Web Assembly & Go Implementation

Definition

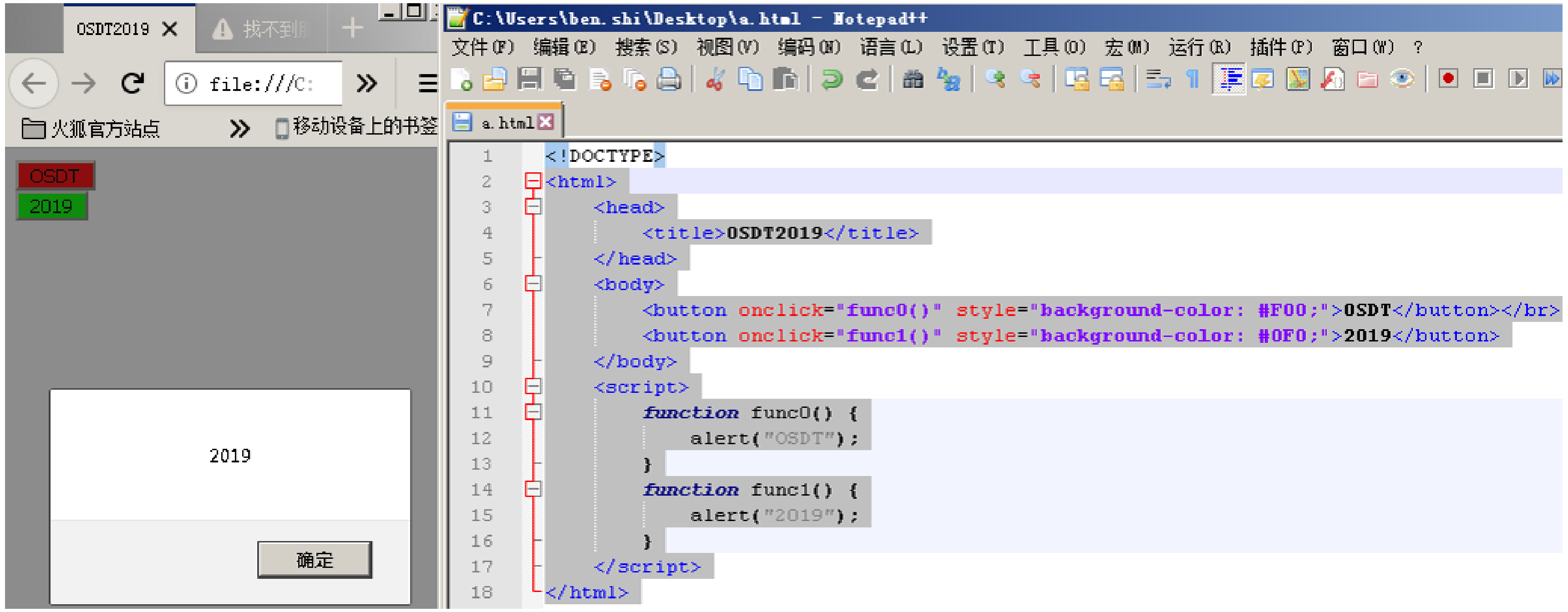
WebAssembly (abbreviated Wasm) is a binary instruction format for a stack-based virtual machine. Wasm is designed as a portable target for compilation of high-level languages like C/C++/Rust, enabling deployment on the web for client and server applications.

Key Points

- Write web program with C/C++/Rust
- AOT

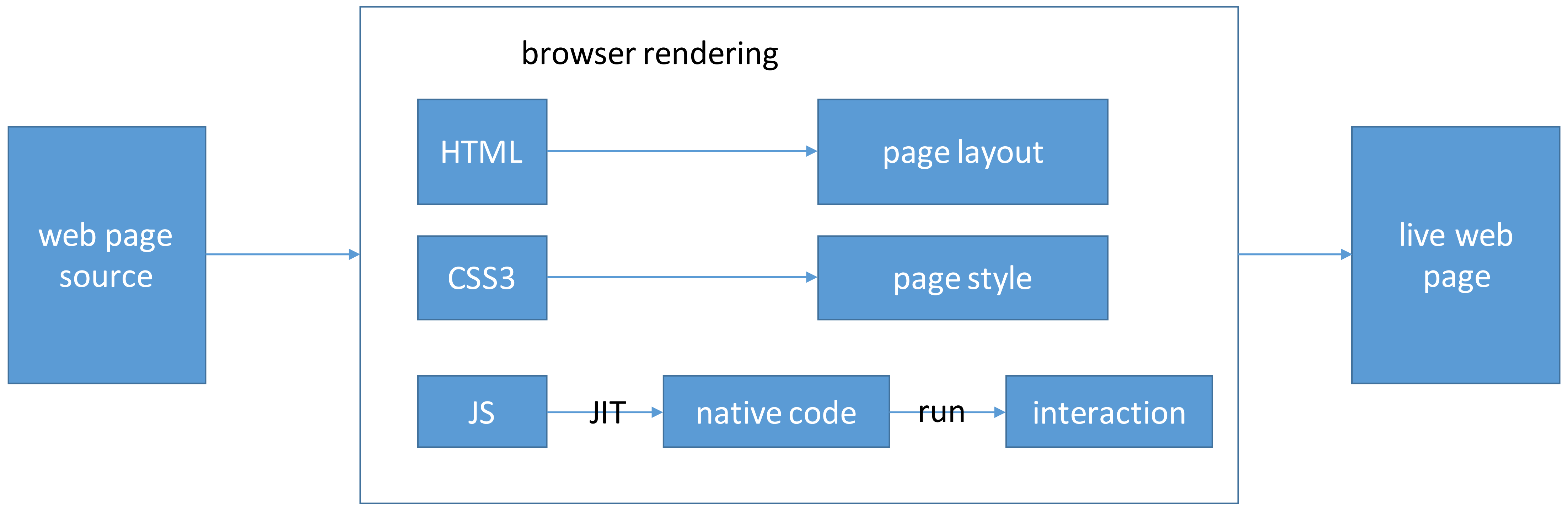
Web Page

- HTML – layout, skeleton
- CSS3 – appearance, beauty, fashion, …
- JavaScript – interaction, soul, mind, …


OSDT2019
找不到
file:///C:
火狐官方站点
移动设备上的书签
OSDT
2019
2019
确定
C:\Users\ben.shi\Desktop\a.html - Notepad++
文件(F) 编辑(E) 搜索(S) 视图(V) 编码(N) 语言(L) 设置(T) 工具(O) 宏(M) 运行(R) 插件(P) 窗口(W) ?
a.html

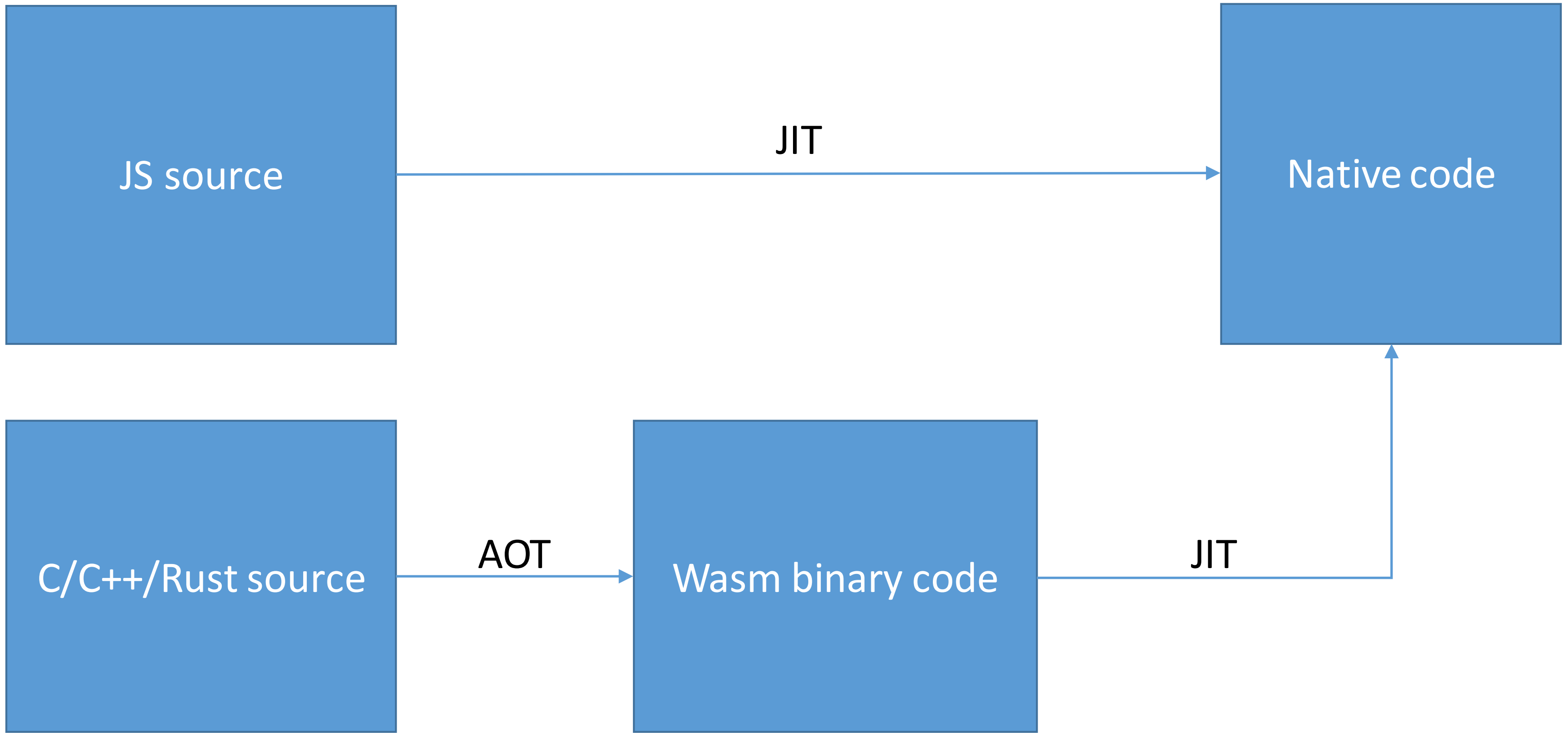
```
<!DOCTYPE>
<html>
    <head>
        <title>OSDT2019</title>
    </head>
    <body>
        <button onclick="func0()" style="background-color: #F00;">OSDT</button></br>
        <button onclick="func1()" style="background-color: #0F0;">2019</button>
    </body>
    <script>
        function func0() {
            alert("OSDT");
        }
        function func1() {
            alert("2019");
        }
    </script>
</html>
```

Web Page Rendering



issue: longer compilation time -> slower web page loading

AOT vs JIT


JS source
JIT
Native code
C/C++/Rust source
AOT
Wasm binary code
JIT

Wasm Specification

- Wasm core: VM architecture, instruction set
- JS API: procedure call between JS and Wasm

- https://webassembly.github.io/spec/

Example: JS & Wasm Call Each Other

- Wasm programs are referred as modules in JS
- Wasm’s start function runs as constructor when a module is instantiated
- JS functions are referred by indexes in Wasm

```
(module
    (import "js" "import1" (func $i1))
    (import "js" "import2" (func $i2))
    (func $main (call $i1))
    (start $main)
    (func (export "f") (call $i2))
)
```

```
var importObj = {js: {
    import1: () => console.log("hello,"),
    import2: () => console.log("world!")
}};
fetch('demo.wasm').then(response =>
    response.arrayBuffer()
).then(buffer =>
    WebAssembly.instantiate(buffer, importObj)
).then(({module, instance}) =>
    instance.exports.f()
);
```

Wasm VM: Stack Based (not register)

- variables are loaded/stored from memory/stack to stack/memory
- operands are poped from stack top, and result is pushed to stack top

i64.sub (and other binary instructions)

1. pop a 64-bit operand as C from the stack;
2. pop a 64-bit operand as B from the stack;
3. push the result of B-C to the stack;

bit-wise not (and other unary instructions)

1. pop a 32-bit bool operand as A from the stack;
2. push the result of ~A to the stack;

```
uint64 A = B - C; // is compiled to

global.get C     // global C -> stack
local.get B      // local B -> stack
i64.sub          // subtraction
local.set A      // stack -> local A
```

Wasm VM Components

- a global universal stack
- memory pieces
- global variables (variables are not in memory)
- local variables (each function private)
- functions
- a start function
- imports / exports
- others: tables, element segments, data segments, …

Data Types

- int32 / uint32
- int64 / uint64
- float32 / float64
- bool (32-bit)

Instructions (1)

- const (push a constant to the stack)
- arithmetic instructions (add, sub, smul, udiv, …)
- bit-wise instructions (and, or, not, xor)
- shift (logic shift, arithmetic shift, rotation)
- float32 / float64 (add, sub, mul, div, …)
- unconditional branch / jmp (no change to stack)
- conditional branch / jmp (consumes the stack top as a 32-bit bool)
- relation (>, >=, ==, !=, <, <=)

Instructions (2)

- local.get / local.set (32-bit or 64-bit)
- global.set / global.set (32-bit or 64-bit)
- load (load8s, load8u, load16s, load16u, load32s, load32u, load64)
- store (store8, store16, store32, store64)
- call / return
- type conversion
- others: abs, min, max, sqrt, …

Tool Chain Support

- clang / llvm
- go
- others

Runtime Environment Support

- firefox / chrome / safari
- node.js (run JS on the server side)

Go Implementation

- compiler (go source -> textual Wasm instruction)
- assembler (textural Wasm instruction -> binary Wasm instruction)
- linker (not ELF, a special format)

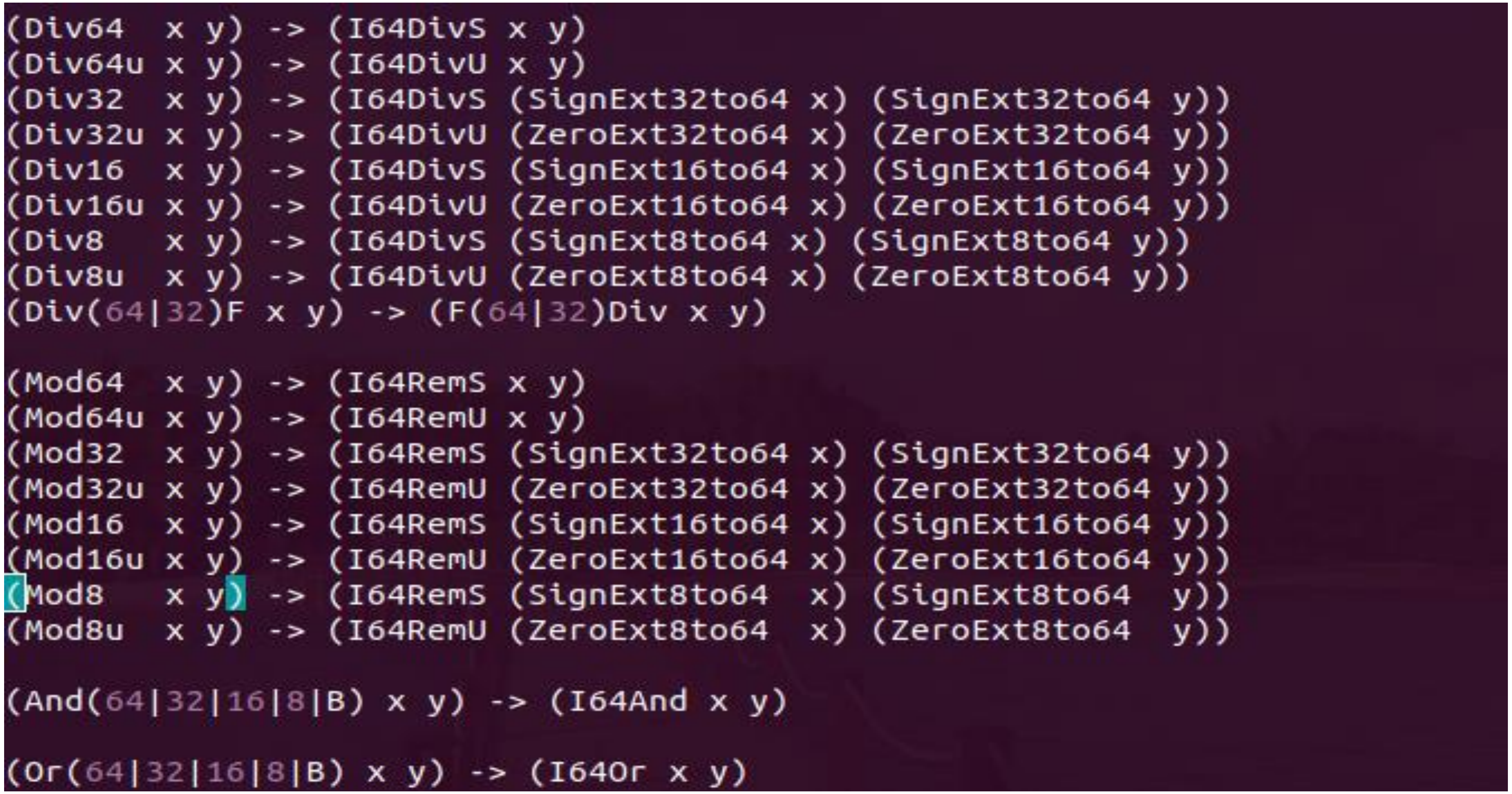
Go Compiler

- (of course) written in go
- SSA based (with well documented IR)
- 48 passes (much less than LLVM)
- efficient register allocator
- arch-dependent peephole
- OS support: Windows, GNU/Linux, *BSD, MacOS, Android
- HW support: arm/aarch64, mips/mips64, x86/amd64, ppc/ppc64
- Wasm support (experimental)

Concerns of Porting Wasm to Go

- unlimited local variables -> 16 fake registers
- GOOS=js GOARCH=wasm (neither real OS nor bare metal)
- a fake stand alone program (depends on a JS wrapper)
- no standard input
- standard output -> JS console.log()
- special output format (not ELF)
- go’s global variables -> Wasm global variable space
- arrays/slices/maps -> memory pieces
- JS is single threaded

Mapping Go IR to Wasm Instruction



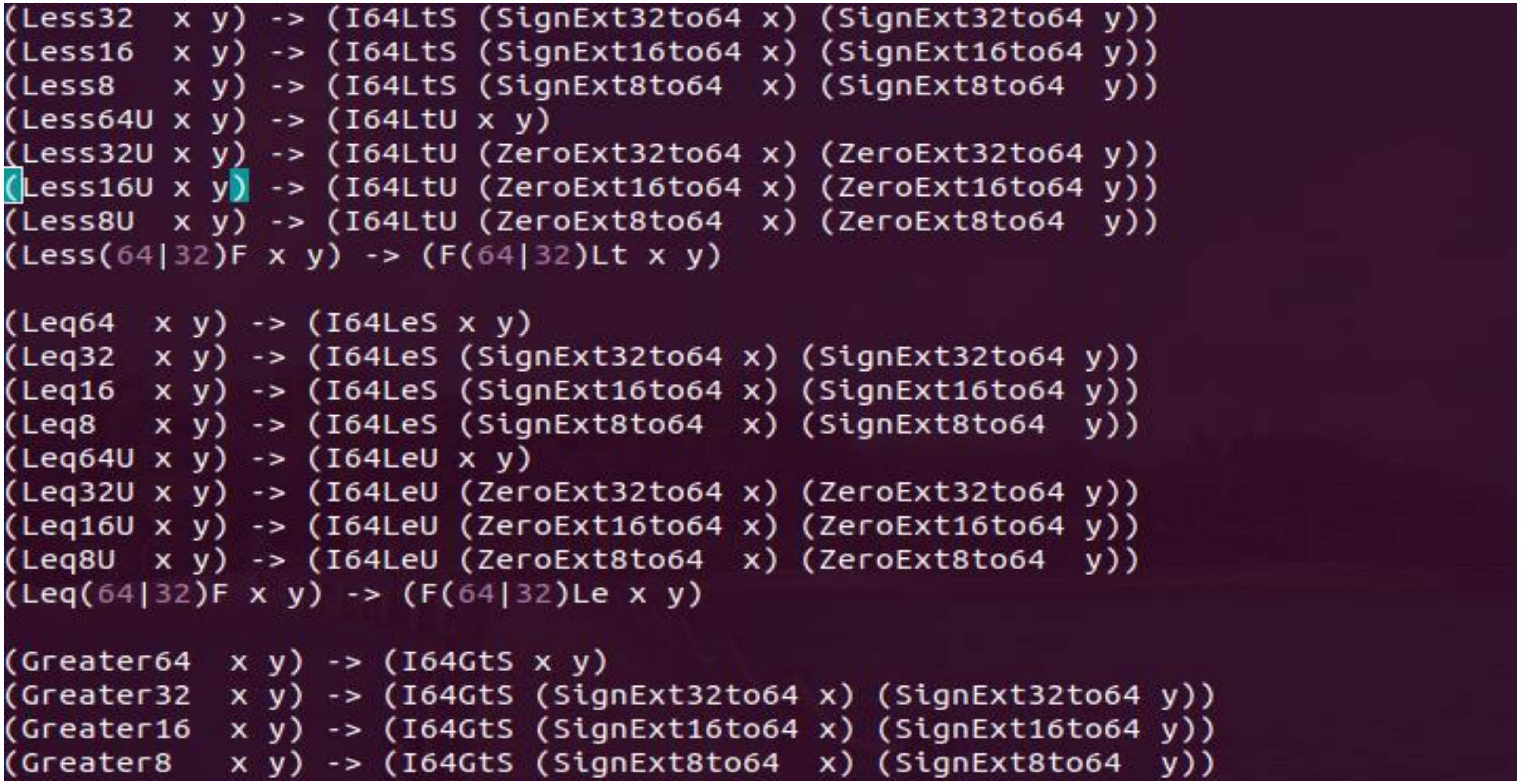
```
(Div64  x y) -> (I64DivS x y)
(Div64u x y) -> (I64DivU x y)
(Div32  x y) -> (I64DivS (SignExt32to64 x) (SignExt32to64 y))
(Div32u x y) -> (I64DivU (ZeroExt32to64 x) (ZeroExt32to64 y))
(Div16  x y) -> (I64DivS (SignExt16to64 x) (SignExt16to64 y))
(Div16u x y) -> (I64DivU (ZeroExt16to64 x) (ZeroExt16to64 y))
(Div8   x y) -> (I64DivS (SignExt8to64 x) (SignExt8to64 y))
(Div8u  x y) -> (I64DivU (ZeroExt8to64 x) (ZeroExt8to64 y))
(Div(64|32)F x y) -> (F(64|32)Div x y)

(Mod64  x y) -> (I64RemS x y)
(Mod64u x y) -> (I64RemU x y)
(Mod32  x y) -> (I64RemS (SignExt32to64 x) (SignExt32to64 y))
(Mod32u x y) -> (I64RemU (ZeroExt32to64 x) (ZeroExt32to64 y))
(Mod16  x y) -> (I64RemS (SignExt16to64 x) (SignExt16to64 y))
(Mod16u x y) -> (I64RemU (ZeroExt16to64 x) (ZeroExt16to64 y))
(Mod8   x y) -> (I64RemS (SignExt8to64  x) (SignExt8to64  y))
(Mod8u  x y) -> (I64RemU (ZeroExt8to64  x) (ZeroExt8to64  y))

(And(64|32|16|8|B) x y) -> (I64And x y)

(Or(64|32|16|8|B) x y) -> (I64Or x y)
```

Mapping Go IR to Wasm Instruction



```
(Less32  x y) -> (I64LtS (SignExt32to64 x) (SignExt32to64 y))
(Less16  x y) -> (I64LtS (SignExt16to64 x) (SignExt16to64 y))
(Less8   x y) -> (I64LtS (SignExt8to64  x) (SignExt8to64  y))
(Less64U x y) -> (I64LtU x y)
(Less32U x y) -> (I64LtU (ZeroExt32to64 x) (ZeroExt32to64 y))
(Less16U x y) -> (I64LtU (ZeroExt16to64 x) (ZeroExt16to64 y))
(Less8U  x y) -> (I64LtU (ZeroExt8to64  x) (ZeroExt8to64  y))
(Less(64|32)F x y) -> (F(64|32)Lt x y)

(Leq64  x y) -> (I64LeS x y)
(Leq32  x y) -> (I64LeS (SignExt32to64 x) (SignExt32to64 y))
(Leq16  x y) -> (I64LeS (SignExt16to64 x) (SignExt16to64 y))
(Leq8   x y) -> (I64LeS (SignExt8to64  x) (SignExt8to64  y))
(Leq64U x y) -> (I64LeU x y)
(Leq32U x y) -> (I64LeU (ZeroExt32to64 x) (ZeroExt32to64 y))
(Leq16U x y) -> (I64LeU (ZeroExt16to64 x) (ZeroExt16to64 y))
(Leq8U  x y) -> (I64LeU (ZeroExt8to64  x) (ZeroExt8to64  y))
(Leq(64|32)F x y) -> (F(64|32)Le x y)

(Greater64  x y) -> (I64GtS x y)
(Greater32  x y) -> (I64GtS (SignExt32to64 x) (SignExt32to64 y))
(Greater16  x y) -> (I64GtS (SignExt16to64 x) (SignExt16to64 y))
(Greater8   x y) -> (I64GtS (SignExt8to64  x) (SignExt8to64  y))
```

Experimental

A = B + C; // LLVM compilation

```
local.get C   // local 32-bit C -> stack
local.get B   // local 32-bit B -> stack
i32.add       // pop C, pop B, push B+C
local.set A   // stack -> local A
```

A = B + C; // inefficient go compilation

```
local.get C        // local 32-bit C -> stack
i64.extends32      // sign-extend stack top to 64-bit
local.get B        // local 32-bit B -> stack
i64.extends32      // sign-extend stack top to 64-bit
i64.add            // pop two operands, push their sum
i32.trunc64        // truncate stack top to 32-bit
local.set A        // stack -> local A
```

Todo

- emit real float32 and int32 instructions
- support newer instructions
- map more go library system calls to JS built-in functions
- stability & performance & code density
- performance of go routines
- GC (go’s built-in GC vs JS’s GC)
- other compromised designs (such as local variables -> 16 registers)

Thank You