GCC's assembler syntax

This page is meant to consolidate GCC's official extended asm syntax into a form that is consumable by mere mortals. It is intended to be accessible by people who know a little bit of C and a little bit of assembly.

Guide to this guide

> This documentation is better viewed on a desktop computer.
>
> Right below is the index of this page. Beyond that, here is some useful notation:
>
> - Non-trivial `asm` examples come with a [godbolt] link. Matt Godbolt's website hosts *Compiler Explorer*, a very useful resource to check out how compilers handle some given input.
> - Instructions link to the x86 instruction documentation that is also hosted on this website.
> - asm arguments are **colored** to be more easily identifiable.
> - Constraint strings have their background highlighted and have a tooltip that explains the constraint.

Index

Syntax overview

`asm` is a **statement**, not an expression.

```
asm <optional stuff> (
    "assembler template"
    : outputs
    : inputs
    : clobbers
    : labels)
```

Quick notes:

- the starting `asm` keyword is either `asm` or `__asm__`, at your convenience
- `<optional stuff>` may be empty, or the keyword(s) `volatile` or `goto` (explained below)
- `"assembler template"` is a required string that encodes the instruction(s) that you want to run (explained below)
- there is a combined maximum of 30 inputs, outputs and labels per `asm` statement
- outputs, inputs, clobbers, and labels are all optional. Colons are only required up to the parameter that you wish to use, and having nothing between colons is valid when you wish to skip a parameter. For instance, each of these (explained below) are valid:
  - `asm("movq %0, %0" : "+rm" (foo));`
  - `asm("addl %0, %1" : "+r" (foo) : "g" (bar));`
  - `asm("lfence" : /* no output */ : /* no input */ : "memory");`

Assembler template syntax

The assembler template contains instructions and operand placeholders. You should think of it as a format string. This string will be "printed" to the assembly file that the compiler generates. Instructions must be separated by a newline (literally `\n`), and it's popular to prefix them with a tab or spaces (although this is not necessary). I like to end lines with `" \n "`, which puts a cosmetic space between the instruction and the newline delimiter, and sets up indentation for the next instruction. The template is a good use case for C's string literal concatenation.

```
        asm("nop \n  "
            "nop \n  "
            "nop")
```

In this format string:

- you specify arguments using `%N`, where `N` refers to an argument by its **zero-based** number in order of appearance (outputs and inputs being in the same "namespace", outputs being first)
- you specify named arguments using `%[Name]` (see below for how to specify names)
- you specify a literal `%` by using `%%`, which you *will* need if you reference registers directly in x86 assembly with the AT&T syntax
- the GCC documentation additionally specifies `%=`, `%{`, `%|` and `%}`, which this page does not cover
- the compiler additionally supports format modifiers (much like `printf` supports `%u` and `%hu`), which are architecture-dependent, and also not covered by this document

Outputs and inputs

Outputs and inputs are the parameters to your assembler template "format string". They are comma-separated. They both use either one of the two following patterns:

```
              "constraint" (expression)
       [Name] "constraint" (expression)
```

Note that although this looks metasyntactical, these are in fact the literal spellings that you must use: `[Name]` (when used) **needs** to be enclosed in square brackets, `"constraint"` **needs** to be enclosed in double quotes, and `(expression)` **needs** to be enclosed in parentheses.

`"constraints"`, and the kind of `(expression)`s that are valid, are explained in the Constraints, Outputs and Inputs sections below.

When you specify the optional `[Name]` field, you become able to refer to that input or output using the `%[Name]` syntax in the assembler template. For instance:

```
int foo = 1;
asm("inc %[IncrementMe]" : [IncrementMe] "+r" (foo));
// foo == 2
```

Even when names are specified, you can still refer to operands using the numbered syntax. Corollary: named arguments contribute to the sequence of numbered arguments. The second argument of an `asm` statement is always available as `%1`, regardless of whether the first argument has a name or not.

Constraints

Constraint strings bridge your C expressions to assembler operands. They are necessary because the compiler doesn't know what kind of operands are valid for what instructions. In fact, for all the compiler cares, operands aren't even required to be operands to any instructions: you could use them in comments or instruction names, or *not use them at all*, as long as the output string is valid to the assembler.

For a valid example, in "`imul %0, %1, %2`":

- the first operand **has** to be a register
- the second operand may be a register or a memory address
- the last operand **has** to be a constant integer value

The constraint string for each operand must communicate these requirements to GCC. For instance, it will ensure that the destination value lives in a register that can be used at the point of this statement.

GCC defines many types of constraints, but on 2019's desktop/mobile platforms, those are the constraints that are the most likely to be used:

- `r` specifies that the operand must be a general-purpose register
- `m` specifies that the operand must be a memory address
- `i` specifies that the operand must be an integer constant
- `g` specifies that the operand must be a general-purpose register, or a memory address, or an integer constant (effectively the same as "rmi")

As implied by the `g` constraint, it is possible to specify multiple constraints for each argument. By specifying multiple constraints, you allow the compiler to pick the operand kind that suits it best when the same instruction has multiple forms. This is useful on x86 and not so much on ARM, because x86 overloads mnemonics with many operand types.

For example, in this code:

```
int add(int a, int b) {
    asm("addl %1, %0" : "+r" (a) : "rm" (b));
    return a;
}
```

This is one way the compiler could choose to satisfy the `r` constraint:

```
add:
    // Per x86_64 System V calling convention:
    // * a is held in edi.
    // * b is held in esi.
    // The return value will be held in eax.

    // The compiler chooses to move `a` to eax before
    // the add (it could arbitrarily do it after).
    movl %edi, %eax

    // `b` does not need to be moved anywhere, it is
    // already in a register.

    // The compiler can emit the addition.
    addl %esi, %eax
```

```
            // The result of the addition is returned.
            ret
```

Note that when it comes to `i`, the satisfiability of the constraint may depend on your optimization levels. Passing an integer literal or an enum value always works, but when it comes to variables, it depends on the compiler's ability to fold constants. For instance, this will work at `-O1` and above, but not `-O0`, because the compiler needs to establish that `x` has a constant value:

```
int x = 3;
asm("int %0" :: "i" (x) : "memory");
// [godbolt] produces "int 3" at -O1 and above;
// [godbolt] errors out at -O0
```

GCC documents the full list of platform-independent constraints, as well as the full list of platform-specific constraints.

Outputs

Outputs specify **lvalues** where the result should be stored at the end of the operation. As a refresher, the concept of lvalue in C is nebulous, but something that is assignable is usually an lvalue (variables, dereferenced pointers, array subscripts, structure fields, etc). Most lvalues are accepted as operands, as long as the constraint can be respected: for instance, it's possible to pass a bitfield with `r` (register), but not with `m` (memory) as you cannot take the address of a bitfield. (The same applies to Clang's vector types).

In addition, the constraint string of an output **must** be prefixed with either `=` or `+`.

- `+` means that the output is actually a read-write value. The operand initially has the value contained by the expression. It's fine to read from this output operand at any point in the assembly string.

- `=&` means that the output is an *early-clobber output*. Its initial value is unspecified. It is not a bug to read from an `=&` operand once it has been assigned a value.
- `=` means that the output is write-only. The compiler can choose to give an `=` output the same location as an input: for that reason, it is *usually* a bug to write to it before the last instruction of your assembly snippet.
- `=@cc`*`COND`* is a special case of `=` that allows you to query the result of a condition code at the end of your assembly statement. You cannot reference a condition output in your assembly template.

For instance, almost all x86 instructions read and write to their first operand: you need to use the `+` prefix to communicate that. A simple example would be:

```
asm("addl %1, %0" : "+rm" (foo) : "g" (bar))
```

The initial value of `foo` is important here, because it's what `bar` will be added to. (The whole constraint string additionally specifies that `foo` may be referenced in the assembler string as a register or a memory address, since x86 has instruction forms for both.)

On the other hand, almost no ARM instruction uses the initial value of its destination register. In those cases, you would use `=` to communicate that. The ARM equivalent to the above would be:

```
asm("add %0, %1, %2" : "=r" (foo) : "r" (foo), "r" (bar))
```

(ARM only supports computations on registers, so all the operands are register operands.)

When you use `=` for an output, the compiler may reuse a location that was used as an input. This can lead to incorrect code generation if you also read from that output. For instance, in this assembly code:

```
asm("movl $123, %0 \n  "
    "addl %1, %0"
```

```
          : "=&r"(foo)
          : "r"(bar));
```

The `=&` constraint is necessary: the `mov` that writes to `%0` isn't the last instruction of the sequence, and this `asm` statement has inputs. If we used `=`, the compiler could have selected the same register for `%1` and `%0`! Note that this works and it is safe if your statement has a single instruction, such as in the ARM case above. However, with our two-instruction example, instead of doing the equivalent of `foo = bar + 123`, this code could now be doing `"movl $123, %eax; addl %eax, eax"`, which is just `foo = 246`. You solve this problem by using `=&`, which tells the compiler that it can't use the same location as it used for an input.

Condition code outputs

`=@cc`*`COND`* is a special case of `=` that allows you to get the value of a condition code at the end of your assembly code. You must replace *`COND`* with the architecture-dependent name of the condition code that you want to query. For x86, the entire list of possible conditions can be found in the `setcc` documentation. For instance, `=@ccnz` will fill your output with the result of the `setnz` instruction (true if the result is non-zero, false otherwise). You cannot use a condition code operand in your assembly string, even as it contributes to the numbering of operands. As a concrete example (of different flags):

```
// [godbolt]
asm("subq %3, %2"
    : "=@ccc"(*carry), "=@cco"(*overflow)
    : "g"(left), "g"(right));
```

The carry flag takes slot 0, and the overflow takes slot 1, even though they cannot be referenced.

Inputs

Inputs can be any value, as long as it makes sense for the constraint. For instance, to use the `i` constraint, the

compiler must be able to find that the value is a constant. They are not prefixed with anything.

Instead of passing a constraint, you may pass the index of an output constraint (optionally prefixed with %). When you do so, you tell the compiler to use the same location for that input and that output. This is not valid for + outputs: the compiler will emit a diagnostic. In fact, you may think of a + output as a shorthand for "="(foo) as an output and "0"(foo) as an input. Using = or =& in that case makes no difference.

As such, these two examples have the same result:

```
asm("xorq %0, %0" : "+r"(foo));
asm("xorq %1, %0" : "=r"(foo) : "0"(foo));
```

The use of volatile

IMPORTANT!

- If the compiler determines that the inputs of your asm statement never change, **it may move the asm statement** (out of a loop, for instance).
- If the compiler determines that the outputs of your asm statement are not used, **it may REMOVE the asm statement.**

There are a few ways to control how aggressively the compiler optimizes asm statements and the code around it. The best way is to ensure that each output is properly communicated, such that you can benefit from the compiler's dead code elimination to remove your assembly statement when it is actually not needed. This would happen, for instance, if you had an assert that checked a condition code filled in by an asm statement, and compiling out the assert in a release build caused the asm statement to no longer be doing anything useful.

This is great when you can tell the compiler about each output, but it's possible that your assembly statement

> modifies a value in a way that you can't express. For instance, you could modify the value that is pointed to by one of your inputs. If you modify *some* memory which is not directly related to an output, you may use the `"memory"` clobber parameter, as explained below. This will prevent the compiler from moving your `asm` statement before or after other memory accesses, while also not preventing the compiler from optimizing local variables into registers.
>
> Finally, it's possible that your assembly code has side effects that cannot be encoded as outputs, and that aren't really about memory either. For instance, there are many system calls that do not modify memory, so you may well use a syscall instruction without specifying a memory clobber. In that case, you would specify `volatile` so that the instruction isn't removed if it is found that its outputs aren't used.
>
> ```
> asm volatile("syscall" : "=a"(ret_val) :: "rcx", "r11");
> ```
>
> `Volatile` is implied for `asm` statements without outputs.
>
> Note that this prevents the `asm` statement from being entirely removed, but it does not prevent it from being moved around. Using the memory clobber and producing accurate input/output dependencies is still necessary to get correct results.

Clobbers

> Clobbers are the list of writable locations that the assembly code might have modified, and which have not been specified in outputs. These can be:
>
> - Register names (on x86, both `register` and `%register` are accepted, such as `rax` or `%rax`)
> - The special name `cc`, which specifies that the assembly altered condition flags. On platforms that keep multiple sets of condition flags as separate registers, it's also

possible to name indvidual registers: for instance, on PowerPC, you can specify that you clobber `cr0`.
- The special name `memory`, which specifies that the assembly wrote to memory that is not explicitly referenced by an output (for instance, by dereferencing an input pointer). A `memory` clobber prevents the compiler from reordering memory operations across the `asm` statement (although it *does not* prevent the processor from doing it: you need to use an actual memory fence to achieve this).

The compiler will helpfully generate a diagnostic if you attempt to use an unknown clobber location.

Some architectures implicitly clobber some registers on any `asm` statement, with no way of opting out. One relevant example is that on x86, the flags register is always clobbered. (A previous version of this document recommended that all `asm` statements for x86 explicitly clobber `cc`.) The compiler will not emit a diagnostic if you explicitly clobber a register that is also implicitly clobbered. There does not appear to be a documented list of per-architecture adjustments to clobbers: the source informs that at the time of writing (October 2019), 7 architectures do it (CRIS, x86, MN103, NDS32, PDP11, IBM RS/6000, Visium), but this author is not familiar enough with most of them to tell what's going on.

As an example of clobbering, the `rdtscp` instruction takes no operand and modifies `rax`, `rdx` and `rcx` implicitly. Like `rdtsc`, it writes the processor's time-stamp counter to `rax` and `rdx`, but it also writes the processor ID to `rcx`. Suppose that you're just after the processor ID: you would have to put `rax` and `rdx` in the clobber list, since they are modified and they are not specified as outputs.

```
// [godbolt]
asm(
    "rdtscp \n  "
    "movq %%rcx, %0\n  "
    : "=rm" (cpu_id)
```

```
        : /* no inputs */
        : "rax", "rcx", "rdx")
```

Note that it's possible (and most likely preferable) to avoid the mov from rcx using x86-specific register constraints, as explained in the examples section.

Labels and the goto keyword

It's always possible to branch within your asm code. With a little bit of extra effort, it's also possible to branch *out* of your asm statement, to labels that are available to your enclosing C function. To achieve this, you use asm goto.

When you use asm goto, it becomes impossible to specify outputs (this quirk is due to fairly fundamental decisions in the internal code representation of GCC), and you become able to specify labels as a fourth kind of parameter in your asm statement. Label arguments do not have constraints and cannot be named: they must be referred to by %N. For example:

```
// [godbolt]
int add_overflows(long lhs, long rhs) {
    asm goto(
        "movq %%rax, %[left]\n  "
        "addq %[right], %%rax\n  "
        "jo %2"
        : // can't have outputs
        : [left] "g" (lhs), [right] "g" (rhs)
        : "rax"
        : on_overflow);
    return 0; // no overflow
    on_overflow: return 1; // had an overflow
}
```

(This example's functionality can arguably be replicated with a flag output operand. See Outputs above.)

Labels can only be C labels: they can't be other code addresses, such as functions or an indirect goto label variable.

Note that an `asm goto` statement is always implicitly `volatile`.

Fancy examples

All examples are x86_64. This section additionally uses x86-specific constraints that bind to named registers:

- `a`, `b`, `c`, `d`: the `a`, `b`, `c`, `d` registers, respectively (the "a" register being contextually `al`, `ax`, `eax` or `rax`, for example)
- `D`: the `di` register
- `S`: the `si` register

As a matter of style and correctness, you typically get the best results if you succeed at using as few instructions as possible in your `asm` statement. If you can make the compiler do all the moving using constraints, it will be able to generate better code around your `asm` statement. All of the following examples specify a single instruction, and tell the compiler how to arrange registers to make it work.

Rotate left

This example left-rotates a 64-bit integer. The encoding of `rol` is tricky, requiring the count operand to either be an immediate value, or (very specifically) the `cl` register.

```
[godbolt]
int rotate_left(unsigned long long value, unsigned char count) {
    asm("rolq %[count], %0"
        : "+a" (value)
        : [count] "ci" (count));
    return value;
}
```

Using the `ci` constraint, the compiler is able to use a constant value when it is available, or else fall back on the `cl` register.

Do a two-word multiplication

Multiply two 64-bit integers and return the 128-bit result (by address). Here we use the x86-specific `a` and `d` constraints, which specify "the `rax` register" and "the `rdx` register", respectively. The variant of the register is chosen by the compiler to be size-appropriate; 32-bit values get `eax`, for instance.

This example is interesting because it has implicitly-referenced outputs. Using the `=d` output constraint to bind `*hi`, we cause the compiler to move the value of `rdx` to `*hi` at the end of the assembly code, even though it is not explicitly used by the assembly string.

```
// [godbolt]
void imul128(
        int64_t left, int64_t right,
        int64_t* lo, int64_t* hi) {
    asm(
        "imulq %[rhs]"
        : "=a" (*lo), "=d" (*hi)
        : [lhs] "0" (left), [rhs] "rm" (right));
}
```

Another reasonable option would have been to set `*lo = left` in C and use the `+a` constraint. This method is demonstrated below.

Call the Linux `write` system call

By binding inputs to registers using x86-specific constraints, we let the compiler do any necessary lifting to move argument values to the syscall argument registers. On x86_64 Linux, in the normal case, no moving at all is required for the first 4 arguments.

The operating system uses the value of `rax` to determine which system call is being invoked.

```
// [godbolt]
int do_write(int fp, void* ptr, size_t size) {
        int rax = SYS_write;
        asm volatile(
```

```
                    "syscall"
                    : "+a"(rax)
                    : "D" (fp), "S" (ptr), "d" (size)
                    : "rcx", "r11");
                return rax;
        }
```

Here, we use `+a` for `rax`, as `syscall` implicitly reads from it (through the operating system's behavior).