WEBASSEMBLY illustrated

exploring some mental models and implementations

Takenobu T.

WIP

Rev. 0.01.0

NOTE

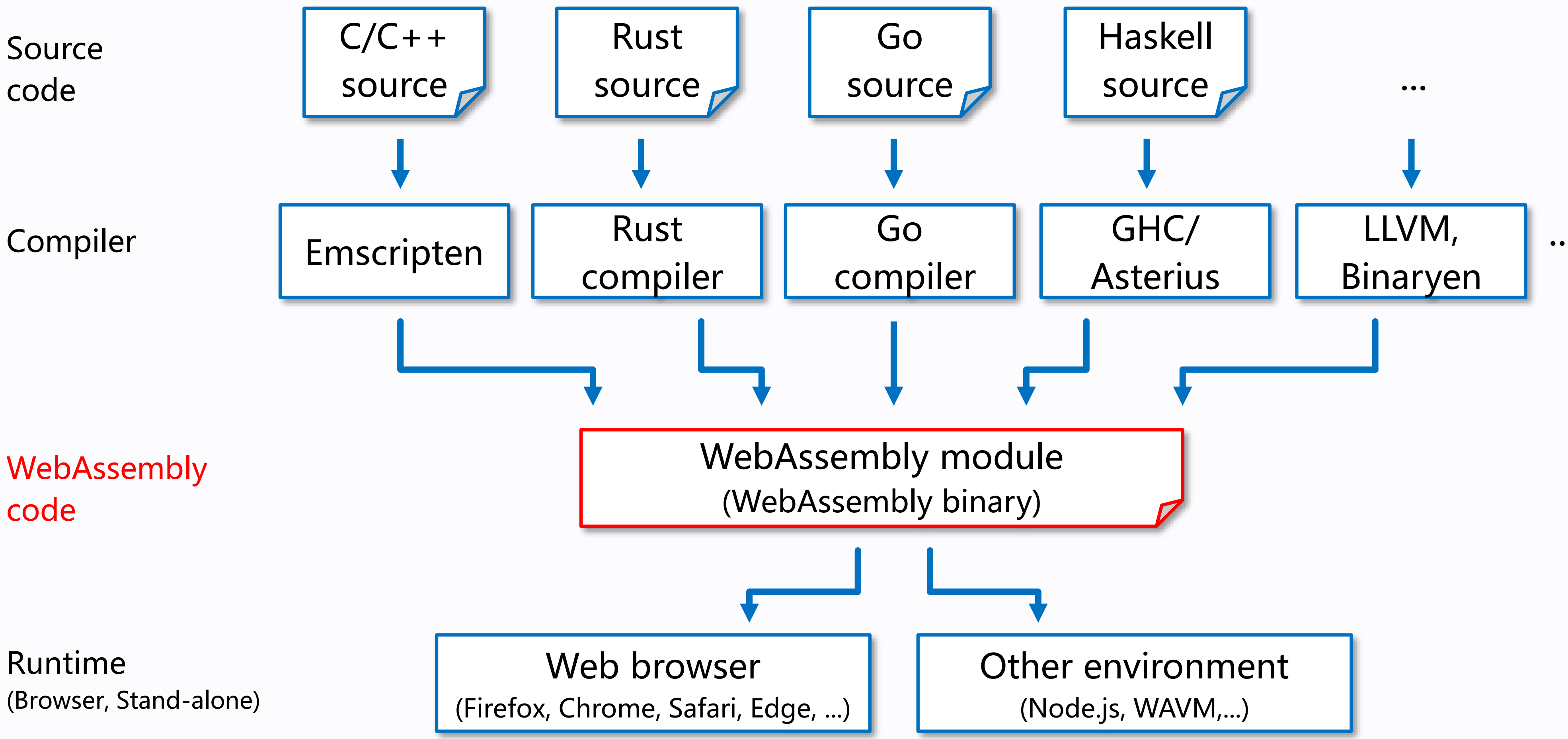
- Please refer to the official documents in detail.
- This information is based on "WebAssembly Specification Release 1.0 (Draft, last updated Oct 31, 2018)".
- This information is current as of Nov, 2018. Still work in progress.

Contents

1. Introduction

Overview

WebAssembly is a code format



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

References : [1] Ch.1.1, [2], [3], [6]

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

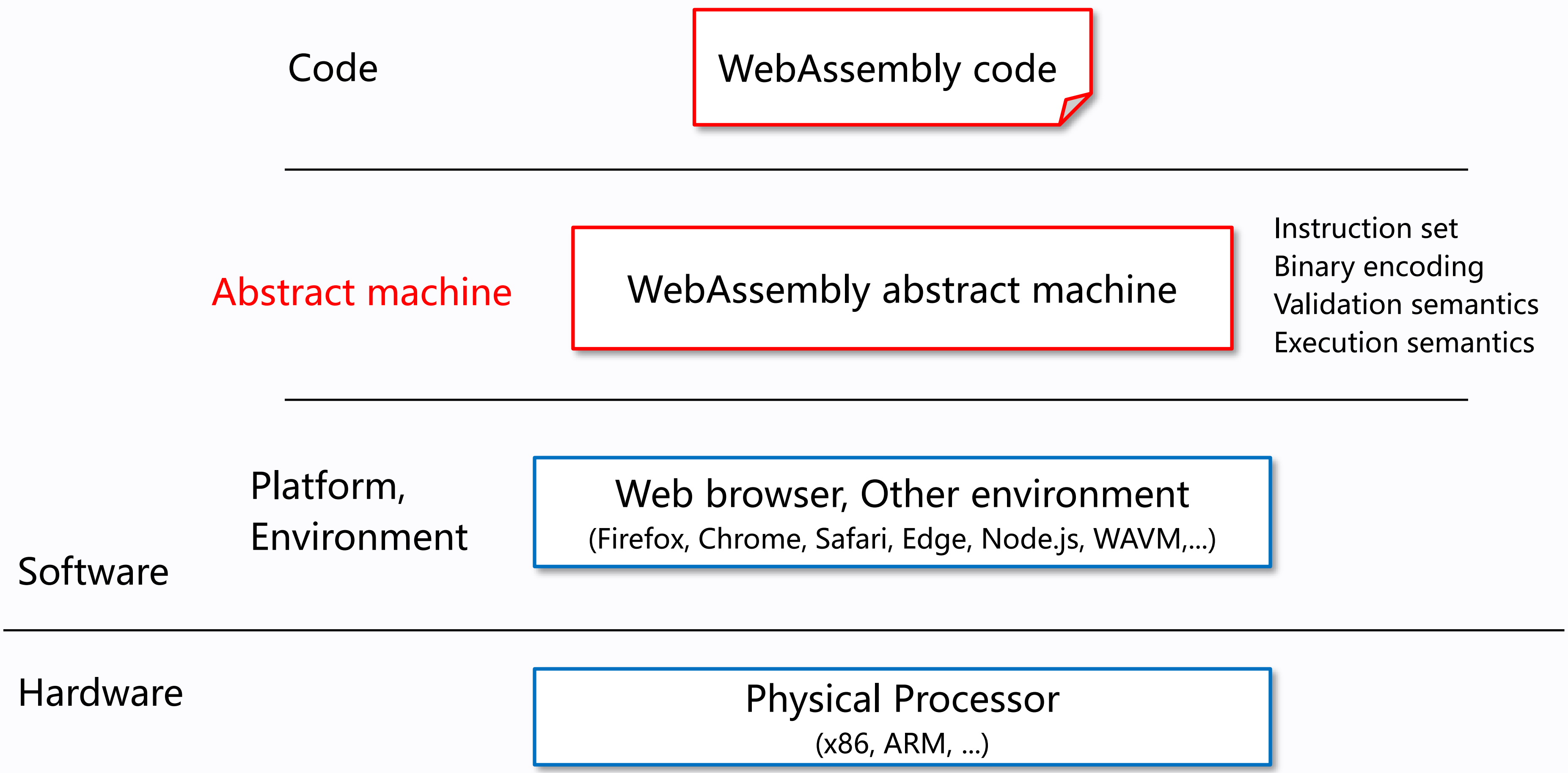
```
0x0061736d010000 ...
```

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, [7]

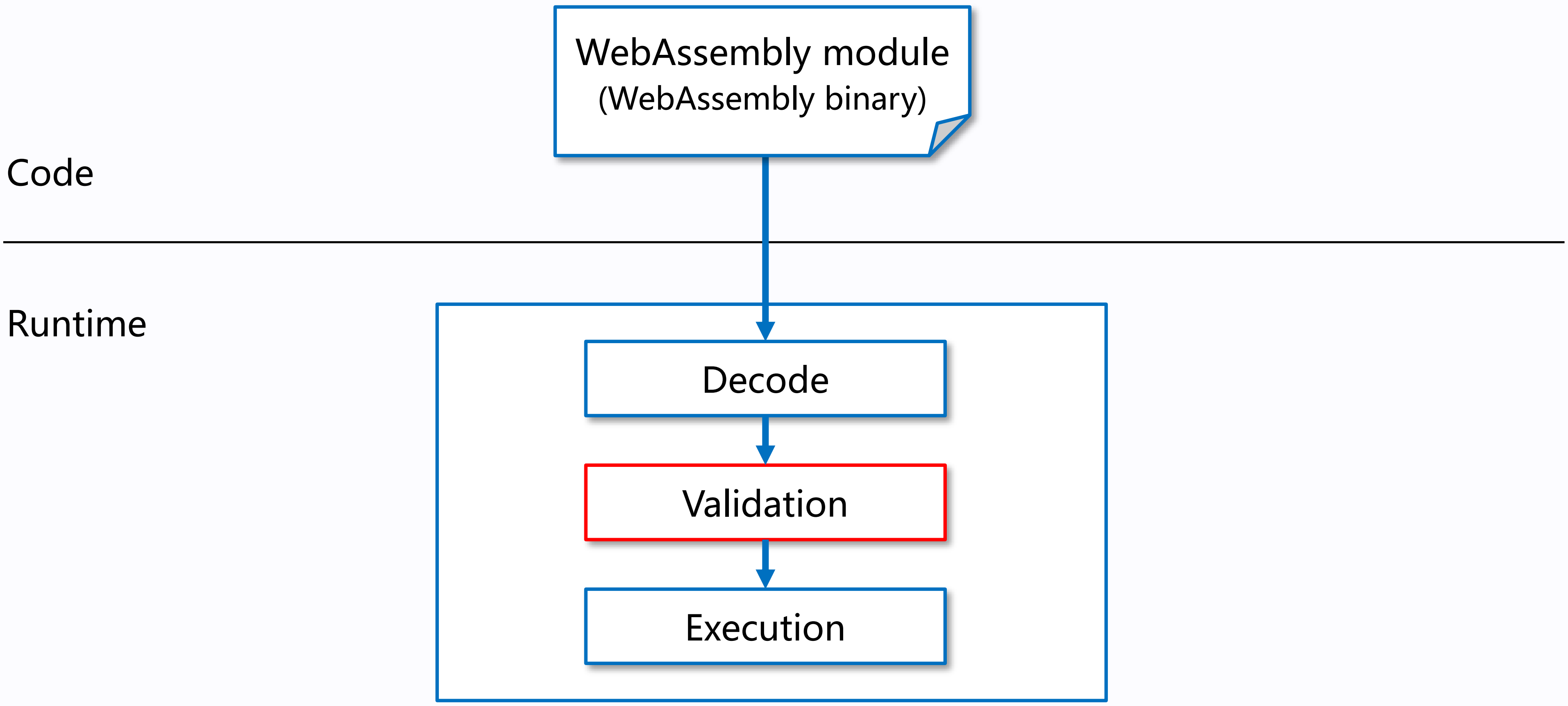
Abstract machine is defined



WebAssembly is a virtual instruction set architecture (virtual ISA).
Execution behavior is defined in terms of an abstract machine.

References : [1] Ch.1.1, [2], [3]

Validation



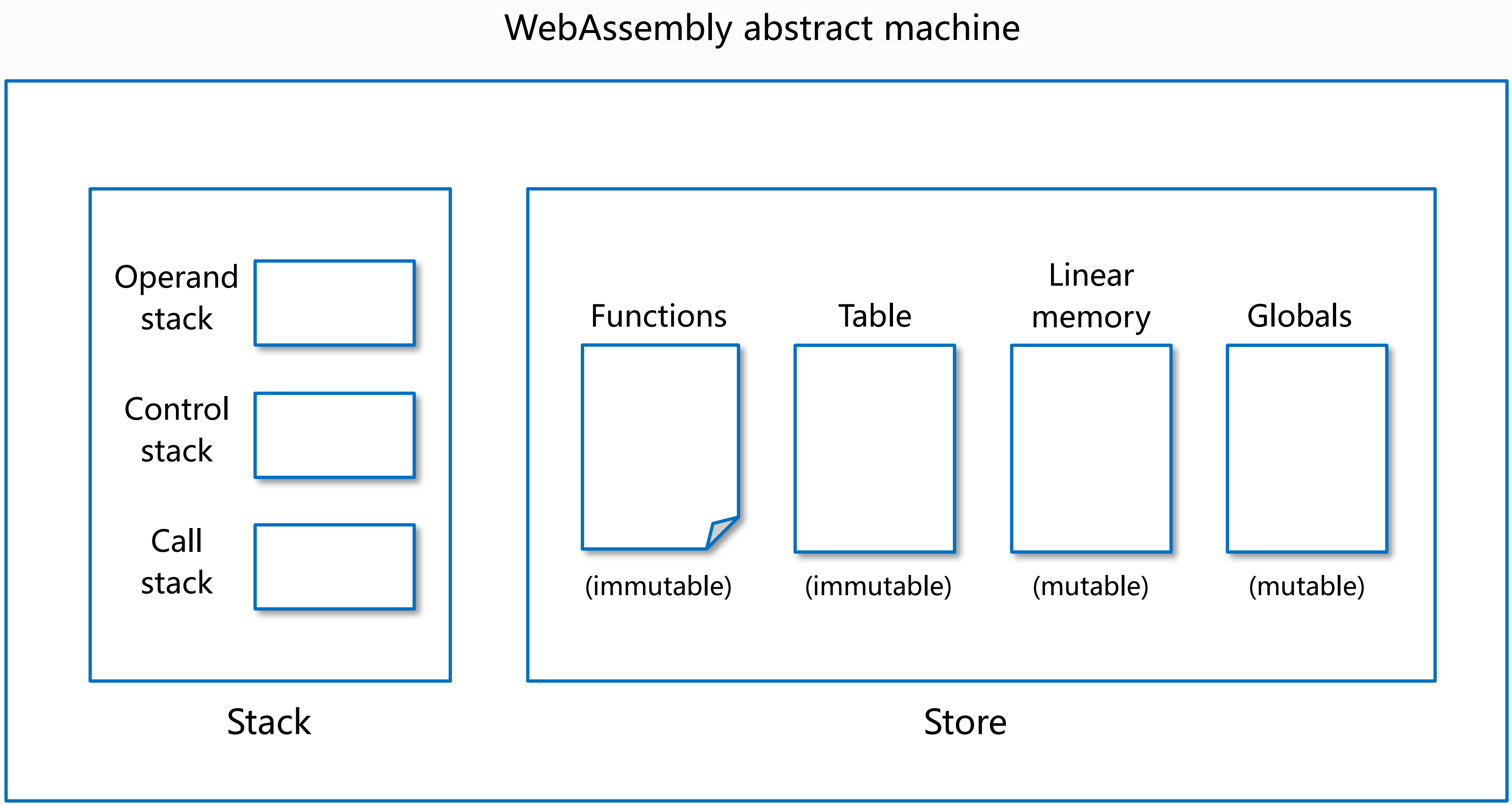
Validation checks that a WebAssembly module is well-formed.
Validity is defined by a type system.
The type system of WebAssembly is sound, implying both type safety and memory safety with respect to the WebAssembly semantics.

References : [1] Ch.1, Ch.3, Ch.7, [2]

2. WebAssembly abstract machine

Abstract machine

WebAssembly abstract machine

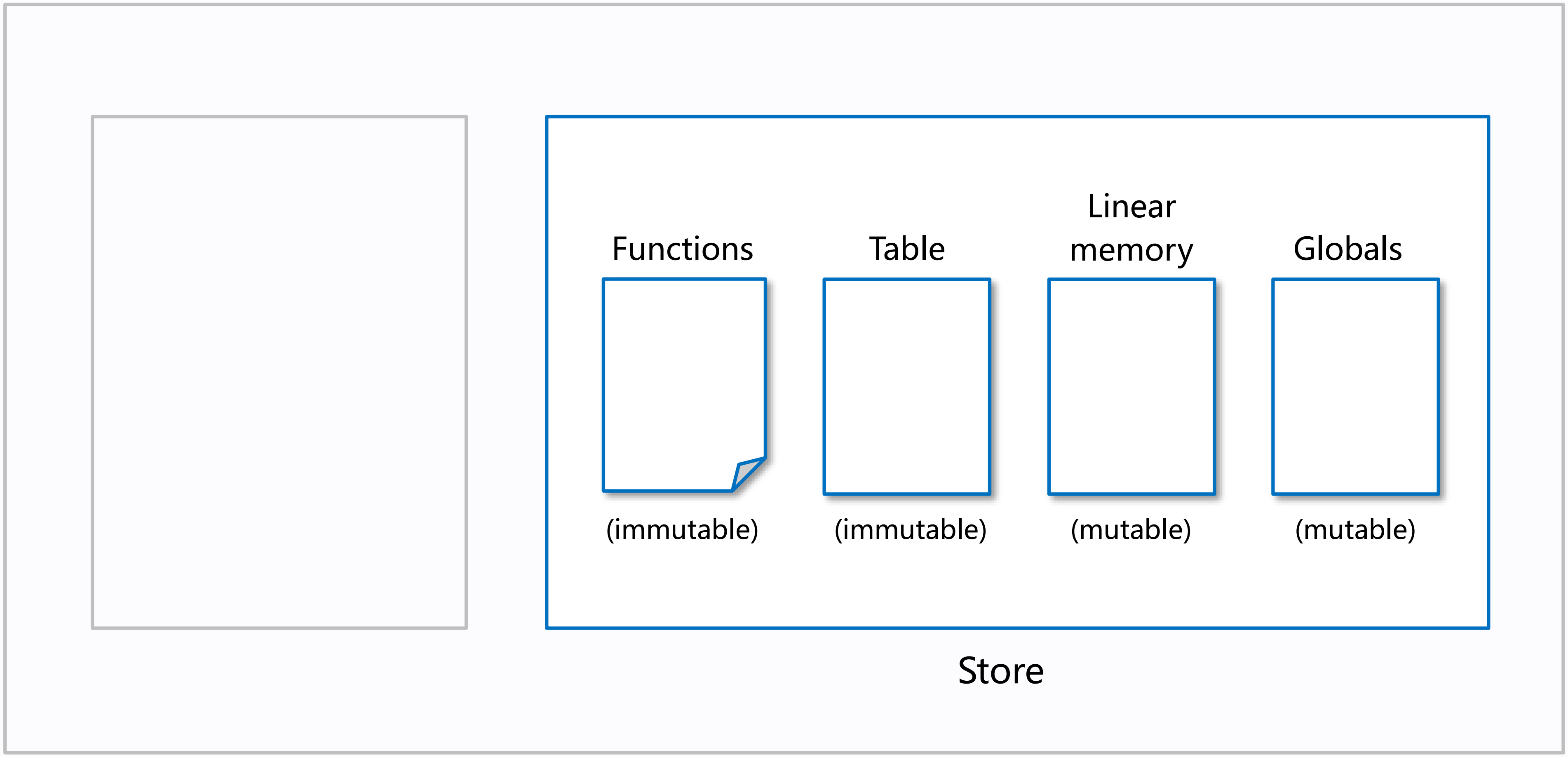


WebAssembly abstract machine is based on a stack machine.
The abstract machine includes a store and an implicit stack.

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

Store

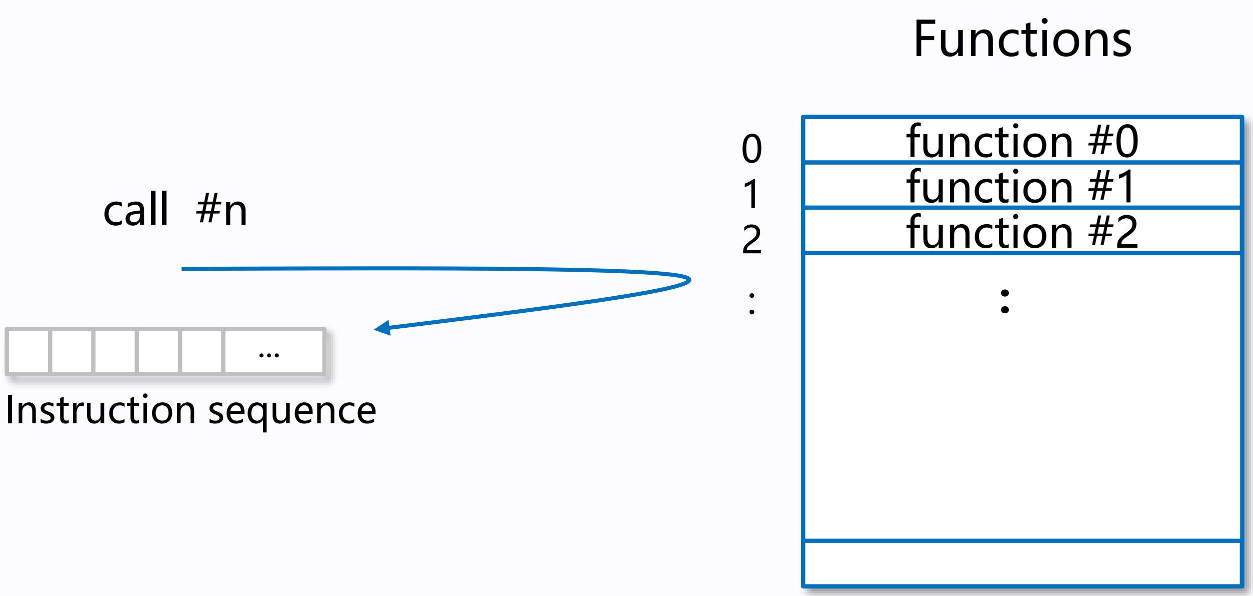
Store



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

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

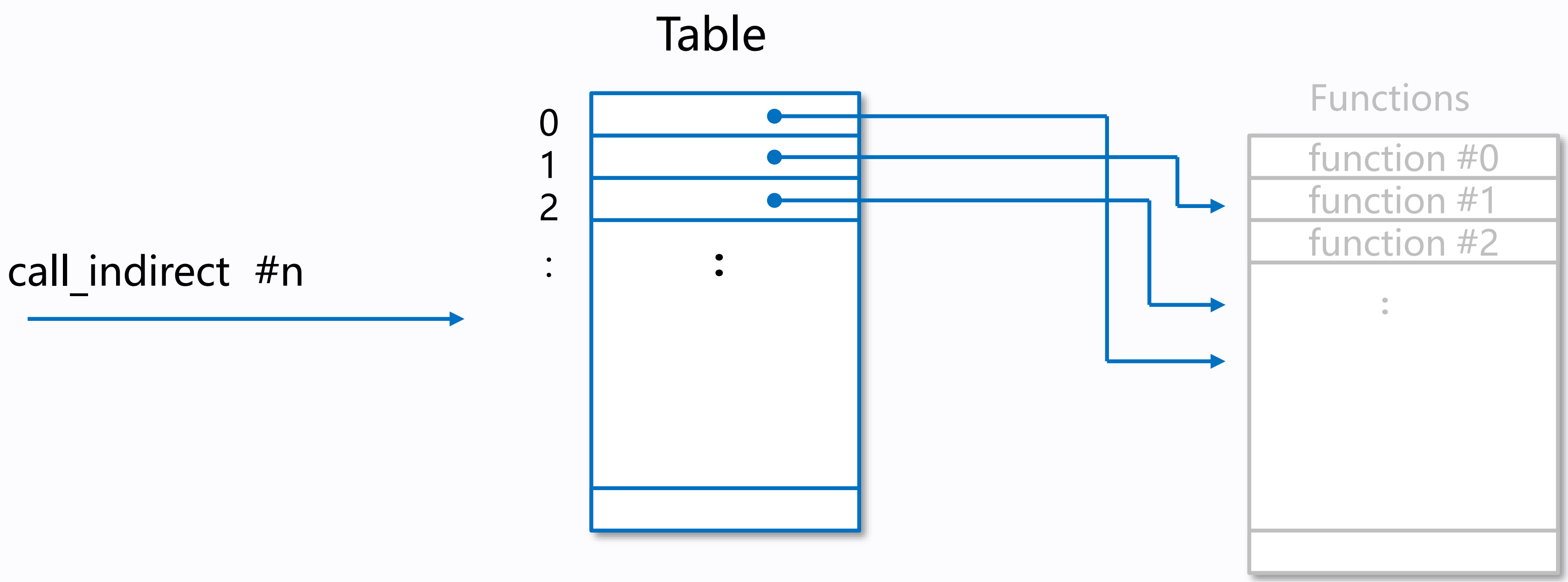
Functions



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

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

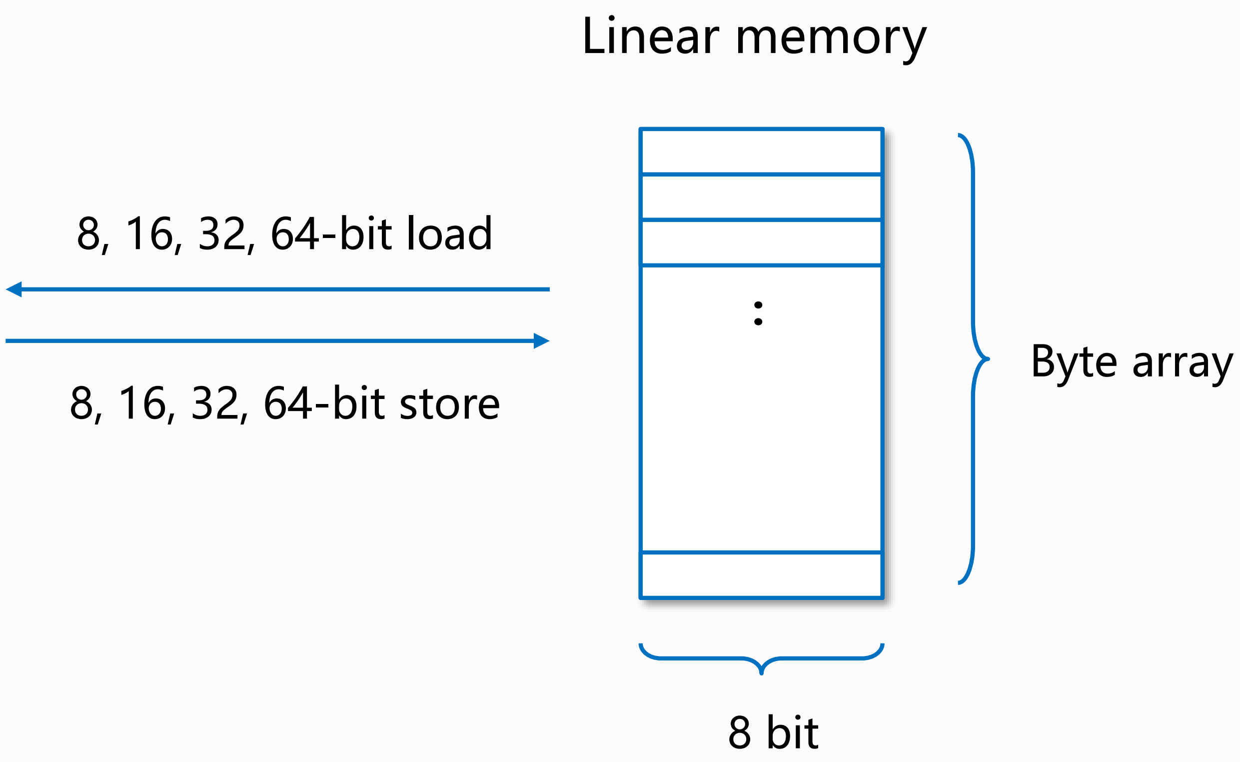
Table



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.2, Ch.4, [2], [4]

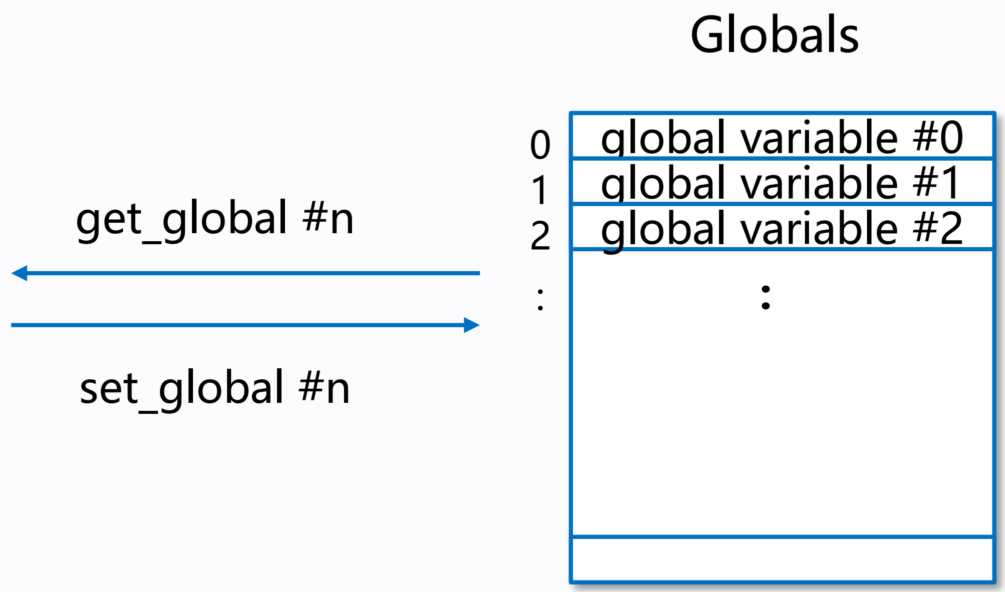
Linear memory



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.2, Ch.4, [2], [4]

Globals



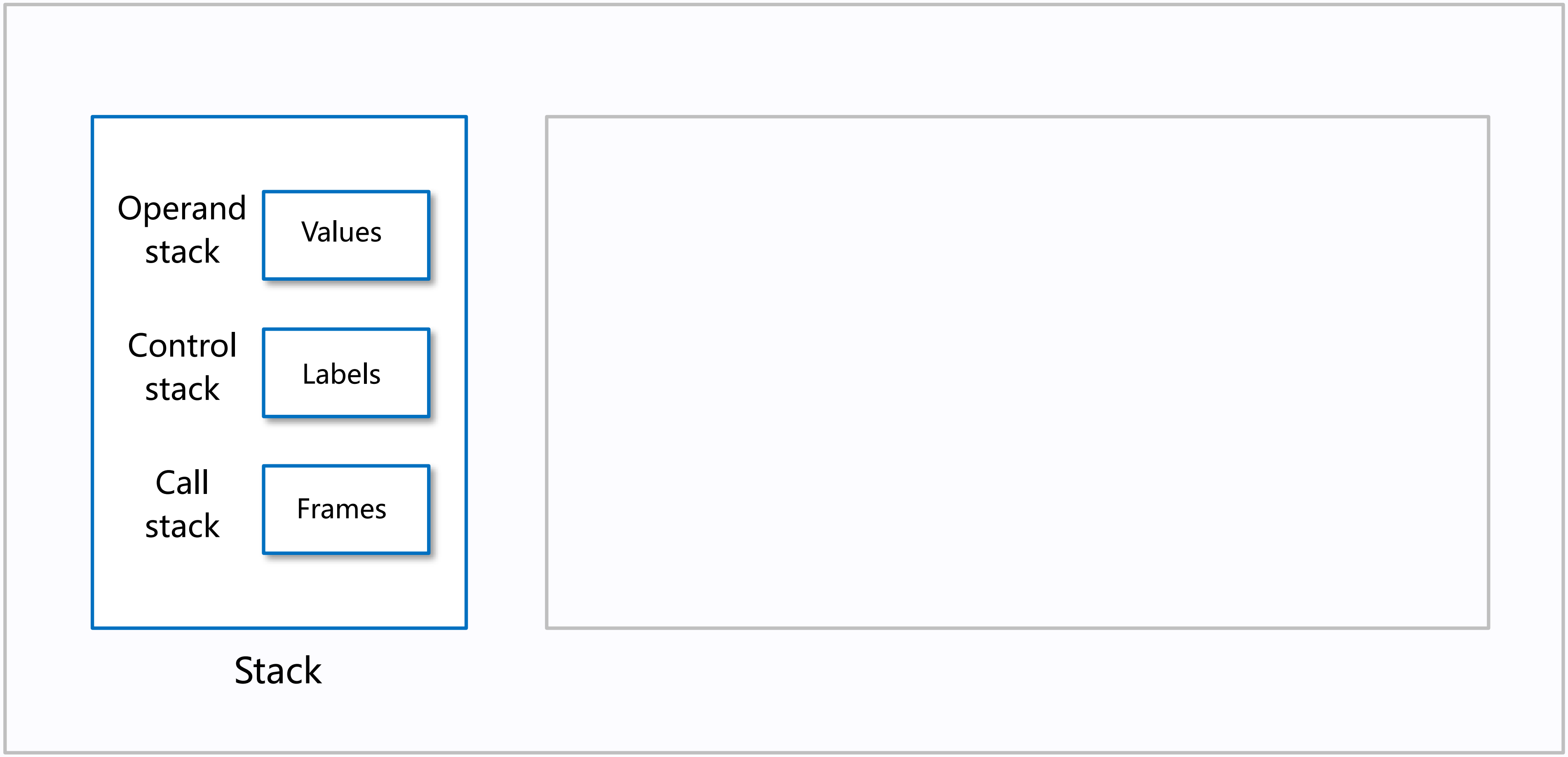
The globals component defines a vector of global variables.
The globals are referenced through global indices.
The global variables hold a value and can either be mutable or immutable.

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

2. WebAssembly abstract machine

Stack

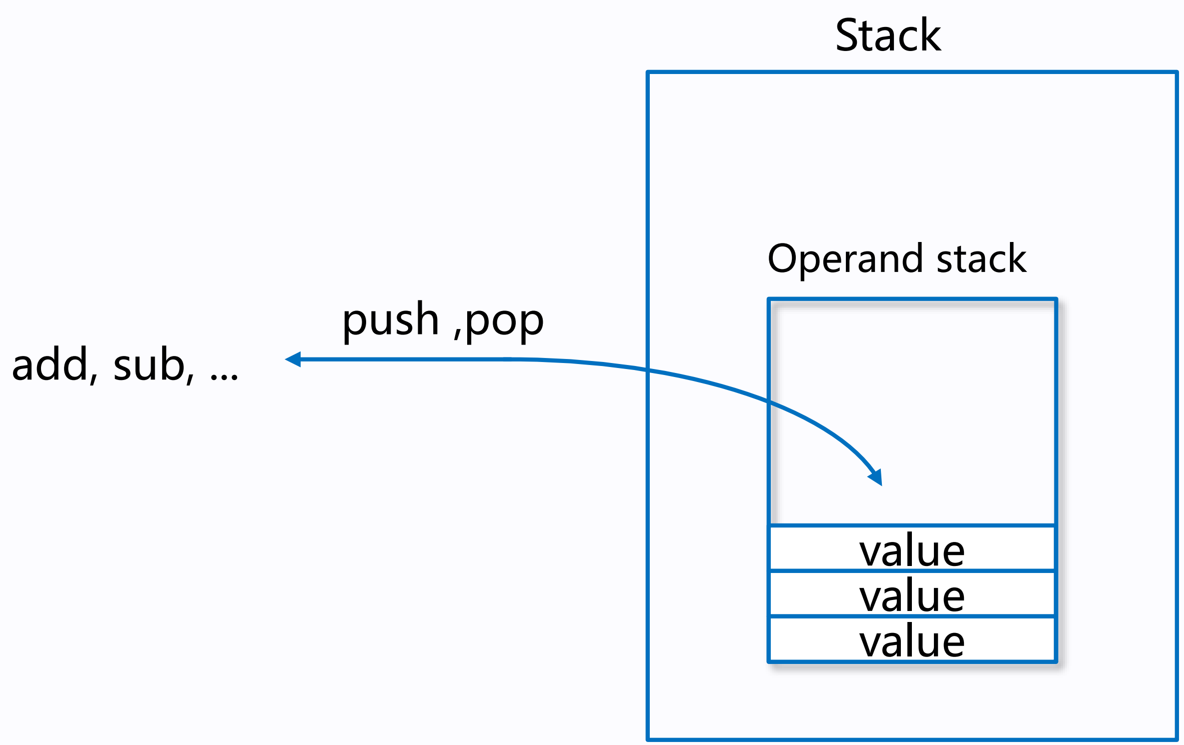
Stack



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

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

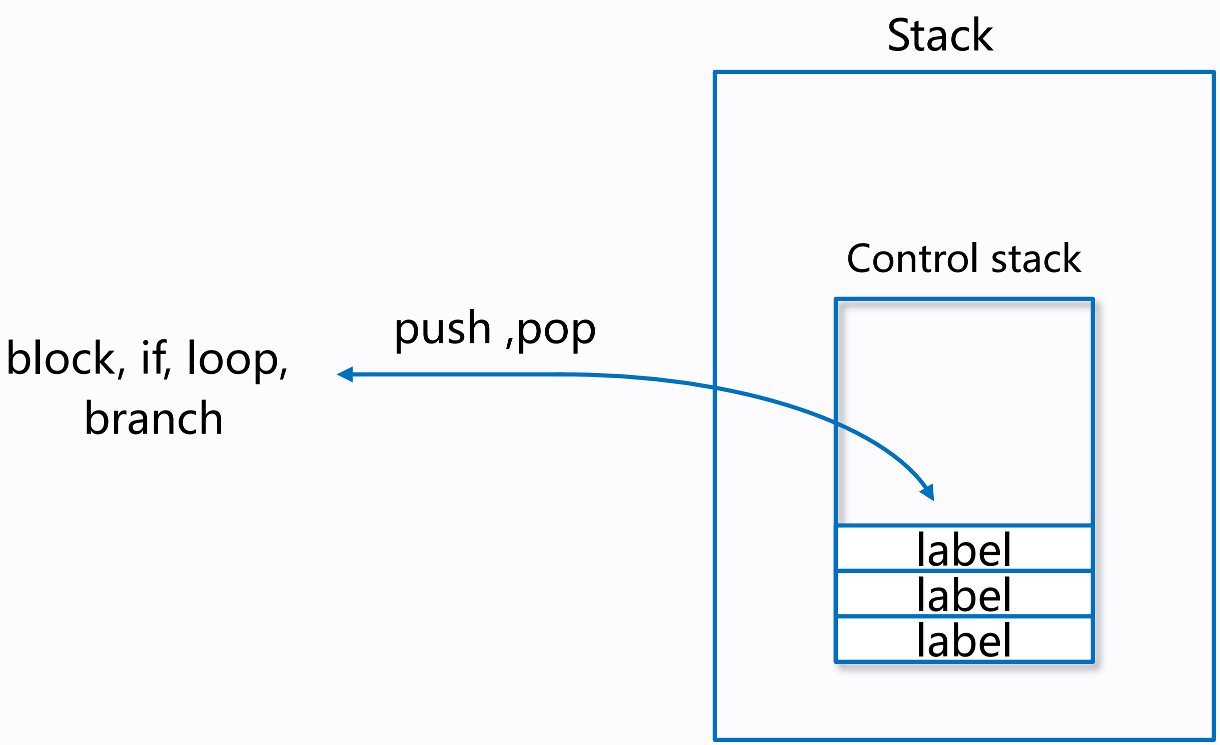
Operand stack



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, [2], [4]

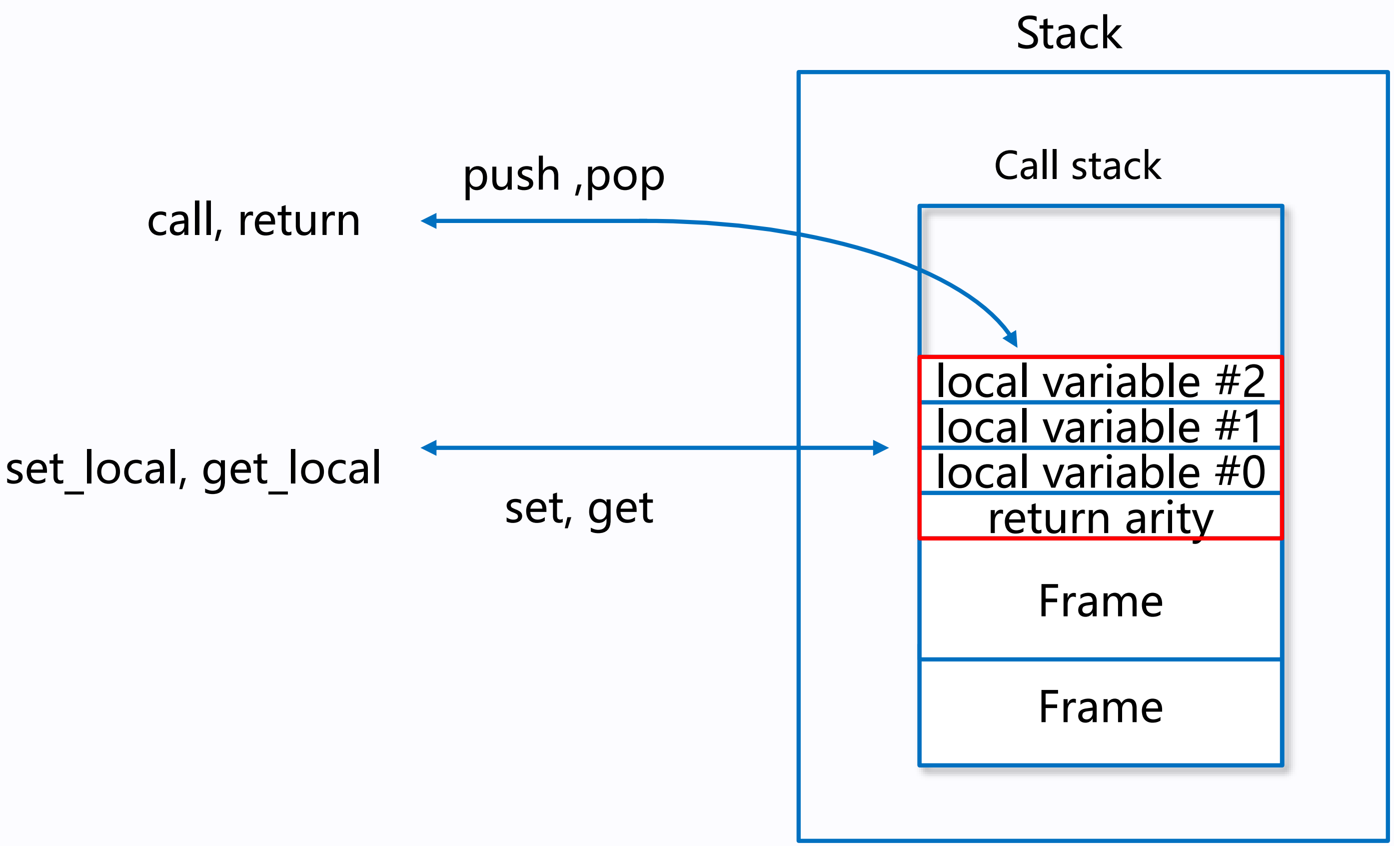
Control stack



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, [2], [4]

Call stack

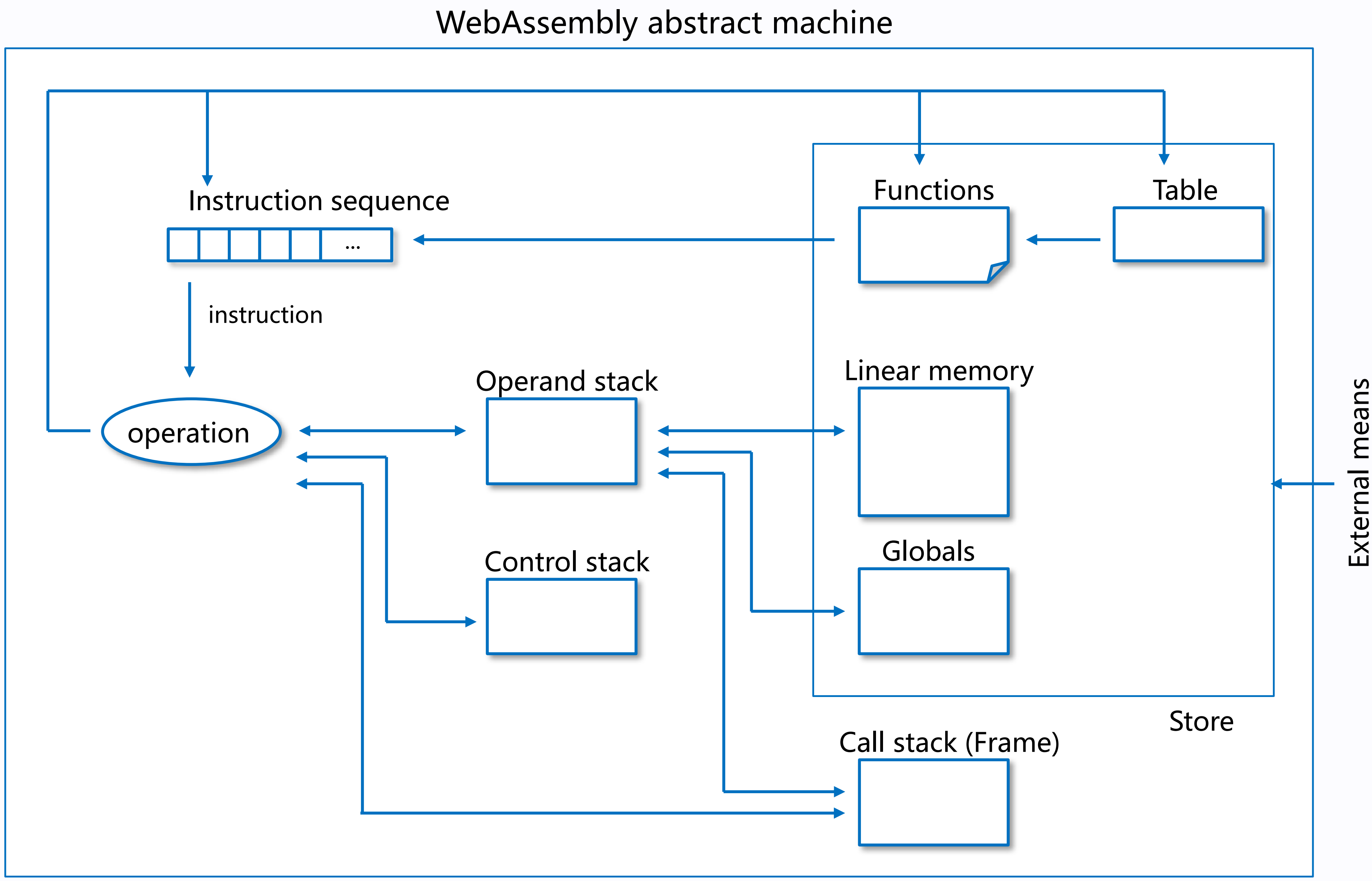


Frames hold the values of its local variables (including arguments).
Frames also carry the return arity of the respective function.

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

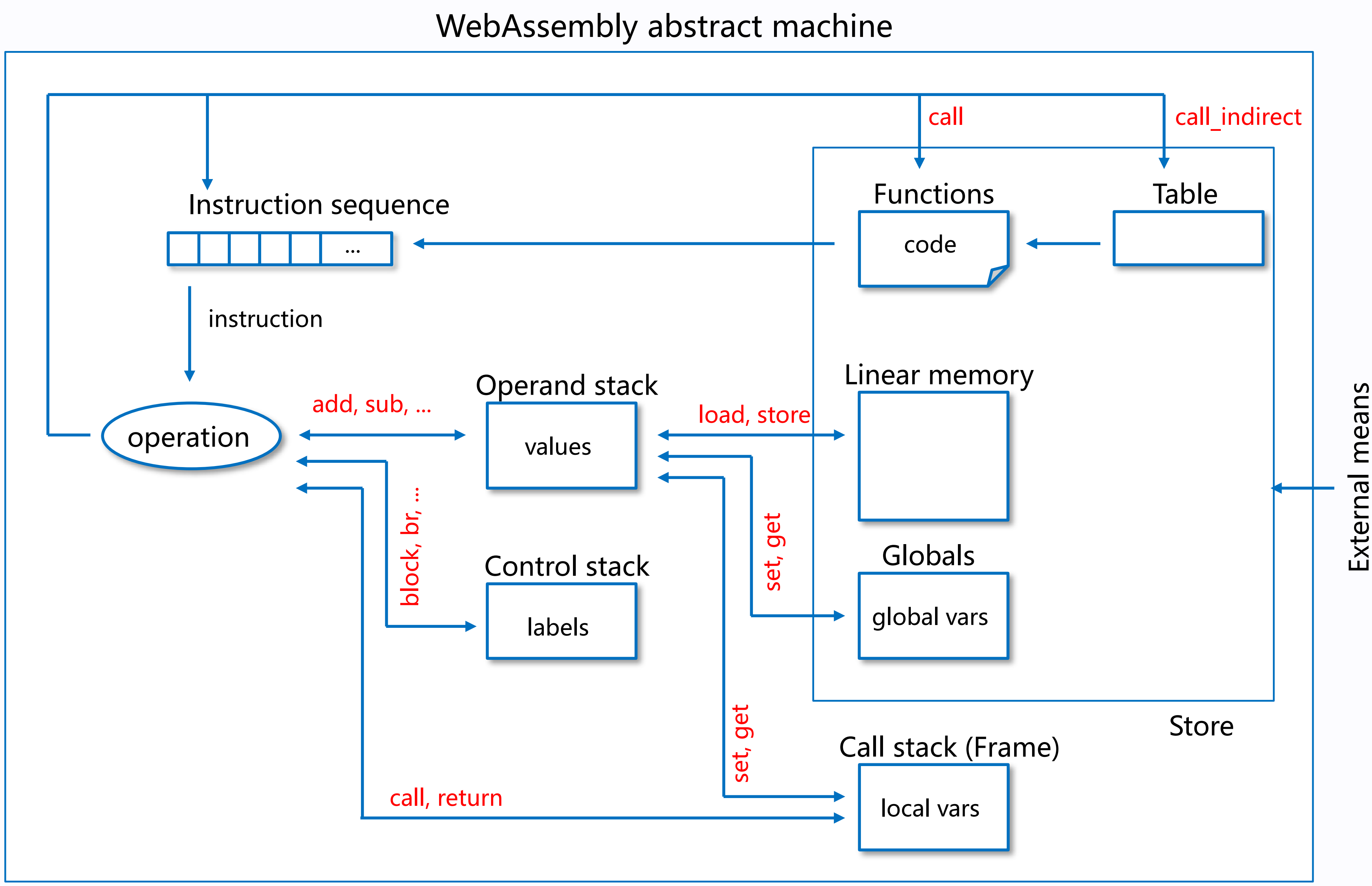
Computational model

Computational model



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

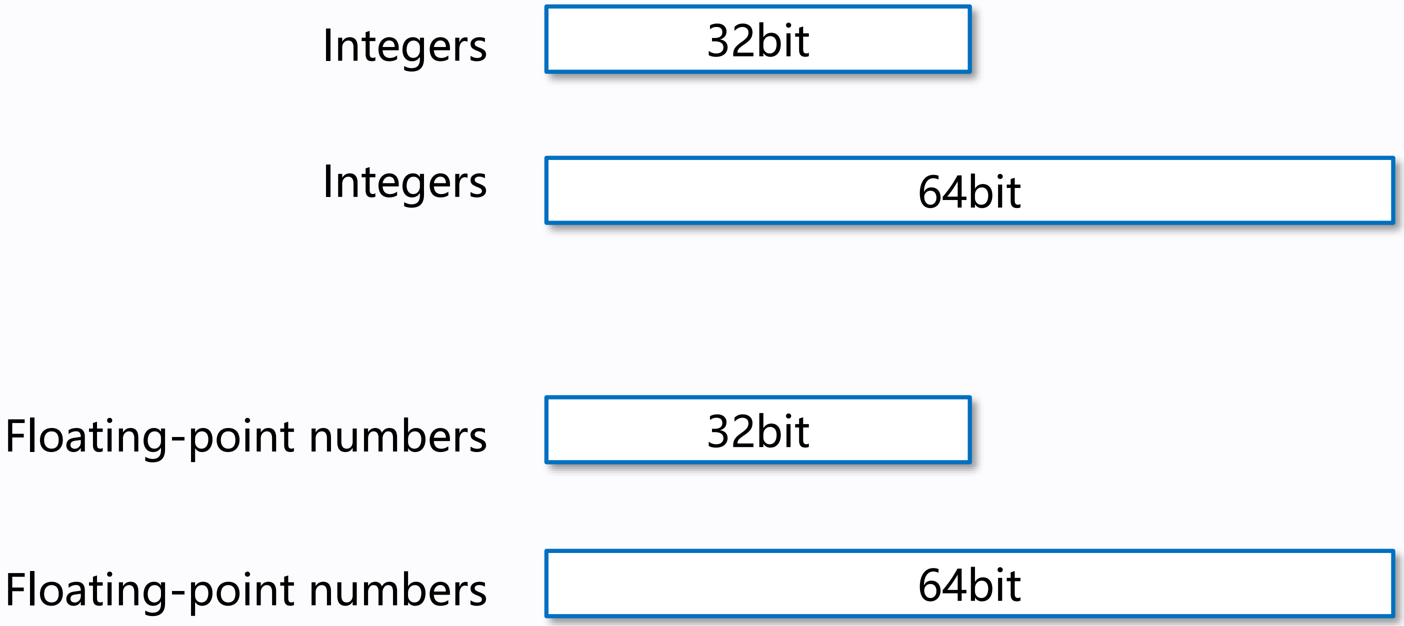
Computational model



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

Type

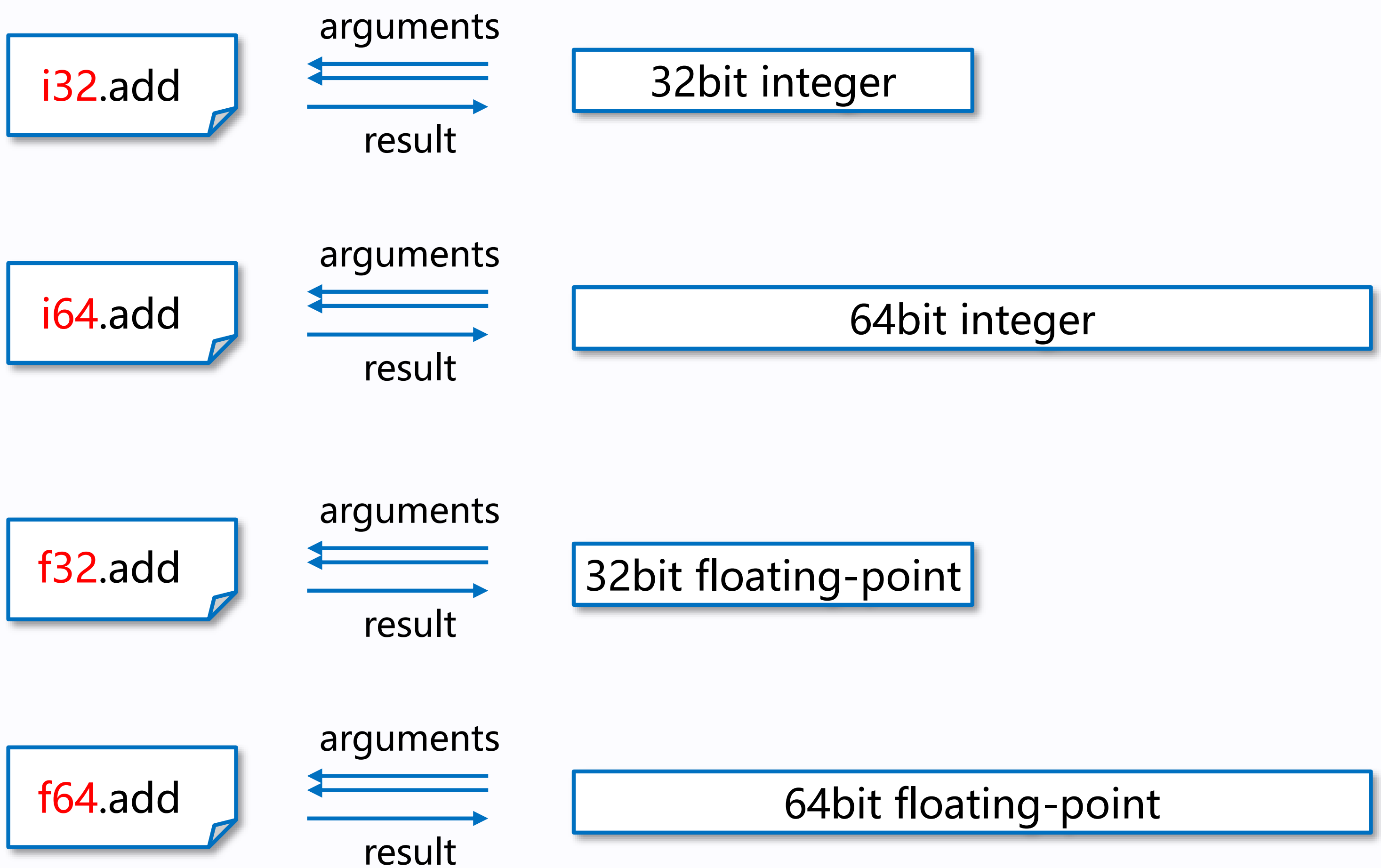
Value types



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

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

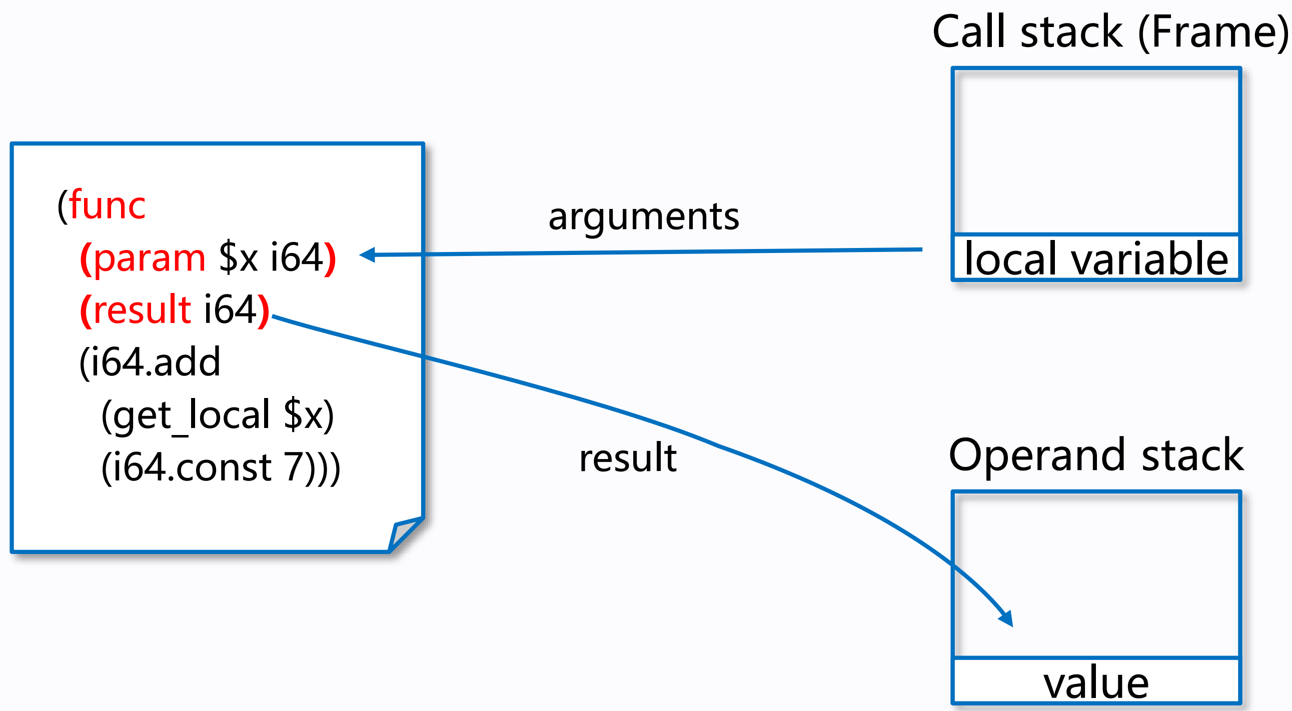
Instructions have type annotations



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, [2]

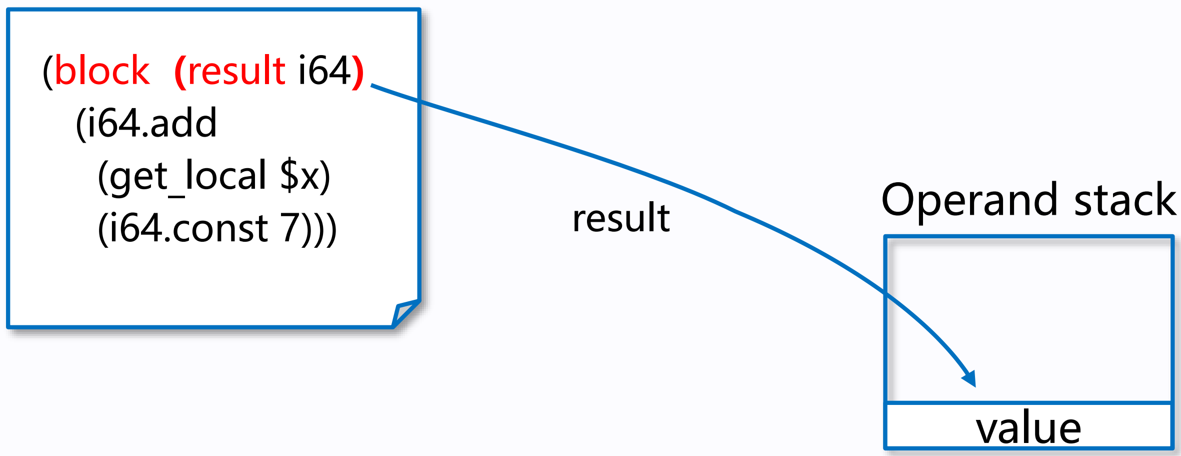
Functions have type declarations



Each function takes a sequence of WebAssembly values as parameters and returns a sequence of values as results as defined by its function type.

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

Control blocks have also a type declaration

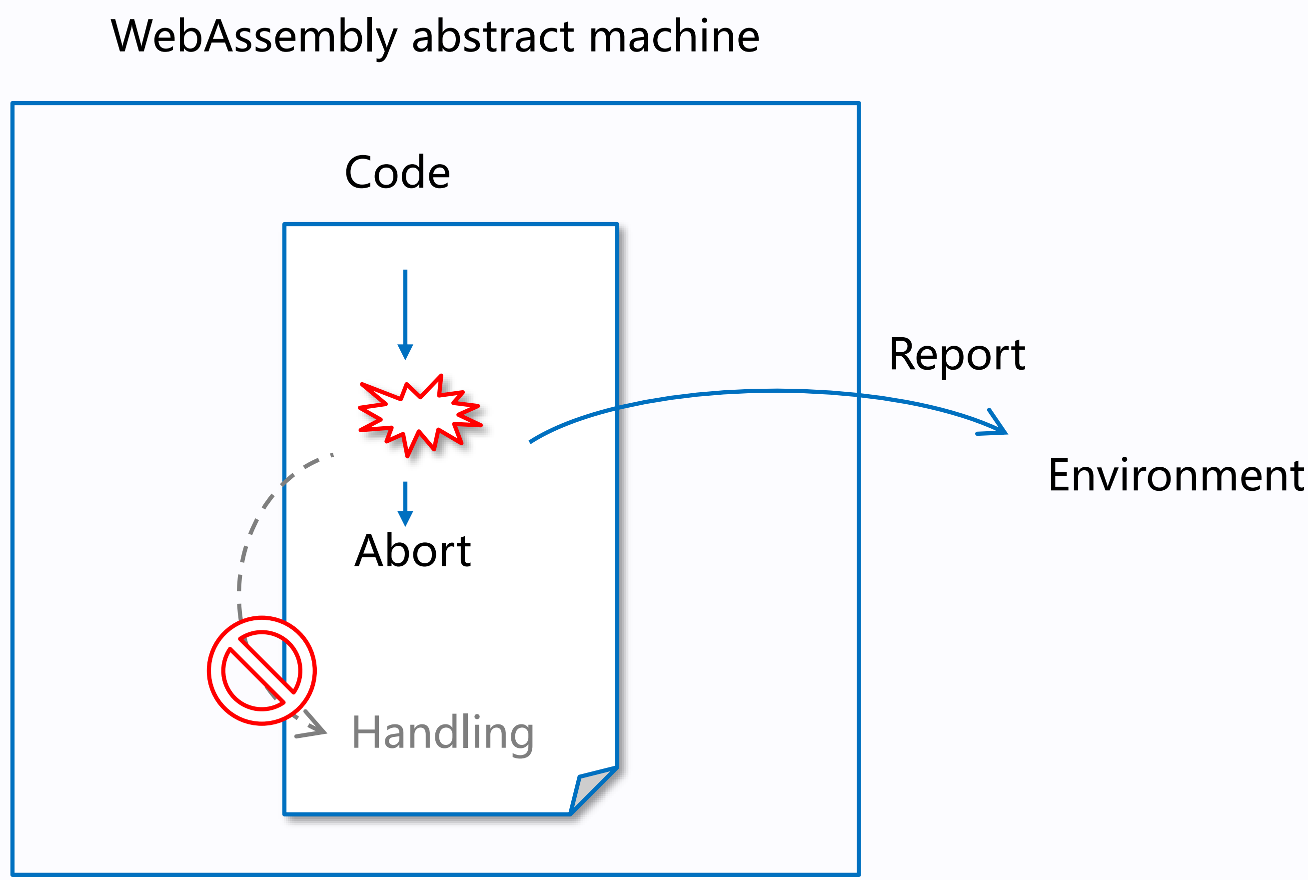


Every control construct is annotated with a function type.

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

Trap

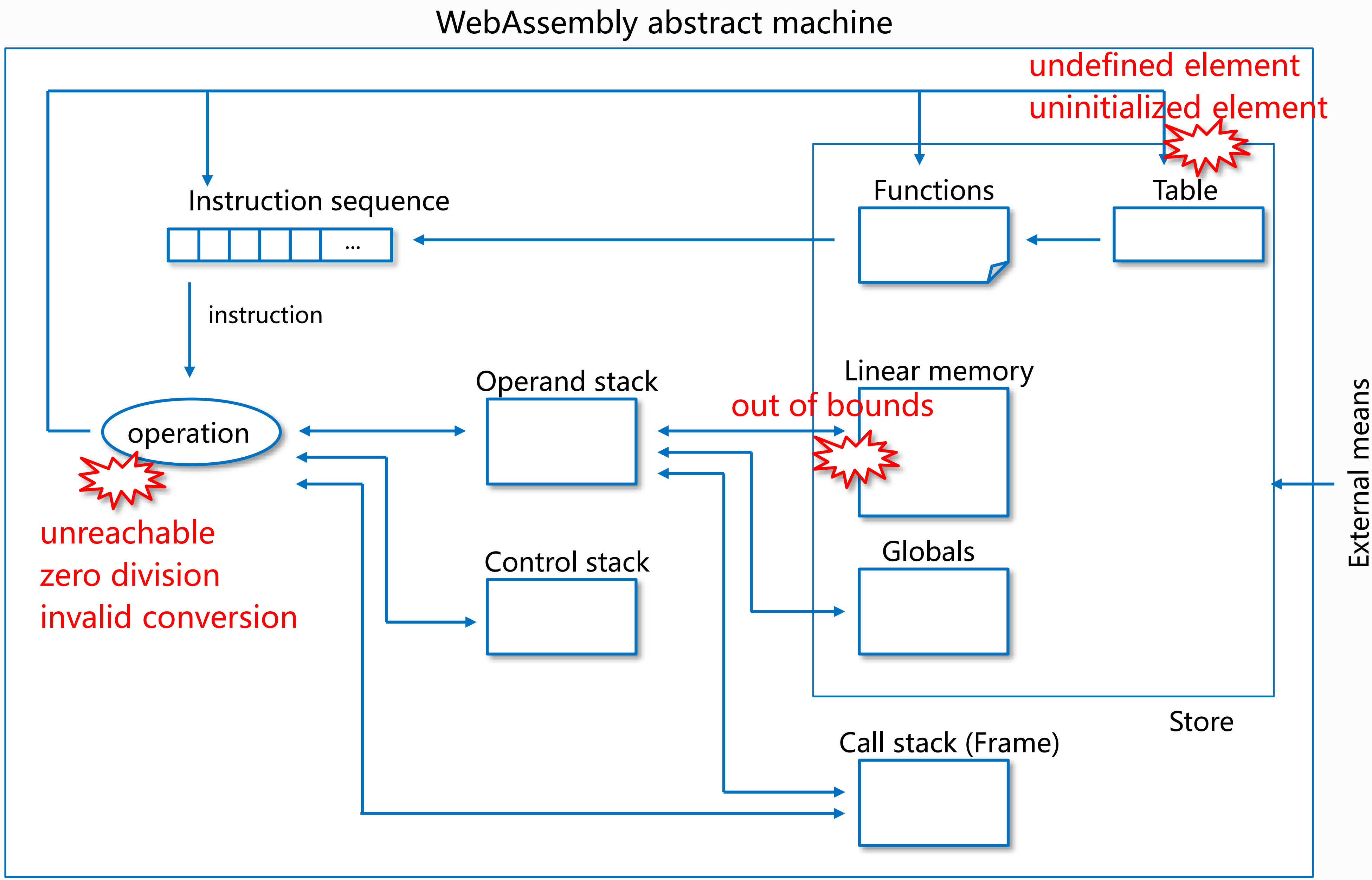
Trap



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.

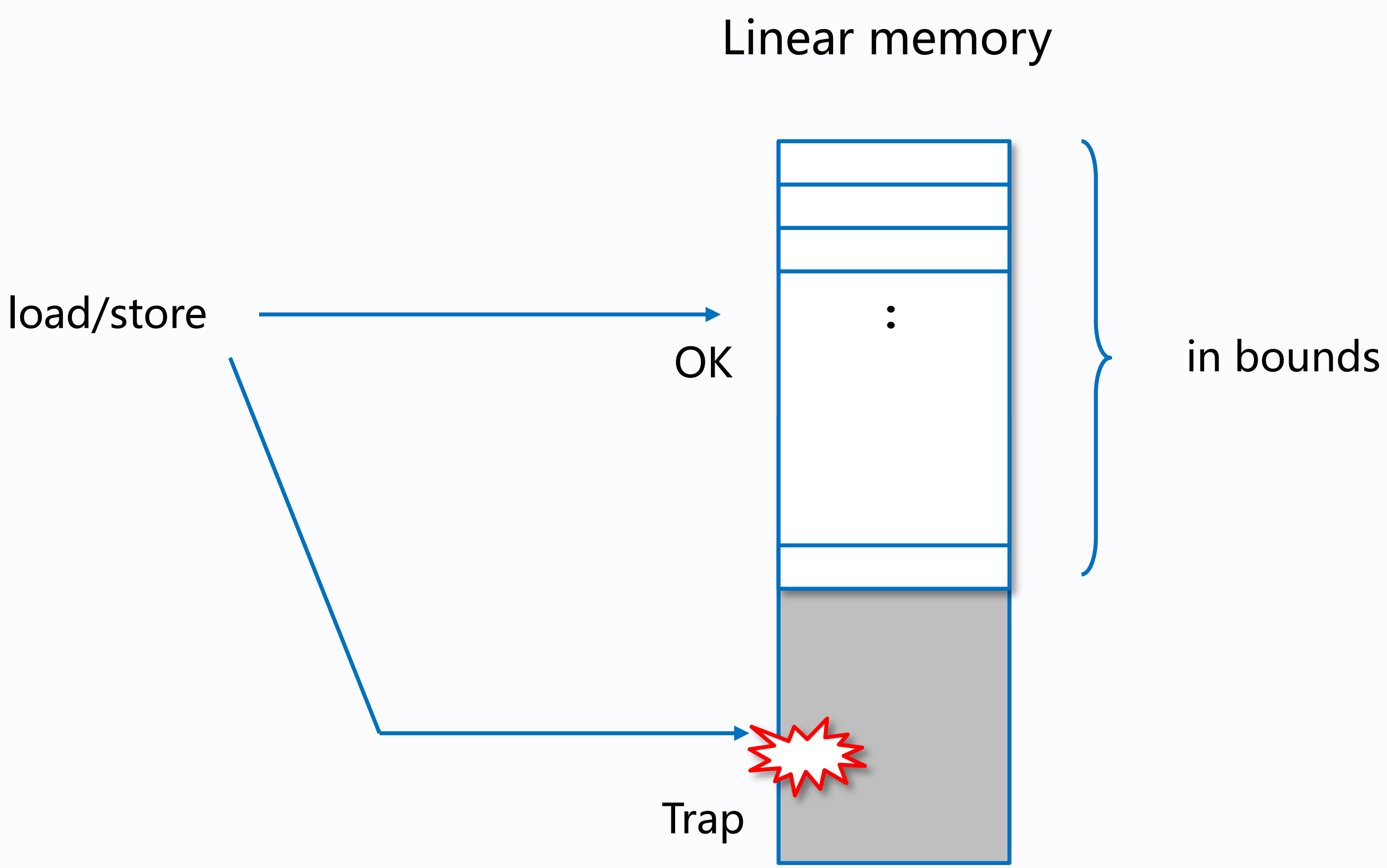
References : [1] Ch.1, Ch.4 , [2], [4]

Trap



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

Linear memory

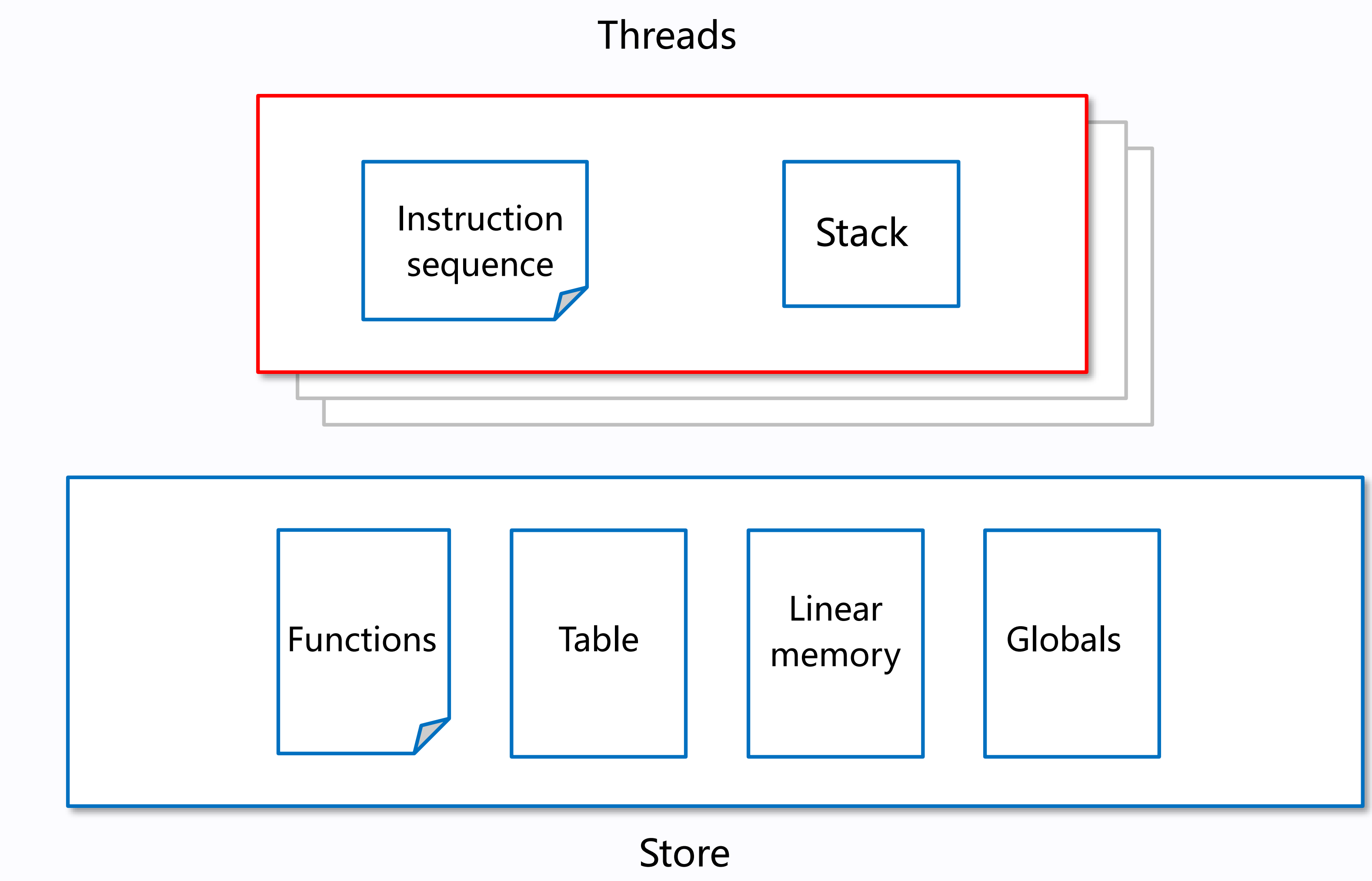


A trap occurs if an access is not within the bounds of the current memory size.

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

Thread

Thread



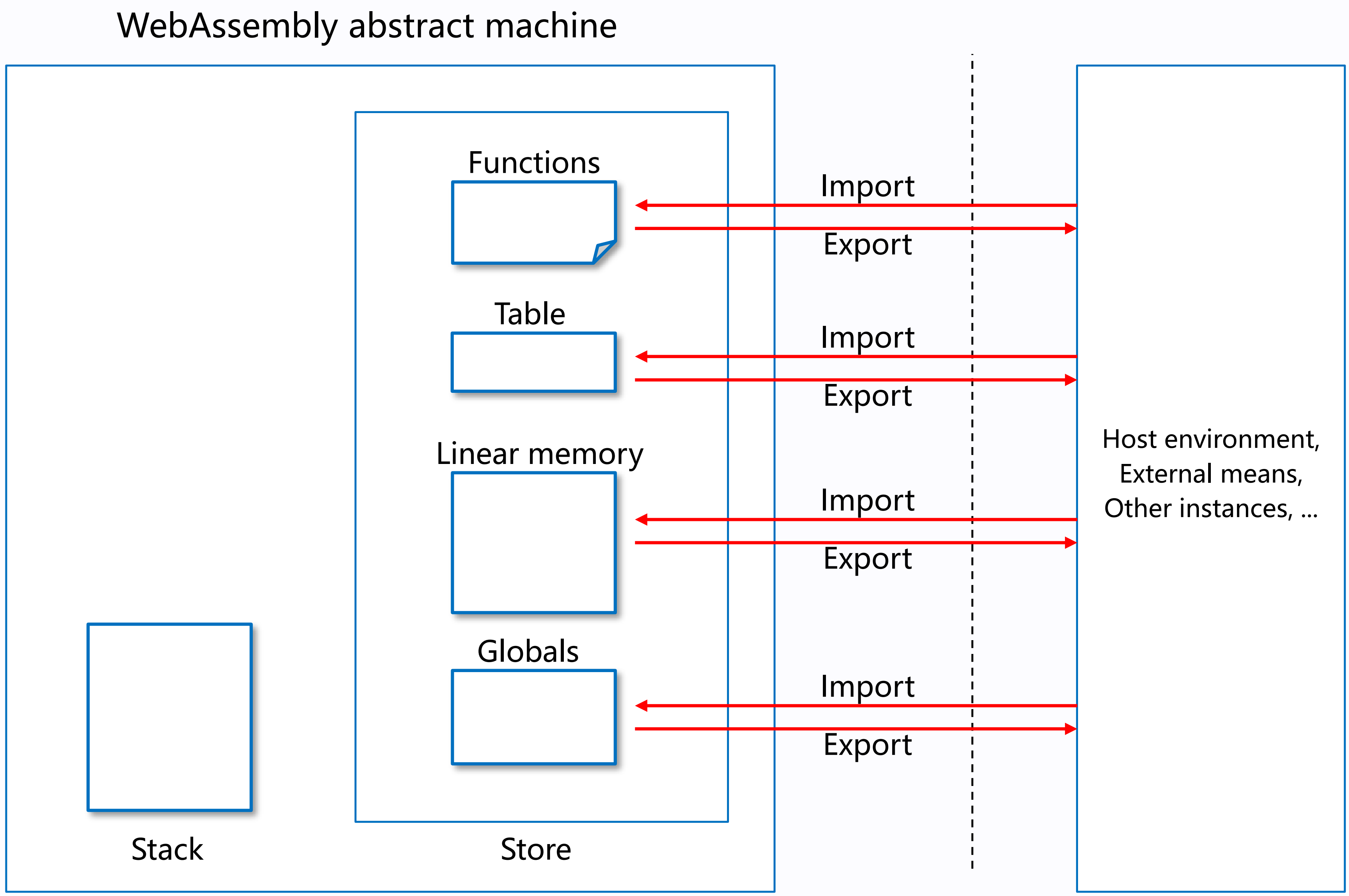
The current version of WebAssembly is single-threaded,
but configurations with multiple threads may be supported in the future.

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

2. WebAssembly abstract machine

External interface

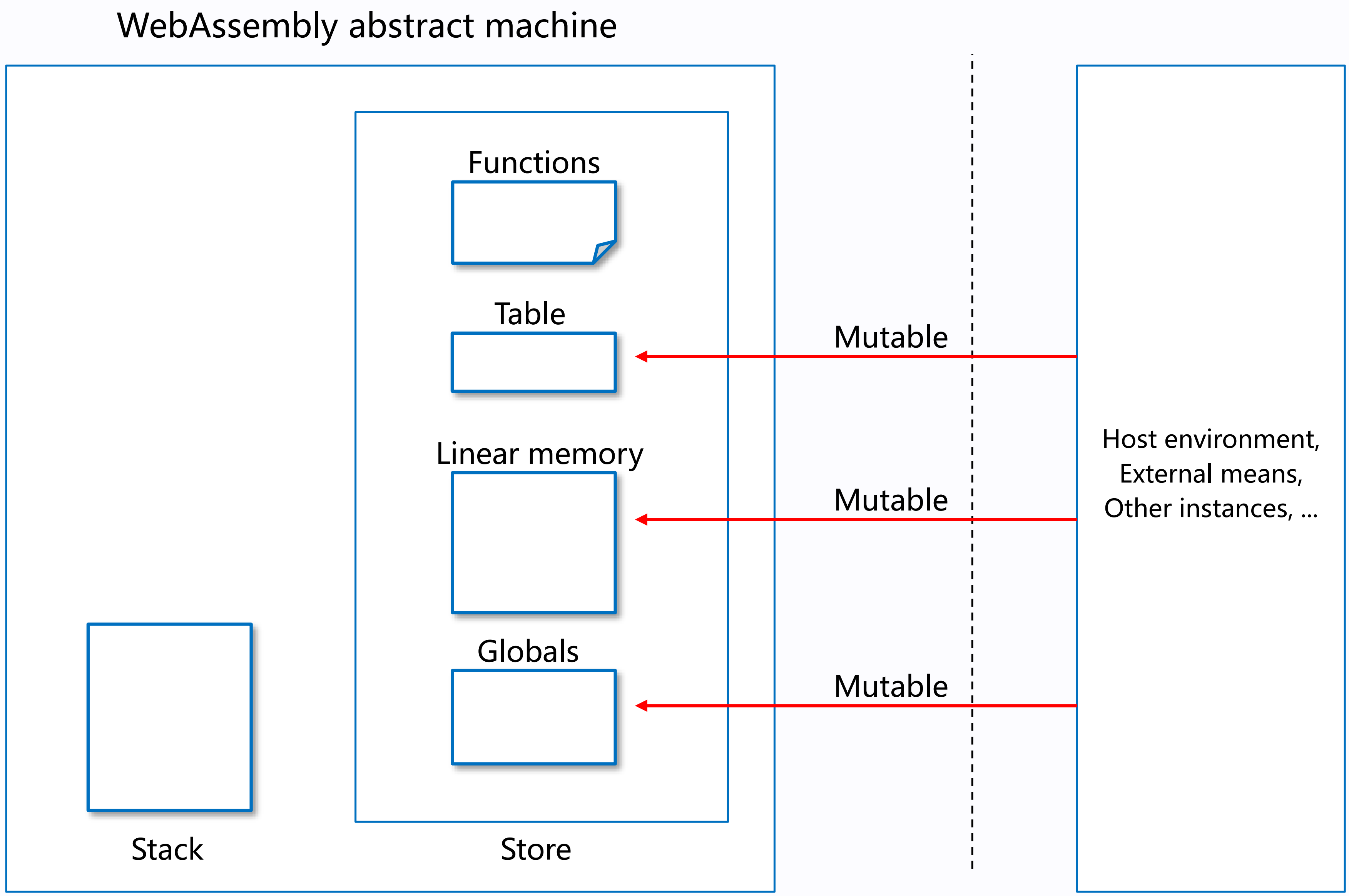
Import and export



Functions, table, memory and globals may be shared via import/export.

References : [1] Ch.2, Ch.4. [2]. [5]

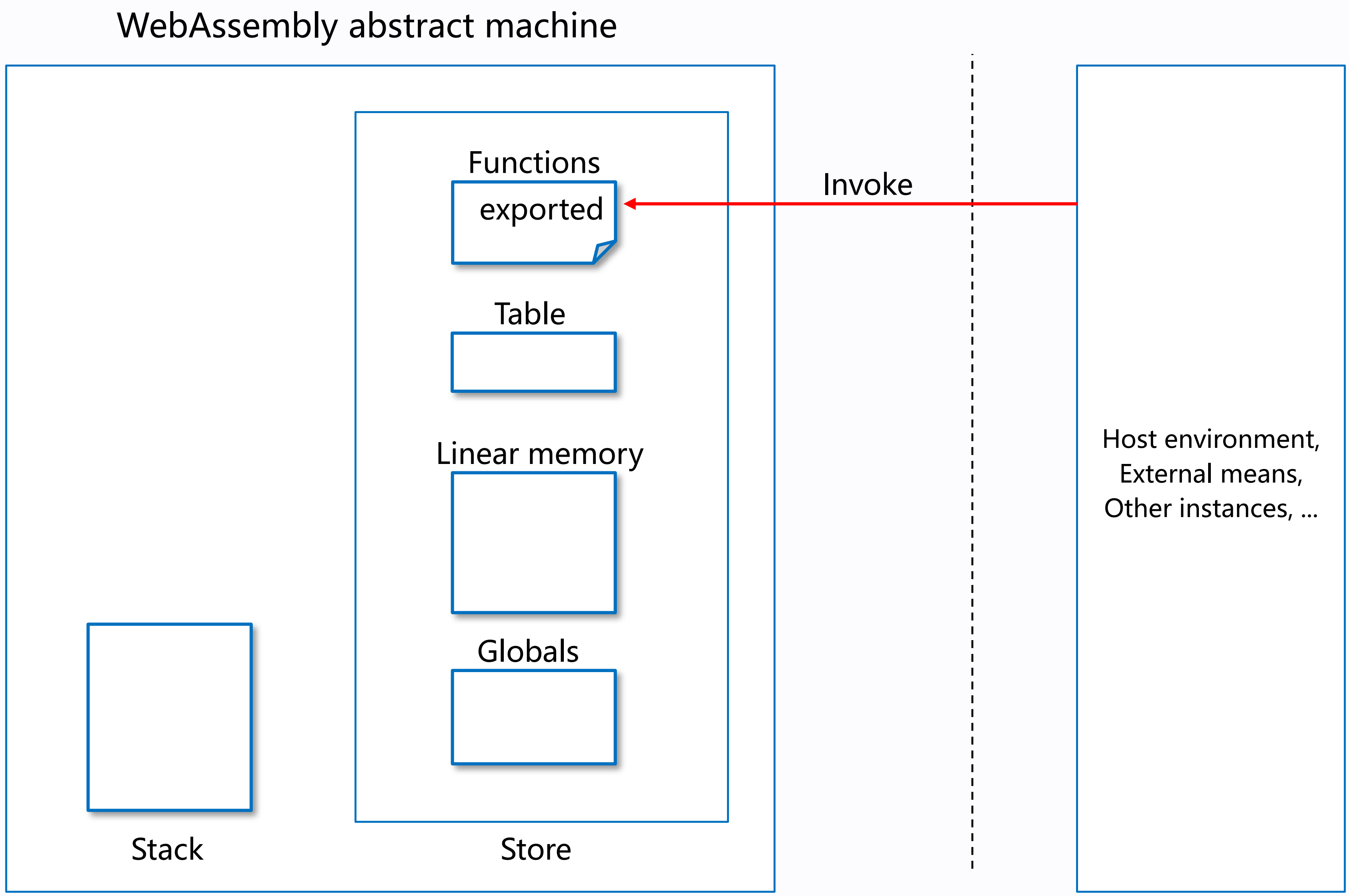
Mutation from external



Table, memory and globals can be mutated by external mean.

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

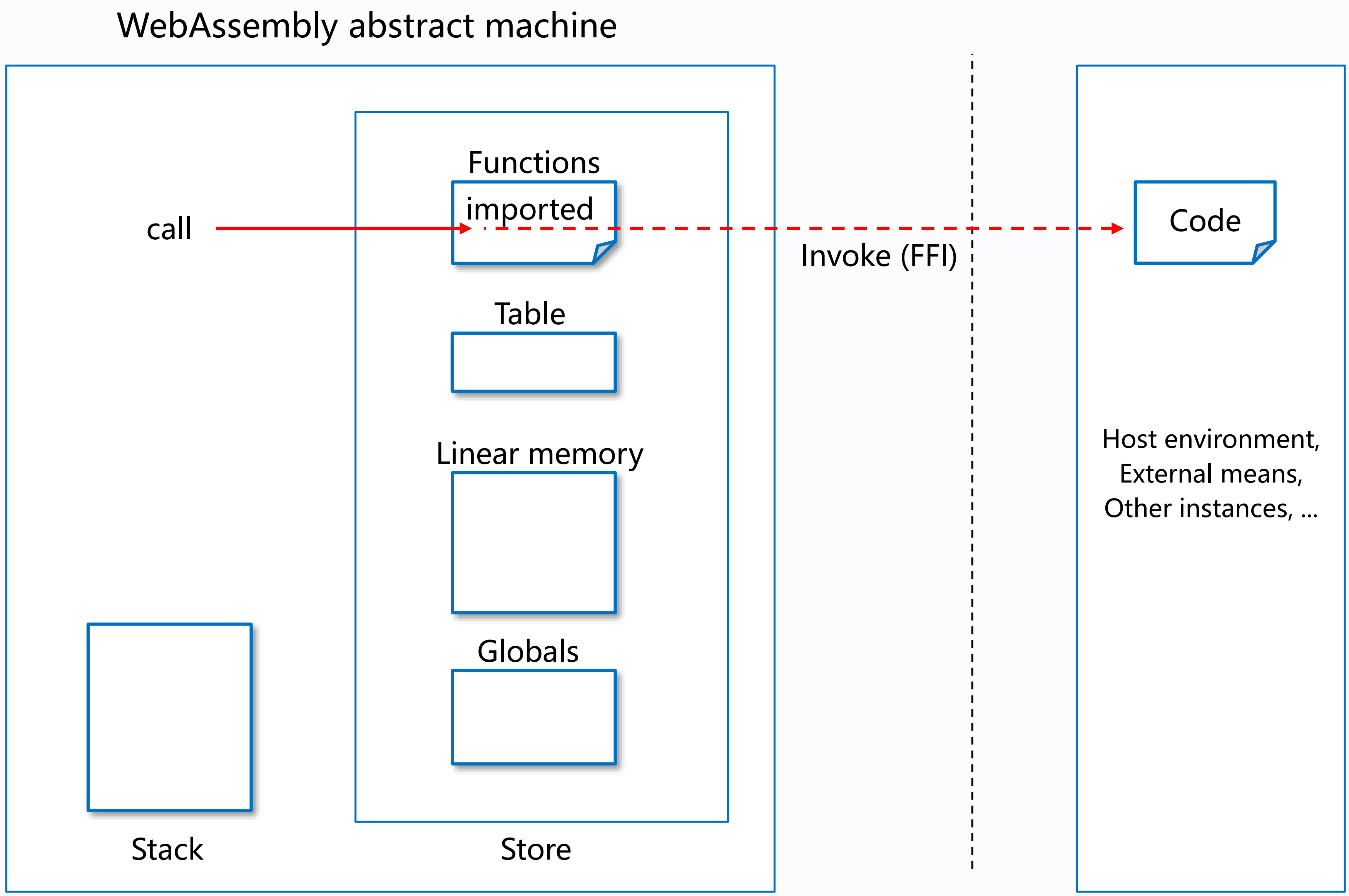
Invoke from external



Any exported function can be invoked externally.

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

Foreign call



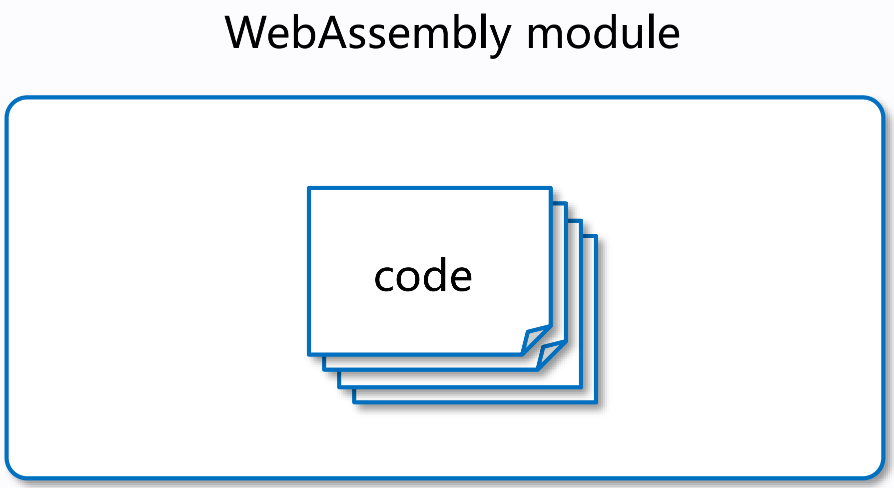
Call instructions can invoke an imported function.

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

3. WebAssembly module

Module

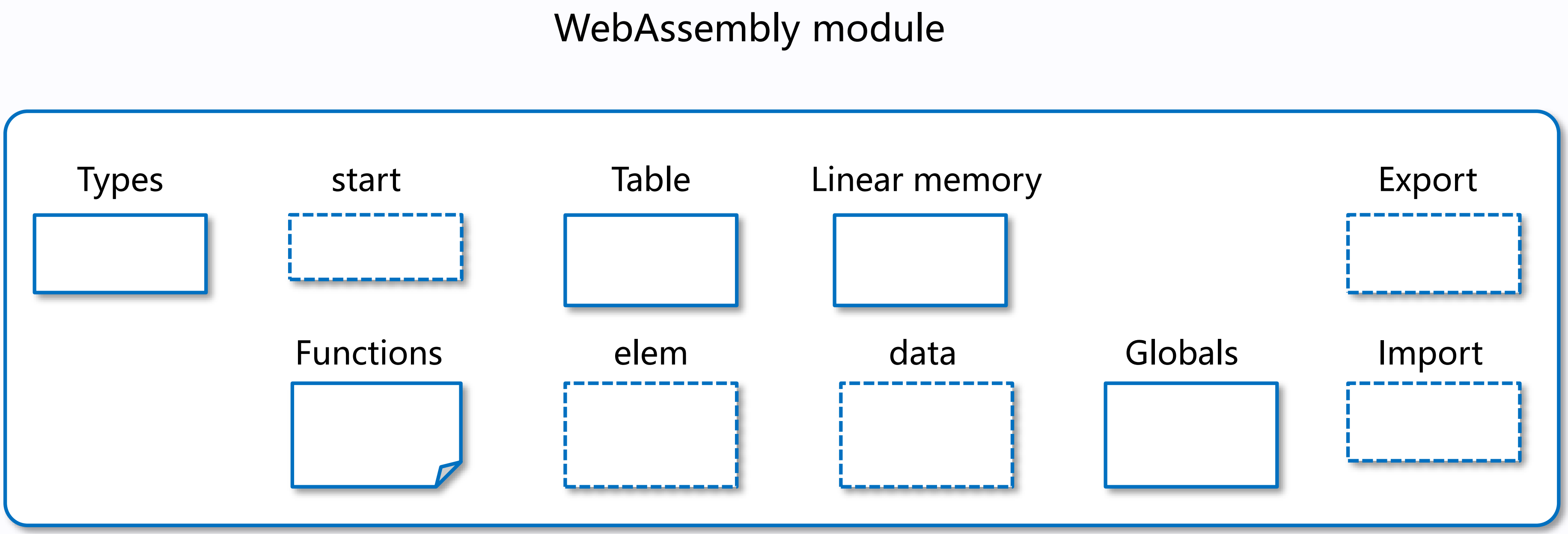
WebAssembly module



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.2, Ch.4, [2], [4], [5]

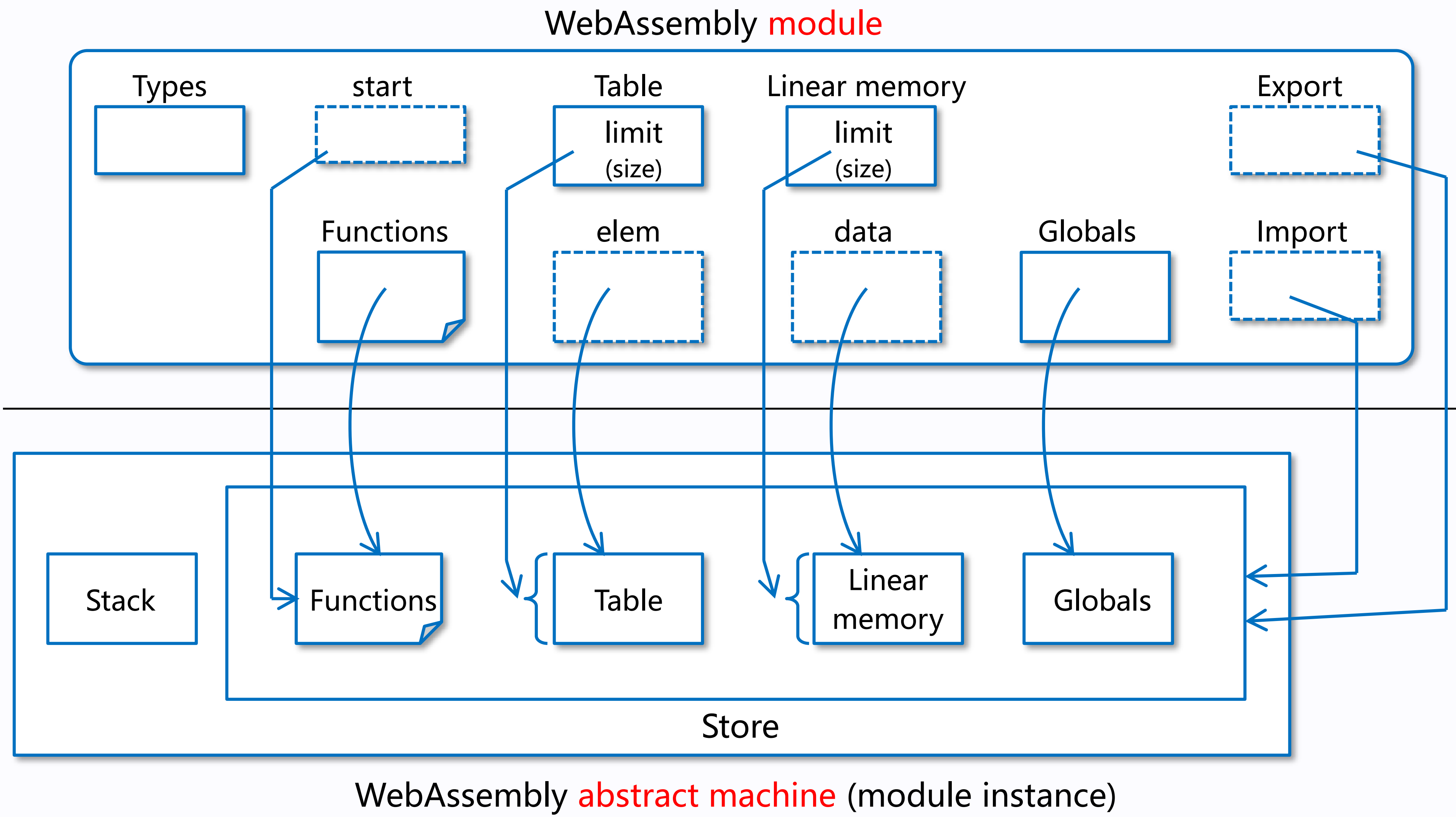
WebAssembly module



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, [2], [4], [5]

WebAssembly module and abstract machine

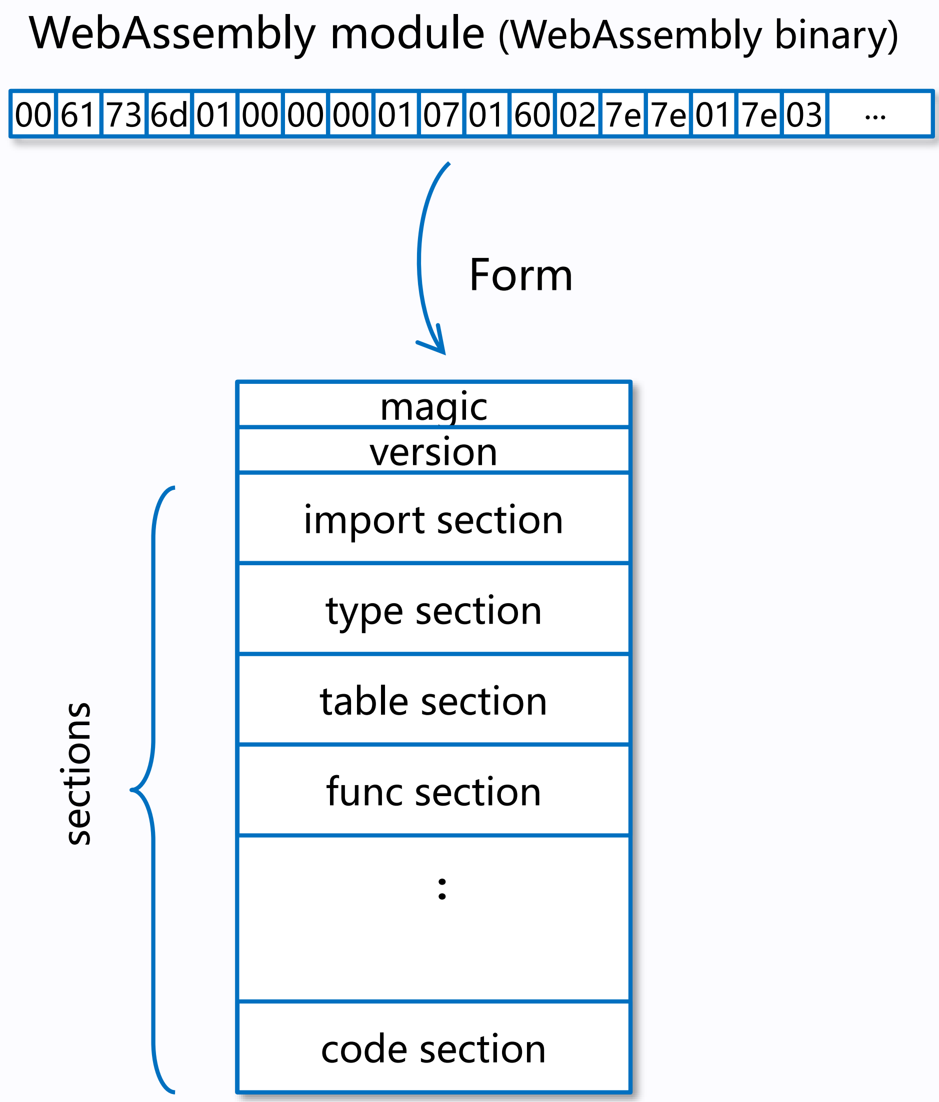


A module corresponds to the static representation of a program.
A module instance corresponds to a dynamic representation.

References : [1] Ch.1, Ch.2, Ch.4, [2], [4], [5]

Binary encoding

Binary encoding of modules



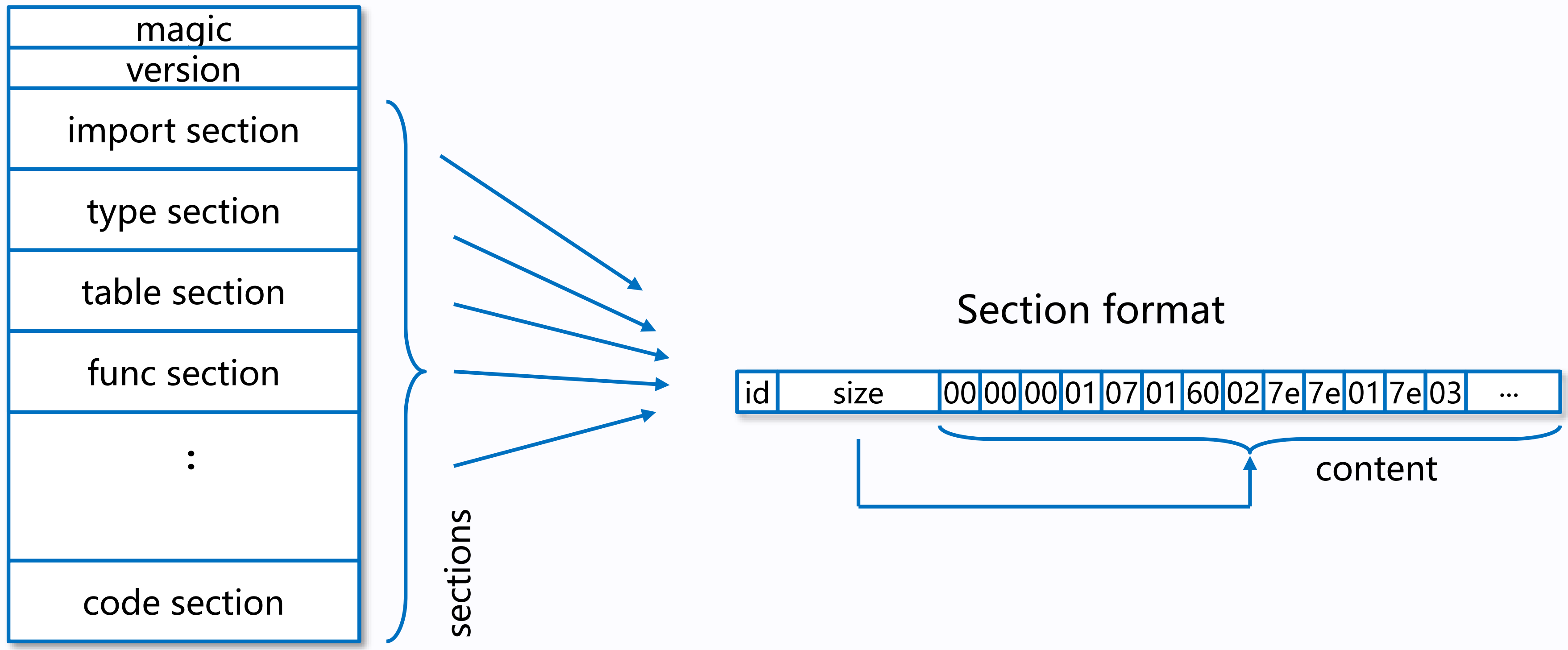
Binary encoding of modules

The binary encoding of modules is organized into sections.

References : [1] Ch.5, [7], [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, [7], [5]

Example of WebAssembly module

[text format]

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

(`wat2wasm –v` command)

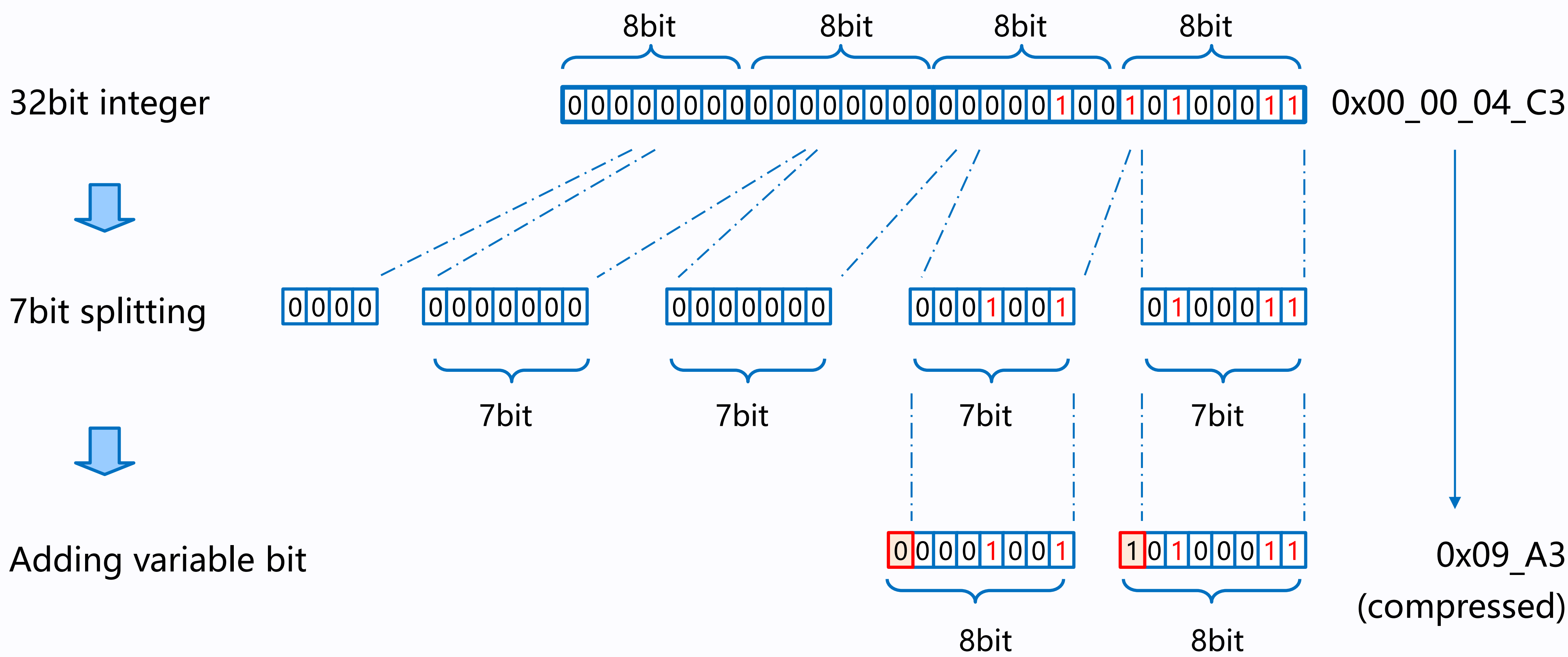
[binary format]

```
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, Ch.6, [7], [5]

Integer encoding with LEB128



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

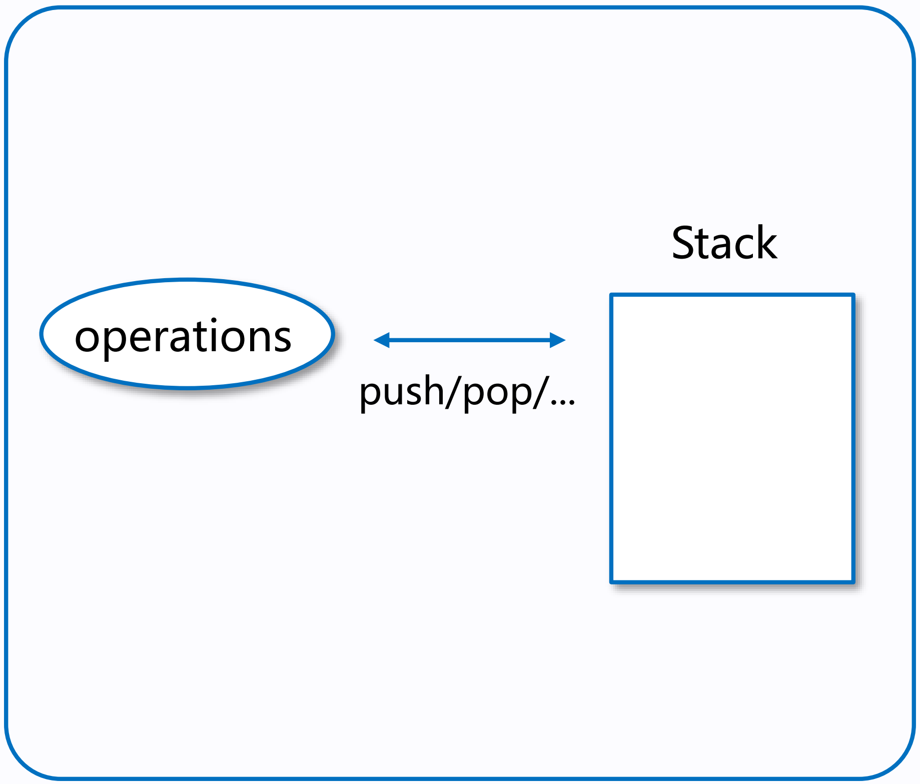
References : [8], [1] Ch.5, [2]

4. WebAssembly instructions

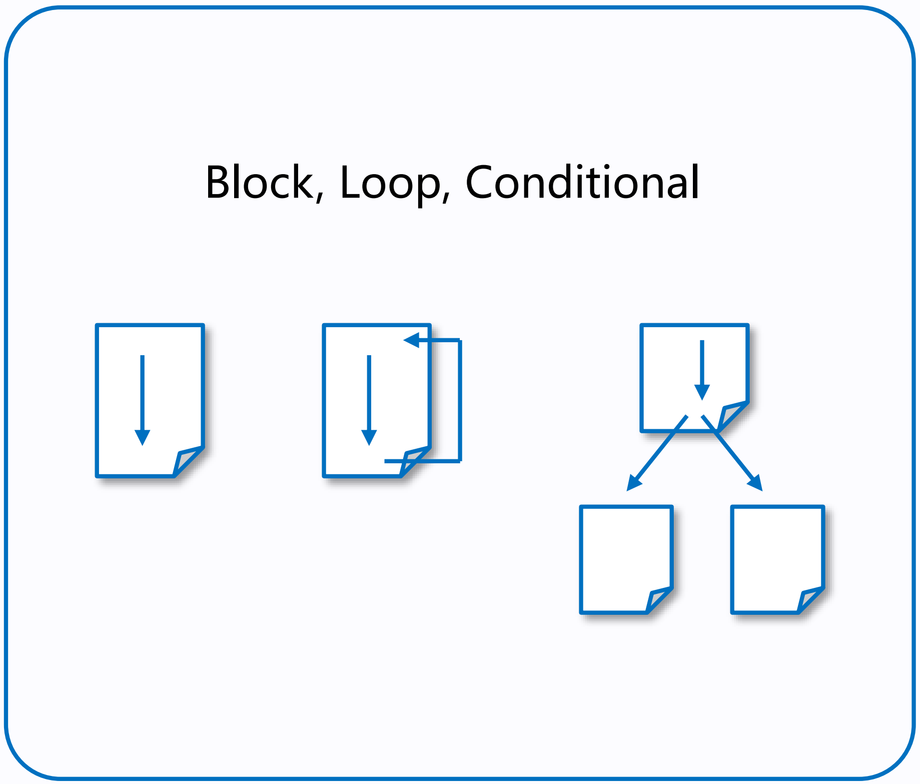
Instructions

Instructions

Simple instructions



Control instructions

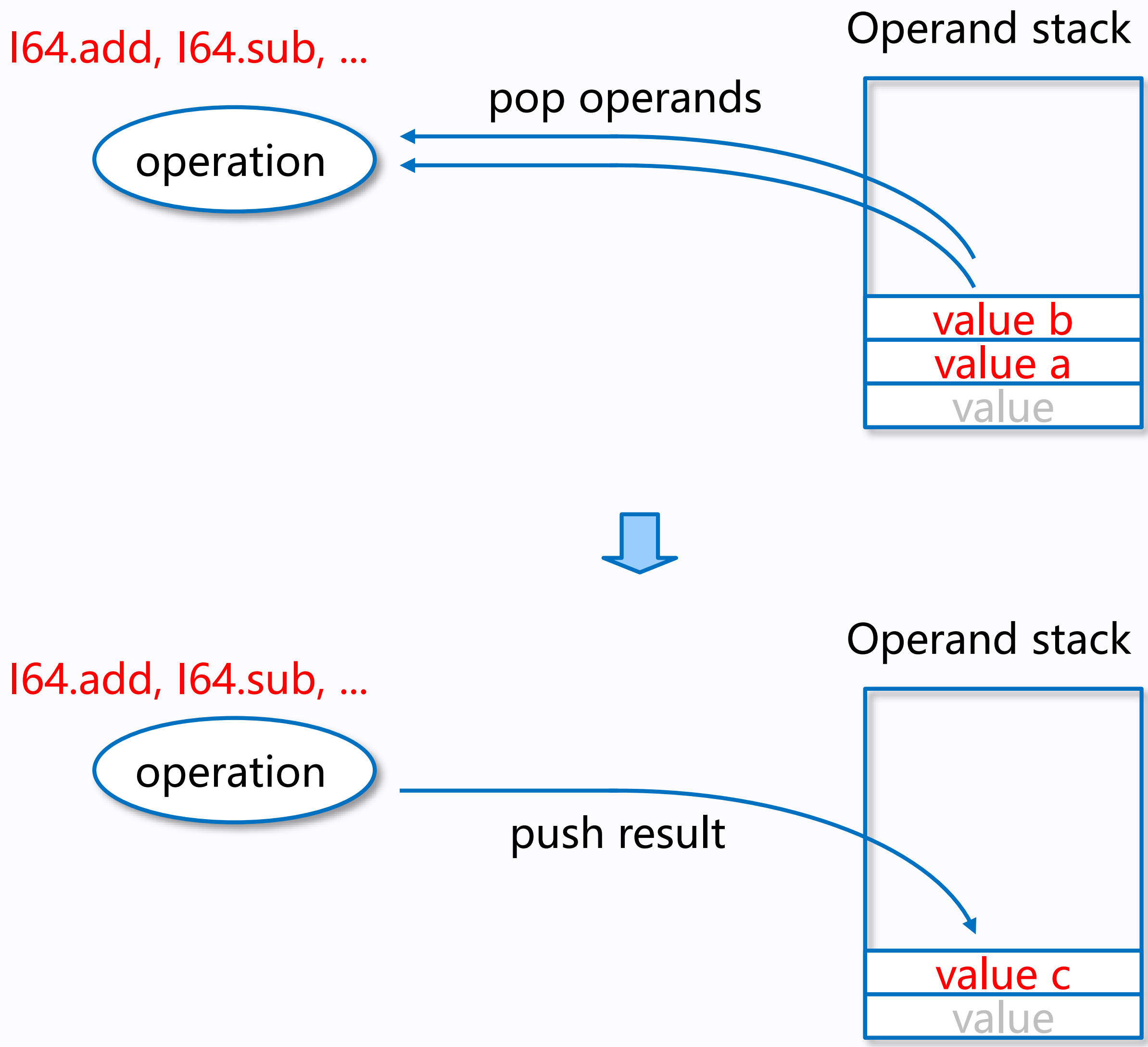


Instructions fall into two main categories.
Simple instructions perform basic operations on data.
Control instructions alter control flow.

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

Simple instructions

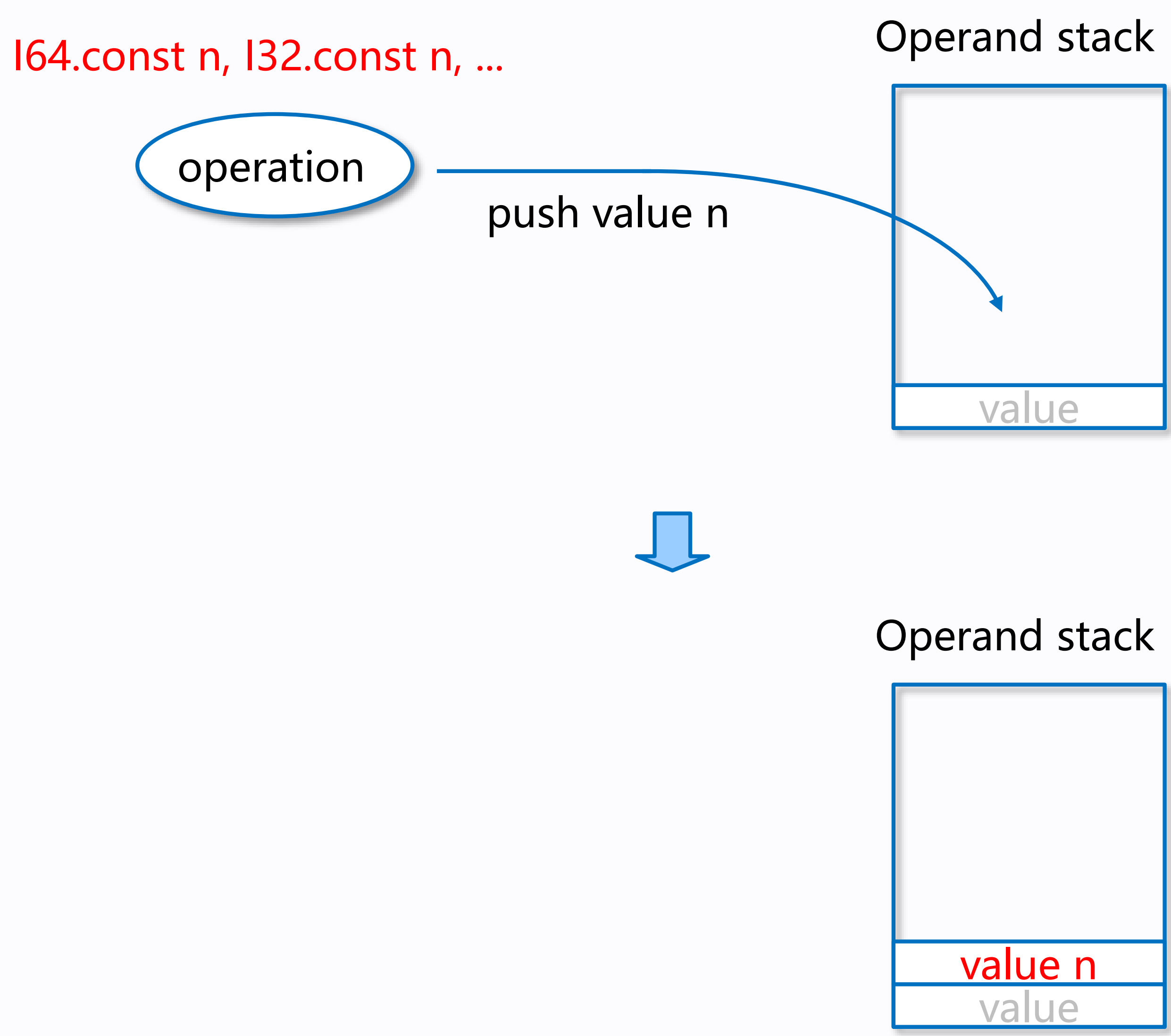
Numeric instructions



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

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

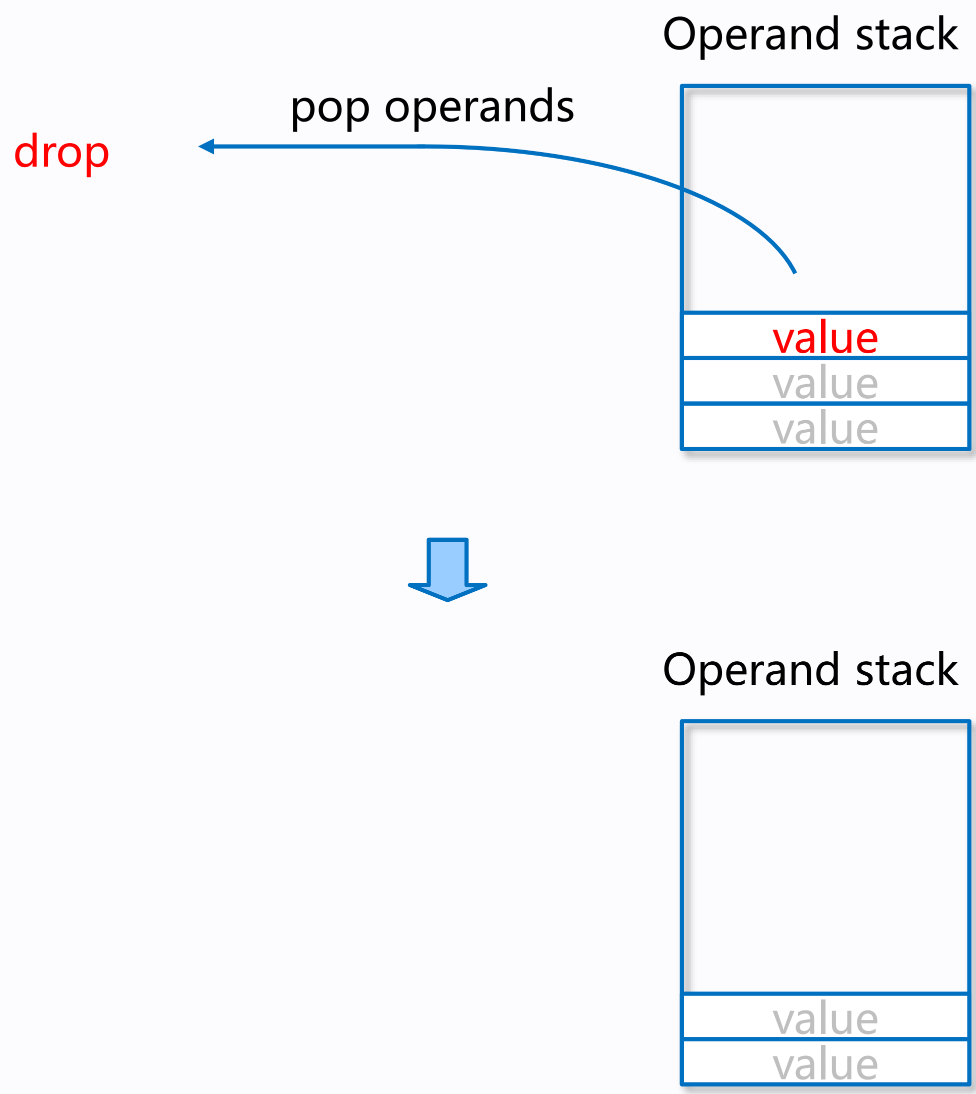
Numeric instructions : const



The const instruction pushes the value to the stack.

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

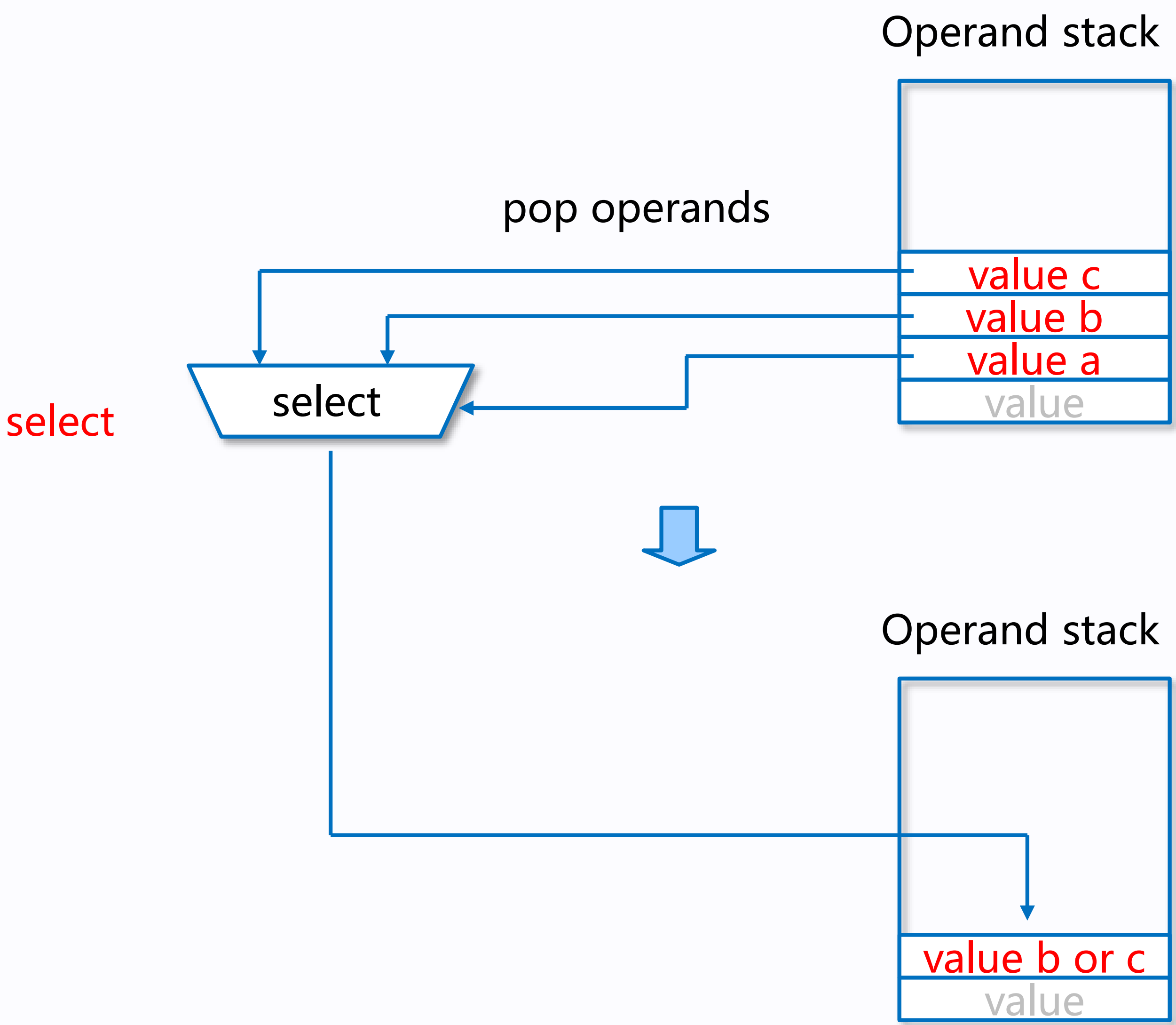
Parametric instructions : drop



The drop instruction simply throws away a single operand.

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

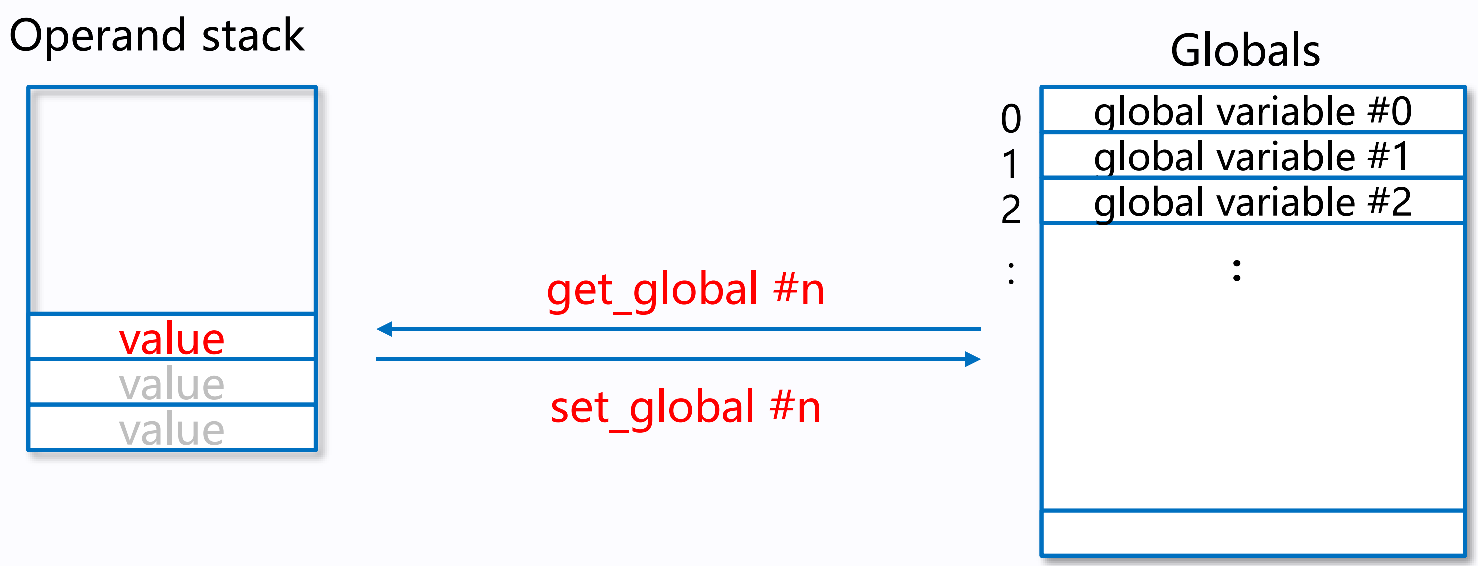
Parametric instructions : select



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 , [2], [4]

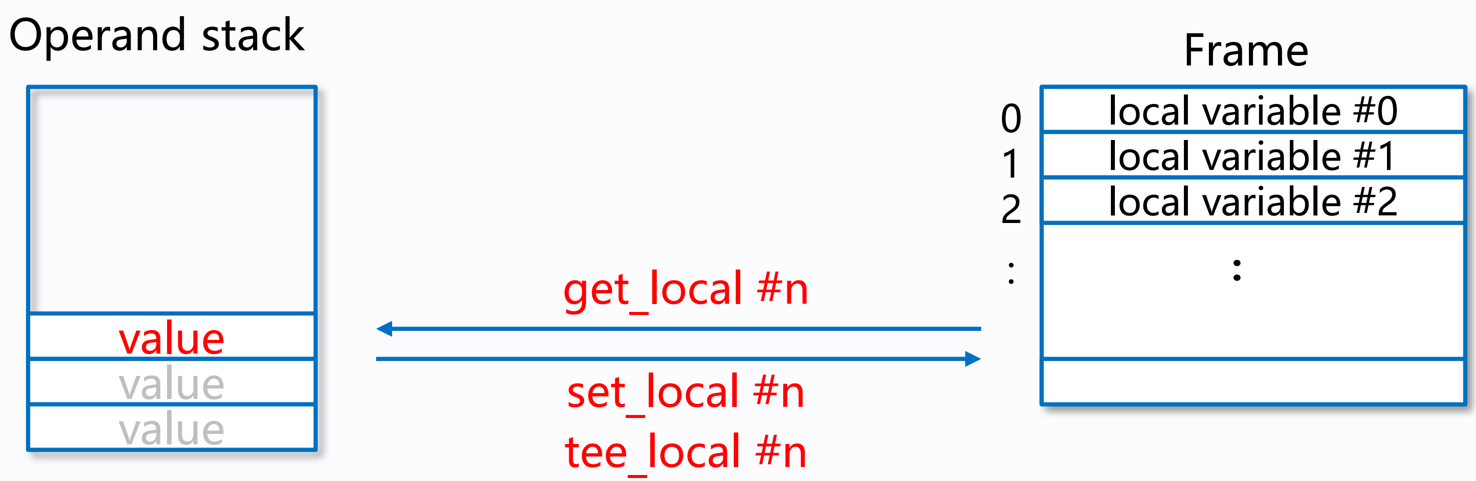
Global variable instructions



Global variable instructions get or set the values of variables.

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

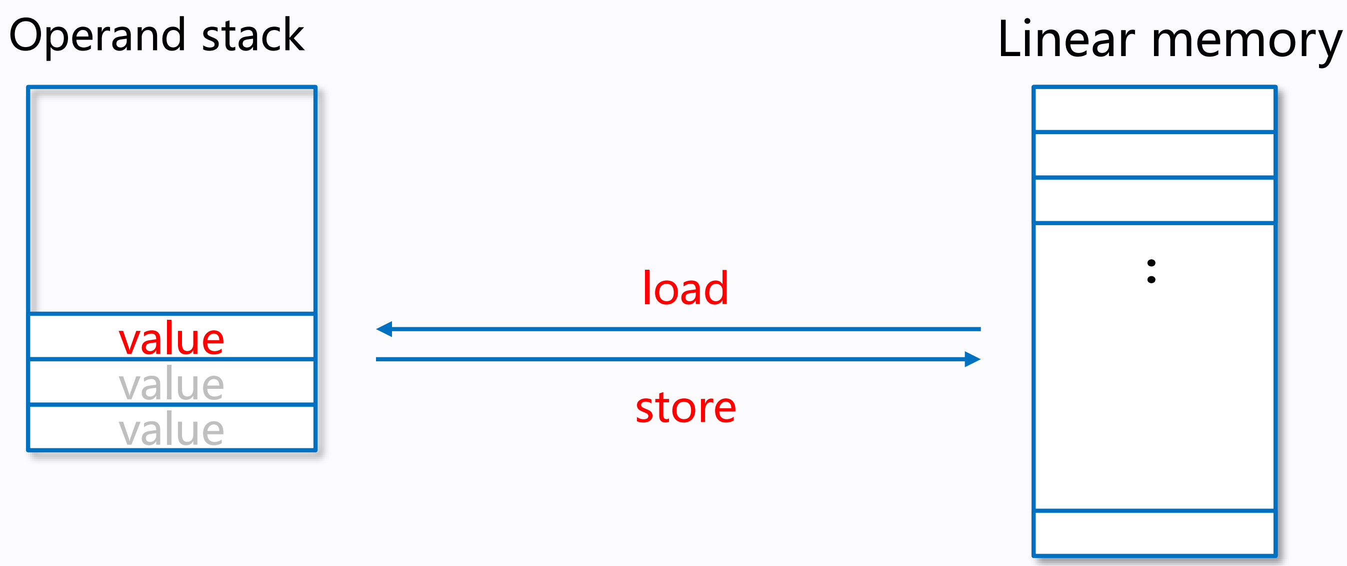
Local variable instructions



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

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

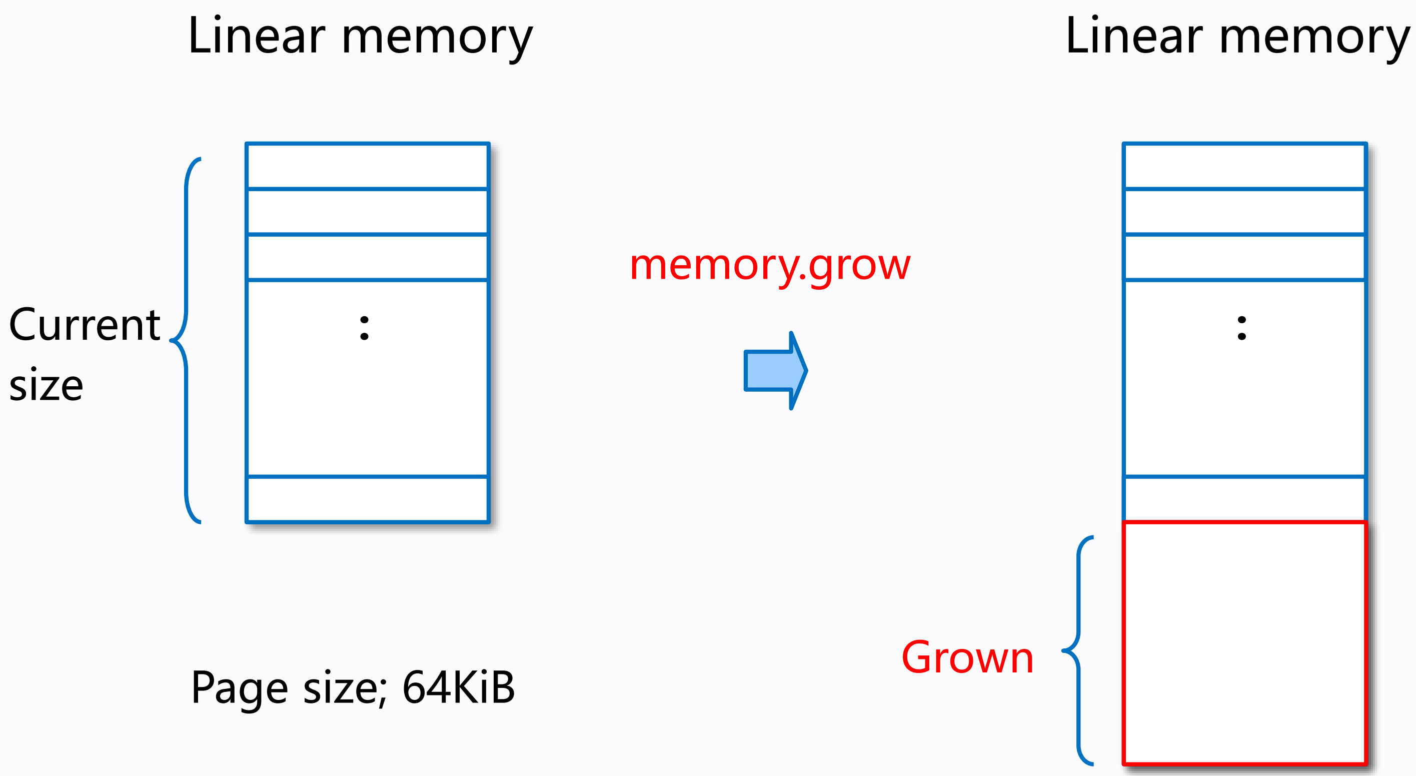
Memory instructions : load, store



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

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

Memory instructions : memory.grow

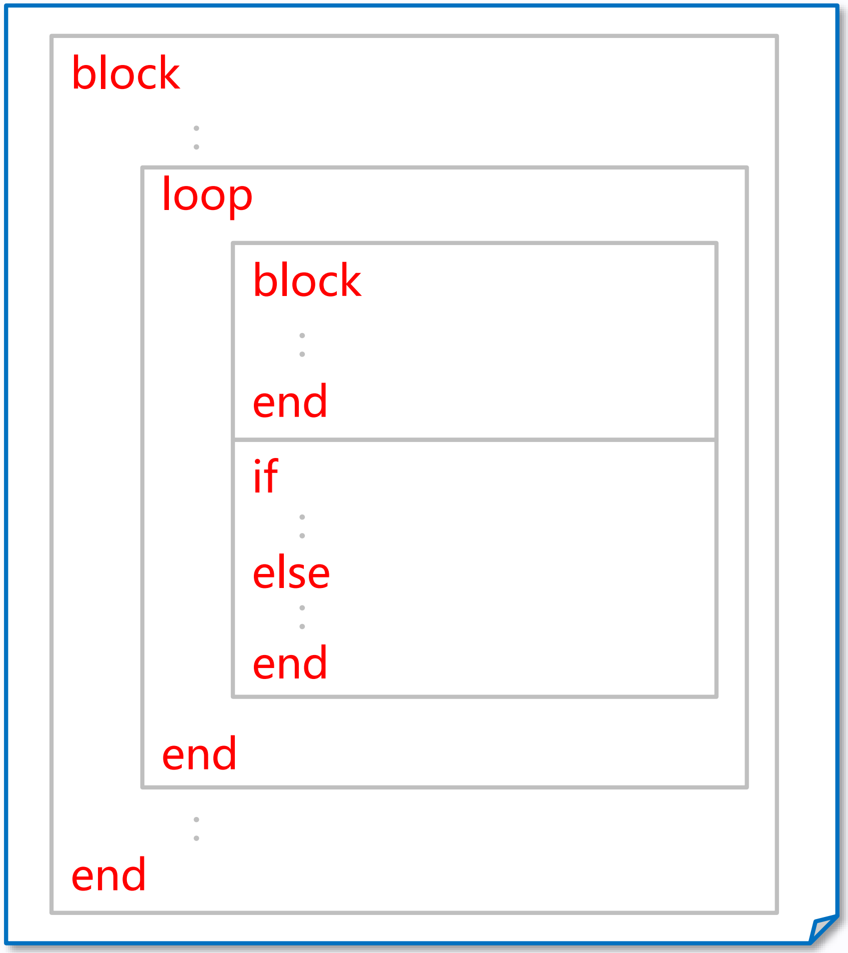


The memory.grow instruction grows memory by a given delta.
The memory.grow instruction operate in units of page size (64KiB).

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

Control instructions

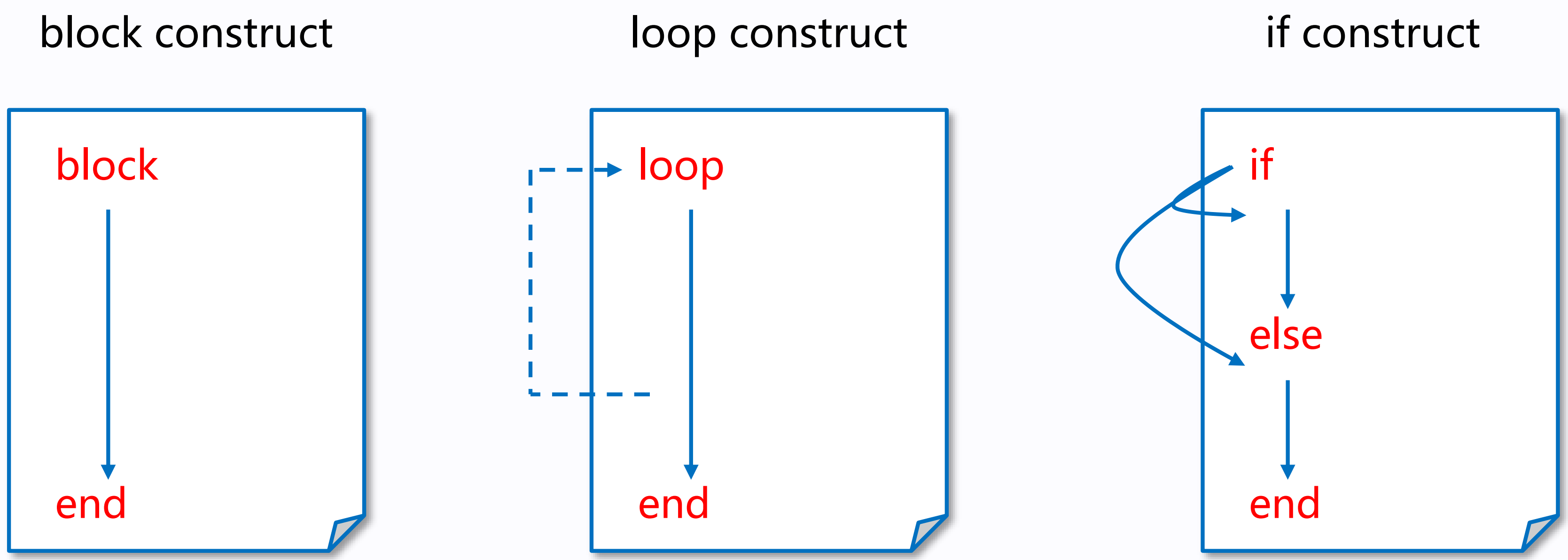
Control flow is structured



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], [4]

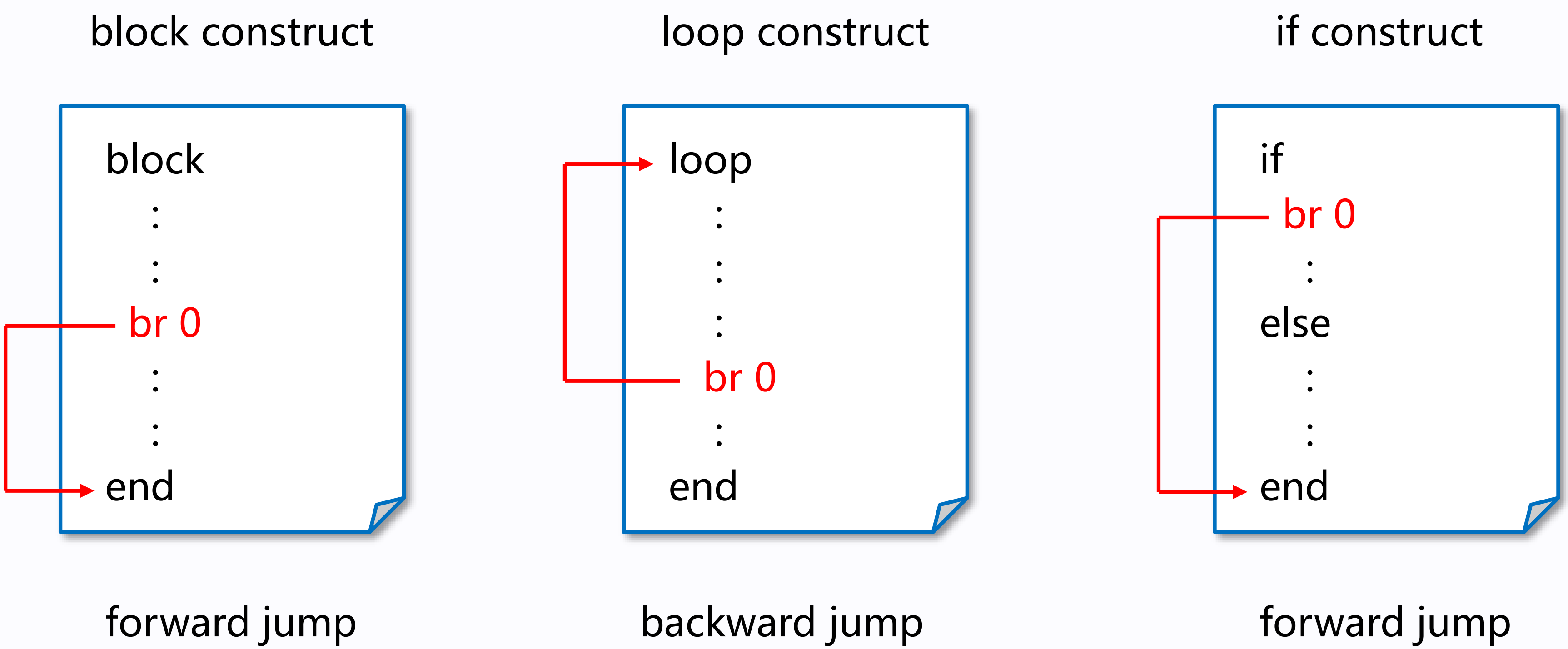
Structured control instructions



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

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

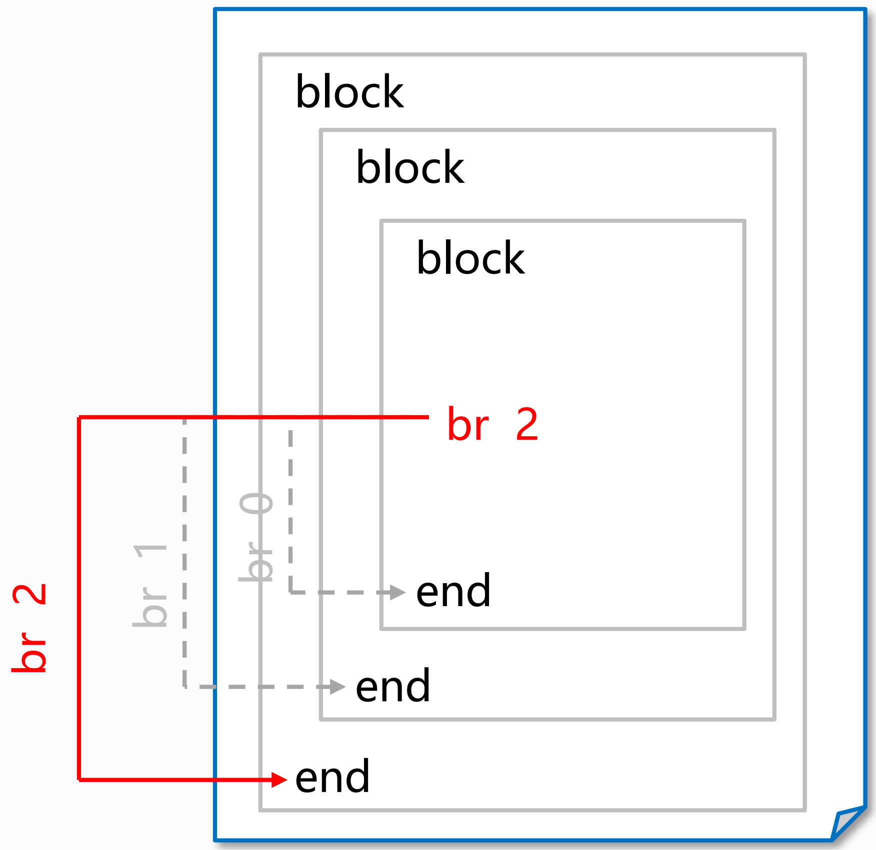
Control constructs and branch instruction



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], [4]

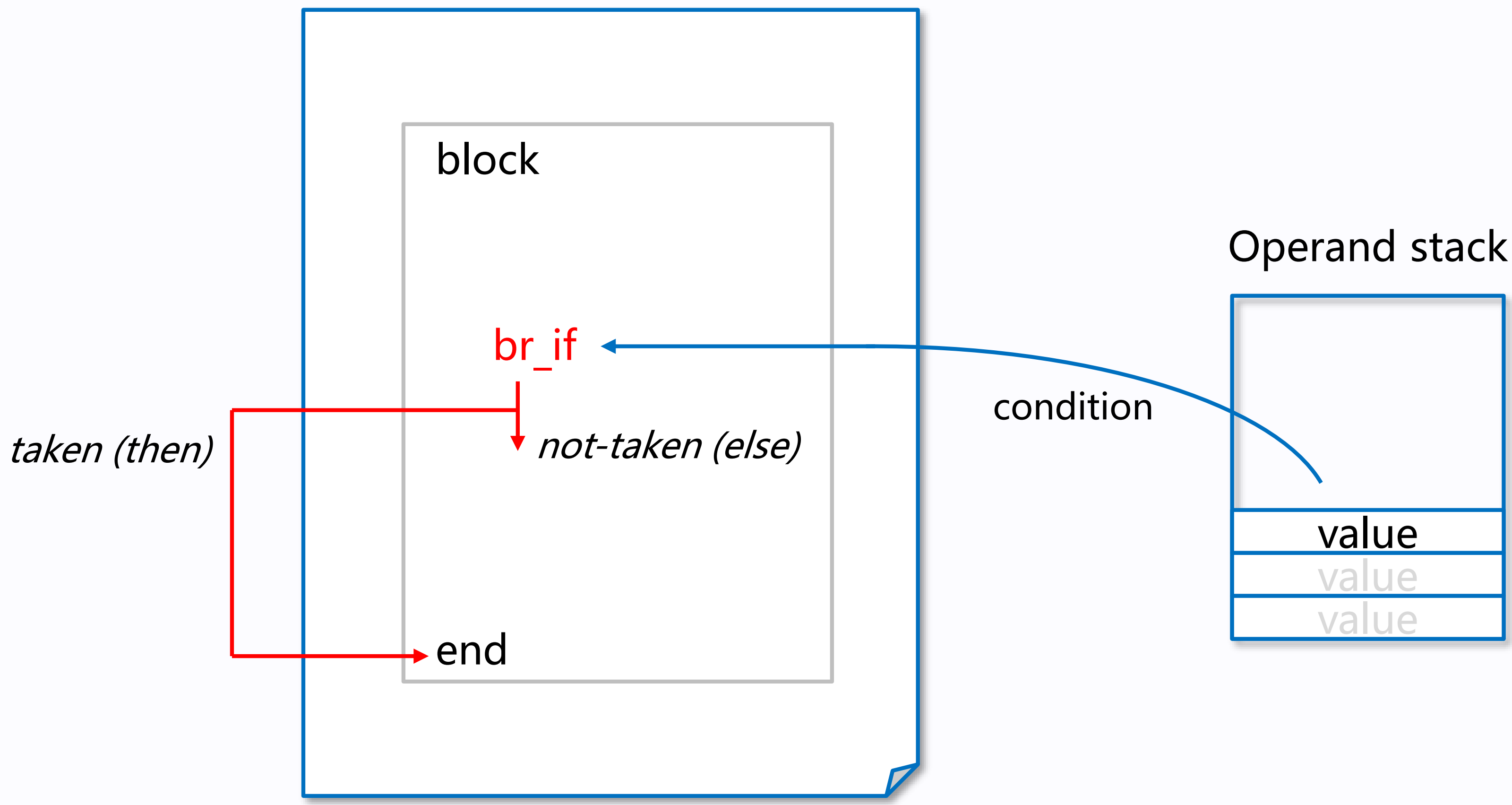
Nested constructs and branch instruction



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], [4]

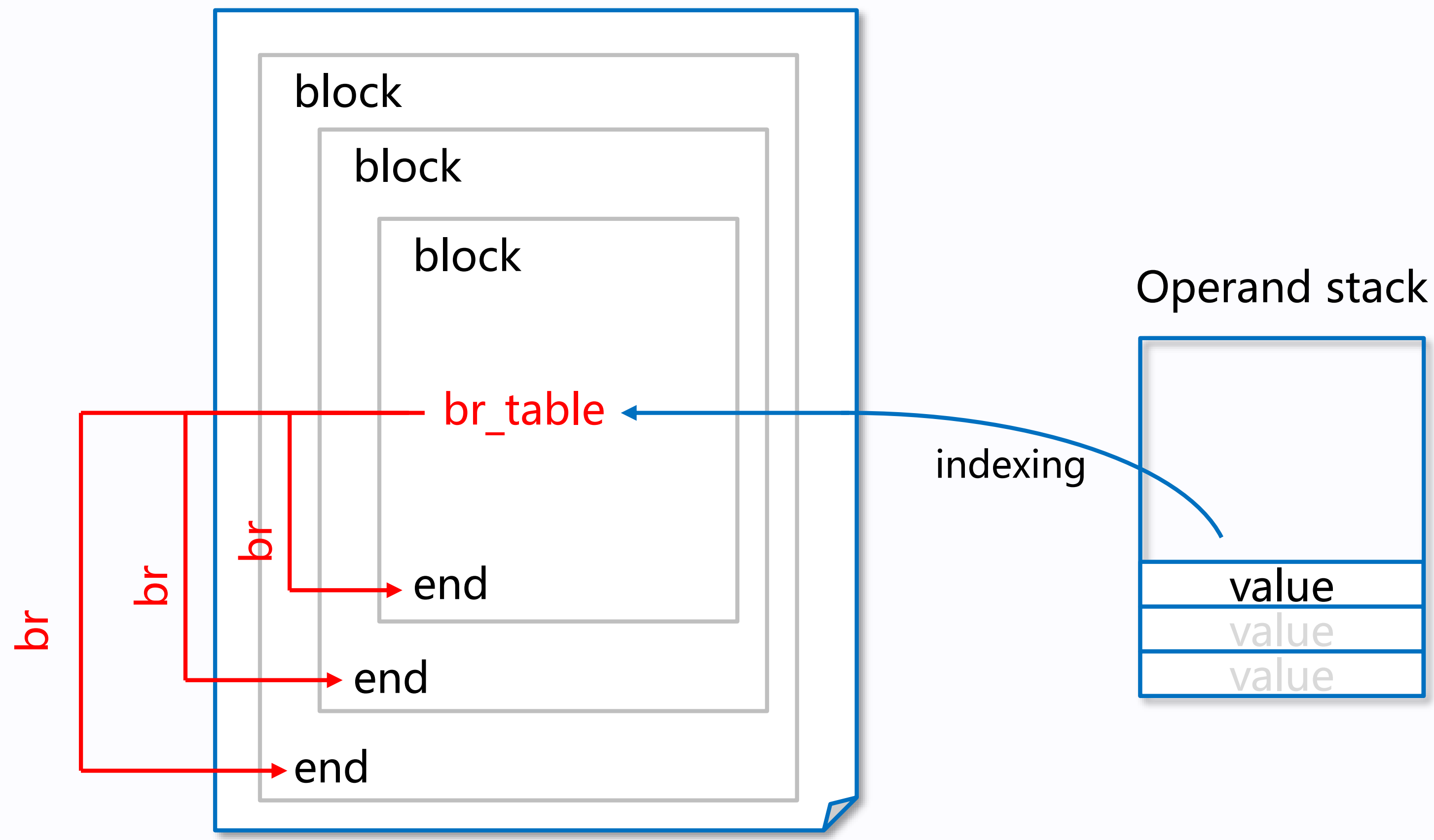
Conditional branch instruction



The br_if instruction performs a conditional branch.

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

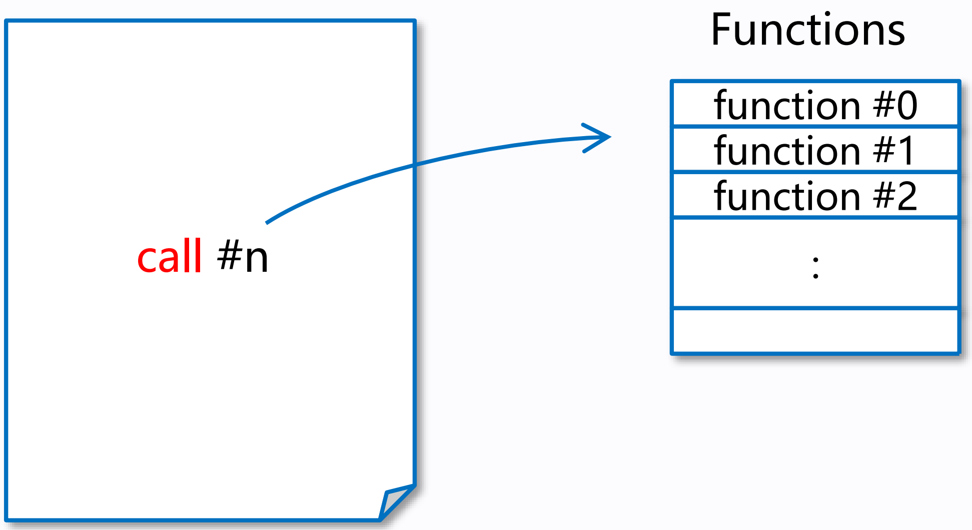
Table branch instruction



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

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

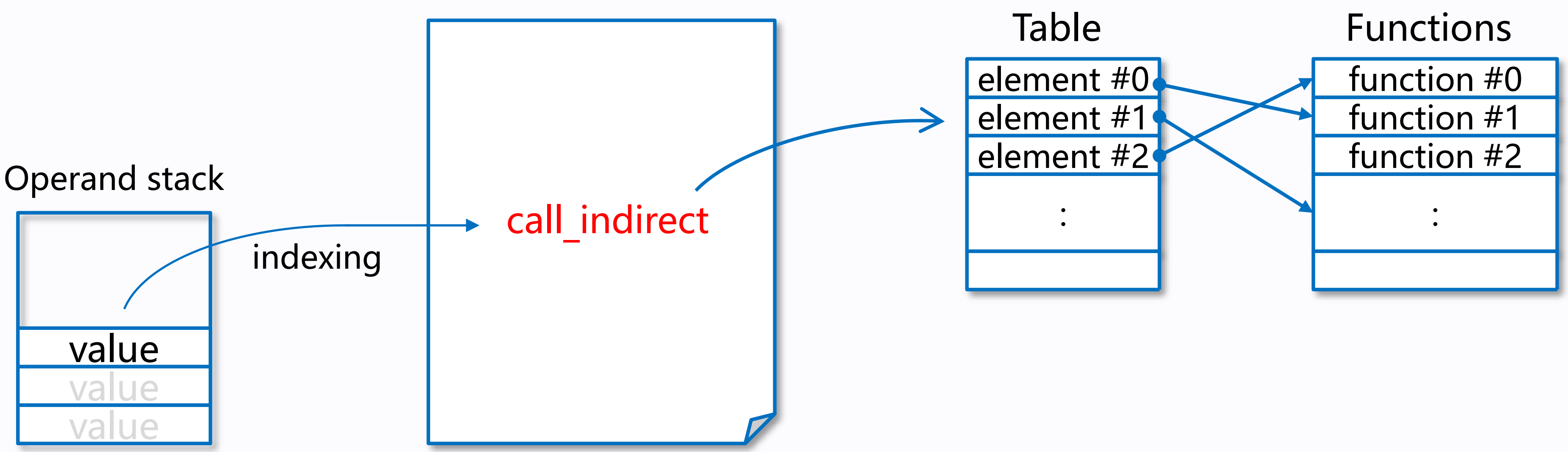
Call instruction



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], [4]

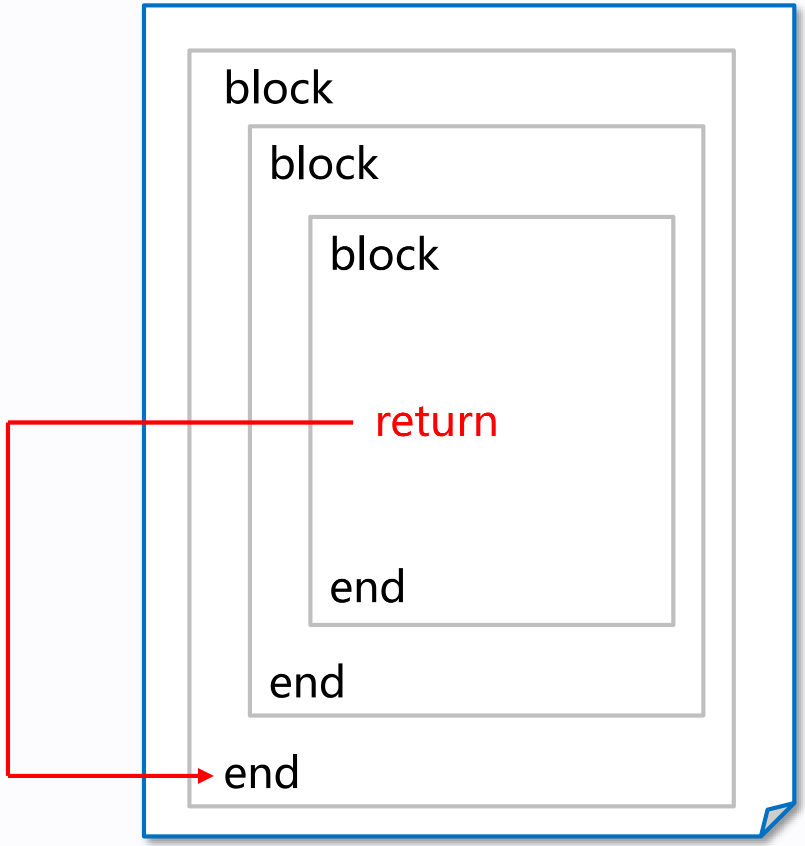
Indirect call instruction



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

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

Return instruction

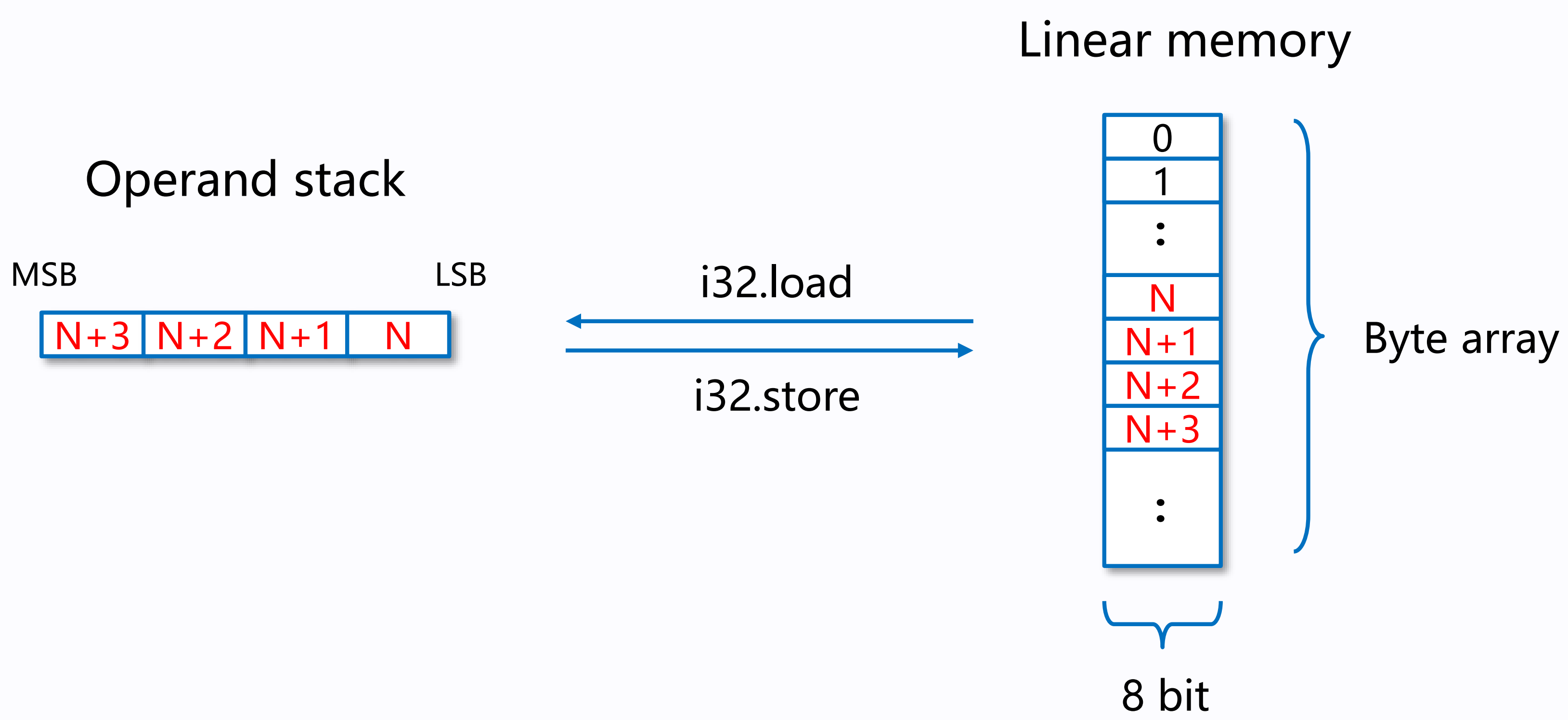


The return instruction is an unconditional branch to the outermost block, which implicitly is the body of the current function.

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

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], [4]

Appendix A

Appendix A

Semantics

Validation and execution semantics

The semantics is derived from the following article:
"Bringing the Web up to Speed with WebAssembly" [2]

Validation semantics: typing rules

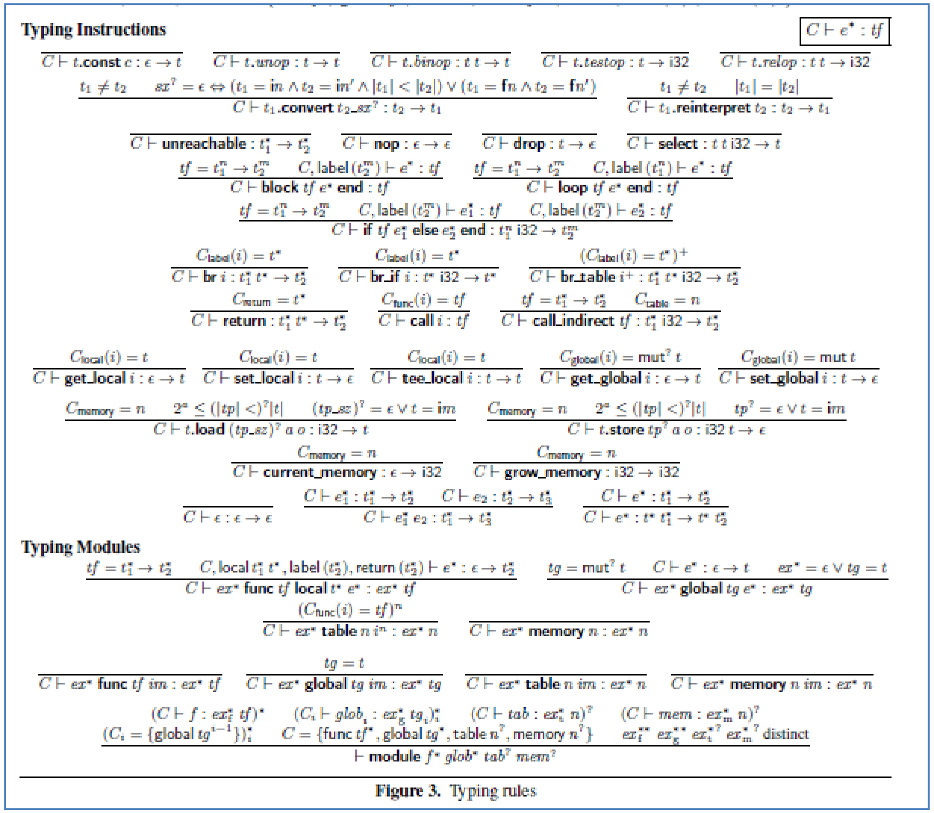


Figure 3. Typing rules

Execution semantics: reduction rules

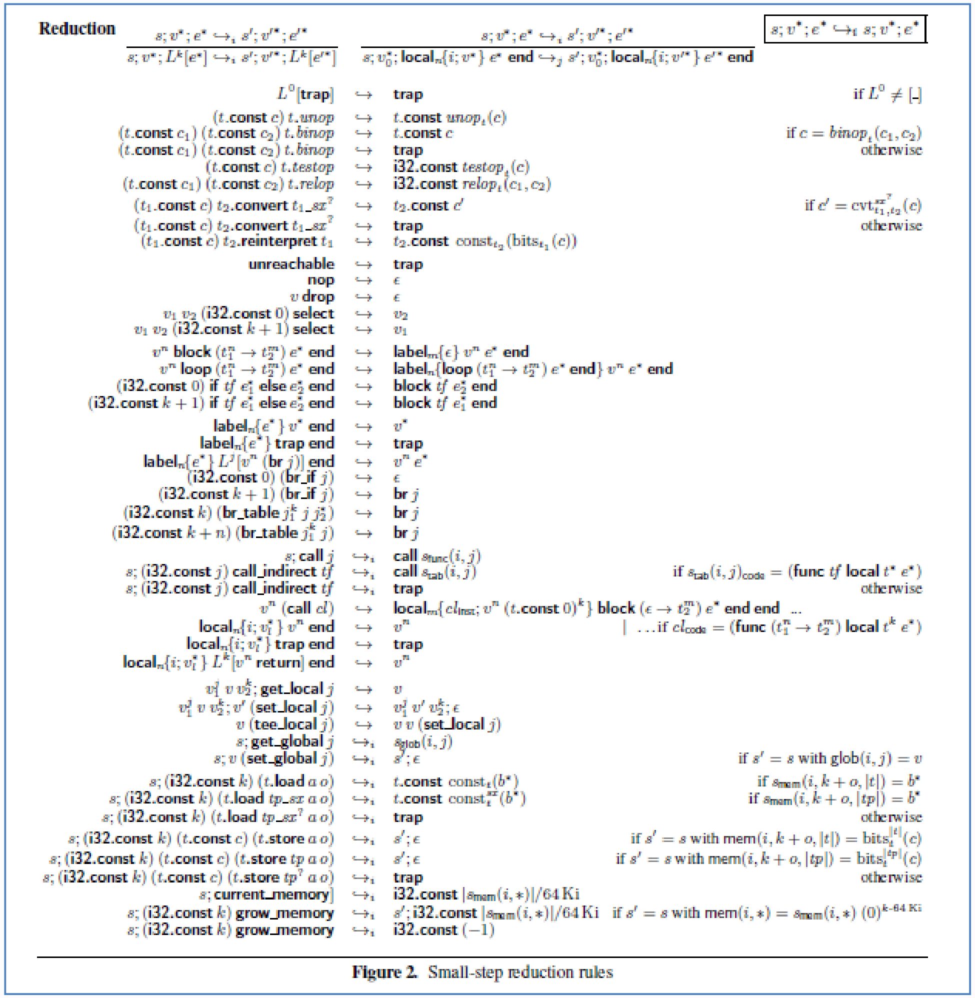


Figure 2. Small-step reduction rules

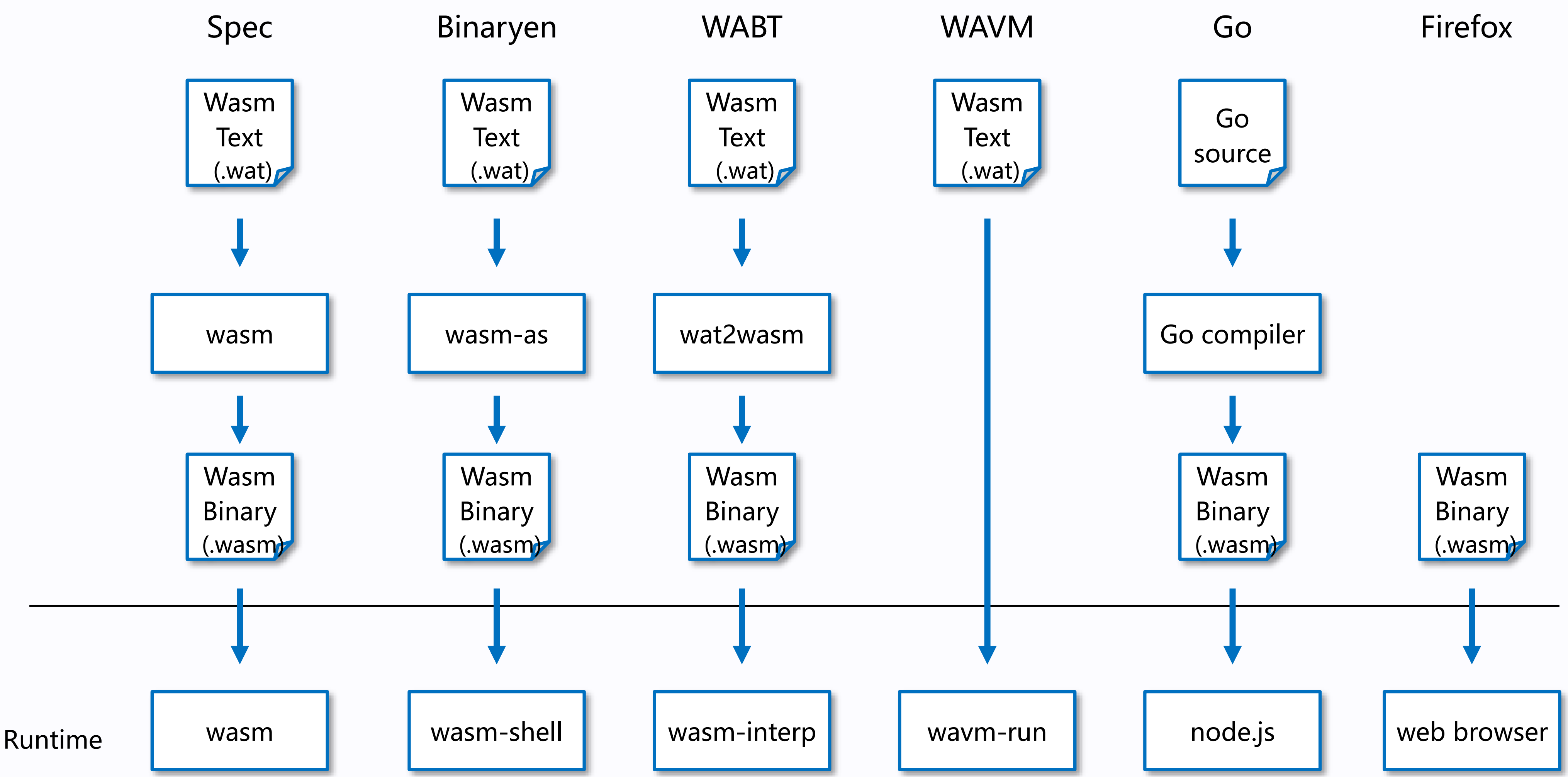
References : [2], [1] Ch.3, Ch.4, Ch.7

Appendix B

Appendix B

Implementations

Implementations



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

Reference 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 01 :: vs' ->
        vs', [Plain (Block (ts, es2)) @@ e.at]
```

References : [C1]

Interpreter : WABT

https://github.com/WebAssembly/wabt
[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

https://github.com/WAVM/WAVM
[Lib/LLVMJIT/EmitFunction.cpp]

```
 // Decode the WebAssembly opcodes and emit LLVM IR for them.
        OperatorDecoderStream decoder(functionDef.code);
        UnreachableOpVisitor unreachableOpVisitor(*this);
        OperatorPrinter operatorPrinter(irModule, functionDef);
        Uptr opIndex = 0;
        while(decoder && controlStack.size())
        {
                irBuilder.SetCurrentDebugLocation(
                        llvm::DILocation::get(llvmContext, (unsigned int)opIndex++,
0, diFunction));
                if(ENABLE_LOGGING)
{ logOperator(decoder.decodeOpWithoutConsume(operatorPrinter)); }

                if(controlStack.back().isReachable) { decoder.decodeOp(*this); }
                else
                {
                        decoder.decodeOp(unreachableOpVisitor);
                }
        wavmAssert(irBuilder.GetInsertBlock() == returnBlock);

        if(EMIT_ENTER_EXIT_HOOKS)
```

References : [C4]

Web browser : Firefox

https://github.com/mozilla/gecko-dev
[js/src/wasm/WasmBaselineCompile.cpp]

```
switch (op.b0) {
         case uint16_t(Op::End):
           if (!emitEnd()) {
               return false;
           }

           if (iter_.controlStackEmpty()) {
               if (!deadCode_) {
                   doReturn(funcType().ret(), PopStack(false));
               }
               return iter_.readFunctionEnd(iter_.end());
           }
           NEXT();

         // Control opcodes
         case uint16_t(Op::Nop):
           CHECK_NEXT(iter_.readNop());
         case uint16_t(Op::Drop):
           CHECK_NEXT(emitDrop());
         case uint16_t(Op::Block):
           CHECK_NEXT(emitBlock());
         case uint16_t(Op::Loop):
```

References : [C5]

Appendix B

CLI development utilities

Assemble

Assemble Wasm text format (.wat) to Wasm binary format (.wasm) :

Binaryen :

```
$ wasm-as sample.wat
```

WABT :

```
$ wat2wasm sample.wat
```

```
$ wat2wasm -v sample.wat
```

Spec :

```
$ wasm -d sample.wat
```

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

Disassemble

Disassemble Wasm binary format (.wasm) to Wasm text format (.wat)

Binaryen :

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

WABT :

```
$ wasm2wat sample.wasm
```

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

Spec :

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

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

Desugar

Desugar Wasm text format (.wat) to Wasm text format (.wat)

WABT :

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

References : [C3]

Dump information

Dump Wasm binary format (.wasm) information :

WABT :

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

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

Spec :

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

References : [C1], [C3]

Run

Run Wasm binary format (.wasm) and Wasm text format (.wat) :

WABT : Run Wasm binary format with trace

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

WAVM : Run Wasm text format

```
$ wavm-run sample.wat
```

Spec : Run Wasm binary format

```
$ wasm sample.wasm  -e '(invoke “XXX”)'
```

References : [C1], [C3], [c4]

REPL

REPL (Read-Eval-Print-Loop) :

Spec :

```
$ wasm  -
```

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

References : [C1]

Appendix B

Test suite

Test suite and Wasm text format 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
```

Note: `.wast` extension means command-script and Wasm text format.

References : [C1]

Appendix B

Desugar examples

Desugar example

Text format

syntactic sugar

```
(func (result i32)
  (i32.add
    (i32.const 1)
    (i32.const 2)))
```

Text format

core syntax

```
(func (result i32)
  i32.const 1
  i32.const 2
  i32.add)
```

References : [7], [1] Ch.6, Ch.2, Ch.4

Desugar example

Text format

syntactic sugar

```
(func (result i32)
  (i32.add
    (i32.const 1)
    (i32.mul
      (i32.const 2)
      (i32.const 3))))
```

Text format

core syntax

```
(func (result i32)
  i32.const 1
  i32.const 2
  i32.const 3
  i32.mul
  i32.add)
```

References : [7], [1] Ch.6, Ch.2, Ch.4

Desugar example

Text format

syntactic sugar

```
(func (result i32)
  (block (result i32)
    (i32.add
      (i32.const 1)
      (i32.const 2))))
```

Text format

core syntax

```
(func (result i32)
  block (result i32)
    i32.const 1
    i32.const 2
    i32.add
  end)
```

References : [7], [1] Ch.6, Ch.2, Ch.4

Desugar example

Text format

syntactic sugar

```
(func
  (block $label_a
    (block $label_b
      br $label_a)))
```

Text format

core syntax

```
(func
  block
    block
      br 1
    end
  end)
```

References : [7], [1] Ch.6, Ch.2, Ch.4

Desugar example

Text format

syntactic sugar

```
(func (result i32)
  (if (result i32) (get_global 0)
    (then (i32.const 1))
    (else (i32.const 2))))
```

Text format

core syntax

```
(func (result i32)
  get_global 0
  if (result i32)
    i32.const 1
  else
    i32.const 2
  end)
```

References : [7], [1] Ch.6, Ch.2, Ch.4

Appendix C

Future

Future directions

* zero-cost exception, threads, SIMD

* tail call, stack switching, coroutines

* garbage collectors

References : [2], [3], [4]

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

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

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

References

[C1] spec: WebAssembly specification, reference interpreter, and test suite.
https://github.com/WebAssembly/spec

[C2] Binaryen: Compiler infrastructure and toolchain library for WebAssembly, in C++
https://github.com/WebAssembly/binaryen

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

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

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