FH | JOANNEUM
University of Applied Sciences

WASM

WebAssembly - Tutorial

John Feiner

Web Assembly (WASM)

Examples

(A) Online: Try out in WASM fiddle

(B) Simple: Using embedded (precompiled) code

(C) Complete: Load (precompiled code) from server

WASM

(A) Online

John Feiner

Compile to .wasm module

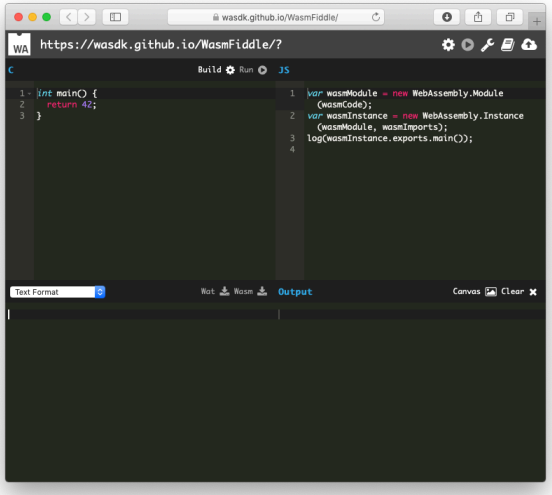
1.) frontend: e.g. C (lang), Rust, Kotlin, ...

2.) IR optimiser LLVM

3.) backend:

Option A) LLVM wasm backend

Option B) Emscripten asm2wasm



https://wasdk.github.io/WasmFiddle/

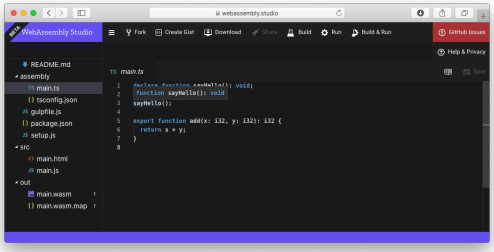
Installation Windows/Linux/Mac:

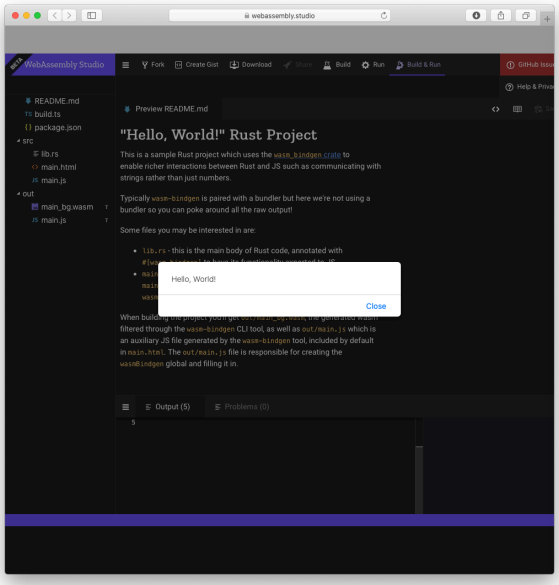
Emscripten SDK: "emsdk"

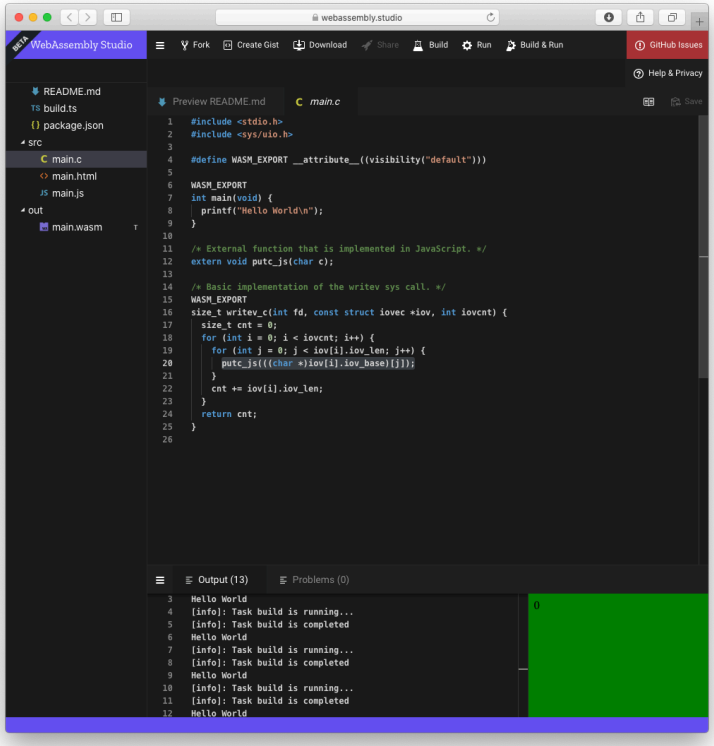
(includes Clang, Python, Node.js,)

WASM Studio

Compile online:
Rust, C, C++,...





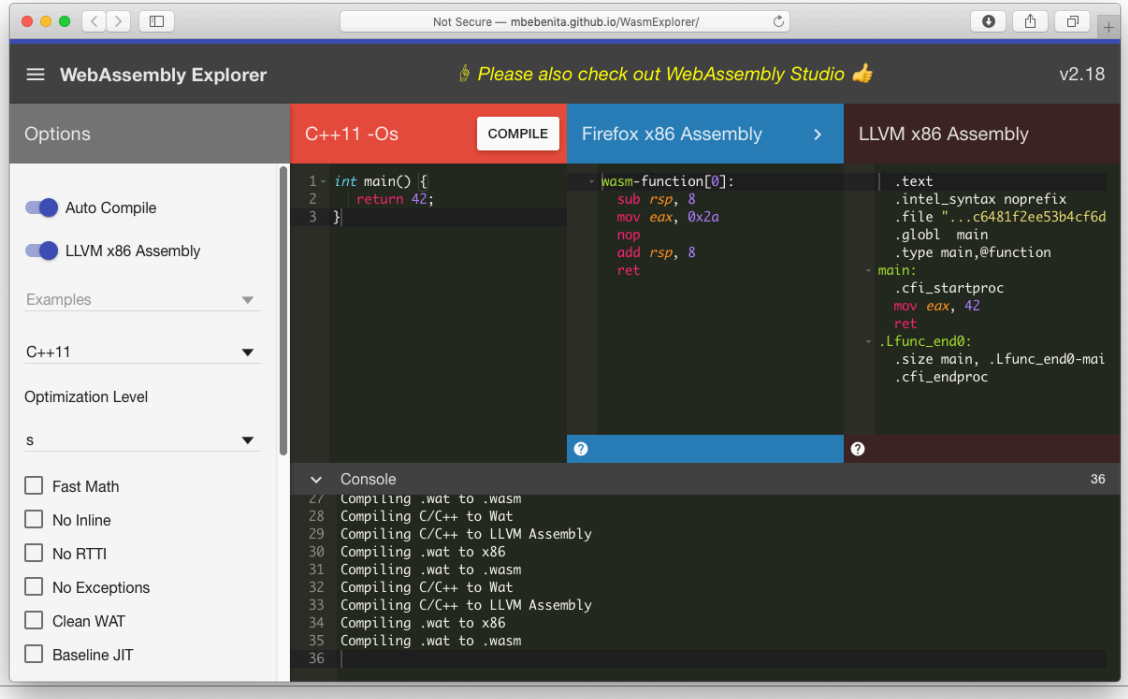


https://webassembly.studio

Optional: WASM Compiler

```
int main() {
  return 42;
}
```

C->Wat->wasm



Firefox x86 Assembly

LLVM x86 Assembly

http://mbebenita.github.io/WasmExplorer/

Optional: WAT to WASM Compiler

```
int main() {
  return 42;
}
```

C->Wat->wasm

```
(module
 (table 0 anyfunc)
 (memory $0 1)
 (export "memory" (memory $0))
 (export "main" (func $main))
 (func $main (; 0 ;) (result i32)
  (i32.const 45)
 )
)
```

S-Expressions (symbolic expression)

user readable representation
i.e. Text (*.wat)

```
00 61 73 6D 0D 00 00 00 01 86 80 ...
```

Uint8Array

https://developer.mozilla.org/en-US/docs/WebAssembly/Text_format_to_wasm#Converting_the_text_.wat_into_a_binary_.wasm_file

WASM

(B) Simple

John Feiner

C -> WASM — Simple 1/2

1 hello42.js

```
int main() {
  return 42;
}
```

https://wasdk.github.io/WasmFiddle/

```
var wasmCode = new Uint8Array(
[0,97,115,109,1,0,0,0,1,133,128,128,128,0,1,96,0,1,127,3,130,128,128,128,0,
1,0,4,132,128,128,128,0,1,112,0,0,5,131,128,128,128,0,1,0,1,6,129,128,128,1
28,0,0,7,145,128,128,128,0,2,6,109,101,109,111,114,121,2,0,4,109,97,105,11
0,0,0,10,138,128,128,128,0,1,132,128,128,128,0,0,65,42,11]);

var wasmModule =
        new WebAssembly.Module(wasmCode);
var wasmInstance
        = new WebAssembly.Instance(wasmModule, wasmImports = {} );

document
  .getElementById('out')
  .innerText = "The static wasm code returned: " +
                    wasmInstance.exports.main()
```

C -> WASM — Simple 2/2


```
<!DOCTYPE html>
<html>
 <head>
  <script src="hello42.js" defer></script>
  <title>Hello static WASM 42</title>
 </head>
 <body>
    <div id="out">output 42 of static wasm will appear here</div>
 </body>
</html>
```

2 index.html

WASM

(C) Complete

John Feiner

C -> WASM — Step 1: Code Some Functionality

```
int add(int a, int b) {
  return a+b;
}

int main() {
  return 42;
}

int sub(int a, int b) {
  return a-b;
}
```

1

hello.c

C -> WASM — Step 2: Compile

```
emcc \
  -s EXPORTED_FUNCTIONS='["_add","_sub","_main"]' \
  -Os \
  -s SIDE_MODULE=1 \
  hello.c
```

2

C -> WASM — Step 3: Provide HTML

```
<!DOCTYPE html>
<html>
 <head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="shortcut icon" href="/favicon.ico" />
  <script src="hello.js" defer></script>
  <title>Hello WASM</title>
 </head>
 <body>
    <div id="out">output will appear here</div>

    <input id="a" type="number" value="3" size="3" min="0"/>
    <input id="b" type="number" value="4" size="3"/>
    <button type="button" id="run">Run wasm function</button>
 </body>
</html>
```

unobtrusive JavaScript

3

hello.html

C -> WASM — Step 4: Provide JS via TypeScript

```
document.getElementById('out').innerText="Loading JS file worked.\n"

if (!('WebAssembly' in window)) {
  document.getElementById('out').innerText +=
     " ERROR! Please enable browser support for wasm! Then try again.\n";
}else{
  document.getElementById('out').innerText +=
     " OK, WebAssembly is supported.\n"
}
...
```

hello.ts

4a

```
...
function loadWebAssembly(filename, theImportObj = {}) {
// Fetch the file and compile it
  const fetchPromise = fetch(filename)
  return fetchPromise
    .then(response => response.arrayBuffer())
    .then(buffer => WebAssembly.compile(buffer))
    .then(module => {
     // Create the instance.
      return new WebAssembly.Instance(module, theImportObj);
  });
}
...
```

Prepare a function to
FETCH
COMPILE
CREATE INSTANCE

hello.ts

4b


```
...
function init() {
  const memory = new WebAssembly.Memory({
        initial: 256, maximum: 256 });
  const env = {
    'abortStackOverflow': _ => { throw new Error('overflow'); },
    'table': new WebAssembly.Table({
        initial: 0, maximum: 0, element: 'anyfunc' }),
    'tableBase': 0,
    'memory': memory,
    '__memoryBase': 1024,
    'STACKTOP': 0,
    'STACK_MAX': memory.buffer.byteLength,
  };
  const importObject = { env };
  ...
```

Configure The env:
6MB mem, (empty) table, error function,
...
Table to store function references
Mem (=resizeable ArrayBuffer)
i.e. no of 64KiB pages

...We also need to specify an **importObject**: this provides the environment Web Assembly runs in as well as any other parameters to instantiation. At a minimum, you need to provide **memory** and an **environment** object like this—add this at the end your script tag...

4c

hello.ts

https://hacks.mozilla.org/2017/02/creating-and-working-with-webassembly-modules/

https://kiosk-dot-codelabs-site.appspot.com/codelabs/web-assembly-intro/index.html?index=..%2F..index#3

```
  ...
  loadWebAssembly('a.out.wasm', importObject)
    .then(instance => {
      const exports = instance.exports;
      const theMainFunction: Function = <Function>exports.main;
      const theAddFunction: Function = <Function>exports.add;

      var button = document.getElementById('run');
      button.addEventListener('click', function () {
        document.getElementById('out').innerText +=
          "\n * Wasm 'main' returned " + theMainFunction();
        const a = (document.getElementById('a') as HTMLInputElement).value;
        const b = (document.getElementById('b') as HTMLInputElement).value;
        document.getElementById('out').innerText +=
          "\n *  wasm 'add(" + a + "," + b + ")' returned " + theAddFunction(a, b);
      }, false);
    }
    );
}
init();
```

USE EXPORTED FUNCTIONALITY

4d

hello.ts

Compile JS to TypeScript

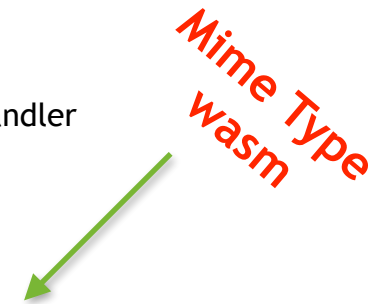
tsc hello.ts

C -> WASM — Step 5: Provide Server

```
#!/usr/bin/env python3
import sys
import http.server
from http.server import HTTPServer, BaseHTTPRequestHandler
import socketserver
PORT= int(sys.argv[1])  if len(sys.argv) > 1 else 8000

Handler = http.server.SimpleHTTPRequestHandler
Handler.extensions_map['.wasm'] = 'application/wasm'
httpd = socketserver.TCPServer(  ('localhost', PORT),  Handler )

print("Modified Python3 server for WASM (application/wasm) is serving at port", PORT)
httpd.serve_forever()
```



5

wasm-serve,py

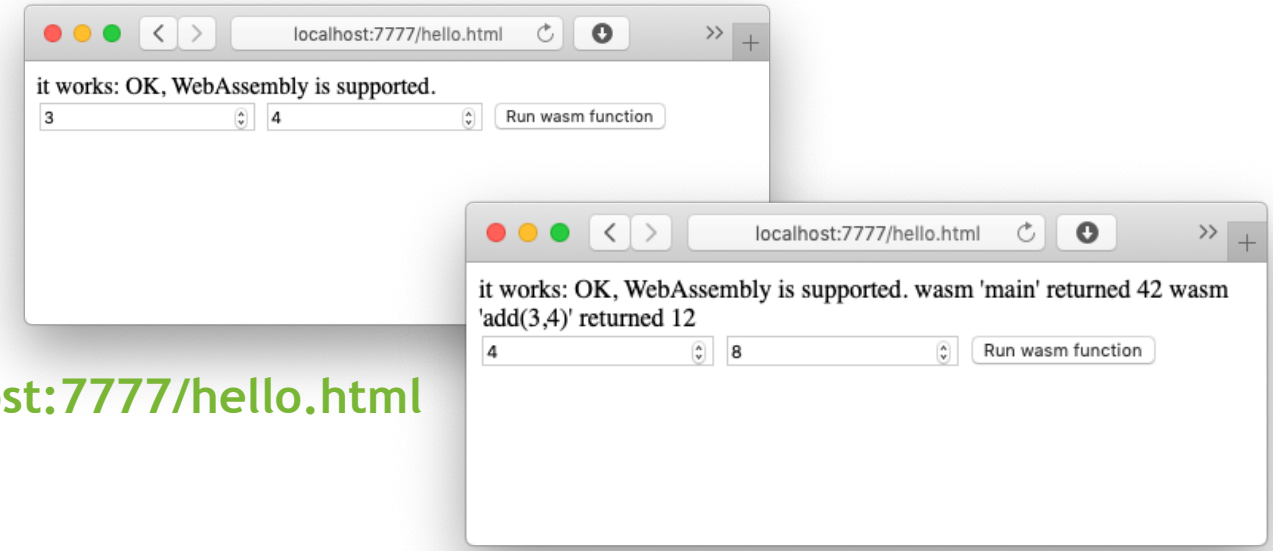
https://curiousprog.com/2018/10/08/serving-webassembly-files-with-a-development-web-server/

https://gist.github.com/HaiyangXu/ec88cbdce3cdbac7b8d5

C -> WASM — Step 6: Run Server

6

./wasm-serve.py



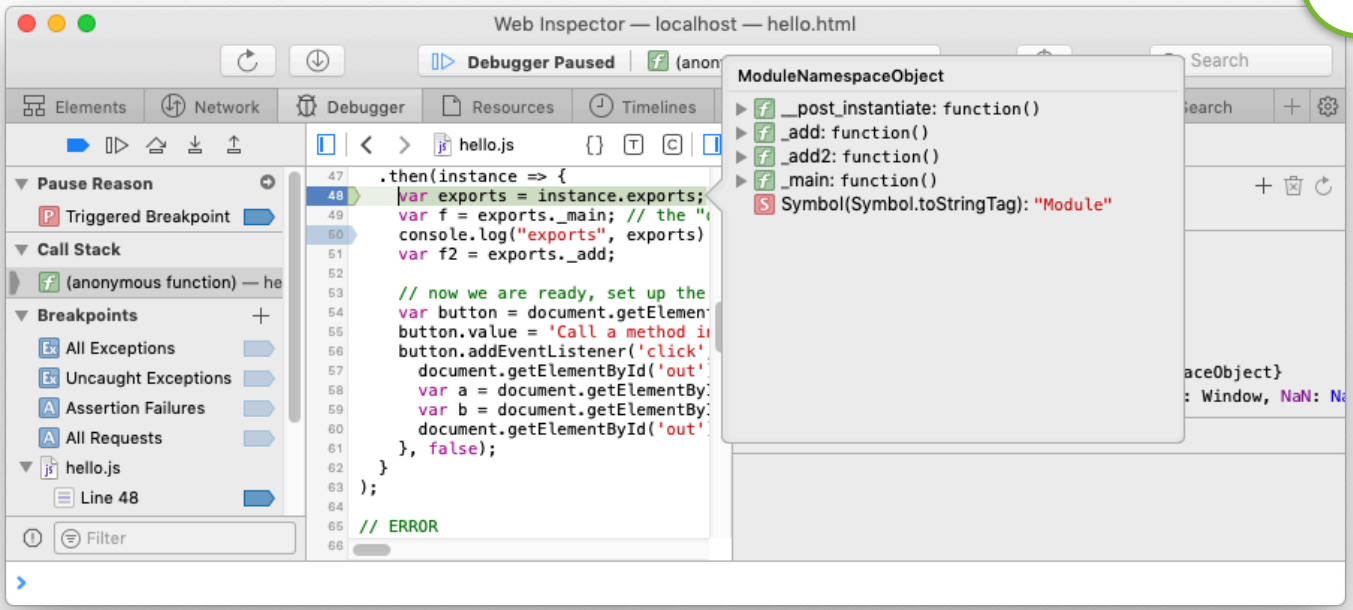
http://localhost:7777/hello.html

https://curiousprog.com/2018/10/08/serving-webassembly-files-with-a-development-web-server/

https://gist.github.com/HaiyangXu/ec88cbdce3cdbac7b8d5

C -> WASM — Step 7: Inspect

7



Debugging:
Set breakpoints

C -> WASM — Step 8: Modify and ...

8a Add new function — Find highest prime for given input

8b Measure duration 10, 100, 1.000, 10.000 calls

8c Measure and compare same function implemented in JavaScript

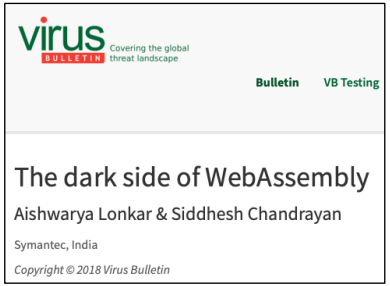
C -> WASM — Step 8: ... and inspect

8d

Optional: reverse engineering wasm

8e

Read about Security


virus BULLETIN Covering the global threat landscape

Bulletin VB Testing

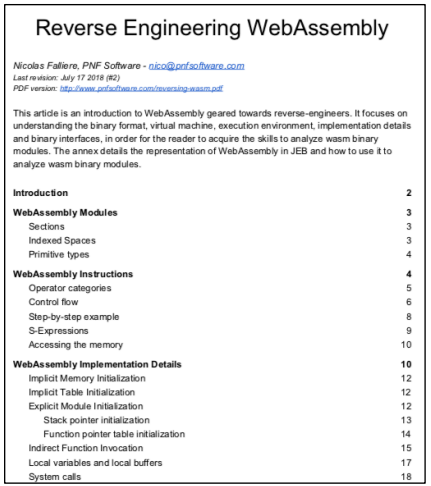
The dark side of WebAssembly

Aishwarya Lonkar & Siddhesh Chandrayan

Symantec, India

Copyright © 2018 Virus Bulletin

https://www.virusbulletin.com/virusbulletin/2018/10/dark-side-webassembly/


Reverse Engineering WebAssembly

Nicolas Falliere, PNF Software - nico@pnfsoftware.com

Last revision: July 17 2018 (62)

PDF version: http://www.pnfsoftware.com/reversing-wasm.pdf

This article is an introduction to WebAssembly geared towards reverse-engineers. It focuses on understanding the binary format, virtual machine, execution environment, implementation details and binary interfaces, in order for the reader to acquire the skills to analyze wasm binary modules. The annex details the representation of WebAssembly in JEB and how to use it to analyze wasm binary modules.

https://www.pnfsoftware.com/reversing-wasm.pdf