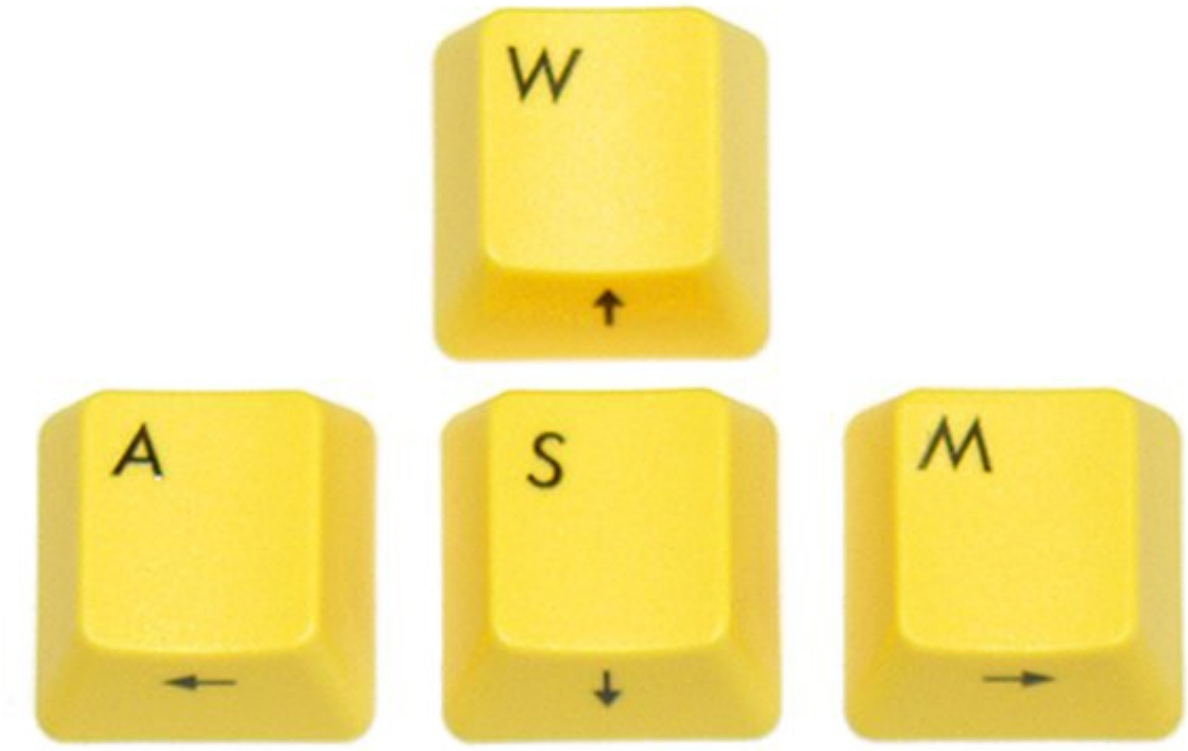

WebAssembly Specification

Release 1.0

WebAssembly Community Group

Mar 13, 2017

Contents

CHAPTER 1

Introduction

1.1 Introduction

WebAssembly (abbreviated Wasm [2]) is a *safe, portable, low-level code format* designed for efficient execution and compact representation. Its main goal is to enable high performance applications on the Web, but it does not make any Web-specific assumptions or provide Web-specific features, so can be employed in other environments as well.

WebAssembly is an open standard developed by a W3C Community Group[1] that includes representatives of all major browser vendors.

This document describes version 1.0 of the core WebAssembly standard. It is intended that it will be superseded by new incremental releases with additional features in the future.

1.1.1 Design Goals

The design goals of WebAssembly are the following:

- Fast, safe, and portable *semantics*:
  - **Fast**: executes with near native code performance, taking advantage of capabilities common to all contemporary hardware.
  - **Safe**: code is validated and executes in a memory-safe [3], sandboxed environment preventing data corruption or security breaches.
  - **Well-defined**: fully and precisely defines valid programs and their behavior in a way that is easy to reason about informally and formally.
  - **Hardware-independent**: can be compiled on all modern architectures, desktop or mobile devices and embedded systems alike.
  - **Language-independent**: does not privilege any particular language, programming model, or object model.
  - **Platform-independent**: can be embedded in browsers, run as a stand-alone VM, or integrated in other environments.
  - **Open**: programs can interoperate with their environment in a simple and universal manner.
- Efficient and portable *representation*:
  - **Compact**: a binary format that is fast to transmit by being smaller than typical text or native code formats.

[2] A contraction of "WebAssembly", not an acronym, hence not using all-caps.

[1] https://www.w3.org/community/webassembly/

[3] No program can break WebAssembly's memory model. Of course, it cannot guarantee that an unsafe language compiling to WebAssembly does not corrupt its own memory layout, e.g. inside WebAssembly's linear memory.

- **Modular**: programs can be split up in smaller parts that can be transmitted, cached, and consumed separately.
- **Efficient**: can be decoded, validated, and compiled in a fast single pass, equally with either just-in-time (JIT) or ahead-of-time (AOT) compilation.
- **Streamable**: allows decoding, validation, and compilation to begin as soon as possible, before all data has been seen.
- **Parallelizable**: allows decoding, validation, and compilation to be split into many independent parallel tasks.
- **Portable**: makes no architectural assumptions that are not broadly supported across modern hardware.

WebAssembly code is also intended to be easy to inspect and debug, especially in environments like web browsers, but such features are beyond the scope of this specification.

1.1.2 Scope

At its core, WebAssembly is a *virtual instruction set architecture (virtual ISA)*. As such, it has many use cases and can be embedded in many different environments. To encompass their variety and enable maximum reuse, the WebAssembly specification is split and layered into several documents.

This document is concerned with the core ISA layer of WebAssembly. It defines the instruction set, binary encoding, validation, and execution semantics. It does not, however, define how WebAssembly programs can interact with a specific environment they execute in, nor how they are invoked from such an environment.

Instead, this specification is complemented by additional documents defining interfaces to specific embedding environments such as the Web. These will each define a WebAssembly *application programming interface (API)* suitable for a given environment.

1.2 Overview

1.2.1 Concepts

WebAssembly encodes a low-level, assembly-like programming language. This language is structured around the following main concepts.

Values WebAssembly provides only four basic *value types*. These are integers and IEEE-754 floating point[4] numbers, each in 32 and 64 bit width. 32 bit integers also serve as Booleans and as memory addresses. The usual operations on these types are available, including the full matrix of conversions between them. There is no distinction between signed and unsigned integer types. Instead, integers are interpreted by respective operations as either unsigned or signed in 2's complement representation.

Instructions The computational model of WebAssembly is based on a *stack machine*. Code consists of sequences of *instructions* that are executed in order. Instructions manipulate values on an implicit *operand stack* [5] and fall into two main categories. Simple instructions perform basic operations on data. They pop arguments from the operand stack and push results back to it. *Control* instructions alter control flow. Control flow is *structured*, meaning it is expressed with well-nested constructs such as blocks, loops, and conditionals. Branches can only target such constructs.

Traps Under some conditions, certain instructions may produce a *trap*, which immediately aborts excecution. Traps cannot be handled by WebAssembly code, but are reported to the outside environment, where they typically can be caught.

[4] http://ieeexplore.ieee.org/document/4610935/

[5] In practice, implementations need not maintain an actual operand stack. Instead, the stack can be viewed as a set of anonymous registers that are implicitly referenced by instructions. The type system ensures that the stack height, and thus any referenced register, is always known statically.

Functions Code is organized into separate *functions*. Each function takes a sequence of values as parameters and returns a sequence of values as results. [6] Functions can call each other, including recursively, resulting in an implicit call stack that cannot be accessed directly. Functions may also declare mutable *local variables* that are usable as virtual registers.

Tables A *table* is an array of opaque values of a particular *element type*. It allows programs to select such values indirectly through a dynamic index operand. Currently, the only available element type is an untyped function reference. Thereby, a program can call functions indirectly through a dynamic index into a table. For example, this allows emulating function pointers with table indices.

Linear Memory A *linear memory* is a contiguous, mutable array of untyped bytes. Such a memory is created with an initial size but can be dynamically grown. 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. A trap occurs if access is not within the bounds of the current memory size.

Modules 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*. Definitions can also be *imported*, specifying a module/name pair and a suitable type. Each definition can optionally be *exported* under one or more names. In addition to definitions, a module can define initialization data for its memory or table that takes the form of *segments* copied to given offsets. It can also define a *start function* that is automatically executed.

Embedder A WebAssembly implementation will typically be *embedded* into a *host* environment. This environment defines how loading of modules is initiated, how imports are provided (including host-side definitions), and how exports can be accessed. However, the details of any particular embedding are beyond the scope of this specification, and will instead be provided by complementary, environment-specific API definitions.

1.2.2 Semantic Phases

Conceptually, the semantics of WebAssembly is divided into three phases. For each part of the language, the specification specifies each of them.

Decoding WebAssembly modules are distributed in a *binary format*. *Decoding* processes that format and converts it into an internal representation of a module. In this specification, this representation is modelled by *abstract syntax*, but a real implementation could compile directly to machine code instead.

Validation 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.

Execution Finally, a valid module can be *executed*. Execution can be further divided into two phases:

Instantiation. An *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, further WebAssembly computations can be initiated by *invoking* an exported function of an instance. Given the required arguments, that executes the respective function and returns its results.

Instantiation and invocation are operations within the embedding environment.

[6] In the current version of WebAssembly, there may be at most one result value.

CHAPTER 2

Structure

2.1 Conventions

WebAssembly is a programming language that does not have a concrete syntax (other than the auxiliary text format). For conciseness, however, its structure is described in the form of an *abstract syntax*. All parts of this specification are defined in terms of this abstract syntax, including the decoding of the *binary format*.

2.1.1 Grammar

The following conventions are adopted in defining grammar rules for abstract syntax.

- Terminal symbols (atoms) are written in sans-serif: $\mathsf{i32}, \mathsf{end}$.
- Nonterminal symbols are written in italic: $\mathit{valtype}, \mathit{instr}$.
- A^n is a sequence of $n \geq 0$ iterations of A.
- A^* is a possibly empty sequence of iterations of A. (This is a shorthand for A^n used where n is not relevant.)
- A^+ is a non-empty sequence of iterations of A. (This is a shorthand for A^n where $n \geq 1$.)
- $A^?$ is an optional occurrence of A. (This is a shorthand for A^n where $n \leq 1$.)

Each non-terminal defines a syntactic class.

2.1.2 Auxiliary Notation

When dealing with syntactic constructs the following notation is also used:

- ϵ denotes the empty sequence.
- $|s|$ denotes the length of a sequence s.
- $s[i]$ denotes the i-th element of a sequence s, starting from 0.

Productions of the following form are interpreted as *records* that map a fixed set of fields field_i to values x_i, respectively:

$$r \ ::= \ \{\mathsf{field}_1 \ x_1, \mathsf{field}_2 \ x_2, \dots\}$$

The following notation is adopted for manipulating such records:

- $r.\mathsf{field}$ denotes the field component of r.
- $r, \mathsf{field}\ s$ denotes the same record as r but with the sequence s appended to its field component.

2.2 Values

2.2.1 Bytes

The simplest form of value are raw uninterpreted *bytes*. In the abstract they are represented as hexadecimal literals.

$$byte \quad ::= \quad \text{0x00} \mid \ldots \mid \text{0xFF}$$

Conventions

- The meta variable b range over bytes.
- The meta function $\text{byte}(n)$ denotes the byte representing the natural number $n < 256$.

2.2.2 Integers

Different classes of *integers* with different value ranges are distinguished by their *size* and their *signedness*.

$$\begin{array}{lcl} uint_N & ::= & 0 \mid 1 \mid \ldots \mid 2^N{-}1 \\ sint_N & ::= & -2^{N-1} \mid \ldots \mid -1 \mid 0 \mid 1 \mid \ldots \mid 2^{N-1}{-}1 \\ int_N & ::= & uint_N \mid sint_N \end{array}$$

The latter class defines *uninterpreted* integers, whose signedness interpretation can vary depending on context. A 2's complement interpretation is assumed for out-of-range values. That is, semantically, when interpreted as unsigned, negative values $-n$ convert to $2^N - n$, and when interpreted as signed, positive values $n \geq 2^{N-1}$ convert to $n - 2^N$.

Conventions

- The meta variables m, n, i, j, k range over unsigned integers.
- Numbers may be denoted by simple arithmetics.

2.2.3 Floating-point Numbers

Floating-point numbers are represented as binary values according to the IEEE-754[7] standard.

$$float_N \quad ::= \quad byte^{N/8}$$

The two possible sizes N are 32 and 64.

Conventions

- The meta variable z ranges over floating point values.

2.2.4 Vectors

Vectors are bracketed sequences of the form $[A^n]$ (or $[A^*]$), where the A-s can either be values or complex constructions.

$$vec(A) \quad ::= \quad [A^*]$$

[7] http://ieeexplore.ieee.org/document/4610935/

Conventions

- $|v|$ denotes the length of a vector v.
- $v[i]$ denotes the i-th element of a vector v, starting from 0.

2.2.5 Names

Names are vectors of bytes interpreted as character strings.

$$name \quad ::= \quad vec(byte)$$

Todo

Unicode?

2.3 Types

2.3.1 Value Types

Value types classify the individual values that WebAssembly code can compute with and the values that a variable accepts.

$$valtype \quad ::= \quad \mathsf{i32} \mid \mathsf{i64} \mid \mathsf{f32} \mid \mathsf{f64}$$

The types int_{32} and int_{64} classify 32 and 64 bit integers, respectively. Integers are not inherently signed or unsigned, their interpretation is determined by individual operations.

The types $float_{32}$ and $float_{64}$ classify 32 and 64 bit floating points, respectively. They correspond to single and double precision floating point types as defined by the IEEE-754[8] standard

Conventions

- The meta variable t ranges over value types where clear from context.

2.3.2 Result Types

Result types classify the results of functions or blocks, which is a sequence of values.

$$resulttype \quad ::= \quad valtype^?$$

Note: In the current version of WebAssembly, at most one value is allowed as a result. However, this may be generalized to sequences of values in future versions.

---

[8] http://ieeexplore.ieee.org/document/4610935/

2.3.3 Function Types

Function types classify the signature of functions, mapping a sequence of parameters to a sequence of results.

$$\begin{array}{lcl} functype & ::= & valtype^* \to resulttype \end{array}$$

2.3.4 Memory Types

Memory types classify linear memories and their size range.

$$\begin{array}{lcl} memtype & ::= & limits \\ limits & ::= & \{\mathsf{min}\ uint_{32}, \mathsf{max}\ uint_{32}{}^?\} \end{array}$$

The limits constrain the minimum and optionally the maximum size of a table. If no maximum is given, the table can grow to any size. Both values are given in units of page size.

2.3.5 Table Types

Table types classify tables over elements of *element types* within a given size range.

$$\begin{array}{lcl} tabletype & ::= & limits\ elemtype \\ elemtype & ::= & \mathsf{anyfunc} \end{array}$$

Like memories, tables are constrained by limits for their minimum and optionally the maximum size. These sizes are given in numbers of entries.

The element type anyfunc is the infinite union of all *function types*. A table of that type thus contains references to functions of heterogeneous type.

Note: In future versions of WebAssembly, additional element types may be introduced.

2.3.6 Global Types

Global types classify global variables, which hold a value and can either be mutable or immutable.

$$\begin{array}{lcl} globaltype & ::= & \mathsf{mut}^?\ valtype \end{array}$$

2.3.7 External Types

External types classify imports and exports and their respective types.

$$\begin{array}{lcl} externtype & ::= & \mathsf{func}\ functype \mid \\ & & \mathsf{table}\ tabletype \mid \\ & & \mathsf{mem}\ memtype \mid \\ & & \mathsf{global}\ globaltype \end{array}$$

2.4 Modules

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*.

$$
\begin{array}{rcll}
module & ::= & \{ & \mathsf{types}\ vec(functype), \\
& & & \mathsf{funcs}\ vec(func), \\
& & & \mathsf{tables}\ vec(table), \\
& & & \mathsf{mems}\ vec(mem), \\
& & & \mathsf{globals}\ vec(global), \\
& & & \mathsf{elem}\ vec(elemseg), \\
& & & \mathsf{data}\ vec(dataseg), \\
& & & \mathsf{start}\ funcidx^{?}, \\
& & & \mathsf{imports}\ vec(import), \\
& & & \mathsf{exports}\ vec(export) \quad \}
\end{array}
$$

Each of the vectors – and thus the entire module – may be empty.

2.4.1 Indices

Definitions are referenced with zero-based *indices*. Each class of definition has its own *index space*, as distinguished by the following classes.

$$
\begin{array}{lcl}
typeidx & ::= & uint_{32} \\
funcidx & ::= & uint_{32} \\
tableidx & ::= & uint_{32} \\
memidx & ::= & uint_{32} \\
globalidx & ::= & uint_{32} \\
localidx & ::= & uint_{32} \\
labelidx & ::= & uint_{32}
\end{array}
$$

The index space for functions, tables, memories and globals includes respective imports declared in the same module. The indices of these imports precede the indices of other definitions in the same index space.

The index space for locals is only accessible inside a function and includes the parameters and local variables of that function, which precede the other locals.

Label indices reference block instructions inside an instruction sequence.

Conventions

- The meta variable l ranges over label indices.
- The meta variable x ranges over indices in any of the other index spaces.

2.4.2 Expressions

Function bodies, initialization values for *globals* and offsets of *element* or *data* segments are given as expressions, which are sequences of *instructions* terminated by an end marker.

$$
expr \quad ::= \quad instr^{*}\ \mathsf{end}
$$

In some places, validation restricts expressions to be *constant*, which limits the set of allowable insructions.

2.4.3 Types

The types component of a module defines a vector of *function types*.

All function types used in a module must be defined in the type section. They are referenced by *type indices*.

Note: Future versions of WebAssembly may add additional forms of type definitions.

2.4.4 Functions

The funcs component of a module defines a vector of *functions* with the following structure:

$$func \quad ::= \quad \{\text{type } typeidx, \text{locals } vec(valtype), \text{body } expr\}$$

The type of a function declares its signature by reference to a *type* defined in the module. The parameters of the function are referenced through 0-based *local indices* in the function's body.

The locals declare a vector of mutable local variables and their types. These variables are referenced through *local indices* in the function's body. The index of the first local is the smallest index not referencing a parameter.

The body is an *instruction* sequence that must evaluate to a stack matching the function type's *result type*.

Functions are referenced through *function indices*, starting with the smallest index not referencing a function *import*.

2.4.5 Tables

The tables component of a module defines a vector of *tables* described by their *table type*:

$$table \quad ::= \quad \{\text{type } tabletype\}$$

A table is a vector of opaque values of a particular table *element type*. The min size in the *limits* of the table type of a definition specifies the initial size of that table, while its max, if present, restricts the size to which it can grow later.

Tables can be initialized through *element segments*.

Tables are referenced through *table indices*, starting with the smallest index not referencing a table *import*. Most constructs implicitly reference table index 0.

Note: In the current version of WebAssembly, at most one table may be defined or imported in a single module, and *all* constructs implicitly reference this table 0. This restriction may be lifted in future versions.

2.4.6 Memories

The mems component of a module defines a vector of *linear memories* (or *memories* for short) as described by their *memory type*:

$$mem \quad ::= \quad \{\text{type } memtype\}$$

A memory is a vector of raw uninterpreted bytes. The min size in the *limits* of the memory type of a definition specifies the initial size of that memory, while its max, if present, restricts the size to which it can grow later. Both are in units of page size.

Memories can be initialized through *data segments*.

Memories are referenced through *memory indices*, starting with the smallest index not referencing a memory *import*. Most constructs implicitly reference memory index 0.

Note: In the current version of WebAssembly, at most one memory may be defined or imported in a single module, and *all* constructs implicitly reference this memory 0. This restriction may be lifted in future versions.

2.4.7 Globals

The globals component of a module defines a vector of *global variables* (or *globals* for short):

$$global \quad ::= \quad \{\text{type } globaltype, \text{init } expr\}$$

Each global stores a single value of the given *global type*. Its type also specifies whether a global is immutable or mutable. Moreover, each global is initialized with an init value given by a constant initializer *expression*.

Globals are referenced through *global indices*, starting with the smallest index not referencing a global *import*.

2.4.8 Element Segments

The initial contents of a table is uninitialized. The elem component of a module defines a vector of *element segments* that initialize a subrange of a table at a given offset from a static vector of elements.

$$elemseg \quad ::= \quad \{\text{table } tableidx, \text{offset } expr, \text{init } vec(funcidx)\}$$

The offset is given by a constant *expression*.

Note: In the current version of WebAssembly, at most one table is allowed in a module. Consequently, the only valid $tableidx$ is 0.

2.4.9 Data Segments

The initial contents of a memory are zero bytes. The data component of a module defines a vector of *data segments* that initialize a range of memory at a given offset with a static vector of bytes.

$$dataseg \quad ::= \quad \{\text{mem } memidx, \text{offset } expr, \text{init } vec(byte)\}$$

The offset is given by a constant *expression*.

Note: In the current version of WebAssembly, at most one memory is allowed in a module. Consequently, the only valid $memidx$ is 0.

2.4.10 Start Function

The start component of a module denotes the function index of an optional *start function* that is automatically invoked when the module is instantiated, after tables and memories have been initialized.

2.4.11 Exports

The exports component of a module defines a set of *exports* that become accessible to the host environment once the module has been instantiated.

$$\begin{array}{lll} export & ::= & \{\text{name } name, \text{desc } exportdesc\} \\ exportdesc & ::= & \text{func } funcidx \mid \\ & & \text{table } tableidx \mid \\ & & \text{mem } memidx \mid \\ & & \text{global } globalidx \end{array}$$

Each export is identified by a unique *name*. Exportable definitions are *functions*, *tables*, *memories*, and *globals*, which are referenced through a respective descriptor.

Note: In the current version of WebAssembly, only *immutable* globals may be exported.

2.4.12 Imports

The imports component of a module defines a set of *imports* that are required for instantiation.

$$
\begin{array}{lll}
import & ::= & \{\text{module } name, \text{name } name, \text{desc } importdesc\} \\
importdesc & ::= & \text{func } typeidx \mid \\
 & & \text{table } tabletype \mid \\
 & & \text{mem } memtype \mid \\
 & & \text{global } globaltype
\end{array}
$$

Each import is identified by a two-level *name* space, consisting of a module name and a unique name for an entity within that module. Importable definitions are *functions*, *tables*, *memories*, and *globals*. Each import is specified by a descriptor with a respective type that a definition provided during instantiation is required to match.

Every import defines an index in the respective index space. In each index space, the indices of imports go before the first index of any definition contained in the module itself.

Note: In the current version of WebAssembly, only *immutable* globals may be imported.

2.5 Instructions

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

Note: In the current version of WebAssembly, at most one result value can be returned by an instruction. This restriction may be lifted in future versions.

In addition to dynamic operands from the stack, some instructions also have static *immediate* arguments which are part of the instruction itself, typically indices or type annotations.

Some instructions are *structured* in that they bracket nested sequences of instructions.

The following sections group instructions into a number of different categories.

2.5.1 Numeric Instructions

Numeric instructions provide basic operations over numeric values of specific type. These operations closely match respective operations available in hardware.

$$
\begin{array}{lll}
nn, mm & ::= & 32 \mid 64 \\
sx & ::= & \mathsf{u} \mid \mathsf{s} \\
instr & ::= & \mathsf{i}nn\mathsf{.const}\ int_{nn} \mid \mathsf{f}nn\mathsf{.const}\ float_{nn} \mid \\
 & & \mathsf{i}nn\mathsf{.eqz} \mid \\
 & & \mathsf{i}nn\mathsf{.eq} \mid \mathsf{i}nn\mathsf{.ne} \mid \mathsf{i}nn\mathsf{.lt\_}sx \mid \mathsf{i}nn\mathsf{.gt\_}sx \mid \mathsf{i}nn\mathsf{.le\_}sx \mid \mathsf{i}nn\mathsf{.ge\_}sx \mid \\
 & & \mathsf{f}nn\mathsf{.eq} \mid \mathsf{f}nn\mathsf{.ne} \mid \mathsf{f}nn\mathsf{.lt} \mid \mathsf{f}nn\mathsf{.gt} \mid \mathsf{f}nn\mathsf{.le} \mid \mathsf{f}nn\mathsf{.ge} \mid \\
 & & \mathsf{i}nn\mathsf{.clz} \mid \mathsf{i}nn\mathsf{.ctz} \mid \mathsf{i}nn\mathsf{.popcnt} \mid \\
 & & \mathsf{i}nn\mathsf{.add} \mid \mathsf{i}nn\mathsf{.sub} \mid \mathsf{i}nn\mathsf{.mul} \mid \mathsf{i}nn\mathsf{.div\_}sx \mid \mathsf{i}nn\mathsf{.rem\_}sx \mid \\
 & & \mathsf{i}nn\mathsf{.and} \mid \mathsf{i}nn\mathsf{.or} \mid \mathsf{i}nn\mathsf{.xor} \mid \\
 & & \mathsf{i}nn\mathsf{.shl} \mid \mathsf{i}nn\mathsf{.shr\_}sx \mid \mathsf{i}nn\mathsf{.rotl} \mid \mathsf{i}nn\mathsf{.rotr} \mid \\
 & & \mathsf{f}nn\mathsf{.abs} \mid \mathsf{f}nn\mathsf{.neg} \mid \mathsf{f}nn\mathsf{.sqrt} \mid \\
 & & \mathsf{f}nn\mathsf{.ceil} \mid \mathsf{f}nn\mathsf{.floor} \mid \mathsf{f}nn\mathsf{.trunc} \mid \mathsf{f}nn\mathsf{.nearest} \mid \\
 & & \mathsf{f}nn\mathsf{.add} \mid \mathsf{f}nn\mathsf{.sub} \mid \mathsf{f}nn\mathsf{.mul} \mid \mathsf{f}nn\mathsf{.div} \mid \\
 & & \mathsf{f}nn\mathsf{.min} \mid \mathsf{f}nn\mathsf{.max} \mid \mathsf{f}nn\mathsf{.copysign} \mid \\
 & & \mathsf{i32.wrap/i64} \mid \mathsf{i64.extend\_}sx\mathsf{/i32} \mid \mathsf{i}nn\mathsf{.trunc\_}sx\mathsf{/f}mm \mid \\
 & & \mathsf{f32.demote/f64} \mid \mathsf{f64.promote/f32} \mid \mathsf{f}nn\mathsf{.convert\_}sx\mathsf{/i}mm \mid \\
 & & \mathsf{i}nn\mathsf{.reinterpret/f}nn \mid \mathsf{f}nn\mathsf{.reinterpret/i}nn
\end{array}
$$

Numeric instructions are divided by *value type*. For each type, several structural subcategories can be distinguished:

- *Constants*: return a static constant.
- *Unary Operators*: consume one operand and produce one result of the respective type.
- *Binary Operators*: consume two operands and produce one result of the respective type.
- *Tests*: consume one operand of the respective type and produce a Boolean results.
- *Comparisons*: consume two operands of the respective type and produce a Boolean results.
- *Conversions*: consume a value of one type and produce a result of another (the source type of the conversion is the one after the $/$).

Several integer instructions come in two flavours, where a signedness annotation sx distinguishes whether the operands are to be interpreted as unsigned or signed integers. For all other instructions, the sign interpretation is irrelevant in a 2's complement representation.

2.5.2 Parametric Instructions

Instructions in this group can operate on operands of any *value type*.

$$
\begin{array}{lll}
instr & ::= & \ldots \mid \\
 & & \mathsf{drop} \mid \\
 & & \mathsf{select}
\end{array}
$$

The drop operator simply throws away its operand.

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

2.5.3 Variable Instructions

Variable instructions are concerned with the access to local or *global* variables.

$$
\begin{array}{rcl}
\mathit{instr} & ::= & \dots \mid \\
& & \mathsf{get\_local}\ \mathit{localidx} \mid \\
& & \mathsf{set\_local}\ \mathit{localidx} \mid \\
& & \mathsf{tee\_local}\ \mathit{localidx} \mid \\
& & \mathsf{get\_global}\ \mathit{globalidx} \mid \\
& & \mathsf{set\_global}\ \mathit{globalidx} \mid
\end{array}
$$

These instructions get or set the values of variables, respectively. The tee_local instruction is like set_local but also returns its argument.

2.5.4 Memory Instructions

Instructions in this group are concerned with linear memory.

$$
\begin{array}{rcl}
\mathit{memop} & ::= & \{\mathsf{offset}\ \mathit{uint}_{32}, \mathsf{align}\ \mathit{uint}_{32}\} \\
\mathit{instr} & ::= & \dots \mid \\
& & \mathsf{i}nn\mathsf{.load}\ \mathit{memop} \mid \mathsf{f}nn\mathsf{.load}\ \mathit{memop} \mid \\
& & \mathsf{i}nn\mathsf{.store}\ \mathit{memop} \mid \mathsf{f}nn\mathsf{.store}\ \mathit{memop} \mid \\
& & \mathsf{i}nn\mathsf{.load8\_}sx\ \mathit{memop} \mid \mathsf{i}nn\mathsf{.load16\_}sx\ \mathit{memop} \mid \mathsf{i64.load32\_}sx\ \mathit{memop} \mid \\
& & \mathsf{i}nn\mathsf{.store8}\ \mathit{memop} \mid \mathsf{i}nn\mathsf{.store16}\ \mathit{memop} \mid \mathsf{i64.store32}\ \mathit{memop} \mid \\
& & \mathsf{current\_memory} \mid \\
& & \mathsf{grow\_memory}
\end{array}
$$

Memory is accessed with load and store instructions for the different *value types*. They all take a *memory immediate* memop that contains an address *offset* and an *alignment* hint. Integer loads and stores can optionally specify a *storage size* that is smaller than the width of the respective value type. In the case of loads, a sign extension mode sx is required to select appropriate behavior.

The static address offset is added to the dynamic address operand, yielding a 33 bit *effective address* that is the zero-based index at which the memory is accessed. All values are read and written in little endian[9] byte order. A trap results if any of the accessed memory bytes lies outside the address range implied by the memory's current size.

Note: Future version of WebAssembly might provide memory instructions with 64 bit address ranges.

The current_memory instruction returns the current size of a memory. The grow_memory instruction grows memory by a given delta and returns the previous size, or -1 if enough memory cannot be allocated. Both instructions operate in units of page size.

Note: In the current version of WebAssembly, all memory instructions implicitly operate on *memory index* 0. This restriction may be lifted in future versions.

The precise semantics of memory instructions is described in the Instruction section.

[9] https://en.wikipedia.org/wiki/Endianness#Little-endian

2.5.5 Control Instructions

Instructions in this group affect the flow of control.

$$
\begin{array}{rcl}
\mathit{instr} & ::= & \dots \mid \\
& & \mathsf{nop} \mid \\
& & \mathsf{unreachable} \mid \\
& & \mathsf{block}\ \mathit{resulttype}\ \mathit{instr}^*\ \mathsf{end} \mid \\
& & \mathsf{loop}\ \mathit{resulttype}\ \mathit{instr}^*\ \mathsf{end} \mid \\
& & \mathsf{if}\ \mathit{resulttype}\ \mathit{instr}^*\ \mathsf{else}\ \mathit{instr}^*\ \mathsf{end} \mid \\
& & \mathsf{br}\ \mathit{labelidx} \mid \\
& & \mathsf{br\_if}\ \mathit{labelidx} \mid \\
& & \mathsf{br\_table}\ \mathit{vec}(\mathit{labelidx})\ \mathit{labelidx} \mid \\
& & \mathsf{return} \mid \\
& & \mathsf{call}\ \mathit{funcidx} \mid \\
& & \mathsf{call\_indirect}\ \mathit{typeidx}
\end{array}
$$

The nop instruction does nothing.

The unreachable instruction causes an unconditional trap.

The block, loop and if instructions are *structured* instructions. They bracket nested sequences of instructions, called *blocks*, terminated with, or separated by, end or else pseudo-instructions. As the grammar prescribes, these control instructures must be well-nested. A structured instruction can produce a value as described by the annotated *result type*.

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 furhter out. Consequently, labels can only be referenced from *within* the associated structured control instruction. This also implies that branches can only be directed outwards, "breaking" from the block of the control construct they target. The exact effect depends on that control construct. In case of block or if it is a *forward jump*, resuming execution after the matching end. In case of loop it is a *backward jump* to the beginning of the loop.

Note: This enforces *structured control flow*. Intuitively, a branch targeting a block or if behaves like a break statement, while a branch targeting a loop behaves like a continue statement.

Branch instructions come in several flavors: br performs an unconditional branch, br_if performs a conditional branch, and br_table performs an indirect branch through a dense table that is an immediate to the instruction, or to a default target if the operand is out of bounds. The return instruction is a shortcut for an unconditional branch to the outermost block, which implicitly is the body of the current function. A branch *unwinds* the operand stack up to the height where the targeted control instruction was entered. However, forward branches that target a control instruction with a non-empty result type consume a matching operand first and push it back on the operand stack after unwinding, as a result for the terminated instruction.

The call instruction invokes another function, consuming the necessary arguments from the stack and returning the result values of the call. The call_indirect instruction calls a function indirectly through an operand indexing into a *table*. Since tables may contain function elements of heterogeneous type anyfunc, the callee is dynamically checked against the function type indexed by the instruction's immediate, and the call aborted with a trap if it does not match.

Note: In the current version of WebAssembly, call_indirect implicitly operates on *table index* 0. This restriction may be lifted in future versions.

CHAPTER 3

Validation

CHAPTER 4

Instantiation

CHAPTER 5

Execution

CHAPTER 6

Instructions

6.1 Overview

Todo

Describe

6.2 Control Instructions

6.2.1 Block Instructions

6.2.2 Branch Instructions

6.2.3 Call Instructions

6.2.4 Miscellaneous Control Instructions

6.3 Variable Instructions

6.4 Parametric Instructions

6.5 Numeric Instructions

6.5.1 Numeric Instructions

6.5.2 Integer Test Instructions

6.5.3 Integer Comparison Instructions

6.5.4 Floating Point Comparison Instructions

6.5.5 Unary Integer Instructions

6.5.6 Unary Floating Point Instructions

6.5.7 Binary Integer Instructions

6.5.8 Binary Floating Point Instructions

6.5.9 Conversion Instructions

6.5.10 Reinterpretation Instructions

6.6 Memory Instructions

6.6.1 Load Instructions

6.6.2 Memory Instructions

6.6.3 Store Instructions

6.7 Instruction Sequences

Todo

Describe

CHAPTER 7

Binary Format

CHAPTER 8

Appendix: Formal Properties

Todo

Describe and sketch proof (progress, preservation, uniqueness)

CHAPTER 9

Appendix: Validation Algorithm

Todo

Describe algorithm, state correctness properties (soundness, completeness)

CHAPTER 10

Appendix: Text Format

Todo

Describe

CHAPTER 11

Appendix: Name Section

Todo

Describe

U

V