WASM: untrusted at any speed

Curtis Millar

Monday, 9th September, 2019

What is WASM?

WebAssembly (WASM) is a specification of a stack machine.

- Works in a wide variety of systems

What is WASM?

WebAssembly (WASM) is a specification of a stack machine.

- Works in a wide variety of systems
  - Embeds into other languages or systems via a runtime

What is WASM?

WebAssembly (WASM) is a specification of a stack machine.

- Works in a wide variety of systems
  - Embeds into other languages or systems via a runtime
  - Allows a single compiled module to be used in any context

What is WASM?

WebAssembly (WASM) is a specification of a stack machine.

- Works in a wide variety of systems
  - Embeds into other languages or systems via a runtime
  - Allows a single compiled module to be used in any context
  - Does not need to be rebuilt for specific architectures or platforms

What is WASM?

WebAssembly (WASM) is a specification of a stack machine.

- Works in a wide variety of systems
  - Embeds into other languages or systems via a runtime
  - Allows a single compiled module to be used in any context
  - Does not need to be rebuilt for specific architectures or platforms
- High performance

What is WASM?

WebAssembly (WASM) is a specification of a stack machine.

- Works in a wide variety of systems
  - Embeds into other languages or systems via a runtime
  - Allows a single compiled module to be used in any context
  - Does not need to be rebuilt for specific architectures or platforms
- High performance
  - Compiles to native machine code at runtime

What is WASM?

WebAssembly (WASM) is a specification of a stack machine.

- Works in a wide variety of systems
  - Embeds into other languages or systems via a runtime
  - Allows a single compiled module to be used in any context
  - Does not need to be rebuilt for specific architectures or platforms
- High performance
  - Compiles to native machine code at runtime
  - Provides low-overhead and portable foreign-function interface

What is WASM?

WebAssembly (WASM) is a specification of a stack machine.

- Works in a wide variety of systems
  - Embeds into other languages or systems via a runtime
  - Allows a single compiled module to be used in any context
  - Does not need to be rebuilt for specific architectures or platforms
- High performance
  - Compiles to native machine code at runtime
  - Provides low-overhead and portable foreign-function interface
- Minimal trust requirements

What is WASM?

WebAssembly (WASM) is a specification of a stack machine.

- Works in a wide variety of systems
  - Embeds into other languages or systems via a runtime
  - Allows a single compiled module to be used in any context
  - Does not need to be rebuilt for specific architectures or platforms
- High performance
  - Compiles to native machine code at runtime
  - Provides low-overhead and portable foreign-function interface
- Minimal trust requirements
  - Guarantees a number of safety and security properties

What does WASM look like?

Rust source

```rust
#[no_mangle]
pub extern fn add(a: u64, b: u64) -> u64 {
    a + b
}
```

Compiled WAT (WebAssembly Text)

```wat
(module
  (type (;1;) (func (param i64 i64) (result i64)))
  (func $add (type 1) (param i64 i64) (result i64)
    local.get 1
    local.get 0
    i64.add)
  (export "add" (func $add)))
```

WASM data, types, & references

- Four basic data types: `i32`, `i64`, `f32`, & `f64`

WASM data, types, & references

- Four basic data types: `i32`, `i64`, `f32`, & `f64`
- Everything is bundled into a 'module'

WASM data, types, & references

- Four basic data types: `i32`, `i64`, `f32`, & `f64`
- Everything is bundled into a 'module'
  - Directly refrenced functions

WASM data, types, & references

- Four basic data types: `i32`, `i64`, `f32`, & `f64`
- Everything is bundled into a 'module'
  - Directly refrenced functions
  - Indexable globals region (for immutable single values)

WASM data, types, & references

- Four basic data types: `i32`, `i64`, `f32`, & `f64`
- Everything is bundled into a 'module'
  - Directly refrenced functions
  - Indexable globals region (for immutable single values)
  - Indexable linear memory (for mutable, structured, and shared data)

WASM data, types, & references

- Four basic data types: `i32`, `i64`, `f32`, & `f64`
- Everything is bundled into a 'module'
  - Directly refrenced functions
  - Indexable globals region (for immutable single values)
  - Indexable linear memory (for mutable, structured, and shared data)
  - Indexable function tables (for external and indirect function calls)

WASM data, types, & references

- Four basic data types: `i32`, `i64`, `f32`, & `f64`
- Everything is bundled into a 'module'
  - Directly refrenced functions
  - Indexable globals region (for immutable single values)
  - Indexable linear memory (for mutable, structured, and shared data)
  - Indexable function tables (for external and indirect function calls)
- Instructions operate on a stack,

WASM data, types, & references

- Four basic data types: `i32`, `i64`, `f32`, & `f64`
- Everything is bundled into a 'module'
  - Directly refrenced functions
  - Indexable globals region (for immutable single values)
  - Indexable linear memory (for mutable, structured, and shared data)
  - Indexable function tables (for external and indirect function calls)
- Instructions operate on a stack,
  - Variables pased as operands to functions and instructions

WASM data, types, & references

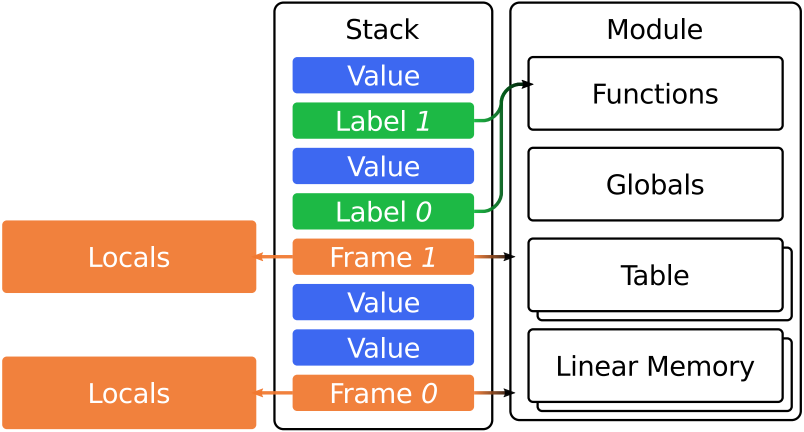
- Four basic data types: `i32`, `i64`, `f32`, & `f64`
- Everything is bundled into a 'module'
  - Directly refrenced functions
  - Indexable globals region (for immutable single values)
  - Indexable linear memory (for mutable, structured, and shared data)
  - Indexable function tables (for external and indirect function calls)
- Instructions operate on a stack,
  - Variables pased as operands to functions and instructions
  - Function frames & locals

WASM data, types, & references

- Four basic data types: `i32`, `i64`, `f32`, & `f64`
- Everything is bundled into a 'module'
  - Directly refrenced functions
  - Indexable globals region (for immutable single values)
  - Indexable linear memory (for mutable, structured, and shared data)
  - Indexable function tables (for external and indirect function calls)
- Instructions operate on a stack,
  - Variables pased as operands to functions and instructions
  - Function frames & locals
  - Block scope

WASM data, types, & references

- Four basic data types: `i32`, `i64`, `f32`, & `f64`
- Everything is bundled into a 'module'
  - Directly refrenced functions
  - Indexable globals region (for immutable single values)
  - Indexable linear memory (for mutable, structured, and shared data)
  - Indexable function tables (for external and indirect function calls)
- Instructions operate on a stack,
  - Variables pased as operands to functions and instructions
  - Function frames & locals
  - Block scope
- Runtime passes data via the linear memories and functions via tables


Stack
Value
Label 1
Value
Label 0
Frame 1
Value
Value
Frame 0
Locals
Locals
Module
Functions
Globals
Table
Linear Memory

Portability

- Designed to be ahead-of-time compiled (similar to Java or .NET)

Portability

- Designed to be ahead-of-time compiled (similar to Java or .NET)
- Can alternatively be interpreted (but that is slow)

Portability

- Designed to be ahead-of-time compiled (similar to Java or .NET)
- Can alternatively be interpreted (but that is slow)
- Requires runtime embedding

Portability

- Designed to be ahead-of-time compiled (similar to Java or .NET)
- Can alternatively be interpreted (but that is slow)
- Requires runtime embedding
  - Compile the module into native machine code

Portability

- Designed to be ahead-of-time compiled (similar to Java or .NET)
- Can alternatively be interpreted (but that is slow)
- Requires runtime embedding
  - Compile the module into native machine code
  - Implement FFI into compiled machine code

Portability

- Designed to be ahead-of-time compiled (similar to Java or .NET)
- Can alternatively be interpreted (but that is slow)
- Requires runtime embedding
  - Compile the module into native machine code
  - Implement FFI into compiled machine code
  - Manage memory for passed data

Portability

- Designed to be ahead-of-time compiled (similar to Java or .NET)
- Can alternatively be interpreted (but that is slow)
- Requires runtime embedding
  - Compile the module into native machine code
  - Implement FFI into compiled machine code
  - Manage memory for passed data
- Write in any source language (e.g. Rust, C, Go, or TypeScript)

(Almost) native speed

- Compiles down to similar machine code as if directly from the language

(Almost) native speed

- Compiles down to similar machine code as if directly from the language
- Currently does not support some optimisations available in native compilation (e.g. SIMD)

(Almost) native speed

- Compiles down to similar machine code as if directly from the language
- Currently does not support some optimisations available in native compilation (e.g. SIMD)
- Can load multiple modules into the same address space/runtime

(Almost) native speed

- Compiles down to similar machine code as if directly from the language
- Currently does not support some optimisations available in native compilation (e.g. SIMD)
- Can load multiple modules into the same address space/runtime
  - Low switching overhead

(Almost) native speed

- Compiles down to similar machine code as if directly from the language
- Currently does not support some optimisations available in native compilation (e.g. SIMD)
- Can load multiple modules into the same address space/runtime
  - Low switching overhead
  - Better hardware utilisation

(Almost) native speed

- Compiles down to similar machine code as if directly from the language
- Currently does not support some optimisations available in native compilation (e.g. SIMD)
- Can load multiple modules into the same address space/runtime
  - Low switching overhead
  - Better hardware utilisation
- Used like libraries or lightweight processes

Security & Trust

- Compliant runtime compilers must ensure bounds checking on table and linear memory access

Security & Trust

- Compliant runtime compilers must ensure bounds checking on table and linear memory access
- Code cannot access data outside of linear memory

Security & Trust

- Compliant runtime compilers must ensure bounds checking on table and linear memory access
- Code cannot access data outside of linear memory
- Code cannot access raw pointers from tables (can only call into functions with code that adheres to the interface type, even for indirect calls)

Security & Trust

- Compliant runtime compilers must ensure bounds checking on table and linear memory access
- Code cannot access data outside of linear memory
- Code cannot access raw pointers from tables (can only call into functions with code that adheres to the interface type, even for indirect calls)
- Cannot call methods not explicitly provided to it

Security & Trust

- Compliant runtime compilers must ensure bounds checking on table and linear memory access
- Code cannot access data outside of linear memory
- Code cannot access raw pointers from tables (can only call into functions with code that adheres to the interface type, even for indirect calls)
- Cannot call methods not explicitly provided to it
- Type-safe (data types, memory stores, function calls)

Security & Trust

- Compliant runtime compilers must ensure bounds checking on table and linear memory access
- Code cannot access data outside of linear memory
- Code cannot access raw pointers from tables (can only call into functions with code that adheres to the interface type, even for indirect calls)
- Cannot call methods not explicitly provided to it
- Type-safe (data types, memory stores, function calls)
- Prevents direct access between modules loaded into the same address space (although timing channels may still exist)

Security & Trust

- Compliant runtime compilers must ensure bounds checking on table and linear memory access
- Code cannot access data outside of linear memory
- Code cannot access raw pointers from tables (can only call into functions with code that adheres to the interface type, even for indirect calls)
- Cannot call methods not explicitly provided to it
- Type-safe (data types, memory stores, function calls)
- Prevents direct access between modules loaded into the same address space (although timing channels may still exist)
- Potential security enhancements (not required, but currently possible):

Security & Trust

- Compliant runtime compilers must ensure bounds checking on table and linear memory access
- Code cannot access data outside of linear memory
- Code cannot access raw pointers from tables (can only call into functions with code that adheres to the interface type, even for indirect calls)
- Cannot call methods not explicitly provided to it
- Type-safe (data types, memory stores, function calls)
- Prevents direct access between modules loaded into the same address space (although timing channels may still exist)
- Potential security enhancements (not required, but currently possible):
  - Runtime code and memory layout randomisation

Mandelbrot (Demo)

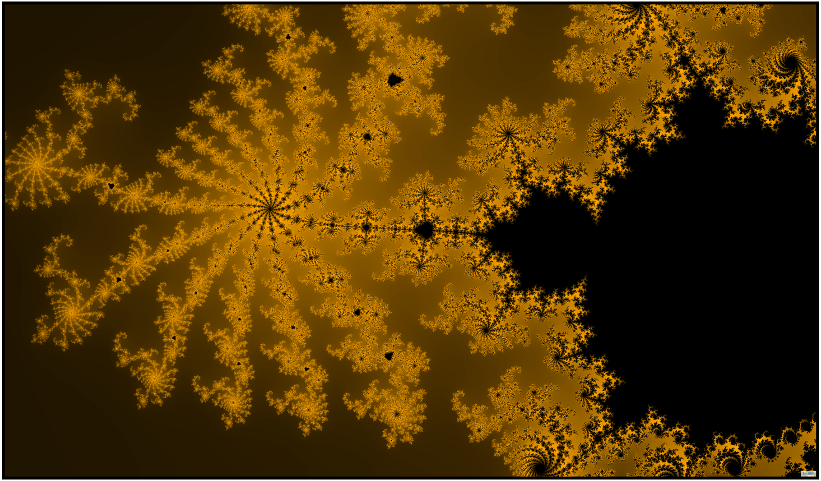


Figure 1: http://almondbread.cse.unsw.edu.au

Theme

```
#[no_mangle]
pub extern
fn color_pixel(steps: u32, re: f64, im: f64) -> u32 {
    let level = steps as u8;
    let color = Color {
        red: level,
        green: level,
        blue: level,
    };
    color.into()
}

#[no_mangle]
pub extern fn max_steps() -> u32 { 256 }
```

```rust
extern "C" {
    // Canvas
    fn canvas_width() -> u32;
    fn canvas_height() -> u32;
    fn draw_pixel(x: u32, y: u32, color: u32);
    fn paint();

    // Theme
    fn color_pixel(steps: u32, re: f64, im: f64) -> u32;
    fn max_steps() -> u32;

    // Progress bar
    fn progress(progress: f64);
}
```

```rust
#[no_mangle]
pub extern
fn render(center_re: f64, center_im: f64, zoom: u32) {
    let (width, height) = canvas_dimensions();
    let distance = 1f64 / ((1u64 << zoom) as f64);
    for pixel_y in 0u32..height {
        for pixel_x in 0u32..width {
            let re = (pixel_x - width/2) as f64;
            let im = (pixel_y - height/2) as f64;
            let c = Complex::new(
                center_re + re * distance,
                center_im + im * distance,
            );
            draw_steps(pixel_x, pixel_y, c);
        }
    }
    unsafe { paint(); }
}
```

```
fn draw_steps(x: u32, y: u32, c: Complex<f64>) {
    let mut z = Complex::new(0f64, 0f64);
    let mut steps = 0;
    let max_steps = unsafe { max_steps() };
    while z.norm_sqr() < 4.0 && steps < max_steps {
        z = z * z + c;
        steps += 1;
    }
    unsafe {
        draw_pixel(x, y, color_pixel(steps, c.re, c.im));
    }
}
```

- WASM in Python

- WASM in Python
- WASM in Firefox

Where can I use WASM now?

- Major browsers (Chrome, Firefox, Edge, Safari, etc.)

Where can I use WASM now?

- Major browsers (Chrome, Firefox, Edge, Safari, etc.)
- Content Delivery Networks (Cloudflare Workers, AWS Lambda)

Where can I use WASM now?

- Major browsers (Chrome, Firefox, Edge, Safari, etc.)
- Content Delivery Networks (Cloudflare Workers, AWS Lambda)
- Language runtime embeddings (wasmtime[1] & wasmer[2] for Rust, Python, C/C++, Go, PHP, Ruby, Postgres, .NET, R, Swift, & POSIX)

[1] https://github.com/CraneStation/wasmtime
[2] https://wasmer.io/

WASM in the future

- Standard WASM runtime interface (WASI)

WASM in the future

- Standard WASM runtime interface (WASI)
- Interface types (automatic interface generation)

WASM in the future

- Standard WASM runtime interface (WASI)
- Interface types (automatic interface generation)
- Threads, atomic primitives, and safe concurrent data access

WASM in the future

- Standard WASM runtime interface (WASI)
- Interface types (automatic interface generation)
- Threads, atomic primitives, and safe concurrent data access
- Garbage collected data

WASM in the future

- Standard WASM runtime interface (WASI)
- Interface types (automatic interface generation)
- Threads, atomic primitives, and safe concurrent data access
- Garbage collected data
- Reference types (`anyref`)

WASM in the future

- Standard WASM runtime interface (WASI)
- Interface types (automatic interface generation)
- Threads, atomic primitives, and safe concurrent data access
- Garbage collected data
- Reference types (`anyref`)
- Explicit tail call

WASM in the future

- Standard WASM runtime interface (WASI)
- Interface types (automatic interface generation)
- Threads, atomic primitives, and safe concurrent data access
- Garbage collected data
- Reference types (`anyref`)
- Explicit tail call
- Simultaneous Instruction, Multiple Data (SIMD) operations

More references

- Official site: https://webassembly.org/

More references

- Official site: https://webassembly.org/
- Mozilla Hacks blog:
  https://hacks.mozilla.org/category/webassembly/

More references

- Official site: https://webassembly.org/
- Mozilla Hacks blog:
  https://hacks.mozilla.org/category/webassembly/
- Mozilla Developer Network:
  https://developer.mozilla.org/en-US/docs/WebAssembly

More references

- Official site: https://webassembly.org/
- Mozilla Hacks blog: https://hacks.mozilla.org/category/webassembly/
- Mozilla Developer Network: https://developer.mozilla.org/en-US/docs/WebAssembly
- WebAssembly Rocks: http://www.wasmrocks.com/

More references

- Official site: https://webassembly.org/
- Mozilla Hacks blog: https://hacks.mozilla.org/category/webassembly/
- Mozilla Developer Network: https://developer.mozilla.org/en-US/docs/WebAssembly
- WebAssembly Rocks: http://www.wasmrocks.com/
- Even more references: https://github.com/mbasso/awesome-wasm