Interval Arithmetic for WASM

WASM’s Identity Crisis

- What started as a browser accelerator aspires to become intermediate representation for trustless distributed computing (e.g. Golem, Internet Computer, etc.).
- Distributed computing needs digital security.
- But it also needs analog security to be practical in the real world.
- WASM needs to deliver both if it wants to succeed long term as “the world’s assembly language”.

Analog Security

- Floating-point contains no uncertainty information.
- It’s trivially easy to implement logic that arrives at the same grossly wrong answer, even in double-precision.
- People have died and rockets have exploded due to such issues.
- Analog security means having upper and lower limits on a result.

But Why WASM?

- All this can be done in WASM byte code.
- ...an order-of-magnitude more slowly.
- Chicken-and-egg: HLL is nonperformant so it's unpopular. LLL support is essentially nonexistent because of the latter.
- The buck stops with WASM. Build it, and the optimized HLL libraries will come. With a bit of luck, hardware vendors may follow at some point. If not, performance still improves.

Proposed Ground Rules

- Only provide primitives (which actually might be implemented in hardware later). No actual interval instructions.
- Nothing left undefined which could facilitate side channels or verification failures (even though integer divide-by-zero is undefined).
- Identical results on any platform.

Rounding Fundamentals #1

- Interval arithmetic requires rounding: (1) toward negative infinity, (2) toward positive infinity, (3) inward toward zero, and (4) outward away from zero.
- Mode 4 is unsupported in hardware, so we either need to emulate it or provide primitives for doing so more efficiently. We'll discuss that later.
- Sign bit (2 states) is not the same as arithmetic sign (3 states). Worst case is (-0.0).

Rounding Fundamentals #2

- The consensus seems to be that we need 4 flavors of each instruction. But instead of Mode 4, we should support nearest-or-even, which is the current implied mode (Mode 0).
- Therefore, absent Mode 4 emulation, we must offer easy sign bit access, which is nontrivial because it treats a float as an int.
- But this isn’t enough because certain transcendental functions have error terms in excess of what rounding can provide.

ULP Operations #1

- No practical implementation is feasible without dealing with units-of-the-last-place (ULPs).
- Computing the ULP is an int operation applied to a float. This type casting involves grotesque manipulations in an HLL.
- There are caveats with infinities, zeroes, and denormals, all of which require int manipulation and branching or conditional movement.
- Taylor series require “nudging” limits on results by a ULP after rounding (which can change ULP!).

ULP Operations #2

- Some transcendental operations have complicated error terms with their own interval considerations. These could be approximated by a float times ULP, although it would reduce accuracy in exchange for a modest time savings.
- A ULP or half ULP is often the threshold for stopping Taylor series computation. These values actually have different and expensive corner cases and are both worth accelerating.

ULP Operations #3

- The programmer could just add or subtract the (half) ULP manually under the appropriate rounding mode.
- But this will occur all over the place and eat user resources to store the ULP.
- So better to offer fire-and-forget nudge primitives which do this implicitly and efficiently.

Proposed Instructions #1

- 1. 4 rounding flavors of all existing instructions where it matters. Hardware support seems comprehensive on nVidia today, and is going to waste.
- 2. LoadULP. Return ULP of float with same sign.
- 3. LoadHalfULP. Same thing, but different expensive corner cases.
- 4. NudgeInward. Use ULP to push toward zero.
- 5. NudgeOutward. Opposite of #4.

Proposed Instructions #2

- 6. NudgePositive. Add |ULP|.
- 7. NudgeNegative. Subtract |ULP|.
- 8. GetSignBit. From float to int[0] or int[0:7].
- 9. SetSignBit. From (int[0] or int[0:7])!=0 to float.
- 10. CopySignBit. From float to float.
- 11. XorSignBits. From float to float.
- Instructions 8-11 are less efficient than direct branching on various sign conditions.

Proposed Instructions #3

- 12. BranchIfSignZero. Refers to sign bit, not arithmetic sign. Hint that it will be taken.
- 13. BranchIfSignOne. Hint not taken.
- 14. BranchIfSignsEqual. Hint taken.
- 15. BranchIfSignsUnequal. Hint not taken.
- Somehow or other, we need to confront the requirement for different rounding policies, or other behavior, based on sign.

Issue Link

- github.com/WebAssembly/design/issues/1384