Stable Inline Assembly

Florian “Florob” Zeitz

2022-06-20

What is Inline Assembly?

- Handwritten assembly to compile into the final binary
- Written *inline* in function bodies
- Integrates with surrounding code
- Available in some systems programing languages

Why do we need Inline Assembly?

- low-level control
  - configuration registers
  - hardware not otherwise accessible
- performance
- predictable timing
  - cryptography (constant time)
  - embedded
- convenience
  - no separate assembly file
  - no need for a `build.rs`
- Rust philosophy: safe with **`unsafe {}`** escape hatches where needed

Why do we need Inline Assembly?

- low-level control
  - configuration registers
  - hardware not otherwise accessible
- performance
- predictable timing
  - cryptography (constant time)
  - embedded
- convenience
  - no separate assembly file
  - no need for a `build.rs`
- Rust philosophy: safe with `unsafe {}` escape hatches where needed

Why do we need Inline Assembly?

- low-level control
  - configuration registers
  - hardware not otherwise accessible
- performance
- predictable timing
  - cryptography (constant time)
  - embedded
- convenience
  - no separate assembly file
  - no need for a `build.rs`
- Rust philosophy: safe with `unsafe {}` escape hatches where needed

Why do we need Inline Assembly?

- low-level control
  - configuration registers
  - hardware not otherwise accessible
- performance
- predictable timing
  - cryptography (constant time)
  - embedded
- convenience
  - no separate assembly file
  - no need for a `build.rs`
- Rust philosophy: safe with **`unsafe {}`** escape hatches where needed

Why do we need Inline Assembly?

- low-level control
  - configuration registers
  - hardware not otherwise accessible
- performance
- predictable timing
  - cryptography (constant time)
  - embedded
- convenience
  - no separate assembly file
  - no need for a `build.rs`
- Rust philosophy: safe with **`unsafe {}`** escape hatches where needed

Why not use intrinsics?

- "Just" using an intrinsic per instruction has often been suggested
- Can be re-ordered
- Instructions can be inserted in between
- Hard to scale
- Unreasonable for rare use-cases

However: Definitely use them when applicable

Assembly Primer

- Textual representation of CPU instructions
- Usually a mnemonic followed by operands
- Modifies registers, memory, flags, ...
- Labels to jump to
- Immediate values

Examples

x86 (AT&T)

```
1  jmp foo
2  mov $5, %eax
3  foo:
4  mov 5(%eax, %ebx, 2), %edx
```

x86 (Intel)

```
1  jmp foo
2  mov eax, 5
3  foo:
4  mov edx, [eax + ebx*2 + 5]
```

ARM

```
1  ldmiaeq sp!, {r4-r7, pc}
2  ldrb r0, [sp, #32]
```

PowerPC

```
1  lwz r0, 8(r1)
2  rlwimi r11, r1, 0, 30, 28
```

Prior Art

- Available in some systems programing languages
- Some use a template style (e. g. GNU-C, LLVM-IR, ldc)
- Some use an embedded DSL (e. g. D, MSVC)

Example: gcc


```
1 asm [volatile] ("template" : outs : ins : clobber)
```

- Uses a template strings
- Compiler fills template based on constraints
- Template result interpreted by the assembler (compiler doesn't have to know instructions)

gcc Template

- `%0`, or `%[name]` as placeholders
- Escapes: `%%`, `%{`, `%}`, `%|`

gcc Arguments

- Specify outputs, inputs, and clobbers
- Constraints determine how to fill the template (register, address, immediate)
- All inputs have to be read before any output is written
- Clobbers specify touched state, that can not be inferred from inputs and outputs
- Compiler moves arguments into registers, saves clobbered registers, knows to reload changed memory

gcc Constraints

- Constraint codes are usually a single letter
- Output
  - start with = or + (input and output)
  - can be modified with & to allow write before all inputs are read (early-clobber)
- Input
  - can be tied to outputs by specifying the operand number as constrained
- Clobbers
  - Used registers
  - Modified flags
  - If memory was modified

gcc Examples: Add 5 to variable


```
1 int var = 0;
2 asm ("add $5, %0" : "+r"(var));
```

gcc Examples: Get L1 cache size


```
1  int ebx, ecx;
2  asm (
3      "mov $4, %%eax;"
4      "xor %%ecx, %%ecx;"
5      "cpuid;"
6      "mov %%ebx, %0;"
7      : "=r"(ebx), "=c"(ecx)
8      :
9      : "eax", "ebx", "edx"
10 );
11 printf("L1 Cache: %i\n", ((ebx >> 22) + 1)
12     * (((ebx >> 12) & 0x3ff) + 1)
13     * ((ebx & 0xfff) + 1)
14     * (ecx + 1));
```

gcc Examples: Increment


```
1  int in = 3, out;
2  asm (
3      "mov $1, %0\r\n"
4      "add %1, %0\r\n"
5      : "=r"(out)
6      : "r"(in)
7  );
```

Is this correct?

gcc Examples: Increment


```
1 int in = 3, out;
2 asm (
3     "mov $1, %0\r\n"
4     "add %1, %0\r\n"
5     : "=r"(out)
6     : "r"(in)
7 );
```

Is this correct? No, `in` and `out` could be in the same register.

gcc-style Pros/Cons

Pros

- Existing implementations support all architectures
- Arbitrary assembly supported
- Can be optimized

Cons

- Hard to write
- Easy to get constraints wrong
- Missing constraints lead to undefined behavior

Example: D


```
1 int var = 0;
2 asm {
3     mov RAX, RDX;
4     add var, 3 + 2;
5     add var, RAX;
6 }
```

- Implemented as DSL inside `asm {}`
- Standardized per CPU family
- Many D expressions allowed

DSL-style Pros/Cons

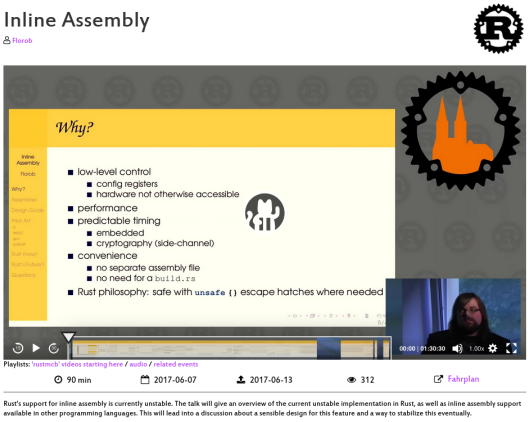
Pros

- Easy to write
- Tight integration with surrounding code
- Can be optimized
- Impossible to forget constraints

Cons

- Much work to support a new architecture
- Existing implementations only support x86(_64)
- Existing implementations don't support some instructions and registers
- Neither assembly nor language syntax

Inline Assembly 2017



https://media.ccc.de/v/rustcgn.2017.06.inline-assembly

Global ASM

- Stable since Rust 1.59, but specified since 2016
- Include arbitrary assembly in the `.text` section
- Only in "global scope" (outside functions)
- Replaces linking to externally compiled assembly
- Can use `#[cfg()]` directives

Global ASM


```
std::arch::global_asm!(r#"
    .globl add_5
    add_5:
        mov rax, rdi
        add rax, 5
        ret
"#);

extern {
    fn add_5(n: u64) -> u64;
}

fn main() {
    println!("{}", unsafe { add_5(37) });
}
```

llvm_asm!()


```
1 unsafe {
2     llvm_asm!("template"
3         : outs
4         : ins
5         : clobbers
6         : options
7     );
8 }
```

- Used to be `asm!()`
- Straight binding to LLVM IR (gcc-style)
- Ever so slightly different from both LLVM IR and GNU-C
- Renamed, deprecated, now removed

llvm_asm!()


```
1 let mut var = 0;
2 unsafe {
3     llvm_asm!("add $$5, $0" : "+r"(var));
4 }
```

- No good route to stabilization
- Had various known bugs
- Hard to support on non-LLVM backends

(Pre-)RFCs

2014 "RFC 129: refine the `asm!` extension" by pczarn

2018 "[Pre-RFC]: Inline assembly" by Florob

2019 "[Pre-RFC #2]: Inline assembly" by Amanieu

2019 "RFC 2836: Introduce the ASM project group" by Amanieu

2020 "RFC 2850: Inline assembly" by Amanieu

2020 "RFC 2843: Add llvm_asm! and deprecate asm!" by Amanieu (merged)

2020 "RFC 2873 Inline assembly" by Amanieu (merged)

Design Goals (of the initial Pre-RFC)

- Easy to use
- Portable
  - Between platforms (x86, AMD64, ARM, AVR, PowerPC, RISC-V, ...)
  - Between backends (LLVM, Cretone, gcc, ...)
- Support all instructions (with all operand types)

asm!()


```
asm!("template", operands, options);
```

- Stabilized in Rust 1.59
- Fundamentally gcc-style
- Easier syntax than gcc
- Safer defaults than gcc

Template

- Designed after Rust format strings
- Multiple template arguments allowed, each representing one line
- `{}` or `{name}` as placeholder
- Modifiers via `{:h}`
- Escapes: `{{, }}`

Arguments

- Outputs and inputs, in any order
- Clobbers are just ignored outputs
- Inputs can be read before outputs

Constraints

- Register classes (e. g. `reg, xmm_reg`)
- Specific registers as string literals (e. g. `"eax"`)
- No memory, supported by using an address in a register

Example: Outputs


```
1  use std::arch::asm;
2
3  let x: u64;
4  unsafe {
5      asm!("mov {}, 42", out(reg) x);
6  }
```

Example: Inputs


```
1  use std::arch::asm;
2
3  let input: u64 = 3;
4  let output: u64;
5  unsafe {
6      asm!(
7          "mov {0}, 1",
8          "add {0}, {1}",
9          out(reg) output,
10         in(reg) input,
11     );
12 }
```

Is this correct?

Example: Inputs


```
1  use std::arch::asm;
2
3  let input: u64 = 3;
4  let output: u64;
5  unsafe {
6      asm!(
7          "mov {0}, 1",
8          "add {0}, {1}",
9          out(reg) output,
10         in(reg) input,
11     );
12 }
```

Is this correct? Yes, outputs are early-clobber by default

Example: Late Outputs


```
1  use std::arch::asm;
2
3  let input: u64 = 3;
4  let output: u64;
5  unsafe {
6      asm!(
7          "mov {0}, {1}",
8          "add {0}, 1",
9          lateout(reg) output,
10         in(reg) input,
11     );
12 }
```

- Outputs that may only be written after all inputs are read
- Allows for more optimized code

Example: Combined Input and Output


```
1  use std::arch::asm;
2
3  let mut val: u64 = 6;
4  let val_plus5: u64;
5  unsafe {
6      asm!("add {}, 1", inout(reg) val);
7
8      asm!(
9          "add {}, {}",
10         inout(reg) 5u64 => val_plus5,
11         in(reg) val,
12     );
13 }
```

Example: Get L1 Cache Size


```
let rbx: u64;
let rcx: u64;
unsafe { asm!(
    "push rbx",
    "cpuid",
    "mov {scratch}, rbx",
    "pop rbx",
    scratch = lateout(reg) rbx, // rbx is reserved by LLVM
    inlateout("rax") 4u64 => _,
    inlateout("rcx") 0u64 => rcx,
    out("rdx") _
) };
println!("L1 Cache: {}", ((rbx >> 22) + 1)
    * (((rbx >> 12) & 0x3ff) + 1)
    * ((rbx & 0xfff) + 1) * (rcx + 1));
```

Safer defaults


```
let mut a: u64 = 4;
let b: u64 = 4;
unsafe { asm!(
    "add {0}, {1}",
    inlateout(reg) a, in(reg) b,
    options(pure, nomem, nostack),
) };
assert_eq!(a, 8);
```

- Assumes volatile by default, `option(pure)` to override
- Assumes memory access by default, `option(nomem)` to override
- Assumes stack access by default, `option(nostack)` to override

Goals?

- Currently supported architectures: x86, AMD64, ARM, AArch64, RISC-V
- Full support only in LLVM backend
- Can fall back to treating inline assembly like a function call

Thank you for your attention. Any questions?

https://babelmonkeys.de/~florob/talks/RC-2022-06-20-rust-inline-asm.pdf