WEBASSEMBLY

ZERO-LATENCY BACKEND IN YOUR BROWSER

A NEW STANDARD FOR THE WEB

"WebAssembly or wasm is a new portable, size- and load-time-efficient format suitable for compilation to the web."

Developed by the W3C Community Group with Apple, Google, Mozilla and Microsoft

WEBASSEMBLY PROMISES

1. Designed to be encoded in a size- and load-time-efficient binary format.
2. WebAssembly is designed to be pretty-printed in a textual format for debugging, testing, etc.
3. Describes a memory-safe, sandboxed execution environment that may even be implemented inside existing JavaScript virtual machines.

HISTORY OF BROWSER JS ENGINES

MOZILLA'S *MONKEYS

Engine	Live	Function
SpiderMonkey	R.I.P.2008	The interpreter
Tracemonkey	R.I.P.2011	Initial JIT compiler
Jägermonkey	R.I.P.2013	Baseline JIT
IonMonkey	2012 -	Full optimizing JIT
Baseline	2013 -	Jägermonkey repl.
OdinMonkey	2013 -	Ahead-Of-Time Compiler (ASM.JS)
BaldrMonkey	2016 -	WASM Compiler

GOOGLE CHROME TENTACLES

Engine	Live	Function
Full-code-gen	R.I.P.2017?	Fast compiler unopt code
Crankshaft	R.I.P.2017?	Slow compiler opt. code
Ignition	2016 -	The bytecode interpreter
Turbofan	2015 -	The optimizing JIT

NATIVE CODE ON THE BROWSER BEFORE WEBASSEMBLY

1. ASM.JS: Highly optimized subset of javascript for ahead-of-time compilation
2. (P)NaCl: Google chrome's extensions and binary format for a native compilation target for the web

WHY WEBASSEMBLY?

Cause Online Games and pr0n rule the web?

No there is more: Cryptography, Audio, Image Processing, VR, Virtual Machines, CAD,...

WHY WEBASSEMBLY AGAIN?

- Avoid plugins (e.g. flash, activex, java)
- Bring existing applications to the web (too big to rewrite)
- Port high-performance native libraries to the web

I WANT MY NATIVE CODE RUNNING ON THE WEB NOW! BUT HOW?

EMSCRIPTEN TOOLCHAIN

- C/C++/Rust => LLVM => Emscripten => WASM
- A Bunch of APIs for Filesystem, WebGL, HTML5, etc.
- Integrated Binders: WebIDL, embind

EMSCRIPTEN TOOLCHAIN INTERMEDIATES

C/C++ => LLVM => Emscripten => Binaryen

C/C++ => Bitcode => *WAST* => WASM

STEP BY STEP: BUILD NATIVE CODE TO WEBASSEMBLY

STARTER: THE OBLIGATORY HELLO WORLD

```cpp
#include <iostream>

int main(int argc, char* argv[]) {

  std::cout << "Hello world" << "\n";

  return 0;
}
```

Compile with:

```
$ em++ hello.cpp -o hello.js -s BINARYEN=1 -s "BINARYEN_METHOD='n
$ ls -la
-rw-r--r-- 1 user staff 1774226 May 15 21:44 hello.asm.js
-rw-r--r-- 1 user staff     110 May 15 21:40 hello.cpp
-rw-r--r-- 1 user staff  276090 May 15 21:44 hello.js
-rw-r--r-- 1 user staff  394912 May 15 21:44 hello.wasm
```

STEP 1: PREPARE YOUR BUILDSYSTEM TO USE EMSCRIPTEN

Autotools: Use emconfigure / emmake wrappers

```
cd project-dir/
emconfigure ./configure --prefix=...
emmake make
```

CMake: Use Emscripten.Toolchain

```
cd project-dir/build
cmake -DCMAKE_TOOLCHAIN_FILE=/PATH/TO/Emscripten.cmake ../src
```

STEP 2: CHECK YOUR LIBRARY DEPENDENCIES

- Build library dependencies with emscripten, too!!!
- Build and link library dependencies as static libraries
- No support for shared libraries yet (Post-MVP: ES6 Modules)

STEP 3: CHECK FOR WEBASSEMBLY LIMITATIONS

- NO arch assumption: endianess, alignment
- NO stack manipulation: setjmp/longjmp
- NO dlopen/dlsym support
- NO native assembler support (e.g. cryptography)
- NO threading support yet
- NO SIMD support yet
- NO cross-language exception handling support
- NO 64-bit support yet
- NO full IEEE 754-2008 floating point support yet

STEP 4: PREPARE YOUR INTERFACE FOR YOUR JS-CODE

```
#ifndef IMAGEPROCESSOR_H_
#define IMAGEPROCESSOR_H_

#include <map>
#include <string>
#include <vector>

namespace mayflower  {
namespace wasm {

class ImageProcessor {
 public:
  explicit ImageProcessor(const std::string& path)
      : path_(path) {};

  ImageProcessor(const ImageProcessor& ip) = delete;
```

STEP 5: WRITE THE BINDERS FOR JAVASCRIPT

e.g. embind:

```
#include <emscripten/bind.h>
#include "imageprocessor.hpp"

EMSCRIPTEN_BINDINGS(ImageProcessor){
emscripten::register_vector<int>("VectorInt");
emscripten::register_map<std::string, long>("MapStringLong");

emscripten::class_<mayflower::wasm::ImageProcessor>("ImageProcess
  .constructor<std::string>()
  .function("dimensions", &mayflower::wasm::ImageProcessor::dimen
  .function("histogram", &mayflower::wasm::ImageProcessor::histog
  .function("resize", &mayflower::wasm::ImageProcessor::resize)
  .function("crop", &mayflower::wasm::ImageProcessor::crop);
}
```

STEP 6: RUN YOUR BUILD

```
$ cd build
$ make
$ ls -la
-rw-r--r-- 1 user staff 4571975 Apr 28 imageprocessor.asm.js
-rw-r--r-- 1 user staff  178348 Apr 28 imageprocessor.js
-rw-r--r-- 1 user staff  578728 Apr 28 imageprocessor.wasm
```

STEP 7: TEST YOU BUILD IN NODEJS

For example with cmake & ctest:

```
$ cd build
$ ctest

Test project /Path/to/wasm-imageeditor/build
    Start 1: imageprocessor-test
1/1 Test #1: imageprocessor-test ..............   Passed    0.67

100% tests passed, 0 tests failed out of 1

Total Test time (real) =   0.68 sec
```

STEP 8: LOAD YOUR WASM MODULE IN YOUR REACT APP

Load the module in a react component through fetch:

```
fetch('imageprocessor.wasm')
  .then((response) => response.arrayBuffer())
  .then((buffer) => {
    Module.arguments = this.props.binArguments;
    Module.environment = this.props.environment;
    Module.locateFile = this.props.locateMemFile;
    Module.logReadFiles = this.props.logReadFiles;
    Module.noExitRuntime = this.props.noExitRuntime;
    Module.noInitialRun = this.props.noInitialRun;
    Module.print = this.props.print;
    Module.printErr = this.props.printErr;
    Module.preInit = this.props.preInit;
    Module.preRun = this.props.preRun;
    Module.postRun = this.props.postRun;
    Module.wasmBinary = buffer;
  });
```

STEP 9: LOAD EMSCRIPTEN SHELL CODE

```
fetch('imageprocessor.js')
  .then((response) => response.blob())
  .then((responseBlob) => {
    const script = document.createElement('script');
    const src = URL.createObjectURL(responseBlob);

    script.src = src;
    script.async = false;
    document.getElementById('ImageProcessor').appendChild(script)
});
```

STEP 10: CALL YOUR WASM MODULE IN YOUR REACT APP

```
// Instantiate wasm through window.Module latter ES6 module
const ip = new Module.ImageProcessor(targetFilename);

// Call your native good through your bindings
const dims = ip.dimensions();
const newWidth = dims.get("x") * (zoomFactor / 100);
const newHeight = dims.get("y") * (zoomFactor / 100);

ip.crop(0, 0, newWidth, newHeight);

// Destroy your native module
ip.delete();
```

DEMO

WEBASSEMBLY + REACTJS IMAGEEDITOR

Open Demo

- Frontend: ReactJS + Redux + Typescript
- Backend: C++ + Boost/Gil + libjpg
- Bundler: Webpack2
- Buildsystem: cmake

https://github.com/periklis/wasm-imageeditor

WEBASSEMBLY INTEGRATIONS

Type	Possibility
Filesystem	NodeFS, IndexedDB, MEMFS
Audio	AudioContext through SDL
WebGL	WebGL through SDL
WebAPIs	emschripten/html5.h
Call JS in C/C++	Use emscripten.h
Worker API	Use emscripten.h for "threads"
Stdio	Use Module.print/printErr

WEBASSEMBLY BROWSER SUPPORT

Browser	Can i use wasm?	Can i use asm.js
Firefox	53	52
Chrome	57	49*
Edge	14	14

**Chrome without Ahead-Of-Time-Compilation*

SO LONG AND THANKS FOR THE FISH!

Periklis Tsirakidis

Github: github.com/periklis