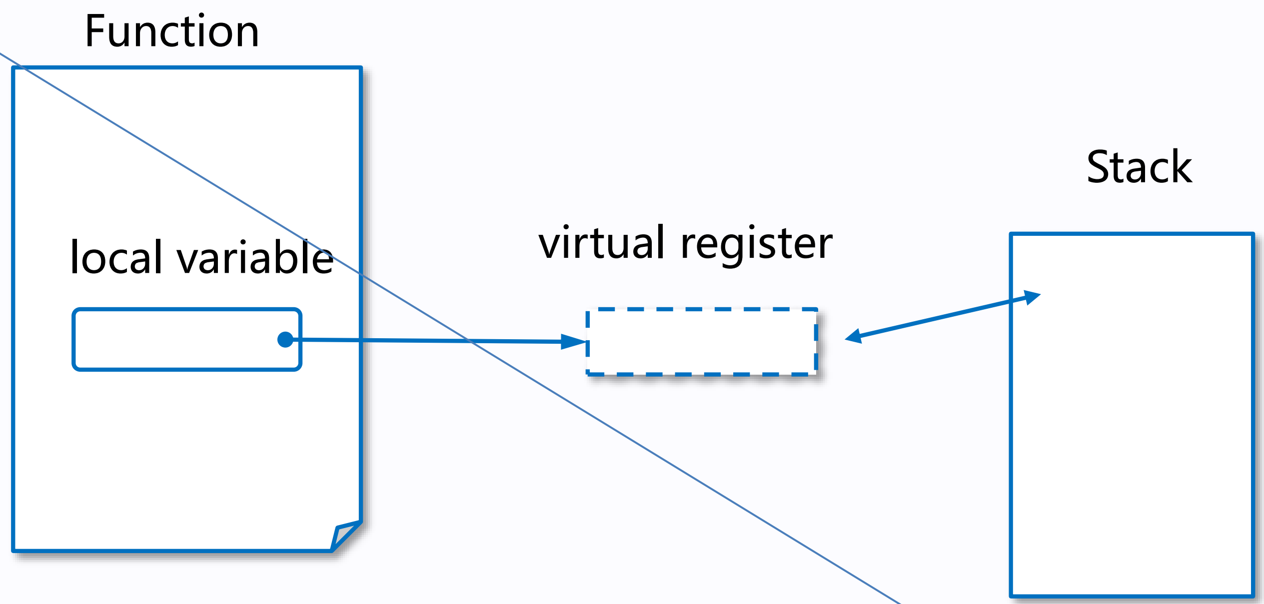
structured stack
[rational, why]

[spec, 4.2.12]

Note: It is possible to model the WebAssembly semantics using separate stacks for operands, control constructs, and calls. However, because the stacks are interdependent, additional book keeping about associated stack heights would be required. For the purpose of this specification, an interleaved representation is simpler.

References : [1] Ch.1.2, Ch1.4, [4]

Functions

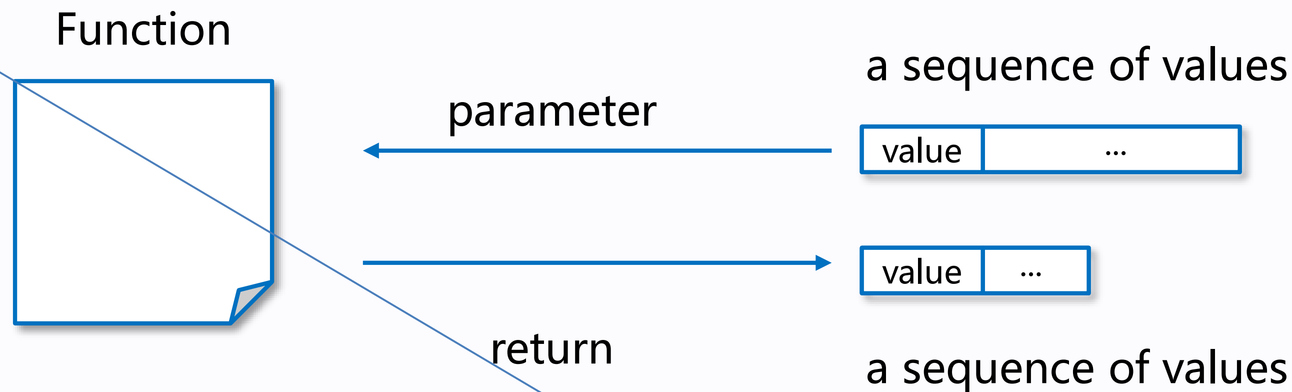


[spec, 1.2.1]

Functions may also declare mutable local variables that are usable as virtual registers.

References : [1] Ch.1.2

Functions



[spec, 1.2.1]

Code is organized into separate functions. Each function takes a sequence of values as parameters and
returns a sequence of values as results.

References : [1] Ch.1.2

WebAssembly module instance

Module, instantiate, invoke

Decode, Validate, Execute

References : [1] Ch.1.1,

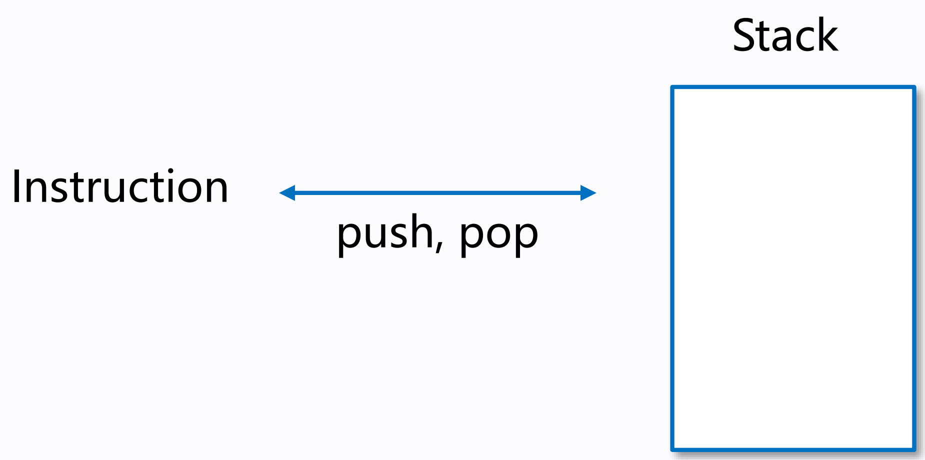
WebAssembly abstract machine

Module and abstract machine

References : [1] Ch.1.1,

Computational model

The computational model of WebAssembly is based on a stack machine.
Code ha meirei no sequence. inorder.
Anmokuno operand stack manipulate.
Stack, anonymous register



[rational, why]

[spec, 1.2.1]

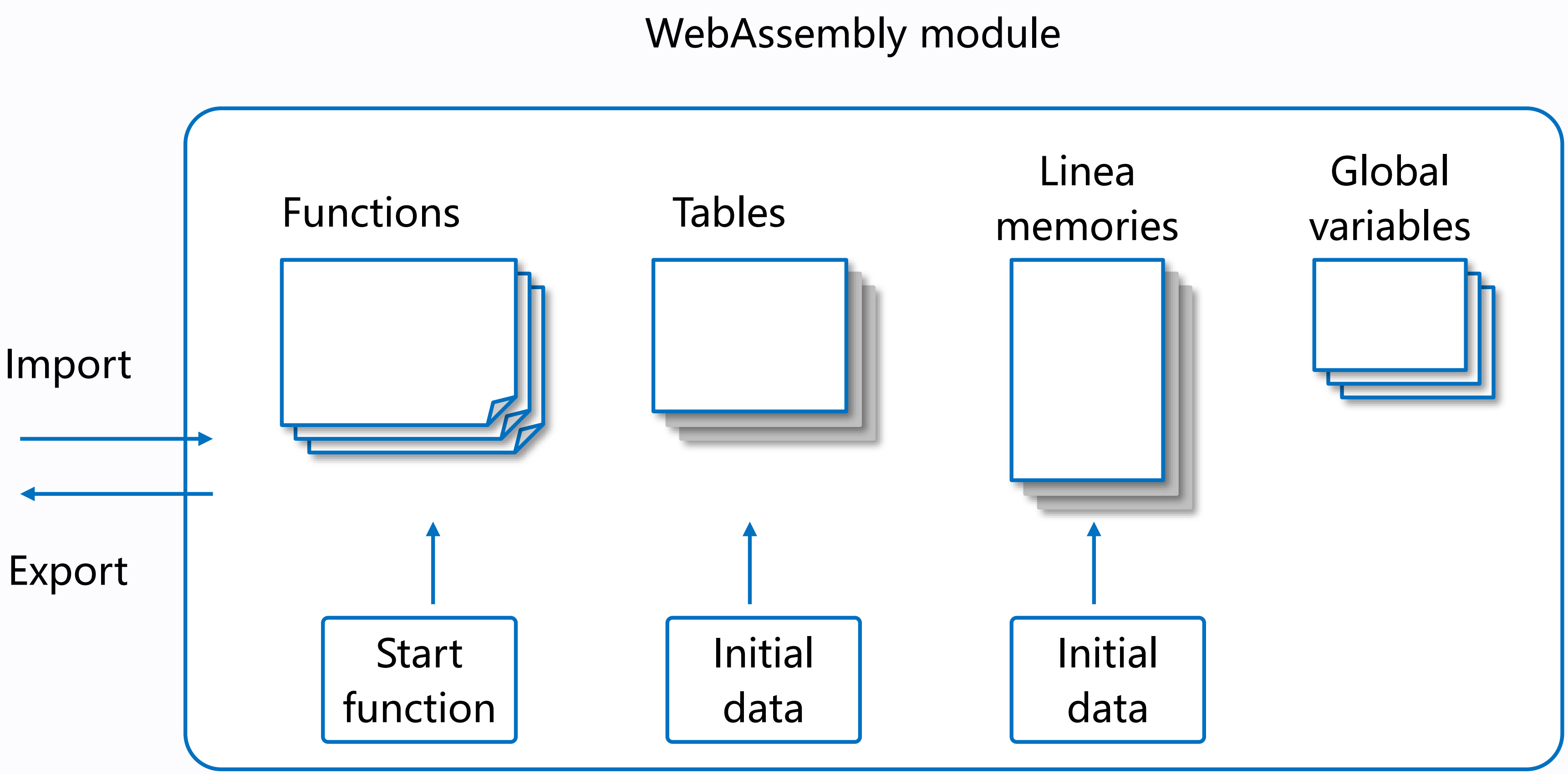
The computational model of WebAssembly is based on a stack machine.

[spec, 2.4]

WebAssembly code consists of sequences of instructions.
Its computational model is based on a stack machine in that instructions manipulate values on an implicit operand stack, consuming (popping) argument values and producing or returning (pushing) result values.

References : [1] Ch.1.2

Modules



References : [1] Ch.1.2

Modules

[spec, 2.5]

@@@ see tegaki, No.10

WebAssembly programs are organized into modules, which are the unit of deployment, loading, and compilation.
A module collects definitions for types, functions, tables, memories, and globals.
In addition, it can declare imports and exports and provide initialization logic in the form of data and element segments or a start function.

References : [1] Ch.1.2

Modules

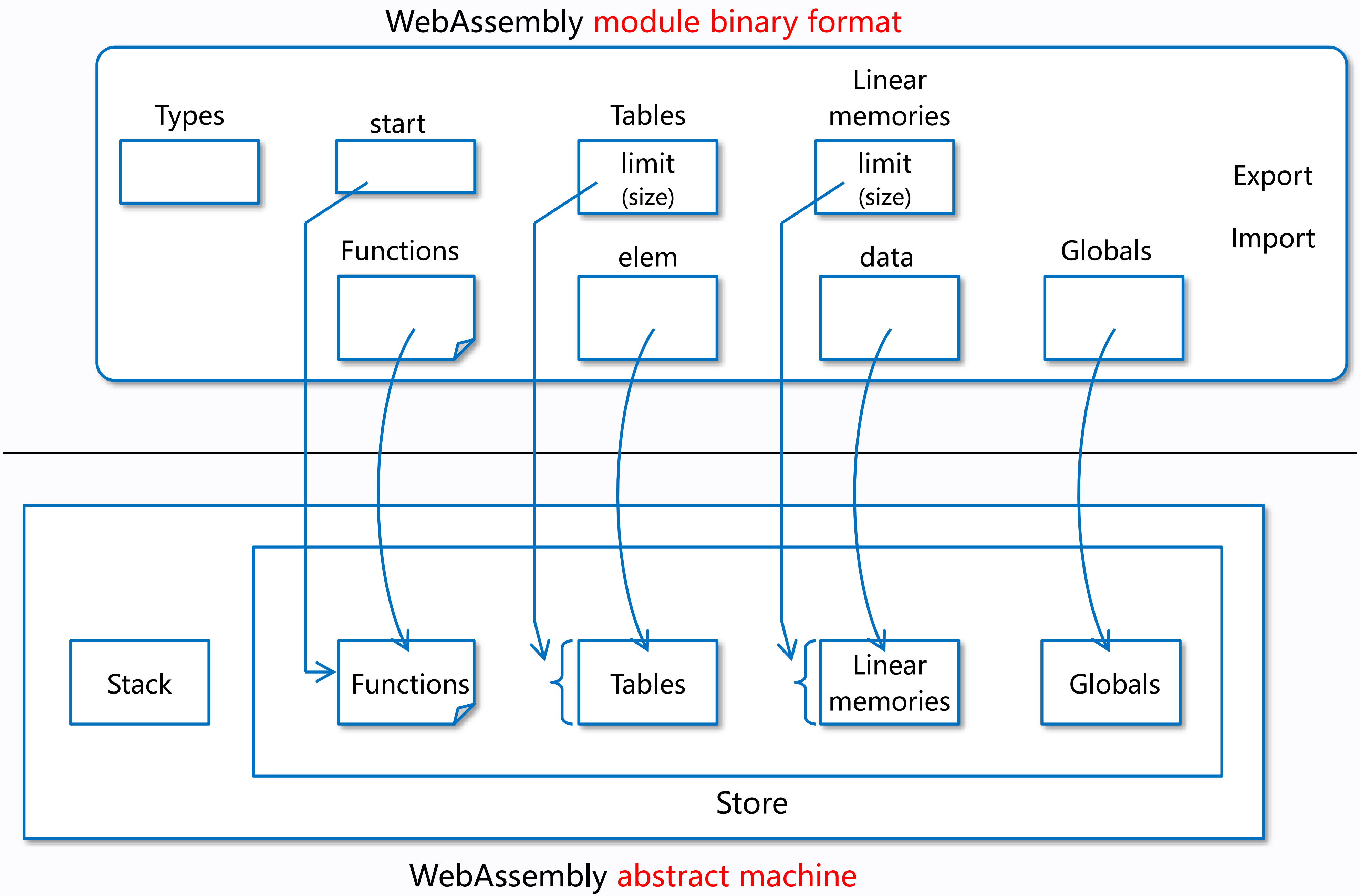
[spec, 2.5]

@@@ see tegaki, No.10

WebAssembly programs are organized into modules, which are the unit of deployment, loading, and compilation.
A module collects definitions for types, functions, tables, memories, and globals.
In addition, it can declare imports and exports and provide initialization logic in the form of data and element segments or a start function.

References : [1] Ch.1.2

Instantiation from module binary to abstract machine



References : [1] Ch.1.2, Ch1.4

Instantiation and invocation

[spec, 1.2.2]

Instantiation

module -> instance -> (State (stoe) + Stack)

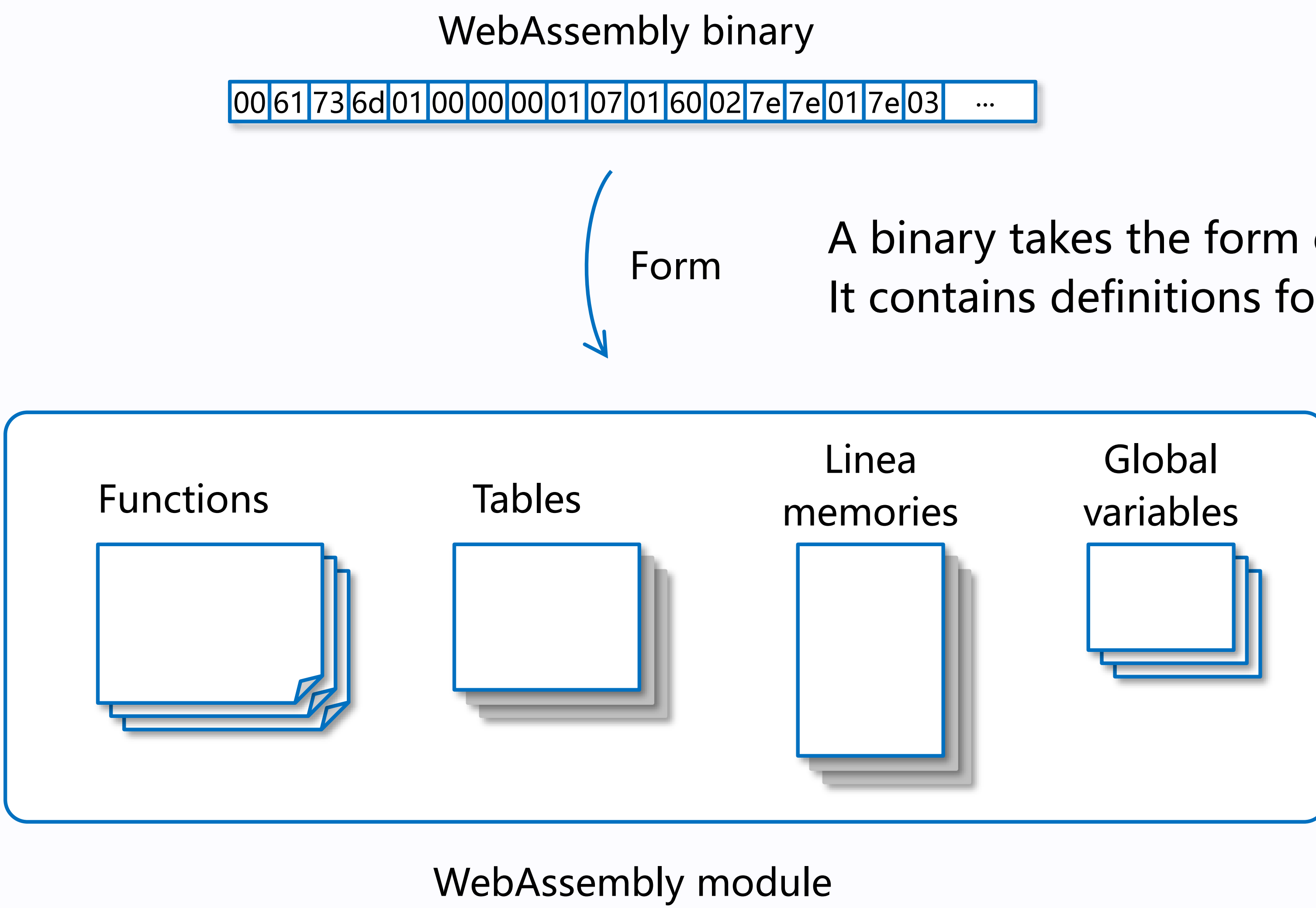
Invocation

?

References : [1] Ch.1.2

Modules

WebAssembly modules are distributed in a binary format. [spec, 1.2.2]



[paper, 2.1]

A binary takes the form of a module.
It contains definitions for functions, gl

[spec, 1.2.1]

A WebAssembly binary takes the form of a module that contains definitions for functions, tables, and linear memories, as well as mutable or immutable global variables.

References : [1] Ch.1.2

Modules

[spec, 4.2.5]

@@@ see tegaki, 1026-2

A module instance is the runtime representation of a module.
It is created by instantiating a module, and collects runtime representations of all entities that are imported, defined, or exported by the module.

References : [1] Ch.1.2

Instantiation and invocation

[spec, 1.2.2]

Instantiation. A module instance is the dynamic representation of a module, complete with its own state and execution stack. Instantiation executes the module body itself, given definitions for all its imports. It initializes globals, memories and tables and invokes the module’s start function if defined. It returns the instances of the module’s exports.

Invocation. Once instantiated, furtherWebAssembly computations can be initiated by invoking an exported function on a module instance. Given the required arguments, that executes the respective function and returns its results.

Instantiation and invocation are operations within the embedding environment.

References : [1] Ch.1.2

Instructions

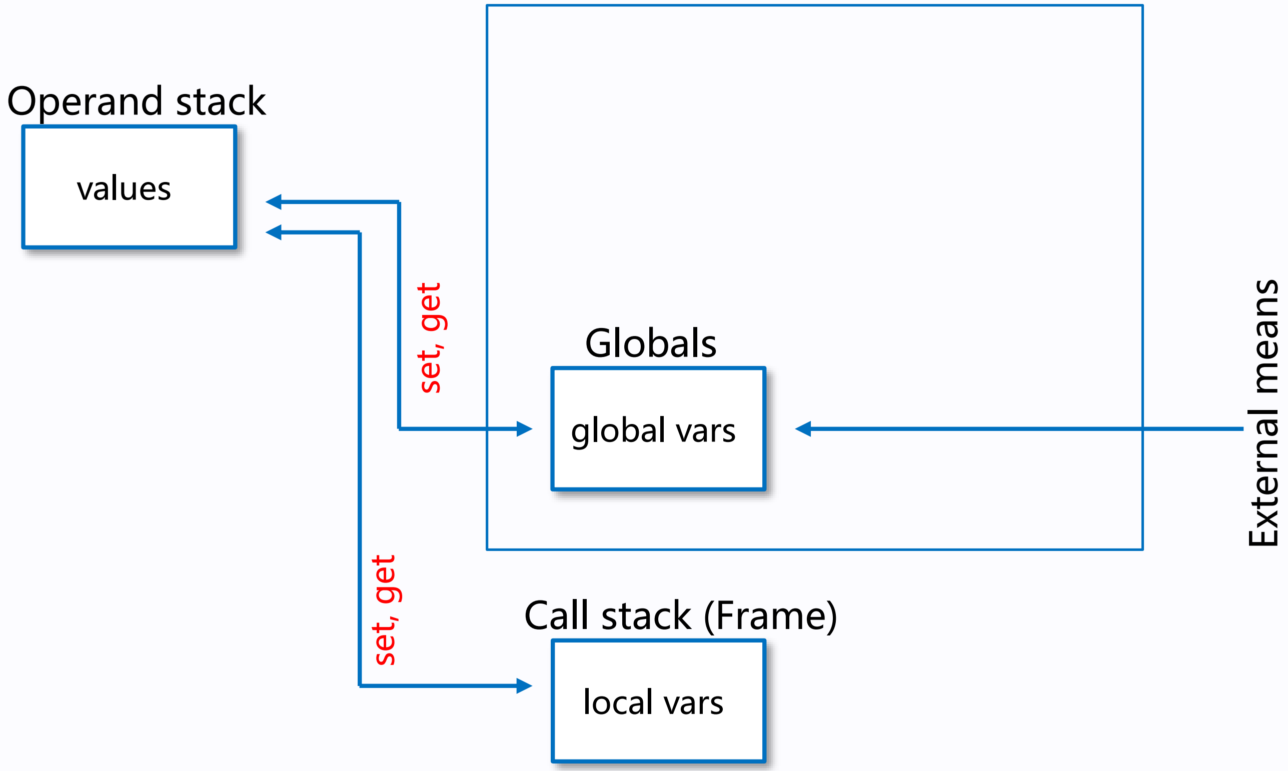
[spec, 2.4]

@@@ see tegaki, No.7

WebAssembly code consists of sequences of instructions. Its computational model is based on a stack machine in that instructions manipulate values on an implicit operand stack, consuming (popping) argument values and producing or returning (pushing) result values.

References : [1] Ch.1.2

Variables



[spec, 2.4.3]

[spec, 2.5.3] local

[spec, 2.5.6] global

References : [1] Ch.1.2, Ch1.4

Validation

Compile-phase static validation
Loading-phase static validation
Runtime-phase dynamic validation

verifiable code format

speed and simplicity of validation is key to good performance
simple validation semantics

[paper. 8]

References : [1] Ch.1.1,

Memory safety

Boundary check
Pages unit allocation and virtual memory protection (MMU)

Split Inst and data

References : [1] Ch.1.1,

Validated control flow

Structured control flow; restricted branch target. (break and continue)
(Simple, so fast validation)

Index call only

Type check

References : [1] Ch.1.1,

(Type system)

Typed assembler
Soundness with semantics

References : [1] Ch.1.1,

Control instructions

[spec, 2.4.5]

@@@ see tegaki, No.8-9

Control flow is structured, meaning it is expressed with well-nested constructs such as blocks, loops, and conditionals.
Branches can only target such constructs.

[rational, why]

structured control flow allows simpler and more efficient verification

[rational, control flow]

Structured control flow provides simple and size-efficient binary encoding and compilation.

[spec, 2.4.5]

> Intuitively, a branch targeting a block or if behaves like a break statement, while a branch targeting a loop behaves like a continue statement.

Validation

Validation

[spec, 1.2.2]

A decoded module has to be valid.
Validation checks a number of well-formedness conditions to guarantee that the module is meaningful and safe.
In particular, it performs type checking of functions and the instruction sequences in their bodies, ensuring for example that the operand stack is used consistently.

[spec, 3.1]

Validation checks that a WebAssembly module is well-formed. Only valid modules can be instantiated.

References : [1] Ch.1.2

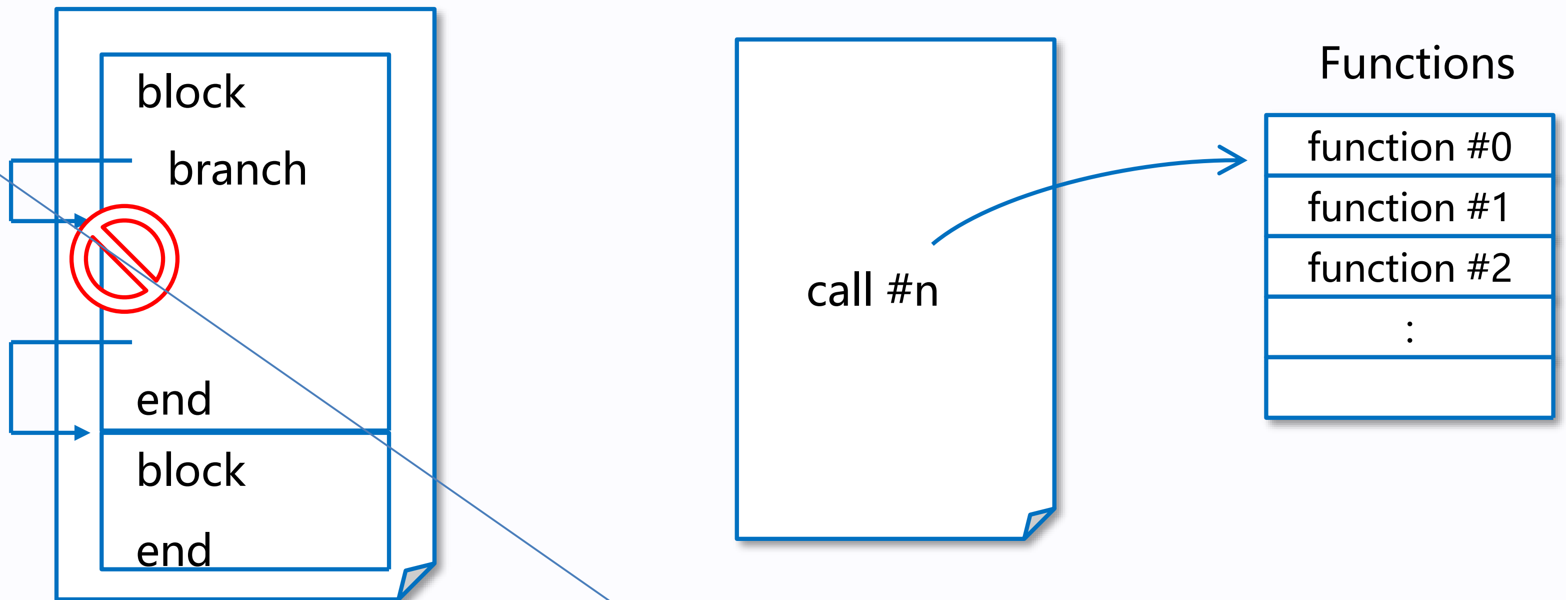
Types

[spec, 2.3]

Various entities in WebAssembly are classified by types.
Types are checked during validation, instantiation, and possibly execution.

References : [1] Ch.2.3

Structured control flow

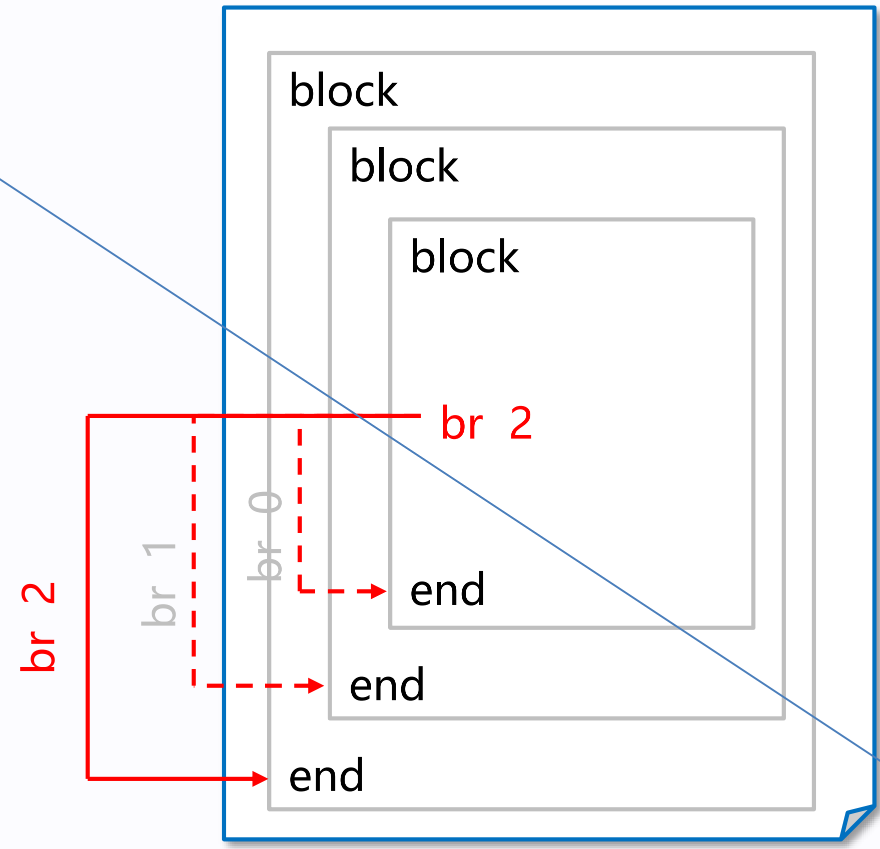


Branch target is restricted to control block.
Call target is restricted to indices.

Structured control flow; restricted branch target. (break and continue)
(Simple, so fast validation)

References : [2], [1] Ch.3,

Nest and branch instructions



Each structured control instruction introduces an implicit label. Labels are targets for branch instructions that reference them with label indices. Unlike with other index spaces, indexing of labels is relative by nesting depth,

that is, label 0 refers to the innermost structured control instruction enclosing the referring branch instruction, while increasing indices refer to those farther out.

References : [1] Ch.2, Ch.4, [2]

call_indirect and dynamic link

KEEP

Indirect Calls Function pointers can be emulated with the call indirect instruction which takes a runtime index into a global table of functions defined by the module. The functions in this table are not required to have the same type. Instead, the type of the function is checked dynamically against an expected type supplied to the call indirect instruction. The dynamic signature check protects integrity of the execution environment; a successful signature check ensures that a single machine-level indirect jump to the compiled code of the target function is safe. In case of a type mismatch or an out of bounds table access, a trap occurs. The heterogeneous nature of the table is based on experience with asm.js’ s multiple homogeneous tables; it allows more faithful representation of function pointers and simplifies dynamic linking. To aid dynamic linking scenarios further, exported tables can be grown and mutated dynamically through external APIs.

References : [1] Ch.2, Ch.4, [2]

import and safe foreign call

KEEP

External and Foreign Calls Functions can be imported to a module and are specified by name and signature. Both direct and indirect calls can invoke an imported function, and through export/import, multiple module instances can communicate.
Additionally, the import mechanism serves as a safe foreign function interface through which a WebAssembly program can communicate with its embedding environment. For

References : [1] Ch.2, Ch.4, [2]

LEB128

KEEP

[spec, 5.2.2]

All integers are encoded using the LEB12828 variable-length integer encoding, in either unsigned or signed variant.

Unsigned integers are encoded in unsigned LEB128 format.

Signed integers are encoded in signed LEB12830 format.

References : [1] Ch.4.1

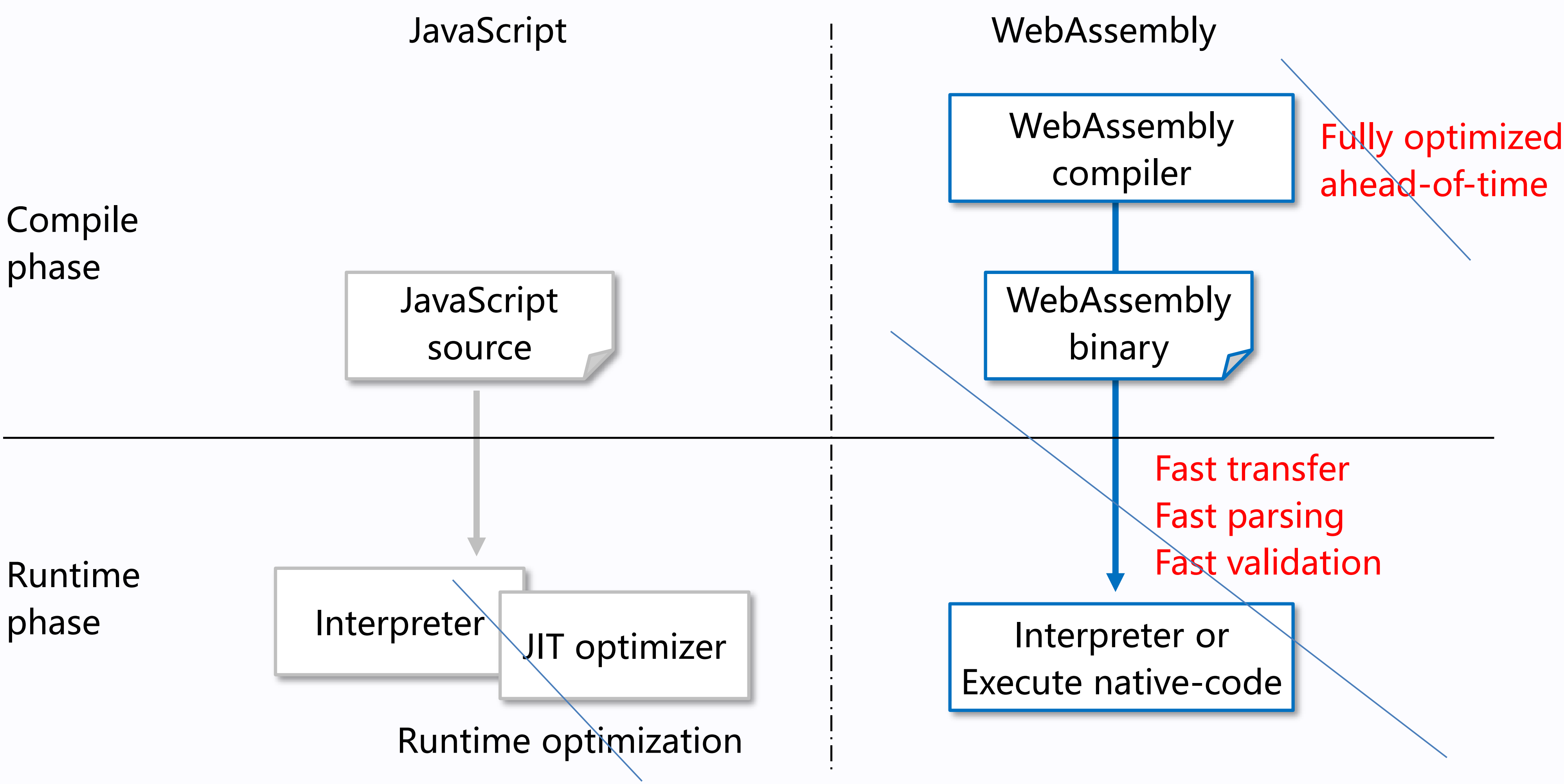
1. Introduction

Portable

1. Introduction

Fast

Low-level code generation with ahead-of-time



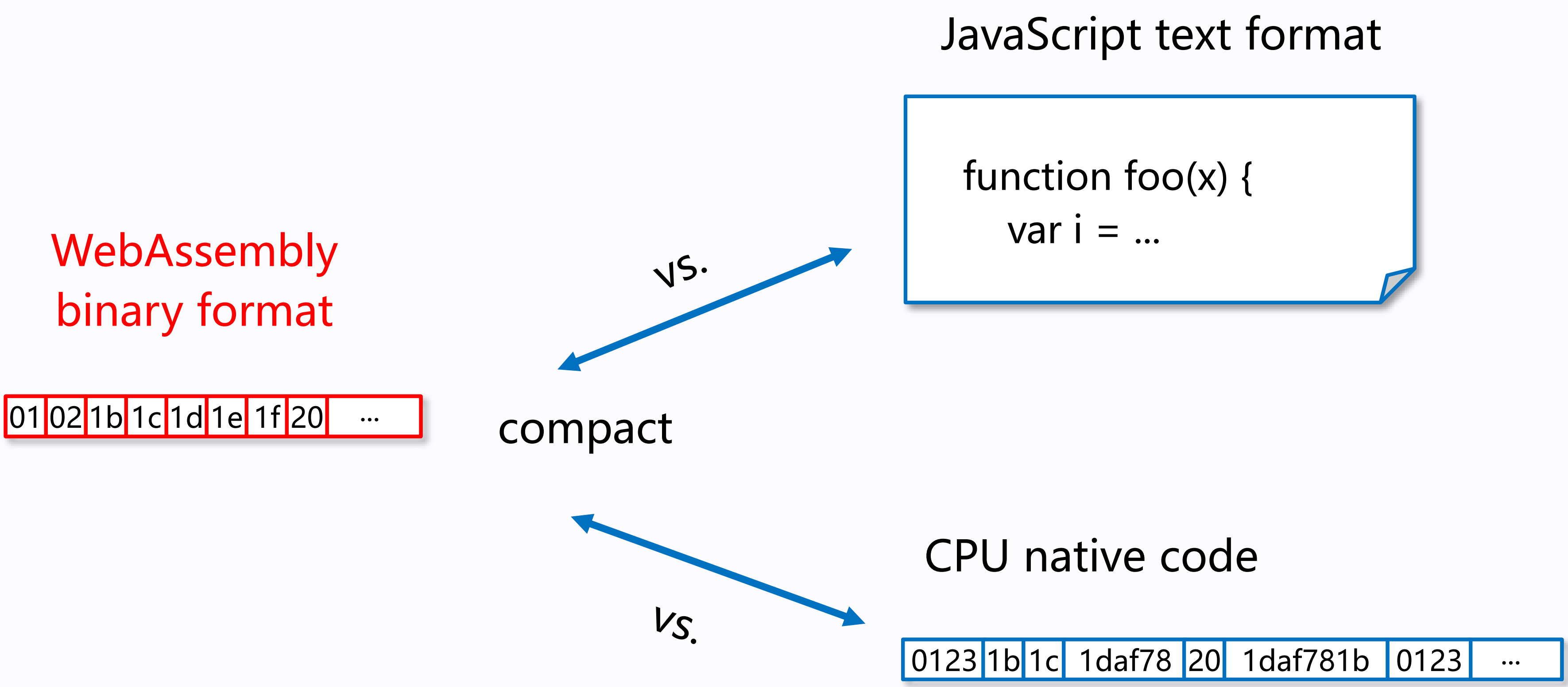
Executes with near native code performance,
taking advantage of capabilities common to all contemporary hardware.

References : [2], [3], [1] Ch.1

1. Introduction

Compact

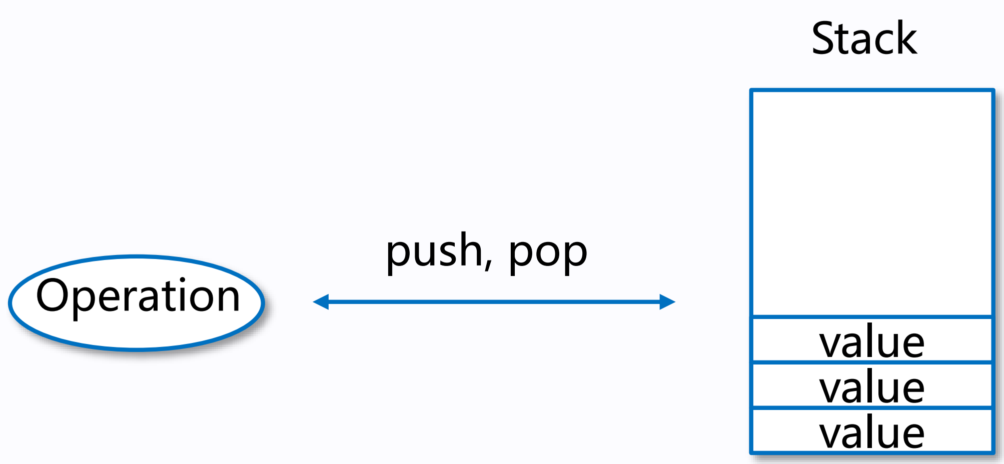
Compact binary format



A binary format is fast to transmit by being smaller than typical text or native code formats.

References : [1] Ch.1.1, [2], [4]

Stack machine



WebAssembly is based on a stack machine.

The stack machine allows smaller binary encoding than registers.

References : [1] Ch.1.1, [2], [4]

Safety

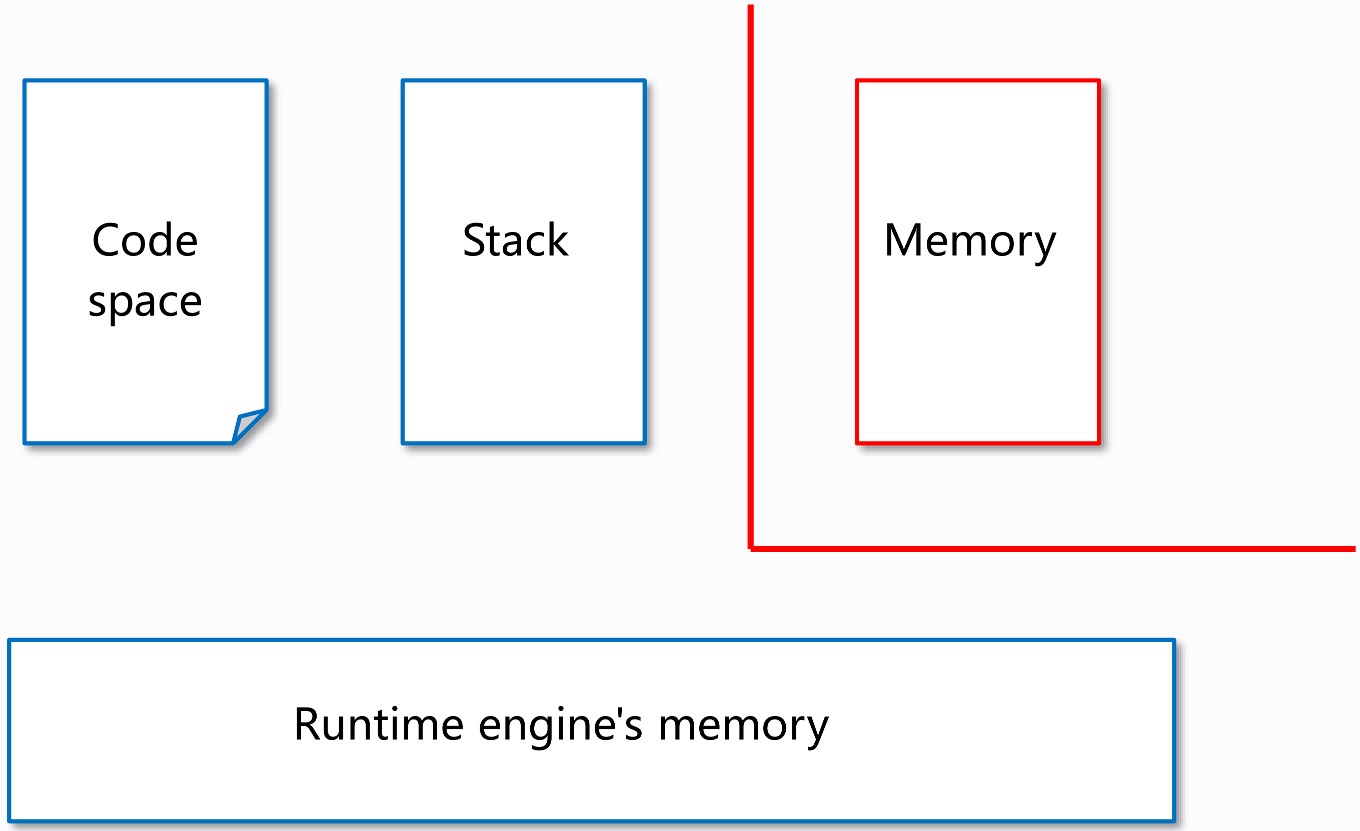
Soundness

Typed programming language

Typing check

References : [2], [1] Ch.1

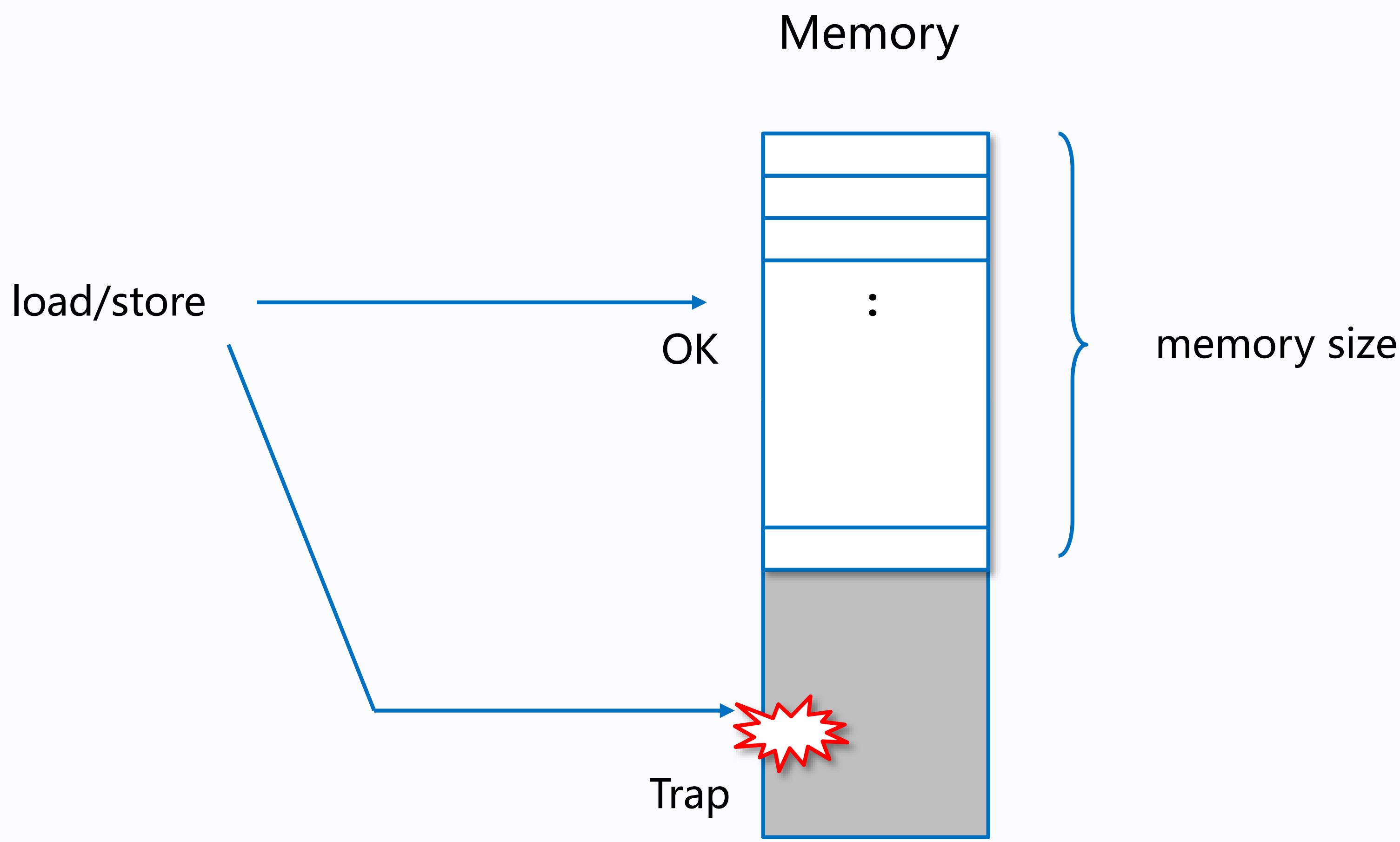
Memory safety



Linear memory is disjoint from code space, the execution stack, and the engine's data structures

References : [2], [1] Ch.4

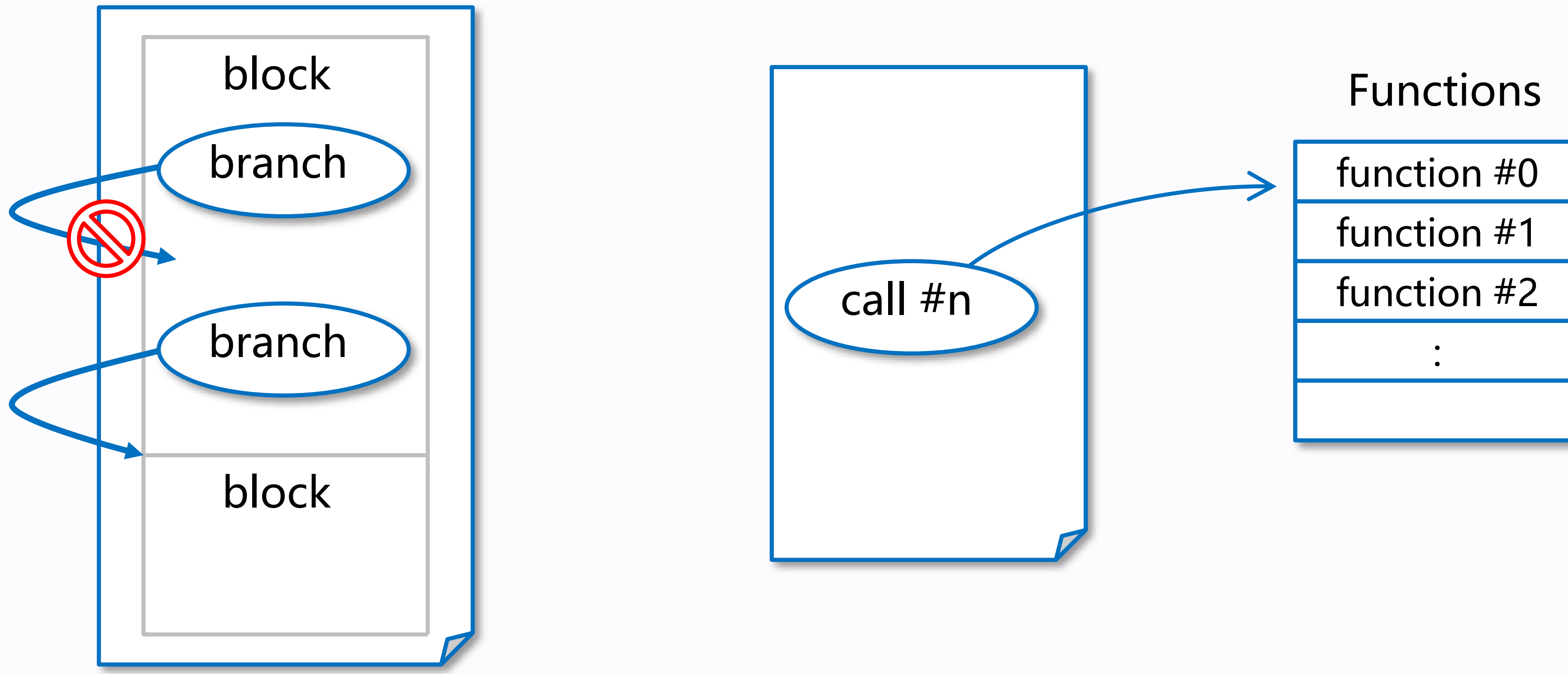
Memory safety



All memory access is dynamically checked against the memory size.

References : [2], [1] Ch.4

Structured control flow



Branches can only target such constructs.

Branch target is restricted to control block.
Call target is restricted to indices.

Structured control flow; restricted branch target. (break and continue)
(Simple, so fast validation)

References : [2], [1] Ch.3,