WebAssembly

IF3110 – Web-based Application Development

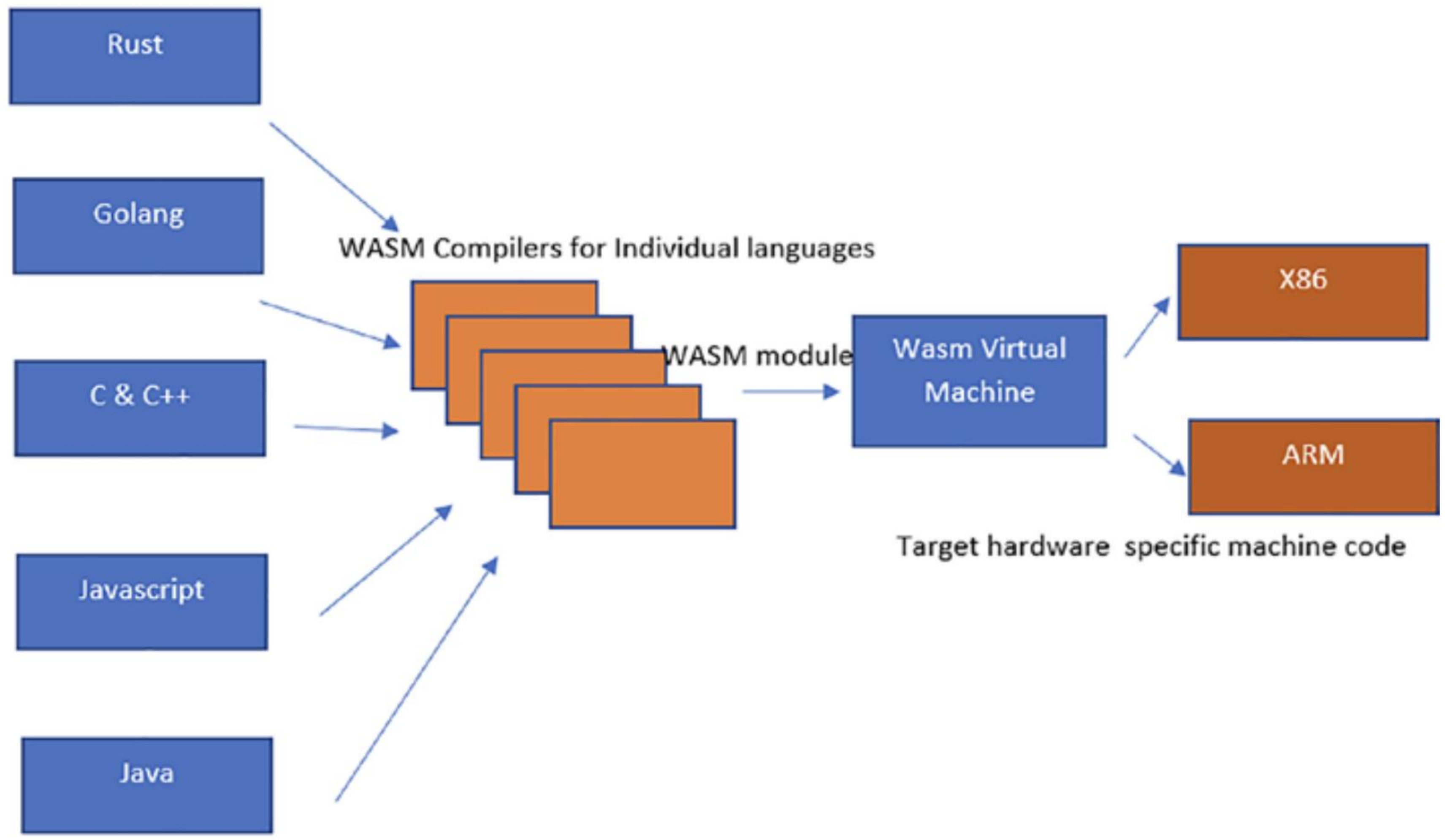
Reference

- https://webassembly.org/
- https://developer.mozilla.org/en-US/docs/WebAssembly

Definition

- WebAssembly (WASM) is a low-level, binary instruction format designed to run code on the web at near-native speed.
- Key Features
  - Platform-independent.
  - Web compatibility (runs in the browser alongside JavaScript).
  - Secure and sandboxed execution.
  - Designed for performance-critical applications.

Architecture


Rust
Golang
C & C++
Javascript
Java
WASM Compilers for Individual languages
WASM module
Wasm Virtual Machine
X86
ARM
Target hardware specific machine code

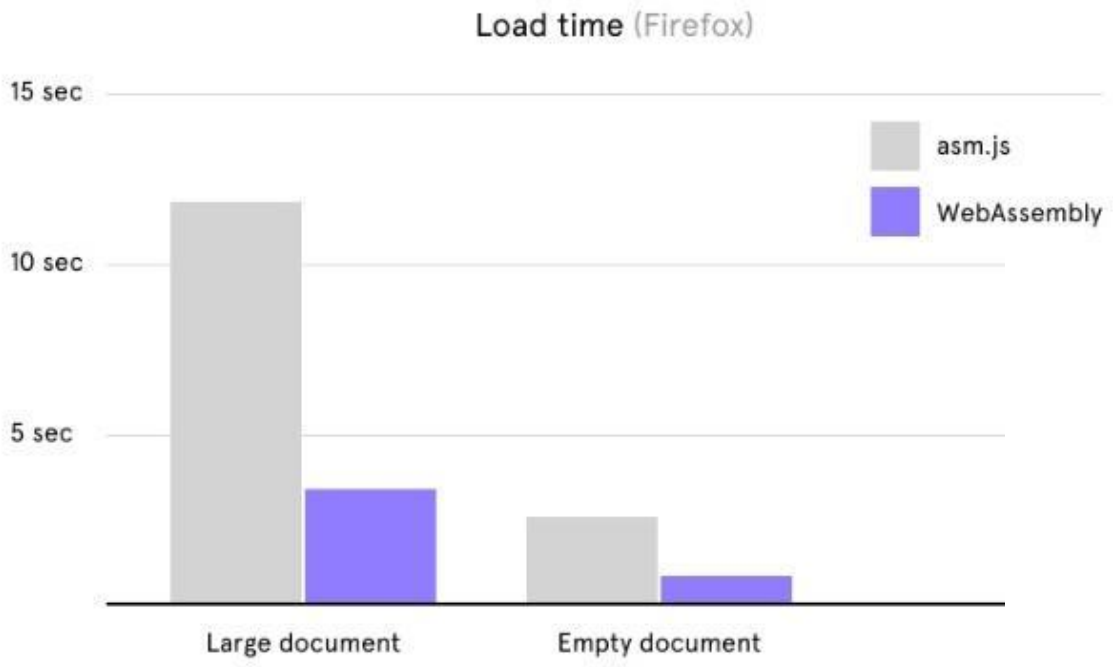
Background

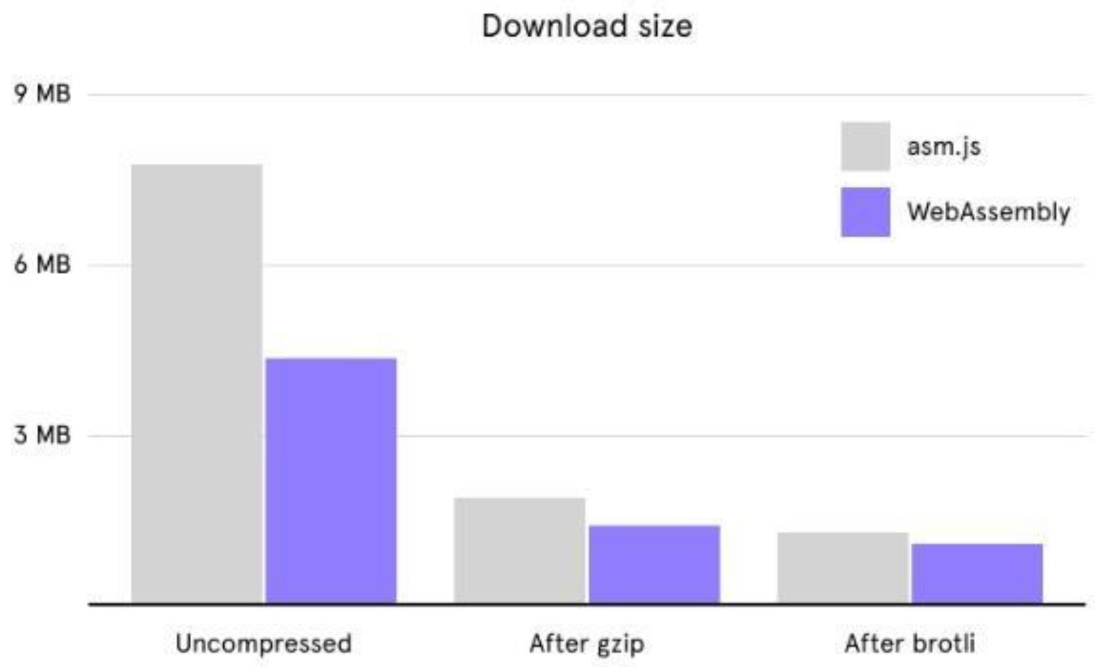
- JavaScript is the only language natively supported by browsers, but **it's not ideal for performance-intensive** tasks like gaming, video editing, or scientific simulations.
- Developers needed a way to run code written in other languages (e.g., C, C++, Rust) on the web with high performance.
- **Timeline**:
  - 2015: WebAssembly announced as a W3C project.
  - 2017: First stable release supported by major browsers (Chrome, Firefox, Edge, Safari).
  - 2020+: WASM expands beyond the browser (e.g., server-side, IoT).

WASM in the Real World

- Figma: Uses WASM for its vector graphics editor.
- Autodesk: Ported desktop CAD software to the web using WASM.
- Blender: (Experimental) WASM builds for 3D modeling in the browser.
- Google Earth: Uses WASM for rendering 3D maps.

Case Study: JS (asm.js) vs WebAssembly in Figma





https://www.figma.com/blog/webassembly-cut-figmas-load-time-by-3x/

Stack Based VM

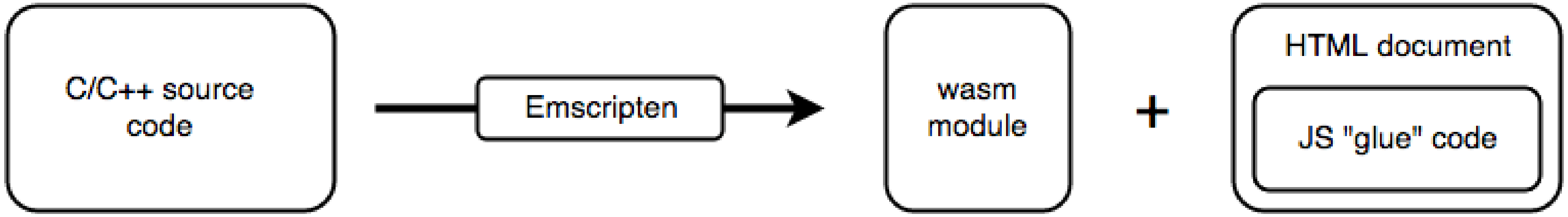
- WASM VM is implemented as stack-based virtual machine that emulates real CPU
- Example: Stack based VM for performing arithmetic.
- Four key objects:
  - Stack Pointer, pointing to the top of the stack
  - Instruction Pointer, pointing to the next instruction
  - Instructions
  - Stack Data Structure

Common file types

- .wasm
  - WebAssembly binary format
  - Compiled binary file
- .wat
  - WebAssembly text format
  - Human readable version (source code)
- .wast
  - Also text format, useful for multiple module and / or testing

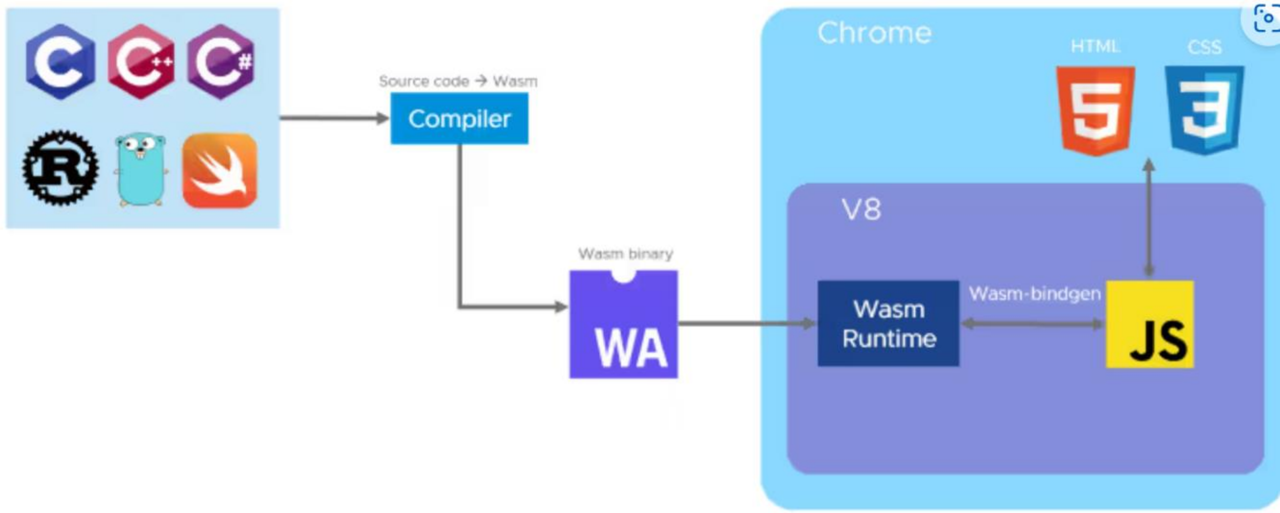
Example workflow: Porting from C++/C

- Emscripten first feeds the C/C++ into clang+LLVM — a mature open-source C/C++ compiler toolchain, shipped as part of XCode on OSX for example.
- Emscripten transforms the compiled result of clang+LLVM into a Wasm binary.
- By itself, WebAssembly cannot currently directly access the DOM; it can only call JavaScript, passing in integer and floating point primitive data types. Thus, to access any Web API, WebAssembly needs to call out to JavaScript, which then makes the Web API call. Emscripten therefore creates the HTML and JavaScript glue code needed to achieve this.



WASM in action

- WASM module, compiled .wasm file, is loaded into the browser and run at WASM runtime (part of the browser)
- JavaScript can call functions exported by the WASM module and pass data to/from it.
- WASM cannot directly access browser APIs (e.g., Canvas, WebGL, Fetch)
- JavaScript is used to update the DOM based on computations or results from WASM.



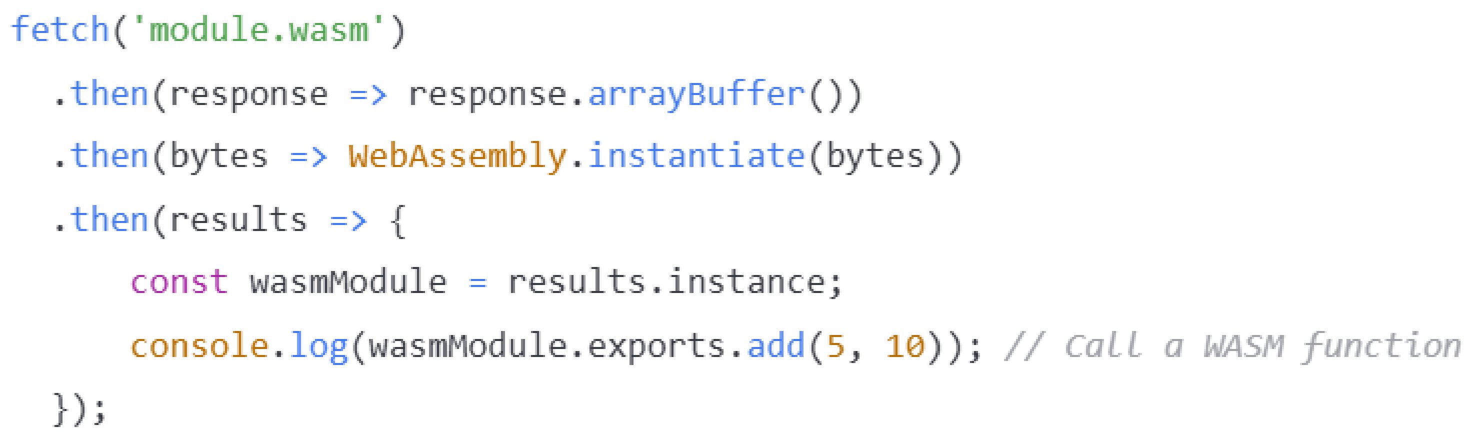
https://medium.com/@akshayshan28/building-web-apps-with-react-webassembly-and-go-1a3b2b138b63

Web Assembly as Module-based designed

- WASM code is organized into modules, which are self-contained units of functionality.
- Modules make WASM reusable and composable.
- Developers can load and execute multiple WASM modules in a single application.
- Modules contain
  - core logic functionality
  - functions/variables that can be accessed by JavaScript or other modules
  - functions/variables that the module needs from the host environment (e.g., JavaScript or Web API)

Linear memory model

- How memory is managed and accessed by WebAssembly programs → **contiguous, resizable array of bytes**
- Designed to be **simple, efficient, and portable,** making it suitable for low-level programming tasks
- **Shared** between the WebAssembly module and the host environment as ArrayBytes in JS



```
fetch('module.wasm')
  .then(response => response.arrayBuffer())
  .then(bytes => WebAssembly.instantiate(bytes))
  .then(results => {
      const wasmModule = results.instance;
      console.log(wasmModule.exports.add(5, 10)); // Call a WASM function
  });
```

```
const result = wasmModule.exports.add(5, 10); // Call the "add" function
in WASM
```

```
const canvas = document.getElementById('myCanvas');
const ctx = canvas.getContext('2d');
wasmModule.exports.draw(ctx); // WASM performs calculations, JS updates
the canvas
```

```
const result = wasmModule.exports.calculate();
document.getElementById('output').innerText = `Result: ${result}`;
```

Final Remarks

- WASM is not limited **only**
  - Running in browser → server-side WASM
  - Compatible with Web → wasmtime, IoT, etc