WebAssembly
Adding a new opcode (Turbofan)

PLCT Post-Intern Tech Report

Cao Yuxiang
2021-12-19

Contents of This Report

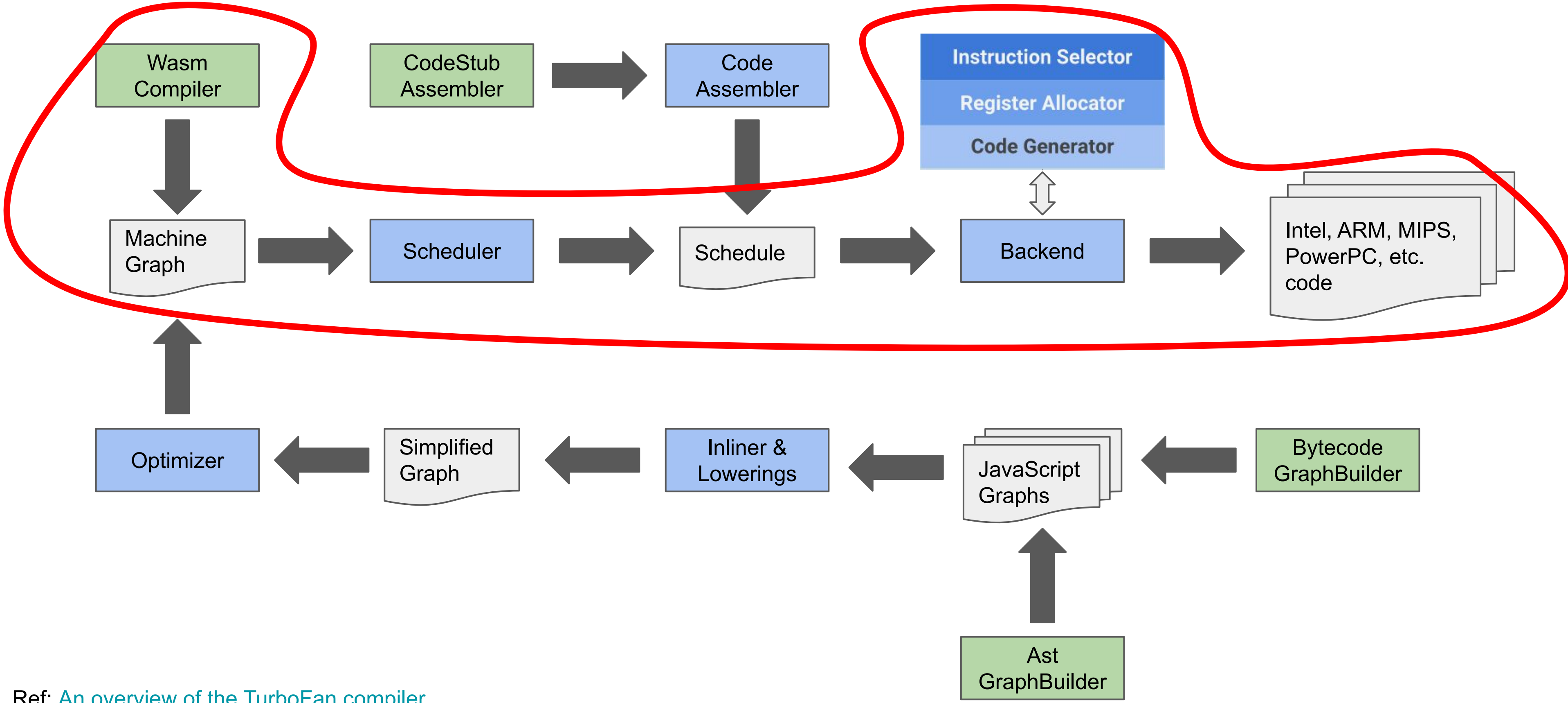
1. Objectives
2. Wasm Compilation Pipeline (TurboFan only)
3. Some notes about tests
4. Some notes about tracing
5. How to add a new opcode

1. Objectives

Through practicing the official guide of Adding a new WebAssembly opcode

1. Learn wasm compilation in code level
2. Learn how to use/update tests to verify new code

2.1 Overview of Wasm Compilation Pipeline



Ref: An overview of the TurboFan compiler

2.2 Wasm Compilation Pipeline Explanation

1. Wasm Compiler: Where byte code of wasm is read and translated to Machine Graph(Sea of Node Intro), which is platform independent IR
2. Schechler: Generate Control Flow and Blocks from Machine Graph for backend as input
3. Backend:
   a. Instruction Selector
   b. Instruction Schechler
   c. Register Allocator
   d. Code Generator

2.3 Wasm Compilation Pipeline Source code

Code branch:

- 9.8-lkgr

Commit id:

- 4f96c8522f166c9fc969c114824adf2292374fea

2.4 Call Hierarchy of compile wasm function

Call Hierarchy from top to where backend functions are triggered

```
CompileWasmFunction (@ src/wasm/wasm-engine.cc:705)
    ExecuteCompilation (where to choose liftoff or Turbofan) (@ src/wasm/function-compiler.cc:160)
        ExecuteFunctionCompilation (@ src/wasm/function-compiler.cc:43)
            ExecuteTurbofanWasmCompilation
            (Turbofan Compilation Entrance)(@ src/wasm/function-compiler.cc:136)
                BuildGraphForWasmFunction (@ src/compiler/wasm-compiler.cc:8155)
                GenerateCodeForWasmFunction (@ src/compiler/wasm-compiler.cc:8175)
                    Run optimization (Graph Reduction) (src/compiler/pipeline.cc:3229-3246)
                    ComputeScheduledGraph (@ src/compiler/pipeline.cc:3257)
                    SelectInstructions (@ src/compiler/pipeline.cc:3260)
                    AssembleCode (@ src/compiler/pipeline.cc:3261)
                    Use CodeGenerator to generate asm (@ src/compiler/pipeline.cc:3264-3275)
                    Use Disassembler to generate de asm (@ src/compiler/pipeline.cc:3282-3291)
                Return
```

2.4 Call Hierarchy of compile wasm function (backend)

Call Hierarchy from where backend functions are triggered to where backend is actually called

```
GenerateCodeForWasmFunction (@ src/compiler/wasm-compiler.cc:8175)
      Run optimization (Graph Reduction) (src/compiler/pipeline.cc:3229-3246)
      ComputeScheduledGraph (@ src/compiler/pipeline.cc:3257)
      SelectInstructions (@ src/compiler/pipeline.cc:3260)
            InstructionSelectionPhase.Run()
                  Create InstructionSelector(defined @ src/compiler/backend/instruction-selector.h)
                  Call InstructionSelector::SelectInstructions() (@ src/compiler/pipeline.cc:2189)
                        Call InstructionScheduler @ src/compiler/backend/instruction-selector.cc:112-115
      AssembleCode (@ src/compiler/pipeline.cc:3261)
            Call InitializeCodeGenerator() (@ pipeline.cc:549)
                  Where CodeGenerator is defined in src/compiler/backend/code-generator.h
      Use CodeGenerator to generate assembly code (@ src/compiler/pipeline.cc:3264-3275)
      Use Disassembler to generate dis-assembly code (@ src/compiler/pipeline.cc:3282-3291)
      Retrun
```

3.1 Wasm direct related tests

Locations:

- test/cctest/wasm
- test/unittests/wasm

3.2 How to run tests

How to run cctest ?

- tools/dev/gm.py x64.debug cctest/{TestFileName}/{TestName}
- python2 ./tools/run-tests.py --outdir=x64.debug cctest/{TestFileName}/{TestName}

How to run unittest ?

- tools/dev/gm.py x64.debug unittests/{TestName}
- python2 ./tools/run-tests.py --outdir=out/x64.debug unittests/{TestName}
- out/x64.debug/unittests --gtest_list_tests --gtest_filter={TestName}

{TestFileName}: The basename of test files under test/cctest folder, must full name

{TestName}: The name of test (Need expand macro), can use Wildcard (case sensitive)

4. Some notes about tracing

For help us to understand the running process of V8 more easily, it’s a good way to use d8’s tracing functionality

Enable tracing by add flag: **--enable-tracing**

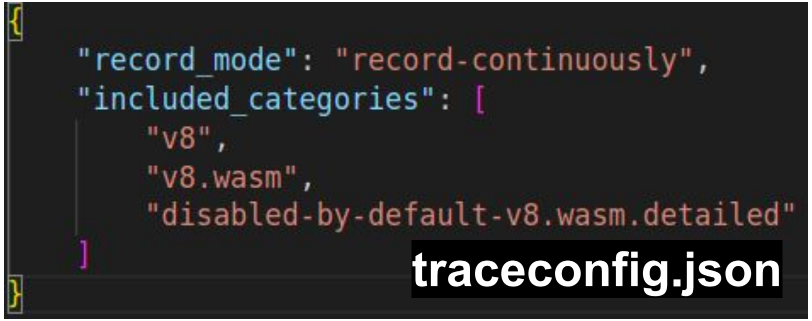
Setup config for tracing:
--trace-config=traceconfig.json

In **traceconfig.json**, use array of string to indicate which part of code you want to trace.

The name string could be found through searching macro in source code of V8:

TRACE_EVENT**

Usually, the category name is first argument



```
{
    "record_mode": "record-continuously",
    "included_categories": [
        "v8",
        "v8.wasm",
        "disabled-by-default-v8.wasm.detailed"
    ]
}
```

traceconfig.json

Example 1:
TRACE_EVENT0("v8.wasm", "wasm.SerializeModule");

For the “v8.wasm” in the above picture.

Example 2:
TRACE_EVENT0(TRACE_DISABLED_BY_DEFAULT("v8.wasm.detailed"),
"wasm.AsyncCompileJob");

For the "disabled-by-default-v8.wasm.detailed" in the above picture

4. Some notes about tracing

Example commands:

```
../v8/out/x64.debug/d8 --noliftoff --single-threaded --enable-tracing
--trace-config=traceconfig.json rust_demo.js
```

It disable the liftoff baseline wasm compiler and ask d8 to running in single thread

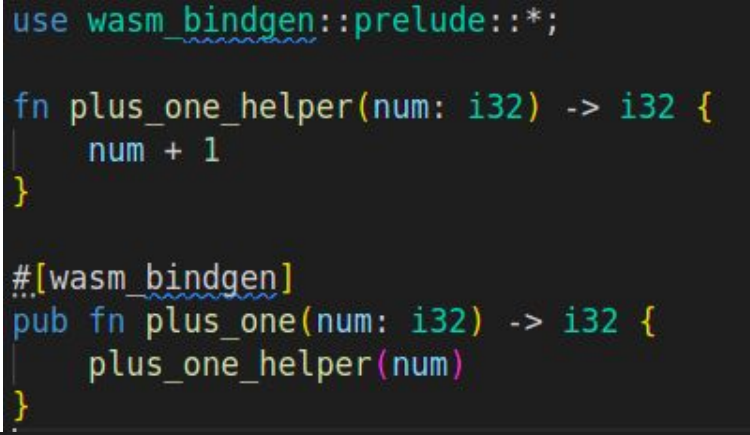
It will generate a v8_trace.json file showing trace of function calls in the v8, which is very helpful

4. Some notes about tracing

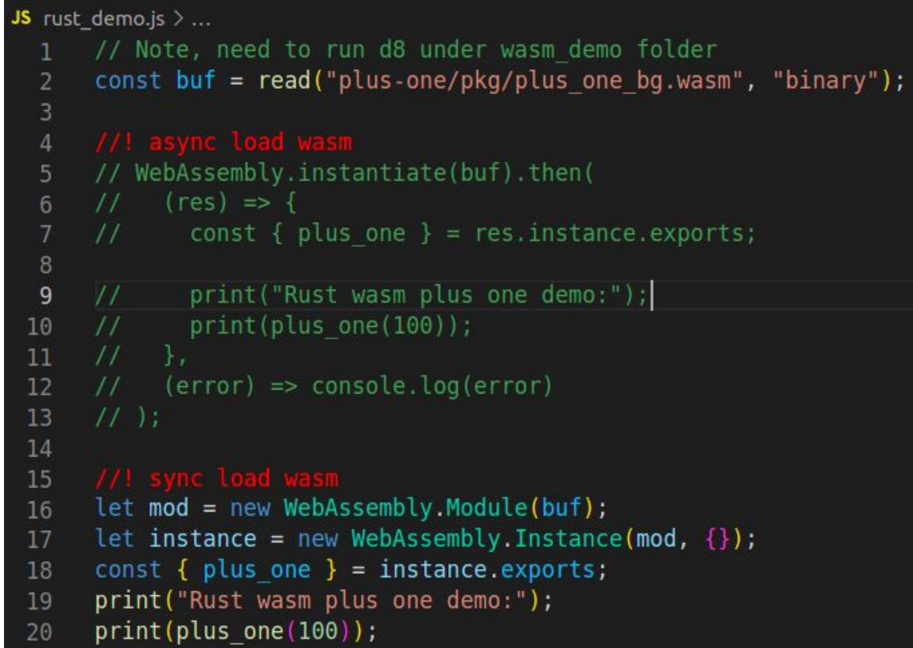
To trace how wasm file is being load and compiled in V8, we also need a demo wasm file for it.

Here I use **rust** with **wasm-pack** to create a simple demo wasm module, learn more here.

Also need to create a js file to load wasm file and call function in it.



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

fn plus_one_helper(num: i32) -> i32 {
    num + 1
}

#[wasm_bindgen]
pub fn plus_one(num: i32) -> i32 {
    plus_one_helper(num)
}
```



```javascript
// Note, need to run d8 under wasm_demo folder
const buf = read("plus-one/pkg/plus_one_bg.wasm", "binary");

//! async load wasm
// WebAssembly.instantiate(buf).then(
//   (res) => {
//     const { plus_one } = res.instance.exports;

//     print("Rust wasm plus one demo:");
//     print(plus_one(100));
//   },
//   (error) => console.log(error)
// );

//! sync load wasm
let mod = new WebAssembly.Module(buf);
let instance = new WebAssembly.Instance(mod, {});
const { plus_one } = instance.exports;
print("Rust wasm plus one demo:");
print(plus_one(100));
```

5. How to add a new opcode

WorkFlow:

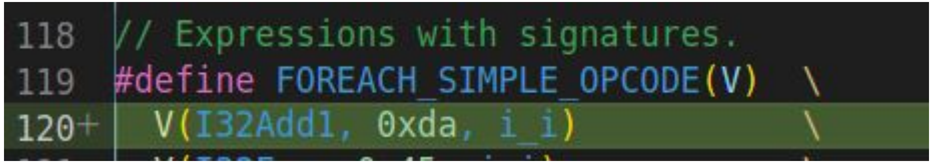
1. Update code
2. Pass compile
3. Pass current tests
4. Add new tests
5. Back to step 1

Reference:

- V8 docs: WebAssembly - adding a new opcode

5.1 Add new opcode definition

Add definition @ src/wasm/wasm-opcodes.h:



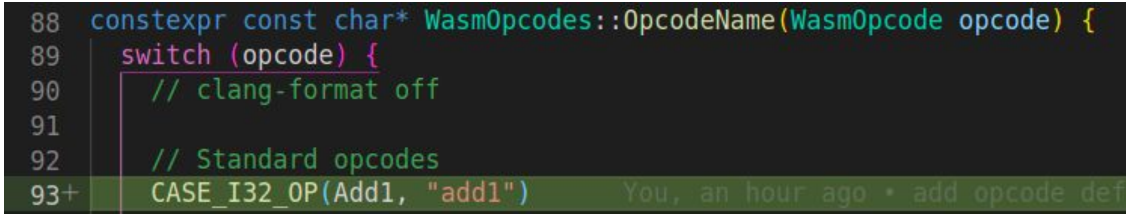
```
// Expressions with signatures.
#define FOREACH_SIMPLE_OPCODE(V)  \
  V(I32Add1, 0xda, i_i)           \
```

Compile and run tests:
$ tools/dev/gm.py x64.debug cctest/test-run-wasm/*

Got compile error:

```
In file included from
../../src/wasm/wasm-opcodes.cc:12:

../../src/wasm/wasm-opcodes-inl.h:89:11: error:
enumeration value 'kExprI32Add1' not handled
in switch [-Werror,-Wswitch]

  switch (opcode) {

          ^~~~~~

1 error generated.
```

According to compile error msg:
We need to add code to for opcode Name getter function



```
constexpr const char* WasmOpcodes::OpcodeName(WasmOpcode opcode) {
  switch (opcode) {
    // clang-format off

    // Standard opcodes
    CASE_I32_OP(Add1, "add1")        You, an hour ago • add opcode def
```

5.1.1 More about Opcode and Wasm Decoder

Actually, all Opcodes are used to create a WasmFullDecoder class in:

src/wasm/function-body-decoder-impl.h

WasmFullDecoder mainly responsible for:

- Decode wasm in unit of function

5.1.2 Call Hierarchy of decode wasm module

Call Hierarchy from top to where opcodes are used

```
WasmEngine::SyncCompile() (@ src/wasm/wasm-engine.cc:536)
      DecodeWasmModule() (@ src/wasm/wasm-engine.cc:544)
            ModuleDecoderImpl decoder  (@ src/wasm/module-decoder.cc:2160)
            decoder.DecodeModule() (@ src/wasm/module-decoder.cc:2168)
                  DecodeSection() (@ src/wasm/module-decoder.cc:1427)
                        DecodeTableSection() (@ src/wasm/module-decoder.cc:469)
                              consume_init_expr() (@ src/wasm/module-decoder.cc:846)
                                    WasmFullDecoder decoder (@ src/wasm/module-decoder.cc:1792)
                                    decoder.DecodeFunctionBody() (@ src/wasm/module-decoder.cc:1799)
                        DecodeGlobalSection() (@ src/wasm/module-decoder.cc:474)
                              consume_init_expr() (@ src/wasm/module-decoder.cc:874)
                                    WasmFullDecoder decoder (@ src/wasm/module-decoder.cc:1792)
                                    decoder.DecodeFunctionBody()  (@ src/wasm/module-decoder.cc:1799)
```

5.1.3 Add new opcode definition - Test again

Run tests again, and we got test error:

```
===
cctest/test-run-wasm/Build_Wasm_SimpleExprs
===

#

# Fatal error in ../../src/compiler/wasm-compiler.cc, line 1339

# Unsupported opcode 0xda:i32.add1

#

#FailureMessage Object: 0x7ffc4b7b2ba0
```

This is because we lack implementation for TurboFan to generate Node for new opcode in Machine Graph (SON)

5.2.1 Create Node in Machine Graph for new opCode Reuse Int32Add()

In src/compiler/wasm-compiler.cc

Code on the blog is outdated, but we can simplify refer to how wasm::kExprI32Add is implemented in WasmGraphBuilder::Binop method:

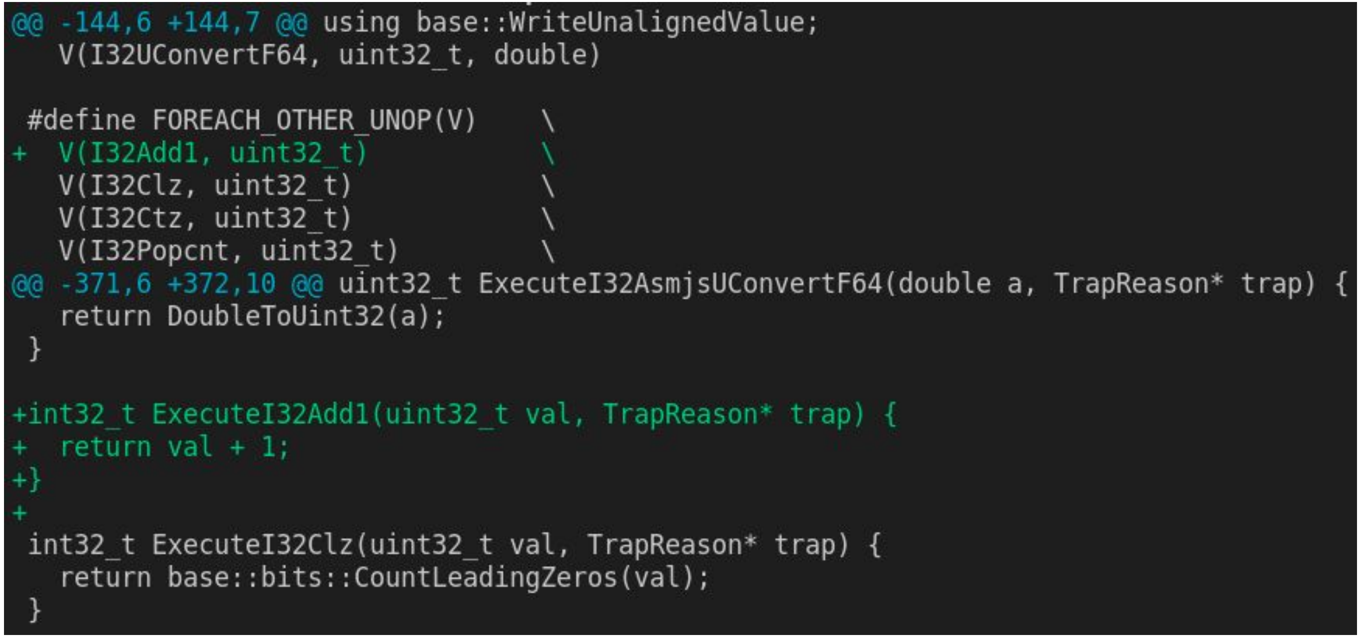
```
switch (opcode) {
   case wasm::kExprI32Add:
     op = m->Int32Add();
     Break;
   . . .
}
return graph()->NewNode(op, left, right);
```

```
switch (opcode) {
   case wasm::kExprI32Add1:
     return graph()->NewNode(m->Int32Add(), input,
Int32Constant(1));
   . . .
}
```

5.2.2 An unsolved problem - cctest use Interpreter in default

tools/dev/gm.py x64.debug cctest/test-run-wasm/RunWasmInterpreter_Int32Add1

cctest default to use interpreter to process wasm, so need to update test/common/wasm/wasm-interpreter.cc:



```
@@ -144,6 +144,7 @@ using base::WriteUnalignedValue;
   V(I32UConvertF64, uint32_t, double)

 #define FOREACH_OTHER_UNOP(V)    \
+  V(I32Add1, uint32_t)           \
   V(I32Clz, uint32_t)            \
   V(I32Ctz, uint32_t)            \
   V(I32Popcnt, uint32_t)         \
@@ -371,6 +372,10 @@ uint32_t ExecuteI32AsmjsUConvertF64(double a, TrapReason* trap) {
   return DoubleToUint32(a);
 }

+int32_t ExecuteI32Add1(uint32_t val, TrapReason* trap) {
+  return val + 1;
+}
+
 int32_t ExecuteI32Clz(uint32_t val, TrapReason* trap) {
   return base::bits::CountLeadingZeros(val);
 }
```

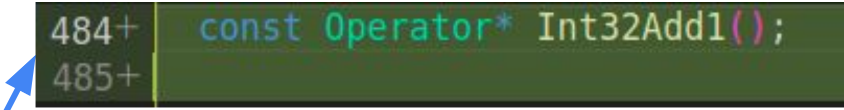
5.2.3 Create Node in Machine Graph for new opCode
Create new TurboFan machine operators

In src/compiler/wasm-compiler.cc:

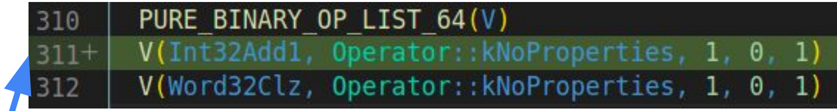
```
switch (opcode) {
   case wasm::kExprI32Add1:
     return graph()->NewNode(m->Int32Add1(), input);
   . . .
}
```

Then need to update these files:

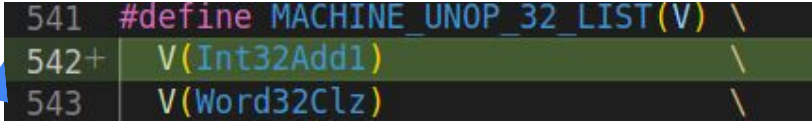
1. **src/compiler/machine-operator.h**
2. **src/compiler/machine-operator.cc**
3. list of opcodes that the machine understands **src/compiler/opcodes.h**
4. verifier **src/compiler/verifier.cc**



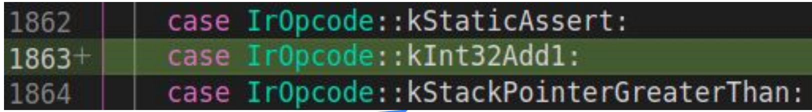
```
  const Operator* Int32Add1();

```



```
PURE_BINARY_OP_LIST_64(V)
V(Int32Add1, Operator::kNoProperties, 1, 0, 1)
V(Word32Clz, Operator::kNoProperties, 1, 0, 1)
```



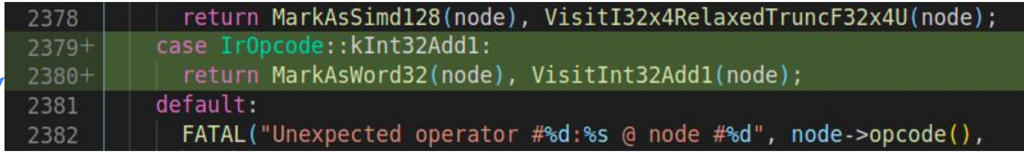
```
#define MACHINE_UNOP_32_LIST(V) \
  V(Int32Add1)                  \
  V(Word32Clz)                  \
```



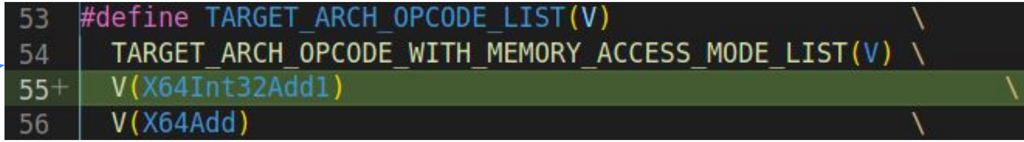
```
    case IrOpcode::kStaticAssert:
    case IrOpcode::kInt32Add1:
    case IrOpcode::kStackPointerGreaterThan:
```

5.3 Update Backend for new opcode

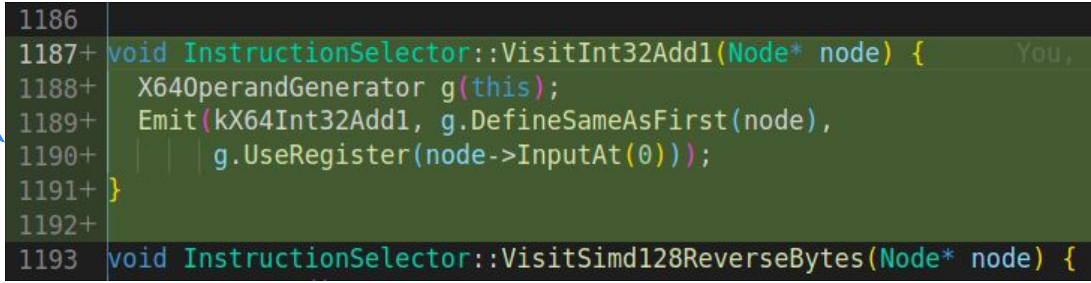
- **Instruction selection (x64 as an example)**
  - src/compiler/backend/instruction-selector.cc
  - src/compiler/backend/x64/instruction-selector-x64.h
  - src/compiler/backend/x64/instruction-selector-x64.cc



```
      return MarkAsSimd128(node), VisitI32x4RelaxedTruncF32x4U(node);
+   case IrOpcode::kInt32Add1:
+     return MarkAsWord32(node), VisitInt32Add1(node);
    default:
      FATAL("Unexpected operator #%d:%s @ node #%d", node->opcode(),
```



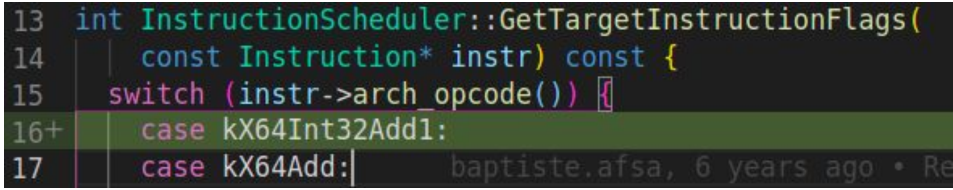
```
 #define TARGET_ARCH_OPCODE_LIST(V)                              \
   TARGET_ARCH_OPCODE_WITH_MEMORY_ACCESS_MODE_LIST(V) \
+  V(X64Int32Add1)                                               \
   V(X64Add)                                                     \
```



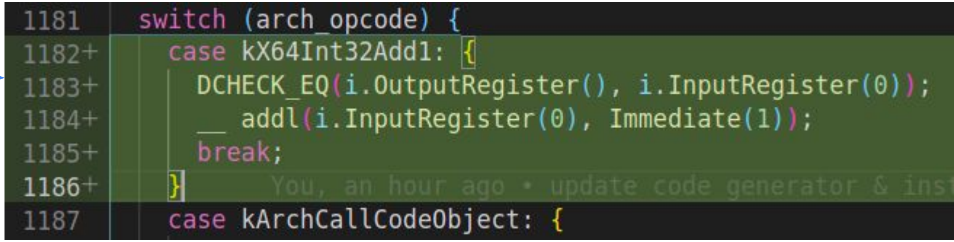
```

+void InstructionSelector::VisitInt32Add1(Node* node) {      You,
+  X64OperandGenerator g(this);
+  Emit(kX64Int32Add1, g.DefineSameAsFirst(node),
+       g.UseRegister(node->InputAt(0)));
+}
+
 void InstructionSelector::VisitSimd128ReverseBytes(Node* node) {
```

5.3 Update Backend for new opcode

- **Instruction scheduling**
  - home/nick/v8/v8/src/compiler/backend/x64/instruction-scheduler-x64.cc
- **Code generation**
  - src/compiler/backend/x64/code-generator-x64.cc
- If the new opcode need a new assembly instruction, we need to add new implementation of new assembly instruction in:
  - src/compiler/backend/x64/assembler-x64.cc



```
int InstructionScheduler::GetTargetInstructionFlags(
    const Instruction* instr) const {
  switch (instr->arch_opcode()) {
    case kX64Int32Add1:
    case kX64Add:       baptiste.afsa, 6 years ago • Re
```



```
  switch (arch_opcode) {
    case kX64Int32Add1: {
      DCHECK_EQ(i.OutputRegister(), i.InputRegister(0));
      __ addl(i.InputRegister(0), Immediate(1));
      break;
    }       You, an hour ago • update code generator & inst
    case kArchCallCodeObject: {
```

Questions?

Thank you