

Web Assembly: Key Concepts and Implementation

A Technical Overview of WASM and Its
Performance Benefits

Mohit Varanasi

Roll Number-24CS10112

Motivation Behind Web Assembly

- ▶ Web applications require high performance for tasks like gaming, video editing, and simulations.
- ▶ JavaScript has limitations in execution speed and efficiency for computational-heavy tasks.
- ▶ Web Assembly provides near-native performance by allowing execution of compiled code in the browser.
- ▶ Web Assembly is very useful for tasks like cryptography where time is of the essence.
- ▶ Web Assembly (WASM) is highly beneficial for cloud computing, enabling serverless computing capabilities.

What is web assembly

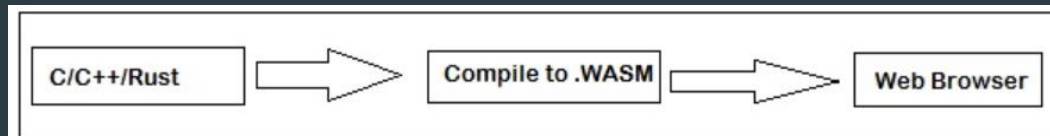
Web Assembly (wasm) is a simple machine model and executable format with an extensive specification.

It is designed to be portable, compact, and execute at or near native speeds.

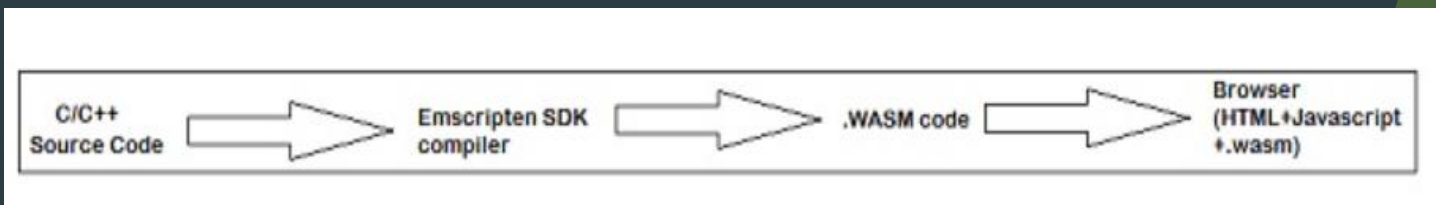
Although it has gained attention in the JavaScript and web communities, it is not dependent on any specific host environment. Therefore, we can anticipate that, in the near future, it will evolve into a portable executable format applicable across various contexts.

How Web Assembly Runs on a Browser

- ▶ Web Assembly runs in a secure sandboxed environment within modern web browsers.
- ▶ It works alongside JavaScript and can be called from JavaScript code.
- ▶ Uses a stack-based virtual machine to execute pre-compiled binary instructions.
- ▶ Web Assembly is called in javascript which is then called in the script part of html . In this Way Web Assembly is used to run in the browser.



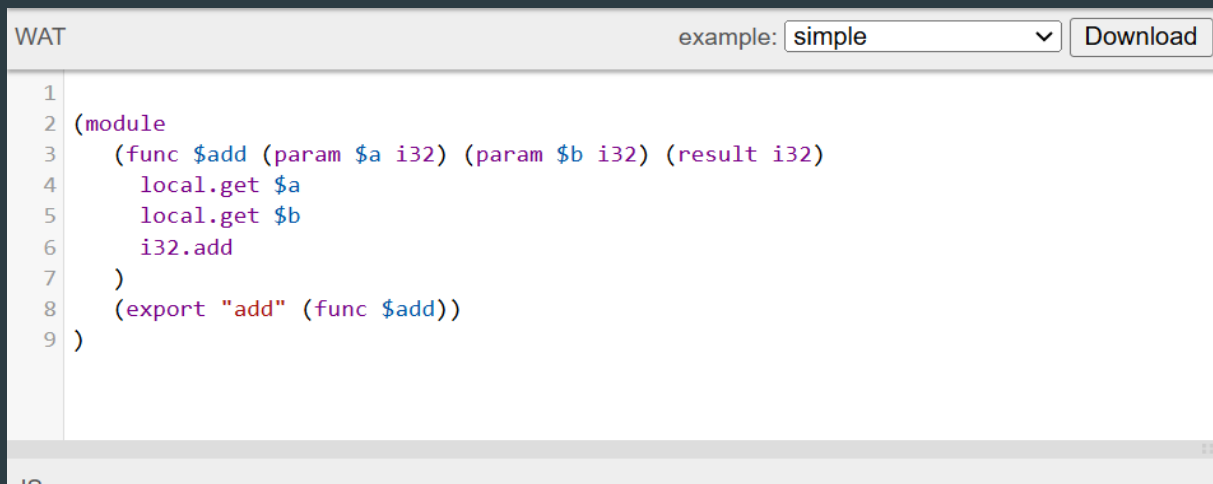
General compilation of Web Assembly



Compilation of Web Assembly through Rust

Textual Format of Web Assembly Code

- ▶ Web Assembly has a binary format (.wasm) for efficient execution.
- ▶ Also has a human-readable textual representation (.wat) using S-expressions.
- ▶ Example:



The screenshot shows a web-based editor for WebAssembly Text (WAT). At the top, there's a label 'WAT' on the left, a dropdown menu with 'example: simple' in the center, and a 'Download' button on the right. Below this is a text area containing the following WAT code:

```
1
2 (module
3   (func $add (param $a i32) (param $b i32) (result i32)
4     local.get $a
5     local.get $b
6     i32.add
7   )
8   (export "add" (func $add))
9 )
```

0000000: 0061 736d	; WASM_BINARY_MAGIC	0000027: 6a	; i32.add
0000004: 0100 0000	; WASM_BINARY_VERSION	0000028: 0b	; end
; section "Type" (1)		0000021: 07	; FIXUP func body size
0000008: 01	; section code	000001f: 09	; FIXUP section size
0000009: 00	; section size (guess)	; section "name"	
000000a: 01	; num types	0000029: 00	; section code
; func type 0		000002a: 00	; section size (guess)
000000b: 60	; func	000002b: 04	; string length
000000c: 02	; num params	000002c: 6e61 6d65	name ; custom section name
000000d: 7f	; i32	0000030: 01	; name subsection type
000000e: 7f	; i32	0000031: 00	; subsection size (guess)
000000f: 01	; num results	0000032: 01	; num names
0000010: 7f	; i32	0000033: 00	; elem index
0000009: 07	; FIXUP section size	0000034: 03	; string length
; section "Function" (3)		0000035: 6164 64	add ; elem name 0
0000011: 03	; section code	0000031: 06	; FIXUP subsection size
0000012: 00	; section size (guess)	0000038: 02	; local name type
0000013: 01	; num functions	0000039: 00	; subsection size (guess)
0000014: 00	; function 0 signature index	000003a: 01	; num functions
0000012: 02	; FIXUP section size	000003b: 00	; function index
; section "Export" (7)		000003c: 02	; num locals
0000015: 07	; section code	000003d: 00	; local index
0000016: 00	; section size (guess)	000003e: 01	; string length
0000017: 01	; num exports	000003f: 61	a ; local name 0
0000018: 03	; string length	0000040: 01	; local index
0000019: 6164 64	add ; export name	0000041: 01	; string length
000001c: 00	; export kind	0000042: 62	b ; local name 1
000001d: 00	; export func index	0000039: 09	; FIXUP subsection size
0000016: 07	; FIXUP section size	0000039: 18	; FIXUP section size
; section "Code" (10)			
000001e: 0a	; section code		
000001f: 00	; section size (guess)		
0000020: 01	; num functions		
; function body 0			
0000021: 00	; func body size (guess)		
0000022: 00	; local decl count		
0000023: 20	; local.get		
0000024: 00	; local index		
0000025: 20	; local.get		
0000026: 01	; local index		

Wasm code for The given Wat code

Writing WASM Modules in C++/Rust

- ▶ Web Assembly modules can be written in languages like C, C++, and Rust.
- ▶ This is done because web assembly wat is not easily human readable so the code is compiled in high level languages which have more resources and then compiled into .wasm format.
- ▶ Code is compiled into .wasm format using LLVM-based toolchains.
- ▶ Example (Rust):

```
pub extern "C" fn add(a: i32, b: i32) -> i32 {  
    a + b  
}
```

Steps To compile into WASM from Rust

First ensure you have rust installed . If not install it using Rustup:

Use this in the command line "curl --proto '=https' --tlsv1.2 -sSf <https://sh.rustup.rs> | sh"

Then add the web assembly target

For that we do this:

```
"rustup target add wasm32-unknown-unknown"
```

Then we create new Rust project using the cargo function:

```
cargo new wasm_project  
cd wasm_project
```


Write rust code

make the functions public to use in javascript

Compile to web assembly :

```
"cargo build --release --target wasm32-unknown-unknown"
```

The file will be in the location -"target/wasm32-unknown-unknown/release/your_project.wasm"

Install wasm-bindgen then to integrate this with javascript.

```
"cargo install wasm-bindgen-cli"
```

Then we use the function `wasm-bindgen` to integrate javascript and rust.

```
"wasm-bindgen --target web --out-dir wasm_output  
target/wasm32-unknown-unknown/release/your_project.wasm"
```

Alternatively-

You can run two other commands-

```
"cargo install wasm-pack"
```

```
wasm-pack build --target web"
```

Then we import the code through javascript.

Loading and Using Web Assembly in JavaScript

Web Assembly can be loaded and executed using JavaScript's Web Assembly API.

Example:

Javascript code for accessing the add function
Written in wat

JS

```
1 const wasmInstance =  
2   new WebAssembly.Instance(wasmModule, {});  
3 const {add} = wasmInstance.exports;  
4 for (let i = 0; i < 10; i++) {  
5   console.log(add(i, i));  
6 }  
7
```

How web assembly works in javascript

- ▶ Web assembly is based on a stack machine model .
- ▶ So, the input is all pushed onto the stack .
- ▶ Then the input is all removed and then the output is pushed back onto the stack.
- ▶ Memory in Web Assembly is an arraybuffer that holds the data. You can allocate memory by using the Javascript API `WebAssembly.memory()`.
- ▶ Web Assembly memory is stored in an array format i.e. a flat memory model that is easy to understand and perform the execution.

Coding	You can easily write code in Javascript. The code written is human readable and saved as .js. When used inside the browser you need to use a <script> tag.	WebAssembly and it is saved as .wat. It is difficult to write the code in .wat format. It is best to compile the code from some other high level language instead of writing from start in .wat. You cannot execute the .wat file inside the browser and has to convert to .wasm using the compilers or online tools available.
Execution	The code written in javascript when used inside the browser has to be downloaded, parsed, compiled and optimized.	We have WebAssembly code in .wasm already compiled and in binary format.
Memory Management	Javascript assigns memory when, variables are created and the memory is released when not used and are added to garbage collection.	Memory in WebAssembly is an arraybuffer that holds the data. You can allocate memory by using the Javascript API WebAssembly.memory(). WebAssembly memory is stored in an array format i.e. a flat memory model that is easy to understand and perform the execution. The disadvantage of memory model in WebAssembly is – <ul style="list-style-type: none"> Complex calculation takes time. WebAssembly does not support garbage collection that does not allow reuse of the memory and the memory is wasted.
Load Time & Performance	In case of javascript, when called inside the browser, the javascript file has to be downloaded, and parsed. Later, the parser converts the source code to bytecode that the javascript engine executes the code in the browser. The Javascript engine is very powerful and hence, the load time and performance of javascript is very fast in comparison to WebAssembly.	A most important goal of WebAssembly is to be faster than JavaScript. Wasm code generated from high-level languages is smaller in size and hence, the load time is faster. But, languages like GO, when compiled to wasm produce a big file size for a small piece of code. WebAssembly is designed in such a way that it is faster in compilation, and can run across all the major browsers. WebAssembly still has to add lots of improvements in terms of performance in comparison to javascript.
Debugging	Javascript is human-readable and can be debugged easily. Adding breakpoints to your javascript code inside the browser allows you to easily debug the code.	WebAssembly provides the code in text format, that is readable but, still very difficult to debug. Firefox does allow you to view the wasm code in .wat format inside the browser. You cannot add breakpoints in .wat and that is something that will be available in the future.
Browser Support	Javascript works well in all browsers.	All major web browsers have support for WebAssembly.

Overall Comparison : Javascript V/S Web Assembly

Ease to Code comparison Javascript V/S Web Assembly

- ▶ Javascript:
- ▶ You can easily write code in Javascript. The code written is human readable and saved as .js. When used inside the browser you need to use a `<script>` tag.
- ▶ Web Assembly:
- ▶ The code can be written in text format in Web Assembly and it is saved as .wat. It is difficult to write the code in .wat format. It is best to compile the code from some other high-level language instead of writing from start in .wat.
- ▶ You cannot execute the .wat file inside the browser and you have to convert it to .wasm using the compilers or online tools available.

Execution Comparison Javascript V/S Web Assembly

- ▶ Javascript:
- ▶ The code can be written in text format in WebAssembly and it is saved as .wat. It is difficult to write the code in .wat format. It is best to compile the code from some other high level language instead of writing from start in .wat.
- ▶ You cannot execute the .wat file inside the browser and has to convert to .wasm using the compilers or online tools available.
- ▶ Web Assembly:
- ▶ We have WebAssembly code in .wasm already compiled and in binary format.

Memory and Management Comparison Javascript V/S Web Assembly

- ▶ Javascript:
 - ▶ Javascript assigns memory when, variables are created and the memory is released when not used and are added to garbage collection.
- ▶ Web Assembly:
 - ▶ Memory in WebAssembly is an arraybuffer that holds the data. You can allocate memory by using the Javascript API `WebAssembly.memory()`.
 - ▶ WebAssembly memory is stored in an array format i.e. a flat memory model that is easy to understand and perform the execution.
 - ▶ The disadvantage of memory model in WebAssembly is –
 - Complex calculation takes time.
 - Webassembly does not support garbage collection that does not allow reuse of the memory and the memory is wasted.

Performance Comparison: JavaScript vs Web Assembly

```
JS Factorial: 10303.952880859375 ms
WASM Factorial: 44.64794921875 ms
```

20 factorial

```
JS Factorial: 0.051025390625 ms
JS Result: 2432902008176640000
WASM Factorial: 0.1279296875 ms
WASM Result: 2432902008176640000n
```

1 to 20
factorial (1000
times)

```
JS Factorial: 14.7080078125 ms
WASM Factorial: 7.884033203125 ms
```

1 to 20
factorial
(10,000,000
times)

- ▶ Web Assembly executes at near-native speed, while JavaScript runs through an interpreter or JIT compiler.
- ▶ however, web assembly is slower than javascript for small tasks as the transition from web assembly to javascript itself increases the time for computation.
- ▶ Benchmarks show WASM outperforms JavaScript for computationally heavy tasks.
- ▶ A possible reason for this is that when a javascript file is called inside a web browser the file has to be downloaded and then parsed. Then the parser converts the code into source code .

Ease of Debugging Comparison Javascript V/S Web Assembly

- ▶ Javascript:
- ▶ Javascript is human-readable and can be debugged easily. Adding breakpoints to your javascript code inside the browser allows you to easily debug the code.
- ▶ Web Assembly:
- ▶ WebAssembly provides the code in text format, that is readable but, still very difficult to debug. Firefox does allow you to view the wasm code in .wat format inside the browser.
- ▶ You cannot add breakpoints in .wat and that is something that will be available in the future.
- ▶ So , it might be easier to debug in the future but at the moment it is a difficult task.

Comparison of Rust Vs Web Assembly in different browsers

- ▶ We can see that the time taken for running Javascript in all the browsers is almost same.
- ▶ We can see that firefox is faster compared to every other browser in terms of web assembly run time.
- ▶ This can probably be attributed to the fact that firefox is one of the front runners in compiling of web assembly. Due to this, the integration of web assembly features is more efficient .

Browser	Web Assembly Run time	Javascript Run time
Brave	~50ms	~10s
Edge	~40ms	~10s
Chrome	~50ms	~10s
opera	~40ms	~10s
firefox	~10ms	~15s

Challenges and Limitations of Web Assembly

- ▶ Debugging Web Assembly is very difficult and that is a major problem.
- ▶ Tools for Web Assembly need to be improved.
- ▶ There is a problem in the memory model which needs to be fixed.
- ▶ There are also security concerns related to web assembly

Thank You!