# CSCI 1951Q Final Project: Contributing Changes to Wasmtime Cranelift

*Bisheshank C. Aryal, Edward Wibowo, James Hu*

## Summary

For our final project, we implemented a previously missing optimization in Cranelift, Wasmtime's compiler backend, that simplifies a redundant `select + icmp` instruction pattern in its intermediate representation. Specifically, when a `select` instruction with constant operands is immediately compared against those constants, the pattern can be collapsed into a single boolean operation. Using Cranelift's ISLE domain-specific langauge, we introduce four new rewrite rules that handle equality and inequality comparisons and all constant orderings. Key challenges included ISLE's restriction on terminator instruction simplification, explicit boolean type requirements, and managing similar pattern variations. The resulting solution preserves full semantic correctness, has been validated through comprehensive testing, and is now integrated upstream in Wasmtime.

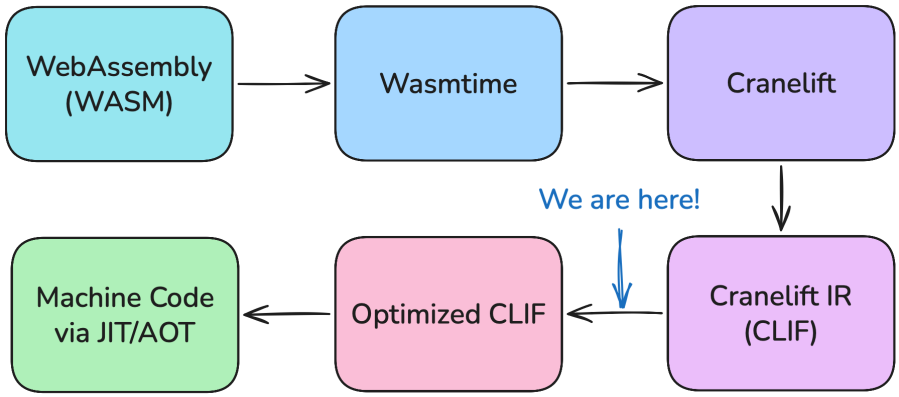


Figure 1: WebAssembly compilation pipeline through Wasmtime showing the optimization target: WASM bytecode is processed by Wasmtime runtime, compiled to Cranelift IR, optimized in ISLE, and lowered to machine code.

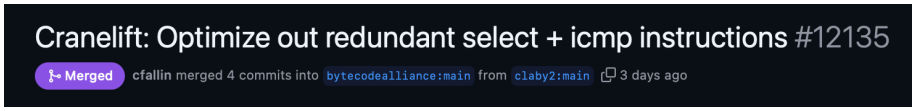


Figure 2: Our optimization was successfully merged into the Wasmtime codebase.

PR: wasmtime/pull/12135.

Original issue: wasmtime/issues/11578

## Introduction

Our final project involves introducing a missing optimization in Cranelift, a compiler backend used by Wasmtime. Wasmtime is a runtime for WebAssembly that leverages Cranelift to generate machine code either at runtime (via JIT) or ahead-of-time.

Cranelift produces optimized machine code from a custom interemediate representation (Cranelift IR). Here is an example of what this IR looks like in textual form:

```
function %non_icmp_inner(i64) -> i8 {
block0(v0: i64):
    v1 = iconst.i64 6
    v3 = iconst.i64 7
    v4 = select v0, v1, v3
    v5 = icmp eq v4, v1
    return v5
}
```

It is interesting to note that this textual form, known as CLIF, is primarily used for testing/debugging purposes. The actual IR used by Wasmtime is represented as an in-memory data structure. Cranelift's goal is to transform this IR into fast, architecture-dependent machine code. Doing so often requires IR-level mid-end optimization passes.

Cranelift uses a DSL called ISLE (Instruction Selection Lowering Expressions) to express both lowering and optimization logic. ISLE consists of *rules* that can be thought of as pattern matching. A unique feature of ISLE is that it is type-aware. Example types are IR `Value`s and architecture-specific `Reg`s that correspond to Rust types. ISLE benefits from common type system wins like preserving invariants via types: tracking "flag" registers in architecture-specific code happens with `ProducesFlags` and `ConsumesFlags` typed values. For this project, we focused on mid-end optimizations that take as input Cranelift IR and produces (optimized) Cranelift IR (rather than machine code).

This project specifically aimed to create mid-end optimizations that were missing in Cranelift as pointed out by wasmtime/issues/11578. This issue points out a potential optimization involving `icmp`, `select`, and `brif` IR instructions.

For reference, `icmp` returns 1 if the comparison is true and 0 otherwise.[1] `select` returns the first argument if its condition is truthy (not `0`) and the second argument otherwise.

Specifically, in the following CLIF code:

```
function %a(i64) -> i64 {
block0(v2: i64):
    v3 = band_imm v2, -562949953421310
    v4 = icmp_imm eq v3, 0
    v5 = iconst.i64 6
    v6 = iconst.i64 7
    v7 = select v4, v6, v5  ; v6 = 7, v5 = 6
    v8 = icmp_imm eq v7, 6
    brif v8, block2, block1

block1:
    v9 = iconst.i64 100
    jump block3(v9)

block2:
```

[1] The function can also return −1 for vector operations, but that is not in scope for this project.

```
    v10 = iconst.i64 101
    jump block3(v10)

block3(v11: i64):
    return v11
}
```

One may observe that the value of `v8` is entirely dependent on the value of `v4`. That is, the value of `v8` depends on whether `v7` is equal to `6`, and the value of `v7` is either equal to `v6 = 7` or `v5 = 6` depending on `v4`. Crucially, since we know `v7` is constrained to either be equal to `6` or `7`, we can deduce that `v8` depends solely on `v4`.

Moreover, a further optimization may optimize away the branching if (`brif`) instruction by leveraging yet another `select` instead. However, while we considered this approach, we chose to focus on the former optimization due to difficulties involving simplifying terminating instructions that may modify the control-flow graph. Our attempts are discussed in the Implementation section.

## Implementation

Our implementation consists of ISLE code. At a high level, we target `icmp eq` (equals comparison) and `icmp ne` (not equals comparison) instructions. We consider the instruction `icmp eq, x, k1`, where `x` is any `Value` and `k1` is a known constant value. If we see the value of `x` comes from the instruction `select y, k1, k2`, we know that these two instructions together are equivalent to the inner condition `y`. That is, `icmp eq, x, k1` is completely determined by `y`, meaning we can avoid a redundant `select`. However, this is only true if `k1 != k2` because otherwise an instruction like `select y, k1, k1` would be optimized by constant propagation.

Prior to reaching the aforementioned implementation, we ran into challenges involving optimizations on `brif`, handling non-boolean `select` inner conditions, and dealing with similar `icmp` setups.

### Challenge 1: The brif Instruction

In CLIF, the `brif` instruction jumps to one of two basic blocks depending on a condition value. The original issue report includes a reproducible test case where the final optimization may affect a `brif` instruction by swapping the order of the target basic blocks based on the structure of the `select` and `icmp`. Our first idea was to target the `brif` instruction in ISLE as the instruction pattern for our optimization. That is, we would find `brif` instructions that are conditioned on `icmp` + `select` instructions and coalesce the blocks into simpler instructions. However, we found that the simplify rules we wrote on `brif` did not propogate through. After some debugging, we discovered the following limitation in the ISLE source code:

```
/// Find the best simplification of the given skeleton instruction, if any,
/// by consulting our `simplify_skeleton` ISLE rules.
fn simplify_skeleton_inst(&mut self, inst: Inst) -> Option<SkeletonInstSimplification> {
    // We cannot currently simplify terminators, or simplify into
    // terminators. Anything that could change the control-flow graph is off
    // limits.
```

Figure 3: Code snippet showing restrictions on ISLE simplifying terminators.

Further details can be found in the corresponding code comments [1]. In summary, it is difficult to simplify terminating instructions (such as `brif`) because they modify the control-flow graph. Modifying the control-flow graph introduces a host of problems. For example, it may change the domination relation between blocks, which may in turn invalidate uses of some variables.

Hence, we avoided optimizations on `brif`. We noticed that any condition that follows the `icmp` + `select` pattern described above may be subject to our optimization: the `brif` instruction is not a required pattern-match condition. Our resulting optimization targets `icmp` on the result of a `select` to simplify to a boolean value without modifying control flow.

### Challenge 2: Handling Non-Boolean `select` Inner Conditions

CLIF's `icmp` instruction has the concept of "truthiness" where 0 is false and anything else is true. Thus, a naive optimization of replacing the `icmp` + `select` combo with the `select`'s inner condition fails in cases such as Listing 1 when the inner condition can be any value, not just `0` or `1`.

```
function %a(i64) -> i8 {
block0(v0: i64):
  v1 = select v0, 100, 0
  v2 = icmp_imm eq v1, 100
  return v2
}
```

Listing 1: Test case that breaks naive optimization. The inner condition is `v0` which can be any 64-bit integer.

A naive optimization rewrites this to Listing 2. However, this transformation loses the "casting" effect of the `icmp` + `select` instruction combo and results in invalid IR. More concretely, the unoptimized `icmp` instruction will "cast" the potentially truthy inner condition of `select` to an `i8` boolean (0 or 1). We account for this by inserting an `icmp` against 0, replicating the "cast". This gives us the correct optimization in Listing 3.

```
function %a(i64) -> i8 {
block0(v0: i64):
  return v0
}
```

Listing 2: Naive optimization results in invalid IR: v0, an `i64`, does not match the return type of `i8`.

```
function %a(i64) -> i8 {
block0(v0: i64):
  v1 = icmp_imm ne v0, 0
  return v1
}
```

Listing 3: Inserting an `icmp_imm ne .. 0` to "cast" the inner condition of `select` into a boolean.

**Challenge 3: Similar `icmp` Setups**

Our first intuition was to focus on matching on instructions of the form `icmp eq, x, k1` where `x = select y, k1, k2`.

However, we quickly noticed that we can apply similar optimizations on

- `icmp eq, x, k2`: which compares `x` with `k2` instead of `k1`.
- `icmp ne, x, k1`: which uses `ne` instead of `eq`.
- `icmp ne, x, k2`: which uses `ne` instead of `eq` *and* compares `x` with `k2` instead of `k1`.

Following careful casework, we determined `icmp eq, x, k2` and `x = icmp ne, y, k1` simplify to the *negation* of the corresponding `select` instruction's condition. We noted that constant propagation automatically pushes constant terms leftwards, meaning analogous cases such as `icmp eq, k1, x` etc. (with `k1` on the left-hand side) are automatically captured.

**Simplify Rules**

In the end, we produced four simplifying rewrite rules to Cranelift. One of the four is listed below:

```
(rule (simplify (eq _
  (select select_ty inner_cond
    (iconst_u _ k1)
    (iconst_u _ k2))
  (iconst_u _ k1)))
  (if-let false (u64_eq k1 k2))
  (ne select_ty inner_cond (iconst_u select_ty 0)))
```

Listing 4: One of the four simplifying rewrite rules we introduced written in ISLE.

Firstly, Listing 4 matches on instructions of the form `icmp eq, x, k1` where `x = select y, k1, k2`. Then, the `if-let` check ensures that `k1 != k2`. Finally, it replaces it with `icmp ne, y, 0`, avoiding a `select` instruction.

## Evaluation

Overall, we implemented four simplifying rewrite rules that optimized `select` + `brif`. To ensure our rules do not break existing optimizations, we developed additional tests that ensure our optimizations propagate correctly and also do not change program semantics.

### Test Results and Validation

We leverage two types of tests:

1. Semantic preservation: we wrote tests that evaluated the optimized and non-optimized versions of functions to ensure they returned the same results.
2. Optimization correctness: we used snapshot testing to ensure our optimization produces the expected IR transformations.

While looking through the Cranelift codebase, we were surprised to see that many existing optimizations are only tested via snapshot testing. We wanted to ensure our changes do not introduce silent bugs, so we were motivated to include semantic preservation tests prior to implementing our optimization.

For example, the following is a semantic preservation test we wrote:

```
test interpret
test run
set opt_level=none
target x86_64
target aarch64
set opt_level=speed

function %non_icmp_inner(i64) -> i8 {
block0(v0: i64):
    v1 = iconst.i64 6
    v3 = iconst.i64 7
    v4 = select v0, v1, v3
    v5 = icmp eq v4, v1
    return v5
}

; run: %non_icmp_inner(0) == 0
; run: %non_icmp_inner(1) == 1
; run: %non_icmp_inner(5) == 1
```

Listing 5: Semantic preservation test that runs the function

This test is run twice, once with optimizations disabled and once with optimizations enabled. The `run` directives ensure that both versions of the function return the same results for various inputs.

The following example is an optimization correctness test:

```
function %a(i64) -> i8 {
block0(v0: i64):
    v1 = iconst.i64 6
    v3 = iconst.i64 7
    v4 = select v0, v1, v3
    v5 = icmp eq v4, v1
    return v5
}

; check: function %a(i64) -> i8 fast {
; check: block0(v0: i64):
; nextln:     v6 = iconst.i64 0
; nextln:     v7 = icmp ne v0, v6  ; v6 = 0
; nextln:     return v7
; nextln: }
```

Listing 6: Optimization correctness test that checks the optimized IR.

This test simply ensures that the optimized IR matches the expected output after applying our optimization.

Some architectures like x86 have optimizations for comparing against `0` vs. other values. [2] However, we have not measured performance impact.

**Result**

We submitted a pull request to the Wasmtime codebase which can be viewed at wasmtime/pull/12135. The pull request passed all CI checks and was eventually merged into the main codebase. As a result, the original issue wasmtime/issues/11578 was closed.

**Future Work**

While our optimization successfully reduces IR size via removing redundant `select` instructions, we left out an additional optimization involving `brif` instructions due to the challenges discussed earlier.

Future work may involve overcoming the challenges of modifying control-flow graph altering instructions in ISLE. This may involve deeper changes as discussed in the original issue thread.

> This one is considerably harder: it requires support for seeing through blockparams during mid-end opts, which has very subtle interactions with the single-pass acyclic nature of our rewrite system; and it requires editing the control-flow graph, which also has complex interactions with the way that the rewrite pass works. At some point we'd like to support this, but it would require pretty deep investment from core Cranelift folks to think it through.
>
> — cfallin

# Bibliography

[1] Bytecode Alliance, “wasmtime: cranelift/codegen/src/egraph.rs.” [Online]. Available: https://github.com/bytecodealliance/wasmtime/blob/85eed831c029e76241a83fa61f04be131f9f14b2/cranelift/codegen/src/egraph.rs#L524

[2] “Intel® 64 and IA-32 Architectures Optimization Reference Manual,” vol. 1, no. 248966–48. Aug. 2023. [Online]. Available: https://www.intel.com/content/www/us/en/content-details/671488/intel-64-and-ia-32-architectures-optimization-reference-manual-volume-1.html