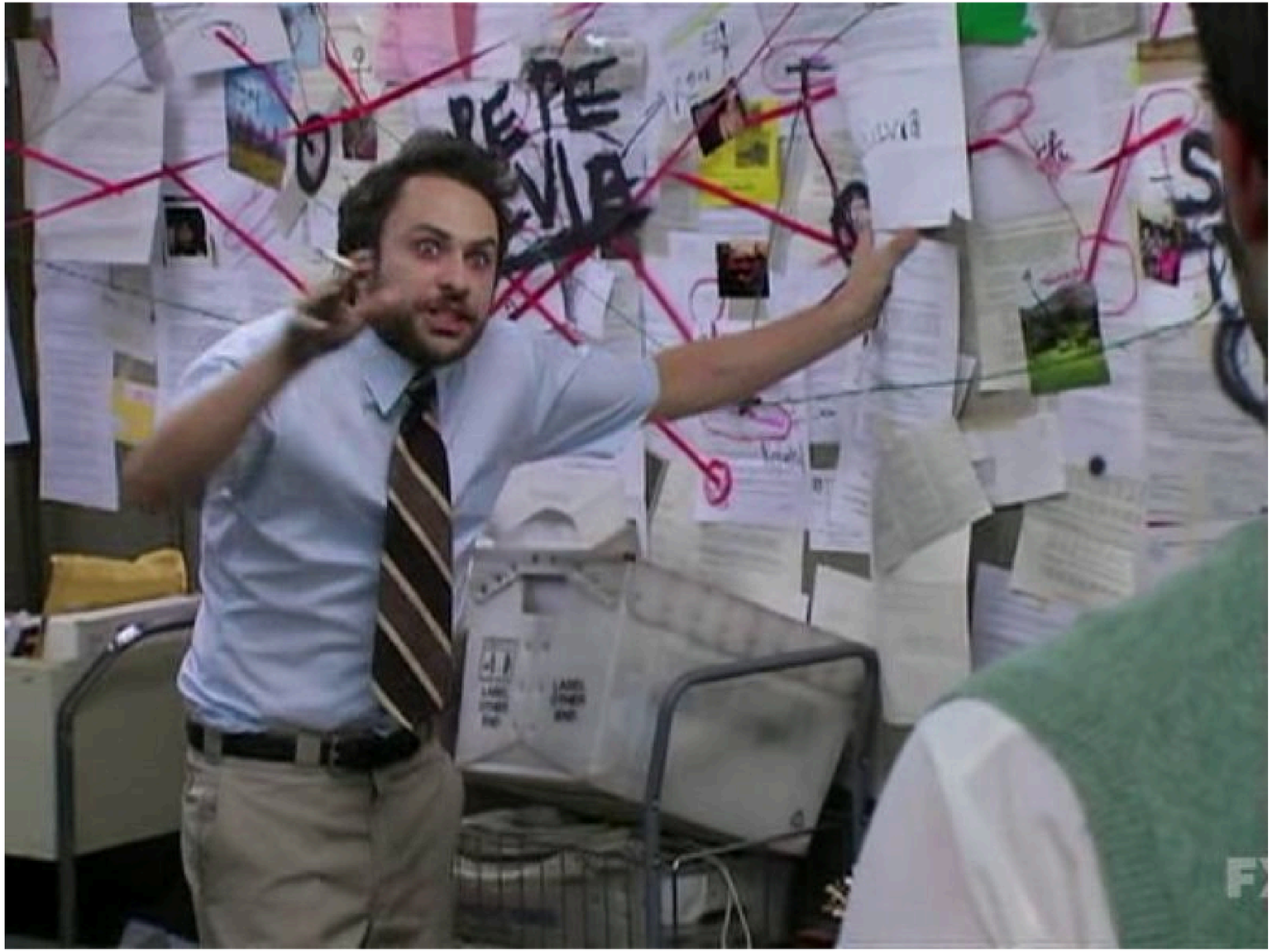
Anatomy of a Wasm Runtime

$whoami

- Hi! I’m Siddharth(siddharthtewari.me)
- Currently a 4th year student at PESU-EC campus
- Interested in backend engineering and all things systems
- Love talking about all things Wasm and distributed systems
- Aspiring audiophile
- Reach out to me on twitter/x: @sidT_008



A small aside:

- Wasm has its own semantics
- For example traps in Wasm do not mean the same thing as in say x86 or RISC-V.

So, what is WebAssembly?

- Stack based Virtual Machine?
- Binary format?
- Assembly-like language?
- A specification?
- All of the above?

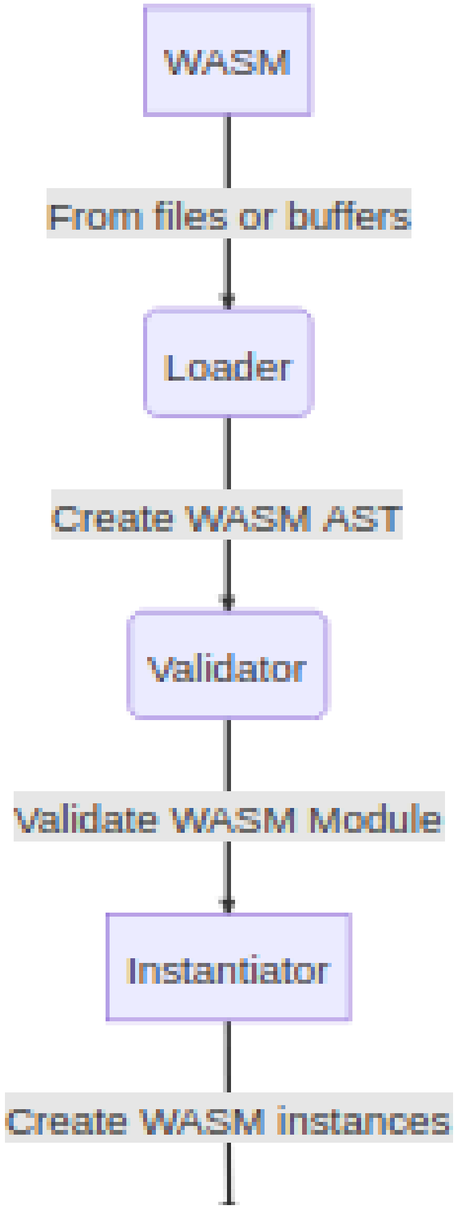

Friendship ended with JAVASCRIPT
Now
WEBASSEMBLY
is my
best friend
WA
JS
JS

What is WasmEdge?

- WasmEdge is a lightweight, high-performance, and extensible WebAssembly runtime. It is the fastest Wasm VM today.
- WasmEdge is a sandbox project hosted by the CNCF.
- Use cases include modern web application architectures, microservices on the edge cloud, serverless SaaS APIs, embedded functions, smart contracts, and much more


WASM
From files or buffers
Loader
Create WASM AST
Validator
Validate WASM Module
Instantiator
Create WASM instances
AOT section found?
Yes
No
AOT Engine
Interpreter Engine
Execute WASM
Execute WASM
WasmEdge Engine
Host Function Call / Access Runtime Data
WasmEdge Runtime
Call Host Functions
Access Runtime Data
Host Functions
Runtime Data Manager
System Call
AI-related Function Call
Crypto-related Function Call
Socket-related Function Call
Access Memory
Access Stack
Access Cross Module
WASM System Interface, WASI
WASI-NN
WASI-Crypto
WasmEdge-WASI-Socket
Memory Manager
Stack Manager
Registered Module/Function Manager

Lets have a closer look now!


WASM
From files or buffers
Loader
Create WASM AST
Validator
Validate WASM Module
Instantiator
Create WASM instances

Encoding into the binary format:

- Reference: https://webassembly.github.io/spec/core/syntax/index.html
- This is essentially the step where your code is encoded into the Wasm Binary Format
- The end product is a Wasm module. These are the fundamental unit of deployment, loading, and compilation.

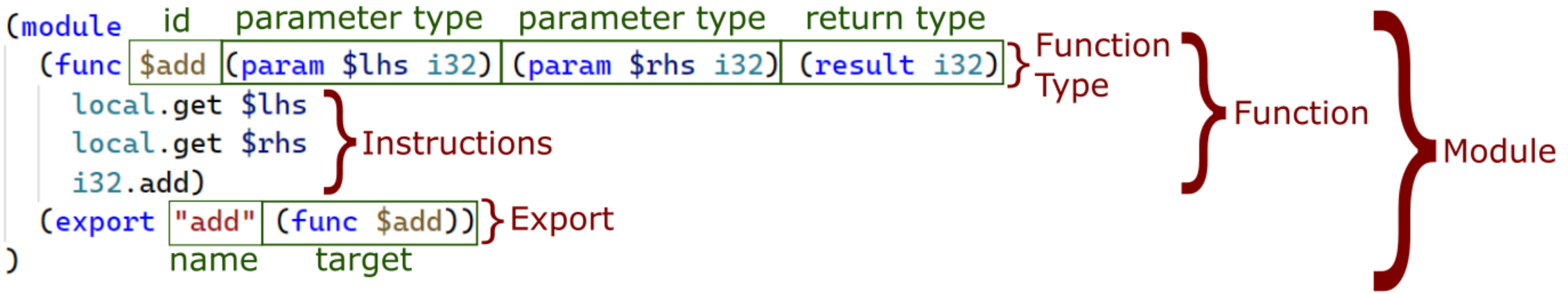
- Essentially an AST representation is created initially and then serialised and validated.
- If you’d like to see what a Wasm Module’s AST looks like you can convert any Wasm binary(.wasm file) into a .wat file(WebAssembly Text format)
- .wat files describe the AST in the form of S-expressions.

A Closer look:

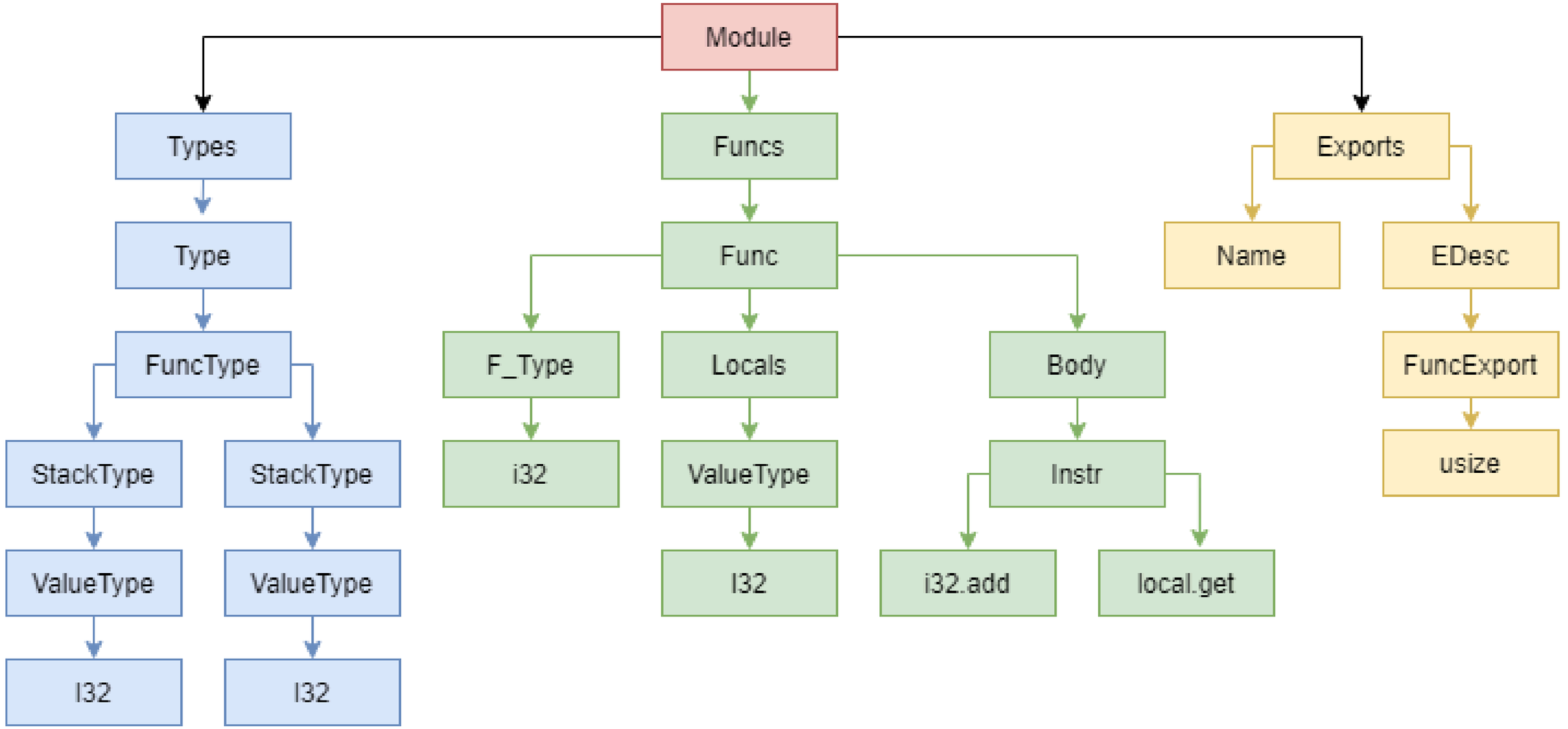
- The AST can have multiple types of nodes. Description nodes, instruction nodes, module nodes, section nodes and more.
- Module node embeds pretty much everything.
- There’s various sections like the memory section, table section, data section, global sections etc.
- The encoding of a module starts with a 4-byte magic number and a version field.

A Closer look:

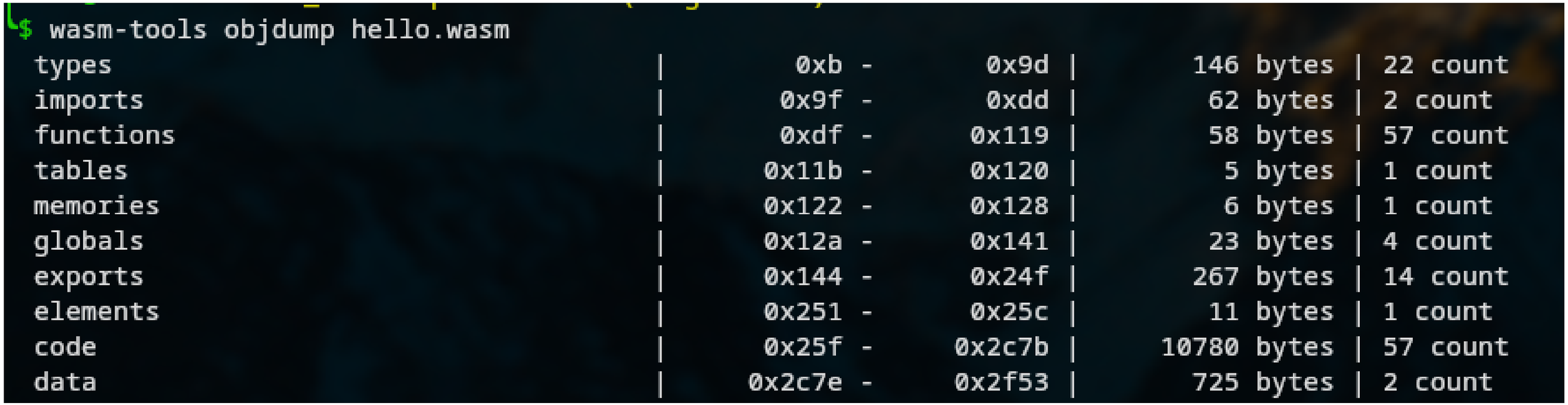
- The binary encoding of modules is organized into sections.
- Most sections correspond to one component of a module record
- The exception here is that Function definitions are split into two sections, separating their type declarations in the function section from their bodies in the code section.


(module
(func $add (param $lhs i32) (param $rhs i32) (result i32)
local.get $lhs
local.get $rhs
i32.add)
(export "add" (func $add))
)
id
parameter type
parameter type
return type
Function Type
Instructions
Export
name
target
Function
Module

[Source]


Module
Types
Type
FuncType
StackType
StackType
ValueType
ValueType
I32
I32
Funcs
Func
F_Type
i32
Locals
ValueType
I32
Body
Instr
i32.add
local.get
Exports
Name
EDesc
FuncExport
usize

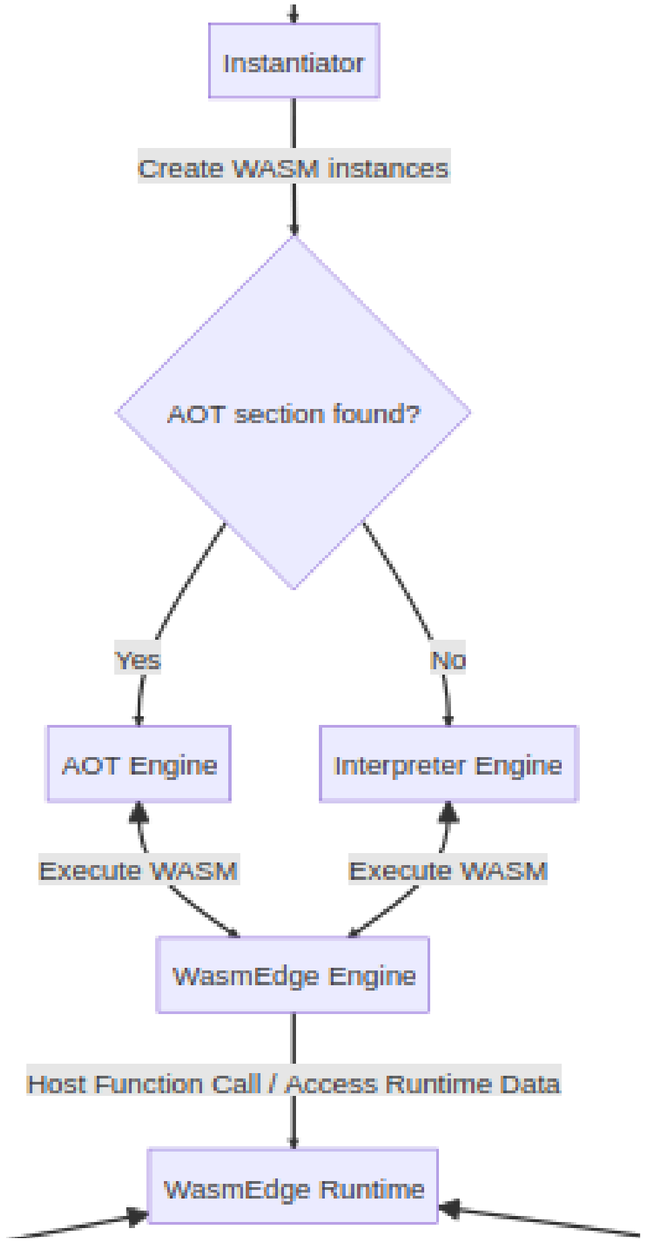
[Source]


$ wasm-tools objdump hello.wasm
types | 0xb - 0x9d | 146 bytes | 22 count
imports | 0x9f - 0xdd | 62 bytes | 2 count
functions | 0xdf - 0x119 | 58 bytes | 57 count
tables | 0x11b - 0x120 | 5 bytes | 1 count
memories | 0x122 - 0x128 | 6 bytes | 1 count
globals | 0x12a - 0x141 | 23 bytes | 4 count
exports | 0x144 - 0x24f | 267 bytes | 14 count
elements | 0x251 - 0x25c | 11 bytes | 1 count
code | 0x25f - 0x2c7b | 10780 bytes | 57 count
data | 0x2c7e - 0x2f53 | 725 bytes | 2 count

Id	Section
0	custom section
1	type section
2	import section
3	function section
4	table section
5	memory section
6	global section
7	export section
8	start section
9	element section
10	code section
11	data section
12	data count section

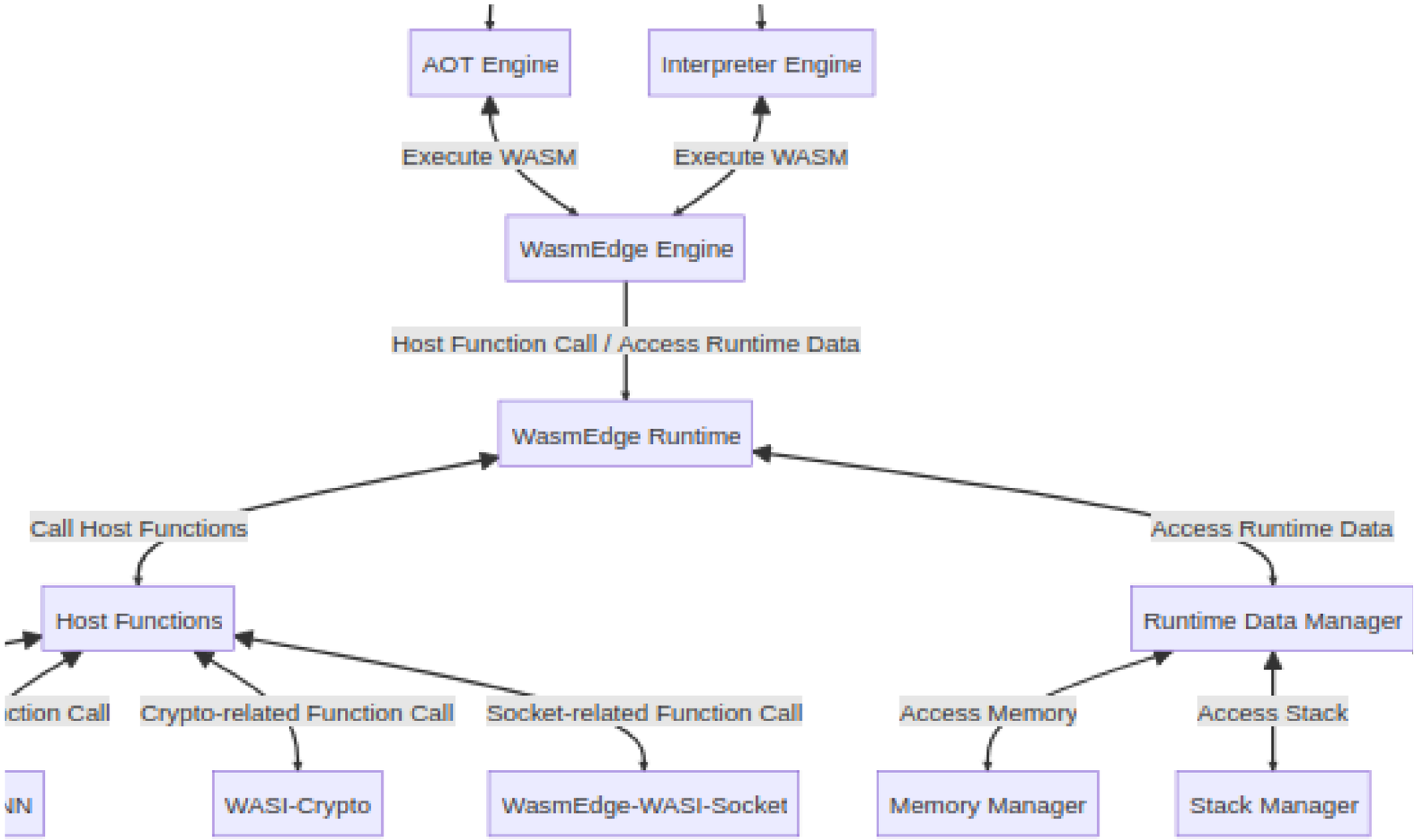
Each section consists of:

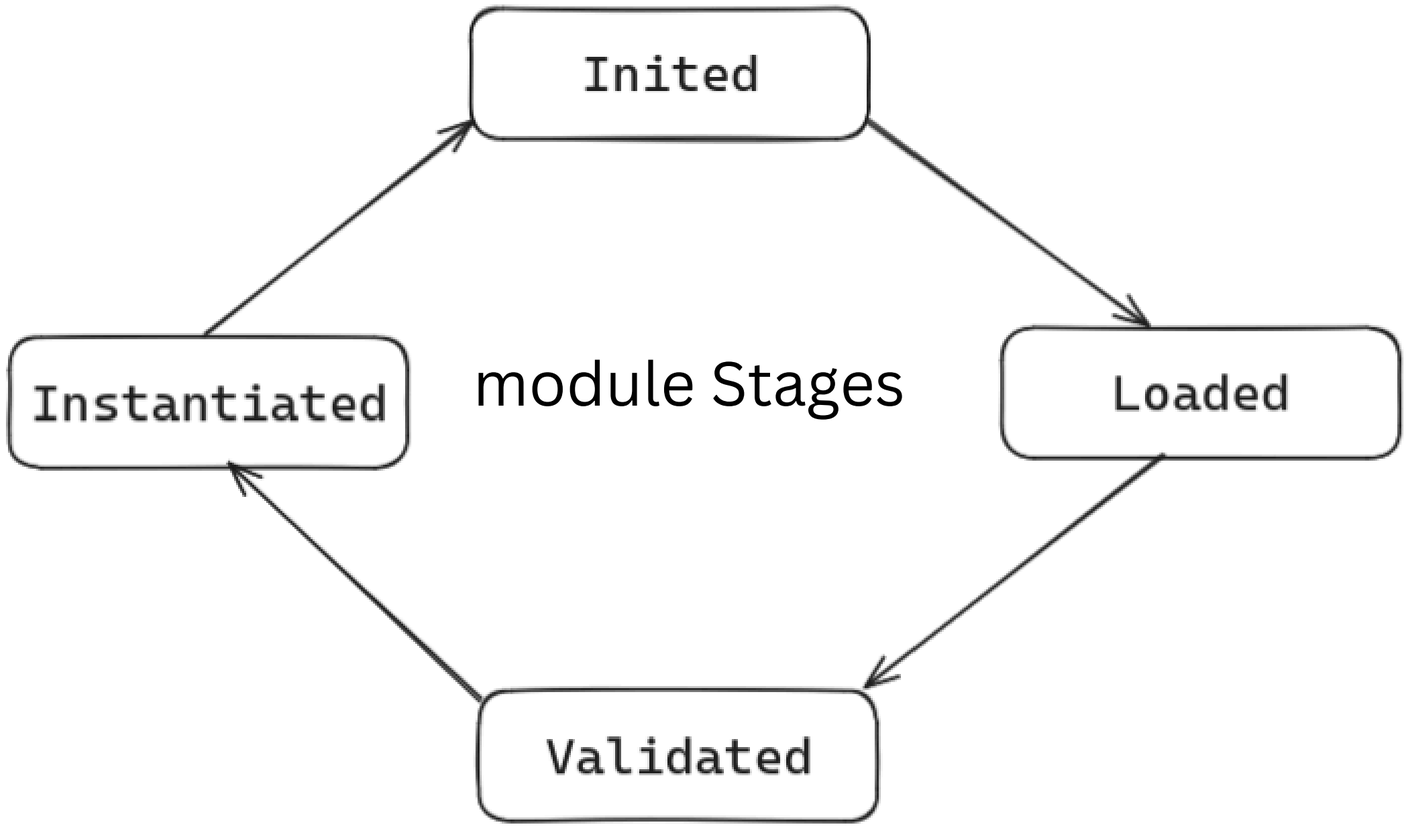
- a one-byte section id
- the size of the contents, in bytes
- the actual contents, whose structure is dependent on the section id.



https://news.ycombinator.com/item?id=33797615

Onto the runtime!


AOT Engine
Interpreter Engine
Execute WASM
Execute WASM
WasmEdge Engine
Host Function Call / Access Runtime Data
WasmEdge Runtime
Call Host Functions
Access Runtime Data
Host Functions
Runtime Data Manager
ction Call
Crypto-related Function Call
Socket-related Function Call
Access Memory
Access Stack
WASI-Crypto
WasmEdge-WASI-Socket
Memory Manager
Stack Manager


Inited
Loaded
Validated
Instantiated
module Stages

Lets recap

- The WasmEdge runtime follows a general flow: parsing the Wasm file, validating the parsed Wasm file, compiling the validated Wasm file into native code, and then executing the compiled code.

The stack and the store

- The stack and the store are the two most frequently interacted with components of the runtime an instantiated module interacts with
- We often say the stack is implicit, this is because you never directly interact with it. It is used to keep track of function calls and intermediate results.

- The stack has 3 kinds of entries:
  - Values: the operands of instructions.
  - Labels: active structured control instructions that can be targeted by branches.
  - Activations: the call frames of active function calls.

What goes into the stack?

- Activation frames/Call frames are structures that represents the state of an active function call.
- A frame is created each time a function is called and is removed when the function returns.
- Important for control flow integrity.
- hold the values of its locals (including fn arguments) in the order corresponding to their static local indexing + a reference to the function’s own module instance

```cpp
const Instance::ModuleInstance *getModule() const noexcept { return Module; }
Instance::MemoryInstance *getMemoryByIndex(uint32_t Index) const noexcept {
  if (Module == nullptr) {
    return nullptr;
  }
  if (auto Res = Module->getMemory(Index); Res) {
    return *Res;
  }
  return nullptr;
}
```

- Stack manager internally provides the stack control for Wasm execution with **validated** modules. All operations of the instructions have already passed validation and no unexpected operations will occur.
- This is an implicit stack! We cannot directly modify this stack, we can interact with it with instructions. This ensures CFI(control flow integrity)

Store Manager

- The store manager serves as a centralized class for the WebAssembly instance’s state, managing the linear memory, global variables, tables, and function references.
- New instances of functions, tables, memories, and globals are allocated in the store where it is kept track of.

Store Manager

$$
\begin{array}{lll}
store & ::= & \{\ \mathsf{funcs}\ \ funcinst^*, \\
 & & \ \ \mathsf{tables}\ \ tableinst^*, \\
 & & \ \ \mathsf{mems}\ \ meminst^*, \\
 & & \ \ \mathsf{globals}\ \ globalinst^*, \\
 & & \ \ \mathsf{elems}\ \ eleminst^*, \\
 & & \ \ \mathsf{datas}\ \ datainst^*\ \}
\end{array}
$$

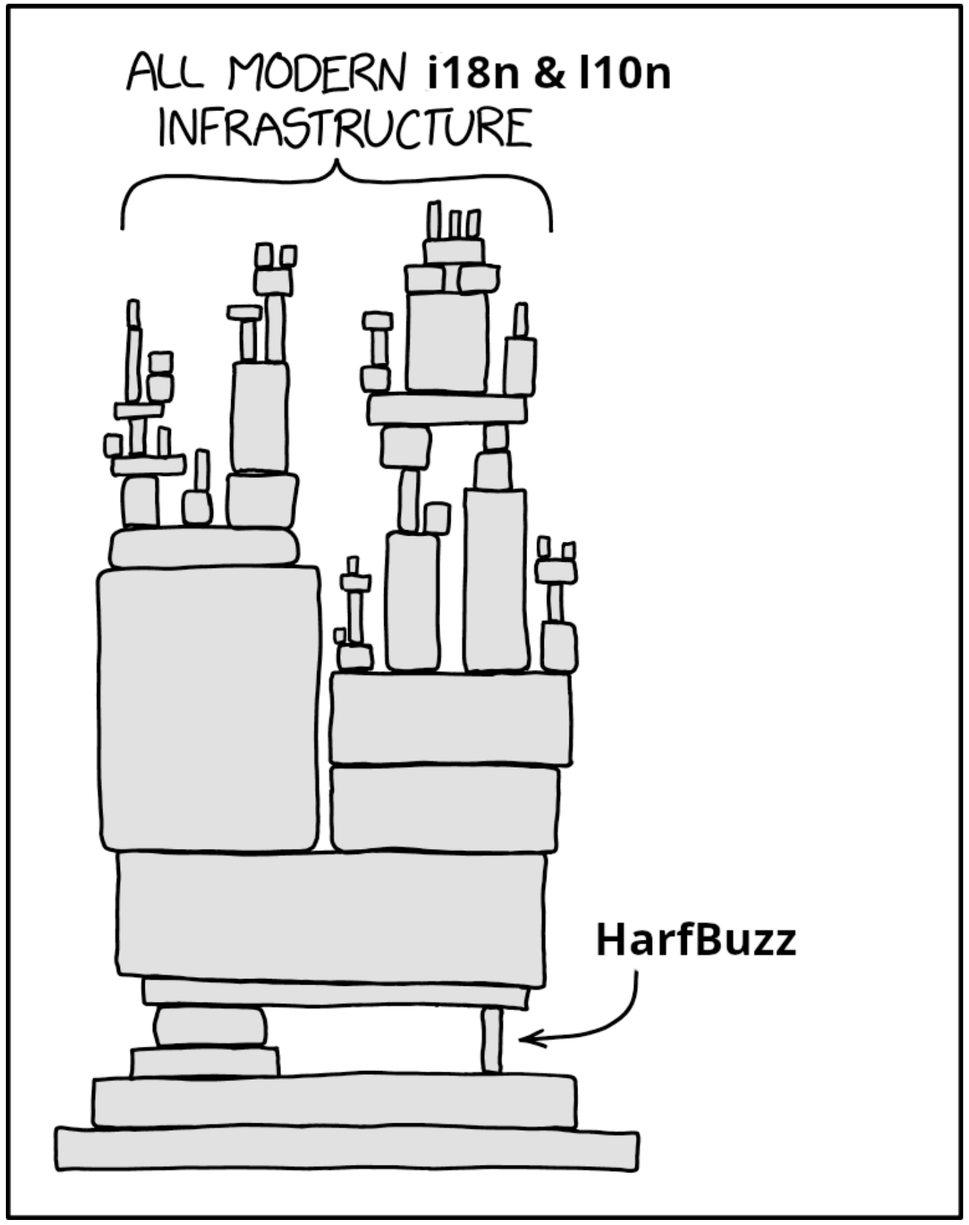
Linear Memory Model

- Linear memory? Its exactly what it sounds like. Its represented as just a vector of raw bytes.
- Its represented in terms of page size(the min/max possible size)
- Each module is given its own linear memory. You can exchange data between modules through host functions.

Traps

- Traps in Wasm are pretty much generated whenever there is a out of bounds access or a type overflow, there’s a non exhaustive list of cases
- But point being, whenever a trap is generated it is not handled by the Wasm runtime, trap handling is passed off to the host embedding.
- (My job is to implement the coredump spec for the WasmEdge runtime)

Why I love Wasm



https://fuglede.github.io/llama.ttf/

https://dingboard.com

https://wingolog.org/archives/2024/01/08/missing-the-point-of-webassembly

“WebAssembly is a new fundamental abstraction boundary. WebAssembly is a new way of dividing computing systems into pieces and of composing systems from parts.”

Other things you should check out:

- WASI’s latest preview
- WasmEdge Plugin system
- The Wasm GC proposal
- This article: https://wingolog.org/archives/2023/11/24/tree-shaking-the-horticulturally-misguided-algorithm

Fin