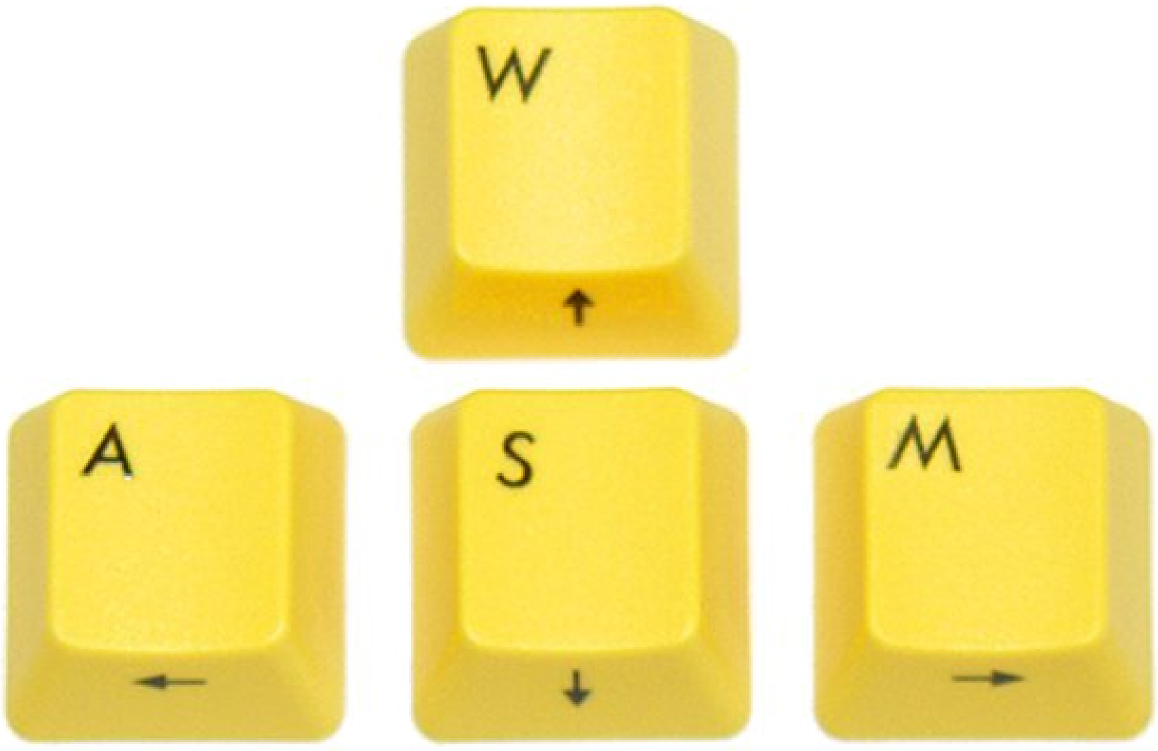

WebAssembly Specification

Release 1.0

WebAssembly Community Group

Feb 20, 2017

Contents

CHAPTER 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 It is intended that it will be superseded by new incremental releases with additional features in the future.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 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.
  - **Modular**: programs can be split up in smaller parts that can be transmitted, cached, and consumed separately.

[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.

- **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.

1.2 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 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.

CHAPTER 2

Concepts

2.1 Overview

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 numbers, each in 32 and 64 bit width. 32 bit integers also function as Booleans and as memory addresses. The usual basic 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, different signedness is handled by different operations, selecting either unsigned or 2's complement signed interpretation.
- **Instructions**. The computational model of WebAssembly is based on a *stack machine*. Code consists of sequences of *instructions* that are executed in order. They manipulate values on an implicit *operand stack*. [4] Instructions fall into two main categories. *Simple* instructions perform basic operations on data. They pop argument values from the operand stack and push result values back to it. *Control* instructions alter control flow. They are either sources or targets of *branches*. Control flow is *structured*, i.e., it is expressed with well-nested constructs such as blocks, loops, and conditionals. Branches can only target such constructs.
- **Traps**. Some instructions may produce a *trap* under some conditions, which immediately aborts excecution. Traps cannot be handled by WebAssembly code, but are reported to the outside environment, where they typically can be caught.
- **Functions**. Code is organized into separate *functions*. Each function takes a sequence of values as parameters and returns zero or one values as a result. Functions can call each other, including recursively. The call stack is implicit and cannot be accessed directly. Nor is the address of a function accessible. 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, code can perform an indirect call to a function indexed through 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) in little endian format. Integer access 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. It can also define mutable or immutable *global variables*. Each definition can optionally be *exported* under one or more names. Definitions can also be *imported*, specifying a module/name pair and a suitable type.

[4] 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.

To use it, a module must first be *instantiated*. An *instance* is the dynamic representation of a module, complete with its own state and execution stack. Instantiation requires providing definitions for all imports, which may be exports from previously created instances. WebAssembly computations can be initiated by invoking an exported function of an instance. 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 upon instantiation. It can also define a *start function* that is automatically executed after instantiation.

- **Validation**. To be instantiated, a module must be *valid*. Validation performs *type-checking* of functions and their bodies, as well as checking of other well-formedness conditions, e.g., that a module has no duplicate export names.
- **Embedder**. A WebAssembly implementation will typically be *embedded* into a *host* environment. Instantiation of modules and invocation of exports is controlled from this environment. An embedder can also provide external means to create and initialize memories or tables imported by a module. Furthermore, it may allow to supply functions from the host environment as imports to WebAssembly modules. However, form and semantics of such functionality are outside the scope of this specification, and will instead be provided by complementary, environment-specific API definitions.

2.2 Conventions

Todo

Describe

Testing math: $C \vdash e^* : t^*$ defined for C by

$$\frac{C \vdash e^* : t_1^* \rightarrow t_2^*}{C \vdash \mathsf{func}\ tf\ \mathsf{local}\ t^*\ e^*}$$

and bla.

2.3 Binary Format

Todo

Describe

2.4 Validation

Todo

Describe

2.5 Execution

Todo

Describe

CHAPTER 3

Modules

3.1 Overview

Todo

Describe general structure of modules

3.2 Instantiation

Todo

Describe the semantics of module instantiation

3.3 Invocation

Todo

Describe the semantics of invoking exports

3.4 Type Section

Todo

Describe

3.5 Import Section

Todo

Describe

3.6 Function Section

Todo

Describe

3.7 Table Section

Todo

Describe

3.8 Memory Section

Todo

Describe

3.9 Global Section

Todo

Describe

3.10 Export Section

Todo

Describe

3.11 Start Section

Todo

Describe

3.12 Element Section

Todo

Describe

3.13 Code Section

Todo

Describe

3.14 Data Section

Todo

Describe

3.15 User Section

Todo

Describe

3.16 Summary: Abstract Grammar of Modules

Todo

Grammar summary

3.17 Summary: Validation of Modules

Todo

Summary of typing rules

3.18 Summary: Execution of Modules

3.18.1 Instantiation

Todo

Summary of instantiation rules

3.18.2 Invocation

Todo

Summary of invocation rules

3.19 Summary: Binary Encoding of Modules

Todo

Encoding summary

CHAPTER 4

Instructions

4.1 Overview

Todo

Describe

4.2 Control Instructions

4.2.1 Block Instructions

4.2.2 Branch Instructions

4.2.3 Call Instructions

4.2.4 Miscellaneous Control Instructions

4.3 Variable Instructions

4.4 Parametric Instructions

4.5 Numeric Instructions

4.5.1 Numeric Instructions

4.5.2 Integer Test Instructions

4.5.3 Integer Comparison Instructions

4.5.4 Floating Point Comparison Instructions

4.5.5 Unary Integer Instructions

4.5.6 Unary Floating Point Instructions

4.5.7 Binary Integer Instructions

4.5.8 Binary Floating Point Instructions

4.5.9 Conversion Instructions

4.5.10 Reinterpretation Instructions

4.6 Memory Instructions

4.6.1 Load Instructions

4.6.2 Memory Instructions

4.6.3 Store Instructions

4.7 Instruction Sequences

Todo

Describe

4.8 Summary: Abstract Grammar of Instructions

Todo

Grammar summary

4.9 Summary: Validation of Instructions

Todo

Summary of typing rules

4.10 Summary: Execution of Instructions

Todo

Summary of reduction rules

4.11 Summary: Binary Encoding of Instructions

Todo

Encoding summary

CHAPTER 5

Appendix: Formal Properties

Todo

Describe and sketch proof (progress, preservation, uniqueness)

CHAPTER 6

Appendix: Validation Algorithm

Todo

Describe algorithm, state correctness properties (soundness, completeness)