

Web Evolution and WebAssembly



COMP 520 - Special Topics Lecture I

David Herrera

March 29, 2019

Contents

- Limitations of JavaScript
- Evolution of Web performance via asm.js
- WebAssembly
 - Design
 - Pipeline
 - Decoding
 - Validation
 - Execution
 - Examples

JavaScript - What is JavaScript?

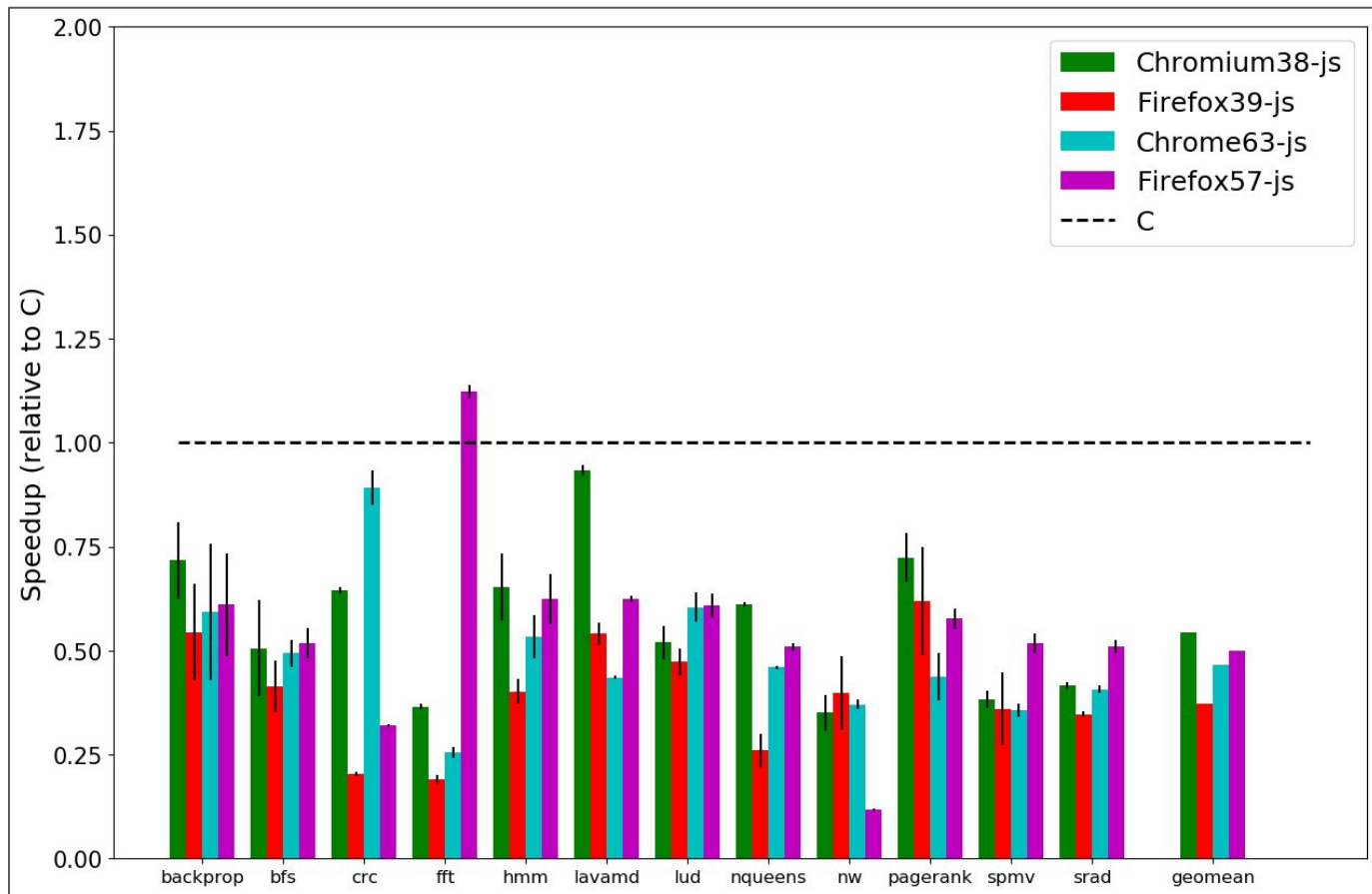
- **Dynamic**, high-level language
- **10 days!**, Famously designed and prototyped in ten days by Brendan Eich
- **Little Performance Design**: Language was not designed with performance in mind.
- **Web Language**: Has been the main programming language for the web since 1999



Limitations of JavaScript

- **Tough Target:** Dynamically typed nature makes it a “tough target” of static languages such as C and C++, as well as a relatively slow language.
- **Lacks Parallelism:** No real parallelism supported natively. (At least not widely supported by all browsers, or general with full control)
- **Number type:** Numbers are restricted to doubles, float64. This means that for instance, an i64 number cannot be represented natively in JavaScript.

Let's look at speed



JavaScript as a target language for C/C++?

- **Is it Doable?** Yes, since JavaScript is turing complete, it should be able to represent any sort of weird semantics.

JavaScript as a target language for C/C++?

- **Is it Doable?** Yes, since JavaScript is turing complete, it should be able to represent any sort of weird semantics.
- **Is it efficient?** Let's look at *Emscripten*
- **Emscripten** is a static compiler from LLVM to JavaScript created in 2011
- **Asm.js** is a “typed” subset of JavaScript which serves as a target for Emscripten
- Initial goal was to support a large enough subset of C and C++ constructs that could be run on the web.
- Any language that has front-end to **LLVM** can compile to **asm.js**

Example

```
function DiagModule(stdlib, foreign, heap) {  
  "use asm";  
  // Variable Declarations  
  var sqrt = stdlib.Math.sqrt;  
  
  // Function Declarations  
  function square(x) {  
    x = +x;  
    return +(x*x);  
  }  
  function diag(x, y) {  
    x = +x;  
    y = +y;  
    return +sqrt(square(x) + square(y));  
  }  
  return { diag: diag };  
}
```

- **Number Representation:**

```
+x // Double  
y|0 // Integer  
Math.fround(x) // Float
```

- **Memory:**

```
HEAP32[p >> 2] | 0
```


Let's look at some of the problems faced
by asm.js

Main memory representation

How do we represent C main memory in JavaScript?

Main memory representation

How do we represent C main memory in JavaScript?

How about just a simple array?

```
var HEAP = new Array(size);
```

- This **HEAP** will serve as both C's **stack** and **heap**
- Every element represents a byte, and the addresses are integer indices to the array.

Ok, let's do something simple

What does this code do?

```
int x = 12345;  
printf("The first byte: %d\n", *((char *)&x));
```

- **Recall:** An integer normally has 4 bytes in c, while a char is made up of 1 byte.

Ok, let's do something simple

What does this code do?

```
int x = 12345;  
printf("The first byte: %d\n", *((char *)&x));
```

- **Recall:** An integer normally has 4 bytes in c, while a char is made up of 1 byte.
- This sort of program is said to not respect the **Load-Store Consistency (LSC)** property

Ok, let's do something simple

What does this code do?

```
int x = 12345;  
printf("The first byte: %d\n", *((char *)&x));
```

- **Recall:** An integer normally has 4 bytes in c, while a char is made up of 1 byte.
- This sort of program is said to not respect the **Load-Store Consistency (LSC)** property

How do we represent it in JavaScript?

Char from Int in JavaScript

Here is the JavaScript Implementation:

```
var x_value = 12345;
var x_addr = stackAlloc(4);
HEAP[x_addr] = (x_value >> 0 ) & 255;
HEAP[x_addr+1] = (x_value >> 8 ) & 255;
HEAP[x_addr+2] = (x_value >> 16 ) & 255;
HEAP[x_addr+3] = (x_value >> 24 ) & 255;
printf("first byte: %d\n",HEAP[x_addr]); //Implemented in JavaScript
```

Char from Int in JavaScript

Here is the JavaScript Implementation:

```
var x_value = 12345;
var x_addr = stackAlloc(4);
HEAP[x_addr] = (x_value >> 0 ) & 255;
HEAP[x_addr+1] = (x_value >> 8 ) & 255;
HEAP[x_addr+2] = (x_value >> 16 ) & 255;
HEAP[x_addr+3] = (x_value >> 24 ) & 255;
printf("first byte: %d\n",HEAP[x_addr]); //Implemented in JavaScript
```

- What is the problem?

Char from Int in JavaScript

Here is the JavaScript Implementation:

```
var x_value = 12345;
var x_addr = stackAlloc(4);
HEAP[x_addr] = (x_value >> 0 ) & 255;
HEAP[x_addr+1] = (x_value >> 8 ) & 255;
HEAP[x_addr+2] = (x_value >> 16 ) & 255;
HEAP[x_addr+3] = (x_value >> 24 ) & 255;
printf("first byte: %d\n",HEAP[x_addr]); //Implemented in JavaScript
```

- What is the problem?
 - 8 operations and 4 accesses to simply set an integer value!

What was **asm.js** solution to this problem?

- Only support programs that respect **Load-Store Consistency**.
- How do we make sure that a program respects it? Is it efficient?

What was **asm.js** solution to this problem?

- Only support programs that respect **Load-Store Consistency**.
- How do we make sure that a program respects it? Is it efficient?

Solution: Don't check for it!

- Assume property holds and offer a compiler flag to check
- Now we can simply represent an integer with one element in the array.
- Further optimize with **variable nativization**


```
HEAP[x_addr] = 12345;
```


```
var x_value = 12345;
```

A diagram illustrating the transformation of a memory access into a variable access. On the left, a box contains the code 'HEAP[x_addr] = 12345;'. An arrow points from this box to a box on the right containing the code 'var x_value = 12345;'. This represents the process of 'variable nativization' where a memory location is replaced by a local variable.

Continuing with asm.js...

Unity **Blog**

Subscribe to RSS 



First Unity game in WebGL: Owlchemy Labs' conversion of Aaaaa! to asm.js

Kristyna Hougaard, October 14, 2014

Source: <https://blogs.unity3d.com/2014/10/14/first-unity-game-in-webgl-owlchemy-labs-conversion-of-aaaaa-to-asm-js/>

Continuing with asm.js...

- Novel ideas from asm.js
 - Supporting a large subset of C and C++ efficiently.
 - The C/C++ programs supported must be cut down in order to perform operations efficiently
 - Make a typed subset of JavaScript which can be highly optimized by a specialized section of the JavaScript JIT compilers.

Continuing with asm.js...

- Novel ideas from asm.js
 - Supporting a large subset of C and C++ efficiently.
 - The C/C++ programs supported must be cut down in order to perform operations efficiently
 - Make a typed subset of JavaScript which can be highly optimized by a specialized section of the JavaScript JIT compilers.
- **asm.js** has since grown to be supported by most browser vendors.
- In 2013, typed arrays became the standard, all due to **asm.js**
 - *Int8Array, Int16Array, Int32Array, Float64Array, Float32Array etc.*
 - All of this have an **ArrayBuffer** as their underlying representation. This array buffer is a byte array.

Limitations of asm.js

- **Parallelism**, JavaScript lacking proper support for parallelism
 - No data parallelism, e.g. no SIMD instructions
 - No task parallelism, e.g. shared memory or other parallel primitives.

Limitations of asm.js

- **Parallelism**, JavaScript lacking proper support for parallelism
 - No data parallelism, e.g. no SIMD instructions
 - No task parallelism, e.g. shared memory or other parallel primitives.
- **No garbage collection**, asm.js has no garbage collection, the HEAP array is only cleaned up by calls to **free()**. Similar to C.

Limitations of asm.js

- **Parallelism**, JavaScript lacking proper support for parallelism
 - No data parallelism, e.g. no SIMD instructions
 - No task parallelism, e.g. shared memory or other parallel primitives.
- **No garbage collection**, asm.js has no garbage collection, the HEAP array is only cleaned up by calls to **free()**. Similar to C.
- **Slow**, Compilation and initialization of an asm.js module is slow.
 - Still has to parse normal JavaScript
 - JavaScript does not come in a “compressed” format i.e. a binary syntax

Limitations of asm.js

- **Parallelism**, JavaScript lacking proper support for parallelism
 - No data parallelism, e.g. no SIMD instructions
 - No task parallelism, e.g. shared memory or other parallel primitives.
- **No garbage collection**, asm.js has no garbage collection, the HEAP array is only cleaned up by calls to **free()**. Similar to C.
- **Slow**, Compilation and initialization of an asm.js module is slow.
 - Still has to parse normal JavaScript
 - JavaScript does not come in a “compressed” format i.e. a binary syntax
- **Hard to scale**, in order to grow **asm.js** to support more constructs from typed languages, JavaScript must also grow

Enter WebAssembly...

- WebAssembly, or "*wasm*", is a general-purpose virtual ISA designed to be a compilation target for a wide variety of programming languages.
- Currently supported and in active development by all the major browser vendors
- Promises to bridge the gap in performance through different mechanisms

WebAssembly enhancing performance

How?

- Support for various integer, and floating types natively
- Increase in loading speed via a fast binary decoding, and streaming compilation.
- Support for data parallelism via SIMD instruction set
- Support for task parallelism via threads.
- A garbage collector for the “main” memory.

WebAssembly - Contents

- Design goals
- Performance
- Representation
- Pipeline
 - Encoding/Decoding
 - Validation
 - Execution
- Examples

Design Goals

- **Fast:** Execute with near native speed

Design Goals

- **Fast:** Execute with near native speed
- **Safe:** Code is validated and executes in a memory safe environment

Design Goals

- **Fast:** Execute with near native speed
- **Safe:** Code is validated and executes in a memory safe environment
- **Well-Defined:** Fully and precisely defines valid programs in a way that can be verified formally and informally

Design Goals

- **Fast:** Execute with near native speed
- **Safe:** Code is validated and executes in a memory safe environment
- **Well-Defined:** Fully and precisely defines valid programs in a way that can be verified formally and informally
- **Hardware-Independent:** Works as an abstraction over most popular hardware architectures for fast compilation. No operation that is specific to a hardware architecture is likely to be supported.

Design Goals

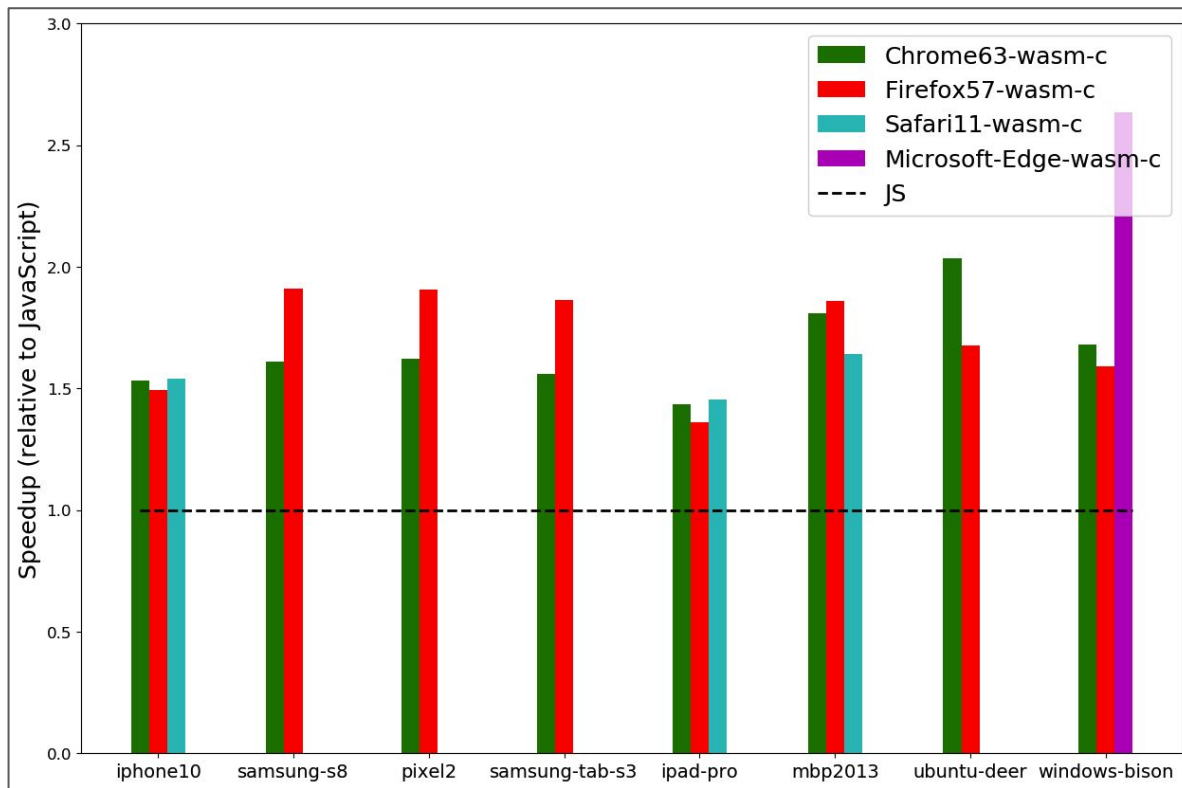
- **Fast:** Execute with near native speed
- **Safe:** Code is validated and executes in a memory safe environment
- **Well-Defined:** Fully and precisely defines valid programs in a way that can be verified formally and informally
- **Hardware-Independent:** Works as an abstraction over most popular hardware architectures for fast compilation. No operation that is specific to a hardware architecture is likely to be supported.
- **Language-Independent:** Does not favor any particular language, Object Model, or programming model.

Design Goals

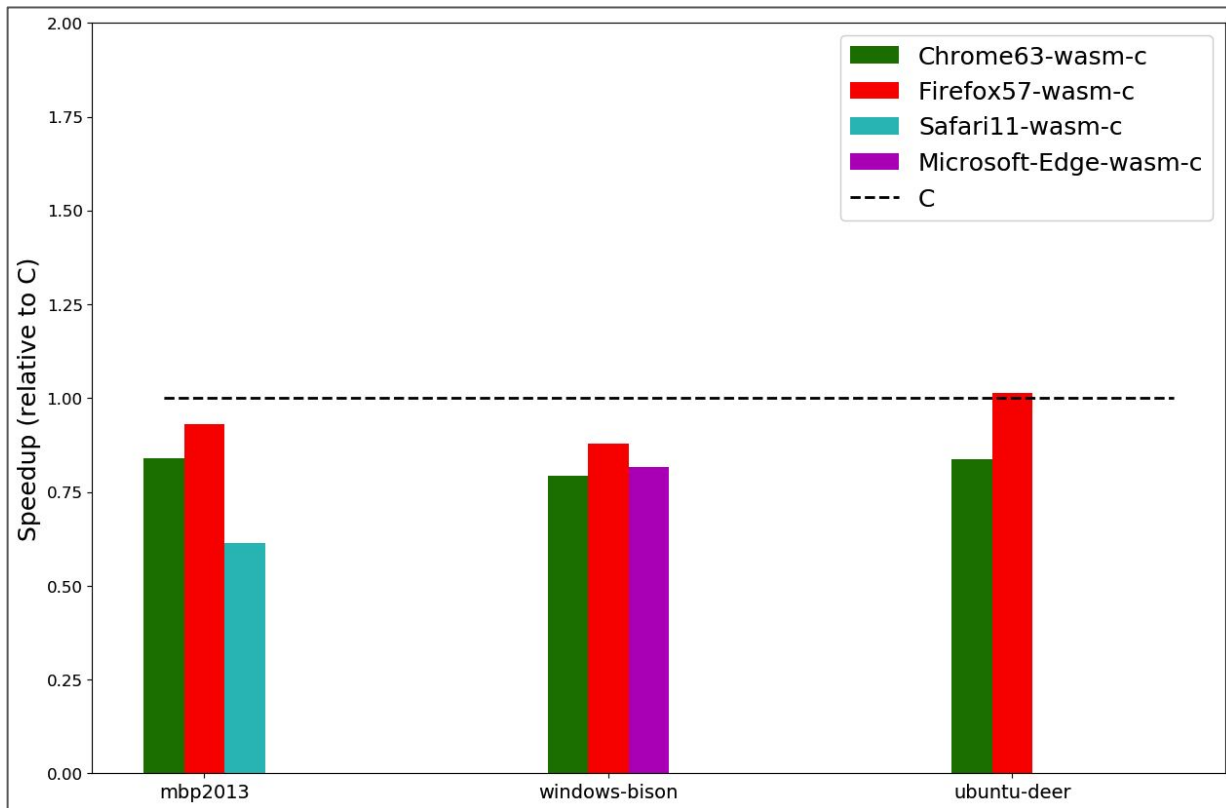
- **Fast:** Execute with near native speed
- **Safe:** Code is validated and executes in a memory safe environment
- **Well-Defined:** Fully and precisely defines valid programs in a way that can be verified formally and informally
- **Hardware-Independent:** Works as an abstraction over most popular hardware architectures for fast compilation. No operation that is specific to a hardware architecture is likely to be supported.
- **Language-Independent:** Does not favor any particular language, Object Model, or programming model.
- **Platform-Independent:** Does not depend on the Web, it can run as an independent VM in any environment. (Mozilla recently announce WASI, the WebAssembly System Interface)

Does it deliver on the “close-to-native”
performance?

Let's first see versus JavaScript



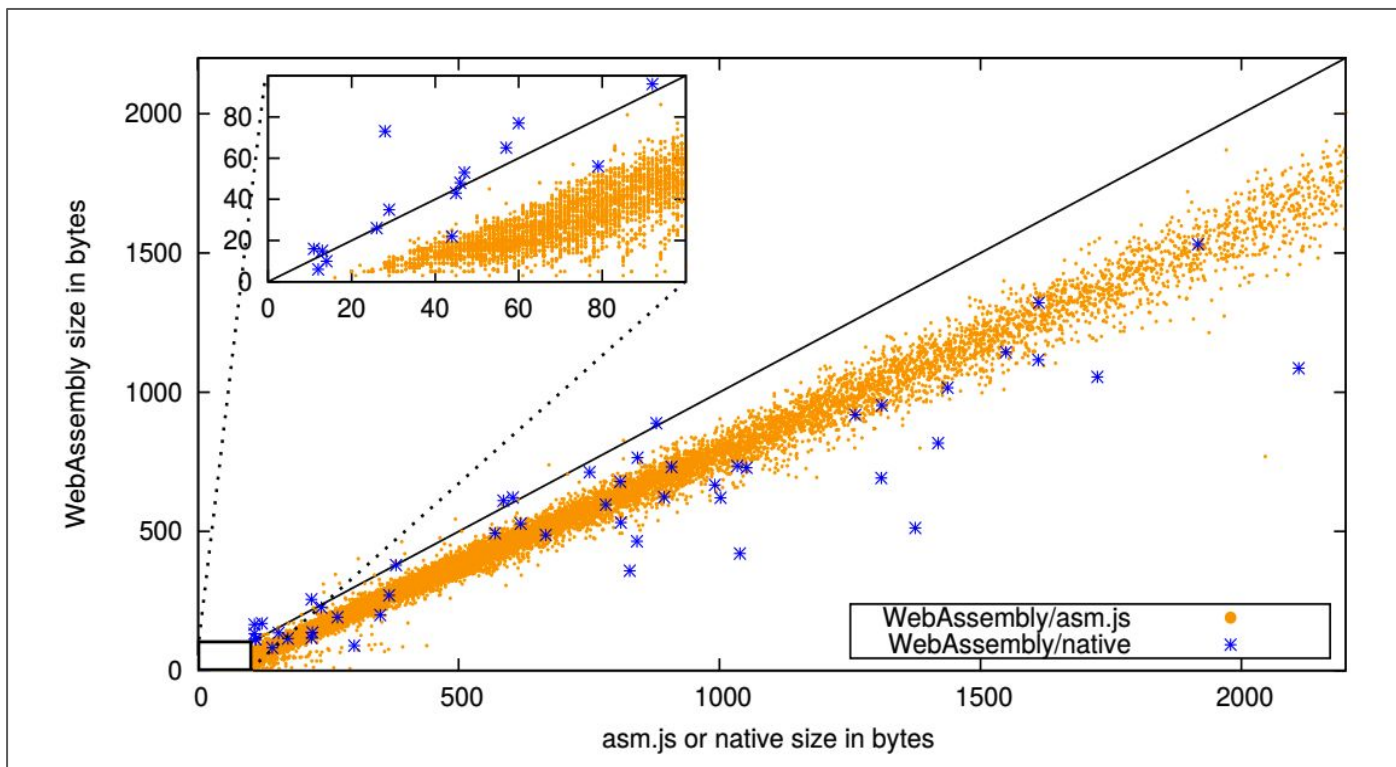
What about versus C?



Representation Design

- **Compact**, binary representation
- **Modular**, can be split up into smaller parts that can be transmitted, cached and consumed separately
- **Efficient**, can be decoded, validated and compiled in a fast single pass, with a JIT or AOT compilation
- **Streamable**, allows decoding, validation and compilation and fast as possible
- **Parallelizable**, allows validation, compilation and splitting into many parallel tasks.

How good is this binary representation?



Source: <https://dl.acm.org/citation.cfm?id=3062363>

Representations

- **Textual**, human-readable, debuggable.
- **Binary**, actual representation used by the computer. Easy to decode, and smaller to transmit.

Representations - Textual - .wat

- Human readable, textual representation

```
(module $main
  (import "console" "mem" (func $print (param i32 i32) (result i32)))
  (import "mem" "main" (memory 1))
  (global $MEM_TOP i32 (i32.const 16))
  (table 0 anyfunc)
  (data (i32.const 16) "Hello World\n\00")
  (export "sayHello" (func $hello))
  (func $hello (; 1 ;) (result i32)
    get_global $MEM_TOP
    i32.const 12
    call $print
  )
)
```

- Compile to wasm: `$wat2wasm say_hello.wat -o say_hello.wasm`

Representations - binary - .wasm

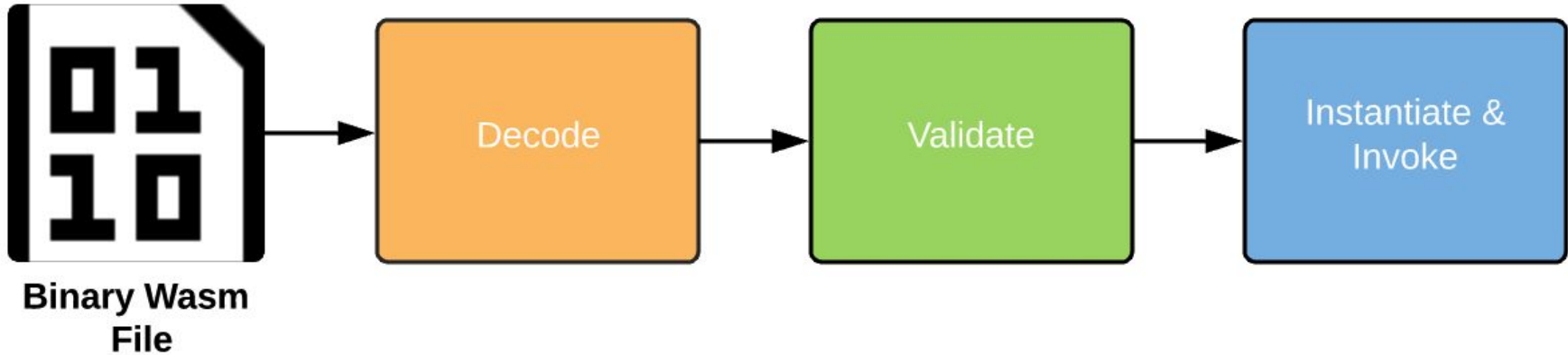
- Binary representation

```
→ src git:(master) x hexdump -C print_string.wasm
00000000  00 61 73 6d 01 00 00 00  01 0b 02 60 02 7f 7f 01  |.asm.....`....|
00000010  7f 60 00 01 7f 02 1b 02  07 63 6f 6e 73 6f 6c 65  |.`.....console|
00000020  03 6d 65 6d 00 00 03 6d  65 6d 04 6d 61 69 6e 02  |.mem...mem.main.|
00000030  00 01 03 02 01 01 04 04  01 70 00 00 06 06 01 7f  |.....p.....|
00000040  00 41 10 0b 07 0c 01 08  73 61 79 48 65 6c 6c 6f  |.A.....sayHello|
00000050  00 01 0a 0a 01 08 00 23  00 41 0c 10 00 0b 0b 13  |.....#.A.....|
00000060  01 00 41 10 0b 0d 48 65  6c 6c 6f 20 57 6f 72 6c  |..A...Hello Worl|
00000070  64 0a 00                                     |d..|
00000073
```

WebAssembly Types

- WebAssembly is a stack-based IR, similar to the JVM.
- Four basic types:
 - i32, i64, f32, f64
- There is no distinction between signed and unsigned integer types, instead operations are specialized to be signed or unsigned.
- There is a full-matrix of operations for conversions between the types
- i32 integers serve as booleans, addresses, and values.

WebAssembly Pipeline



Decoding/Encoding

- This follows a simple Grammar! Or Binary Grammar, just like the ones you have been doing for your assignments!
- **Procedure:** Decode from binary to hex, then use the grammar!

Let's go see some rules of this grammar

- Bytes encode themselves

```
byte ::= 0x00 ⇒ 0x00
      |     ...
      | 0xFF ⇒ 0xFF
```

- Types:

```
valtype ::= 0x7F ⇒ i32
          | 0x7E ⇒ i64
          | 0x7D ⇒ f32
          | 0x7C ⇒ f64
```

Source: <https://webassembly.github.io/spec/core/binary/types.html>

Validation

- Usually done along with decoding, in one pass
- All declarations, imports and function types defined on top of the file

Validation

- Usually done along with decoding, in one pass
- All declarations, imports and function types defined on top of the file

What to validate?

Validation

- Usually done along with decoding in one pass
- All declarations, imports and function types defined on top of the file

What to validate?

- Decoded values for a given type are valid for that type and within appropriate limits, i.e. an i32 literal does not overflow

Validation

- Usually done along with decoding in one pass
- All declarations, imports and function types defined on top of the file

What to validate?

- Decoded values for a given type are valid for that type and within appropriate limits, i.e. an i32 literal does not overflow
- ***Stack, what about the stack?***

Stack Validation

- Similar to **JVM** stack height must remain consistent after each instructions
- Stack contents must have the right type after each operation
- Examples:
 - At the end of a function, we have the right type and height for returning.
 - When we set a local of certain type, the stack height is of at least 1 and has the same type as the local.
 - When adding two i32 numbers, the stack height decreases by 1 and the type on top of the stack is i32.

Stack Validation

- Similar to **JVM** stack height must remain consistent after each instructions
- Stack contents must have the right type after each operation
- Examples:
 - At the end of a function, we have the right type and height for returning.
 - When we set a local of certain type, the stack height is of at least 1 and has the same type as the local.
 - When adding two i32 numbers, the stack height decreases by 1 and the type on top of the stack is i32.
- **Validation** has been formally defined in terms of type rules.

Validation - Set/Get Local

- Usage:

```
;; set_local  
(local $arg1 i32)  
i32.const 32  
set_local $arg1
```

```
;; get_local  
(local $arg1 i32)  
  
get_local $arg1
```

- Validation Typerules:

$$\frac{C. \text{locals}[x] = t}{C \vdash \text{set_local } x : [t] \rightarrow []}$$

$$\frac{C. \text{locals}[x] = t}{C \vdash \text{get_local } x : [] \rightarrow [t]}$$

Validation - Let's see a few examples

- **Binops:** add | sub | mul | div_sx | rem_sx | and | or | xor | shl | shr_sx | rotl | rotr
 - **Usage:**

```
(param $arg1 f64)
get_local $arg1
f64.const 5.0
f64.add
```

```
(param $arg1 i32)
get_local $arg1
i32.const 5.0
i32.add
```

- **Validation Typerule:**

$$\frac{}{C \vdash t.binop : [t\ t] \rightarrow [t]}$$

Execution

- Finally after a module is verified, it goes through two phases:
 - **Instantiation:** Dynamic representation of a module, the module imports are loaded, global tables, and memory segments are initialized, and its own execution stack and state are set. Finally its start function is ran.
 - **Invocation:** Once instantiated, a module instance is ready to be used by its host/embedding environment via the exported functions defined in the module.
 - The task of Instantiation and invocation is the responsibility of the host environment.

Stack

- We talked about validation of the stack, similar to *static typing*.
- During execution we care about the actual values
- Again, this was formally defined using formal **reduction rules**.

Stack

- We talked about validation of the stack, similar to *static typing*.
- During execution we care about the actual values
- Again, this was formally defined using formal **reduction rules**.
- There are three types of stack contents
 - **Values**, i32, i64, f32, f64 constants.
 - **Labels**, branching labels/targets
 - **Frames**, a function's run-time representation
- Other implementations may choose to have three separate stacks but the interleaving of the three stack values makes the implementation simpler.

Let's take a step back - control flow instructions

- WebAssembly, unlike other low-level languages is based on **structured control flow**
- **if/else:**

```
i32.const 1 ;; [1]
if (result i32) ;;[]
|   ;; instruction*
else
|   ;; instruction*
end
;;[t:i32]
```

loop/block/br statements

C while loop

```
int i;  
i = 0;  
while(i<5)  
{  
    // instructions  
    i++;  
}
```

WebAssembly

```
i32.const 0  
set_local $i } Initialization  
loop $l1  
    block $l0  
        get_local $i }  
        i32.const 5 } Condition  
        i32.ge_s }  
        br_if $l0 }  
        ;; instructions  
        i32.const 1 }  
        get_local $i } Increase  
        i32.add } counter  
        set_local $i }  
        br $l1 }  
    end  
end
```

Stack Contents

```
;;[...] -> [..., 0]  
;; [..., 0] -> [...]  
;; [...] -> [..., $l1] (loop start)  
;; [..., $l1] -> [..., $l1, $l0]  
;; [..., $l1, $l0] -> [..., $l1, $l0, $i]  
;; [..., $l1, $l0, $i] -> [..., $l1, $l0, $i, 5]  
;; [..., $l1, $l0, $i, 5] -> [..., $l1, $l0, (i>=5)]  
;; if 1: [..., $l1, $l0] -> [...]  
;; if 0: [..., $l1, $l0] -> [..., $l1, $l0]  
;; [..., $l1, $l0] -> [..., $l1, $l0, 1]  
;; [..., $l1, $l0, 1] -> [..., $l1, $l0, 1, $i]  
;; [..., $l1, $l0, 1, $i] -> [..., $l1, $l0, $i+1]  
;; [..., $l1, $l0, $i+1] -> [..., $l1, $l0]  
;; [..., $l1, $l0] -> [...] Jumps back to  
;; (loop start)
```

loop/block/br statements

WebAssembly

```
i32.const 0 } Initialization  
set_local $i }  
loop $l1  
  block $l0  
    get_local $i }  
    i32.const 5 } Condition  
    i32.ge_s }  
    br_if $l0 }  
    ;; instructions  
    i32.const 1 }  
    get_local $i } Increase  
    i32.add } counter  
    set_local $i }  
    br $l1 }  
  end  
end
```

Stack Contents

```
;;[...] -> [..., 0]  
;;[...] -> [..., 0]  
;;[...] -> [..., $l1] (loop start)  
;;[...] -> [..., $l1, $l0]  
;;[...] -> [..., $l1, $l0, $i]  
;;[...] -> [..., $l1, $l0, $i, 5]  
;;[...] -> [..., $l1, $l0, $i, 5] -> [..., $l1, $l0, (i >= 5)]  
;;if 1: [...] -> [...]  
;;if 0: [...] -> [..., $l1, $l0]  
;;[...] -> [..., $l1, $l0, 1]  
;;[...] -> [..., $l1, $l0, 1, $i]  
;;[...] -> [..., $l1, $l0, 1, $i] -> [..., $l1, $l0, $i+1]  
;;[...] -> [..., $l1, $l0, $i+1] -> [..., $l1, $l0]  
;;[...] -> [...] Jumps back to  
;; (loop start)
```

Falling through **end** of block

- As mentioned, WebAssembly execution has been formally defined in terms of ***small-steps rules***

```
block $l1 (result i32) ;; [..., $l1]
|
|   i32.const 1           ;; [..., $l1, 1]
end ;; popping 1 i32 value
|   ;; results in $l1 on top.
;; [..., 1]
```

Falling through **end** of block

- As mentioned, WebAssembly execution has been formally defined in terms of *small-steps rules*

```
block $l1 (result i32) ;; [..., $l1]
|
|   i32.const 1           ;; [..., $l1, 1]
end ;; popping 1 i32 value
|   ;; results in $l1 on top.
;; [..., 1]
```

$$\text{label}_n \{ \text{instr}^* \} \text{val}^m \text{end} \hookrightarrow \text{val}^m$$

func end_block(stack,L)

// Let **m** be the number of values on
// top of the stack

- Pop **m** values from stack
- **Assert:** Due to validation, **L** should be the label on top of the sack
- Pop **L**
- Push the **m** values back in the stack
- Jump to instruction immediately after block

end

Examples

Example - Factorial

```
int factorial(int n) {  
    int i, sum;  
    sum = 1;  
    i = 2;  
    while (i <= n) {  
        sum = sum * i;  
        i = i + 1;  
    }  
    return sum;  
}
```

```
(func $factorial (param $n i32) (result i32)  
  (local $i i32)(;int i;) (local $sum i32)(;int n;)  
  ;; sum=1;  
  i32.const 1 ;; ;push i32 1 onto stack-> [1]  
  set_local $sum ;; sum = 1; pop top from stack set $sum  
  ;; i=2;  
  i32.const 2;; ;push i32 2 onto stack-> [1]  
  set_local $i ;; i=2; pop top from stack set $i  
  ;;....while....
```

Example - Factorial

```
int factorial(int n) {  
    int i, sum;  
    sum = 1;  
    i = 2;  
    while (i <= n) {  
        sum = sum * i;  
        i = i + 1;  
    }  
    return sum;  
}
```

```
;; while(i<=n)  
loop $l0 ;;@1  
    block $l1;;@0  
        ;;Evaluate condition  
        get_local $i ;; load i  
        get_local $n ;; load n  
        i32.gt_s ;; i > n  
        br_if $l1 ;; if i > n go to end of block  
        ;; sum = sum * i;  
        get_local $sum ;; ;push value of $sum onto stack  
        get_local $i ;; ;push value of $i onto stack  
        i32.mul ;; sum * i; pop top two values, push i32 result  
        set_local $sum ;; sum = sum * i;  
        ;; i = i+1;  
        get_local $i ;; load i onto stack  
        i32.const 1 ;; load 1 onto stack  
        i32.add ;; pop $i and 1, add and push i32 result  
        set_local $i ;; pop result and set i  
        br $l0 ;; Break to beginning of loop  
    end $l1;;@0  
end $l0;;@1  
;; return sum;  
get_local $sum ;; push local $sum to stack  
return
```

Textual Representation - S-Expressions

- Finally the textual presentation, can be compressed a little bit by the use of *s-expressions*
- Parenthesis indicate the start of a new child
- The order of evaluation is **child** then **parent**
- For a binary operation, **left child, right child, parent.**
- Example:

```
i32.const 42  
i32.const 121  
i32.add
```

```
(i32.add (i32.const 42) (i32.const 121))
```

Example - Factorial

```
int factorial(int n) {  
    int i, sum;  
    sum = 1;  
    i = 2;  
    while (i <= n) {  
        sum = sum * i;  
        i = i + 1;  
    }  
    return sum;  
}
```

```
(func $factorial (param $n i32) (result i32)  
  (local $i i32)(;int i;) (local $sum i32)(;int n;)  
  (set_local $sum (i32.const 1)) ;; sum=1;  
  (set_local $i (i32.const 2)) ;; i=2;  
  loop $l0 ;;@l ;; while(i<=n)  
    block $l1;;@0  
      ;;Evaluate condition  
      (i32.gt_s (get_local $i) (get_local $n));; i > n  
      br_if $l1 ;; if i > n go to end of block  
      ;; sum = sum * i;  
      (i32.mul (get_local $sum) (get_local $i))  
      ;; i = i+1;  
      (set_local $i (i32.add (get_local $i)(i32.const 1)))  
      br $l0 ;; Break to beginning of loop  
    end $l1;;@0  
  end $l0;;@l  
  (return (get_local $sum)) ;; return sum;  
)
```

The End

Thank you!