Extracted from:

Programming WebAssembly with Rust

Unified Development for Web, Mobile, and Embedded Applications

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Programming WebAssembly with Rust

Unified Development for Web, Mobile, and Embedded Applications

Kevin Hoffman
edited by Andrea Stewart

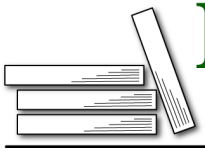
Programming WebAssembly with Rust

Unified Development for Web, Mobile, and Embedded Applications

Kevin Hoffman

The Pragmatic Bookshelf

Raleigh, North Carolina



Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

The team that produced this book includes:

Publisher: Andy Hunt
VP of Operations: Janet Furlow
Managing Editor: Susan Conant
Development Editor: Andrea Stewart
Copy Editor: Jasmine Kwityn
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

ISBN-13: 978-1-68050-636-5
Book version: P1.0—March 2019

For my grandfather—Walter K. MacAdam—
inventor, tinkerer, and IEEE president. He
quietly supported my exposure to computers
and programming throughout my childhood,
often in ways I didn't know until after his death.
I always wanted to grow up to be like him, and
I only wish he could've seen this book.

CHAPTER 1

WebAssembly Fundamentals

With WebAssembly, there is a symbiotic relationship between the compiled WebAssembly binary (called a *module*) and the host responsible for interpreting it. This relationship is at the heart of everything that you can do with this new technology, and understanding where the boundaries are between module and host is key to being able to build effective WebAssembly applications.

WebAssembly can be viewed at two different levels—the raw, foundational level and at the higher level of other programming languages using WebAssembly as a target. Before you can understand and appreciate what languages like Rust are doing when they produce WebAssembly modules, you'll need to know what WebAssembly can do, what it can't, and how to use language-independent tools.

This chapter gets you started at the foundation level, giving you an overview of what WebAssembly is, how it works, and how other features can be built upon this foundation. By the end of this chapter, you'll be able to create and build your own WebAssembly modules using cross-platform language tools and your favorite code editor. While what you learn in these first few chapters may not be things you do on a daily basis, the context they provide will be invaluable as you build real applications with WebAssembly.

Introducing WebAssembly

If the modern programmable web is merely a toy store, then WebAssembly is a toy warehouse filled with toys as far as the eye can see. Developers who spend most of their time working on the front end often long for some of the features, testability, and constraints prevalent in the world of servers and services. Likewise, folks who spend most of their time toiling behind the counter, away from the customer but carefully assembling all the parts of their order, often long for the fluid, expressive, blank canvas world the front

end represents. People who work in both worlds are keenly aware of the paradigm differences between the two and why the grass isn't always greener on the other side.

What if "sides" or "front" or "back" didn't matter anymore? What if there was a new way of doing things, where you could write loosely coupled business logic that flows between servers, services, clients, and browsers without any shenanigans? What if you could have the best parts of the front- and back-end worlds and still choose the most appropriate language for your problems?

By the time you reach the end of this book, you'll have learned enough about WebAssembly development that these propositions won't sound like they came from a snake-oil salesman. They'll ring true and hopefully inspire you to start building amazing new WebAssembly applications.

What Is WebAssembly?

The WebAssembly home page[1] says that it is a binary instruction format for a stack-based virtual machine. *Wasm* (a contraction, not an acronym, for WebAssembly) is designed to be *portable* (capable of running on different OSes, architectures, and environments without modification), and used as a compilation target for higher-level languages like C++, Rust, Go, and many others. The website also claims that Wasm enables deployment on the web for client and server applications alike.

Let's pick this definition apart a bit, because it's rather dense.

First, and most importantly, WebAssembly is a *portable binary instruction format*. This is very similar to the original intent behind Java's bytecode and, if you're familiar with the .NET Framework, you may recognize this concept as implemented in *ILASM*, the low-level instruction set supporting the Common Language Runtime. You'll see this in depth in the next chapter, but for now it should suffice to know that the operations encoded in a WebAssembly module are not tightly coupled to any one hardware architecture or operating system, and these operations are just codes that a parser knows how to interpret.

Next, there's the phrase *stack-based virtual machine*. We'll go over this in detail soon, but the short explanation is that this stack machine simultaneously contributes to WebAssembly's tremendous speed, power, and several of its limitations.

1. webassembly.org

Finally, there's a spot in the definition on which I fundamentally disagree. The phrase "deployment on the web" might limit your thinking and your imagination. This is a portable format that can run *anywhere* you can build a *host*, which you'll also be learning about later. Limiting WebAssembly's scope to the web (despite its name) does it a disservice.

What WebAssembly Is Not

The first question I get asked once I get on my WebAssembly soapbox is, "Isn't Wasm just another transpile target for JS?" *Transpiling* is translating from one high-level source language to another high-level source language. This is in contrast to the usual compiling, which takes a high-level source language and translates it into a low-level machine code. For example, converting TypeScript or React JSX into browser-executable JavaScript is done through transpiling. WebAssembly is *not* a JavaScript transpile target (though you can actually compile *TypeScript* into a Wasm module if you're into that sort of thing).

WebAssembly is also *not* meant to replace JavaScript. This is somewhat of a controversial opinion, as a large group of WebAssembly devotees online are convinced that it represents the death knell of JavaScript. While it might signal the beginning of a new era in which you write significantly less *manual* JavaScript, you still need JS to host WebAssembly 1.0 in the browser.

It's also not intended as a mere replacement for (or successor to) Flash, Silverlight, Adobe AIR, or Java Applets. As you'll discover, WebAssembly isn't run as a process outside the browser. How seamlessly it integrates with the user experience is entirely up to the developer and the tools they use.

Another important thing to remember is that WebAssembly, on its own, isn't a programming language. While there is both a binary and a text format, writing it by hand for anything beyond a few samples would take far too long and be too difficult to test and troubleshoot. Knowing how to write it by hand, however, will help you make the right decisions as you learn to build WebAssembly applications with *Rust*.

WebAssembly doesn't stand on its own. Like a game cartridge without a console or a BluRay disc without a player, it's incomplete in isolation. Much like a symbiote that needs to feed off of its host to survive, WebAssembly can't interact with anything outside the bounds of its own sandbox unless the host allows it. All I/O and other interactions are done entirely at the behest of the host such as a browser or a console application. While this might sound like

an unfair limitation, you'll see a number of times throughout this book why this is actually a good thing.

Try It Out

The best way to start learning something is to jump right in. Imagine one of those ball pits kids get to play in but, sadly, us adults are usually forbidden. As learners, our first exposure to new material is like jumping into this pit, surrounded by a dizzying array of colors and buried up to our necks in confusion. As we struggle to make our way to the far end of the ball pit, we gradually get our footing, the colors become more familiar, the shape and landscape of the ball pit gets smaller, and we're able to reason about it. After a while, we crawl out of the pit, having learned enough that we're eager to jump back in and discover more.

For WebAssembly, you're going to jump into the ball pit by using an online tool called *WebAssembly Studio*. In December 2017, *Mozilla* started working on this project as a way to provide a low-friction introduction to WebAssembly. It's a combination of some of their other WebAssembly tools like *WasmFiddle*.[2] This tool is entirely online and lets you create new projects in C, Rust, or *AssemblyScript*, a tool for compiling TypeScript to WebAssembly. It has gone through bursts of community activity and contribution since its creation.

Open up your browser (any up-to-date version of Firefox, Edge, Chrome, or Safari should work) and point it at webassembly.studio. You'll come to a screen that presents the following options for creating a new project:

- Empty C Project
- Empty Rust Project
- Empty AssemblyScript Project
- Hello World in C
- Hello World Rust Project

Choose "Empty Rust Project" and then click the Create button. This will create a new project within WebAssembly Studio, including a `README.md` file, a `src` directory, and the file you want to explore: `main.rs`. You'll see the following code:

```
#[no_mangle]
pub extern "C" fn add_one(x: i32) -> i32 {
  x + 1
}
```

2. wasdk.github.io/WasmFiddle/

The WebAssembly Studio site, with its almost entirely black color scheme, doesn't make for print-friendly screenshots so none have been included here.

If you're familiar with Rust, then this should be self-explanatory. If you're new to Rust, don't worry—you'll spend quite a bit of time working through Rust's syntax throughout the book. This code listing defines a function that adds 1 to a 32-bit signed integer, returning that value in a signed 32-bit integer. Believe it or not, this code can produce a WebAssembly module.

Using Rust's `no_mangle` macro tells the compiler not to change the signature of the function during compilation. As you'll see in upcoming chapters, there are some aspects of executing WebAssembly that require some naming conventions.

Click the Build and Run button and you'll see the blue bar on the bottom of the editor flash. Then, the white square canvas in the bottom right-hand corner will display the number *42*.

Congratulations, you've just written, compiled, and executed your first WebAssembly module. Where did the number 42 come from? Take a look at the `main.js` file by clicking on it in your browser:

```
fetch('../out/main.wasm').then(response =>
  response.arrayBuffer()
).then(bytes => WebAssembly.instantiate(bytes)).then(results => {
  instance = results.instance;
➤  document.getElementById("container").innerText = instance.exports.add_one(41);
}).catch(console.error);
```

Don't worry if some of this JavaScript looks unfamiliar to you—it'll be second nature by the time we're done. The indicated line of code invokes the `add_one` function in the WebAssembly module and places the result value inside the `container` DOM element. And now you've built a WebAssembly module that adds 1 to any number. It might not be all that impressive, but you've taken that first leap into WebAssembly.

The pattern of writing Rust code, compiling to WebAssembly, and then running the module in a browser is one that you'll repeat many times throughout this book. For now, though, let's take a step back from this code and take a look at what makes WebAssembly tick so you can have a better idea of what's happening in the preceding example.

Understanding WebAssembly Architecture

In this section, you'll get a good look inside the engine that makes WebAssembly work. Its unique architecture makes it incredibly powerful, portable, and efficient—though this power comes with some limitations.

Stack Machines

The type of computer that you're using right now is likely a *Register Machine*. Laptops, desktops, mobile devices, virtual machines, even microcontrollers and embedded devices are register machines. A register machine is a machine (physical or virtual) where the processor instructions explicitly refer to certain registers, or data storage locations, on the processor. Accessing these registers is fast and efficient because the data is available directly within the CPU.

For example, if you want to add two numbers together, you'd use the `ADD` instruction and you'd pass it the names of two registers as parameters, as shown in this bit of x86 assembly:

```
ADD al, ah
```

In the preceding code, the values contained in `ah` and `al` will be added together, with the result stored in `al`.

WebAssembly is a *stack machine*. In a stack machine, most of the instructions assume that the operands are sitting on the stack, rather than stored in specified registers. The WebAssembly stack is a LIFO (Last In, First Out) stack. If you're unfamiliar with the concept of a stack: it is as its name implies—values are piled (stacked) on top of each other, and unlike arrays where you can access any data regardless of location in the pile, stacks only allow you to pop data off or push data onto the top.

To add two numbers in a stack machine, you *push* those numbers onto the top of the stack. Then you push the `ADD` instruction onto the stack. The two operands and the instruction are then popped off the top and the result of the addition is pushed on in their place.

There are a number of advantages to a stack machine that made it an appealing choice for WebAssembly: their small binary size, efficient instruction coding, and ease of portability just to name a few.

There are some fairly well-known stack machines, including the *Java Virtual Machine* (JVM) and the bytecode executor for the .NET *Common Language Runtime*. In the case of those virtual machines, developers are spared the

effort of writing assembly or thinking in prefix or Polish[3] (where the operator comes first) notation because of the intermediate steps and code generation happening behind the scenes.

Data Types

Admit it—you've been spoiled. Modern programming languages with hashes, lists, arrays, sets, extra-large numbers, and tuples have spoiled you. These languages also probably let you create your own types through structs or classes. Some of them even let you overload operators, and some of those overloads can even work on custom types. The world is your oyster and you have few limits. That is not the world of WebAssembly. As their name should imply, *assembly* languages are designed to be made up of primitives that can be used as building blocks by higher level languages.

WebAssembly 1.0 has exactly four data types:

Type	Description
i32	32-Bit Integer
i64	64-Bit Integer
f32	32-Bit Floating-Point Number
f64	64-Bit Floating-Point Number

One aspect of this relatively limited set of data types is that WebAssembly doesn't assign any intrinsic signed-ness to numbers as they're stored. The assumption of whether a number is signed or unsigned is only performed at the time of an operation. For example, while there's only one i32 data type, there are signed and unsigned versions of that type's arithmetic operators, e.g. i32.add and i32.add_u.

When you're using a high-level language that compiles to WebAssembly on your behalf, you shouldn't have to worry about this subtlety. But when you're writing raw Wasm in the text format by hand, it could trip you up in unexpected ways.

Control Flow

WebAssembly's handling of control flow is a little different than other, less portable assembly languages. WebAssembly goes to great lengths to ensure that its control flow can't invalidate type safety, and can't be hijacked by attackers even with a "heap corruption"[4]-style attack in linear memory. For

3. en.wikipedia.org/wiki/Polish_notation
4. https://pdfs.semanticscholar.org/14f1/4b032235c345dfb3b3ecc8a879bbe4072407.pdf

example, many assembly languages allow easily exploited blind jump instructions, whereas you'll discover that WebAssembly does not. This additional layer of safety pairs well with the safety-first philosophy of Rust.

Wasm control flow is accomplished the same way everything else is within a stack machine—by pushing things onto, and popping things off of, the stack. For example, with an if instruction, if whatever is at the top of the stack evaluates as true (non-zero), then the if branch will be executed.

Take a look at an example of the if statement in action:

```
(if (i32.eq (call $getHealth) (i32.const 0))
  (then (call $doDeath))
  (else (call $stillAlive))
)
```

In this code, if our hypothetical player's health has reached 0, then we'll call the doDeath function, otherwise we'll call the stillAlive function. All those seemingly extra parentheses will make sense later in the chapter.

WebAssembly has the following control flow instructions available:

Instruction	Description
if	Marks the beginning of an if branching instruction.
else	Marks the else block of an if instruction
loop	A labeled block used to create loops
block	A sequence of instructions, often used within expressions
br	Branch to the given label in a containing instruction or block
br_if	Identical to a branch, but with a prerequisite condition
br_table	Branches, but instead of to a label it jumps to a function index in a table
return	Returns a value from the instruction (1.0 only supports one return value)
end	Marks the end of a block, loop, if, or a function
nop	No self-respecting assembly language is without an operation that does nothing

Linear Memory

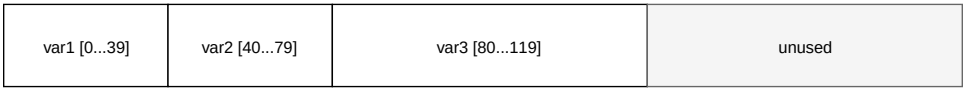
As you work with linear memory, you'll truly begin to appreciate the extent to which modern high-level languages have spoiled you. With most languages, you can quickly and easily create a new instance of something on the heap with an operator like new.

Internally, the compiler knows the size of this thing (or has some trick to compensate for not knowing). When you pass an instance of something to a function, the compiler knows whether you're passing a pointer or a value and how to arrange that value on your stack or heap in order to make the data available to a function.

WebAssembly doesn't have a heap in the traditional sense. There's no concept of a `new` operator. In fact, you don't allocate memory at the object level because there are no objects. There's also no garbage collection (at least not in the 1.0 MVP).

Instead, WebAssembly has *linear memory*. This is a contiguous block of bytes that can be declared internally within the module, exported out of a module, or imported from the host. Think of it as though the code you're writing is restricted to using a single variable that is a byte array. Your WebAssembly module can grow the linear memory block in increments called *pages* of 64KB if it needs more space. Sadly, determining if you need more space is entirely up to you and your code—there's no runtime to do this for you.

This image with variables and byte offsets illustrates just one way to store data in a block of linear memory (how you choose to use and fill linear memory is entirely up to you and your code):



In addition to the efficiency of direct memory access, there's another reason why it's ideal for WebAssembly: *security*. While the host can read and write any linear memory given to a Wasm module at any time, the Wasm module can never access any of the host's memory.

Direct DOM Access Is an Illusion

If you've seen WebAssembly demos that look like they're directly accessing the browser DOM from inside the module—that's an illusion. The host and module are sharing a block of linear memory, and the host is choosing to execute bespoke JavaScript to translate the contents of that shared memory area into updates to the DOM, just like you saw at the beginning of this chapter. This may change in future versions of WebAssembly, but for now, this remains little more than smoke and mirrors.

As you'll see in the coming chapters, linear memory is crucial to being able to create powerful applications with WebAssembly. Before using high-level languages like Rust, you should learn how to manipulate linear memory

manually so you can appreciate the extent of the work done on your behalf by tools and code generation and understand the impact of your designs.