REF:

http://ericw.ca/notes/a-tiny-guide-to-gcc-inline-assembly.html

**A Tiny Guide to GCC Inline Assembly 12 Jun 2008**

So I’m working on a real-time operating system for school, and in the process I’ve needed to write a ton of IA32 inline assembly. GCC’s inline assembly syntax isn’t immediately straightforward so it’s been an interesting process of trial, error, and documentation to piece together the specifics. This guide presents my accumulated knowledge on the subject.

***Assembly Syntax***

GCC uses AT&T assembly syntax. The highlights:

* **instruction source, destination**  
  The first operand is the source, the second is the destination.
* **%register**  
  Register names are prefixed with a percent sign. (Or a %% in certain circumstances; see the second on operands below.)
* **$literal**  
  Literal values are prefixed with a dollar sign. The literal $10 specifies decimal 10 while $0x10 specifies hexadecimal 16.
* **instruction{b,w,l}**  
  The instruction suffix denotes the operand size. The b, w, and l specify byte (8-bit), word (16-bit), and long word (32-bit) memory references. (Always include the size! If you omit it the GNU assembler will attempt to guess for you which is usually a Bad Idea.)
* **segment:offset(base, index, scale)**  
  Memory access syntax. Note that the offset and scale constants are *not* prefixed with $ but the register references still need a %.
* **ljmp/lcall $segment, $offset**  
  Control transfer instructions may be prefixed with an l to indicate a far jump to another code segment. (Similarly, there is lret $stackadjust.)
* **\*branch-address**  
  Branch addressing using literals or registers is prefixed with an asterisk.

Here are a few examples of valid code that illustrate these points.

pushl %eax

movl $8, %ebx

movb $0x11, %al

movl %es:16(%ebx, %edi, 4), %eax

ret \*100

jmp \*%ecx

lcall $0x10, farcalllabel

***Inline Syntax***

The basic format for GCC inline assembly is as follows.

\_\_asm\_\_

\_\_volatile\_\_ /\* optional \*/

(

assembly code

: output operands /\* optional \*/

: input operands /\* optional \*/

: clobber list /\* optional \*/

);

For example, below is code to turn on bit 1 in flag then store the value in new\_flag.

int flag, new\_flag;

\_\_asm\_\_

(

"movl %1, %%eax \n"

"orw $2, %%ax \n"

"movl %%ax, %0 \n"

: "=r"(new\_flag) /\* output \*/

: "r"(flag) /\* input \*/

: "%eax" /\* clobbered register \*/

);

***Preamble***

The \_\_asm\_\_ keyword marks the start of the inline assembly statement. While using asm without the underscores is also valid in some contexts, it will not compile with the -std=c99 option. Moreover, the underscores prevent conflicts with asm defined elsewhere in your code.

The optional \_\_volatile\_\_ keyword indicates the assembly code has important side-effects and guarantees GCC will not delete it if it is reachable. It does not, however, guarantee that the assembly code will not be moved relative to other code.

***Code***

The assembly code specifies the instructions to execute. Each instruction (or label) is enclosed within double quotes and terminated by a newline.

***Operands***

The general pattern for an operand is "constraint"(expression) and multiple operands are separated by commas.

In the assembly code each operand is reference by number, where %0 is the first output operand, %1 is the second, and so on, and %N-1 is the last input operand. Because the operands are indicated by a percent sign the register names must now be prefixed with two percent signs, like %%eax.

C expressions provide the input and output operands for the assembly code. An output expression (an lvalue) specifies where a result should be stored. An input expression specifies either a location (lvalue) or value (rvalue) as input to the code.

Constraints help to decided the addressing mode and registers used for the input and output operands. Of the many constraints available, only a few are used frequently. These we discuss below.

* **m**: The operand is stored in memory, at any memory address. (Instructions will operate on the data directly in memory.)
* **r**: The operand is stored in a general-purpose register. (GCC generates code to transfer the operand to or from memory and the register it chooses.)
* **i**: The operand is an immediate integer.
* **0,…,9**: The operand matches the operand with the specified number. (GCC will use the same variable for both operands. The two operands that match must be one input-only operand and one output-only operand.)

Constraints may also have modifiers which provide additional control over the behavior of the operands. Three common constraints are:

* **=**: Operand is write-only
* **+**: Operand is both read and written
* **&**: Operand is clobbered early (i.e., is modified before the instruction is finished using the input operands, meaning it may not lie in a register used as an input operand or any part of memory)

***Clobber List***

The clobber list should contain:

* The registers modified, either explicitly or implicitly, by your code.
* If your code modifies the condition code register, “cc”.
* If your code modifies memory, “memory”.

The clobber list informs GCC of the state potentially changed by your code so it won’t make incorrect assumptions about the state and break things (always a Bad Thing).

***Examples***

To further illustrate all the stuff stuffed into this guide, I’ve pulled a few examples from my operating system.

To load the interrupt descriptor table register:

void set\_idt (idt\_pointer\_t \*ptr) {

\_\_asm\_\_ \_\_volatile\_\_ (

"lidt %0 \n" : : "m"(ptr) );

}

To set the kernel code segment:

void set\_kcs () {

\_\_asm\_\_ \_\_volatile\_\_ (

"ljmp %0, $farjmp \n"

"farjmp: \n"

"nop \n"

: "i"(KERNEL\_SEG\_CODE) );

}

To move bytes:

void kcopy (unsigned int src, unsigned int dst,

unsigned int nbytes) {

\_\_asm\_\_ \_\_volatile\_\_ (

"cld \n"

"rep \n"

"movsb \n"

:

: "S"(src), "D"(dst), "c"(nbytes)

: "%esi", "%edi", "%ecx" );

}

***References***

I pulled this information from a variety of sources, chief among them:

* [GCC Extended Asm manual](http://gcc.gnu.org/onlinedocs/gcc-4.3.0/gcc/Extended-Asm.html#Extended-Asm)
* [GNU as manual](http://sourceware.org/binutils/docs-2.18/as/index.html)
* [GCC-Inline-Assemby-HOWTO](http://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html)

ericw.ca / Content © 2009-2021 Eric Woroshow. All rights reserved.

REF:

<https://stackoverflow.com/questions/41949570/gcc-inline-assembly-to-binary>

In C or C++ you can implement inline assembly instructions by doing the following:

asm("assembly code");

or

\_\_asm\_\_ ("assembly code");

Example:

asm("movl %ebx, %eax"); /\* moves the contents of ebx register to eax \*/

\_\_asm\_\_("movb %ch, (%ebx)"); /\* moves the byte from ch to the memory pointed by ebx \*/

And then you can generate an interaction to exchange values between your assembly code, and your C/C++ variables: More on that [here](https://www.codeproject.com/Articles/15971/Using-Inline-Assembly-in-C-C).

My question is: Is each inline assembly instruction directly translated to its binary counterpart by the compiler? Or is this just a sort of "Emulation"? What this is all amounted to, is that I wanted to know if you are actually accessing the processor's registers or all the data handled within the code is stored in the stack, emulating assembly instructions. Sorry if the question is dumb.

1 Answer

When you use the **'\_\_asm\_\_'** or the **'asm'** keywords, GCC/G++ parses the string into assembly instructions. If you use **'extended assembly'**, that is when you include the colons after your string so as to make your variables interact with the **'input/output'** of the registers, the inline assembler does simple substitution, just like in the following example:

\_\_asm\_\_ ("\  
add %ebx,%eax;\  
mov %ebx,%ecx;\  
" : "=c"(output\_variable) : "b"(first\_operand),"a"(second\_operand) : /\*Nothing needed here\*/);

*Note the backslash at the end of each assembly line. It is to escape the line jump so as to use one single pair of quotes to englobe all the assembly code. Also, in order to help the inline assembly parser, each assembly instruction must end with a semicolon*

After the instruction/s in between the double quotes, you can add three colons, between the first two, you declare the variables that are going to receive the output. Note the **"=c"(output\_variable)** means that the value stored in the **ecx** register at the end, will be forwarded to the variable between **()**.

Between the last two colons, you declare the variables/values that are going to serve as an input, so at the begining of the code, the value between parenthesis is assigned to the register represented with the letter between quotes (use comas for more than one input).

After the last colon, goes a list of clobbered registers. However this is optional and useless in this particular case. More about the whole GCC Inline Assembly [here](https://www.codeproject.com/Articles/15971/Using-Inline-Assembly-in-C-C).

So, in summary, the execution goes like this:

* Substitution of input values, if any, into the corresponding registers
* Translation from assembly into binary
* Load output into the assigned variables, if any

So the short answer would be **yes**, once the code is compiled into machine language, the conversion will be 1 on 1, just like a common assembler, and thus it will be using the real registers.