Lecture 13

Rust and WebAssembly

WebAssembly Demo

What is wrong with JavaScript?

- The strength is a WEAKNESS:

Easy to learn + Executes in browser + Dynamic Typing

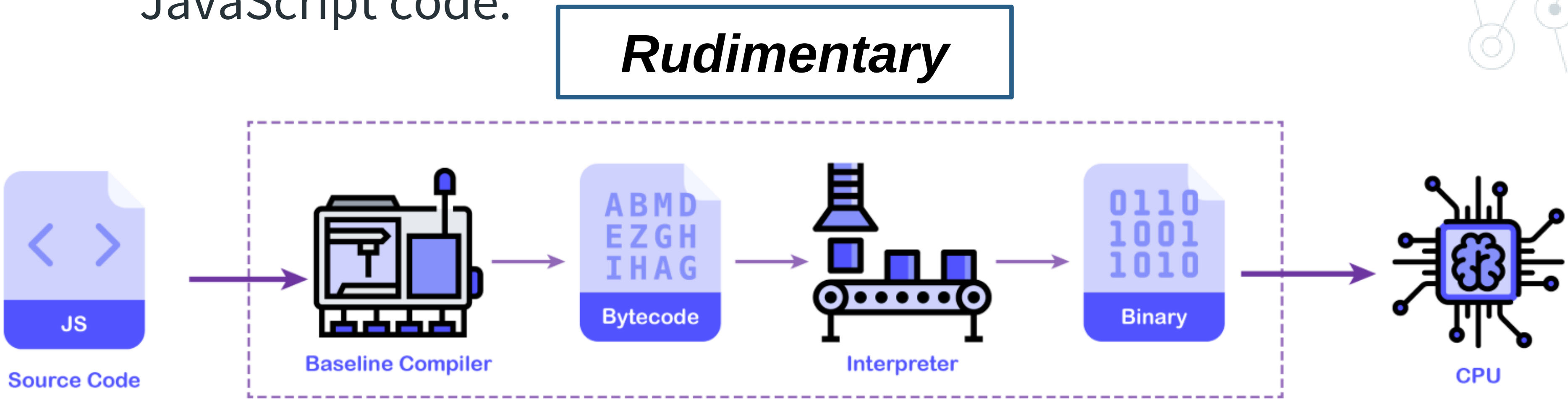
=

LOW PERFORMANCE

- 3D games? image/video editing?
- The cost of downloading, parsing, and compiling very large JavaScript applications can be prohibitive.
- Mobile and other resource-constrained platforms?

How does JavaScript work?

- Every browser provides a JavaScript engine that runs the JavaScript code.



JavaScript Engine (rudimentary)

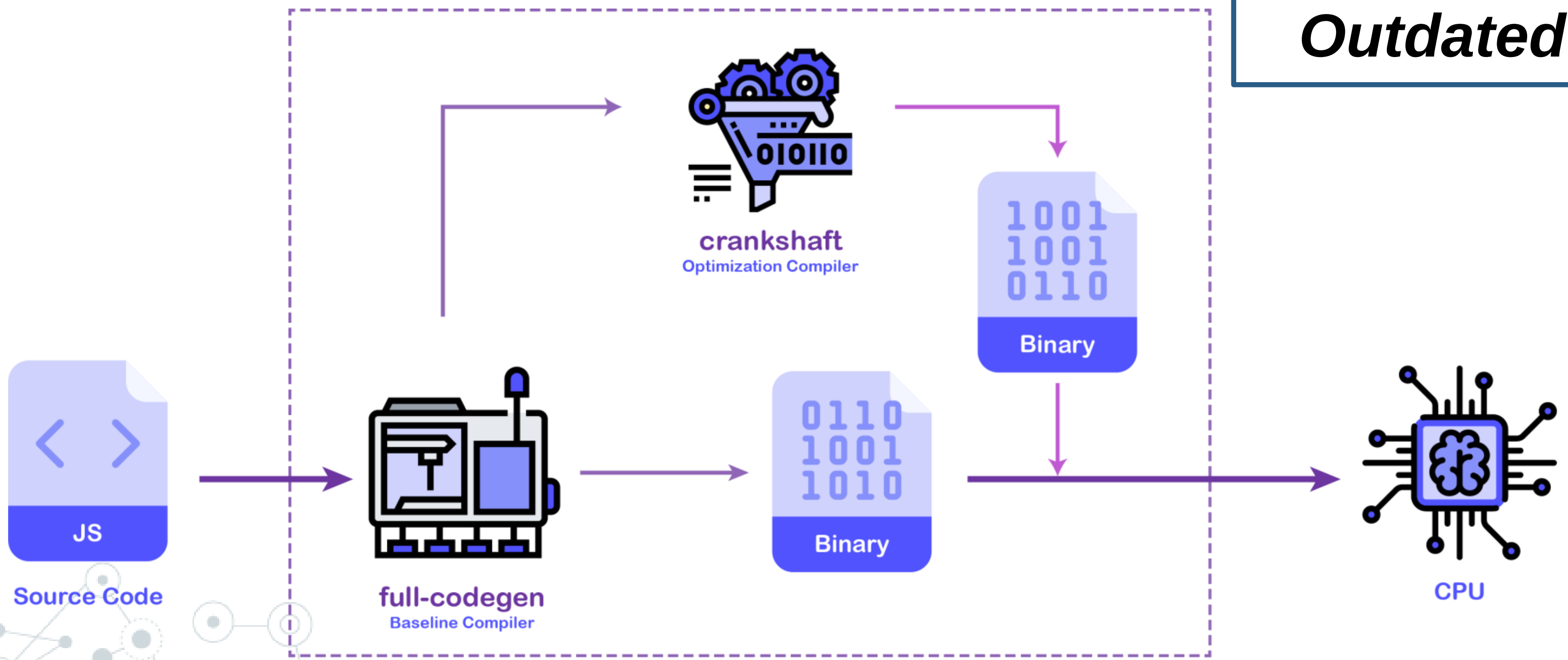
(**Netscape/SpiderMonkey** JavaScript engine)

- When it comes to a highly dynamic and interactive web application, the user experience is very poor with this model.

How does JavaScript work?

- This problem was faced by Google's Chrome browser while displaying Google Maps on the web.
- To increase the JavaScript performance on the web, they used the V8 JavaScript engine.

Outdated

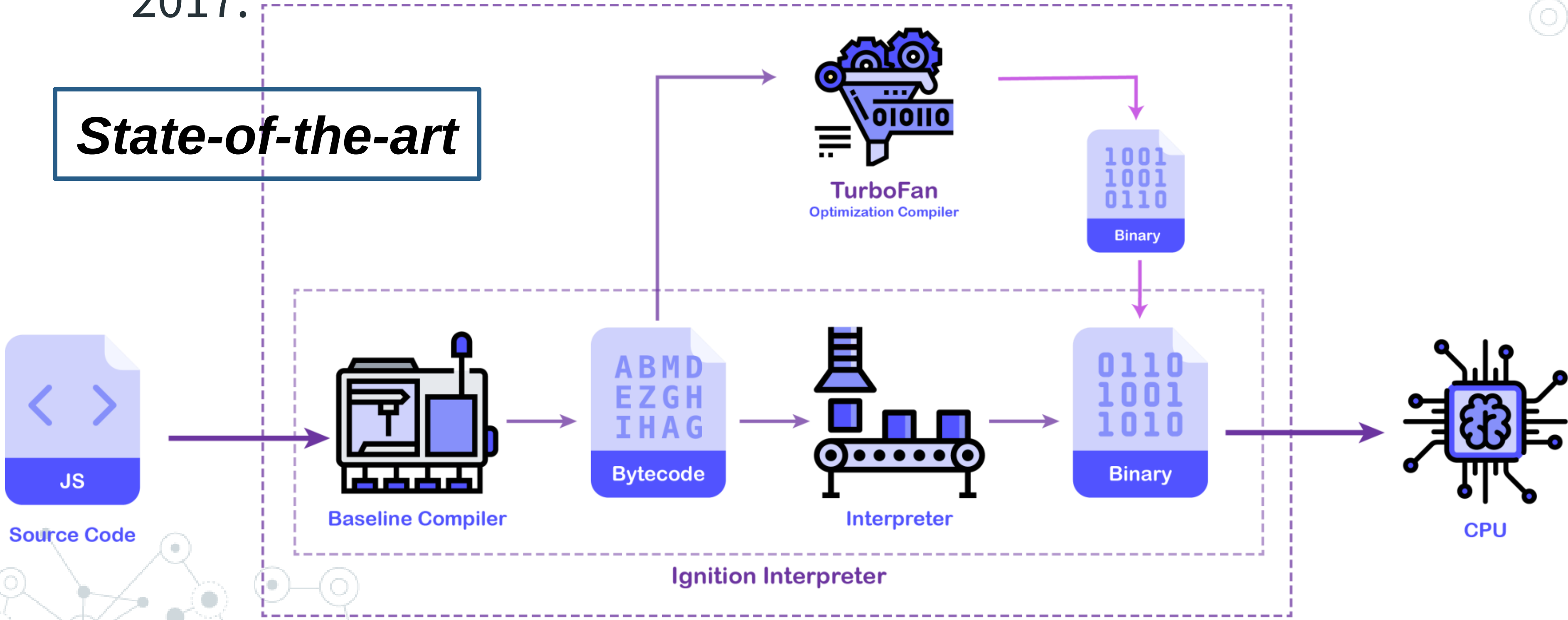


V8 JavaScript Engine (2010)

How JavaScript is optimized?

- The V8 team created a new version of the V8 engine from the ground up.
- This new version of the JavaScript engine was released in 2017.

State-of-the-art



V8 JavaScript Engine (2017)

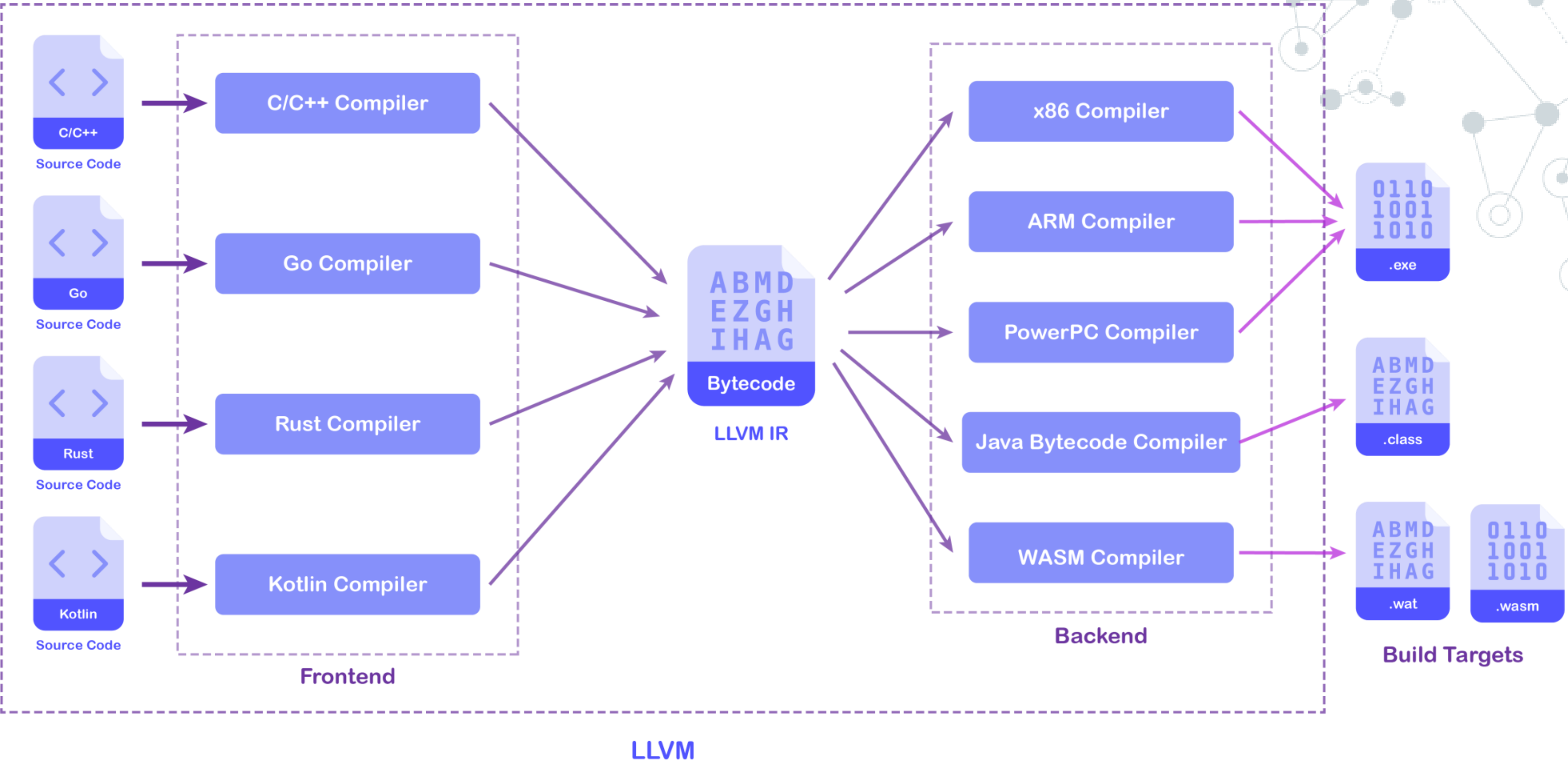
WebAssembly: Game Changer!

WebAssembly: Game Changer!

- WebAssembly, a bytecode standard for web browsers.
- Announced in 17 June 2015.
- Developed by WebAssembly Working Group.
- Build target
- Binary format
- Supports JS.

WebAssembly: Game Changer!

- A language for the web.
- Compiled from other languages.
- Offers maximized, reliable performance.
- Not a replacement for JS.
- It is meant to augment the things that JS was never designed to do.


C/C++
Source Code
Go
Source Code
Rust
Source Code
Kotlin
Source Code
C/C++ Compiler
Go Compiler
Rust Compiler
Kotlin Compiler
Frontend
ABMD
EZGH
IHAG
Bytecode
LLVM IR
x86 Compiler
ARM Compiler
PowerPC Compiler
Java Bytecode Compiler
WASM Compiler
Backend
0110
1001
1010
.exe
ABMD
EZGH
IHAG
.class
ABMD
EZGH
IHAG
.wat
0110
1001
1010
.wasm
Build Targets
LLVM

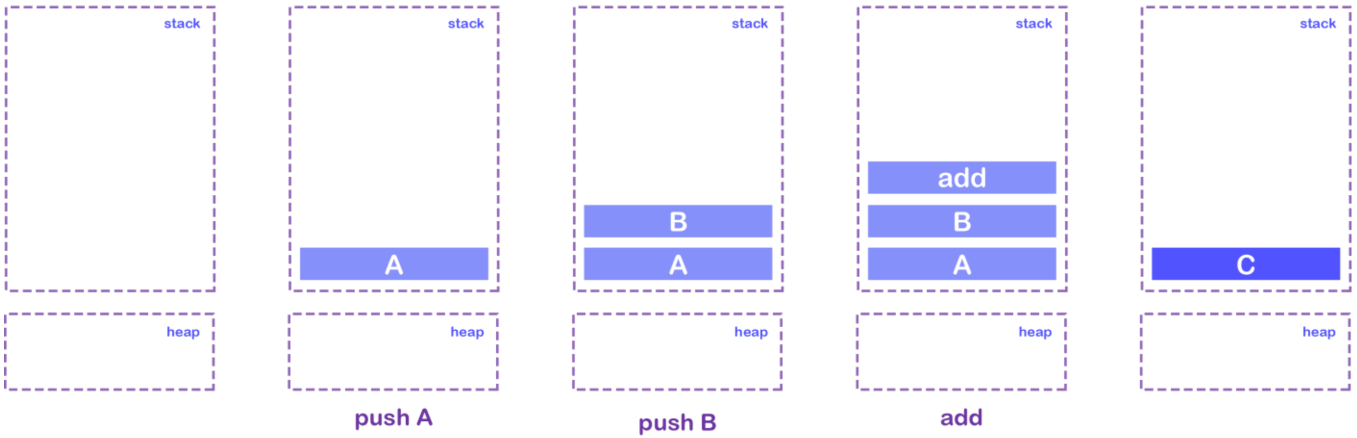
Main Advantages

1. **Performance:** WebAssembly offers strong type guarantees, it gives more consistent and reliable performance than JS.
2. **Portability:** because you can compile from other languages, you can bring open source libraries built in languages like C++.
3. **Flexibility**: the ability to write in other languages.
   - To date, JS has been the only fully supported option.
   - Now with WebAssembly, you get more choice.

WebAssembly is a Stack Machine

- Stack machine is a virtual machine (like a processor) that takes one instruction at a time and pushes it on the stack.
- When an operation is needed to be performed, it will pop values from the stack and compute the result.
- Example: compute the sum of two integers:

```
push A
push B
add
```



WebAssembly is a Stack Machine

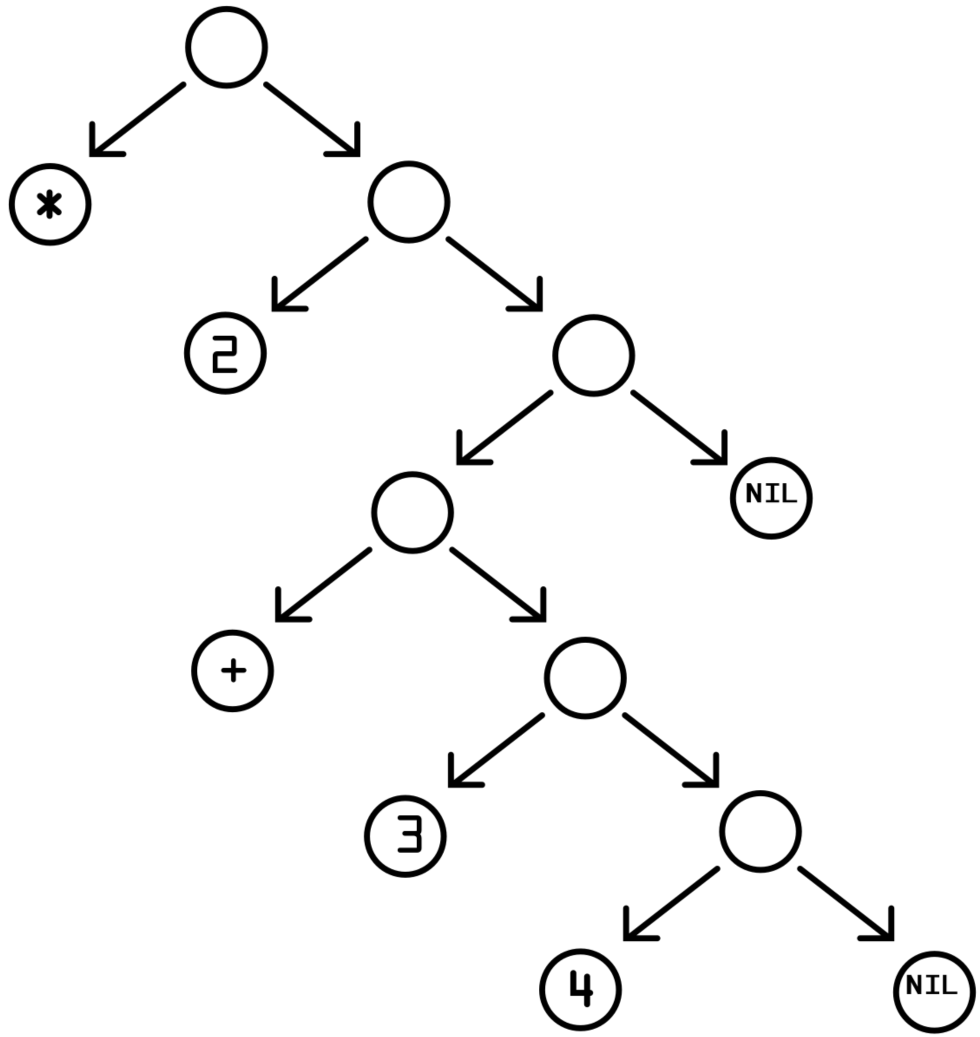
- Here’s an example of a WebAssembly function that computes the quantity **F(x)=2x²+1**.

```
(func $f (param $x i32) (result i32)
                        ;; stack: []
  (get_local $x)        ;; stack: [x]
  (get_local $x)        ;; stack: [x, x]
  (i32.mul)             ;; stack: [x*x]

  (i32.const 2)         ;; stack: [2, x*x]
  (i32.mul)             ;; stack: [2*x*x]

  (i32.const 1)         ;; stack: [1, 2*x*x]
  (i32.add))            ;; stack: [2*x*x+1]
```

WebAssembly is a Stack Machine

- First, the syntax: this is the WebAssembly text format, which is derived from S-expressions.
- The same format commonly used for the Lisp programming.
- An S-expression is either
  - an “atom” (e.g. get_local, 2, or $x),
  - or a list of S-expressions surrounded by parentheses (e.g. (get_local $x), (param $x i32)).

S-expression (* 2 (+ 3 4))



WebAssembly is a Stack Machine

- **MOST** instructions in WebAssembly modify the value stack in some way.

```
(i32.const 1)
(i32.const 2)
(i32.sub)
```

F(x) in a register-based notation

```
(i3f:
        movl   %edi, -4(%rbp) ; *(rbp-4) = x
        shll   $1, %edi       ; edi = edi << 1 (= edi * 2)
        imull  -4(%rbp), %edi ; edi = edi * *(rbp-4)
        addl   $1, %edi       ; edi = edi + 1
        movl   %edi, %eax     ; eax = edi
        retq                    ; return eax2.const 1)
(i32.const 2)
(i32.sub)
```

- Of course, x86 still has a stack (e.g. -4(%rbp)), but it uses that in tandem with registers.

WebAssembly is Build Target

- Binary format - WASM

```
get_local 0
i64.eqz
if (result i64)
    i64.const 1
else
    get_local 0
    get_local 0
    i64.const 1
    i64.sub
    call 0
    i64.mul
end
```

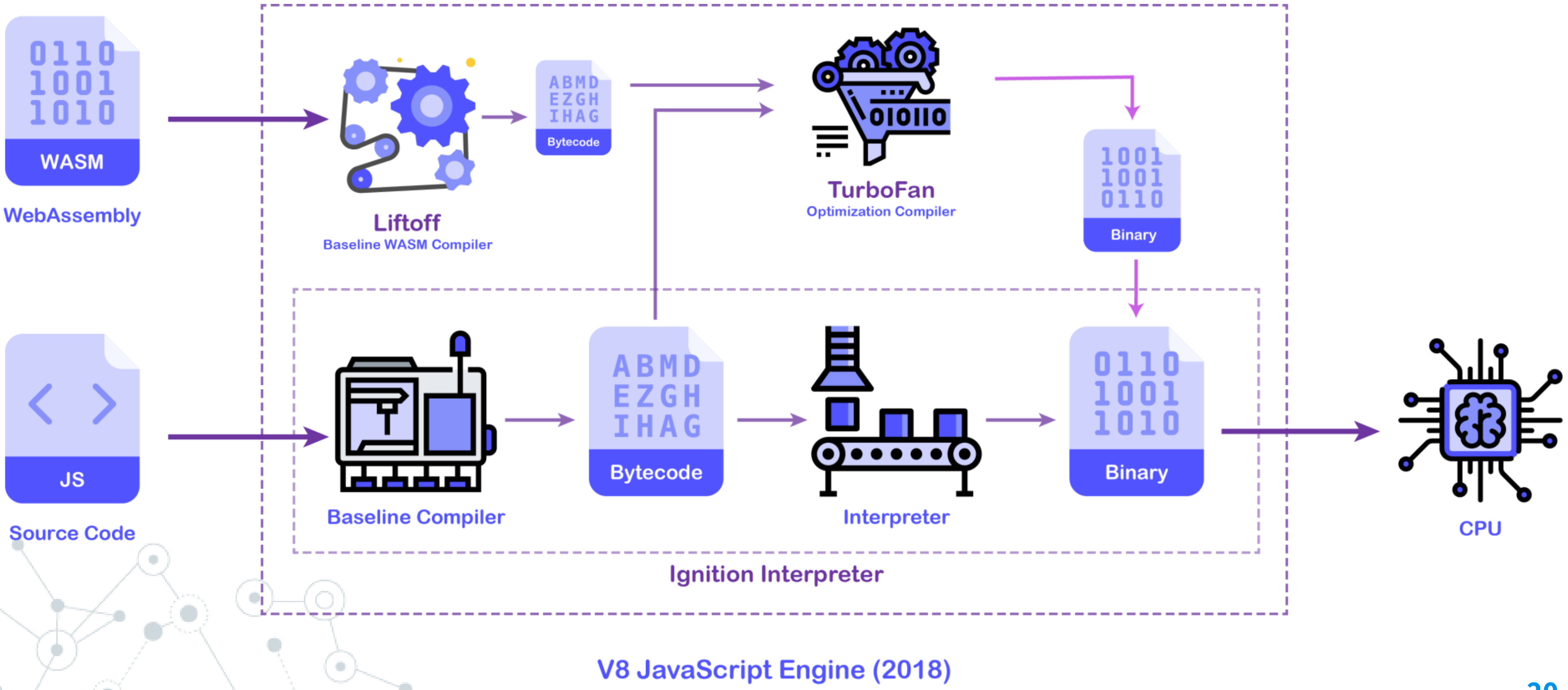
```
20 00
50
04 7E
42 01
05
20 00
20 00
42 01
7D
10 00
7E
0B
```

How WebAssembly works?

- Every browser has a JavaScript engine which runs JavaScript.
- How it can run WebAssembly binary instructions?
- Browsers have introduced a new baseline compiler to compile WebAssembly to a bytecode that JavaScript interpreter can understand.

How WebAssembly works?

- V8 integrated Liftoff, their WebAssembly baseline compiler whose job is to compile WebAssembly into an unoptimized bytecode as quick as possible.



Can we use WebAssembly?

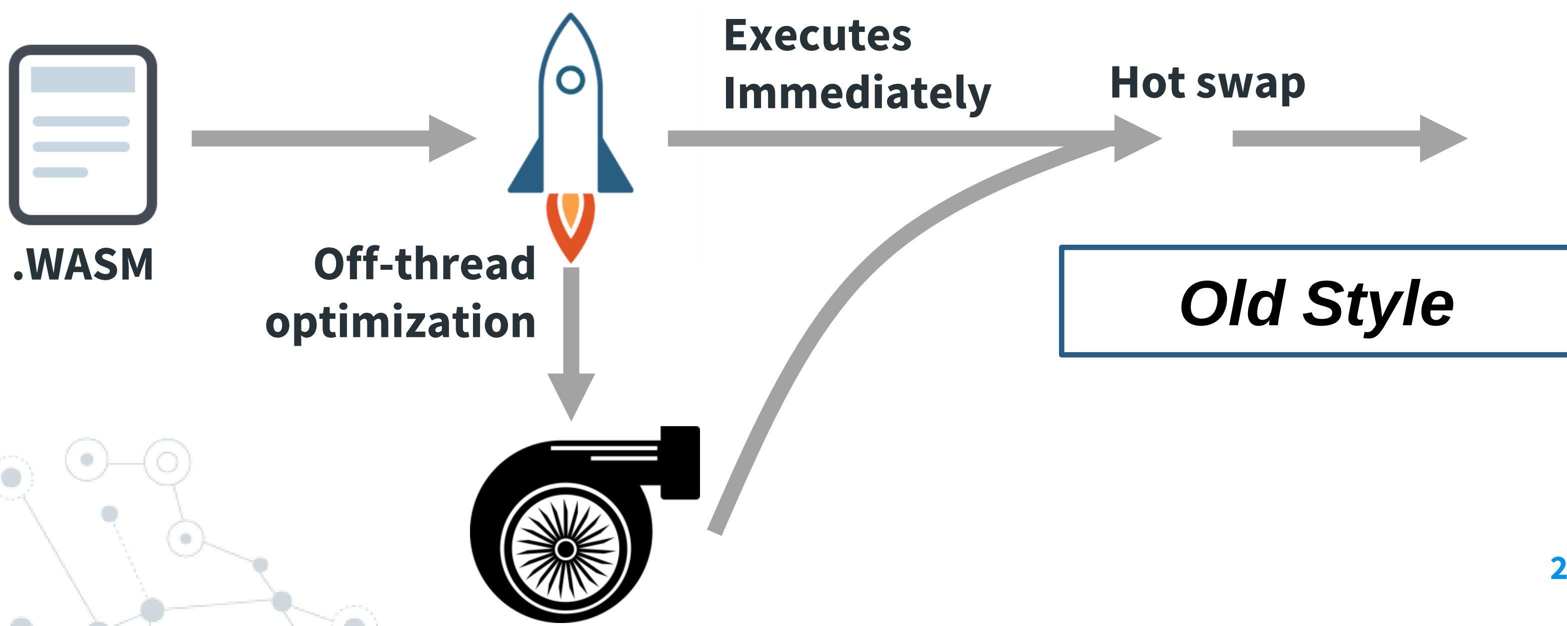
- Shipped in all major browsers.
- The first new language to ship in every major browser since JS.

Why WebAssembly is Fast?

Improvements to WASM

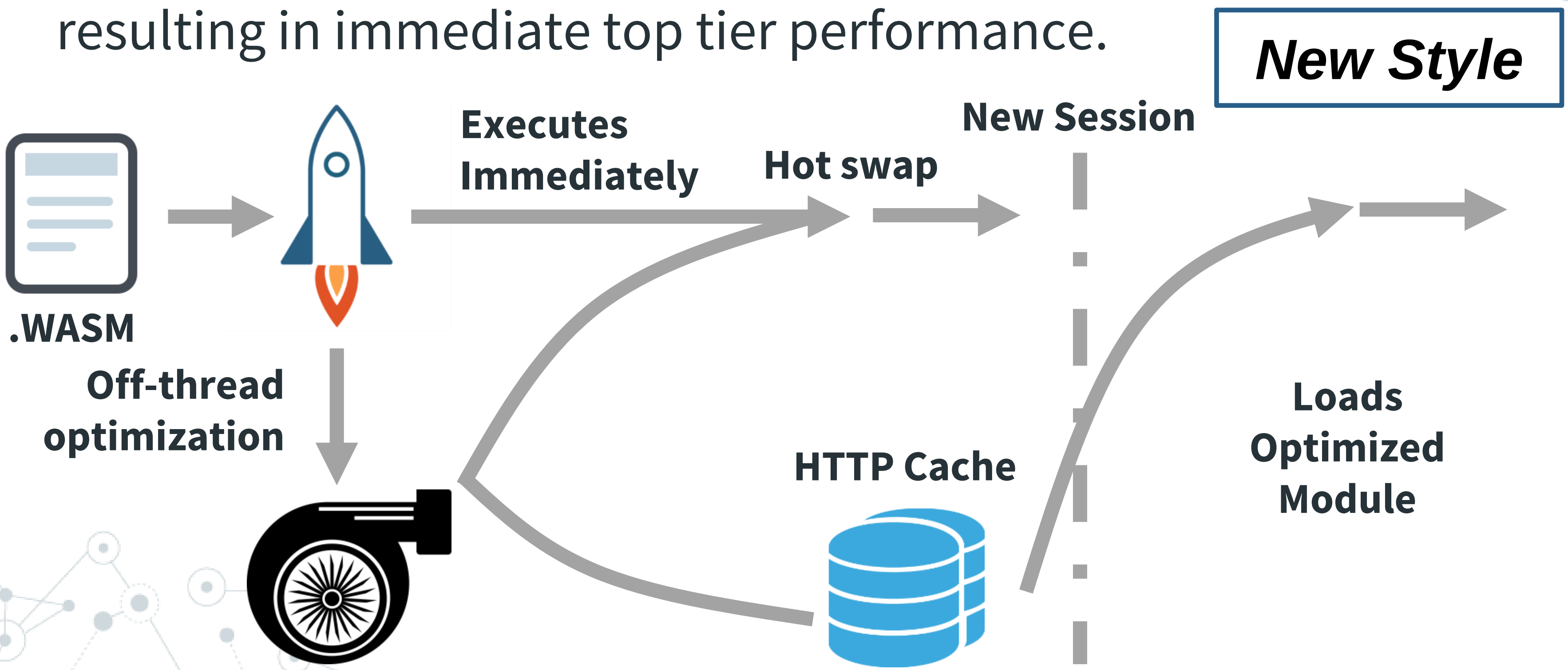
1- Implicit Caching (faster startup time):

- When a site loads a WebAssembly module, it first goes into the Liftoff compiler to execute it.
- Then, the code is further optimized off the main thread though the turbo fan optimizing compiler.
- Then, the results are hot swapped in when ready.



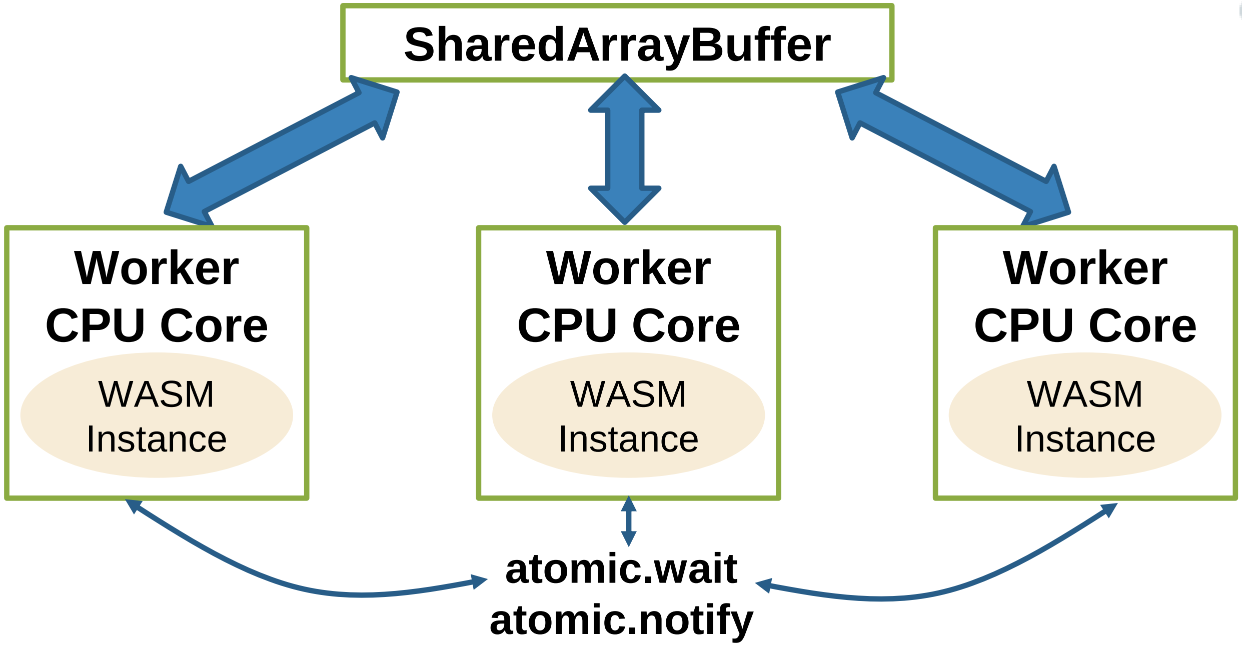
Improvements to WASM

- Now, with implicit caching, we cache that optimized WebAssembly module directly in the HTTP cache.
- Then, after the user leaves the page and comes back, we load that optimized module directly from the cache, resulting in immediate top tier performance.



Improvements to WASM

2- New Features: *WebAssembly Threads*

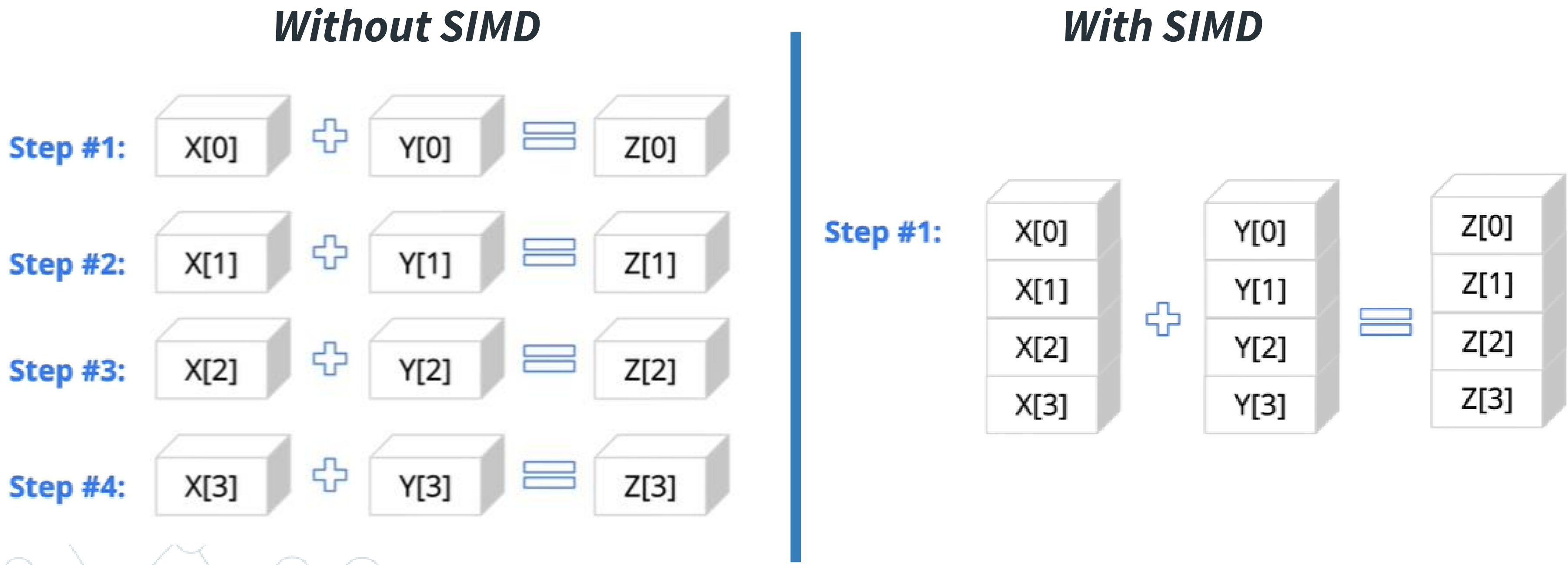


- WebWorkers: allows WebAssembly to run on different CPU cores.
- SharedArrayBuffer: allows WebAssembly to operate on the same piece of memory.
- Atomic Operations: synchronizing WebAssembly so that things happen in the right order.

Improvements to WASM

2- New Features: *SIMD (Single Instruction Multiple Data)*

```
for (i = 0; i<4; i++)
   z[i] = x[i] + y[i]
```



Improvements to WASM

- The CPU is able to vectorize these elements and just jake a single CPU operation to add them together.
- One example of SIMD is the hand tracking system: Mediapipe.page.link/web
- Without SIMD: we are getting about three frames per second
- With SIMD, we get a much smoother 15 frames per second.

Please start the slide show to run the demo

Rust & WebAssembly Demo

- https://m1el.github.io/wasm-asteroids/demo/demo.html

Rust, JavaScipt and WebAssemby

```
use wasm_bindgen::prelude::*;
#[wasm_bindgen]
extern {
	pub fn alert(s: &str);
}
#[wasm_bindgen]
pub fn greet(name: &str) {
alert(&format!("Hello, {}!", name));
}
```

```
use wasm_bindgen::prelude::*;
```

- The first line contains a use command, which imports code from a library into your code.

.

- wasm-bindgen provideS a bridge between the types of JavaScript and Rust. It allows JavaScript to call a Rust API with a string, ….

Calling external functions in JavaScript from Rust

```
#[wasm_bindgen]
extern {
pub fn alert(s: &str);
}
```

- extern, which tells Rust that we want to call some externally defined functions.
- The attribute says "wasm-bindgen knows how to find these functions".
- The third line is a function signature, written in Rust. It says "the alert function takes one argument, a string named s."
- The alert function provided by JavaScript. We call this function in the next section.

Producing Rust functions that JavaScript can call

```
#[wasm_bindgen]
pub fn greet(name: &str)
{
    alert(&format!("Hello,
{}!", name));
}
```

Once again, we see the #[wasm_bindgen] attribute. In this case, it's not modifying an extern block, but a fn; this means that we want this Rust function to be able to be called by JavaScript. It's the opposite of extern: these aren't the functions we need, but rather the functions we're giving out to the world.

The code in detail ….

- This function is named greet, and takes one argument, a string (written &str), name. It then calls the alert function we asked for in the extern block above. It passes a call to the format! macro, which lets us concatenate strings.

- The format! macro takes two arguments in this case, a format string, and a variable to put in it. The format string is the "Hello, {}!" bit. It contains {}s, where variables will be interpolated. The variable we're passing is name, the argument to the function, so if we call greet("Steve") we should see "Hello, Steve!".

- This is passed to alert(), so when we call this function, we will see an alert box with "Hello, Steve!" in it.