Formally Specifying the Rust Core

Aniket Mishra

[2025-12-20 Sat]

Outline

Acknowledgements

I am a BTech student in IIT Gandhinagar, currently in my 3rd year of study.

Acknowledgements

I am a BTech student in IIT Gandhinagar, currently in my 3rd year of study. This work was done during a remote internship with CRYSPEN, a company based in France and Germany that deals with the development of high assurance software.

Acknowledgements

I am a BTech student in IIT Gandhinagar, currently in my 3rd year of study. This work was done during a remote internship with CRYSPEN, a company based in France and Germany that deals with the development of high assurance software. I worked primarily with Karthikeyan Bhargavan (Chief Research Scientist) and Maxime Buyse (Proof and Tool Engineer).

SIMD and Vector Intrinsics

SIMD stands for Single Instruction, Multiple Data. Vector intrinsics are special functions offered by the compiler, that allow the use of SIMD instructions.

SIMD in Rust

The core::arch crate (the Rust equivalent of a C library) is responsible for exposing these vendor-specific intrinsics that typically correspond to a single machine instruction.

```
/// Simple program squares elements in 16 bit chunks.
use core::arch::x86_64::*;
fn main() {
    unsafe {
        let a : __m256i = _mm256_set_epi16
            (0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15);
        let b : __m256i = _mm256_mullo_epi16(a, a);
        println!("{}",_mm256_extract_epi16::<0>(b));
        println!("{}",_mm256_extract_epi16::<15>(b));
    }
}
```

Importance

Rust has been finding popular use in new security-critical and performance-sensitive projects, and SIMD vector intrinsics are often used for performance/efficiency in such projects.

- Dalek: Elliptic Curve Cryptography
- Hashbrown
- Libcrux

Rust Docs

Let us imagine ourselves as a Rust developer looking through these intrinsics. Let us say we want to understand the intrinsic "vabdq_s8", in core::arch::aarch64, used for performing an absolute difference.

Rust Docs

Let us imagine ourselves as a Rust developer looking through these intrinsics. Let us say we want to understand the intrinsic "vabdq_s8", in core::arch::aarch64, used for performing an absolute difference. Since we are Rust developers, our first instinct is to look at the Rust docs. A quick google search leads us to this.

Rust Docs

Let us imagine ourselves as a Rust developer looking through these intrinsics. Let us say we want to understand the intrinsic "vabdq_s8", in core::arch::aarch64, used for performing an absolute difference. Since we are Rust developers, our first instinct is to look at the Rust docs. A quick google search leads us to this.

ARM Docs

Not a great start. The Rust documentation barely explains anything, all it does is link to the ARM documentation. But atleast the ARM docs are nice! There is some pseudocode to accompany it too.

ARM Docs

Not a great start. The Rust documentation barely explains anything, all it does is link to the ARM documentation. But atleast the ARM docs are nice! There is some pseudocode to accompany it too.

Documentation String

Signed Absolute Difference. This instruction subtracts the elements of the vector of the second source SIMD&FP register from the corresponding elements of the first source SIMD&FP register, places the absolute values of the results into a vector, and writes the vector to the destination SIMD&FP register.

panic!()

Everything seems fine, but when we run the tests, they fail!

panic!()

Everything seems fine, but when we run the tests, they fail! After hours of debugging, in what feels like a miracle, we find the issue.

panic!()

Everything seems fine, but when we run the tests, they fail! After hours of debugging, in what feels like a miracle, we find the issue.

```
use core::arch::aarch64::*;
fn main() {
    unsafe {
            let a = vld1q_s8(&(127 as i8) as *const i8);
            let b = vld1q_s8(&(-2 as i8) as *const i8);

            let result = vabdq_s8(a, b);
            let result =
             *(&result as *const int8x16_t
              as *const [i8; 16]);
            println!("{}", result[0])
    }
}
```

A Bug in the Rust Source????

A Bug in the Rust Source????

```
pub fn vabdq_s8(a: int8x16_t, b: int8x16_t) -> int8x16_t {
    unsafe extern "unadjusted" {
        ...
        #[cfg_attr(target_arch = "arm",
          link_name = "llvm.arm.neon.vabds.v16i8")]
        fn _vabdq_s8(a: int8x16_t, b: int8x16_t)
                     -> int8x16_t;
    }
    unsafe { _vabdq_s8(a, b) }
}
```

A Bug in the Rust Source????

```
pub fn vabdq_s8(a: int8x16_t, b: int8x16_t) -> int8x16_t {
    unsafe extern "unadjusted" {
        ...
        #[cfg_attr(target_arch = "arm",
          link_name = "llvm.arm.neon.vabds.v16i8")]
        fn _vabdq_s8(a: int8x16_t, b: int8x16_t)
                     -> int8x16_t;
    }
    unsafe { _vabdq_s8(a, b) }
}
```

The Rust code seems to use FFI to make a call using LLVM IR! So there's no way the Rust Source is wrong.

A Bug in the Rust Source????

```
pub fn vabdq_s8(a: int8x16_t, b: int8x16_t) -> int8x16_t {
    unsafe extern "unadjusted" {
        ...
        #[cfg_attr(target_arch = "arm",
          link_name = "llvm.arm.neon.vabds.v16i8")]
        fn _vabdq_s8(a: int8x16_t, b: int8x16_t)
                     -> int8x16_t;
    }
    unsafe { _vabdq_s8(a, b) }
}
```

The Rust code seems to use FFI to make a call using LLVM IR! So there's no way the Rust Source is wrong. Frustrated with the hours we spent on this, we sit back and sigh, being done for the day.

The Moral of the Story

Essentially, bad documentation means that even the best Rust programmers need to wade through Intel or ARM assembly documentation (and more!) to understand these intrinsics.

The Moral of the Story

Essentially, bad documentation means that even the best Rust programmers need to wade through Intel or ARM assembly documentation (and more!) to understand these intrinsics. This is especially bad for us verifiers!

The Moral of the Story

Essentially, bad documentation means that even the best Rust programmers need to wade through Intel or ARM assembly documentation (and more!) to understand these intrinsics. This is especially bad for us verifiers!

- HACL-STAR (F*)
- Jasmin (Easy Crypt)
- s2n-bignum (HOL Light)

The Problem

Challenge #15 of the Verify Rust Std puts forth this issue. The problem statement reads as follows:

The Problem

Challenge #15 of the Verify Rust Std puts forth this issue. The problem statement reads as follows:

Challenge 15

Consequently, we believe there is a strong need for a consistent, formal, testable specification of the SIMD intrinsics that can aid Rust developers. Furthermore, we believe that this specification should written in a way that can be used to aid formal verification of Rust programs using various proof assistants.

Rewrite it .. in Rust!

Rewrite it .. in Rust!

Our solution involves writing the models of the intrinsics as pure, functional Rust code that matches the input/output behaviour of the intrinsics. This comes with a few benefits.

Rewrite it .. in Rust!

Our solution involves writing the models of the intrinsics as pure, functional Rust code that matches the input/output behaviour of the intrinsics. This comes with a few benefits.

- Our solution is naturally proof assistant-agnostic.

Rewrite it .. in Rust!

Our solution involves writing the models of the intrinsics as pure, functional Rust code that matches the input/output behaviour of the intrinsics. This comes with a few benefits.

- Our solution is naturally proof assistant-agnostic.
- Anybody interested in using our models, can use the equipment they have developed for modelling and verifying Rust programs in *general*.

Rewrite it .. in Rust!

Our solution involves writing the models of the intrinsics as pure, functional Rust code that matches the input/output behaviour of the intrinsics. This comes with a few benefits.

- Our solution is naturally proof assistant-agnostic.
- Anybody interested in using our models, can use the equipment they have developed for modelling and verifying Rust programs in *general*.
- It allows us to easily test our models against the actual intrinsics.

Rewrite it .. in Rust!

Our solution involves writing the models of the intrinsics as pure, functional Rust code that matches the input/output behaviour of the intrinsics. This comes with a few benefits.

- Our solution is naturally proof assistant-agnostic.
- Anybody interested in using our models, can use the equipment they have developed for modelling and verifying Rust programs in *general*.
- It allows us to easily test our models against the actual intrinsics.

Model Generation

Modelling by hand is not scalable.

Model Generation

Modelling by hand is not scalable. Fortunately, we can use the Rust core itself!

Model Generation

Modelling by hand is not scalable. Fortunately, we can use the Rust core itself! Within the Rust core, we (essentially) find 3 kinds of intrinsics.

- Built-in
- External
- Defined

Built-In Intrinsics

```rust
/// Adds two simd vectors elementwise.
///
/// `T` must be a vector of integers or floats.
#[rustc_intrinsic]
#[rustc_nounwind]
pub unsafe fn simd_add<T>(x: T, y: T) -> T;

/// Subtracts `rhs` from `lhs` elementwise.
///
/// `T` must be a vector of integers or floats.
#[rustc_intrinsic]
#[rustc_nounwind]
pub unsafe fn simd_sub<T>(lhs: T, rhs: T) -> T;
```

External Intrinsics

```
#[allow(improper_ctypes)]
unsafe extern "C" {
    #[link_name = "llvm.x86.avx2.phadd.sw"]
    fn phaddsw(a: i16x16, b: i16x16) -> i16x16;
    #[link_name = "llvm.x86.avx2.phsub.sw"]
    fn phsubsw(a: i16x16, b: i16x16) -> i16x16;
    #[link_name = "llvm.x86.avx2.pmadd.ub.sw"]
    fn pmaddubsw(a: u8x32, b: i8x32) -> i16x16;
    #[link_name = "llvm.x86.avx2.mpsadbw"]
    fn mpsadbw(a: u8x32, b: u8x32, imm8: i8) -> u16x16;
```

Defined Intrinsics

```rust
pub const fn _mm256_hsub_epi32(a: __m256i, b: __m256i) -> __m256i {
    let a = a.as_i32x8();
    let b = b.as_i32x8();
    unsafe {
        let even: i32x8 = simd_shuffle!(a, b, [0, 2, 8, 10, 4, 6, 12, 14]);
        let odd: i32x8 = simd_shuffle!(a, b, [1, 3, 9, 11, 5, 7, 13, 15]);
        simd_sub(even, odd).as_m256i()
    }
}
```

```rust
pub fn _mm256_hsubs_epi16(a: __m256i, b: __m256i) -> __m256i {
    unsafe { transmute(phsubsw(a.as_i16x16(), b.as_i16x16())) }
}
```

Hand-Written Models: Built-In

```rust
pub fn simd_insert<const N: u32, T: Copy>(x: FunArray<N, T>, idx: u32, val: T) -> FunArray<N, T> {
    FunArray::from_fn(|i| if i == idx { val } else { x[i] })
}
```

```rust
pub fn simd_extract<const N: u32, T: Clone>(x: FunArray<N, T>, idx: u32) -> T {
    x.get(idx).clone()
}
```

```rust
pub fn simd_add<const N: u32, T: MachineInteger + Copy>(
    x: FunArray<N, T>,
    y: FunArray<N, T>,
) -> FunArray<N, T> {
    FunArray::from_fn(|i| x[i].wrapping_add(y[i]))
}
```

Hand-Written Models: External

```rust
use crate::abstractions::{bit::MachineInteger, simd::*};
pub fn phaddw(a: i16x16, b: i16x16) -> i16x16 {
    i16x16::from_fn(|i| {
        if i < 4 {
            a[2 * i].wrapping_add(a[2 * i + 1])
        } else if i < 8 {
            b[2 * (i - 4)].wrapping_add(b[2 * (i - 4) + 1])
        } else if i < 12 {
            a[2 * (i - 4)].wrapping_add(a[2 * (i - 4) + 1])
        } else {
            b[2 * (i - 8)].wrapping_add(b[2 * (i - 8) + 1])
        }
    })
}
```

Generated Models: Defined

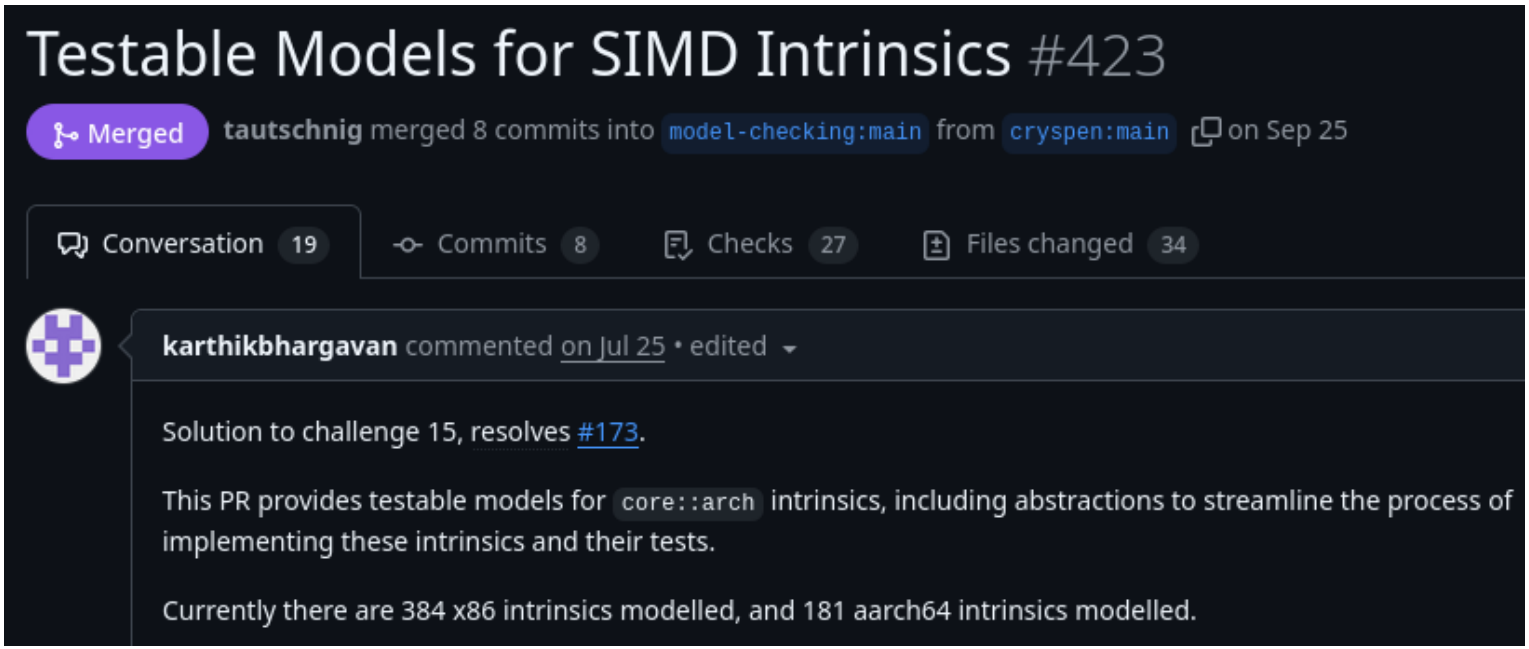
```
pub fn _mm256_abs_epi32(a: __m256i) -> __m256i {
    {
        let a = a.as_i32x8();
        let r = simd_select(simd_lt(a, i32x8::ZERO()), simd_neg(a), a);
        transmute(r)
    }
}
```

```
pub fn _mm256_hadd_epi16(a: __m256i, b: __m256i) -> __m256i {
    {
        transmute(phaddw(a.as_i16x16(), b.as_i16x16()))
    }
}
```

Testing!

```
mk!([100]_mm256_srli_epi16{<0>,<1>,<2>,<3>,<4>,<5>,<6>
mk!([100]_mm256_srli_epi32{<0>,<1>,<2>,<3>,<4>,<5>,<6>
mk!([100]_mm256_srli_epi64{<0>,<1>,<2>,<3>,<4>,<5>,<6>
mk!(_mm_srlv_epi32(a: BitVec, count: BitVec));
mk!(_mm256_srlv_epi32(a: BitVec, count: BitVec));
```

Acceptance of Solution by AWS



Testable Models for SIMD Intrinsics #423

Merged **tautschnig** merged 8 commits into `model-checking:main` from `cryspen:main` on Sep 25

Conversation 19 | Commits 8 | Checks 27 | Files changed 34

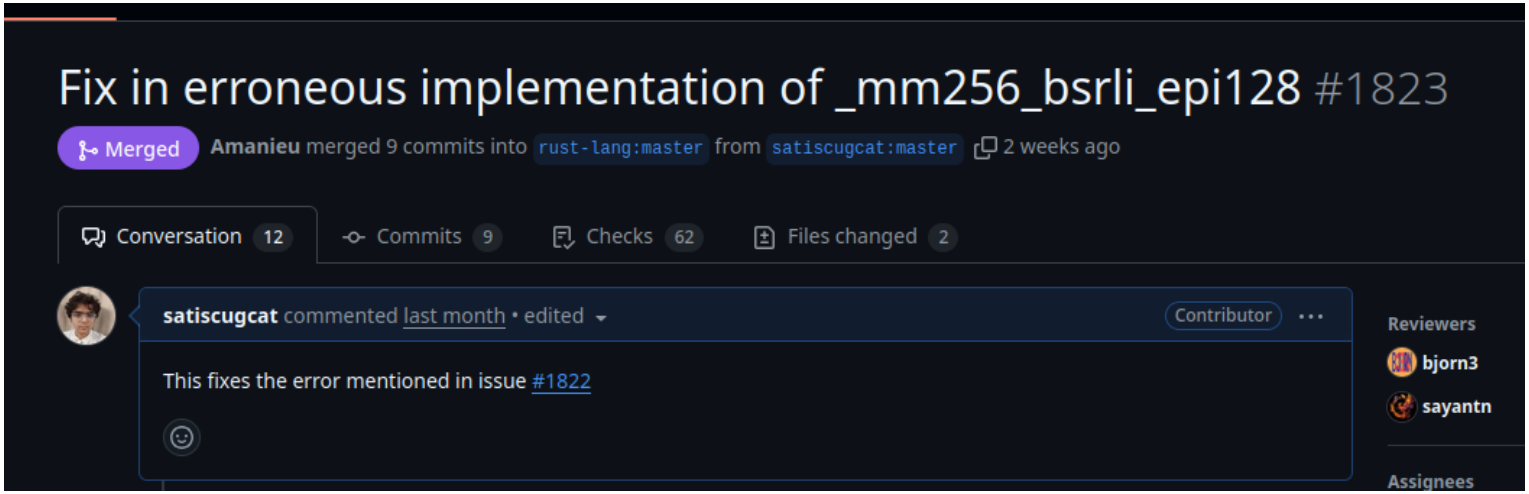
karthikbhargavan commented on Jul 25 • edited

Solution to challenge 15, resolves #173.

This PR provides testable models for `core::arch` intrinsics, including abstractions to streamline the process of implementing these intrinsics and their tests.

Currently there are 384 x86 intrinsics modelled, and 181 aarch64 intrinsics modelled.

Finding a Bug in Rust



Future Work

This work has also already been used for formal verification purposes in the libcrux repository!

Future Work

This work has also already been used for formal verification purposes in the libcrux repository! For future work, we would like to work on the following things.

Future Work

This work has also already been used for formal verification purposes in the libcrux repository! For future work, we would like to work on the following things.

- Currently, we only have models for intrinsics that operate on integers. We would like to extend our approach to floating point numbers.

Future Work

This work has also already been used for formal verification purposes in the libcrux repository! For future work, we would like to work on the following things.

- Currently, we only have models for intrinsics that operate on integers. We would like to extend our approach to floating point numbers.
- We do not provide models for intrinsics that mutate values. This is also a line of further work.

Future Work

This work has also already been used for formal verification purposes in the libcrux repository! For future work, we would like to work on the following things.

- Currently, we only have models for intrinsics that operate on integers. We would like to extend our approach to floating point numbers.
- We do not provide models for intrinsics that mutate values. This is also a line of further work.
- Our methodology efficiently models Rust intrinsics, but does not discover bugs.

Concluding

That is all. Thank you for coming to my talk! I hope it was somewhat informative. If you want to get in contact with CRYSPEN, you can do so via the website. You can find the content of this talk at satiscugcat/fsttcs-presentations. Finally, at IIT Gandhinagar, I am organising a student group gor PL work called ,\ AMBDA. If that sounds interesting to you and you want to be involved/collaborate, please do reach out! My email is aniket.mishra@iitgn.ac.in.