



All Your Assembly Belongs to Rust: Automated Lifting for Uniform Testing and Verification

Charly Castes, [Gurvan Debaussart](#), Thomas Bourgeat

■ Data Center
Systems
Laboratory

■ Verification and
Computer
Architecture
Laboratory

Introduction: Inline assembly in system software

Inline assembly lets programmers *mix* assembly directly into a higher-level language

Introduction: Inline assembly in system software

Inline assembly lets programmers *mix* assembly directly into a higher-level language

```
1 fn high_level_function() {  
2     // ...  
3     asm!(  
4         "do_thing_fast"  
5         "do_thing_constant_time"  
6         "do_thing_machine_level"  
7     );  
8     // ...  
9 }
```

Introduction: Inline assembly in system software

Inline assembly lets programmers *mix* assembly directly into a higher-level language

```
1 fn high_level_function() {  
2     // ...  
3     asm!(  
4         "do_thing_fast"           ← Compression  
5         "do_thing_constant_time"  
6         "do_thing_machine_level"  
7     );  
8     // ...  
9 }
```

Introduction: Inline assembly in system software

Inline assembly lets programmers *mix* assembly directly into a higher-level language

```
1 fn high_level_function() {  
2     // ...  
3     asm!(  
4         "do_thing_fast"           ← Compression  
5         "do_thing_constant_time" ← Cryptography  
6         "do_thing_machine_level"  
7     );  
8     // ...  
9 }
```

Introduction: Inline assembly in system software

Inline assembly lets programmers *mix* assembly directly into a higher-level language

```
1 fn high_level_function() {  
2     // ...  
3     asm!(  
4         "do_thing_fast"           ← Compression  
5         "do_thing_constant_time" ← Cryptography  
6         "do_thing_machine_level" ← Kernels, Hypervisors  
7     );  
8     // ...  
9 }
```

Introduction: Inline assembly in system software

Inline assembly lets programmers *mix* assembly directly into a higher-level language

```
1 fn high_level_function() {  
2     // ...  
3     asm!(  
4         "do_thing_fast"           ← Compression  
5         "do_thing_constant_time" ← Cryptography  
6         "do_thing_machine_level" ← Kernels, Hypervisors  
7     );  
8     // ...  
9 }
```

These programs are *hard* to work with. No tool support (e.g. Miri), requires cross-compilation, break standard verification tools (Kani)...

Introduction: Inline assembly correctness

Difficulty 1: Inline assembly is treated as an opaque block

¹R. Jung. “How to use storytelling to fit inline assembly into Rust”. Mar. 2026.

Introduction: Inline assembly correctness

Difficulty 1: Inline assembly is treated as an opaque block

Insight: A *well-defined* assembly block must have an equivalent pure high-level program¹

¹R. Jung. “How to use storytelling to fit inline assembly into Rust”. Mar. 2026.

Introduction: Inline assembly correctness

Difficulty 1: Inline assembly is treated as an opaque block

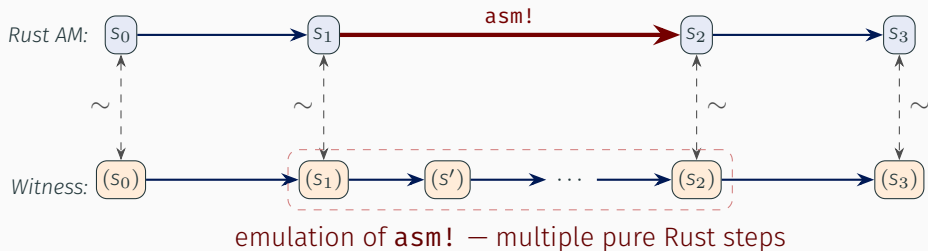
Insight: A *well-defined* assembly block must have an equivalent pure high-level program¹



Introduction: Inline assembly correctness

Difficulty 1: Inline assembly is treated as an opaque block

Insight: A *well-defined* assembly block must have an equivalent pure high-level program¹



¹R. Jung. “How to use storytelling to fit inline assembly into Rust”. Mar. 2026.

Introduction: Emulate to verify

Difficulty 2: Inline assembly depends on state hidden from the high-level Rust Abstract Machine (AM)

Introduction: Emulate to verify

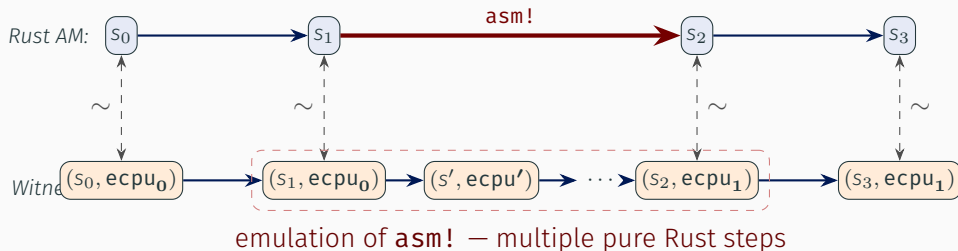
Difficulty 2: Inline assembly depends on state hidden from the high-level Rust Abstract Machine (AM)

Our approach: Build the witness through *emulation*, extend state with an *emulated machine state* (ecpu)

Introduction: Emulate to verify

Difficulty 2: Inline assembly depends on state hidden from the high-level Rust Abstract Machine (AM)

Our approach: Build the witness through *emulation*, extend state with an *emulated machine state* (ecpu)



Introduction: Creating a witness automatically through lifting

Our tool, **Softcore**, constructs an emulation-based witness automatically for RISC-V through a *lifting process*

Introduction: Creating a witness automatically through lifting

Our tool, **Softcore**, constructs an emulation-based witness automatically for RISC-V through a *lifting process*

Since the witness is a pure Rust program, it allows us to:

Introduction: Creating a witness automatically through lifting

Our tool, **Softcore**, constructs an emulation-based witness automatically for RISC-V through a *lifting process*

Since the witness is a pure Rust program, it allows us to:

1. Run and test RISC-V inline assembly on any host architecture

Introduction: Creating a witness automatically through lifting

Our tool, **Softcore**, constructs an emulation-based witness automatically for RISC-V through a *lifting process*

Since the witness is a pure Rust program, it allows us to:

1. Run and test RISC-V inline assembly on any host architecture
2. Verify our programs using any off-the-shelf Rust verification tools (Kani, ...)

Creating an executable model of our target architecture

Creating an executable model of our target architecture

To maximize trust, executable model is built from the official Sail RISC-V specification



Translating assembly into calls to our executable model

Calling the executable model: Connecting worlds

Conceptually, the lifting requires *bridging a gap between two different worlds*

Calling the executable model: Connecting worlds

Conceptually, the lifting requires *bridging a gap between two different worlds*

Calling the executable model: Connecting worlds

Conceptually, the lifting requires *bridging a gap between two different worlds*

Rust
variables
typed values
(i64, &mut [u8])

Calling the executable model: Connecting worlds

Conceptually, the lifting requires *bridging a gap between two different worlds*

Rust

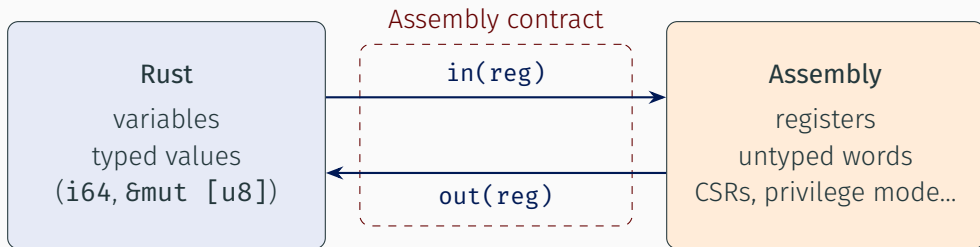
variables
typed values
(i64, &mut [u8])

Assembly

registers
untyped words
CSRs, privilege mode...

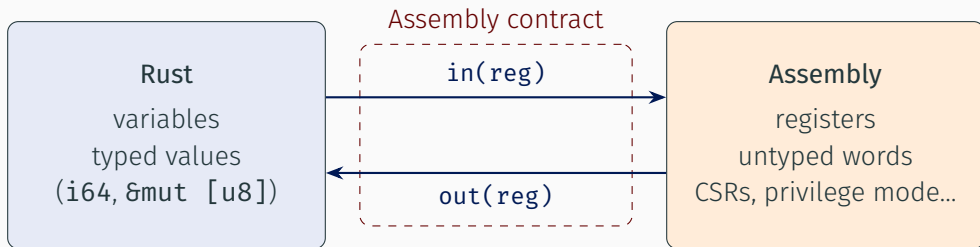
Calling the executable model: Connecting worlds

Conceptually, the lifting requires *bridging a gap between two different worlds*



Calling the executable model: Connecting worlds

Conceptually, the lifting requires *bridging a gap between two different worlds*



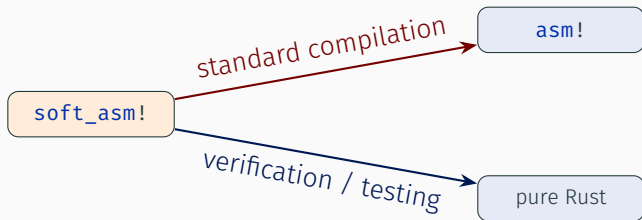
The contract states which Rust values are accessible from Assembly and in which register they can be accessed, and which register holds which result on exit

Calling the executable model: A custom macro

The executable model is called with a new `soft_asm!` macro

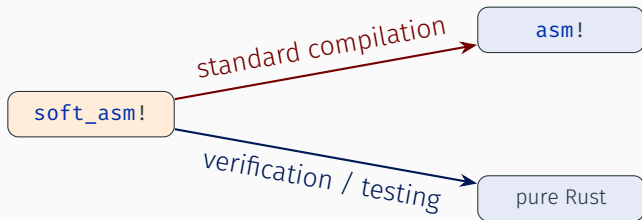
Calling the executable model: A custom macro

The executable model is called with a new `soft_asm!` macro



Calling the executable model: A custom macro

The executable model is called with a new `soft_asm!` macro



Let's see a few examples of what this macro creates!

Calling the executable model: Pure register operations

```
1 fn abs(x: i64) -> i64 {  
2     let sign: i64; let result: i64;  
3     soft_asm!(  
4         "srai {sign}, {x}, 63",  
5         "xor {result}, {x}, {sign}",  
6         "sub {result}, {result}, {sign}",  
7         x      = in(reg) x,  
8         sign   = out(reg) sign,  
9         result = out(reg) result,  
10    );  
11    result  
12 }
```

Calling the executable model: Pure register operations

```
1 fn abs(x: i64) -> i64 {  
2     let sign: i64; let result: i64;  
3     soft_asm!(  
4         "srai {sign}, {x}, 63",  
5         "xor {result}, {x}, {sign}",  
6         "sub {result}, {result}, {sign}",  
7         x      = in(reg) x,  
8         sign    = out(reg) sign,  
9         result  = out(reg) result,  
10    );  
11    result  
12 }
```

The assembly world operates on registers, therefore the actual Rust values have to be placed in and out of registers according to the contract

Calling the executable model: Pure register operations

```
1 fn abs(x: i64) -> i64 {  
2     let sign: i64;  
3     let result: i64;
```

Calling the executable model: Pure register operations

```
1 fn abs(x: i64) -> i64 {  
2     let sign: i64;  
3     let result: i64;  
4     // Contract input  
5     ecpu.set(reg::X5, x as u64); // x = in(reg) x
```

Calling the executable model: Pure register operations

```
1 fn abs(x: i64) -> i64 {  
2     let sign: i64;  
3     let result: i64;  
4     // Contract input  
5     ecpu.set(reg::X5, x as u64); // x = in(reg) x  
6     // Assembly body  
7     ecpu.execute(ast::SHIFTIOP(63, reg::X5, reg::X6, sop::SRAI));  
8     ecpu.execute(ast::RTYPE(reg::X6, reg::X5, reg::X7, rop::XOR));  
9     ecpu.execute(ast::RTYPE(reg::X6, reg::X7, reg::X7, rop::SUB));
```

Calling the executable model: Pure register operations

```
1 fn abs(x: i64) -> i64 {
2     let sign: i64;
3     let result: i64;
4     // Contract input
5     ecpu.set(reg::X5, x as u64); // x = in(reg) x
6     // Assembly body
7     ecpu.execute(ast::SHIFTIOP(63, reg::X5, reg::X6, sop::SRAI));
8     ecpu.execute(ast::RTYPE(reg::X6, reg::X5, reg::X7, rop::XOR));
9     ecpu.execute(ast::RTYPE(reg::X6, reg::X7, reg::X7, rop::SUB));
10    // Contract output
11    sign = ecpu.get(reg::X6) as i64; // sign = out(reg) sign
12    result = ecpu.get(reg::X7) as i64; // result = out(reg) result
13    // Ready to model-check!
14    assert!(result == x.wrapping_abs());
15    result
16 }
```

Calling the executable model: Memory accesses

```
1 fn memset(inp: &mut [u8], fill: u8) {  
2     for i in 0..inp.len() {  
3         soft_asm!(  
4             "sb {fill}, 0({ptr})",  
5             fill = in(reg) fill,  
6             ptr = in(reg) inp[i..].as_mut_ptr(),  
7         );  
8     }  
9 }
```

Calling the executable model: Memory accesses

```
1 fn memset(inp: &mut [u8], fill: u8) {  
2     for i in 0..inp.len() {
```

Calling the executable model: Memory accesses

```
1 fn memset(inp: &mut [u8], fill: u8) {  
2     for i in 0..inp.len() {  
3         // Contract input  
4         ecpu.set(reg::X5, fill as u64);  
5         ecpu.set(reg::X6, inp[i..].as_mut_ptr() as u64);
```

Calling the executable model: Memory accesses

```
1 fn memset(inp: &mut [u8], fill: u8) {  
2     for i in 0..inp.len() {  
3         // Contract input  
4         ecpu.set(reg::X5, fill as u64);  
5         ecpu.set(reg::X6, inp[i..].as_mut_ptr() as u64);  
6         // Assembly body  
7         core::ptr::write(  
8             ecpu.get(reg::X6) as *mut u8,  
9             ecpu.get(reg::X5) as u8  
10        );  
11     }  
12 }
```

Calling the executable model: Control flow

Jumps inside blocks are rebuilt into structured Rust by a simple, best-effort, *relooper*

Calling the executable model: Control flow

Jumps inside blocks are rebuilt into structured Rust by a simple, best-effort, *relooper*

Some assembly control flows are non-local such as *traps*

Calling the executable model: Control flow

Jumps inside blocks are rebuilt into structured Rust by a simple, best-effort, *relooper*

Some assembly control flows are non-local such as *traps*

```
1 soft_asm!("csrrw x1, ustatus, x0");
```

Calling the executable model: Control flow

Jumps inside blocks are rebuilt into structured Rust by a simple, best-effort, *relooper*

Some assembly control flows are non-local such as *traps*

```
1 soft_asm!("csrrw x1, ustatus, x0");
```

We make the *execute* function either terminate properly or return the address of a trap handler

Calling the executable model: Control flow

Jumps inside blocks are rebuilt into structured Rust by a simple, best-effort, *relooper*

Some assembly control flows are non-local such as *traps*

```
1 soft_asm!("csrrw x1, ustatus, x0");
```

We make the *execute* function either terminate properly or return the address of a trap handler

```
1 match ecpu.execute(ast::CSRReg(0, reg::X1, reg::X0, csrop::CSRRW))  
2 {  
3     Trap::Some(handler_pc) => handle_trap(handler_pc, &[handler1, ...]),  
4     Trap::None => (),  
5 };
```

Calling the executable model: Control flow

Jumps inside blocks are rebuilt into structured Rust by a simple, best-effort, *relooper*

Some assembly control flows are non-local such as *traps*

```
1 soft_asm!("csrrw x1, ustatus, x0");
```

We make the *execute* function either terminate properly or return the address of a trap handler

```
1 match ecpu.execute(ast::CSRReg(0, reg::X1, reg::X0, csrop::CSRRW))  
2 {  
3     Trap::Some(handler_pc) => handle_trap(handler_pc, &[handler1, ...]),  
4     Trap::None => (),  
5 };
```

(More on control flow in the paper)

Calling the executable model: System instructions

In our framework, system instructions are nothing special:
They rely on and modify the emulated machine state

Calling the executable model: System instructions

In our framework, system instructions are nothing special:

They rely on and modify the emulated machine state

Afterwards we can query the executable model: is this address readable, writable, or executable in the current mode? Is this PMP entry locked? Which privilege mode are we in?

Evaluation: Practical usage of our lifting procedure

Evaluation: Miralis



Miralis² is a virtual firmware monitor, written in Rust, which runs privileged M-mode assembly in user mode

²C. Castes et al. “The Design and Implementation of a Virtual Firmware Monitor.”. 2025.

Evaluation: Miralis



Miralis² is a virtual firmware monitor, written in Rust, which runs privileged M-mode assembly in user mode

Our previous work used Kani to prove (among other things) that Miralis correctly initialized physical memory protections (PMPs) through a *manual* lift into pure functions

²C. Castes et al. “The Design and Implementation of a Virtual Firmware Monitor.”. 2025.

Evaluation: Miralis



Miralis² is a virtual firmware monitor, written in Rust, which runs privileged M-mode assembly in user mode

Our previous work used Kani to prove (among other things) that Miralis correctly initialized physical memory protections (PMPs) through a *manual* lift into pure functions

Softcore replaced all manually written procedures with a single macro change!

²C. Castes et al. “The Design and Implementation of a Virtual Firmware Monitor.”. 2025.

Flashpoint

Flashpoint³ is our dynamic root of trust firmware for RISC-V (~ 1 kLoC)

³N. S. Kalani et al. “Flashpoint: Privileged-Software Isolation on RISC-V for Dynamic Root of Trust”. *ASPLOS-XXXII*. 2027.

Flashpoint

Flashpoint³ is our dynamic root of trust firmware for RISC-V (~ 1 kLoC)

At a high level, it implements a simple transition system where each transition has to ensure:

1. Correct memory protection configuration at transition exit
2. The code implementing each transition executes without faults

³N. S. Kalani et al. “Flashpoint: Privileged-Software Isolation on RISC-V for Dynamic Root of Trust”. *ASPLOS-XXXII*. 2027.

Evaluation: Flashpoint

Flashpoint

Flashpoint³ is our dynamic root of trust firmware for RISC-V (~ 1 kLoC)

At a high level, it implements a simple transition system where each transition has to ensure:

1. Correct memory protection configuration at transition exit
2. The code implementing each transition executes without faults

Softcore and Kani proved both properties for every transition and symbolic initial state (in ~ 3 hours and ~ 50 GB of memory)

³N. S. Kalani et al. “Flashpoint: Privileged-Software Isolation on RISC-V for Dynamic Root of Trust”. *ASPLOS-XXXII*. 2027.

Limitations

Many:

1. Limited Relooper
2. Limited stack handling
3. No out of blocks jumps
4. No checks of underlying assumption of the Rust AM (e.g. Locking execution of current program)

Conclusion

We presented an *emulation-based* way to run, test, and verify inline assembly in Rust, *including privileged assembly*, with off-the-shelf tools

Conclusion

We presented an *emulation-based* way to run, test, and verify inline assembly in Rust, *including privileged assembly*, with off-the-shelf tools

We used our tool to improve and extend verification of existing low-level software containing privileged assembly

Conclusion

We presented an *emulation-based* way to run, test, and verify inline assembly in Rust, *including privileged assembly*, with off-the-shelf tools

We used our tool to improve and extend verification of existing low-level software containing privileged assembly

Questions?



Check it out!

github.com/CharlyCst/softcore-asm