[asm] Inline assembly

[asm.intro] Support for inline assembly is provided via the `asm!` and `global_asm!` macros. It can be used to embed handwritten assembly in the assembly output generated by the compiler.

[asm.stable-targets] Support for inline assembly is stable on the following architectures:

- x86 and x86-64
- ARM
- AArch64 and Arm64EC
- RISC-V
- LoongArch
- s390x

The compiler will emit an error if `asm!` is used on an unsupported target.

[asm.example] Example

```
use std::arch::asm;

// Multiply x by 6 using shifts and adds
let mut x: u64 = 4;
unsafe {
    asm!(
        "mov {tmp}, {x}",
        "shl {tmp}, 1",
        "shl {x}, 2",
        "add {x}, {tmp}",
        x = inout(reg) x,
        tmp = out(reg) _,
    );
```

```
}
assert_eq!(x, 4 * 6);
```

[asm.syntax]

Syntax

The following ABNF specifies the general syntax:

```
format_string := STRING_LITERAL / RAW_STRING_LITERAL
dir_spec := "in" / "out" / "lateout" / "inout" / "inlateout"
reg_spec := <register class> / "\"" <explicit register> "\""
operand_expr := expr / "_" / expr "=>" expr / expr "=>" "_"
reg_operand := [ident "="] dir_spec "(" reg_spec ")" operand_expr /
sym <path> / const <expr>
clobber_abi := "clobber_abi(" <abi> *("," <abi>) [","] ")"
option := "pure" / "nomem" / "readonly" / "preserves_flags" /
"noreturn" / "nostack" / "att_syntax" / "raw"
options := "options(" option *("," option) [","] ")"
operand := reg_operand / clobber_abi / options
asm := "asm!(" format_string *("," format_string) *("," operand)
[","] ")"
global_asm := "global_asm!(" format_string *("," format_string) *("
operand) [","] ")"
```

[asm.scope]

Scope

[asm.scope.intro] Inline assembly can be used in one of two ways.

[asm.scope.asm] With the `asm!` macro, the assembly code is emitted in a function scope and integrated into the compiler-generated assembly code of a function. This assembly code must obey strict rules to avoid undefined behavior. Note that in some cases the compiler may choose to emit the assembly code as a separate function and generate a call to it.

```
unsafe { core::arch::asm!("/* {} */", in(reg) 0); }
```

[asm.scope.global_asm] With the `global_asm!` macro, the assembly code is emitted in a global scope, outside a function. This can be used to hand-write entire functions using assembly code, and generally provides much more freedom to use arbitrary registers and assembler directives.

```
core::arch::global_asm!("/* {} */", const 0);
```

[asm.ts-args] Template string arguments

[asm.ts-args.syntax] The assembler template uses the same syntax as format strings (i.e. placeholders are specified by curly braces).

[asm.ts-args.order] The corresponding arguments are accessed in order, by index, or by name.

```
let x: i64;
let y: i64;
let z: i64;
// This
unsafe { core::arch::asm!("mov {}, {}", out(reg) x, in(reg) 5); }
// ... this
unsafe { core::arch::asm!("mov {0}, {1}", out(reg) y, in(reg) 5); }
// ... and this
unsafe { core::arch::asm!("mov {out}, {in}", out = out(reg) z, in =
in(reg) 5); }
// all have the same behavior
assert_eq!(x, y);
assert_eq!(y, z);
```

[asm.ts-args.no-implicit] However, implicit named arguments (introduced by RFC #2795) are not supported.

```
let x = 5;
// We can't refer to `x` from the scope directly, we need an operand
like `in(reg) x`
unsafe { core::arch::asm!("/* {x} */"); } // ERROR: no argument named
x
```

[asm.ts-args.one-or-more] An `asm!` invocation may have one or more template string arguments; an `asm!` with multiple template string arguments is treated as if all the strings were concatenated with a `\n` between them. The expected usage is for each template string argument to correspond to a line of assembly code.

```
let x: i64;
let y: i64;
// We can separate multiple strings as if they were written together
unsafe { core::arch::asm!("mov eax, 5", "mov ecx, eax", out("rax") x,
out("rcx") y); }
assert_eq!(x, y);
```

[asm.ts-args.before-other-args] All template string arguments must appear before any other arguments.

```
let x = 5;
// The template strings need to appear first in the asm invocation
unsafe { core::arch::asm!("/* {x} */", x = const 5, "ud2"); } //
ERROR: unexpected token
```

[asm.ts-args.positional-first] As with format strings, positional arguments must appear before named arguments and explicit register operands.

```
let x = 5;
// Named operands need to come after positional ones
unsafe { core::arch::asm!("/* {x} {} */", x = const 5, in(reg) 5); }
// ERROR: positional arguments cannot follow named arguments or
explicit register arguments
```

```
let x = 5;
// We also can't put explicit registers before positional operands
unsafe { core::arch::asm!("/* {} */", in("eax") 0, in(reg) 5); }
// ERROR: positional arguments cannot follow named arguments or
explicit register arguments
```

[asm.ts-args.register-operands] Explicit register operands cannot be used by placeholders in the template string.

```
let x = 5;
// Explicit register operands don't get substituted, use `eax`
explicitly in the string
unsafe { core::arch::asm!("/* {} */", in("eax") 5); }
// ERROR: invalid reference to argument at index 0
```

[asm.ts-args.at-least-once] All other named and positional operands must appear at least once in the template string, otherwise a compiler error is generated.

```
let x = 5;
// We have to name all of the operands in the format string
unsafe { core::arch::asm!("", in(reg) 5, x = const 5); }
// ERROR: multiple unused asm arguments
```

[asm.ts-args.opaque] The exact assembly code syntax is target-specific and opaque to the compiler except for the way operands are substituted into the template string to form the code passed to the assembler.

[asm.ts-args.llvm-syntax] Currently, all supported targets follow the assembly code syntax used by LLVM's internal assembler which usually corresponds to that of the GNU assembler (GAS). On x86, the `.intel_syntax noprefix` mode of GAS is used by default. On ARM, the `.syntax unified` mode is used. These targets impose an additional restriction on the assembly code: any assembler state (e.g. the current section which can be changed with `.section`) must be restored to its original value at the end of the asm string. Assembly code that does not conform to the GAS syntax will result in assembler-specific behavior. Further constraints on the directives used by inline assembly are indicated by Directives Support.