Source to Binary

Journey of V8 javascript engine

Name

@brn (Taketoshi Aono)

Occupation

Web Frontend Engineer

Company

Cyberagent.inc

Blog

http://abcdef.gets.b6n.ch/

Twitter

https://twitter.com/brn227

GitHub

https://github.com/brn

Agenda

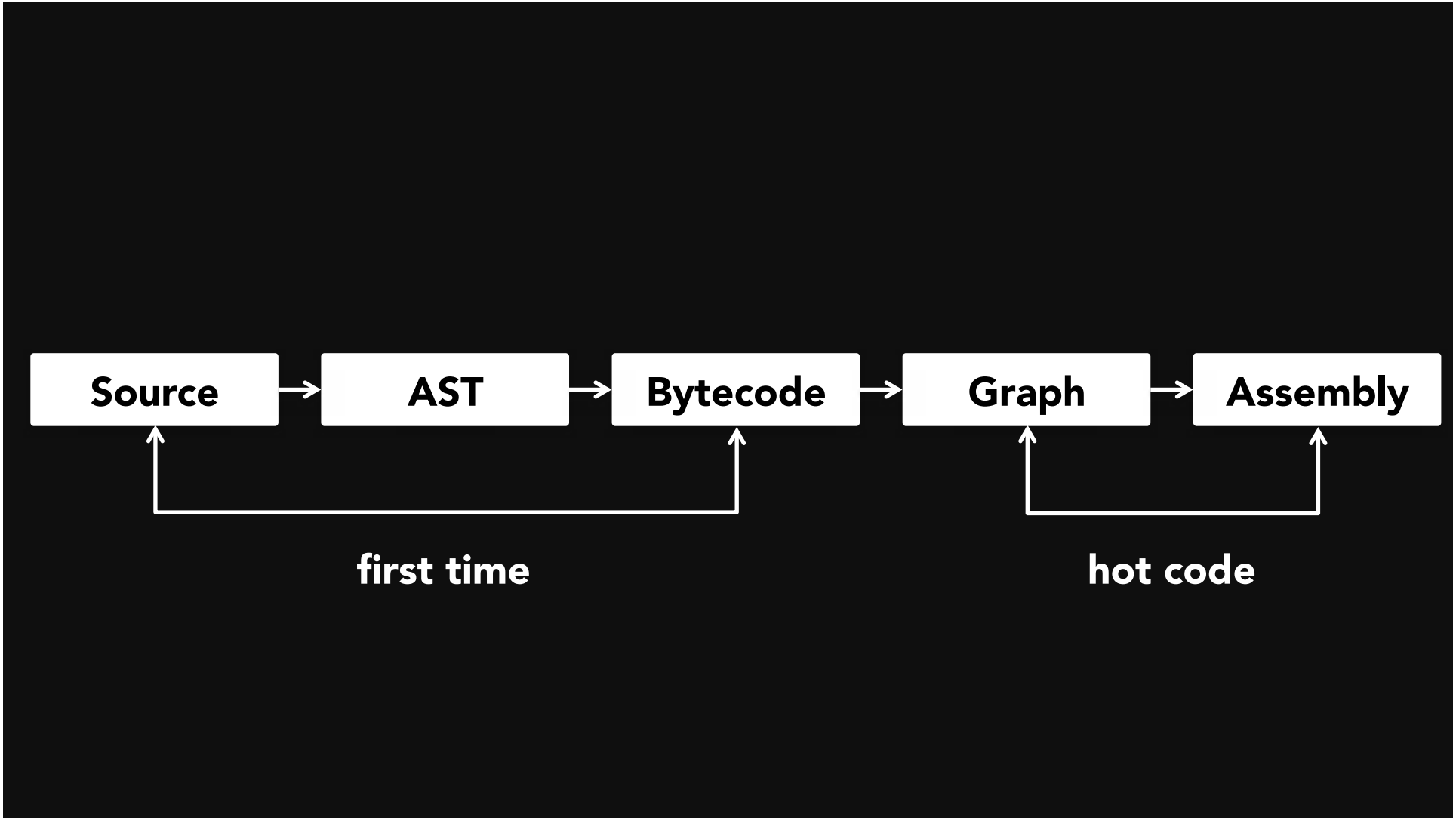
- What is V8?
- Execution flow of V8
- Parsing
- Abstract Syntax Tree
- Ignition – BytecodeInterpreter
- CodeStubAssembler
- Builtins / Runtime
- Optimization / Hidden Class / Inline Caching
- TurboFan / Deoptimization

What is V8?

V8 is javascript engine that has been developed by Google.

It's used as core engine of Google Chrome and Node.JS.

Execution Flow


Source
AST
Bytecode
Graph
Assembly
first time
hot code

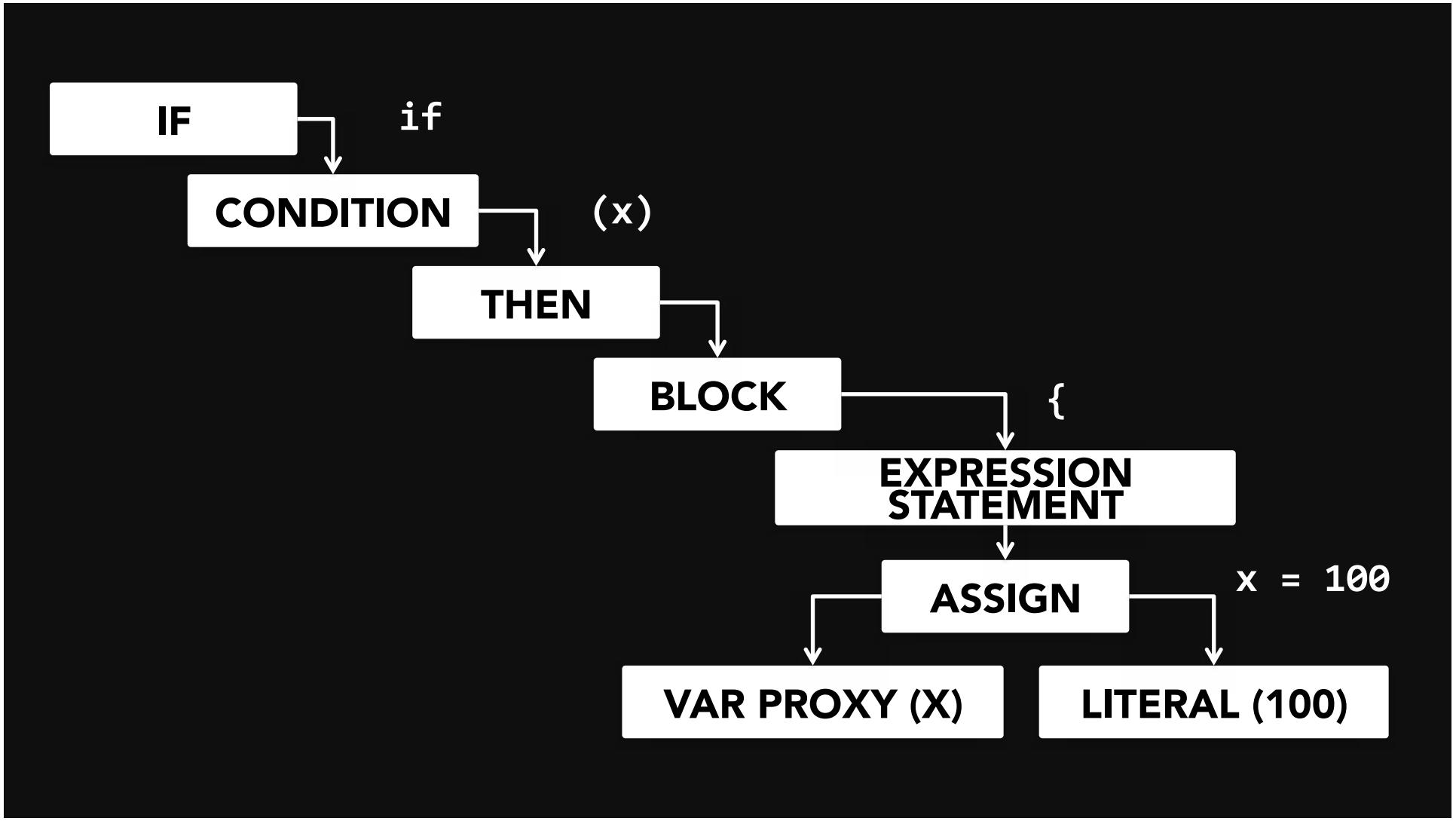
Parsing

Basic parsing

V8 make source code to AST by parsing.

AST is abbreviation of Abstract Syntax Tree.

```
if (x) {
   x = 100
}
```


IF
if
CONDITION
(x)
THEN
BLOCK
{
EXPRESSION STATEMENT
ASSIGN
x = 100
VAR PROXY (X)
LITERAL (100)

Problems

Parsing all functions - Slow

Parsing all source code is bad strategy because if parsed code will not executed, it's waste of time.

Split parsing phase

So split parsing phase to parse lazily.

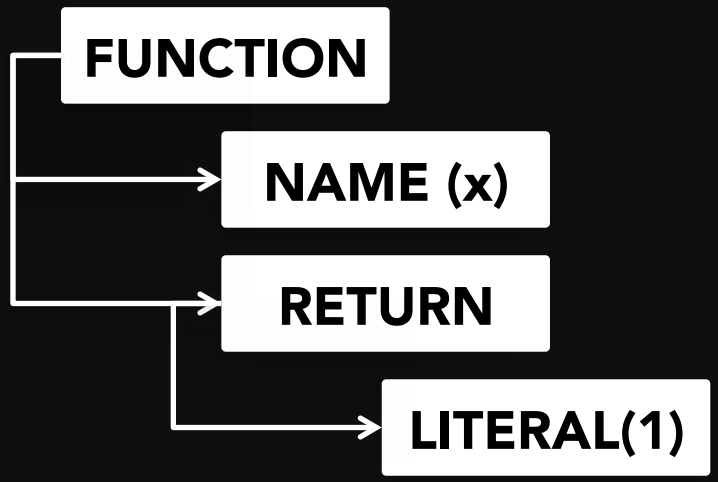
PreParsing

Parse function layout in advance.

```
function x(a, b) {
  return a + b;
}
```

FUNCTION(X)
parameter-count: 2
start-position: 1
end-position: 34
use-super-property: false
...

```
// when x is called
x()
```



Lazy Parsing

V8 delay parsing until function called in runtime.

Function will be compiled only when called.

More About

https://docs.google.com/presentation/d/1b-ALt6W01nIxutFVFmXMOyd_6ou_6qqP6S0Prmb1iDs/present?slide=id.p

Abstract Syntax Tree

AST Rewriting

V8 rewrite AST.

Show some examples.

Subsclass constructor return

Transform derived constructor.

If it return any value, V8 transform that to ternary operator to return this keyword when return value will become undefined.

```
constructor() {
  super();
  return expr
}

constructor() {
  super();
  var tmp;
  return (temp = expr) === undefined?
    this: temp;
}
```

for (let/const/var in/of e)

To use const or let in initialization statement in for-of/in statement, V8 move all statement into block.

```
for (const key of e) {
  ...
}
```

```
{
  var temp;
  for (temp of e) {
    const x =  temp;
    ...
  }
  let x
}
```

AbstractSyntaxTree

Spread operator

Replaced by do and for-of.

```
const x = [1,2,3];
const y = [...x];
```

```
do {
  $R = [];
  for ($i of x)
    %AppendElement($R, $i);
  $R
}
```

Ecmascript? – Binary AST

AST size is fairly big, so Ecmascript has proposal 'Binary AST'.

This proposal proposed to compressed form of AST.


Ignition

Bytecode Interpreter

V8 execute AST by transform it with bytecode which size will 1 ~ 4 byte.

How does it work?

It is one accumulator register based interpreter.

Do you understand?

I can't :(

Pseudo javascript code

So now I show pseudo javascript code which show how Ignition works.

```
const Bytecodes = [0,1,2,3,4,5];
let index = 0;
function dispatch(next) {BytecodeHandlers[next]
();}
const BytecodeHandlers = [
  () => {...; dispatch(Bytecodes[index++])},
  () => {...; dispatch(Bytecodes[index++])},
  () => {...; dispatch(Bytecodes[index++])},
  () => {...; dispatch(Bytecodes[index++])},
  () => {...; dispatch(Bytecodes[index++])},
  () => {...; dispatch(Bytecodes[index++])},
]
dispatch(Bytecodes[index++]);
```

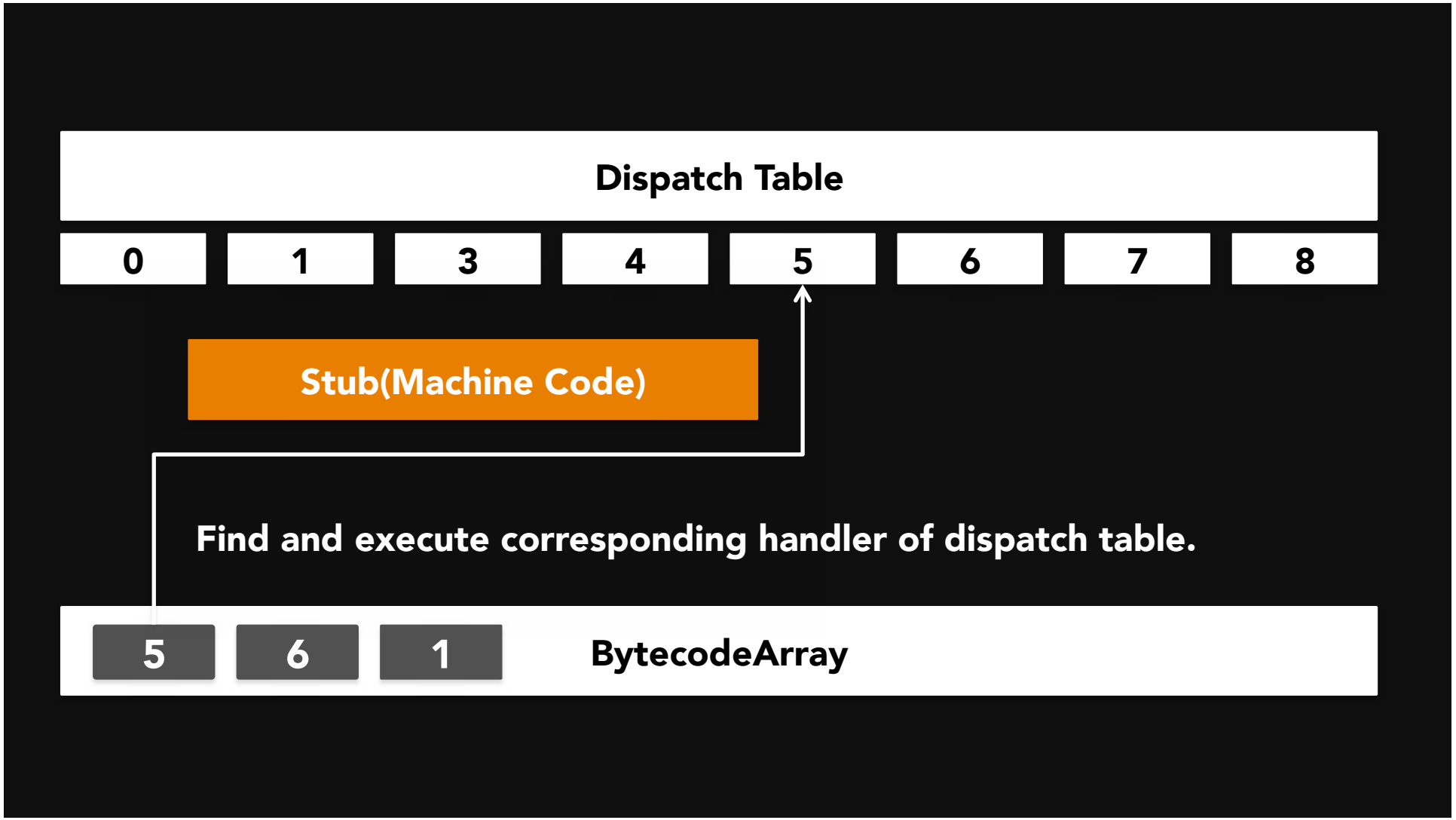
How to create bytecode?

Bytecode will created by AstVisitor which is visitor pattern based class that visit AST by Depth-First-Search and callback each AST.

BytecodeArray

Bytecode will be stored in BytecodeArray.

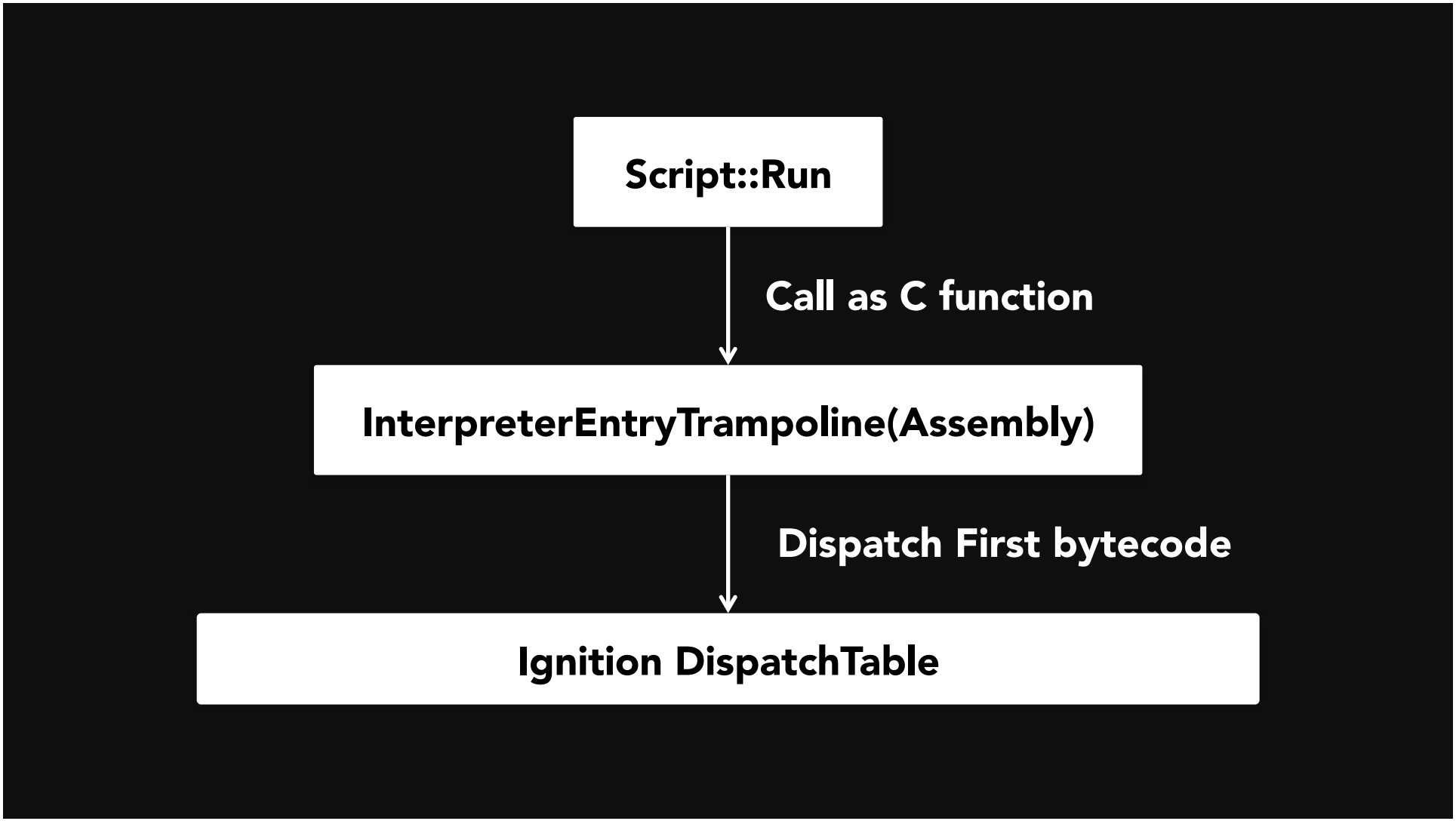
BytecodeArray exists in each javascript function.


Dispatch Table
0
1
3
4
5
6
7
8
Stub(Machine Code)
Find and execute corresponding handler of dispatch table.
5
6
1
BytecodeArray

InterpreterEntryTrampoline

Finally created bytecode will invoke from a builtin code that named as InterpreterEntrynTrampoline.

InterpreterEntryTrampoline is C laanguage function that written in Assembly.


Script::Run
Call as C function
InterpreterEntryTrampoline(Assembly)
Dispatch First bytecode
Ignition DispatchTable

Ignition Handler

In pseudo javascript code, array named BytecodeHandlers is called as Ignition Handler in V8.

Ignition Handler is created by DSL named CodeStubAssembler.

CodeStub Assember

What is CodeStubAssmber?

CodeStubAssembler(CSA) abstracts code generation to graph creation.

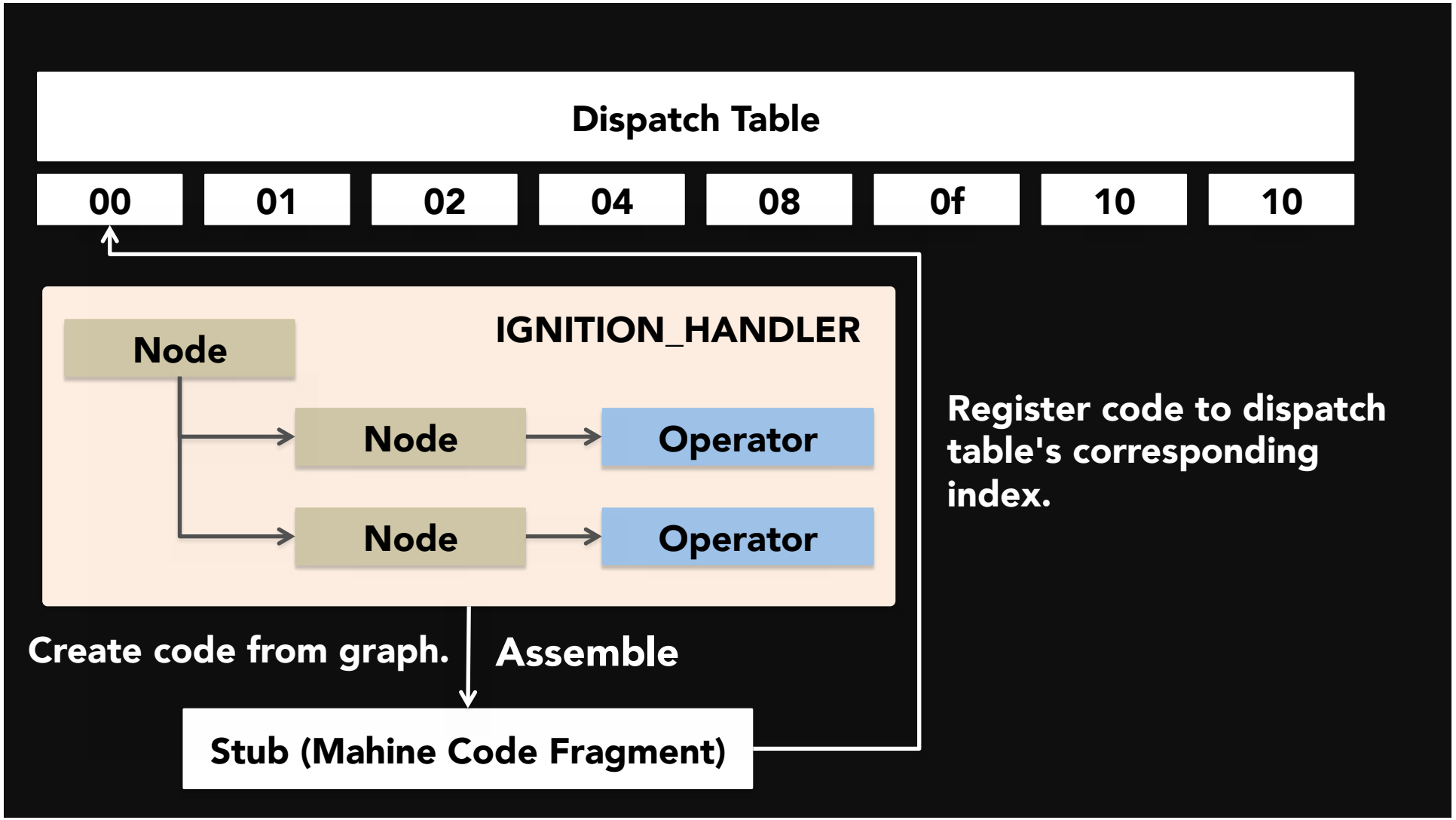
It's just only create execution scheduled node, and CodeGenerator convert it to arch dependent code, so you do not need to become expert of assembly language.

```
IGNITION_HANDLER(JumpIfToBooleanFalse, InterpreterAssembler) {
  Node* value = GetAccumulator();
  // Get Accumulator value.
  Node* relative_jump = BytecodeOperandUImmWord(0);
  // Get operand value from arguments.
  Label if_true(this), if_false(this);
  BranchIfToBooleanIsTrue(value, &if_true, &if_false);
  // If value will true jump to if_true,
  // otherwise jump to if_false.
  Bind(&if_true);
  Dispatch();
  Bind(&if_false);
  // Jump to operand bytecode.
  Jump(relative_jump);
}
```

Graph based DSL

CodeStubAssembler make code very easy and clean.

So it enable add new language functionality fast.


Dispatch Table
00
01
02
04
08
0f
10
10
IGNITION_HANDLER
Node
Node
Operator
Node
Operator
Register code to dispatch table's corresponding index.
Create code from graph.
Assemble
Stub (Mahine Code Fragment)

CodeStubAssembler

Assembler

Emit arch dependent code.

Let see jmp mnemonic.

```
void Assembler::jmp(
    Handle<Code> target,
    RelocInfo::Mode rmode
) {
    EnsureSpace ensure_space(this);
    // 1110 1001 #32-bit disp.
    // Emit assembly.
    emit(0xE9);
    emit_code_target(target, rmode);
}
```

Where to use

The builtins uses Assembler class to write architecture dependent stub.

But there are some CSA based code (*-gen.cc).

Ignition Handler is almost all written in CSA.

Builtins & Runtime

Builtins

Builtins is collection of assembly code fragment which compiled in V8 initialization.

It's called as stub.

Runtime optimization is not applied.

Runtime

Runtime is written in C++ and will be invoked from Builtins or some other assembler code.

It's code fragments connect javascript and C++.

Not optimized in runtime.

Hidden Class

What is Hidden Class?

Javascript is untyped language, so V8 treat object structure as like type.

This called as Hidden Class.

- Hidden Class

```
const point1 = {x: 0, y: 0};
const point2 = {x: 0, y: 0};
```

Map
FixedArray [
{x: {offset: 0}},
{y: {offset: 1}}
]

Hidden Class

Map

If each object is not treat as same in javascript.

But if these object has same structure, these share same Hidden Class.

That structure data store is called as Map.

```
const pointA = {x: 0, y: 0};
const pointB = {x: 0, y: 0};
// pointA.Map === pointB.Map;

const pointC = {y: 0, x: 0};
// pointA.Map !== pointC.Map

const point3D = {x: 0, y: 0, z: 0};
// point3D.Map !== pointA.Map
```

```
class PointA {
  constructor() {
    this.x = 0;
    this.y = 0;
  }
}
const pointAInstance = new PointA();

class PointB {
  constructor() {
    this.y = 0;
    this.x = 0;
  }
}
const pointBInstance = new PointB();
// PointAInstance.Map !== PointBInstance.Map
```

Layout

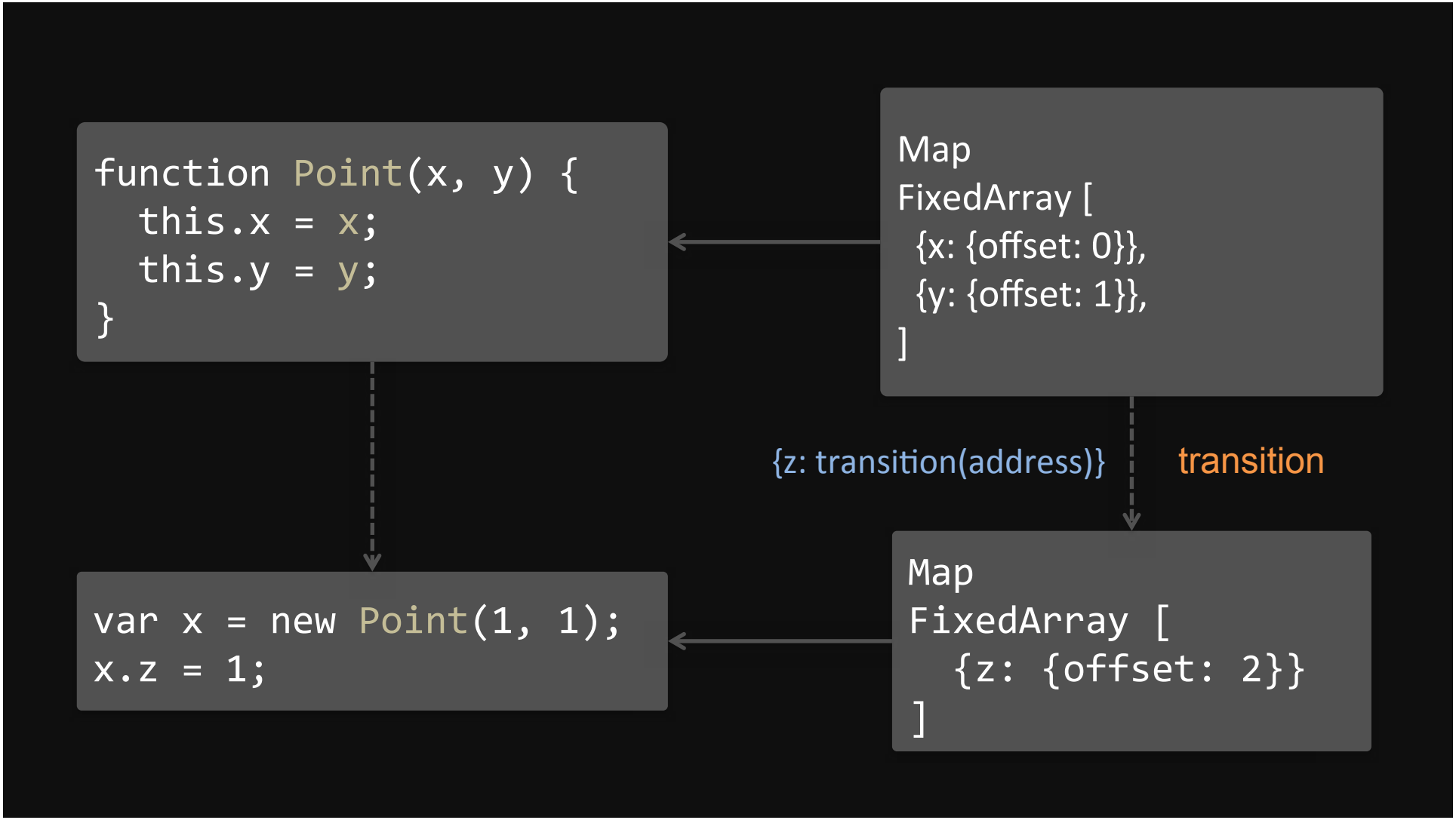
Map object checks object layout very strictly, so if literal initialization order, property initialization order or property number is different, allocate other Map.

Map Transition

But, isn't it pay very large cost to allocate new Map object each time when property changed?

So V8 share Map object if property changed, and create new Map which contains new property only.

That is called as Map Transition.


function Point(x, y) {
this.x = x;
this.y = y;
}
Map
FixedArray [
{x: {offset: 0}},
{y: {offset: 1}},
]
{z: transition(address)}
transition
var x = new Point(1, 1);
x.z = 1;
Map
FixedArray [
{z: {offset: 2}}
]

What's Happening?

Why Hidden Class exists?

Because it make property access or type checking to more fast and safe.

Inline Caching

What is Inline Caching

Cache accessed object map and offset to speed up property access.

```
function x(obj) {
  return obj.x + obj.y;
}

x({x: 0, y: 0});
x({x: 1, y: 1});
x({x: 2, y: 2});
```

Search Property

To find property from object, it's need search HashMap or FixedArray.

But if executed each time when property accessed, it's very slow.

Reduce Property Access

In that examples, repeatedly access to x and y of same Map object.

If V8 already know obj has {x, y} Map, V8 know memory layout of object.

So it's able to access offset directly to speed up.

Cache

So remember access of specific Map.

If V8 accesses any property, it record the Map object and speed up second time property access.

```
x({x: 0, y: 0});
// uninitialized
x({x: 1, y: 1});
// stub_call
x({x: 2, y: 2});
// found ic
x({x: 1, y: 1, z: 1})
// load ic miss
x({x: 1, y: 1, foo() {}});
// load ic miss
```

Cache Miss

Cache miss will be occurred when Map was changed, so new property will be loaded and stored in cache.

But it's impossible to record all Map, so max 4 Map will record.

Cahce State

Cache has below state.

- PreMonomorphic
- Monomorphic
- Polymorphic
- Megamorphic

Pre Monomorphic

It's shows initialization state.

But it's exists only convenience of coding.

So it's meaningless for ours.

Monomorphic

State which exists only one Map.

Ideal states.

Polymorphic

Some Map stored in FixedArray and search these Mpas each time when property accessed.

But cache is still enabled, so still fast.

Megamorphic

To many cache miss hit occurred, V8 stop recording Map.

Always call GetProperty function from stub.

Very slow state.

Optimization

Hot or Small

Optimizing code every time is very waste of resource.

So V8 is optimizing code when below conditions satisfied.

- Function is called (Bytecode length of function / 1200) + 2 times and exhaust budget.
- Function is very small (Bytecode length is less than 90)
- Loops

Optimization Budget

Optimization budget is assigned to each functions.

If function exhaust that budget, that function becomes candidate of optimization.

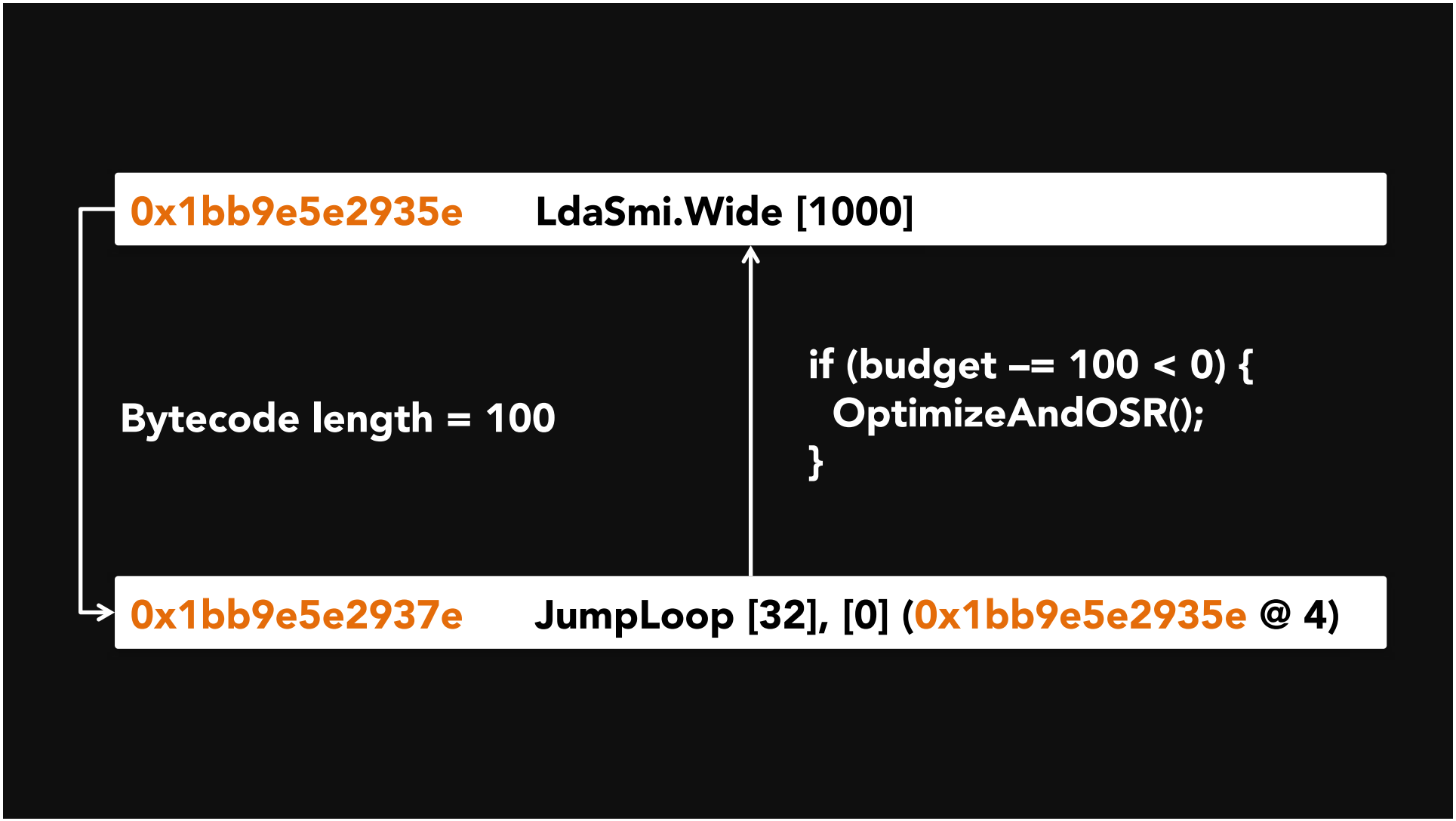
For Loop

V8 emits JumpLoop bytecode for loop statement.

In this JumpLoop bytecode, V8 subtract weight that is offset of backword jump address from budget.

If budget becomes less than 0, optimization will occurs.

```
function id(v) {return v;}
function x(v) {
  for (var i = 0; i < 10000; i++) {
    id(v + i);
  }
}
x(1);
```


0x1bb9e5e2935e LdaSmi.Wide [1000]
Bytecode length = 100
if (budget –= 100 < 0) {
OptimizeAndOSR();
}
0x1bb9e5e2937e JumpLoop [32], [0] (0x1bb9e5e2935e @ 4)

OSR - OnStackReplacement

Optimized code will be installed by replacing jump address.

It's called as OSR – OnStackReplacement.

For Function

V8 emits Return Bytecode for function.

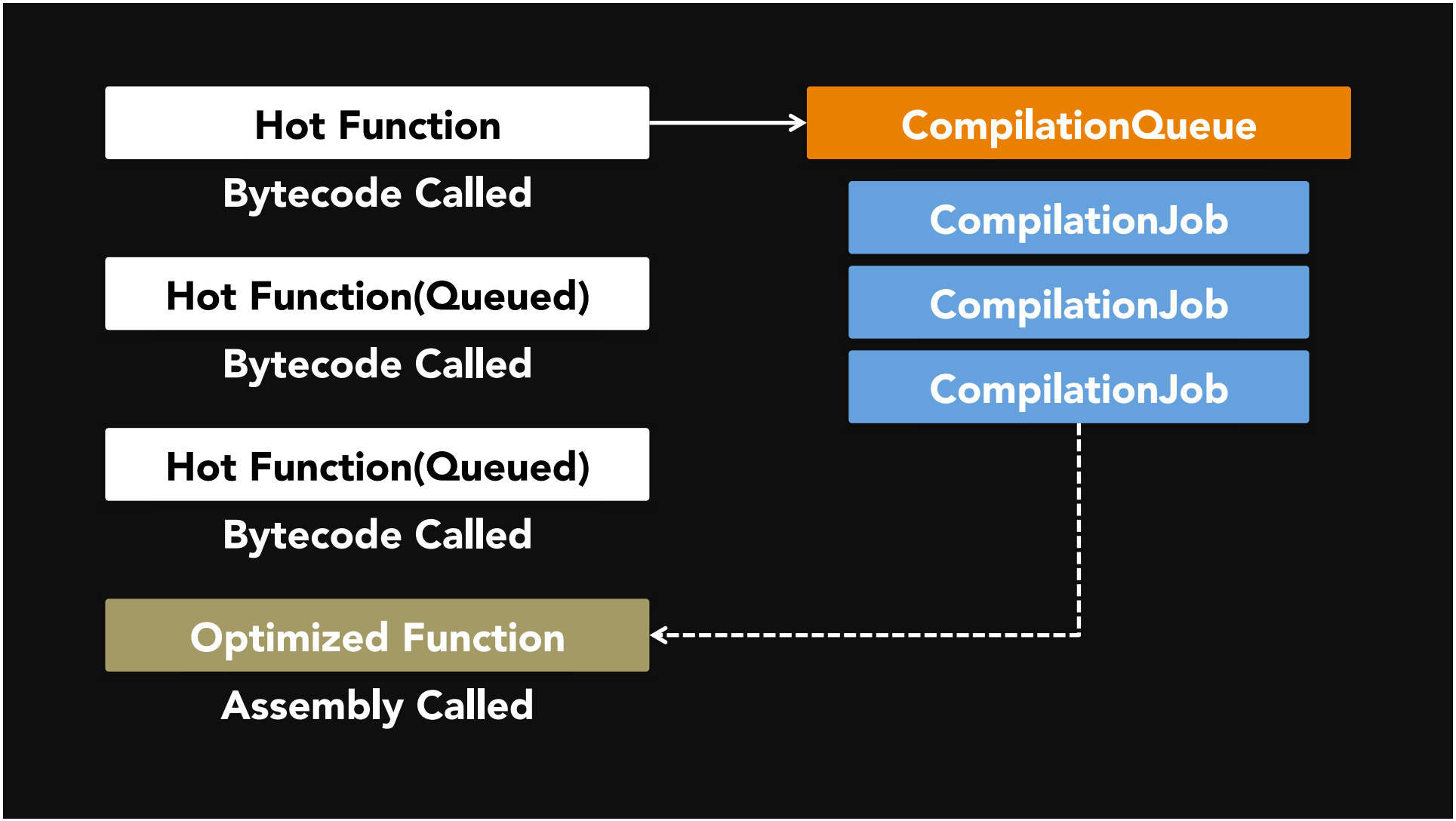
V8 check budget in that Bytecode.

```
function x() {
  const x = 1 + 1;
}
x();
```


0x3d22953a917a StackCheck
Bytecode length 30
0x3d22953a9180 Return
if (budget -= 30 < 0) {
OptimizeConcurrent();
}

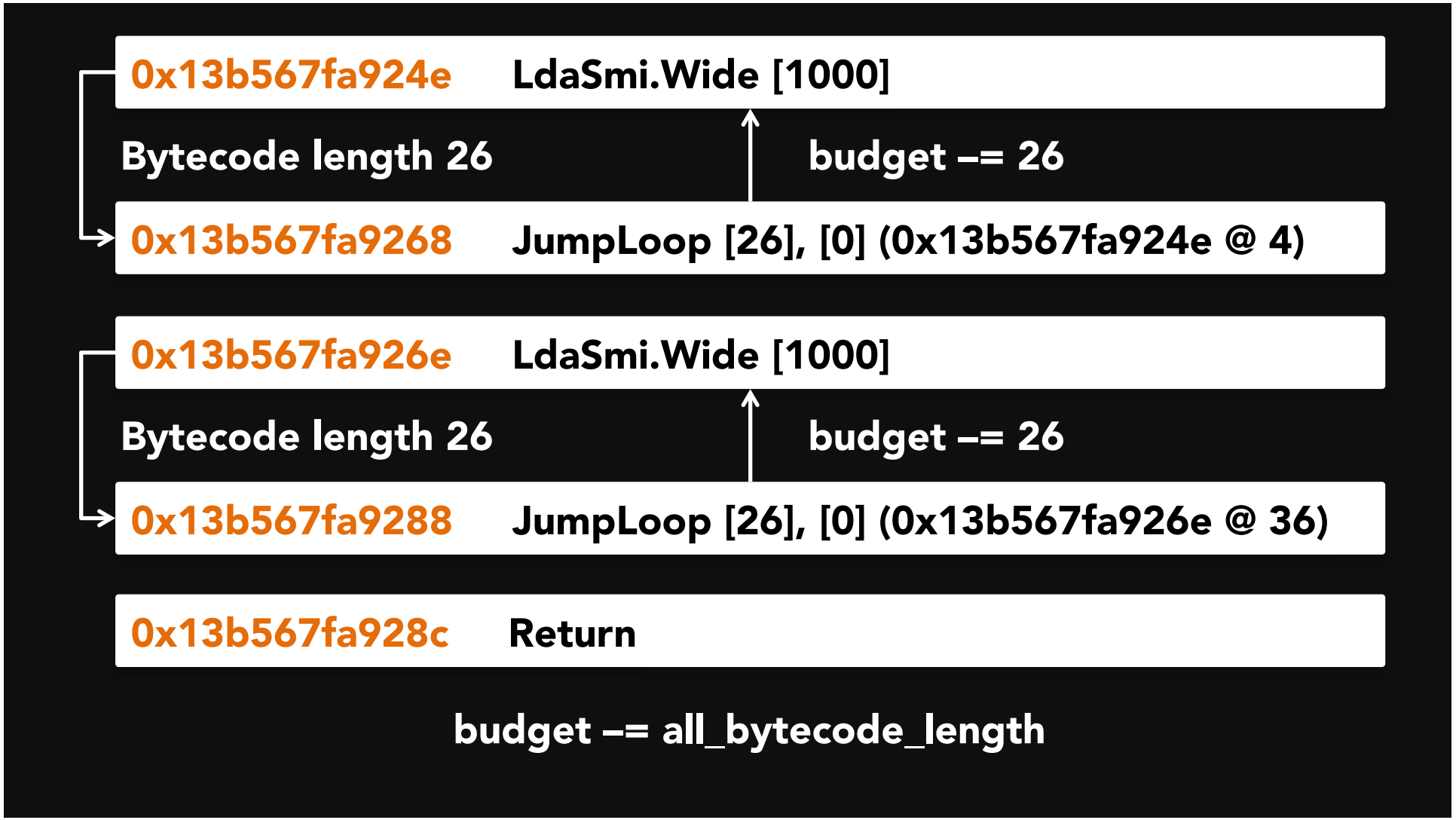
Concurrent Compilation

Function optimized concurrently, so next function call might not optimized.


Hot Function
Bytecode Called
Hot Function(Queued)
Bytecode Called
Hot Function(Queued)
Bytecode Called
Optimized Function
Assembly Called
CompilationQueue
CompilationJob
CompilationJob
CompilationJob

```
const x = x => x;
const y = () => {
  for (let i = 0; i < 1000; i++) {
    x(i);
  }

  for (let i = 0; i < 1000; i++) {
    x(i);
  }
};
y();
```


0x13b567fa924e LdaSmi.Wide [1000]
Bytecode length 26
budget -= 26
0x13b567fa9268 JumpLoop [26], [0] (0x13b567fa924e @ 4)
0x13b567fa926e LdaSmi.Wide [1000]
Bytecode length 26
budget -= 26
0x13b567fa9288 JumpLoop [26], [0] (0x13b567fa926e @ 36)
0x13b567fa928c Return
budget -= all_bytecode_length

Budget for function

Even if loop is splitted, all budget will be checked in Return Bytecode, so it's optimized very well.

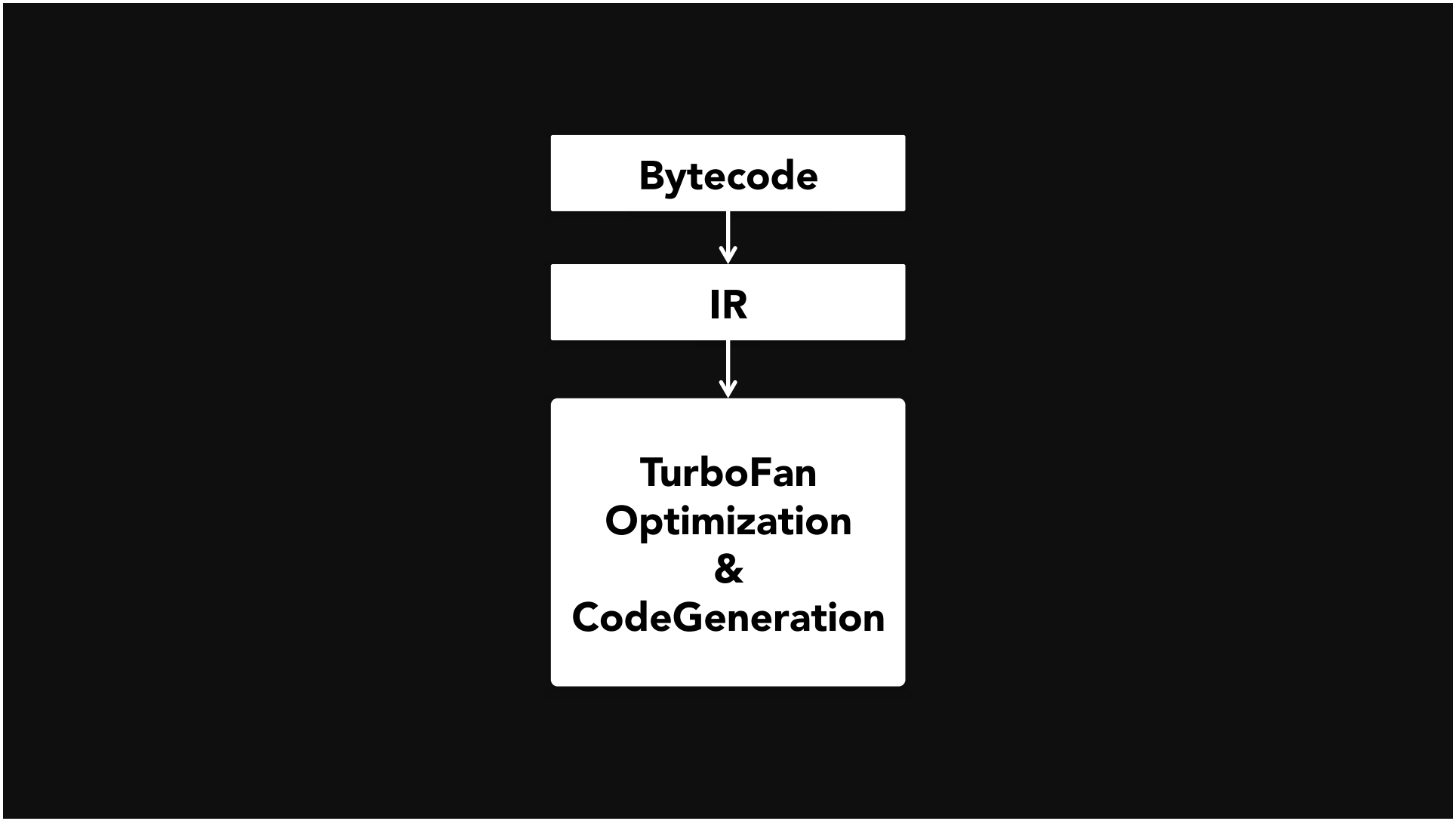

TurboFan

What is TurboFan?

TurboFan is optimization stack of V8.

V8 create IR(Intermediate Representation) from bytecode when optimization.

TurboFan create and optimize that IR.


Bytecode
IR
TurboFan
Optimization
&
CodeGeneration

TurboFan

IR

Abstract execution block.

It's called as Control Flow Graph

```
#22:Branch[None](#21:SpeculativeNumberLessThan, #9:Loop)
#28:IfTrue(#22:Branch)
#30:JSStackCheck(#11:Phi, #32:FrameState,
#21:SpeculativeNumberLessThan, #28:IfTrue)
#33:JSLoadGlobal[0x2f3e1c607881 <String[1]: a>, 1]
(#11:Phi, #34:FrameState, #30:JSStackCheck,
#30:JSStackCheck)
#2:HeapConstant[0x2f3e1c6022e1 <undefined>]()
#39:FrameState
#36:StateValues[sparse:^^](#12:Phi, #33:JSLoadGlobal)
#37:FrameState#35:Checkpoint(#37:FrameState,
#33:JSLoadGlobal, #33:JSLoadGlobal)
#38:JSCall[2, 15256, NULL_OR_UNDEFINED]
(#33:JSLoadGlobal, #2:HeapConstant, #11:Phi,
#39:FrameState, #35:Checkpoint, #33:JSLoadGlobal)
#9:Loop(#0:Start, #38:JSCall)
```

Optimization

TurboFan optimize graph.

Show some optimization.

inline

Inlining function call.

trimming

Remove dead node.

type

Type inference.

typed-lowering

Replace expr to more simple expr depend on type.

loop-peeling

Move independent expr to outside of loop.

loop-exit-elimination

Remove LoopExit.

load-elimination

Remove useless load and checks.

simplified-lowering

Simplify operator by more concrete value.

generic-lowering

Convert js prefixed call to more simple call or stub call.

dead-code-elimination

Remove dead code.

Code generation

Finally, InstructionSelector allocates registers, and CodGenerator generate assembly from IR.

Deoptimization

What is Deoptimization?

Deoptimization mean back to bytecode from machine assembly when unexpected value was passed to assembly code.

Of course less Deoptimization is more better.

Let's see example.

```
const id = x => x;
const test = obj => {
  for (let i = 0; i < 100000; i++) {
    id(obj.x);
  }
};

test({x: 1});
test({x: 1, y: 1});
```

Wrong Map

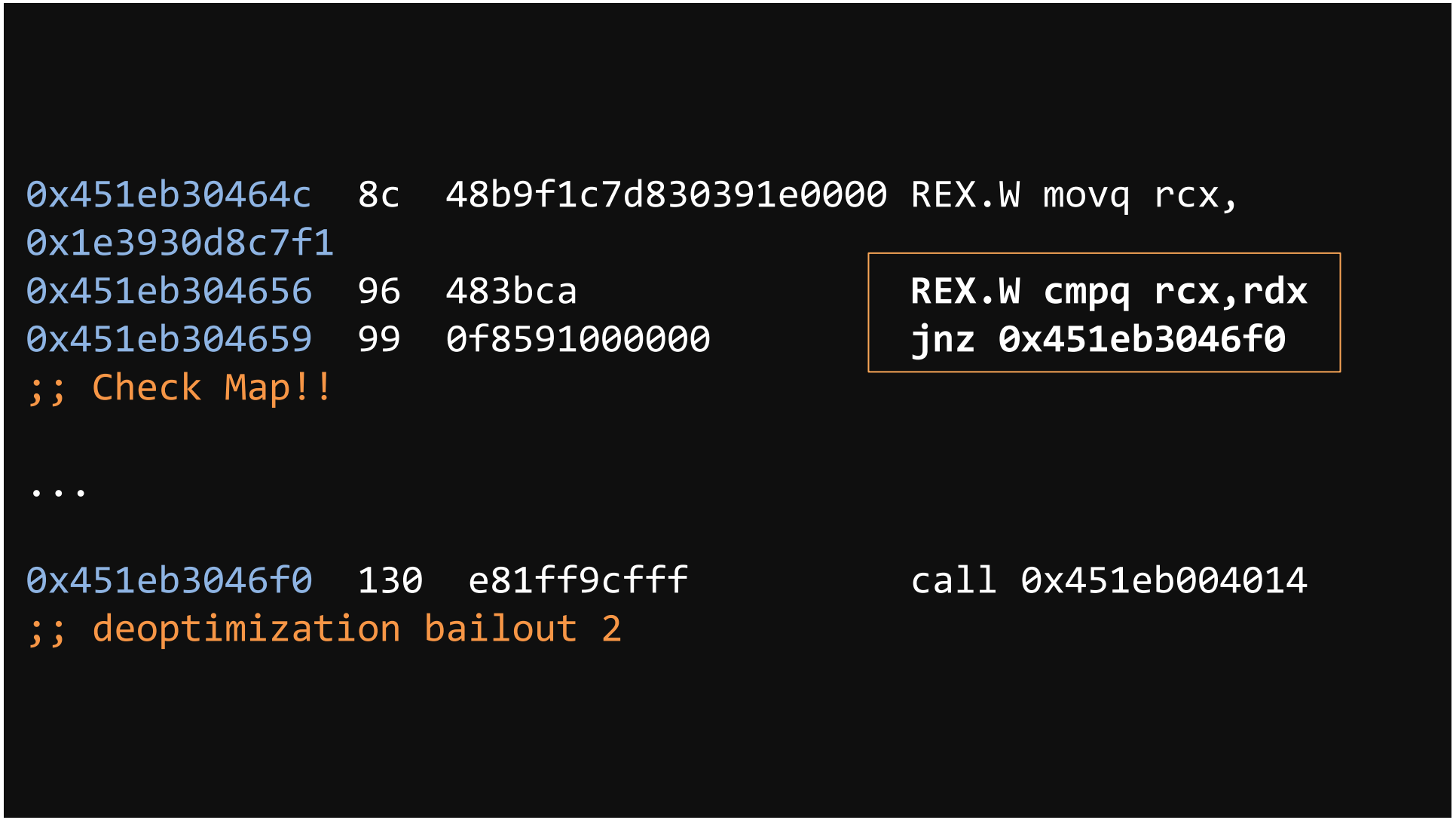
That examples emit optimized assembly for Map of {x},

But second time test function called by Map of {x, y}.

So recompilation occurred.

Let's see assembly code a bit.

Don't be afraid :)


0x451eb30464c 8c 48b9f1c7d830391e0000 REX.W movq rcx,
0x1e3930d8c7f1
0x451eb304656 96 483bca REX.W cmpq rcx,rdx
0x451eb304659 99 0f8591000000 jnz 0x451eb3046f0
;; Check Map!!
...
0x451eb3046f0 130 e81ff9cfff call 0x451eb004014
;; deoptimization bailout 2

Bailout

In this way, emitted code includes Map check code.

When deoptimization is occurred, code backs to bytecodes.

It's called as Bailout.

Summary

This is execution and optimization way of javascript in V8.

Because of time constraints, GC is omitted.

I will write about code reading of V8 to blog.

http://abcdef.gets.b6n.ch/

Thank you for your attention :))