Inline Assembly

Florian "Florob" Zeitz

2017-06-07

Why?

Why?

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

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
```

Properties

- textual representation of the instructions a CPU executes
- mnemonics
- operands
  - register
  - immediate
  - label
  - combinations thereof

Design Goals

- 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?)

$\mathcal{D}$

```
1 asm { mov RAX, RDX; }
```

- implemented as DSL inside `asm {}`
- standardized per CPU family
- can be marked `pure` and `nothrow`
- only x86/AMD64 are currently available
- Docs: `https://dlang.org/spec/iasm.html`

D's x86/AMD64 syntax

- close to regular Intel syntax
- registers, always uppercase
- mnemonics, always lowercase
- local variables accessed as `var[EBP]`
  (just `var` when not in `naked` code)
- many D expressions allowed
- `$` represents the PC, but can't be used for PIC
- pseudo-ops for:
  - alignment
  - data definition
  - prefixes (`lock`, `rep`, …)

D Examples: Add 5 to variable


```
1  int var = 0;
2  asm {
3      add var, 3 + 2;
4  }
```

D Examples: Get L1 cache size


```
1  int ebx, ecx;
2  asm {
3      mov EAX, 4;
4      xor ECX, ECX;
5      cpuid;
6      mov ebx, EBX;
7      mov ecx, ECX;
8  }
9  writefln("L1 Cache: %s",
10     ((ebx >> 22) + 1) * (((ebx >> 12) & 0x3ff) + 1)
11     * ((ebx & 0xfff) + 1) * (ecx + 1));
```

MSVC


```
1  __asm { mov rax, rdx }
```

- DSL after `__asm`
- only available for x86 (Pentium 4 and AMD Athlon opcodes)
- uses MASM (Microsoft Macro Assembler) expressions and C/C++ elements
- allows jumps to C labels from ASM and vice versa
- allows calls to C functions
- C/C++ operators cannot be used
- Docs: `https://docs.microsoft.com/en-us/cpp/assembler/inline/inline-assembler`

MSVC, supported MASM

- not supported:
  - data definitions
  - macros
- supported:
  - alignment
  - comments

MSVC, supported C/C++ elements

- symbols (labels, function names, variable names)
  MASM reserved words take precedence over symbols
- constants
- comments
- macros and preprocessor directives
- type names where a MASM type would be legal

MSVC Examples: Add 5 to variable


```
1  int var = 0;
2  __asm {
3      add var, 5
4  }
```

MSVC Examples: Get L1 cache size


```
1  int ebx_v, ecx_v;
2  __asm {
3      mov eax, 4
4      xor ecx, ecx
5      cpuid
6      mov ebx_v, ebx
7      mov ecx_v, ecx
8  }
9  std::cout << "L1 Cache: "
10     << ((ebx_v >> 22) + 1) * (((ebx_v >> 12) & 0x3ff) + 1)
11         * ((ebx_v & 0xfff) + 1) * (ecx_v + 1))
12     << '\n';
```

gcc


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

- special syntax within `asm ()`
- uses a template strings
- template interpreted by the assembler (compiler doesn't have to know instructions)
- compiler fills template based on constraints
- Docs: `https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html`

gcc Template

- `%0`, or `%[name]` as placeholders
- supports modifiers, e.g. `%w0` to print a HImode register name (`%ax`)
- multiple dialects as `{mov %%eax, %%ebx | mov ebx, eax}`
  - specify one or all aviailable dialects
  - purely defined by order
- escapes: `%%`, `%{`, `%}`, `%|`

gcc Arguments

- specify outputs, inputs, and clobbers
- constraints determine how to reference arguments (register, address, immediate)
- all inputs have to 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
- architecture specific
- e. g. `r` for register, `m` for memory, `i` for immediate
- multiple alternative constraints are allowed, allowing the compiler to choose the most efficient

gcc Constraints

- 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 volatile

- `asm` statements can be declared `volatile`
- disables code motion and dead code elimination
- may be required when side-effects are not otherwise visible, e. g. `asm ("wfi");`

gcc Examples: Add 5 to variable


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

LLVM IR


```
1  %out = call i32 asm "template", "constraints"(i32 %in)
```

- used to implement gcc-style inline assembly
- outputs as return values, inputs as call arguments
- can be qualified with `sideeffect`, `alignstack`, `inteldialect`
- Docs: `http://llvm.org/docs/LangRef.html#inline-assembler-expressions`

LLVM Template

- `$0` as placeholder
- supports modifiers, e. g. `${0:w}`
- escape is `$$`

LLVM Constraints

- constraint codes are a single letter, `^` followed by two letters, or `{reg}`
- architecture specific
- e. g. `r` for register, `m` for memory, `i` for immediate
- multiple alternative constraints are allowed, allowing the compiler to choose the most efficient
- indirect outputs/inputs via the `*` modifier
  - argument is an address that will be read/written from, typically `=*m`

LLVM Constraints

- output
  - start with = (no +)
  - 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
  - start with ~
  - used registers
  - modified flags
  - if memory was modified

LLVM Examples: Add 5 to variable


```
1  %1 = alloca i32, align 4
2  store i32 0, i32* %1, align 4
3  %2 = load i32, i32* %1, align 4
4  %3 = call i32 asm "add $$5, $0", "=r,0"(i32 %2)
5  store i32 %3, i32* %1, align 4
```

LLVM Examples: Get L1 cache size


```
1  %1 = alloca i32, align 4
2  %2 = alloca i32, align 4
3  %3 = call { i32, i32 } asm
4       "mov $$4, %eax;xor %ecx, %ecx;cpuid;mov %ebx, $0;",
5       "=r,={ecx},~{eax},~{ebx},~{edx}"()
6  %4 = extractvalue { i32, i32 } %3, 0
7  %5 = extractvalue { i32, i32 } %3, 1
8  store i32 %4, i32* %1, align 4
9  store i32 %5, i32* %2, align 4
```

Rust `global_asm!()`


```
1 global_asm!(r"
2     add5:
3         mov %rdi, %rax
4         add $5, %rax
5         ret
6 ");
7 extern {
8   fn add5(i: i64) -> i64;
9 }
```

- RFC 1548
- insert assembly verbatim into the module
- not yet implemented

Rust `asm!()`


```
unsafe {
    asm!("template"
         : outs
         : ins
         : clobbers
         : options
        );
}
```

- straight binding to LLVM IR, but supports +
- options are `volatile`, `alignstack`, and `intel`
- Docs: `https://doc.rust-lang.org/unstable-book/asm.html`

Rust Examples: Add 5 to variable


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

Rust Examples: Get L1 cache size


```
let ebx: i32;
let ecx: i32;
unsafe {
  asm!(r"
        mov $$4, %eax;
        xor %ecx, %ecx;
        cpuid;
        mov %ebx, $0;"
    : "=r"(ebx), "={ecx}"(ecx) : : "eax", "ebx", "edx"
  );
}
println!("L1 Cache: {}", ((ebx >> 22) + 1)
  * (((ebx >> 12) & 0x3ff) + 1)
  * ((ebx & 0xfff) + 1) * (ecx + 1));
```

RFC 129


```
asm!("assembly template",
   positional parameters,
   named parameters,
   clobbers and options
);
```

- positional parameters:
  `expr1, expr2_in -> expr2_out, "eax" = expr3_in -> expr3_out, …`
- named parameters:
  `name1 = expr_in_out, name2 = expr_in -> expr_out, …`
- clobbers and options:
  `"eax", "ebx", "memory", "volatile", "intel", …`

RFC 129 Example


```
1  fn addsub(a: int, b: int) -> (int, int) {
2    let mut c = 0;
3    let mut d = 0;
4    unsafe {
5      asm!("add {2:r}, {:=r}\n\t\
6            sub {2:r}, {:=r}",
7          a -> c, a -> d, b);
8    }
9    (c, d)
10 }
```

Some Questions

- Should we go the DSL, or template route?
- Does it make sense to support both?
- What to do about early-clobber?
- What placehoder should we use?
- Should we copy gcc since it's familiar?
- Are single character constraints sensible? Alternatives?
- Are sigil heavy constraints sensible? Alternatives?

My Thoughts

- Templates work better for portability
- We should use the same placeholder as everywhere else `{}`
- Use words over sigils for constraints
- Allow inputs and outputs in any order
- Either do early-clobber by default, or be explicit (`early_out`/`late_out`)

Example: Add 5 to variable


```
1  let mut var = 0;
2  unsafe {
3    asm!("add $5, {}", inout(reg) var);
4  }
```

Example: Get L1 cache size


```
let ebx: i32;
let ecx: i32;
unsafe {
  asm!(r"
        mov $$4, %eax;
        xor %ecx, %ecx;
        cpuid;
        mov %ebx, {};",
    out(reg) ebx, out(ecx) ecx, clobber(eax, ebx, edx)
  );
}
println!("L1 Cache: {}", ((ebx >> 22) + 1)
  * (((ebx >> 12) & 0x3ff) + 1)
  * ((ebx & 0xfff) + 1) * (ecx + 1));
```

Thank you for your attention. Any questions?

https://babelmonkeys.de/~florob/talks/RC-2017-06-07-inline-assembly.pdf