

Spletne tehnologije

Uvod v JavaScript pogone (2. del)

Niko Lukač

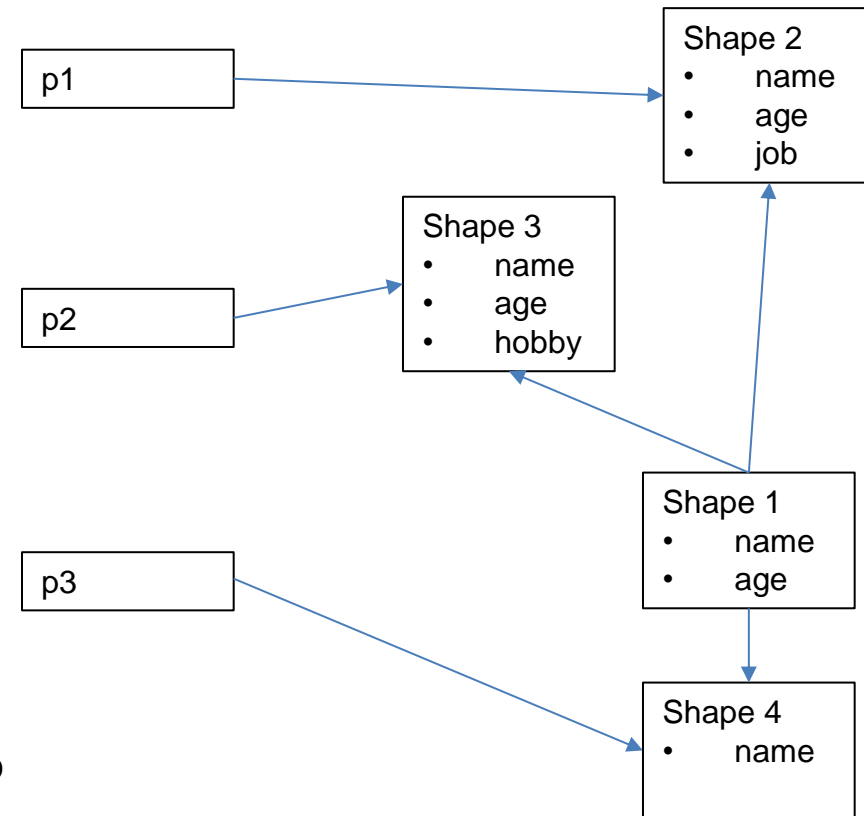
Oblike objektov

- Primer 1:

```
function Person(name, age) {  
  this.name = name;  
  this.age = age;  
}  
var p1 = new Person("Adam", 20);  
var p2 = new Person("Eve", 20);  
//1. snapshot  
p1.job = "Developer";  
p2.hobby = "Reading";  
//2. snapshot  
var p3 = new Person("Bob", 30);  
delete p3.age  
//3. snapshot
```

- Testirajmo v Chrome Devtools (ctrl+shift+i) in inkrementalno pogledamo odtise pomnilnika (Memory heap snapshot)

- Tranzicijska veriga notranjih oblik:



Oblike objektov

- Primer 1:

```
function Person(name, age) {  
    this.name = name;  
    this.age = age;  
}  
var p1 = new Person("Adam", 20);  
var p2 = new Person("Eve", 20);  
var p3 = new Person("Bob", 30);
```

- Testirajmo še IC:

```
get_name = x => { return(x.name); }  
persons=[p1, p2, p3]  
  
start=performance.now()  
for(i=0;i<100000000;i++)  
    get_name(persons[i & Math.floor(Math.random() * (persons.length-1)) ])  
console.log(performance.now()- start)
```

```
// testiramo delovanje IC z naslednjimi ukazi  
// monomorfno ali polimorfno delovanje?  
//p1.job = "Developer";  
// p2.hobby = "Reading";  
// delete p2.age
```

```
start=performance.now()  
for(i=0;i<100000000;i++)  
    get_name(persons[i & Math.floor(Math.random() * (persons.length-1)) ])  
console.log(performance.now() - start)
```

Oblike objektov

- Primer 2:

```
class Person1 {  
  constructor(name) {  
    this.mojeime = name;  
  }  
  test() {  
    console.log("hi")  
  }  
}
```

```
var p1 = new Person1("Bob")
```

```
var p2 = { mojeime: "Janez", test: function() { console.log("hi") } }
```

```
var p3 = { }
```

```
p3.mojeime="Alice"
```

```
var p4 = { }
```

```
p4.mojeime="Lisa"
```

```
p3.test=function() { console.log("hi") }
```

```
p4.test=function() { console.log("hi") }
```

- Enako kot če bi zapisali (spomnimo, da je class sintaktična olepšava):

```
function Person1(name) {  
  this.mojeime = name;  
}  
Person1.prototype.test=function() {console.log("hi") }
```

Oblike objektov

- Primer 2:

```
class Person1 {  
  constructor(name) {  
    this.mojeime = name;  
  }  
  test() {  
    console.log("hi")  
  }  
}
```

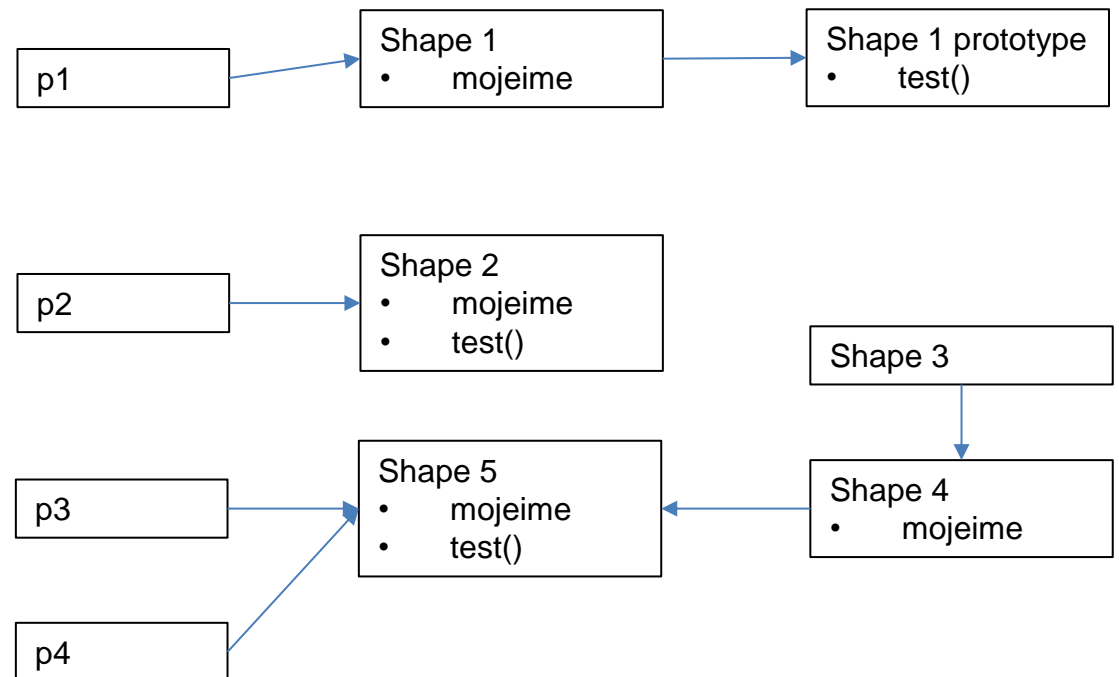
```
var p1 = new Person1("Bob")
```

```
var p2 = { mojeime: "Janez", test: function() { console.log("hi") } }
```

```
var p3 = { }  
p3.mojeime="Alice"  
var p4 = { }  
p4.mojeime="Lisa"  
p3.test=function() { console.log("hi") }  
p4.test=function() { console.log("hi") }
```

- Tranzicijska veriga oblik

- Preverimo še v Chrome Devtools (v memory snapshot iščemo po lastnosti preko ctrl+f in ne glavnega iskalnega okna)



Dedovanje prototipov

- Primer:

```
function Parent() { this.delta = function(n) { return n; } };
```

```
function A(){}; A.prototype = new Parent();
```

```
function B(){}; B.prototype = new A();
```

```
function C(){}; C.prototype = new B();
```

```
function D(){}; D.prototype = new C();
```

```
function E(){}; E.prototype = new D();
```

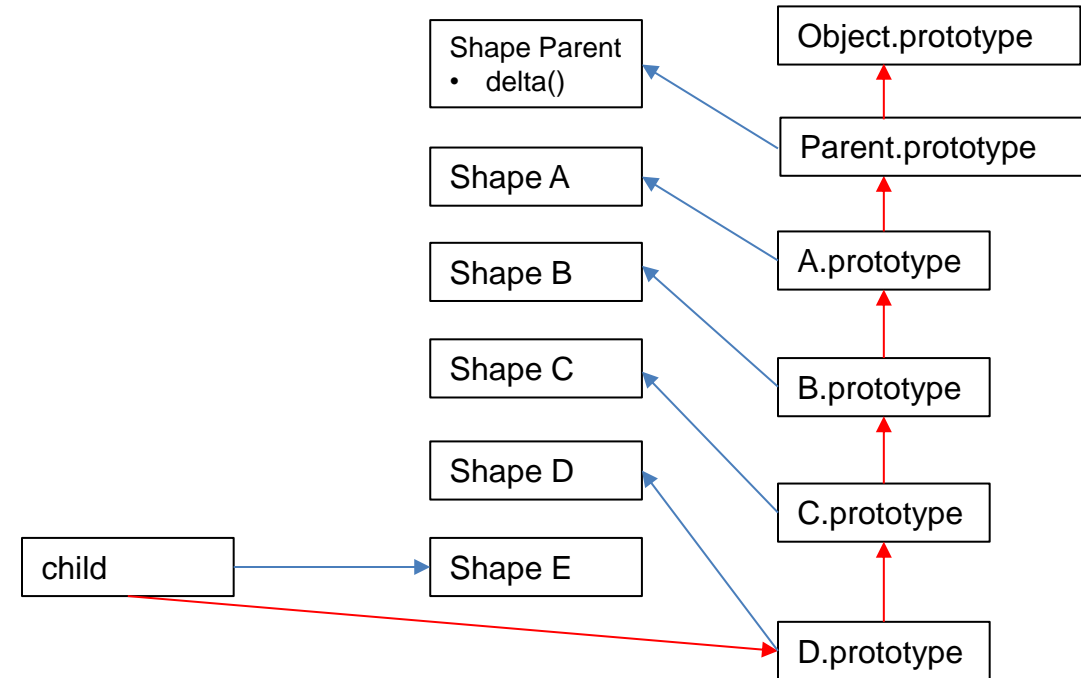
```
function test() {  
  var child = new E();  
  var counter = 0;  
  for(var i = 0; i < 100000; i++) {  
    // Object.prototype.x=5 // testiramo validacijsko celico , poskusimo še ostale prototipe  
    counter += child.delta(10);  
    // delete Object.prototype.x // testiramo validacijsko celico, poskusimo še ostale prototipe  
  }  
}
```

```
start = performance.now();
```

```
test();
```

```
console.log(performance.now() - start)
```

- Tranzicijska veriga s prototipi:



Profiliranje pomnilnika

- Primer:

```
{  
var primer1 = { mojeime: "test"};  
  
start = performance.now();  
for (var i = 0; i < 100000000; i++) primer1[i] = i;  
console.log(performance.now() - start);  
}
```

- Poglejmo v Chrome DevTools (ctrl+shift+i):
 - Odtis pomnilnika
 - Delovanje GC: performance (Memory) -> Record
- Sprotno alociranje pomnilnika bolj obremenjuje GC
- Interno ni težav z oblikami (polja se hranijo v **elements** spremenljivki)

Uhajanje pomnilnika

- Primer:

```
((() => {  
    // testirajte še z večjo velikostjo  
    var arr = new Int8Array(1024*1024*1024);  
    arr.fill(0)  
    setInterval(function() {  
        console.log('.');  
        var node = document.getElementById('Node');  
        if(node) {  
            node.innerHTML = arr;  
        }  
    }, 1000);  
})();
```


Spletne tehnologije

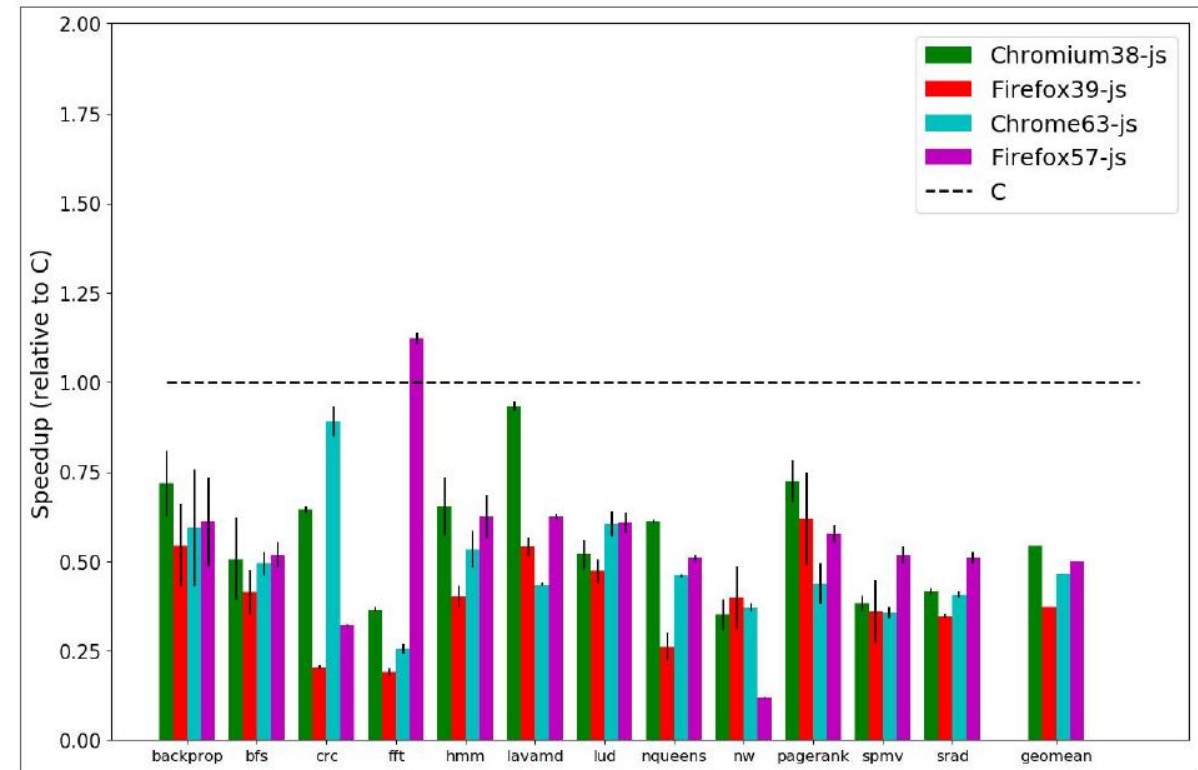
Alternative JavaScripta

Niko Lukač

Motivacija

- **Dinamično tipiziran** jezik kot je JS ne omogoča zmogljivosti nižjih jezikov kot **C** ali **C++**
- Želimo se v določenih spletnih aplikacijah približat meji zmogljivosti strojnega nivoja (**native**)
- Pravi **paralelizem** ni podprt, vsaj ne na strojnem nivoju. Razširjena množica **podatkovnega paralelizma** (SIMD) ni implicitno podprta

JS vs C



Kratka zgodovina...

- HTML (1991)
- JavaScript (1995)
- Vtičniki (plugins):
 - NPAPI (1995-2005)
 - ActiveX (1996-2015)
 - Flash (1996-)
 - Java Applet (1996-)
- Težave vtičnikov ?



Kratka zgodovina...

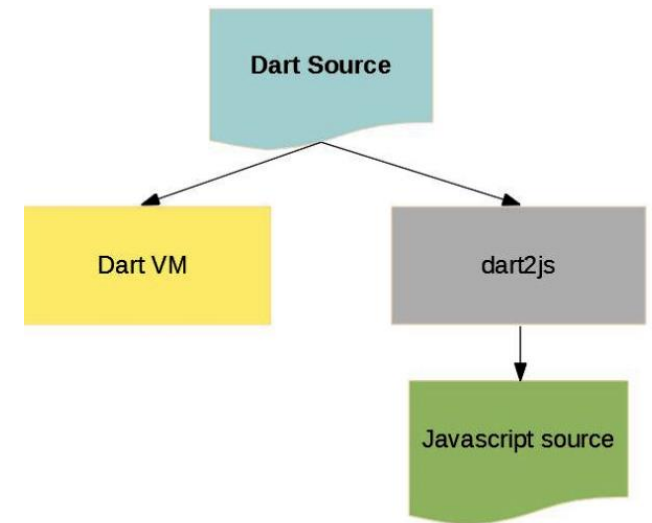
- HTML (1991)
- JavaScript (1995)
- Vtičniki (plugins):
 - NPAPI (1995-2005)
 - ActiveX (1996-2015)
 - Flash (1996-)
 - Java Applet (1996-)
- **Težave vtičnikov ?**
 - Namenska namestitev
 - Namenska uporaba
 - Težave z varnostjo
 - Slaba prenosljivost
- Nove rešitve:
 - DART (2012), Emscripten, **asm.js** (2010, 2013+)
 - **WebAssembly** (2016+)

	Year	Secure	Portable	Ephemeral	Cross Browser	Shared Memory
JavaScript	1995	✓	✓	✓	✓	✗
NPAPI	1995	✗	✗	✗	✗	✓
ActiveX	1996	✗	✗	✗	✗	✓
Flash	1996	~	✓	✓	✗	✗
Java Applets	1996	~ / ✗	✓	✓	✗	✓
Native Client	2008	✓	~	✓	✗	✓
Emscripten	2010	✓	✓	✓	✓	✗
asm.js	2013	✓	✓	✓	~	✗
PNACL	2013	✓	✓	✓	✗	✓
Web Assembly	2016 ?	✓	✓	✓	✓	✓

Vir slike: Nick Bray, Google

DART

- Objektno-orientiran programski jezik razvit s strani Googla v letu 2012, s **sintakso podobno** C, prevede se lahko v strojno kodo ali v JavaScript. Dart VM podpira tudi **JIT**.
- Originalno je bil plan, da **se Dart VM vključi v Chrome**, vendar se to ni uresničilo zaradi premočnosti JS in V8 pogona.
- Trenutno je Dart možno poganjat za Web aplikacije preko **source-to-source prevajnika** v JS.
- Popularna je uporaba za mobilne aplikacije preko **ogrodja Flutter (za UI)**, ki omogoča Dart VM in izvajanje na strojnem nivoju z uporabo **AOT prevajanja**.



Dart

DART benchmark

- Dart JIT vs Dart AOT:

k-nucleotide

source	secs	mem	gz	busy	cpu load
<u>Dart</u>	16.66	326,548	1520	49.39	77% 71% 73% 75%
<u>Dart JIT</u>	24.52	533,456	1520	64.03	38% 81% 99% 43%

spectral-norm

source	secs	mem	gz	busy	cpu load
<u>Dart</u>	1.48	25,380	1196	5.72	97% 97% 97% 95%
<u>Dart JIT</u>	2.00	199,348	1196	6.57	79% 79% 82% 88%

n-body

source	secs	mem	gz	busy	cpu load
<u>Dart</u>	7.30	10,352	1311	7.89	100% 0% 4% 4%
<u>Dart JIT</u>	8.97	125,576	1311	9.17	2% 97% 2% 1%

pidigits

source	secs	mem	gz	busy	cpu load
<u>Dart</u>	3.14	43,200	500	3.19	1% 0% 99% 1%
<u>Dart JIT</u>	3.43	152,196	500	3.68	5% 94% 5% 3%

- Dart AOT vs node.js (V8):

n-body

source	secs	mem	gz	busy	cpu load
<u>Dart</u>	7.30	10,352	1311	7.89	100% 0% 4% 4%
<u>Node js</u>	8.60	36,000	1268	8.85	100% 2% 0% 1%

spectral-norm

source	secs	mem	gz	busy	cpu load
<u>Dart</u>	1.48	25,380	1196	5.72	97% 97% 97% 95%
<u>Node js</u>	1.64	67,912	999	6.11	92% 93% 96% 92%

k-nucleotide

source	secs	mem	gz	busy	cpu load
<u>Dart</u>	16.66	326,548	1520	49.39	77% 71% 73% 75%
<u>Node js</u>	15.82	396,716	1812	44.48	63% 47% 81% 90%

fannkuch-redux

source	secs	mem	gz	busy	cpu load
<u>Dart</u>	12.70	15,288	1220	50.67	100% 100% 100% 100%
<u>Node js</u>	11.38	65,452	1313	45.00	99% 99% 99% 99%

asm.js



- Moderni začetek za približanje **zmogljivosti JS na strojnem nivoju**
- Možna pretvorba C in C++ kode v LLVM ter nato v **poenostavljen dialekt JS** preko **Emscripten** prevajalnika.
- asm.js ni JS (tako npr. nimamo dostopa do GC)

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



```
function f(i) { // asm.js
  "use asm";
  i = i|0;
  return (i + 1)|0;
}
```

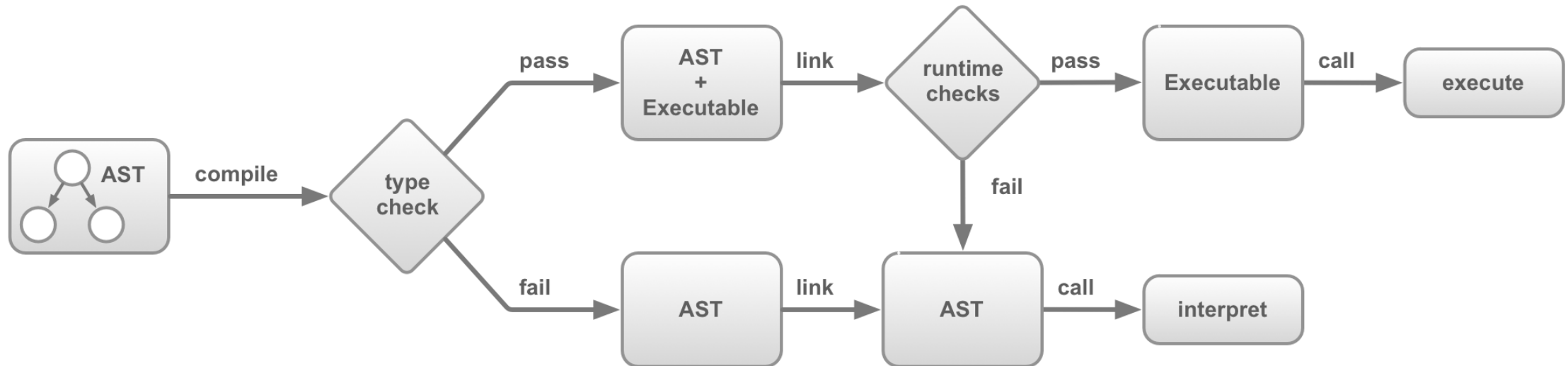
Predstavitev tipov spremenljivk:

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

Od 2013 lahko za tipizirana polja uporabimo **Uint32Array**, itd.

asm.js

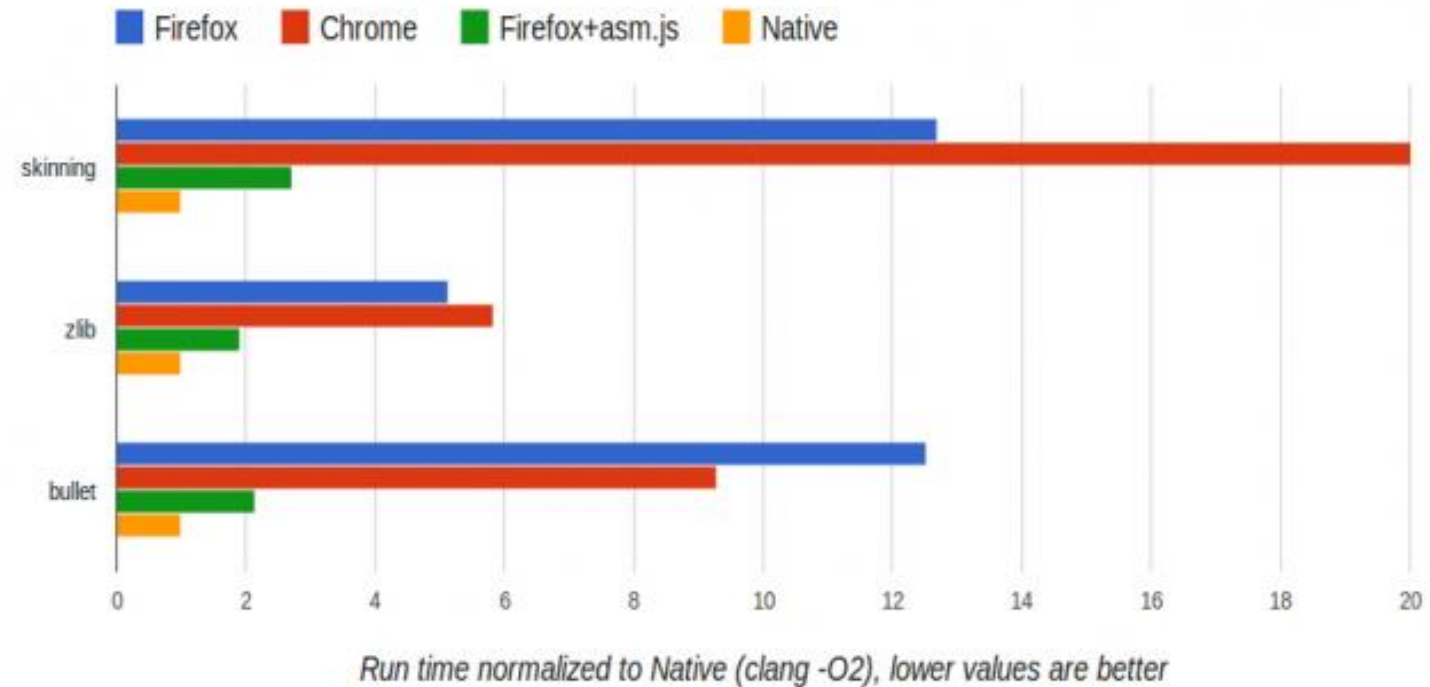
- Praviloma **uporablja AOT**, če **statična in dinamična analiza** uspešna, sicer uporablja klasični interpreter



Vir slike: <http://asmjs.org/spec/latest/>

Omejitve asm.js

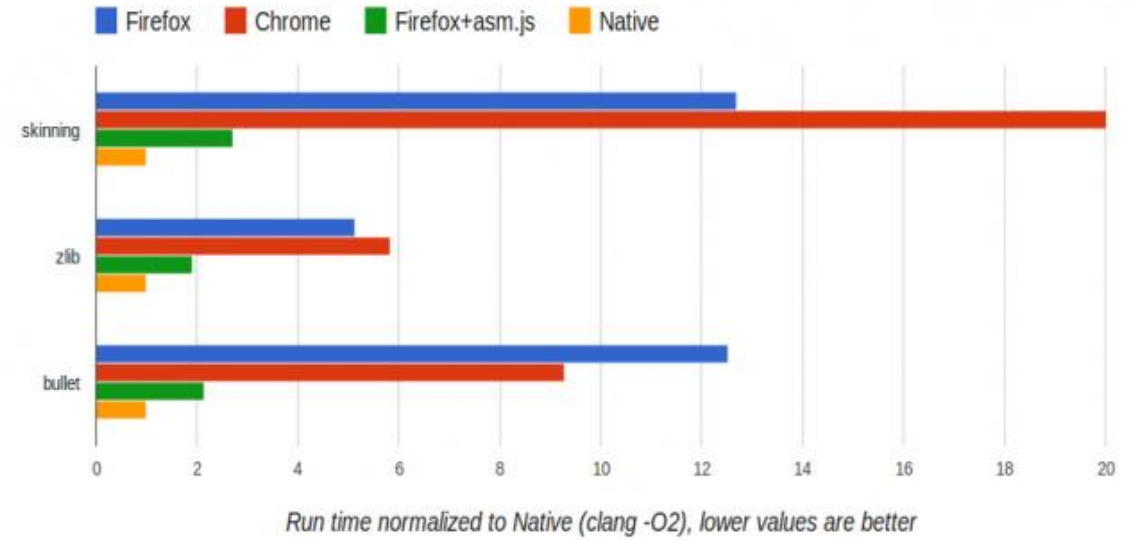
- Kljub višji hitrosti v primerjavi z JS kodo, še vedno **ne dosegamo hitrosti na strojnem nivoju**.
- Omejitve asm.js ?



Benchmark iz 2013, vir: Extremetech

Omejitve asm.js

- Kljub višji hitrosti v primerjavi z JS kodo, še vedno ne dosegamo hitrosti na strojnem nivoju.
- Ne izkorišamo **SIMD (single instruction multiple data)** in ostalih paralelnih podatkovnih inštrukcij, ki jih podpirajo moderni CPU
- Prevajanje in nalaganje asm.js modula je počasno, **kompresiran način prenosa kode se rešuje ad-hoc** (npr. gzip)
- Slaba **skalabilnost**: če želimo več podpore v asm.js, je ta predvsem odvisna od razvoja JS
- Avtomatsko sproščanje pomnilnika ni omogočeno (**garbage collection**), zato slabe implementacije uhajajo pomnilniški prostor



Benchmark iz 2013, vir: Extremetech

WebAssembly



- WebAssembly je splošno namenski **nizkonivojski zbirni jezik**
- Nastopa kot vmesnik med višjim programskimi jeziki ter strojno kodo
 - Omogoča celotno arhitekturo inštrukcij (paradigma **ISA – Instruction set architecture**), ki se enostavno preslikajo v strojne inštrukcije (machine code) danega CPU (npr. x86-64), kar omogoča **doseganje hitrosti blizu strojnega nivoja**
 - **Neodvisen od vhodnega jezika** (npr. C++, Rust itd) in **neodvisen od strojne opreme**
 - **Neodvisen od platforme** (npr. JS pogon), saj se lahko izvaja tudi ločeno v svojem virtualnem stroju (VM) - npr. **Mozilla WebAssembly System Interface**

Zakaj WebAssembly



- Podpora **različnih tipov**
- Podpora podatkovnega paralelizma (**SIMD** inštrukcije itd) (novo, delno v razvoju)
- Podpora strojne večnitnosti (**multithreading**) (novo, delno v razvoju)
- Omogoča tudi avtomatsko čiščenje glavnega pomnilnika (**garbage collection**) (še v razvoju)
- **Modularnost**: možna razdelitev na več neodvisnih manjših delov
- Možno **hitro prevajanje** v strojno kodo (hitrejše od JS), podpora JIT (just-in-time) in AOT (ahead-of-time)
- Kompaknost: možna **binarna predstavitev** inštrukcij
- **Višja varnost**: validacija pomnilniškega prostora (npr. prekoračitve)

Predstavitev WASM

- WebAssembly inštrukcije lahko zapišemo v **tekstovni** (.wat) ali **binarni** (.wasm) obliki (to še vedno **ni strojna koda**)

```
(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
  )
)
```

```
→ src git:(master) ✖ 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.....#.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
```

Kodiranje wat v wasm: `wat2wasm sayhello.wat -o sayhello.wasm`

Obratno: `wasm2wat`

Osnovne lastnosti WebAssembly

- Najbolj osnovni tipi za cela in decimalna števila: **i32, i64, f32, f64**
- Ni razlikovanja predznačenih in nepredznačenih števil (**signed oz. unsigned**), določene operacije delujejo v nepredznačenem načinu
- Možna **implicitna pretvorba** med tipi
- **Hitra pretvorba v binarno obliko** (WASM)
- **i32 se privzeto uporablja za naslove**, binarne (boolean) zastavice in vrednosti

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

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

Osnovne lastnosti WebAssembly

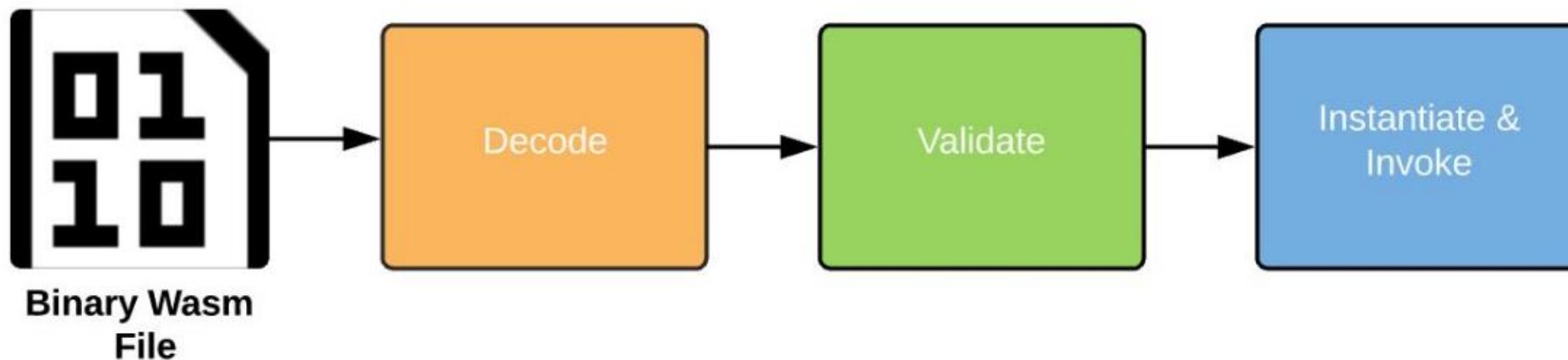
- Več kot 170 osnovnih inštrukcij
 - Gramatika in semantična pravila:
<http://cs242.stanford.edu/f18/lectures/04-2-webassembly-theory.html>
- **Kontrolni del** kode: block, loop, if, else, end
- **Klic funkcij**: call, call_indirect, return
- **Operacije nad pomnilnikom**: load, store, grow_memory itd
- **Aritmetične operacije**: +, -, *, /, %, &, %, itd
- **Konverzija med tipi**: convert, reinterpret, itd
- **Lokalne in globalne spremenljivke**:
get_global, set_global, get_local, set_local

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

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

Cevovod WebAssembly

- Dekodiranje pretvori **wat** v **wasm**.
- Validacija se izvede s **skladovnim strojem (stack machine)**, ki je praviloma hitrejši od **AST (abstract syntax tree)**
- Nato se **prevede (JIT ali AOT)** v strojno kodo, instancira in zažene.



Vir slike: David Herrera, COMP 520

Skladovni stroj WebAssembly

- Vse vrednosti se validirajo (**type safety**), če so v pravih mejah za preprečitev prekoračitve veljavnih mej danega tipa (npr. **integer overflow** pri i32/i64, **buffer overflow**, **stack overflow** itd)
- Dodatna validacija **višine virtualnega sklada** operacij/vrednosti (podobno kot pri JVM) preko **semantičnih pravil** (semantic oz. typerules),
Npr. set in get pri lokalnih spremenljivkah:

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

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

Validation Typerules:

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

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

Skladovni stroj WebAssembly

- Interno se WebAssembly virtualni sklad deli na dva dela:
 - **Kontrolni sklad:** v katerem bloku kode se nahajamo, koda se zaradi vejitev (pogoji in zanke) lahko deli na več posameznih blokov
 - **Sklad vrednosti:** hrani temporarne vrednosti spremenljivk, klici funkcij itd
- Primer enostavnega skladnega stroja:

$x = 2 * (m + n)$

```
pushaddr x
pushconst 2
pushval n
pushval m
add
mult
store
```

m
n
2
@x
?

m + n
2
@x
?

$2 * (m + n)$
@x
?

?

Skladovni stroj WebAssembly

- Še en primer delovanja virtualnega sklada

C while loop

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

WebAssembly

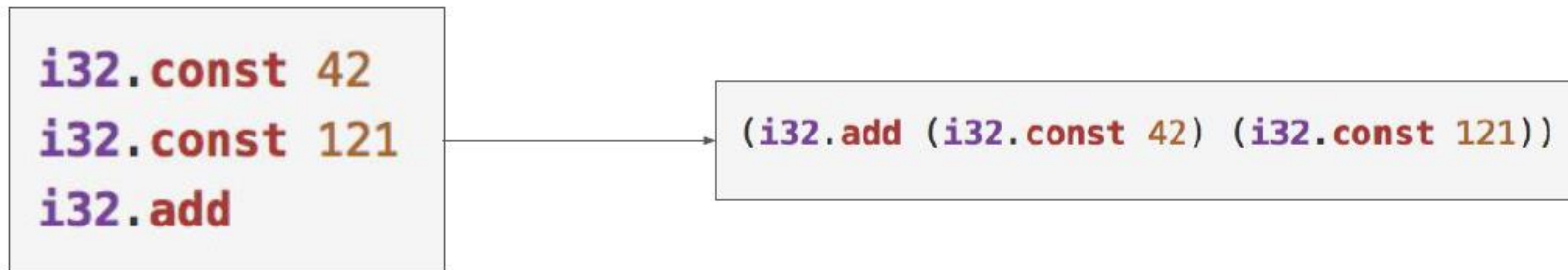
```
i32.const 0  
set_local $i } Initialization  
loop $l1  
    block $l0  
        get_local $i  
        i32.const 5  
        i32.ge_s } Condition  
        br_if $l0  
        ;; instructions  
        i32.const 1  
        get_local $i  
        i32.add } Increase counter  
        set_local $i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

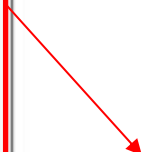
Kompakstna tekstovna predstavitev WASM

- Tekstovna predstavitev (.wat) **se lahko tudi skrči** (s-expression), z ohranjanjem berljivosti inštrukcij, na podlagi sintaktičnega “sladkorčka”.
- Podobno funkcijskim jezikom (npr. LISP) :)



Primer WASM – izračun faktetete

```
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....
```

Primer WASM – izračun faktetete

```
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
```

Zagon WASM

- Primer za pretvorbo C++ kode v WASM ter vključitev v JS
- Npr. prevajanje preko **Emscripten**:

```
em++ -O3 myModule.cpp -s WASM=1 -o myModule.wasm -s "EXPORTED_FUNCTIONS=['_exported_func']"
```

- C koda:

```
extern "C" {  
  
    EMSCRIPTEN_KEEPALIVE  
    void exported_func() {  
        printf("Hi!\n");  
    }  
  
}
```

Zagon WASM

- Primer za pretvorbo C++ kode v WASM ter vključitev v JS
- Npr. prevajanje preko **Emscripten**:

em++ -O3 myModule.cpp -s WASM=1 -o myModule.wasm -s "EXPORTED_FUNCTIONS=['_exported_func']"

- JS koda:

```
WebAssembly.instantiateStreaming(fetch('myModule.wasm'), importObject)
  .then(obj => {
    // Call an exported function:
    obj.instance.exports.exported_func();

    // or access the buffer contents of an exported memory:
    var i32 = new Uint32Array(obj.instance.exports.memory.buffer);

    // or access the elements of an exported table:
    var table = obj.instance.exports.table;
    console.log(table.get(0)());
  })
```

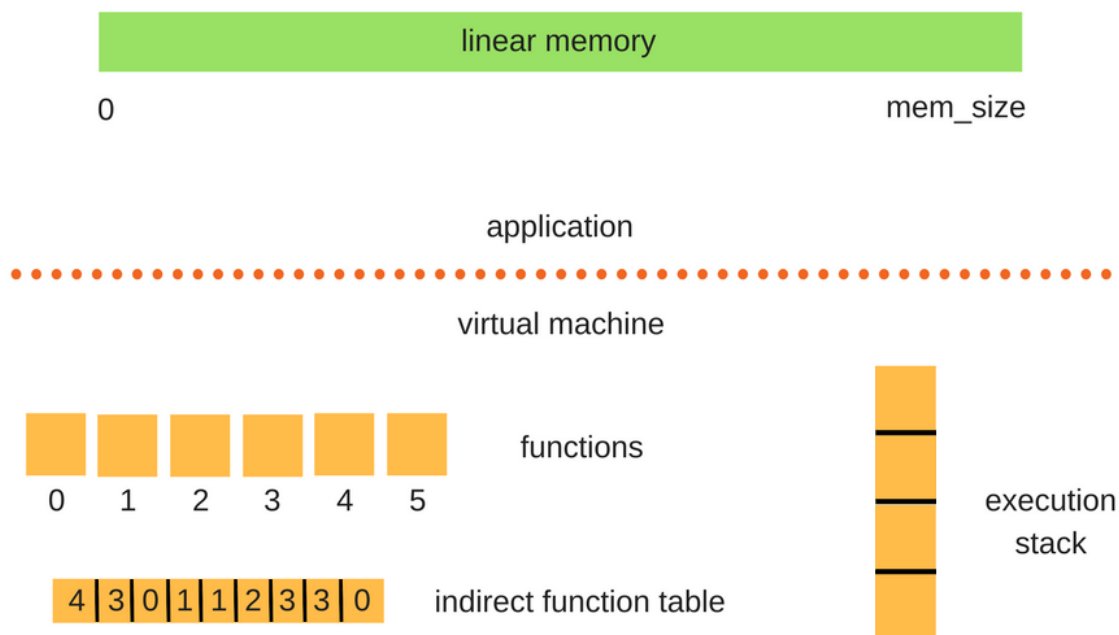
Starejša alternativa od fetch ?

- JS API za WebAssembly:
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/WebAssembly

Arhitekturni pogled WebAssembly

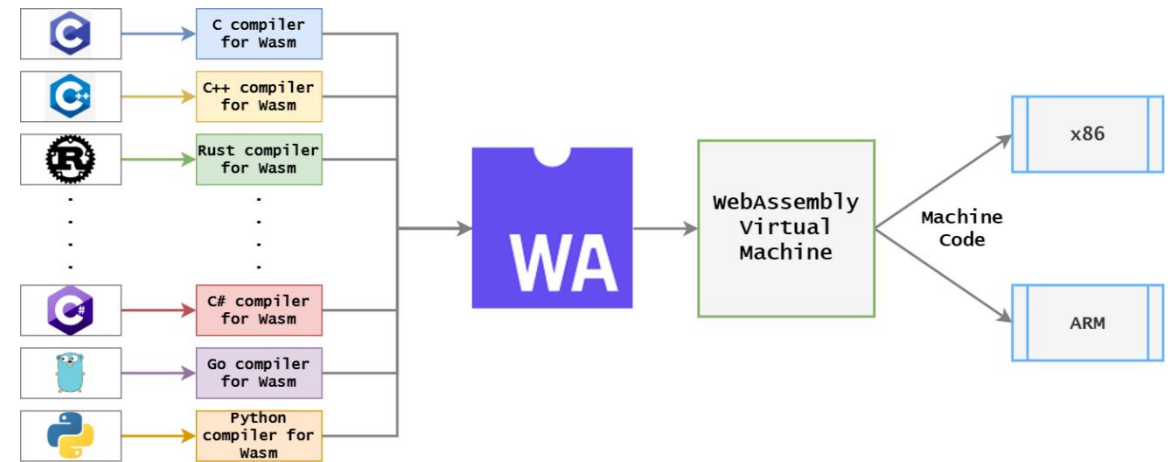
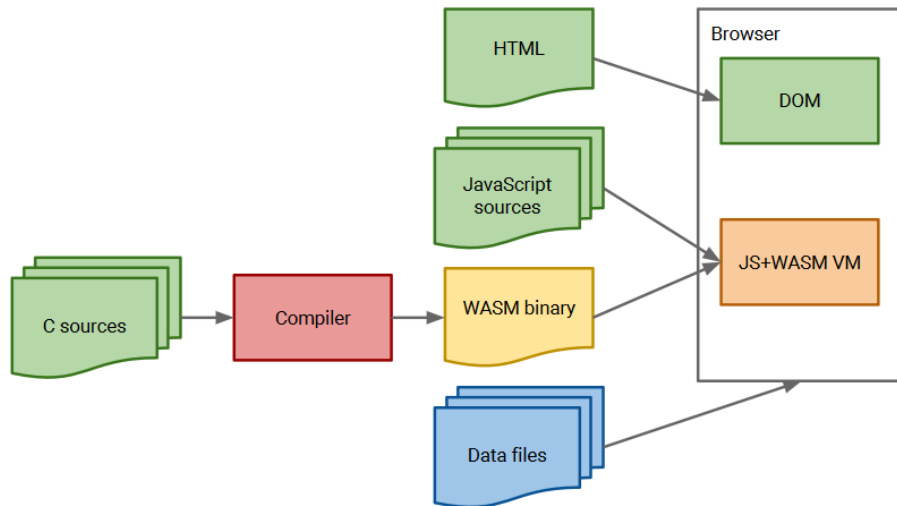
- **Pomnilniška predstavitev**

- V osnovi se uporablja dinamični linearni virtualni pomnilnik za hrambo podatkov
- Maksimalna velikost pomnilnika: 4 GB, za 4 GB obstaja eksperimentalna **wasm64**



Arhitekturni pogled WebAssembly

- Klasično **delovanje v JS pogonih**
 - Praviloma ni **ločenega virtualnega** stroja za izvajanje WASM
- Ali ločen pogon preko WASM VM

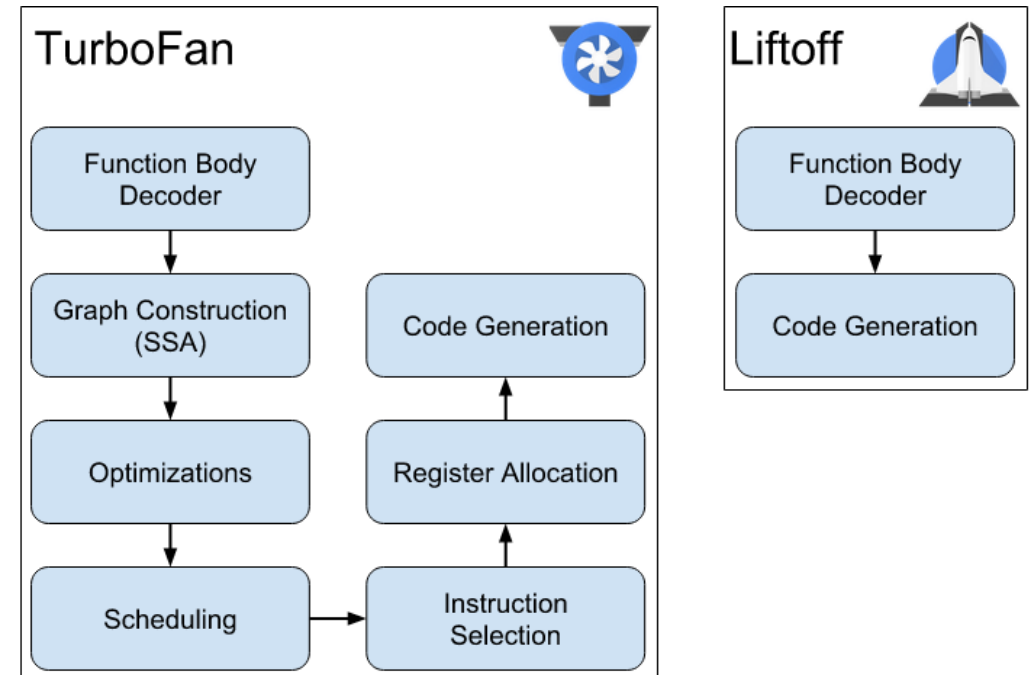


Vir slike: <https://arghya.xyz/articles/webassembly-wasm-wasi/>

Vir slike: David Herrera, COMP 520

Uporaba WASM v V8 JS pogonu

- WASM prevajalnik **Liftoff** deluje poleg **TurboFan** (za JS optimizacijo)
 - **Hitro prevajanje** v strojno kodo (10x hitreje kot z Turbofan)
 - Možno prevajanje ob prenosu (**stream-compilation**) pri prenosu **odsekov wasm/wat datoteke**
- Ob koncu prevajanja z Liftoff uporablja Turbofan v ozadju za dodatno optimizacijo (poleg JS)
- Podprto **predpomnenje (caching)** že prevedenih/optimiziranih WASM (za **ponovno uporabo**)
- WASM v Chrome:
 - Testirajte: <https://earth.google.com/>



Vir slike: <https://v8.dev>

Samostojna (standalone) uporaba WASM

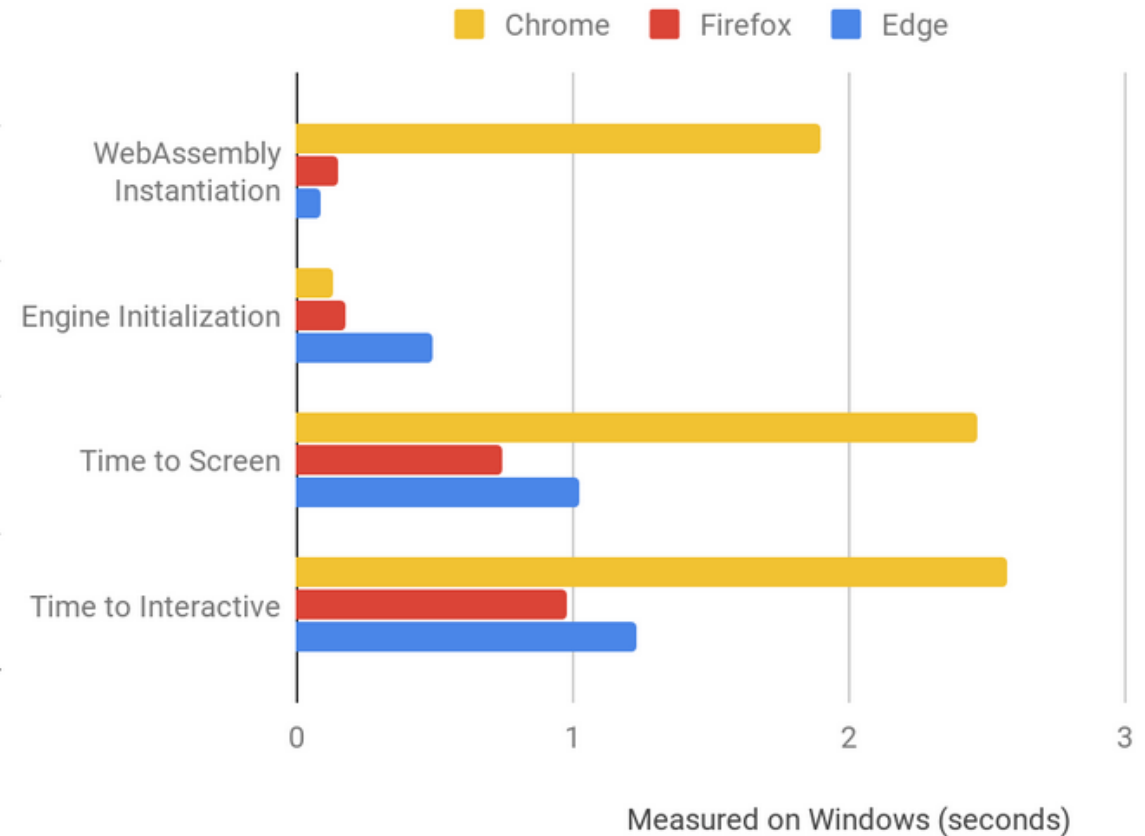
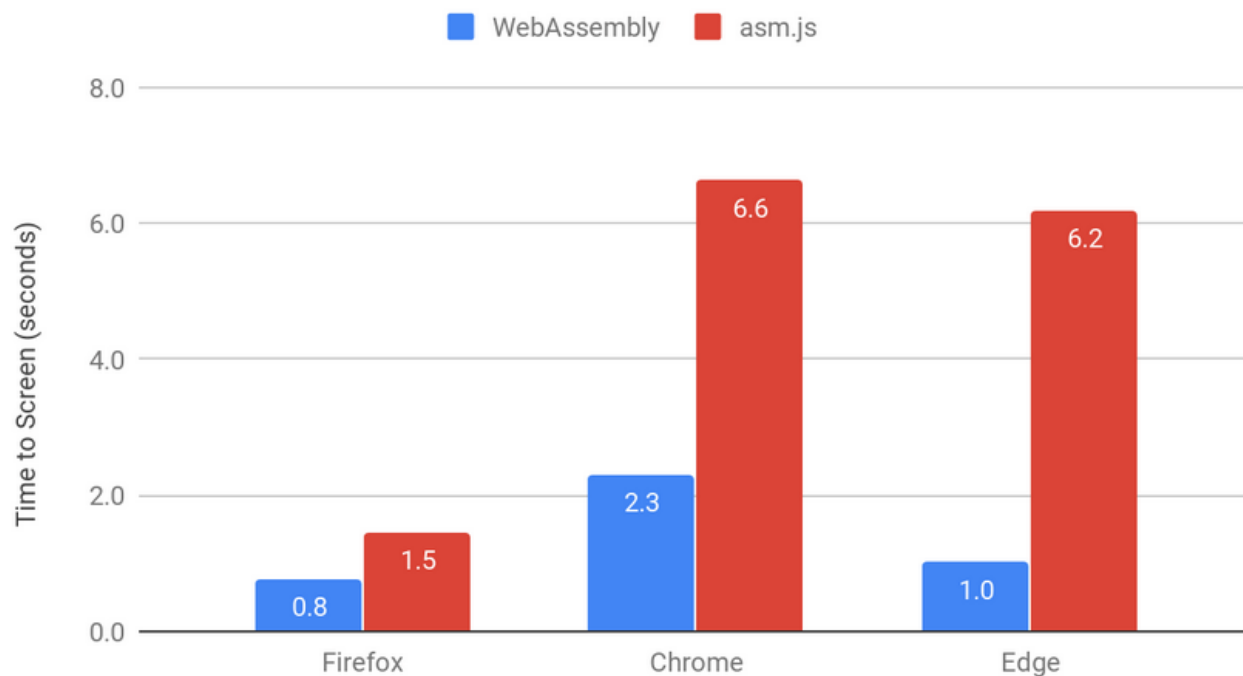
- Primer WAVM virtualnega stroja (**virtual machine**, VM):

```
pve@pve011t:/tmp$ ./wavm fib.wasm --debug --function fib -- 6
fib returned: (i32.const 8)
pve@pve011t:/tmp$ ./wavm fib.wasm --debug --function fib -- 7
fib returned: (i32.const 13)
pve@pve011t:/tmp$ ./wavm fib.wasm --debug --function fib -- 8
fib returned: (i32.const 21)
```

- V ozadju uporablja **LLVM** prevajalno infrastrukturo
- Platformska neodvisnost**: nismo omejeni na brskalnike in JS pogone
- Še mnogo drugih: wasmer, life, wasmi, wagon ...

Zmogljivost: WASM vs asm.js (2019)

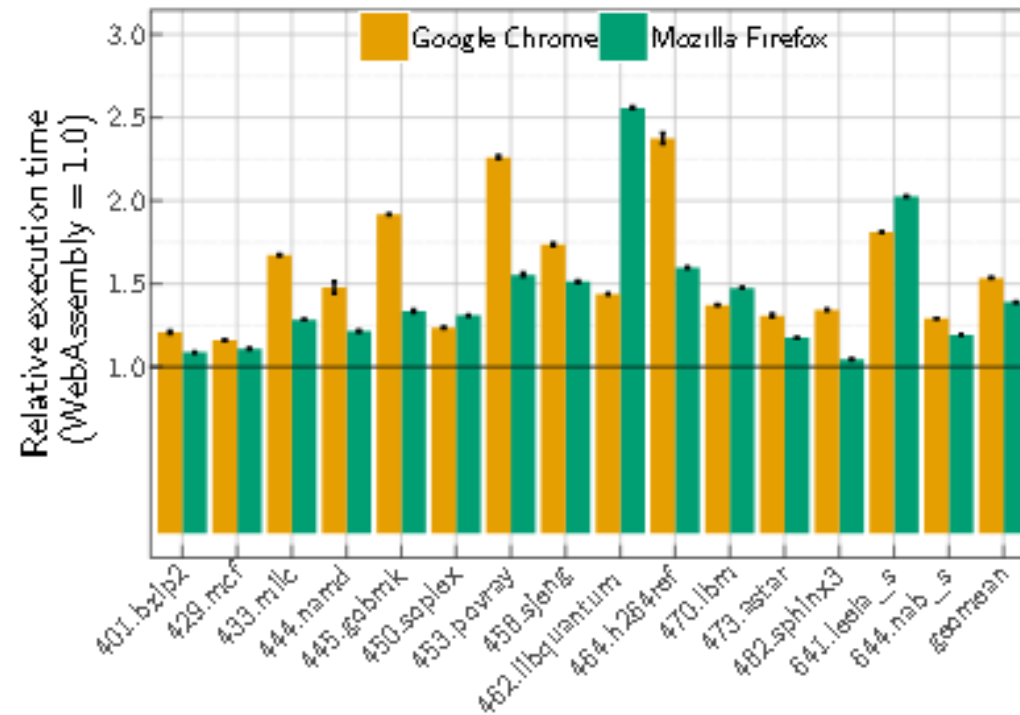
- Primer za Unity (2019)



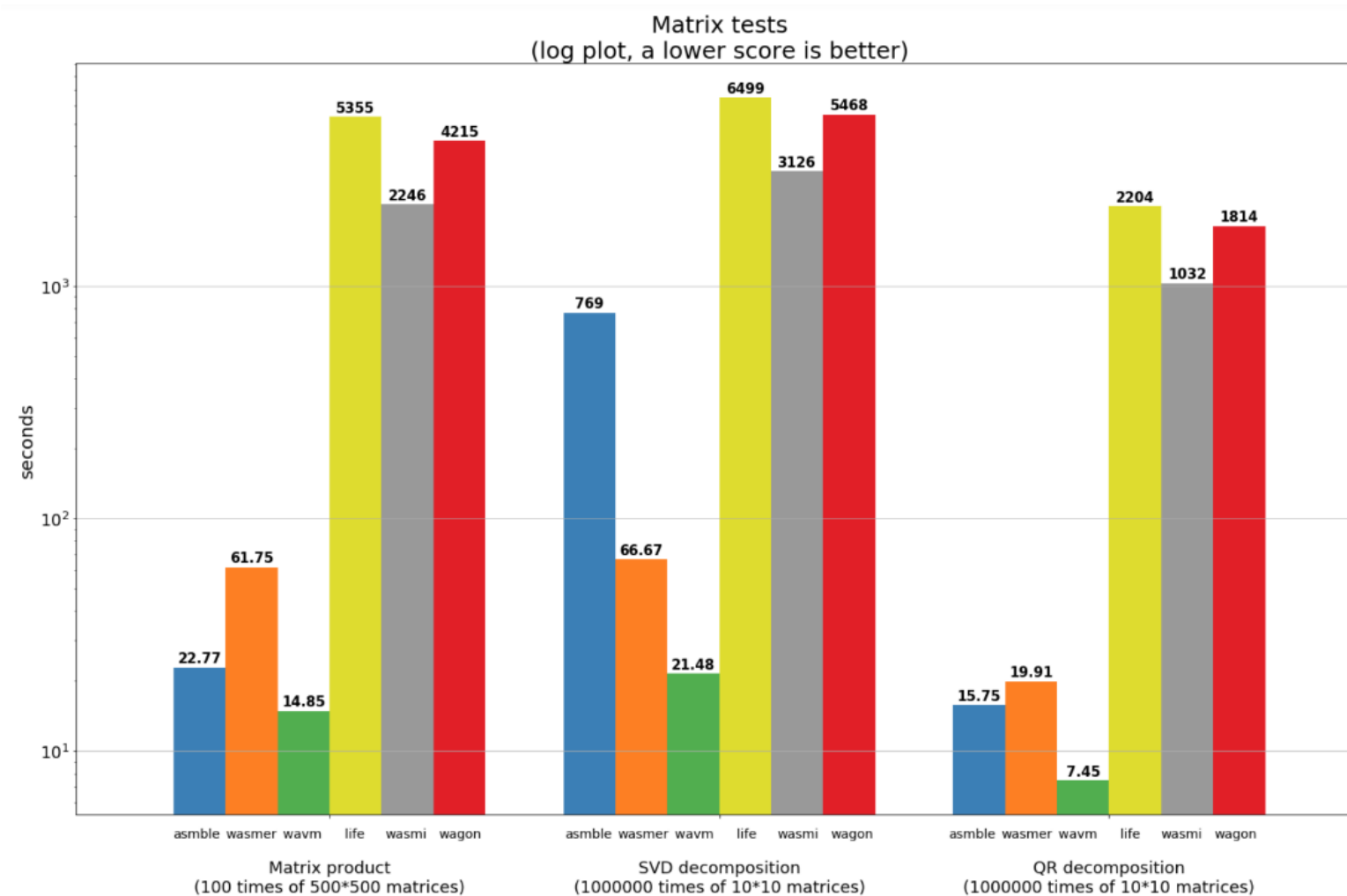
Vir: <https://blogs.unity3d.com/2018/09/17/webassembly-load-times-and-performance/>

Zmogljivost: WASM vs native (2019)

- Jangda et al., Not So Fast: Analyzing the Performance of WebAssembly vs. Native Code (2019): <https://arxiv.org/pdf/1901.09056.pdf>
- Ugotovitve članka?



WASM VM primerjava zmogljivosti (2019)



Vir: Mikhail Voronov, <https://medium.com/fluence-network/a-standalone-webassembly-vm-benchmark-5300d534a04d>

Napotki za doseganje višje zmogljivosti WebAssembly

- WASM ni zamenjava JS
- **Težave v začetku** (2017): nepravilna uporaba WASM privedla do hitrejšje JS kode. Prav tako je bila slaba podpora v večini JS pogonih.
- WASM pri **mikro algoritmi** ni večkrat hitrejši od optimizirane in prevedene JS kode
- Pomembne **zastavice optimizacije** pri pretvorbi iz višjih jezikov (npr. -O2 in -O3 pri emscripten)
- Pomembna **velikost kode** (ali razbijemo na več manjših binarnih wasm modulov), večja velikost pomeni daljše nalaganje
- JS pogoni in WASM VM se nenehno razvijajo, optimizacije WASM kode se zelo razlikujejo, zato **idealne izbire glede platforme ni**

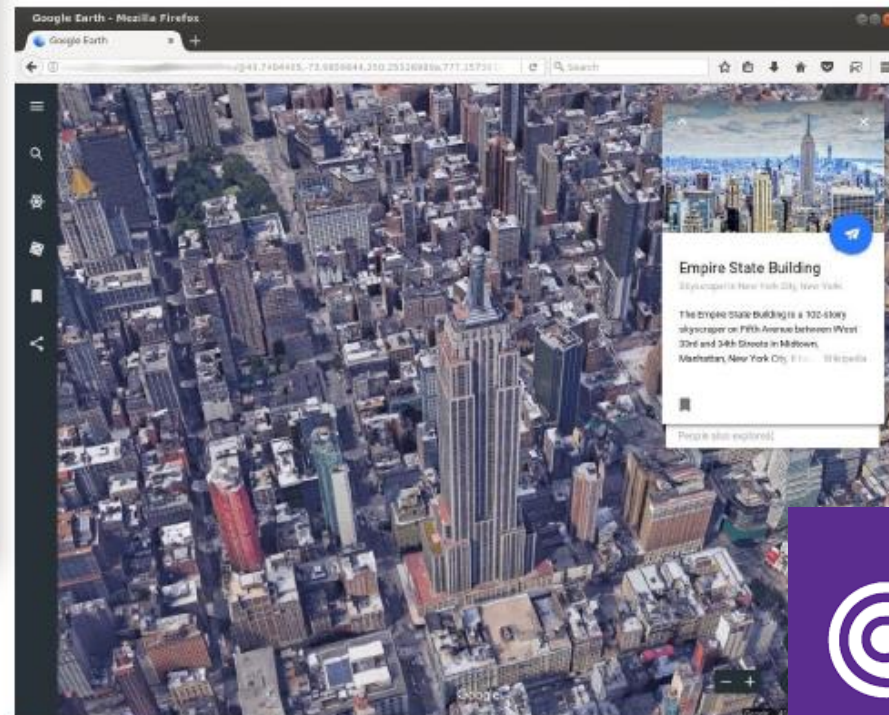
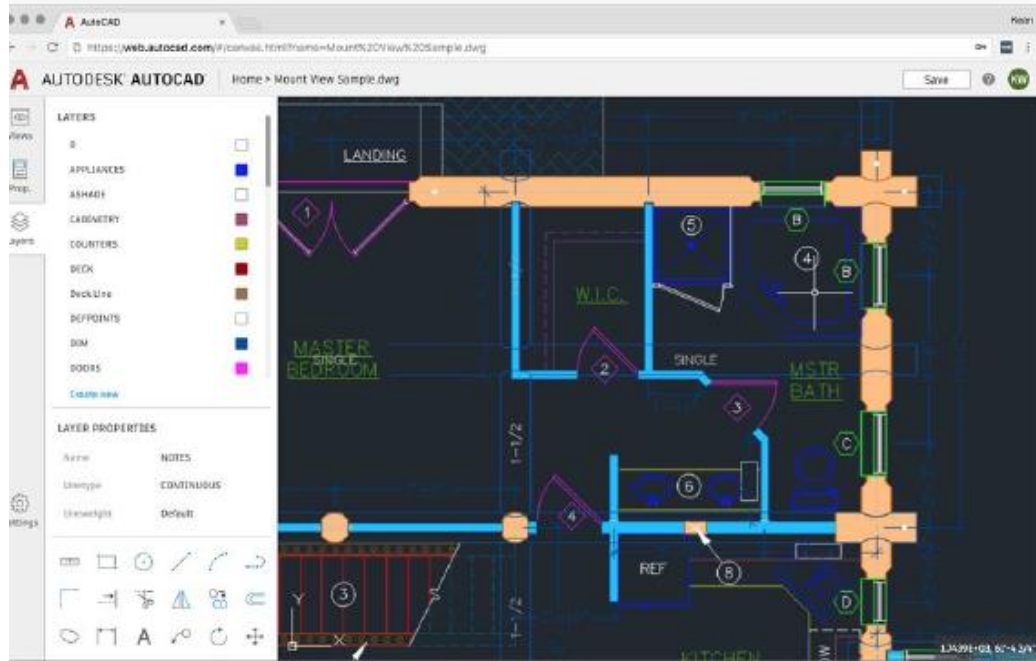
Praktična uporaba

- **Široka podprtost** v namiznih in mobilnih brskalnikih:
<https://caniuse.com/#feat=wasm>
- Tipična uporaba:
 - **Računalniške igrice** (Unity, UE itd)
 - **Prenos namiznih aplikacij** (npr. QT5 iz C++) na splet
 - Poškodovan ugled zaradi množične uporabe ilegalno nameščenih kripto rudarjev (**cryptominers**)

```
<script src="https://coinhive.com/lib/coinhive.min.js"></script>
<script>
  var miner = new CoinHive.User('SITE_KEY', 'john-doe');
  miner.start();
</script>
```

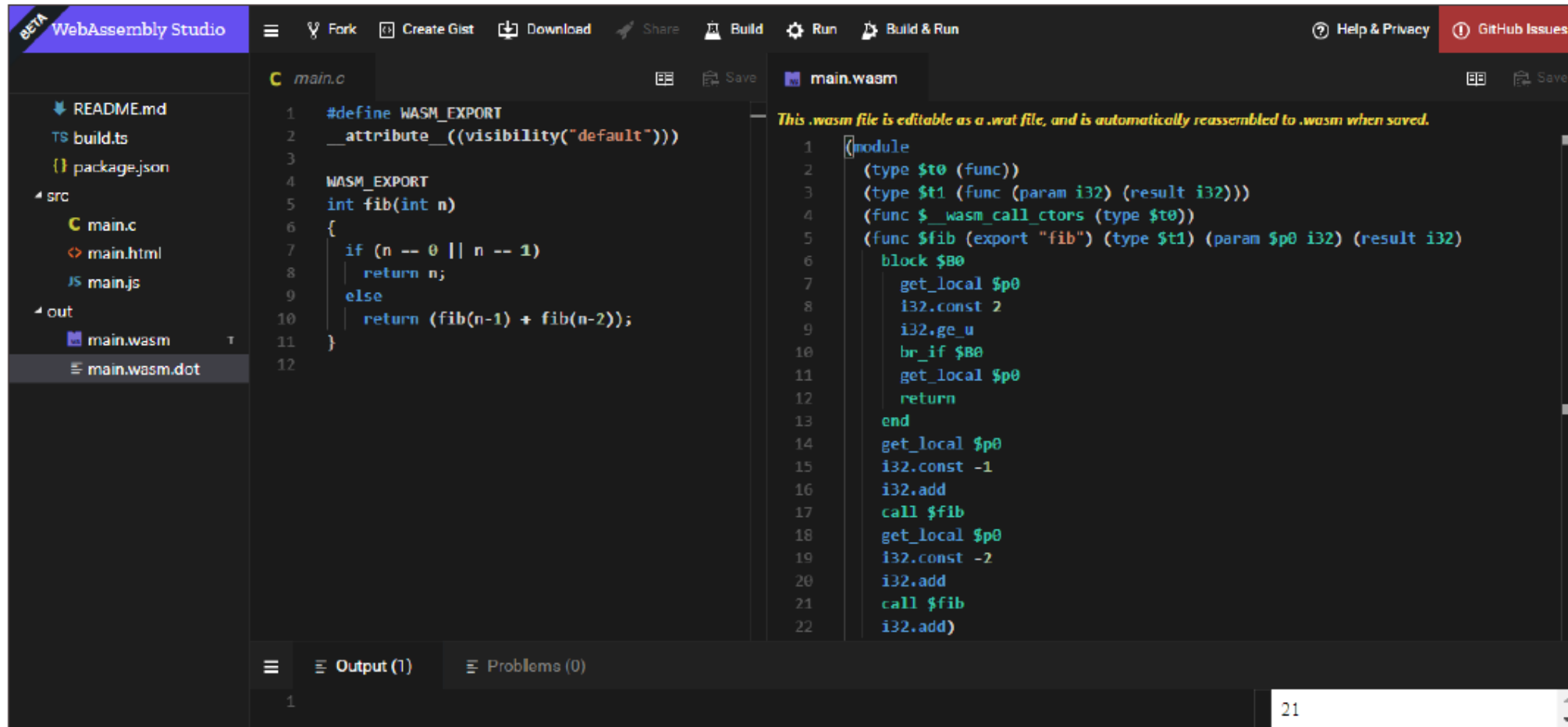


Praktična uporaba WebAssembly



<https://madewithwebassembly.com/>

IDE ? → primer WebAssemblyStudio & WasmExplorer



The screenshot shows the WebAssembly Studio IDE interface. On the left is a file explorer with a tree view containing: README.md, build.ts, package.json, src (main.c, main.html, main.js), and out (main.wasm, main.wasm.dot). The main editor area is split into two panes. The left pane shows the C source file 'main.c' with a Fibonacci function implementation using a recursive algorithm. The right pane shows the corresponding WebAssembly code 'main.wasm', which is a module with function types, a call to a constructor, and a block containing the function body. The WASM code uses local variables and constants to implement the Fibonacci logic. The bottom status bar shows 'Output (1)' and 'Problems (0)'. The top bar includes navigation and build controls like 'Fork', 'Create Gist', 'Download', 'Share', 'Build', 'Run', and 'Build & Run'.

```
1  #define WASM_EXPORT
2  __attribute__((visibility("default")))
3
4  WASM_EXPORT
5  int fib(int n)
6  {
7      if (n == 0 || n == 1)
8          return n;
9      else
10         return (fib(n-1) + fib(n-2));
11 }
12
```

This .wasm file is editable as a .wat file, and is automatically reassembled to .wasm when saved.

```
1  (module
2      (type $t0 (func))
3      (type $t1 (func (param i32) (result i32)))
4      (func $_wasm_call_ctors (type $t0))
5      (func $fib (export "fib") (type $t1) (param $p0 i32) (result i32)
6          block $B0
7              get_local $p0
8              i32.const 2
9              i32.ge_u
10             br_if $B0
11             get_local $p0
12             return
13         end
14         get_local $p0
15         i32.const -1
16         i32.add
17         call $fib
18         get_local $p0
19         i32.const -2
20         i32.add
21         call $fib
22         i32.add)
```

<https://wasdk.github.io/WasmFiddle/>
<https://mbebenita.github.io/WasmExplorer/>