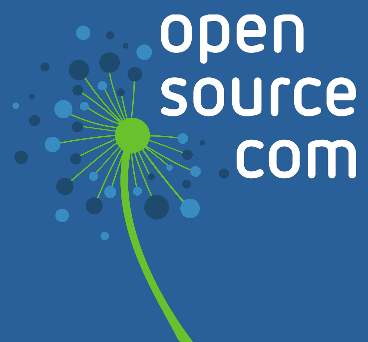


WebAssembly

by Opensource.com

We are Opensource.com

Opensource.com is a community website publishing stories about creating, adopting, and sharing open source solutions. Visit Opensource.com to learn more about how the open source way is improving technologies, education, business, government, health, law, entertainment, humanitarian efforts, and more.

Do you have an open source story to tell? Submit a story idea at opensource.com/story

Email us at open@opensource.com

Table of Contents

Why everyone is talking about WebAssembly....3

How to write 'Hello World' in WebAssembly....7

Make Conway's Game of Life in WebAssembly....11

Writing WebAssembly for speed and code reuse....22

Stepping WebAssembly up a notch with security....36

Create web user interfaces with Qt WebAssembly instead of JavaScript....40

Why should you use Rust in WebAssembly?....44

Why everyone is talking about WebAssembly

By Gordon Haff and Mike Bursell

If you haven't heard of WebAssembly yet, then you will soon. It's one of the industry's best-kept secrets, but it's everywhere. It's supported by all the major browsers, and it's coming to the server-side, too. It's fast. It's being used for gaming. It's an open standard from the World Wide Web Consortium (W3C), the main international standards organization for the web.

"Wow," you may be saying, "this sounds like something I should learn to code in!" You'd be right, but you'd be wrong too; you don't code in WebAssembly. Let's take some time to learn about the technology that's often fondly abbreviated to "Wasm."

Where did it come from?

Going back about ten years, there was a growing recognition that the widely-used JavaScript wasn't fast enough for many purposes. JavaScript was undoubtedly successful and convenient. It ran in any browser and enabled the type of dynamic web pages that we take for granted today. But it was a high-level language and wasn't designed with compute-intensive workloads in mind.

However, although engineers responsible for the leading web browsers were generally in agreement about the performance problem, they weren't aligned on what to do about it. Two camps emerged. Google began its Native Client project and, later, its Portable Native Client variation, to focus on allowing games and other software written in C/C++ to run in a secure compartment within Chrome. Mozilla, meanwhile, won the backing of Microsoft for asm.js, an approach that updated the browser so it can run a low-level subset of JavaScript instructions very quickly (another project enabled the conversion of C/C++ code into these instructions).

With neither camp gaining widespread adoption, the various parties agreed to join forces in 2015 around a new standard called WebAssembly that built on the basic approach taken by asm.js. As CNET's Stephen Shankland wrote at the time, "On today's Web, the browser's JavaScript translates those instructions into machine code. But with WebAssembly, the programmer does a lot of the work earlier in the process, producing a program that's in between the two states. That frees the browser from a lot of the hard work of creating the machine code, but it also fulfills the promise of the Web—that software will run on any device with a browser regardless of the underlying hardware details."

In 2017, Mozilla declared it to be a minimum viable product and brought it out of preview. All the main browsers had adopted it by the end of that year. In December 2019, the WebAssembly Working Group published the three WebAssembly specifications as W3C recommendations.

WebAssembly defines a portable binary code format for executable programs, a corresponding textual assembly language, and interfaces for facilitating interactions between such programs and their host environment. WebAssembly code runs within a low-level virtual machine that mimics the functionality of the many microprocessors upon which it can be run. Either through Just-In-Time (JIT) compilation or interpretation, the WebAssembly engine can perform at nearly the speed of code compiled for a native platform.

Why the interest now?

Certainly, some of the recent interest in WebAssembly stems from that initial desire to run more compute-intensive code in browsers. Laptop users, in particular, are spending more and more of their time in a browser (or, in the case of Chromebooks, essentially all their time). This trend has created an urgency around removing barriers to running a broad range of applications within a browser. And one of those barriers is often some aspect of performance, which is what WebAssembly and its predecessors were originally conceived to address.

However, WebAssembly isn't just for browsers. In 2019, Mozilla announced a project called WASI (WebAssembly System Interface) to standardize how WebAssembly code interacts with operating systems outside of a browser context. With the combination of browser support for WebAssembly and WASI, compiled binaries will be able to run both within and without browsers, across different devices and operating systems, at near-native speeds.

WebAssembly's low overhead immediately makes it practical for use beyond browsers, but that's arguably table stakes; there are obviously other ways to run applications that don't introduce performance bottlenecks. Why use WebAssembly, specifically?

One important reason is its portability. Widely used compiled languages like C++ and Rust are probably the ones most associated with WebAssembly today. However, a wide range of other languages compile to or have their virtual machines in WebAssembly. Furthermore, while WebAssembly assumes certain prerequisites for its execution environments, it is designed to execute efficiently on a variety of operating systems and instruction set architectures. WebAssembly code can, therefore, be written using a wide range of languages and run on a wide range of operating systems and processor types.

Another WebAssembly advantage stems from the fact that code runs within a virtual machine. As a result, each WebAssembly module executes within a sandboxed environment, separated from the host runtime using fault isolation techniques. This implies, among other things, that applications execute in isolation from the rest of their host environment and can't escape the sandbox without going through appropriate APIs.

WebAssembly in action

What does all this mean in practice?

One example of WebAssembly in action is Enarx.

Enarx is a project that provides hardware independence for securing applications using Trusted Execution Environments (TEE). Enarx lets you securely deliver an application compiled into WebAssembly all the way into a cloud provider and execute it remotely. As Red Hat security engineer Nathaniel McCallum puts it, "The way that we do this is, we take your application as inputs, and we perform an attestation process with the remote hardware. We validate that the remote hardware is, in fact, the hardware that it claims to be, using cryptographic techniques. The end result of that is not only an increased level of trust in the hardware that we're speaking to; it's also a session key, which we can then use to deliver encrypted code and data into this environment that we have just asked for cryptographic attestation on."

Another example is OPA, the Open Policy Agent, which announced in November 2019 that you could compile their policy definition language, Rego, into WebAssembly. Rego lets you write logic to search and combine JSON/YAML data from different sources to ask questions like, "Is this API allowed?"

OPA has been used to policy-enable software including, but not limited to, Kubernetes. Simplifying policy using tools like OPA is considered an important step for properly securing Kubernetes deployments across a variety of different environments. WebAssembly's portability and built-in security features are a good match for these tools.

Our last example is Unity. Remember, we mentioned at the beginning of the article that WebAssembly is used for gaming? Well, Unity, the cross-platform game engine, was an early adopter of WebAssembly, providing the first demo of Wasm running in browsers, and, since August 2018, has used WebAssembly as the output target for the Unity WebGL build target.

These are just a few of the ways WebAssembly has already begun to make an impact. Find out more and keep up to date with all things Wasm at https://webassembly.org/

How to write 'Hello World' in WebAssembly

By Stephan Avenwedde

WebAssembly is a bytecode format that virtually every browser can compile to its host system's machine code. Alongside JavaScript and WebGL, WebAssembly fulfills the demand for porting applications for platform-independent use in the web browser. As a compilation target for C++ and Rust, WebAssembly enables web browsers to execute code at near-native speed.

When you talk about a WebAssembly, application, you must distinguish between three states:

1. **Source code (C++ or Rust):** You have an application written in a compatible language that you want to execute in the browser.
2. **WebAssembly bytecode:** You choose WebAssembly bytecode as your compilation target. As a result, you get a `.wasm` file.
3. **Machine code (opcode):** The browser loads the `.wasm` file and compiles it to the corresponding machine code of its host system.

WebAssembly also has a text format that represents the binary format in human-readable text. For the sake of simplicity, I will refer to this as **WASM-text**. WASM-text can be compared to high-level assembly language. Of course, you would not write a complete application based on WASM-text, but it's good to know how it works under the hood (especially for debugging and performance optimization).

This article will guide you through creating the classic *Hello World* program in WASM-text.

Creating the .wat file

WASM-text files usually end with `.wat`. Start from scratch by creating an empty text file named `helloworld.wat`, open it with your favorite text editor, and paste in:

```
(module
;; Imports from JavaScript namespace
(import  "console"  "log" (func  $log (param  i32  i32)))
;; Import log function
(import  "js"  "mem" (memory  1)) ;; Import 1 page of memory (54kb)
;; Data section of our module
(data (i32.const 0) "Hello World from WebAssembly!")
;; Function declaration: Exported as helloWorld(), no arguments
(func (export  "helloWorld")
  i32.const 0  ;; pass offset 0 to log
  i32.const 29  ;; pass length 29 to log (strlen of sample text)
  call  $log
 )
)
```

The WASM-text format is based upon S-expressions. To enable interaction, JavaScript functions are imported with the `import` statement, and WebAssembly functions are exported with the `export` statement. For this example, import the `log` function from the `console` module, which takes two parameters of type `i32` as input and one page of memory (64KB) to store the string.

The string will be written into the `data` section at offset `0`. The `data` section is an overlay of your memory, and the memory is allocated in the JavaScript part.

Functions are marked with the keyword `func`. The stack is empty when entering a function. Function parameters are pushed onto the stack (here offset and length) before another function is called (see `call $log`). When a function returns an `f32` type (for example), an `f32` variable must remain on the stack when leaving the function (but this is not the case in this example).

Creating the .wasm file

The WASM-text and the WebAssembly bytecode have 1:1 correspondence. This means you can convert WASM-text into bytecode (and vice versa). You already have the WASM-text, and now you want to create the bytecode.

The conversion can be performed with the WebAssembly Binary Toolkit (WABT). Make a clone of the repository at that link and follow the installation instructions.

After you build the toolchain, convert WASM-text to bytecode by opening a console and entering:

```
wat2wasm helloworld.wat -o helloworld.wasm
```

You can also convert bytecode to WASM-text with:

```
wasm2wat helloworld.wasm -o helloworld_reverse.wat
```

A `.wat` file created from a `.wasm` file does not include any function nor parameter names. By default, WebAssembly identifies functions and parameters with their index.

Compiling the .wasm file

Currently, WebAssembly only coexists with JavaScript, so you have to write a short script to load and compile the `.wasm` file and do the function calls. You also need to define the functions you will import in your WebAssembly module.

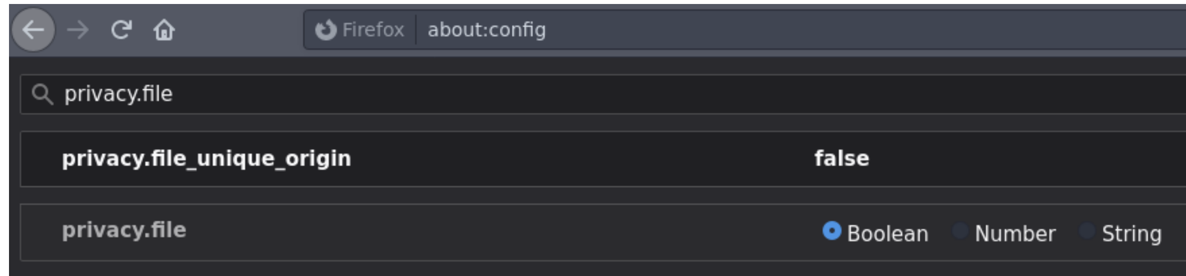
Create an empty text file and name it `helloworld.html`, then open your favorite text editor and paste in:

```
<!DOCTYPE  html>
<html><head>
    <meta  charset="utf-8"><title>Simple template</title></head>
  <body>
    <script>
      var memory = new  WebAssembly.Memory({initial:1});
      function  consoleLogString(offset, length) {
        var  bytes = new  Uint8Array(memory.buffer, offset, length);
        var  string = new  TextDecoder('utf8').decode(bytes);
        console.log(string);
      };
      var  importObject = {
        console: {
          log:  consoleLogString },
        js : { mem:  memory }
      };
      WebAssembly.instantiateStreaming(fetch('helloworld.wasm'), importObject)
      .then(obj  => {
        obj.instance.exports.helloWorld();
      });
    </script>
  </body>
</html>
```

The `WebAssembly.Memory(...)` method returns one page of memory that is 64KB in size. The function `consoleLogString` reads a string from that memory page based on the

length and offset. Both objects are passed to your WebAssembly module as part of the `importObject`.

Before you can run this example, you may have to allow Firefox to access files from this directory by typing `about:config` in the address line and setting `privacy.file_unique_origin` to `true`:



(Stephan Avenwedde, CC BY-SA 4.0)

> **Caution:** This makes you vulnerable to the CVE-2019-11730 security issue.

Now, open `helloworld.html` in Firefox and enter **Ctrl**+**K** to open the developer console.

(Stephan Avenwedde, CC BY-SA 4.0)

Learn more

This Hello World example is just one of the detailed tutorials in MDN's Understanding WebAssembly text format documentation. If you want to learn more about WebAssembly and how it works under the hood, take a look at these docs.

Make Conway's Game of Life in WebAssembly

By Mohammed Saud

Conway's Game of Life is a popular programming exercise to create a cellular automaton, a system that consists of an infinite grid of cells. You don't play the game in the traditional sense. In fact, it's sometimes referred to as a game for zero players.

Once you start the Game of Life, the game plays itself to multiply and sustain "life." In the game, digital cells representing lifeforms are allowed to change states as defined by a set of rules. When the rules are applied to cells through multiple iterations, they exhibit complex behavior and interesting patterns.

The Game of Life simulation is a very good candidate for a WebAssembly implementation because of how computationally expensive it can be; every cell's state in the entire grid must be calculated for every iteration. WebAssembly excels at computationally expensive tasks due to its predefined execution environment and memory granularity, among many other features.

Compiling to WebAssembly

Although it's possible to write WebAssembly by hand, it is very unintuitive and error-prone as complexity increases. Most importantly, it's not intended to be written that way. It would be the equivalent of manually writing assembly language instructions.

Here's a simple WebAssembly function to add two numbers:

```
(func $Add (param $0 i32) (param $1 i32) (result i32)
    local.get $0
    local.get $1
    i32.add
)
```

It is possible to compile WebAssembly modules using many existing languages, including C, C++, Rust, Go, and even interpreted languages like Lua and Python. This list is only growing.

One of the problems with using existing languages is that WebAssembly does not have much of a runtime. It does not know what it means to free a pointer or what a closure is. All these language-specific runtimes have to be included in the resulting WebAssembly binaries. Runtime size varies by language, but it has an impact on module size and execution time.

AssemblyScript

AssemblyScript is one language that is trying to overcome some of these challenges with a different approach. AssemblyScript is designed specifically for WebAssembly, with a focus on providing low-level control, producing smaller binaries, and reducing the runtime overhead.

AssemblyScript uses a strictly typed variant of TypeScript, a superset of JavaScript. Developers familiar with TypeScript do not have to go through the trouble of learning an entirely new language.

Getting started

The AssemblyScript compiler can easily be installed through Node.js. Start by initializing a new project in an empty directory:

```
npm init
npm install --save-dev assemblyscript
```

If you don't have Node installed locally, you can play around with AssemblyScript on your browser using the nifty WebAssembly Studio application.

AssemblyScript comes with `asinit`, which should be installed when you run the installation command above. It is a helpful utility to quickly set up an AssemblyScript project with the recommended directory structure and configuration files:

```
npx asinit .
```

The newly created `assembly` directory will contain all the AssemblyScript code, a simple example function in `assembly/index.ts`, and the `asbuild` command inside `package.json`. `asbuild`, which compiles the code into WebAssembly binaries.

When you run `npm run asbuild` to compile the code, it creates files inside `build`. The `.wasm` files are the generated WebAssembly modules. The `.wat` files are the modules in text format and are generally used for debugging and inspection.

You have to do a little bit of work to get the binaries to run on a browser.

First, create a simple HTML file, `index.html`:

```
<html>
    <head>
        <meta charset=utf-8>
        <title>Game of life</title>
    </head>

    <body>
        <script src='./index.js'></script>
    </body>
</html>
```

Next, replace the contents of `index.js` with the code snippet below to load the WebAssembly modules:

```
const runWasm = async () => {
  const module = await
WebAssembly.instantiateStreaming(fetch('./build/optimized.wasm'));
  const exports = module.instance.exports;
  console.log('Sum = ', exports.add(20, 22));
};
runWasm();
```

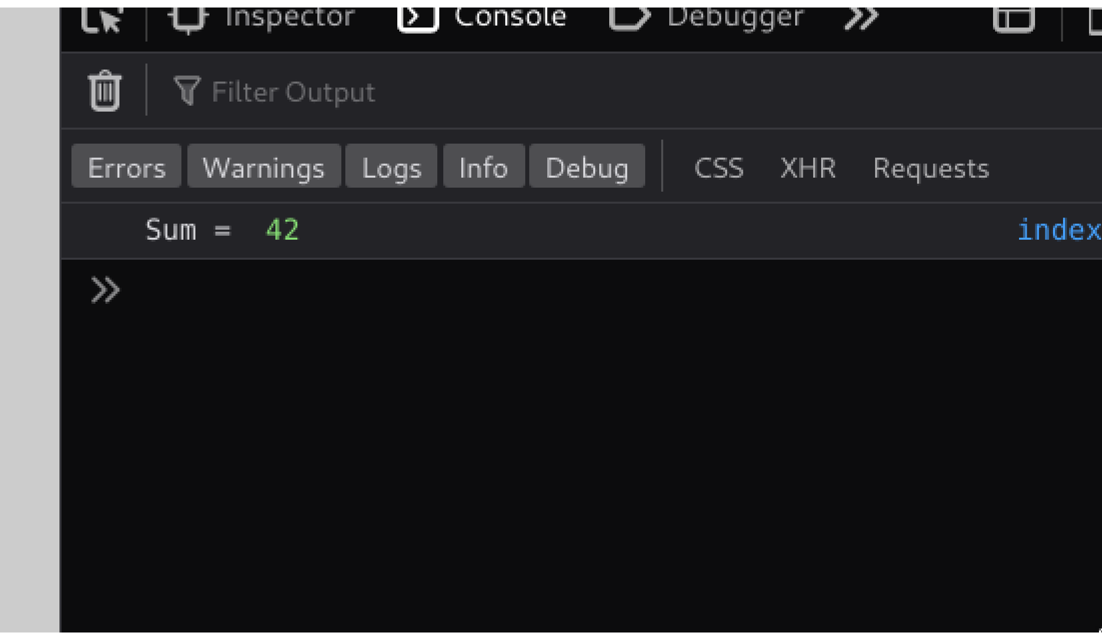
This `fetches` the binary and passes it to `WebAssembly.instantiateStreaming`, the browser API that compiles a module into a ready-to-use instance. This is an asynchronous operation, so it is run inside an async function so that await can be used to wait for it to finish compiling.

The `module.instance.exports` object contains all the functions exported by AssemblyScript. Use the example function in `assembly/index.ts` and log the result.

You will need a simple development server to host these files. There are a lot of options listed in this gist. I used node-static:

```
npm install -g node-static
static
```

You can view the result by pointing your browser to `localhost:8080` and opening the console.



(Mohammed Saud, CC BY-SA 4.0)

Drawing to a canvas

Draw all the cells onto a `<canvas>` element:

```
<body>
    <canvas id=canvas></canvas>
    ...
</body>
```

Add some CSS in the `<head>` section of your page:

```
<style type=text/css>
body { background: #ccc; }
canvas {
  display: block;
  padding: 0;
  margin: auto;
  width: 40%;
  image-rendering: pixelated;
  image-rendering: crisp-edges;
}
```

The `image-rendering` styles are used to prevent the canvas from smoothing and blurring out pixelated images.

You need a canvas drawing context in `index.js`:

```
const canvas = document.getElementById('canvas');
const ctx = canvas.getContext('2d');
```

There are many functions in the [Canvas API] that you could use for drawing—but you need to draw using WebAssembly, not JavaScript.

Remember that WebAssembly does NOT have access to the browser APIs that JavaScript has, and any call that needs to be made should be interfaced through JavaScript. This also means that your WebAssembly module runs the fastest when there is as little communication with JavaScript as possible.

One method is to create ImageData (a data type for the underlying pixel data of a canvas), fill it up with the WebAssembly module's memory, and draw it on the canvas. This way, if the memory buffer is updated inside WebAssembly, it will be immediately available to the `ImageData`.

Define the pixel count of the canvas and create an `ImageData` object:

```
const WIDTH = 10, HEIGHT = 10;
const runWasm = async () => {
...
canvas.width = WIDTH;

canvas.height = HEIGHT;
const ctx = canvas.getContext('2d');
const memoryBuffer = exports.memory.buffer;
const memoryArray = new Uint8ClampedArray(memoryBuffer)
const imageData = ctx.createImageData(WIDTH, HEIGHT);

imageData.data.set(memoryArray.slice(0, WIDTH * HEIGHT * 4));

ctx.putImageData(imageData, 0, 0);
```

The memory of a WebAssembly module is provided in `exports.memory.buffer` as an ArrayBuffer. You need to use it as an array of 8-bit unsigned integers or `Uint8ClampedArray`. Now you can fill up the module's memory with some pixels. In `assembly/index.ts`, you first need to grow the available memory:

```
memory.grow(1);
```

WebAssembly does not have access to memory by default and needs to request it from the browser using the `memory.grow` function. Memory grows in chunks of 64Kb, and the number of required chunks can be specified when calling it. You will not need more than one chunk for now.

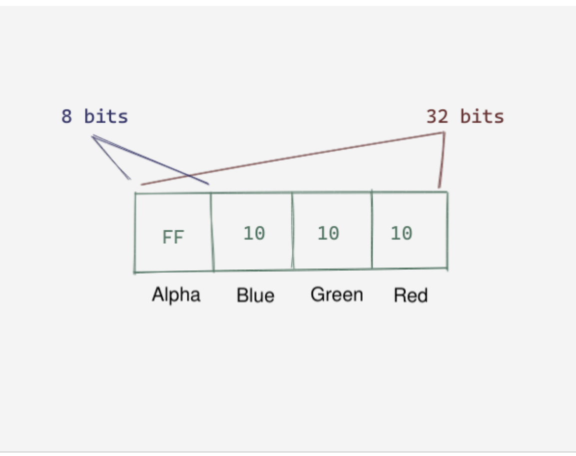
Keep in mind that memory can be requested multiple times, whenever needed, and once acquired, memory cannot be freed or given back to the browser.

Writing to the memory:

```
store<u32>(0, 0xff101010);
```

A pixel is represented by 32 bits, with the RGBA values taking up 8 bits each. Here, RGBA is defined in reverse—ABGR—because WebAssembly is little-endian.

The `store` function stores the value `0xff101010` at index `0`, taking up 32 bits. The alpha value is `0xff` so that the pixel is fully opaque.



(Mohammed Saud, CC BY-SA 4.0)

Build the module again with `npm run asbuild` before refreshing the page to see your first pixel on the top-left of the canvas.

Implementing rules

The Game of Life Wikipedia page summarizes the rules nicely:

1. Any live cell with fewer than two live neighbors dies, as if by underpopulation.
2. Any live cell with two or three live neighbors lives on to the next generation.
3. Any live cell with more than three live neighbors dies, as if by overpopulation.

4. Any dead cell with exactly three live neighbors becomes a live cell, as if by reproduction.

You need to iterate through all the rows, implementing these rules on each cell. You do not know the width and height of the grid, so write a little function to initialize the WebAssembly module with this information:

```
let universe_width: u32;
let universe_height: u32;
let alive_color: u32;
let dead_color: u32;
let chunk_offset: u32;
export function init(width: u32, height: u32): void {
  universe_width = width;
  universe_height = height;
  chunk_offset = width * height * 4;
  alive_color = 0xff101010;
  dead_color = 0xffefefef;
}
```

Now you can use this function in `index.js` to provide data to the module:

```
exports.init(WIDTH, HEIGHT);
```

Next, write an `update` function to iterate over all the cells, count the number of active neighbors for each, and set the current cell's state accordingly:

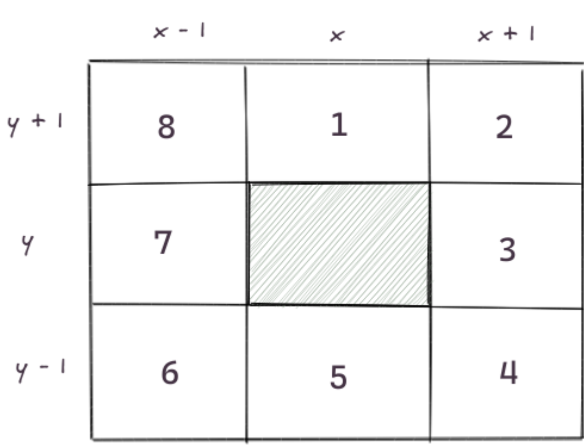
```
export function update(): void {
  for (let x: u32 = 0; x < universe_width; x++) {
    for (let y: u32 = 0; y < universe_height; y++) {
      const neighbours = countNeighbours(x, y);
      if (neighbours < 2) {
        // less than 2 neighbours, cell is no longer alive
        setCell(x, y, dead_color);
      } else if (neighbours == 3) {
        // cell will be alive
        setCell(x, y, alive_color);
      } else if (neighbours > 3) {
        // cell dies due to overpopulation
        setCell(x, y, dead_color);
      }
    }
  }
  copyToPrimary();
}
```

You have two copies of cell arrays, one representing the current state and the other for calculating and temporarily storing the next state. After the calculation is done, the second array is copied to the first for rendering.

The rules are fairly straightforward, but the `countNeighbours()` function looks interesting. Take a closer look:

```
function countNeighbours(x: u32, y: u32): u32 {
  let neighbours = 0;
  const max_x = universe_width - 1;
  const max_y = universe_height - 1;
  const y_above = y == 0 ? max_y : y - 1;
  const y_below = y == max_y ? 0 : y + 1;
  const x_left = x == 0 ? max_x : x - 1;
  const x_right = x == max_x ? 0 : x + 1;
  // top left
  if(getCell(x_left, y_above) == alive_color) {
    neighbours++;
  }
  // top
  if(getCell(x, y_above) == alive_color) {
    neighbours++;
  }
  // top right
  if(getCell(x_right, y_above) == alive_color) {
    neighbours++;
  }
  ...
  return neighbours;
}
```

Creative Commons Attribution Share-alike 4.0



(Mohammed Saud, CC BY-SA 4.0)

Every cell has eight neighbors, and you can check if each one is in the `alive_color` state. The important situation handled here is if a cell is exactly on the edge of the grid. Cellular automata are generally assumed to be on an infinite space, but since infinitely large displays haven't been invented yet, stick to warping at the edges. This means when a cell goes off the top, it comes back in its corresponding position on the bottom. This is commonly known as toroidal space.

The `getCell` and `setCell` functions are wrappers to the `store` and `load` functions to make it easier to interact with memory using 2D coordinates:

```
@inline
function getCell(x: u32, y: u32): u32 {
  return load<u32>((x + y * universe_width) << 2);
}
@inline
function setCell(x: u32, y: u32, val: u32): void {
  store<u32>(((x + y * universe_width) << 2) + chunk_offset, val);
}
function copyToPrimary(): void {
  memory.copy(0, chunk_offset, chunk_offset);
}
```

The `@inline` is an annotation that requests that the compiler convert calls to the function with the function definition itself.

Call the update function on every iteration from `index.js` and render the image data from the module memory:

```
const FPS = 5;
const runWasm = async () => {
  ...
  const step = () => {
    exports.update();
    imageData.data.set(memoryArray.slice(0, WIDTH * HEIGHT * 4));
    ctx.putImageData(imageData, 0, 0);
    setTimeout(step, 1000 / FPS);
  };
  step();
```

At this point, when you compile the module and load the page, it shows nothing. The code works, but you don't have any living cells initially, so there are no new cells coming up.

Create a new function to randomly add cells during initialization:

```
function fillUniverse(): void {
  for (let x: u32 = 0; x < universe_width; x++) {
    for (let y: u32 = 0; y < universe_height; y++) {
      setCell(x, y, Math.random() > 0.5 ? alive_color : dead_color);
    }
  }
  copyToPrimary();
}
export function init(width: u32, height: u32): void {
  ...
  fillUniverse();
```

Since `Math.random` is used to determine the initial state of a cell, the WebAssembly module needs a seed function to derive a random number from.

AssemblyScript provides a convenient module loader that does this and a lot more, like wrapping the browser APIs for module loading and providing functions for more fine-grained memory control. You will not be using it here since it abstracts away many details that would otherwise help in learning the inner workings of WebAssembly, so pass in a seed function instead:

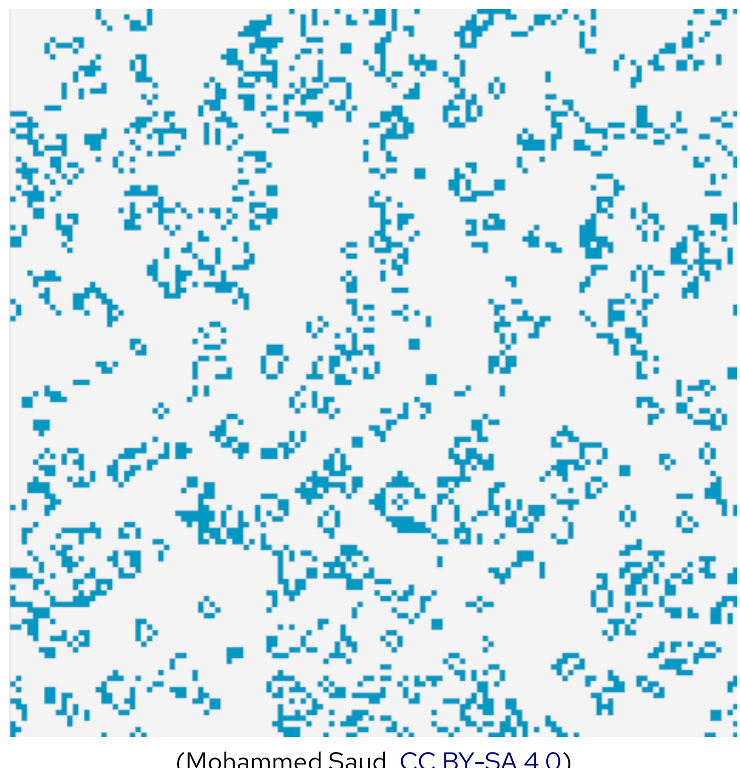
```
  const importObject = {
    env: {
      seed: Date.now,
      abort: () => console.log('aborting!')
    }
```

```
  };
  const module = await
WebAssembly.instantiateStreaming(fetch('./build/optimized.wasm'), importObject);
```

`instantiateStreaming` can be called with an optional second parameter, an object that exposes JavaScript functions to WebAssembly modules. Here, use `Date.now` as the seed to generate random numbers.

It's now possible to run the `fillUniverse` function and finally have life on your grid!

You can also play around with different `WIDTH`, `HEIGHT`, and `FPS` values and use different cell colors.



(Mohammed Saud, CC BY-SA 4.0)

Try the game

If you use large sizes, make sure to grow the memory accordingly.

Here's the complete code.

Writing WebAssembly for speed and code reuse

By Marty Kalin

Imagine translating a non-web application, written in a high-level language, into a binary module ready for the web. This translation could be done without any change whatsoever to the non-web application's source code. A browser can download the newly translated module efficiently and execute the module in the sandbox. The executing web module can interact seamlessly with other web technologies—with JavaScript (JS) in particular. Welcome to WebAssembly.

As befits a language with *assembly* in the name, WebAssembly is low-level. But this low-level character encourages optimization: the just-in-time (JIT) compiler of the browser's virtual machine can translate portable WebAssembly code into fast, platform-specific machine code. A WebAssembly module thereby becomes an executable suited for compute-bound tasks such as number crunching.

Which high-level languages compile into WebAssembly? The list is growing, but the original candidates were C, C++, and Rust. Let's call these three the *systems languages*, as they are meant for systems programming and high-performance applications programming. The systems languages share two features that suit them for compilation into WebAssembly. The next section gets into the details, which sets up full code examples (in C and TypeScript) together with samples from WebAssembly's own *text format* language.

Explicit data typing and garbage collection

The three systems languages require explicit data types such as **int** and **double** for variable declarations and values returned from functions. For example, the following code segment illustrates 64-bit addition in C:

```
long n1 = random();
long n2 = random();
long sum = n1 + n2;
```

The library function **random** is declared with **long** as the return type:

```
long random(); /* returns a long */
```

During the compilation process, C source is translated into assembly language, which is then translated into machine code. In Intel assembly language (AT&T flavor), the last C statement above would be something like the following (with **##** introducing comments):

```
addq %rax, %rdx ## %rax = %rax + %rdx (64-bit addition)
```

The **%rax** and **%rdx** are 64-bit registers, and the **addq** instruction means *add quadwords*, where a *quadword* is 64 bits in size, which is the standard size for a C **long**. The assembly language underscores that executable machine code involves types, with the type given through some mix of the instruction and the arguments, if any. In this case, the *add* instruction is **addq** (64-bit addition) rather than, for example, **addl**, which adds 32-bit values typical of a C **int**. The registers in use are the full 64-bit ones (the **r** in **%rax** and **%rdx**) rather than 32-bit chunks thereof (e.g., **%eax** is the lower 32 bits of **%rax**, and **%edx** is the lower 32 bits of **%rdx**).

The assembly-language addition performs well because the operands are stored in CPU registers, and a reasonable C compiler (at even the default level of optimization) would generate assembly code equivalent to what is shown here.

The three systems languages, with their emphasis on explicit types, are good candidates for compilation into WebAssembly because this language, too, has explicit data types: **i32** for a 32-bit integer value, **f64** for a 64-bit floating-point value, and so on.

Explicit data types encourage optimization for function calls as well. A function with explicit data types has a *signature*, which specifies the data types for the arguments and the value, if any, returned from the function. Below is the signature for a WebAssembly function named **$add**, which is written in the WebAssembly text format language discussed below. The function takes two 32-bit integers as arguments and returns a 64-bit integer:

```
(func $add (param $lhs i32) (param $rhs i32) (result i64))
```

The browser's JIT compiler should have the 32-bit integer arguments and the returned 64-bit value stored in registers of the appropriate sizes.

When it comes to high-performance web code, WebAssembly is not the only game in town. For example, **asm.js** is a JS dialect designed, like WebAssembly, to approach native speed. The asm.js dialect invites optimization because the code mimics the explicit data types in the three aforementioned languages. Here's an example with C and then asm.js. The sample function in C is:

```
int f(int n) {       /** C **/
  return n + 1;
}
```

Both the parameter **n** and the returned value are explicitly typed as **int**. The equivalent function is asm.js would be:

```
function f(n) {      /** asm.js **/
  n = n | 0;
  return (n + 1) | 0;
}
```

JS, in general, does not have explicit data types, but a bitwise-OR operation in JS yields an integer value. This explains the otherwise pointless bitwise-OR operation:

```
n = n | 0;  /* bitwise-OR of n and zero */
```

The bitwise-OR of **n** and zero evaluates to **n**, but the purpose here is to signal that **n** holds an *integer* value. The **return** statement repeats this optimizing trick.

Among the JS dialects, TypeScript stands out for adopting explicit data types, which makes this language attractive for compilation into WebAssembly. (A code example below illustrates this.)

The second feature shared by the three systems languages is that they execute without a garbage collector (GC). For dynamically allocated memory, the Rust compiler automatically writes both the allocation and the deallocation code; in the other two systems languages, the programmer who dynamically allocates memory is responsible for explicitly deallocating this same memory. The systems languages avoid the overhead and complication of automated GC.

This quick overview of WebAssembly can be summarized as follows. Almost any article on the WebAssembly language mentions near-native speed as one of the language's major goals. The *native speed* is that of the compiled systems' languages; hence, these three languages were also the originally designated candidates for compilation into WebAssembly.

WebAssembly, JavaScript, and the separation of concerns

All rumors to the contrary, the WebAssembly language is not designed to replace JS, but rather to supplement JS by providing better performance on compute-bound tasks. WebAssembly also has an advantage when it comes to downloading. A browser fetches a JS module as text, an inefficiency that WebAssembly addresses. A module in WebAssembly has a compact binary format, which speeds up downloading.

Of equal interest is how JS and WebAssembly are meant to work together. JS is designed to read and write the Document Object Model (DOM), the tree representation of a web page. By contrast, WebAssembly does not come with any built-in functionality for the DOM; but WebAssembly can export functions that JS can then call as needed. This separation of concerns means a clean division of labor:

```
DOM<----->JS<----->WebAssembly
```

JS, in whatever dialect, still should manage the DOM, but JS also can use the general-purpose functionality delivered through WebAssembly modules. A code example helps illustrate the division of labor. (The code examples in this article are available in a ZIP file on my website.)

Hailstone sequences and the Collatz conjecture

A production-grade example would have WebAssembly code perform a heavy compute-bound task such as generating large cryptographic key pairs or using such pairs for encryption and decryption. A simpler example fits the bill as a stand-in that is easy to follow. There is number crunching, but of the routine sort that JS could handle with ease.

Consider the function **hstone** (for *hailstone*), which takes a positive integer as an argument. The function is defined as follows:

```
             3N + 1 if N is odd
```

```
hstone(N) =
             N/2 if N is even
```

For example, **hstone(12)** returns six, whereas **hstone(11)** returns 34. If N is odd, then 3N+1 is even; but if N is even, then N/2 could be either even (e.g., 4/2 = 2) or odd (e.g., 6/2 = 3).

The **hstone** function can be used iteratively by passing the returned value as the next argument. The result is a *hailstone sequence* such as this one, which starts with 24 as the original argument, the returned value 12 as the next argument, and so on:

```
24,12,6,3,10,5,16,8,4,2,1,4,2,1,...
```

It takes 10 calls for the sequence to converge to one, at which point the sequence of 4,2,1 repeats indefinitely: (3x1)+1 is 4, which is halved to yield two, which is halved to yield one, and so on. **Plus** magazine offers an explanation of why *hailstone* seems an appropriate name for such sequences.

Note that powers of two converge quickly, requiring just N divisions by two to reach one; for example, 32 = 2^5 has a convergence length of five, and 64 = 2^6 has a convergence length of six. Of interest here is the sequence length from the initial argument to the *first* occurrence of one. My code examples in C and TypeScript compute the length of a hailstone sequence.

The Collatz conjecture is that a hailstone sequence converges to one no matter what the initial argument **N > 0** happens to be. No one has found a counterexample to the Collatz conjecture, nor has anyone found a proof to elevate the conjecture to a theorem. The conjecture, simple as it is to test with a program, remains a profoundly challenging problem in mathematics.

From C to WebAssembly in one step

The *hstoneCL* program below is a non-web application that can be compiled with a regular C compiler (e.g., GNU or Clang). The program generates a random integer value **N > 0** eight times and computes the length of the hailstone sequence starting with N. Two programmer-defined functions, **main** and **hstone**, are of interest when the app is later compiled into WebAssembly.

Example 1. The hstone function in C

```
#include <stdio.h>
#include <stdlib.h>
```

```
#include <time.h>
int hstone(int n) {
  int len = 0;
  while (1) {
    if (1 == n) break;           /* halt on 1 */
    if (0 == (n & 1)) n = n / 2; /* if n is even */
    else n = (3 * n) + 1;        /* if n is odd  */
    len++;                       /* increment counter */
  }
  return len;
}
#define HowMany 8
int main() {
  srand(time(NULL));  /* seed random number generator */
  int i;
  puts("  Num  Steps to 1");
  for (i = 0; i < HowMany; i++) {
    int num = rand() % 100 + 1; /* + 1 to avoid zero */
    printf("%4i %7i\n", num, hstone(num));
  }
  return 0;
}
```

The code can be compiled and run from the command line (with **%** as the command-line prompt) on any Unix-like system:

```
% gcc -o hstoneCL hstoneCL.c  ## compile into executable hstoneCL
% ./hstoneCL                  ## execute
```

Here is the output from a sample run:

```
Num  Steps to 1
  88      17
   1       0
  20       7
  41     109
  80       9
  84       9
  94     105
  34      13
```

The systems languages, including C, require specialized toolchains to translate source code into a WebAssembly module. For the C/C++ languages, Emscripten is a pioneering and still widely used option, one built upon the well-known LLVM (Low-Level Virtual Machine)

compiler infrastructure. My examples in C use Emscripten, which you can [install with this guide](#)).

The *hstoneCL* program can be webified by using Emscription to compile the code—with no change whatsoever—into a WebAssembly module. The Emscription toolchain also creates an HTML page together with JS *glue* (in asm.js) that mediates between the DOM and the WebAssembly module that computes the **hstone** function. Here are the steps:

1. Compile the non-web program *hstoneCL* into WebAssembly:

   ```
   % emcc hstoneCL.c -o hstone.html  ## generates hstone.js and hstone.wasm as well
   ```

   The file *hstoneCL.c* contains the source code shown above, and the **-o** for *output* flag specifies the name of the HTML file. Any name would do, but the generated JS code and the WebAssembly binary file then have the same name (in this case, *hstone.js* and *hstone.wasm*, respectively). Older versions of Emscription (prior to 13) may require the flag **-s WASM=1** to be included in the compilation command.

2. Use the Emscription development web server (or equivalent) to host the webified app:

   ```
   % emrun --no_browser --port 9876 .   ## . is current working directory, any port number you like
   ```

   To suppress warning messages, the flag **--no_emrun_detect** can be included. This command starts the web server, which hosts all the resources in the current working directory; in particular, *hstone.html*, *hstone.js*, and *hstone.webasm*.

3. Open a WebAssembly-enabled browser (e.g., Chrome or Firefox) to the URL *http://localhost:9876/hstone.html*.

This screenshot shows the output from my sample run with Firefox.

```
Num  Steps to 1
81       22
45       16
34       13
81       22
99       25
97      118
79       35
21        7
```

Figure 1. The webified *hstone* program

The result is remarkable, as the full compilation process requires but a single command and no change whatsoever to the original C program.

Fine-tuning the hstone program for webification

The Emscription toolchain nicely compiles a C program into a WebAssembly module and generates the required JS glue, but these artifacts are typical for machine-generated code. For example, the asm.js file produced is almost 100KB in size. The JS code handles multiple scenarios and does not use the most recent WebAssembly API. A simplified version of the webified *hstone* program will make it easier to focus on how the WebAssembly module (housed in the *hstone.wasm* file) interacts with the JS glue (housed in the *hstone.js* file).

There is another issue: WebAssembly code need not mirror the functional boundaries in a source program such as C. For example, the C program *hstoneCL* has two user-defined functions, **main** and **hstone**. The resulting WebAssembly module exports a function named **_main** but does not export a function named **_hstone**. (It's worth noting that the function **main** is the entry point in a C program.) The body of the C **hstone** function might be in some unexported function or simply wrapped into **_main**. The exported WebAssembly functions are

exactly the ones that the JS glue can invoke by name. However, there is a directive to specify which source-language functions should be exported by name in the WebAssembly code.

Example 2. The revised hstone program

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <emscripten/emscripten.h>
int EMSCRIPTEN_KEEPALIVE hstone(int n) {
  int len = 0;
  while (1) {
    if (1 == n) break;           /* halt on 1 */
    if (0 == (n & 1)) n = n / 2; /* if n is even */
    else n = (3 * n) + 1;        /* if n is odd  */
    len++;                       /* increment counter */
  }
  return len;
}
```

The revised *hstoneWA* program, shown above, has no **main** function; it is no longer needed because the program is not designed to run as a standalone application but exclusively as a WebAssembly module with a single exported function. The directive **EMSCRIPTEN_KEEPALIVE** (defined in the header file *emscripten.h*) instructs the compiler to export an **_hstone** function in the WebAssembly module. The naming convention is straightforward: A C function such as **hstone** retains its name—but with a single underscore as its first character in WebAssembly (**_hstone**, in this case). Other compilers into WebAssembly follow different naming conventions.

To confirm that this approach works, the compilation step can be simplified to produce only the WebAssembly module and the JS glue, but not the HTML:

```
% emcc hstoneWA.c -o hstone2.js  ## we'll provide our own HTML file
```

The HTML file now can be simplified to this hand-written one:

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8"/>
    <script src="https://opensource.com/hstone2.js"></script>
  </head>
  <body/>
</html>
```

The HTML document loads the JS file, which in turn fetches and loads the WebAssembly binary file *hstone2.wasm*. By the way, the new WASM file is about half the size of the original example.

The application code can be compiled as before and then launched with the built-in web server:

```
% emrun --no_browser --port 7777 .  ## new port number for emphasis
```

After requesting the revised HTML document in a browser (in this case, Chrome), the browser's web console can be used to confirm that the **hstone** function has been exported as **_hstone**. Here is a segment of my session in the web console, with **##** again introducing comments:

```
> _hstone(27)   ## invoke _hstone by name
< 111           ## output
> _hstone(7)    ## again
< 16            ## output
```

The **EMSCRIPTEN_KEEPALIVE** directive is the straightforward way to have the Emscripten compiler produce a WebAssembly module that exports any function of interest to the JS glue, which this compiler likewise produces. A customized HTML document, with whatever hand-crafted JS is appropriate, can then call the functions exported from the WebAssembly module. Hats off to Emscripten for this clean approach.

Compiling TypeScript into WebAssembly

The next code example is in TypeScript, which is JS with explicit data types. The setup requires Node.js and its *npm* package manager. The following npm command installs AssemblyScript, which is a WebAssembly compiler for TypeScript code:

```
% npm install -g assemblyscript ## install the AssemblyScript compiler
```

The TypeScript program *hstone.ts* consists of a single function, again named **hstone**. Data types such as **i32** (32-bit integer) now follow rather than precede parameter and local variable names (in this case, **n** and **len**, respectively):

```
export function hstone(n: i32): i32 { // exported in WebAssembly
  let len: i32 = 0;
  while (true) {
```

```
    if (1 == n) break;            // halt on 1
    if (0 == (n & 1)) n = n / 2;  // if n is even
    else n = (3 * n) + 1;         // if n is odd
    len++;                        // increment counter
  }
  return len;
}
```

The function **hstone** takes one argument of type **i32** and returns a value of the same type. The function's body is essentially the same as in the C example. The code can be compiled into WebAssembly as follows:

```
% asc hstone.ts -o hstone.wasm  ## compile a TypeScript file into WebAssembly
```

The WASM file *hstone.wasm* is only about 14KB in size.

To highlight details of how a WebAssembly module is loaded, the hand-written HTML file below (**index.html** in the ZIP on my website) includes the script to fetch and load the WebAssembly module *hstone.wasm* and then to instantiate this module so that the exported **hstone** function can be invoked for confirmation in the browser's console.

Example 3. The HTML page for the TypeScript code

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8"/>
    <script>
      fetch('hstone.wasm').then(response =>            <!-- Line 1 -->
      response.arrayBuffer()                           <!-- Line 2 -->
      ).then(bytes =>                                  <!-- Line 3 -->
      WebAssembly.instantiate(bytes, {imports: {}})    <!-- Line 4 -->
      ).then(results => {                              <!-- Line 5 -->
      window.hstone = results.instance.exports.hstone; <!-- Line 6 -->
      });
    </script>
  </head>
  <body/>
</html>
```

The script element in the HTML page above can be clarified line by line. The **fetch** call in Line 1 uses the Fetch module to get the WebAssembly module from the web server that hosts the HTML page. When the HTTP response arrives, the WebAssembly module does so as a sequence of bytes, which are stored in the **arrayBuffer** of the script's Line 2. These bytes

Creative Commons Attribution Share-alike 4.0

constitute the WebAssembly module, which is all of the code compiled from the TypeScript file. This module has no imports, as indicated at the end of Line 4.

At the start of Line 4, the WebAssembly module is instantiated. A WebAssembly module is akin to a non-static class with non-static members in an object-oriented language such as Java. The module contains variables, functions, and various support artifacts; but the module, like the non-static class, must be instantiated to be usable, in this case in the web console, but more generally in the appropriate JS glue code.

The script's Line 6 exports the original TypeScript function **hstone** under the same name. This WebAssembly function is available now to any JS glue code, as another session in the browser's console will confirm.

WebAssembly has a more concise API for fetching and instantiating a module. The new API reduces the script above to only the *fetch* and *instantiate* operations. The longer version shown here has the benefit of exhibiting details; in particular, the representation of a WebAssembly module as a byte array that gets instantiated as an object with exported functions.

The plan is to have a web page load a WebAssembly module in the same way as a JS ES2015 module:

```
<script type='module'>...</script>
```

JS then would fetch, compile, and otherwise handle the WebAssembly module as if it were just another JS module.

The text format language

WebAssembly binaries can be translated to and from *text format* equivalents. The binaries usually reside in files with a WASM extension, whereas their human-readable text counterparts reside in files with a WAT extension. WABT is a set of nearly a dozen tools for dealing with WebAssembly, including ones to translate to and from formats such as WASM and WAT. The conversion tools include the *wasm2wat*, *wasm2c*, and *wat2wasm* utilities.

The text-format language adopts the S-expression (*S* for *symbolic*) syntax popularized by Lisp. An S-expression (*sexpr* for short) represents a tree as a list with arbitrarily many sublists. For example, this sexpr occurs near the end of the WAT file for the TypeScript example:

```
(export "hstone" (func $hstone)) ## export function $hstone by the name "hstone"
```

The tree representation is:

```
        export        ## root
          |
     +----+----+
     |         |
  "hstone"    func    ## left and right children
               |
            $hstone   ## single child
```

In text format, a WebAssembly module is a sexpr whose first term is **module**, which is the root of the tree. Here is an elementary example of a module that defines and exports a single function, which takes no arguments but returns the constant 9876:

```
(module
  (func (result i32)
    (i32.const 9876)
  )
  (export "simpleFunc" (func 0)) // 0 is the unnamed function's index
)
```

The function is defined without a name (i.e., as a lambda) and exported by referencing its index 0, which is the index of the first nested sexpr in the module. The export name is given as a string; in this case, "simpleFunc."

Functions in text format have a standard pattern, which can be depicted as follows:

```
(func <signature> <local vars> <body>)
```

The signature specifies the arguments (if any) and the returned value (if any). For example, here's the signature for an unnamed function that takes two 32-bit integer arguments but returns a 64-bit integer value:

```
(func (param i32) (param i32) (result i64)...)
```

Names can be given to functions, arguments, and local variables. They start with a dollar sign:

```
(func $foo (param $a1 i32) (param $a2 f32) (local $n1 f64)...)
```

The body of a WebAssembly function reflects the underlying *stack machine* architecture of the language. Stack storage is for scratchpad. Consider this example of a function that doubles its integer argument and returns the value:

```
(func $doubleit (param $p i32) (result i32)
  get_local $p
  get_local $p
  i32.add)
```

Each of the **get_local** operations, which can work on local variables and parameters alike, pushes the 32-bit integer argument onto the stack. The **i32.add** operation then pops the top two (and currently only) values from the stack to perform the addition. The sum from the add operation is then the one and only value on the stack and thereby becomes the value returned from the **$doubleit** function.

When the WebAssembly code is translated into machine code, the WebAssembly stack as scratchpad should be replaced, wherever possible, by general-purpose registers. This is the job for the JIT compiler, which translates WebAssembly virtual stack-machine code into real-machine code.

Web programmers are unlikely to write WebAssembly in text format, as compiling from some high-level language is far too attractive an option. Compiler writers, by contrast, might find it productive to work at this fine-grained level.

Wrapping up

Much has been made of WebAssembly's goal of achieving near-native speed. But as the JIT compilers for JS continue to improve, and as dialects well-suited for optimization (e.g., TypeScript) emerge and evolve, it may be that JS also achieves near-native speed. Would this imply that WebAssembly is wasted effort? I think not.

WebAssembly addresses another traditional goal in computing: meaningful code reuse. As even the short examples in this article illustrate, code in a suitable language, such as C or TypeScript, translates readily into a WebAssembly module, which plays well with JS code—the glue that connects a range of technologies used in the web. WebAssembly is thus an inviting way to reuse legacy code and to broaden the use of new code. For example, a high-performance program for image processing, written originally as a desktop application, might also be useful in a web application. WebAssembly then becomes an attractive path to reuse. (For new web modules that are compute-bound, WebAssembly is a sound choice.) My hunch is that WebAssembly will thrive as much for reuse as for performance.

Creative Commons Attribution Share-alike 4.0

Stepping WebAssembly up a notch with security

By Mike Bursell

WebAssembly (also known as Wasm) is taking the world by storm. It started off as technology for browsers, "doing JavaScript right," but has developed into so much more than that. It provides a platform-independent runtime with binaries that can be compiled from many different languages and run (without any further changes or recompilation) on any platform with runtime support. In this article, I will explore a new step toward securely running WebAssembly.

Securing WebAssembly outside the browser

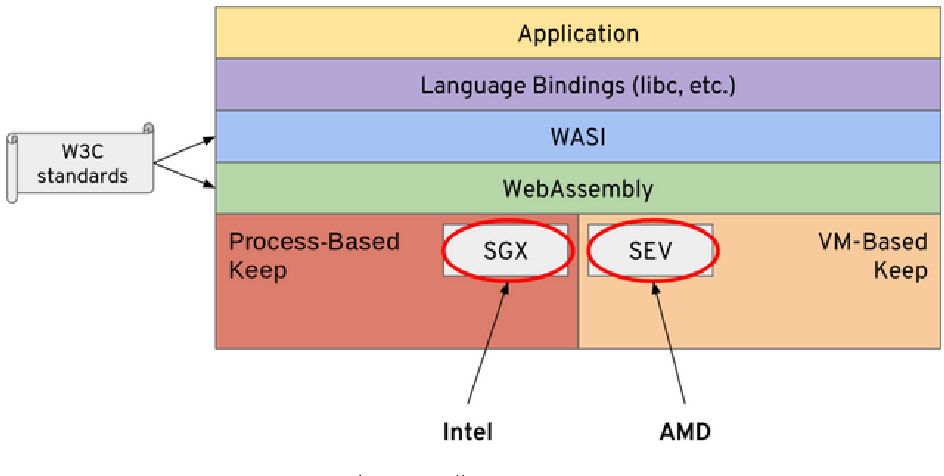
What's particularly interesting for many of us is the rise of WebAssembly System Interface (WASI), a way of running WebAssembly *outside* browsers and in classic server contexts.

WebAssembly and WASI already have some great security features, but I'm involved in a project called Enarx (completely open source, of course!) where we're looking to improve security for workloads in such a way that even the administrator, kernel, or hypervisor can't look inside them or impact their integrity. You will be able to run your workloads on hosts that you don't even trust, apart from the CPU and associated firmware!

We chose WebAssembly as our runtime because it offers the sort of platform-independence and easy integration into existing development environments that we value and that we believe developers and enterprises are looking for when they're designing and deploying workloads that include sensitive material, whether that's data or algorithms. WASI, in particular, turns out to be an excellent fit for the Trusted Execution Environments (TEEs) that we're targeting for running what we call "Keeps": the runtime environments for your workloads.

Hardware abstraction with WebAssembly

It is, however, a mammoth job, particularly as we're abstracting away the underlying processor architectures (currently two: Intel's SGX and AMD's SEV), so that you, the user, don't need to worry about them—all you need to do is write and compile your application (to WebAssembly, of course!), then request that it be deployed. Enarx, then, has lots of moving parts, and one of the key tasks for us has been to start the work to abstract away the underlying processor architectures so that we can prepare the runtime layers on top. Here's a general picture of the software layers and how they sit on top of the hardware platforms:



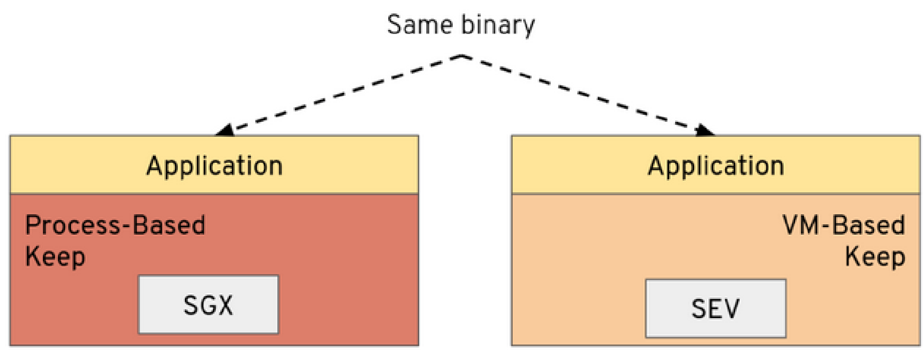
(Mike Bursell, [CC BY-SA 4.0](https://creativecommons.org/licenses/by-sa/4.0/))

Let's step back and examine those layers briefly:

- **Application:** This is what you write, in whichever language you chose (C, C++, Rust, Go, Python, Java, Haskell…) and is the whole point of the project.
- **Language bindings:** When you compile your application into WebAssembly, the compiler that you use does all the work to ensure your application is compiled correctly for WebAssembly and pulls along any pieces it needs.
- **WASI (WebAssembly System Interface):** This is the implementation that allows your WebAssembly application to execute in a server-type context rather than in a browser (which was the initial implementation target for WebAssembly). WASI happens to fit the capabilities offered by TEEs pretty well.
- **WebAssembly:** This is the platform-specific implementation of Wasm that provides the key runtime pieces for the layers above it.
- **Process-based Keep/VM-based Keep:** This is where lots of the implementation of Enarx is happening now: providing a base layer for the various hardware options

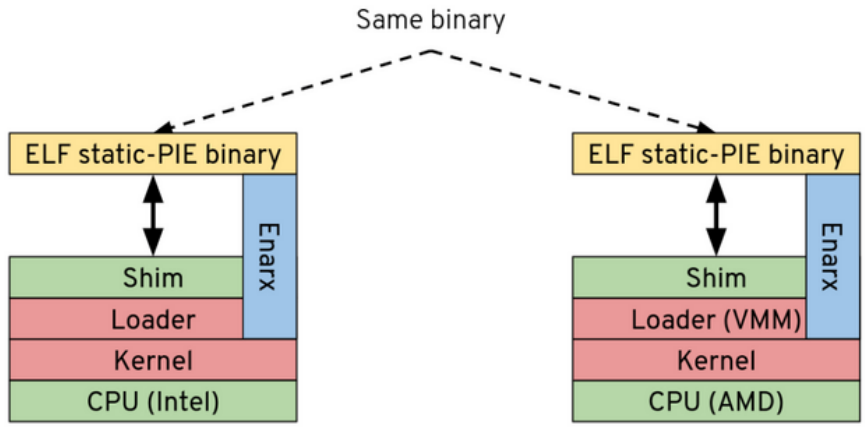
(currently, we're focusing on Intel's SGX and AMD's SEV) on which the WebAssembly layer can sit and execute.

What we announced—and demoed—at the end of March at Red Hat Summit is that Enarx now has an initial implementation of code to allow us to abstract away process-based and virtual machine-based types of architecture (with examples for SGX and SEV), so we can do this:



(Mike Bursell, CC BY-SA 4.0)

This seems deceptively simple, but what's going on under the covers is more than what is exposed in the picture above. The reality is more like this:



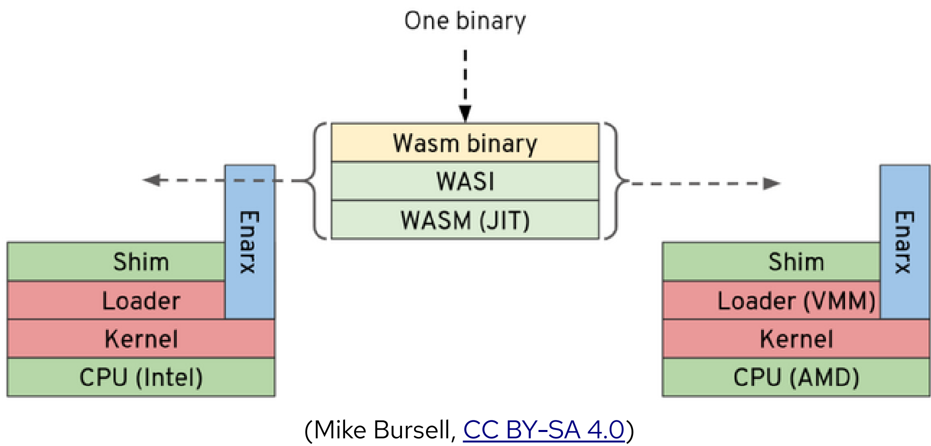
(Mike Bursell, CC BY-SA 4.0)

This gives more detail: the application that's running on both architectures (SGX on the left, SEV on the right) is the very same Linux binary (in fact, to be specific, it's an ELF static-PIE binary - but that's not particularly important at this level of detail). To be clear, this is not only the same source code, compiled for different platforms, but exactly the same binary with the very same hash signature. What's pretty astounding about this is that, in order to make it run on both platforms, the engineering team had to write two sets of seriously low-level code,

including more than a little Assembly language providing the "plumbing" to allow the binary to run on both.

Securing WebAssembly one syscall at a time

This is a very big deal because, although we've only implemented a handful of syscalls on each platform—enough to make our simple binary run and print out a message, we now have a framework on which we know we can build. And what's next? Well, we need to expand that framework so that we can build the WebAssembly layers that will allow WebAssembly applications to run on top:



(Mike Bursell, CC BY-SA 4.0)

There's a long way to go, but this milestone shows that we have an initial framework that we can improve and on which we can build.

What's next?

Enarx is open source. We do all of our design work in the open, along with our daily stand-ups. You're welcome to browse our documentation, RFCs (mostly in draft at the moment), raise issues, and join our calls. You can find loads more information on the Enarx wiki, and we look forward to your involvement in the project.

As I explained at the beginning of this article, the goal is to support WebAssembly binaries, giving you—a developer or enterprise wanting to deploy your applications—a hardware-protected runtime environment where you can execute your workloads without having to trust the host on which they're running. If you're interested in this, please get in touch; we'd love to hear from you.

Create web user interfaces with Qt WebAssembly instead of JavaScript

By Stephan Avenwedde

When I first heard about WebAssembly and the possibility of creating web user interfaces with Qt, just like I would in ordinary C++, I decided to take a deeper look at the technology.

My open source project Pythonic is completely Python-based (PyQt), and I use C++ at work; therefore, this minimal, straightforward WebAssembly tutorial uses Python on the backend and C++ Qt WebAssembly for the frontend. It is aimed at programmers who, like me, are not familiar with web development.

TL;DR

```
git clone https://github.com/hANSIc99/wasm_qt_example
cd wasm_qt_example
python mysite.py
```

Then visit http://127.0.0.1:7000 with your favorite browser.

What is WebAssembly?

WebAssembly (often shortened to Wasm) is designed primarily to execute portable binary code in web applications to achieve high-execution performance. It is intended to coexist with JavaScript, and both frameworks are executed in the same sandbox. Recent performance benchmarks showed that WebAssembly executes roughly 10–40% faster, depending on the browser, and given its novelty, we can still expect improvements. The downside of this great execution performance is its widespread adoption as the preferred malware language. Crypto

miners especially benefit from its performance and harder detection of evidence due to its binary format.

Toolchain

There is a getting started guide on the Qt wiki. I recommend sticking exactly to the steps and versions mentioned in this guide. You may need to select your Qt version carefully, as different versions have different features (such as multi-threading), with improvements happening with each release.

To get executable WebAssembly code, simply pass your Qt C++ application through Emscripten. Emscripten provides the complete toolchain, and the build script couldn't be simpler:

```
#!/bin/sh
source ~/emsdk/emsdk_env.sh
~/Qt/5.13.1/wasm_32/bin/qmake
make
```

Building takes roughly 10 times longer than with a standard C++ compiler like Clang or g++. The build script will output the following files:

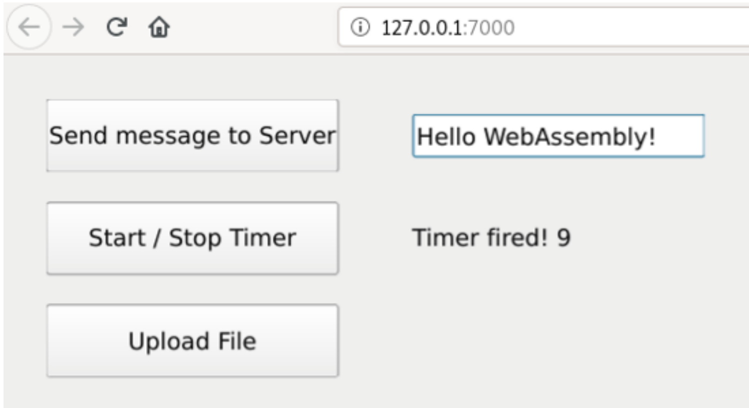
- WASM_Client.js
- WASM_Client.wasm
- qtlogo.svg
- qtloader.js
- WASM_Client.html
- Makefile (intermediate)

The versions on my (Fedora 30) build system are:

- emsdk: 1.38.27
- Qt: 5.13.1

Frontend

The frontend provides some functionalities based on WebSocket.



- **Send message to server:** Send a simple string message to the server with a WebSocket. You could have done this also with a simple HTTP POST request.
- **Start/stop timer:** Create a WebSocket and start a timer on the server to send messages to the client at a regular interval.
- **Upload file:** Upload a file to the server, where the file is saved to the home directory (**~/**) of the user who runs the server.

If you adapt the code and face a compiling error like this:

```
error: static_assert failed due to
 requirement ‘bool(-1 == 1)’ “Required feature http for file
 ../../Qt/5.13.1/wasm_32/include/QtNetwork/qhttpmultipart.h not available.”
QT_REQUIRE_CONFIG(http);
```

it means that the requested feature is not available for Qt Wasm.

Backend

The server work is done by Eventlet. I chose Eventlet because it is lightweight and easy to use. Eventlet provides WebSocket functionality and supports threading.

```
stephan@localhost:~/Dokumente/wasm_qt_example

 6
 7
 8 @websocket.WebSocketWSGI
 9 def startTimer(ws):
10     n_cnt = 0
11     global execTimer
12     while execTimer:
13         print('Timer fired! {}'.format(n_cnt))
14
15         greenthread.sleep(1)
16         n_cnt+=1
17
18         try:
19             ws.send('Timer fired! {}'.format(n_cnt))
20         except Exception as e:
21             print('Client websocket not available')
22             ws.close()
23             return
24
25 @websocket.WebSocketWSGI
26 def processMessage(ws):
27     m = ws.wait()
28     print('Message received: {}'.format(m))
29
30
31 @websocket.WebSocketWSGI
32 def saveData(ws):
NORMAL  mysite.py          startTimer  python  utf-8[unix]  21% ☰  25/118 ln :  1
```

Inside the repository under **mysite/template**, there is a symbolic link to **WASM_Client.html** in the root path. The static content under **mysite/static** is also linked to the root path of the repository. If you adapt the code and do a recompile, you just have to restart Eventlet to update the content to the client.

Eventlet uses the Web Server Gateway Interface for Python (WSGI). The functions that provide the specific functionality are extended with decorators.

Please note that this is an absolute minimum server implementation. It doesn't implement any multi-user capabilities – every client is able to start/stop the timer, even for other clients.

Conclusion

Take this example code as a starting point to get familiar with WebAssembly without wasting time on minor issues. I don't make any claims for completeness nor best-practice integration. I walked through a long learning curve until I got it running to my satisfaction, and I hope this gives you a brief look into this promising technology.

Why should you use Rust in WebAssembly?

By Ryan Levick

WebAssembly (Wasm) is a technology that has the chance to reshape how we build apps for the browser. Not only will it allow us to build whole new classes of web applications, but it will also allow us to make existing apps written in JavaScript even more performant.

What is WebAssembly?

WebAssembly is a binary file format that all major browsers (with the exception of IE 11) have implemented for virtual machines to run. WebAssembly has the ability to start and run much quicker than JavaScript because the binary format is simple and easy for browsers to parse and run in a highly optimized way. If you're interested in the technical details of what makes WebAssembly special, I recommend Lin Clark's posts on the subject.

So, why use it?

Although I originally began looking into WebAssembly as a means to write Rust in another environment (the browser), that's not really what makes WebAssembly special. I enjoy writing JavaScript (and especially TypeScript), and the ecosystem around web development built in JavaScript is a huge asset that shouldn't be just thrown away. WebAssembly is at its best when considered as a supplement to JavaScript—filling in when JavaScript isn't quite performant enough.

WebAssembly can be used to write entire web applications or to replace small bits of existing applications that might not be performant enough with something that runs at near native speed. Also, because WebAssembly is a native-like assembly format, many languages can be

Creative Commons Attribution Share-alike 4.0

compiled down to it, meaning sharing code between other platforms and the web is now much more practical.

Other languages

Many different languages can be compiled down to WebAssembly, including C# and Go, so why not use them instead of Rust? While use of a programming language is always influenced by personal preference, there are many reasons why Rust is the best tool for the job. Because these languages have large runtimes that must be included in the WebAssembly binary, they're only really practical for greenfield projects (i.e., they're practical only as JavaScript replacements). This Go wiki article on Wasm says the smallest achievable binary size uncompressed is around 2MB; this mirrors what I've seen. For Rust, which ships with an extremely minimal runtime (basically just an allocator), the "hello, world" example compiles to 1.6KB on my machine without any post-compile size optimizations (which could bring it down further).

This is not to say that the future of Go or C# in the browser is bleak—I'm quite excited about what might come from those efforts. But the reality is that these technologies will probably always be best for greenfield projects.

C and C++ ship with very small runtimes, just like Rust, and thus can be practical for embedding inside existing apps and libraries. However, Rust makes it really easy to create WebAssembly binaries that have fairly idiomatic JavaScript interfaces using the tools we'll explore in the other articles in this series, while the process in C and C++ is much more manual. The tooling in Rust is absolutely fantastic, and I think it makes the entire experience much more enjoyable. Rust is also a much more memory safe language meaning that a whole class of bugs that are common in C and C++ are impossible to have in safe Rust. If you're used to memory safe languages like JavaScript, Java, and C#, (and even if you're not), you probably want to go with Rust.

Let's keep going!

If you're interested in WebAssembly, I encourage you to dive into whichever WebAssembly-supported language makes you happiest.