Hardware Intrinsics in WebAssembly

Michael McLoughlin
mcloughlin@cmu.edu

CMU 17770 Final Project Report

Contents

1 Introduction

WebAssembly (Wasm) is designed for hardware-independence and execution with "near native code performance, taking advantage of capabilities common to all contemporary hardware" [8]. This goal has been achievable for *core* Wasm built upon now decades-old instruction sets. However, in the post Moore's Law era we have seen increasing specialization in CPU and instruction set design to achieve performance gains. Modern instruction sets include advanced vector extensions and domain-specific instructions, for example for cryptographic and

machine-learning applications. As Wasm's adoption grows there is demand to expand the "near native" performance goal into domains that require these dedicated instruction sets. At the same time, we would like to do so without sacrificing Wasm's hardware- and platform-independence.

One approach to domain-specific acceleration in Wasm is standards evolution. Wasm is not frozen: the community group is active and the proposals process [11] allows for principled extensions of the standard. For example, Wasm 2.0 added 128-bit SIMD instructions [9] and later proposals allowed for faster performance [7]. However, it is often difficult or impossible to design abstraction layers that allow high-performance on multiple target platforms. Even where possible, the standardization process is slow and deliberative by design, so Wasm users would have to wait a long time to benefit from the latest instruction-set extensions. Finally, instruction set extensions have now driven ISAs to include literally thousands of instructions, and it is unclear that the Wasm standard should ever grow to a similar scale. Therefore, while standards evolution is the right approach for longer-term adoption, it would also be desirable to have a *WebAssembly extension mechanism* that can provide direct access to modern instruction sets on a shorter timescale.

The HW-Specialized WebAssembly proposal [3] suggests an extension mechanism for Wasm to support execution of dedicated hardware instructions and optimized library kernels. Under the proposal, accelerated functionality would be accessed via functions tagged with a custom `@builtin` attribute. For example:

```
(func (@builtin "libssl" "vaes_gcm_setiv")
  (param $ctx i32) (param $iv i32) (param $ivlen i32)
  (result i32)
  ;; function body
)
```

The intention is that the attribute signals to supporting engines that the function should be treated as an *intrinsic*; that is, compiled down to a specific machine instruction or library kernel on supporting platforms. Meanwhile, the function body exists to preserve the hardware-independence of Wasm. The so-called *fallback function* can always be executed as a plain Wasm function on any platform. The proposal also discusses other aspects of the problem: mappings of machine-specific types to Wasm equivalents, checking for the accelerated builtin availability, versioning, and sharing the mappings of built-ins to machine instructions in a templates database.

In this project we explore the practicality of this proposal as an approach to hardware-specialization in Wasm. We do so by showing a proof-of-concept for a selected representative use case, namely the SHA-1 cryptographic algorithm using the Cryptographic Extension on AArch64. We successfully demonstrate that C code written against ARM's C intrinsics API can be executed both natively and via Wasm. We achieve Wasm execution by providing a Wasm AArch64 intrinsics C API layer, together with a fork of the Wasmtime Wasm runtime that supports intrinsic calls for a select group of AArch64 instructions. The end

result is SHA-1 execution performance at only 1.3x native, demonstrating the feasibility of "near native" performance with hardware specialization.

2 Project Setup

The HW-Specialized WebAssembly proposal has many aspects to it. For the purposes of project scoping, we selected a representative motivating example to evaluate whether hardware intrinsics in Wasm could work from a user-interface and engine perspective. Specifically, our target is the SHA-1 cryptographic algorithm implemented with AArch64 Cryptographic Extensions. Cryptographic acceleration is one of the primary motivating examples for hardware intrinsics, especially given the importance of fast correct cryptography in foundational networking applications. The dedicated SHA-1 instruction set in AArch64 also has a modest number of instructions, making it possible to develop a proof-of-concept for a realistic example without the need to support thousands of opcodes.

It would be possible to demonstrate hardware intrinsics at the pure Wasm level, however this would yield a technically correct but underwhelming result. Our goal is to show the practicality of a potentially realistic use case. Therefore, we chose to target an implementation of SHA-1 in C using the Arm C Language Extensions [1], which provides access to Cryptographic Extension instructions through the `arm_neon.h` C header. The goal is an implementation of SHA-1 compiled against the ARM native intrinsics that runs natively *and* could also work—with acceleration—when compiled and executed under a Wasm toolchain with intrinsics support.

2.1 SHA-1 Algorithm

The SHA-1 algorithm hashes an arbitrary-length message to a 160-bit hash value. The core of SHA-1 is a *compression function* that combines an incoming 160-bit state with a 512-bit message chunk to produce a new 160-bit state. The performance of SHA-1 boils down to the efficiency of the compression function implementation. The compression function operates on five 32-bit registers A_i, B_i, C_i, D_i, E_i over 80 rounds, each of which consumes a 32-bit word W_i from the *message schedule*. First, the message schedule is computed by initializing the first 16 words $W_0, \ldots, W_{15}$ to the 32-bit big-endian words of the input 512-bit message and then computing:

$$W_i = (W_{i-3} \oplus W_{i-8} \oplus W_{i-14} \oplus W_{i-16}) \lll 1,$$

for $i = 16, \ldots, 79$, where $\oplus$ is bitwise exclusive or and $\lll$ is bitwise left-rotate. The round update function is:

$$
\begin{aligned}
A_{i+1} &= (A_i \lll 5) + F_i(B_i, C_i, D_i) + E_i + K_i + W_i \\
B_{i+1} &= A_i \\
C_{i+1} &= B_i \lll 30 \\
D_{i+1} &= C_i \\
E_{i+1} &= D_i,
\end{aligned}
$$

where F_i is a bitwise function that may be choose, parity or majority depending on the round, and K_i are round-dependent constants. The final state is obtained by adding final register values into the original input state.

At a high-level, SHA-1 is a sequence of pure bitwise and arithmetic operations. It is simple to implement but inherently serial and difficult to vectorize a single computation (multiple independent SHA-1 computations are trivially vectorizable). Therefore, it benefits substantially from custom hardware acceleration.

2.2 AArch64 SHA-1 Instructions

AArch64 Cryptographic Extensions offer a family of size instructions for SHA-1 acceleration:

- `SHA1C Qd,Sn,Vm.4S`: computes *four* rounds of the SHA-1 compression function with registers A, B, C, D in a 128-bit vector register `Qd`, E in the low 32-bits of a vector register `Sn`, and inputs words W_i in `Vm`. The bitwise function is choose, from the `C` mnemonic.
- `SHA1P Qd,Sn,Vm.4S`: as `SHA1C` with parity bitwise function.
- `SHA1M Qd,Sn,Vm.4S`: as `SHA1C` with majority bitwise function.
- `SHA1H Sd,Sn`: computes the SHA-1 fixed rotate by 30 (applied to B) on the low 32-bits of a vector register '`Sn`'. Note that AArch64 has a left-rotate instruction on the general-purpose register file, but the `SHA1H` instruction avoids the expensive move between register files.
- `SHA1SU0 Vd.4S,Vn.4S,Vm.4S`: computes part of the message schedule, called "SHA1 schedule update 0". Operates on three segments of message words in vector registers.
- `SHA1SU1 Vd.4S,Vn.4S`: completes the messages schedule, called called "SHA1 schedule update 1".

These instructions are available from C with the `arm_neon.h` functions [2]:

```
uint32x4_t vsha1cq_u32(uint32x4_t hash_abcd, uint32_t hash_e,
↪  uint32x4_t wk);
uint32x4_t vsha1pq_u32(uint32x4_t hash_abcd, uint32_t hash_e,
↪  uint32x4_t wk);
```

```
uint32x4_t vsha1mq_u32(uint32x4_t hash_abcd, uint32_t hash_e,
↪  uint32x4_t wk);
uint32_t vsha1h_u32(uint32_t hash_e);
uint32x4_t vsha1su0q_u32(uint32x4_t w0_3, uint32x4_t w4_7,
↪  uint32x4_t w8_11);
uint32x4_t vsha1su1q_u32(uint32x4_t tw0_3, uint32x4_t w12_15);
```

These instructions are guaranteed by compiler toolchains to compile down to the corresponding assembly instructions. Note that the C interface introduces subtle changes from the underlying instructions for usability reasons: for example, the `hash_e` argument is a regular 32-bit unsigned type in C but belongs in a vector register.

2.3 Optimization Target

Our target for the purposes of the project is an implementation of SHA-1 in C using the intrinsics interface above. To give a feel for the implementation, four rounds of the compression function are:

```
// Rounds 28-31
e0 = vsha1h_u32(vgetq_lane_u32(abcd, 0));
abcd = vsha1pq_u32(abcd, e1, t1);
t1 = vaddq_u32(m1, vdupq_n_u32(K1));
m2 = vsha1su1q_u32(m2, m1);
m3 = vsha1su0q_u32(m3, m0, m1);
```

Note that since the specialist instructions deal with vector registers, we also have to use other utility intrinsic functions, namely:

```
uint32x4_t vld1q_u32(uint32_t const *ptr);
void vst1q_u32(uint32_t *ptr, uint32x4_t val);
uint32x4_t vaddq_u32(uint32x4_t a, uint32x4_t b);
uint32x4_t vdupq_n_u32(uint32_t value);
uint8x16_t vrev32q_u8(uint8x16_t vec);
uint32_t vgetq_lane_u32(uint32x4_t v, const int lane);
uint32x4_t vreinterpretq_u32_u8(uint8x16_t a);
uint8x16_t vreinterpretq_u8_u32(uint32x4_t a);
```

Alongside the intrinsics-optimized version we also have an implementation of SHA-1 using plain C for comparison. For evaluation purposes we also have: a unit test against a SHA-1 test vector to ensure correctness, and a microbenchmark to evaluate performance.

The implementation was developed and tested first on a AArch64 native platform (Apple M1 with Clang toolchain). The goal is to provide a corresponding toolchain for Wasm that allows the same C implementation to execute essentially unchanged.

3 Intrinsics in WebAssembly

Our approach for AArch64 intrinsics in Wasm requires two main components:

- *C Intrinsics Layer.* This layer bridges ARM's C intrinsics API to actual intrinsics that will be executable in a Wasm engine. In addition, we also want to provide the capability to execute the same program with pure Wasm fallbacks. To achieve this we provide a `wasm_arm_neon.h` header and compatibility implementation `wasm_arm_neon.c`.

- *Engine Integration.* Secondly, the target engine must support intrinsics compilation. That is, it should be able to intercept calls to defined intrinsic functions and JIT compile them to corresponding instructions.

These layers are tightly coupled, since the C layer and engine have to agree on the underlying function calls that will be treated specially. Throughout the project, we learned that design choices at this interface are critical to the runtime performance of the generated Wasm. We will see this by presenting the evolution of the implementation, starting with the baseline approach before discussing the refinements that were necessary to reach near-native performance.

3.1 Baseline C Intrinsics Layer

The first approach to C-to-Wasm mapping was to treat every ARM C function directly as an intrinsic in the engine as well. The `wasm_arm_neon.h` header contains declarations matching all required C intrinsics (Section 2). Then `wasm_arm_neon.c` contains pure-Wasm fallback implementations, for example for the `vsha1cq_u32` call:

```
uint32x4_t vsha1cq_u32(uint32x4_t hash_abcd, uint32_t hash_e,
↪  uint32x4_t wk) {
    uint32_t a = wasm_u32x4_extract_lane(hash_abcd, 0);
    uint32_t b = wasm_u32x4_extract_lane(hash_abcd, 1);
    uint32_t c = wasm_u32x4_extract_lane(hash_abcd, 2);
    uint32_t d = wasm_u32x4_extract_lane(hash_abcd, 3);
    uint32_t e = hash_e;

    SHA1_ROUND(SHA1_CHOOSE, 0);
    SHA1_ROUND(SHA1_CHOOSE, 1);
    SHA1_ROUND(SHA1_CHOOSE, 2);
    SHA1_ROUND(SHA1_CHOOSE, 3);

    return wasm_u32x4_make(a, b, c, d);
}
```

Given this layer, we are able to compile both the SHA-1 intrinsics-backed implementation and the `wasm_arm_neon` library to `.wasm`. The Wasm binaries link and execute correctly, and pass the SHA-1 correctness test. This provides intrinsics fallback implementations by compilation from C.

This fallback provides the benefit of platform compatibility, but the performance is poor at 9.1x times slower than the native executable. In fact, it is even 2.2x slower than the plain C version compiled to Wasm, likely because of

function calls and conversions between scalar and packed-vector representations of the SHA-1 register words.

3.2 Engine Integration

The real benefits of hardware intrinsics come from JIT support in the Wasm engine. For this project we chose to work with Wasmtime [10], a production-grade Wasm runtime backed by the Cranelift optimizing JIT compiler. This was selected on the basis of its existing high-quality code generation for the AArch64 backend, and to demonstrate feasibility in a realistic production-grade compiler.

Wasmtime engine integration for each intrinsic requires:

1. *Assembler Support.* Cranelift has an integrated custom assembler, which has grown to support instructions as needed. Therefore it does not support the SHA-1 intrinsics out of the box.

2. *Register Allocation Metadata.* The Cranelift register allocator requires use/def metadata about instruction operands to compute liveness information.

3. *New IR Instructions.* Cranelift is an optimizing compiler with its own IR, known as "CLIF". Threading through intrinsic calls from Wasmtime requires routing through the IR, therefore each new instruction needs its a corresponding CLIF instruction. Since CLIF already had precedent for adding backend-specific instructions such as `x86_pshufb`, we added new instructions with the `aarch64` prefix.

4. *Instruction Lowering.* Cranelift implements IR-to-machine code lowering in a domain-specific language called ISLE [4]. ISLE defines pattern matching rules on IR instruction sequences and determines the corresponding machine code that will be emitted. Each new instruction required a new lowering rule. In the case of intrinsics these rules are deliberately simple, typically just passing through the dedicated IR instruction to the assembler format. However, one complexity here is type transformations required from CLIF types to the target machine instruction.

5. *Call Interception.* Finally, the remaining piece is interception of intrinsic function calls to emit the corresponding CLIF IR instruction. This part of the integration is arguably the most fragile. Functions are intercepted based on their names. Names need not always be present, but if they are will be present in the Name Custom Section (defined in the WebAssembly Specification Appendix). The interception itself is simply a pattern match on the name in Wasmtime's Wasm-to-CLIF translation logic, emitting CLIF IR for each.

Note that intrinsic C function handling fits in two categories:

- *"True" Intrinsics.* The SHA-1 C intrinsics themselves, such as `vsha1cq_u32`, are implemented mostly as a passthrough from the intrinsic function call to machine instruction, via CLIF IR.

- *Emulated.* Many of the general vector intrinsics already have exact equivalents in CLIF, and in fact Wasm too. For example `vaddq_u32` is a CLIF `iadd` on the vector `I32x4` type. Likewise, `vgetq_lane_u32` is a CLIF `extractlane` instruction.

We added support for 12 intrinsics to Wasmtime: the SHA-1 instructions as "true" intrinsics, and 6 general helpers lowered to existing CLIF instructions.

One problem presented by the SHA-1 instructions in particular was a mismatch between their C API and the target machine instructions. The C intrinsics API takes 32-bit arguments for the SHA-1 E register, but the target instructions expect them to be in the low 32-bits of a vector register. Wasm and CLIF do not have a way to express this: 32-bit integers are expected to live in general purpose registers, not vector registers. As such, the ISLE lowering rules must emit moves between the general purpose and vector register files to account for the mismatch.

3.3 Interim Performance Results

Performance of the SHA-1 benchmark compared to native improved rapidly as each intrinsic was implemented in the runtime, as the following table shows.

vs. Native	Change (including prior)
7.6x	Intrinsic: `vsha1cq_u32`
5.1x	Intrinsics: `vsha1{p,m}q_u32`
5.0x	Intrinsic: `vsha1h_u32`
5.1x	Intrinsic: `vsha1su0q_u32`
3.6x	Intrinsic: `vsha1su1q_u32`
3.8x	Intrinsic: `vaddq_u32`
2.7x	Intrinsic: `vdupq_n_u32`
3.1x	Intrinsic: `vgetq_lane_u32`
2.9x	Intrinsics: `vreinterpretq_{u32_u8,u8_u32}`
3.2x	Intrinsic: `vrev32q_u8`

However, despite implementing almost all intrinsics (excluding load/store operations), the end result still isn't close to our "near native" goal. Inspecting generated code suggested a few reasons why this might be, and ultimately motivated a refinement of the intrinsics interface with substantially better performance.

Inspection suggested the following possible issues with the generated code:

- *Redundant Moves Between Register Classes.* Moves between the general-purpose and vector register files that were required in the lowering phase are persisted to the generated code. Unlike full optimizing compilers such

as GCC and Clang, optimizing JIT compilers like Cranelift prioritize compile speed and accordingly do not have backend-optimization passes that would be required to elide these moves. Moves between register classes are known to cause significant slowdowns in CPU pipelines.

- *Memory Operations.* We did not implement Wasmtime support for the memory intrinsics `vld1q_u32` and `vst1q_u32` for vector load-store. When Clang compiles these intrinsics it uses a memory base offset read from a special global variable. It was not immediately clear how to correctly lower the intrinsics without making assumptions about the Clang memory behavior. This is likely resolvable, but it was more delicate than other intrinsics that perform pure computation. As a result, the generated code still contains function calls for load/store operations, which will no doubt slow down reading the SHA-1 message to be hashed. However, adhoc experiments on this aspect suggested the performance degradation is real but not the predominant effect.

- *Instruction Scheduling.* Again, due to compile speed design considerations in optimizing JIT compilers, Cranelift does not have an instruction scheduling pass. Cryptographic kernel code such as SHA-1 can have performance highly dependent on instruction ordering, therefore the lack of an instruction scheduling pass likely explains some of the difference from native.

The general theme behind these problems is that an optimizing JIT compiler has, by design, a more limited set of optimization passes. Therefore, it is unable to fix up sub-optimal code resulting from semantics mismatches between the C intrinsics API and the underlying target instructions. This motivates changes to the C layer, in order to provide a cleaner interface with the Wasm engine.

3.4 Refined Intrinsics Interface

So far, we had treated the C API as identical to the intrinsics API the engine would intercept. However, this need not be the case. We achieved significant performance gains by doing more at the `wasm_arm_neon` layer. Firstly, by bridging the the C intrinsic API to a different engine intrinsic API, and secondly inlining intrinsic functions that have direct Wasm equivalents.

Bridging to Engine Intrinsics API The redundant moves problem stems from a semantics mismatch between the C API and the corresponding instruction. For example, `vsha1h_u32` takes a `uint32_t hash_e` argument but should compile to `SHA1H Sd,Sn`, where `Sn` is the low bits of a vector register. If the intrinsics call that reaches the Wasm engine persists this mismatch, then we have seen that an optimizing JIT is unlikely to have sufficient passes to fix it. But, what if we arranged for the intrinsic call that reaches the engine to already have a 128-bit parameter type for `hash_e`?

We achieve this by implementing an additional layer, so the intrinsic understood by the engine can be different. Specifically, in the case of the `SHA1H` instruction:

```
uint32x4_t __intrinsic_vsha1h_u32(uint32x4_t hash_e);

static inline uint32_t vsha1h_u32(uint32_t hash_e) {
    return wasm_u32x4_extract_lane(__intrinsic_vsha1h_u32(
↪   wasm_u32x4_splat(hash_e)),
↪   0);
}
```

Here, the `__intrinsic_vsha1h_u32` function is now the one that will be intercepted by the engine, and its API is the same as the assembly instruction.

Notably, the redundant 32-to-128-bit moves still exist, but they now exist as inlined Wasm operators in the C source code (`i32x4.splat` and `i32x4.extract_lane`). Clang is more likely to be able to optimize away these moves as an AOT compiler than a JIT compiler would be.

Inlining Pure Wasm Intrinsics We noted in Section 3.2 that some instructions are implemented as passthrough to machine instructions, but others can be translated to CLIF instructions instead. For example, `vaddq_u32` can be translated to a CLIF `iadd` on the `I32x4` vector type. We could even translate it at a layer before by implementing it with Wasm `i32x4.add`. If a C intrinsic can be implemented efficiently with pure Wasm operators, we could handle it entirely in the C layer, obviating any need to process the intrinsic in the engine.

Therefore, the next change was to implement selected intrinsics as `inline` calls in the header where possible:

```
static inline uint32x4_t vaddq_u32(uint32x4_t a, uint32x4_t b) {
    return wasm_i32x4_add(a, b);
}
```

Again, this has the benefit of improving the results from the AOT compilation since it sees more pure Wasm operators rather than opaque intrinsic calls. Intrinsic calls are now limited to the cases that need them.

Performance Gains from Refined Interface The combination of these changes brought massive improvements, bringing us to 1.3x of native performance.

vs. Native	Change (including prior)
2.5x	Bridging to Engine Intrinsics API
1.3x	Inlining Pure Wasm Intrinsics

We believe the reasons for these improvements are:

- *Improved AOT Compilation.* Inlined pure Wasm implementations offer more optimization opportunities at the C compiler level. In addition, the shims between general purpose and vector types are more likely to be elided by an AOT compiler with a full suite of optimization passes.
- *Fidelity of Engine Intrinsics APIs.* With an extra translation layer, the intrinsics handled by the engine have a closer semantic match to the target hardware instruction.

4 Results

The previous section on implementation (Section 3) presented the results of different techniques. The table below shows the end result compared to alternative SHA-1 implementations and execution strategies.

Implementation	Execution	vs. Native Intrinsics
Intrinsics	Native	1.0x
Intrinsics	Wasmtime *with Intrinsics*	1.3x
Plain C	Native	2.3x
Plain C	Wasmtime Baseline	4.3x
Intrinsics (Fallbacks)	Wasmtime Baseline	9.4x

Overall, these results demonstrate feasibility of the approach. Achieving 1.3x native execution qualifies for our "near native" performance goal.

However, intrinsics fallback performance is disappointing, since it underperforms a simple generic SHA-1 compiled to Wasm by a large margin. The reason for this slowdown is likely the repeated Wasm function calls in place of plain arithmetic/bitwise operations of the Plain C version. If fallbacks performance was important, inlining would be a good approach to try. In addition, the availability check functions proposed by HW-Specialized WebAssembly would allow redirection to a separate non-intrinsics version if necessary. Despite disappointing performance, fallbacks still successfully serve the goal of preserving Wasm hardware-independence.

5 Discussion

In this section we discuss some lessons from this proof-of-concept.

Challenge of Semantics Mismatches Compilation via intrinsics passes through many layers: C intrinsics API, engine intrinsics API, Wasm operators, CLIF IR and machine code representation. Each of these has their own semantics and value representations. The example of 32-bit integer types in Wasm/CLIF causing redundant register moves showed that minor mismatches can have significant performance impacts. These details are important when

the entire goal of hardware intrinsics in Wasm is reaching near-native performance. This project demonstrates just one use case, and we might hope that the idiosyncracies of the SHA-1 instruction set are not widespread. However, it seems likely that this broader problem of semantics mismatches could rear its head in other cases, for example when attempting to use wide vector types (e.g. Intel AVX-512) that do not have Wasm equivalents.

Significance of the Intrinsics API The switch to a refined intrinsics interface at the C layer (Section 3.4) produced massive performance gains. This suggests that near-native intrinsics performance requires:

1. Implementing C instrincs as inlined Wasm operators wherever possible. This allows for improved code quality from the AOT compiler, and reduces the work required by the engine.

2. Engine intrinsics API is as close as possible to machine instructions. This makes the engine work essentially a passthrough, and limits the optimizations required from its JIT.

Making this distinction for a small group of 12 instructions needed for the proof of concept was easy enough. It is interesting to think about how you might achieve this if the goal was to support hundreds or thousands of opcodes. For example, is there automation that could detect the cases that can be efficiency mapped to existing Wasm operators?

Importance of Accompanying Optimizations The underwhelming initial results (Section 3.3) show that merely mapping to the right machine instructions is not enough. Supporting optimization passes are critical. In this case, it was crucial to eliminate redundant moves between register classes, but it is reasonable to expect instances of this problem for other classes intrinsics. Optimizing JITs are designed for compile speed and therefore have a much more limited set of optimizations than a full AOT compiler. In this case we were able to work around missing Cranelift JIT optimizations by moving the problem to the AOT compilation layer, however it is not clear that would always be possible. Indeed, the remaining approximately 30% overhead over native execution may be a difficult gap to close, given the lack of optimizations such as instruction scheduling in JIT compilers. Overall, we might expect that Wasm intrinsics performance would be limited by JIT compiler optimization capabilities.

Engineering Aspects The fork of Wasmtime for this project was modified with this proof-of-concept in mind. While the engineering was reasonable, the approach taken is not one that would scale to adding hundreds or thousands of intrinsic calls. At the time of writing, the ARM intrinsics database contains 12,855 function calls, with 4,344 in the Neon instruction set extension. A full production-grade version of the `wasm_arm_neon` library and accompanying engine support would be a substantial undertaking. You would almost certainly

want automation and code-generation involved (discussed in Future Work, Section 7), but also certain parts of the integration method in Section 3.2 would not scale well. The current hand-written assembler would need to support many more instructions. You also probably would not want to actually extend the CLIF IR to support every intrinsic either, but instead perhaps support an explicit passthrough or intrinsic IR node that would effectively perform a trivial lowering to a wrapped machine instruction. None of these engineering challenges are intractable, but they would need careful thought.

6 Work in Progress: Auto-Derived Fallbacks

In this proof-of-concept, fallback functions were produced by compiling hand-written C functions. However, this would not scale well to thousands of intrinsics, which presents the question of where the fallback functions would come from, and how we could be assured of their correctness. Modern instruction sets are expansive, so hand-written fallbacks would be tedious and error-prone to produce. At the same time, vendors such as ARM now distribute machine-readable specifications [6] of their instruction set semantics, and recent research [5] has helped make them slightly more practical to use.

An initial goal of this project was to explore the question of whether we can automatically derive fallback function definitions from the instruction set specifications.

We have made partial progress towards this goal, however not sufficient to get an end-to-end example working. Specifically, we have implemented:

- Parser for the AST output of the ASLp partial evaluator for the machine-readable specification.
- Initial IR format and translator from AST into IR.
- Forked ASLp to support SHA-1 opcodes. These were not initially supported due to unimplemented syntactic constructs that appear in their semantics. In addition, additional optimization rules were required to make the output consumable.

This work could be continued in future (Section 7). However, early indications suggested that this approach may be more challenging than a simple one pass translator from the specification's semantics language to Wasm. In addition, the SHA-1 examples performed particularly badly in the ASLp tool's symbolic evaluator. Since they perform four rounds of SHA-1 they naturally produce deep nested expressions. Achieving clean results would likely require several fixup passes in the translator.

Given these challenges, and initial fruitful progress with engine integration, the focus of the project shifted in that direction. The work-in-progress is nonetheless is included in the artifact for this project (Section A)

7 Further Work

Avenues for future work:

- *First-class `@builtin` Support.* The most fragile part of the engine integration (Section 3.2) is intrinsic call interception, and its reliance on the custom name section. This was reasonable from the point of view of a point-of-concept, but a real implementation would want to add and handle the `@builtin` attribute specially.
- *Fallbacks Performance.* Our results show that the performance of Wasm fallbacks was very poor (Section 4), even compared to a Plain C version. While fallbacks are still valuable for portability, it would be desirable to avoid egregious performance penalties. One fruitful avenue here might be inlining fallbacks where possible.
- *Coverage.* This project focussed on one example of interest and identified some challenges (Section 5). However, we cannot know how generalizable these lessons are without trying other classes of intrinsics. It would be worth picking other representitive use cases to explore the space: for example, wide vector types like AVX-512 and narrow floating point used in Machine Learning acceleration.
- *Other aspects of HW-Specialized WebAssembly Proposal.* This proof-of-concept did not explore custom types or the idea of a shared intrinsics lowering database, both of which would be interesting to pursue.
- *Fallback Auto-generation.* In this proof-of-concept, fallback functions were produced by compiling hand-written C. It would be valuable to continue the work-in-progress (Section 6) to auto-generate fallback functions from vendor-provided specifications.
- *Automation.* The approach taken so far for engine integration was manageable at low scale, but not for the many thousands of intrinsics available in AArch64. ARM does distribute machine-readable databases of their C intrinsics API in JSON, together with the full semantics of their ISA. While these databases are famously cumbersome to work with, it does raise the prospect of automating large parts of the engine integration. It would be interesting to see how far we could push the automated approach.
- *Engineering.* As discussed in Section 5, the engineering aspects of integrating potentially thousands of intrinsics into a JIT engine would need careful consideration.
- *Verification.* Can we prove that the fallback definitions are semantically equivalent to their machine-code counterparts? Authoritative vendor provided specifications potentially make this a tractable problem.

- *Portable API Derivation.* A more nebulous related research question concerns the challenge of designing standardized portable instruction sets that enable performant access to multiple target architectures. Wasm has achieved this for its core instruction set, abstracting over base ISAs of X86, AArch64, RISC-V, and more. To some extent the fixed SIMD instructions achieve this goal, though relaxed SIMD proposal shows the gaps. In general finding a portable abstraction layer over multiple ISA extensions is a difficult problem. Could automation or synthesis be applied to this problem: given specifications for two or more instruction sets, can you discover a portable ISA over them that minimizes the performance gap?

Acknowledgements

Thank you to Ben Titzer and Arjun Ramesh for instruction and advice in the CMU 17770 class. Thanks also to Chris Fallin and Andrew Brown for advice on integrating intrinsics in the Cranelift engine.

A Artifacts

At the time of writing, not all artifact repositories are public. Access has been provided to class instructors.

Hardware Wasm Hardware intrinsics experimental repository: SHA-1 implementation and evaluation tools (Section 2), results data, work-in-progress fallbacks generator (Section 6), and this report.

- Repository: `github.com/mmcloughlin/hwwasm`
- Version: `769ce9b891b1da2151af7d320001114798444c9a`

Wasmtime with Intrinsics Wasmtime fork with hardware intrinsics support.

- Repository: `github.com/mmcloughlin/hwwasmtime`
- Fork Commits: `v27.0.0...hwwasm`

ASLp Fork ASLp fork with support for symbolic evaluation of SHA-1 instruction specifications. This work contributed to the incomplete fallback function generator (Section 6).

- Repository: `github.com/mmcloughlin/aslp/tree/hwwasm`

References

[1] *Arm C Language Extensions*. URL: https://arm-software.github.io/acle/.

[2] *Arm Neon Intrinsics Reference: SHA1*. URL: https://arm-software.github.io/acle/neon_intrinsics/advsimd.html#sha1.

[3] Andrew Brown. *HW-Specialized WebAssembly*. URL: https://github.com/WebAssembly/design/issues/1528.

[4] *ISLE: Instruction Selection Lowering Expressions*. URL: https://github.com/bytecodealliance/wasmtime/blob/main/cranelift/isle/docs/language-reference.md.

[5] Kait Lam and Nicholas Coughlin. "Lift-off: Trustworthy ARMv8 semantics from formal specifications". In: *Formal Methods in Computer-Aided Design, FMCAD 2023, Ames, IA, USA, October 24-27, 2023*. Ed. by Alexander Nadel and Kristin Yvonne Rozier. IEEE, 2023, pp. 274–283. DOI: 10.34727/2023/ISBN.978-3-85448-060-0_36. URL: https://doi.org/10.34727/2023/isbn.978-3-85448-060-0%5C_36.

[6] Alastair Reid. "Trustworthy specifications of ARM® v8-A and v8-M system level architecture". In: *2016 Formal Methods in Computer-Aided Design (FMCAD)*. 2016, pp. 161–168. DOI: 10.1109/FMCAD.2016.7886675.

[7] *Relaxed SIMD proposal for WebAssembly*. URL: https://github.com/WebAssembly/relaxed-simd.

[8] *WebAssembly Core Specification*. Version 2.0. https://webassembly.github.io/spec/core/_download/WebAssembly.pdf. W3C, Apr. 19, 2022. URL: https://www.w3.org/TR/wasm-core-2/.

[9] *SIMD proposal for WebAssembly*. URL: https://github.com/WebAssembly/simd.

[10] *Wasmtime: A fast and secure runtime for WebAssembly*. URL: https://wasmtime.dev.

[11] *WebAssembly Proposals*. URL: https://github.com/WebAssembly/proposals.