Inline Assembly

The idea behind **Inline Assembly** is to embed assembler instructions in your C/C++ code, using the `asm` keyword, when there's no option but to use Assembly language.

Contents

Overview

Syntax
- Assembler Template
- Output Operands
- Input Operands
- Clobbered Registers List
- Wildcards: How you can let the compiler choose

Using C99

Assigning Labels

asm goto

Intel Syntax

See Also
- Articles
- Forum Threads
- External

Overview

Sometimes, even though C/C++ is your language of choice, you **need** to use some assembler code in your operating system. Be it because of extreme optimization needs or because the code you're implementing is highly hardware-specific (like, say, outputting data through a port), the result is the same: there's no way around it. You must use assembly.

One of the options you have is writing an assembly function and calling it, however there can be times when even the "call" overhead is too much for you. In that case, what you need is inline assembly, which means inserting arbitrary assembly snippets in the middle of your code, using the `asm()` keyword. The way this keyword works is compiler-specific. This article describes the way it works in GCC since it is by far the most used compiler in the OS world.

Syntax

This is the syntax for using the `asm()` keyword in your C/C++ code:

```
asm ( assembler template
    : output operands                   (optional)
    : input operands                    (optional)
    : clobbered registers list          (optional)
    );
```

Assembler template is basically GAS-compatible code, except when you have constraints, in which case register names must start with %% instead of %. This means that the following two lines of code will both move the contents of the `eax` register into `ebx`:

```
asm ("movl %eax, %ebx");
asm ("movl %%eax, %%ebx" : );
```

Now, you may wonder why this %% comes in. This is where an interesting feature of inline assembly comes in: you can make use of some of your C variables in your assembly code. And since, in order to make implementation of this mechanism simpler, GCC names these variables %0, %1, and so on in your assembly code, starting from the first variable mentioned in the input/output operand sections. You're required to use this %% syntax in order to help GCC differentiate between registers and parameters.

How exactly operands work will be explained in more details in later sections. For now, it is sufficient to say that if you write something like that:

```
int a=10, b;
asm ("movl %1, %%eax;
      movl %%eax, %0;"
     :"=r"(b)        /* output */
     :"r"(a)         /* input */
     :"%eax"         /* clobbered register */
     );
```

then you've managed to copy the value of "a" into "b" using assembly code, effectively using some C variables in your assembly code. Congratulations!

The last "clobbered register" section is used in order to tell GCC that your code is using some of the processor's registers, and that it should move any active data from the running program out of this register before executing the asm snippet. In the example above, we move `a` to eax in the first instruction, effectively erasing its content, so we need to ask GCC to clear this register from unsaved data before operation.

Assembler Template

The Assembler Template defines the assembler instructions to inline. The default is to use AT&T syntax here. If you want to use Intel syntax, `-masm=intel` should be specified as a command-line option.

As an example, to halt the CPU, you just have to use the following command:

```
asm( "hlt" );
```

Output Operands

The Output Operands section is used in order to tell the compiler / assembler how it should handle C variables used to store some output from the ASM code. The Output Operands are a list of pairs, each operand consisting of a string literal, known as "constraint", stating where the C variable should be mapped (registers are generally used for optimal performance), and a C variable to map to (in parentheses).

In the constraint, 'a' refers to EAX, 'b' to EBX, 'c' to ECX, 'd' to EDX, 'S' to ESI, and 'D' to EDI (read the GCC manual for a full list), assuming that you are coding for the IA32 architecture. An equation sign indicates that your assembly code does not care about the initial value of the mapped variable (which allows some optimization). With all that in mind, it's now pretty clear that the following code sets EAX = 0.

```
int EAX;
asm( "movl $0, %0"
   : "=a" (EAX)
    );
```

Notice that the compiler enumerates the operand starting with %0, and that you don't have to add a register to the clobbered register list if it's used to store an output operand. GCC is smart enough to figure out what to do all by itself.

Starting with GCC 3.1, you can use more readable labels instead of the error-prone enumeration:

```
int current_task;
asm( "str %[output]"
   : [output] "=r" (current_task)
    );
```

These labels are in a namespace of their own, and will not collide with any C identifiers. The same can be done for input operands, too.

Input Operands

While the Output Operands are generally used for... well... output, the Input Operands allows to parametrize the ASM code; i.e., passing read-only parameters from C code to ASM block. Again, string literals are used to specify the details.

If you want to move some value to EAX, you can do it the following way (even though it would certainly be pretty useless to do so instead of directly mapping the value to EAX):

```
int randomness = 4;
asm( "movl %0, %%eax"
   :
   : "b" (randomness)
   : "eax"
    );
```

Note that GCC will always assume that input operands are read-only (unchanged). The correct thing to do when input operands are written to is to list them as outputs, but without using the equation sign because this time their original value matters. Here is a simple example:

```
asm("mov %%eax,%%ebx": : "a" (amount)); // useless but it gets the idea
```

Eax will contain "amount" and be moved into ebx.

Clobbered Registers List

It is important to remember one thing: *The C/C++ compiler knows nothing about Assembler*. For the compiler, the asm statement is opaque, and if you did not specify any output, it might even come to the conclusion that it's a no-op and optimize it away. Some third-party docs indicate that using asm volatile will cause the keyword to not be moved. However, according to the GCC documentation, *The volatile keyword indicates that the instruction has important side-effects. GCC will not delete a volatile asm if it is reachable.*, which only indicates that it will not be deleted (i.e. whether it may still be moved is an unanswered question). An approach that should work is to use asm (volatile) and put **memory** in the clobber registers, like so:

```
__asm__("cli": : :"memory"); // Will cause the statement not to be moved, but it may be optimized away.
__asm__ __volatile__("cli": : :"memory"); // Will cause the statement not to be moved nor optimized away.
```

Since the compiler uses CPU registers for internal optimization of your C/C++ variables, and doesn't know about ASM opcodes, you have to warn it about any registers that might get clobbered as a side effect, so the compiler can save their contents before making your ASM call.

The Clobbered Registers List is a comma-separated list of register names, as string literals.

Wildcards: How you can let the compiler choose

You don't need to tell the compiler which specific register it should use in each operation, and in general, except you have good reasons to prefer one register specifically, you should better let the compiler decide for you.

Forcing to use EAX over any other register, for instance, may force the compiler to issue code that will save what was previously in eax in some other register or may introduce unwanted dependencies between operations (pipeline optimization broken)

The 'wildcards' constraints allows you to give more freedom to GCC when it comes to input/output mapping:

The "g" constraint : `"movl $0, %0" : "=g" (x)`	x can be whatever the compiler prefers: a register, a memory reference. It could even be a literal constant in another context.
The "r" constraint : `"movl %%es, %0" : "=r" (x)`	you want x to go through a register. If x wasn't optimized as a register, the compiler will then move it to the place it should be. This means that `"movl %0, %%es" : : "r" (0x38)` is enough to load a segment register.
The "N" constraint : `"outl %0, %1" : : "a" (0xFE), "N" (0x21)`	tells the value '0x21' can be used as a constant in the out or in operation if ranging from 0 to 255

There are of course a lot more constraints you can put on the operand selection, machine-dependent or not, which are listed in GCC's manual (see [1] (http://gcc.gnu.org/onlinedocs/gcc-4.4.4/gcc/Simple-Constraints.html#Simple-Constraints), [2] (http://gcc.gnu.org/onlinedocs/gcc-4.

4.4/gcc/Modifiers.html#Modifiers), [3] (http://gcc.gnu.org/onlinedocs/gcc-4.4.4/gcc/Multi_002dAlternative.html#Multi_002dAlternative), and [4] (http://gcc.gnu.org/onlinedocs/gcc-4.4.4/gcc/Machine-Constraints.html#Machine-Constraints)).

Using C99

`asm` is not a keyword when using `gcc -std=c99`. Simply use `gcc -std=gnu99` to use C99 with GNU extensions. Alternatively, you can use `__asm__` as an alternate keyword that works even when the compiler strictly adheres to the standard.

Assigning Labels

It is possible to assign so-called ASM labels to C/C++ keywords. You can do this by using the `asm` command on variable definitions, as seen in this example:

```
int some_obscure_name asm("param") = 5; // "param" will be accessible in inline Assembly.

void foo()
{
   asm("mov param, %%eax");
}
```

Here's an example of how you can access these variables if you don't explicitly state a name:

```
int some_obscure_name = 5;

void foo()
{
   asm("mov some_obscure_name, %%eax");
}
```

Note that you might also be obliged to use **_some_obscure_name** (with a leading underscore), depending on your linkage options.

asm goto

Before GCC 4.5, jumping across inline assembly blocks is not supported. The compiler has no way of keeping track of what's going on, so incorrect code is almost guaranteed to be generated.
You might have been told that "gotos are evil". If you believe that is so, then asm gotos are your worst nightmare coming true. However, they do offer some interesting code optimization options.

asm goto's are not well documented, but their syntax is as follows:

```
 asm goto( "jmp %l[labelname]" : /* no outputs */ : /* inputs */ : "memory" /* clobbers */ : labelname /* any labels used */ );
```

One example where this can be useful, is the CMPXCHG instruction (see Compare and Swap (https://en.wikipedia.org/wiki/Compare-and-swap)), which the Linux kernel source code defines as follows:

```
/* TODO: You should use modern GCC atomic instruction builtins instead of this. */
#include <stdint.h>
#define cmpxchg( ptr, _old, _new ) { \
  volatile uint32_t *__ptr = (volatile uint32_t *)(ptr);   \
  uint32_t __ret;                                     \
  asm volatile( "lock; cmpxchgl %2,%1"           \
    : "=a" (__ret), "+m" (*__ptr)                \
    : "r" (_new), "0" (_old)                     \
    : "memory");                \
  );                                             \
  __ret;                                         \
}
```

In addition to returning the current value in EAX, CMPXCHG sets the zero flag (Z) when successful. Without asm gotos, your code will have to check the returned value; this CMP instruction can be avoided as follows:

```
/* TODO: You should use modern GCC atomic instruction builtins instead of this. */
// Works for both 32 and 64 bit
#include <stdint.h>
#define cmpxchg( ptr, _old, _new, fail_label ) { \
  volatile uint32_t *__ptr = (volatile uint32_t *)(ptr);   \
  asm goto( "lock; cmpxchg %1,%0 \t\n"           \
    "jnz %l[" #fail_label "] \t\n"               \
    : /* empty */                                \
    : "m" (*__ptr), "r" (_new), "a" (_old)       \
    : "memory", "cc"                             \
    : fail_label );                              \
}
```

This new macro could then be used as follows:

```
struct Item {
  volatile struct Item* next;
};

volatile struct Item *head;

void addItem( struct Item *i ) {
  volatile struct Item *oldHead;
again:
  oldHead = head;
  i->next = oldHead;
  cmpxchg( &head, oldHead, i, again );
}
```

Intel Syntax

You can let GCC use intel syntax by enabling it in inline Assembly, like so:

```
asm(".intel_syntax noprefix");
asm("mov eax, ebx");
```

Similarly, you can switch back to AT&T syntax by using the following snippet:

```
asm(".att_syntax prefix");
asm("mov %ebx, %eax");
```

This way you can combine Intel syntax and AT&T syntax inline Assembly. Note that once you trigger one of these syntax types, everything below the command in the source file will be assembled using this syntax, so don't forget to switch back when necessary, or you might get lots of compile errors!

There is also a command-line option `-masm=intel` to globally trigger Intel syntax.

GCC also supports multiple assembler dialects by wrapping text in braces separated by a vertical bar. The two examples above can be combined as follows, where the first version will be emitted when `-masm=att` is used, or no `-masm` is provided, and the second will be used when `-masm=intel` is provided.

```
asm("{mov %ebx, %eax|mov eax, ebx}");
```

See Also

Articles

- Inline Assembly/Examples - useful and commonly used functions

Forum Threads

- asm volatile being moved (http://forum.osdev.org/viewtopic.php?f=11&t=24168&p=196655&hilit=asm+volatile+moved)

External

- GCC Manuals (http://gcc.gnu.org/onlinedocs/)
- Inline assembly for x86 in Linux (by IBM) (http://web.archive.org/web/20041210030000/http://www-106.ibm.com/developerworks/library/l-ia.html)
- Visual C++ Compiler Intrinsics (http://msdn.microsoft.com/en-us/library/26td21ds(VS.80).aspx)