Using Inline Assembly With gcc

Clark L. Coleman (plagiarist/researcher)

1.0 Overview

This is a compilation in FrameMaker of three public domain documents written by others. There is no original content added by myself. The three documents are:

1. A portion of the gcc info page for gcc 2.8.1, dealing with the subject of inline assembly language.
2. A tutorial by Brennan Underwood.
3. A tutorial by colin@nyx.net.

2.0 Information from the gcc info pages

2.1 General and Copyright Information

This is Info file gcc.info, produced by Makeinfo-1.55 from the input file gcc.texi.

This file documents the use and the internals of the GNU compiler.

Published by the Free Software Foundation 59 Temple Place - Suite 330 Boston, MA 02111-1307 USA

File: gcc.info, Node: Extended Asm, Next: Asm Labels, Prev: Inline, Up: C Extensions

2.2 Assembler Instructions with C Expression Operands

In an assembler instruction using 'asm', you can now specify the operands of the instruction using C expressions. This means no more guessing which registers or memory locations will contain the data you want to use.

You must specify an assembler instruction template much like what appears in a machine description, plus an operand constraint string for each operand.

For example, here is how to use the 68881's 'fsinx' instruction:

```
asm ("fsinx %1,%0" : "=f" (result) : "f" (angle));
```

Here 'angle' is the C expression for the input operand while 'result' is that of the output operand. Each has '"f"' as its operand constraint, saying that a floating point register is required. The '=' in '=f' indicates that the operand is an output; all output operands' constraints must use '='. The constraints use the same language used in the machine description (*note Constraints::.).

Each operand is described by an operand-constraint string followed by the C expression in parentheses. A colon separates the assembler template from the first output operand, and another separates the last output operand from the first input, if any. Commas separate output operands and separate inputs. The total number of operands is limited to ten or to the maximum number of operands in any instruction pattern in the machine description, whichever is greater.

If there are no output operands, and there are input operands, then there must be two consecutive colons surrounding the place where the output operands would go.

Output operand expressions must be lvalues; the compiler can check this. The input operands need not be lvalues. The compiler cannot check whether the operands have data types that are reasonable for the instruction being executed. It does not parse the assembler instruction template and does not know what it means, or whether it is valid assembler input. The extended 'asm' feature is most often used for machine instructions that the compiler itself does not know exist. If the output expression cannot be directly addressed (for example, it is a bit field), your constraint must allow a register. In that case, GNU CC will use the register as the output of the 'asm', and then store that register into the output.

The output operands must be write-only; GNU CC will assume that the values in these operands before the instruction are dead and need not be generated. Extended asm does not support input-output or read-write operands. For this reason, the constraint character '+', which indicates such an operand, may not be used.

When the assembler instruction has a read-write operand, or an operand in which only some of the bits are to be changed, you must logically split its function into two separate operands, one input operand and one write-only output operand. The connection between them is expressed by constraints which say they need to be in the same location when the instruction executes. You can use the same C expression for both operands, or different expressions. For example, here we write the (fictitious) 'combine' instruction with 'bar' as its read-only source operand and 'foo' as its read-write destination:

```
asm ("combine %2,%0"
     : "=r" (foo)
     : "0" (foo), "g" (bar));
```

The constraint '"0"' for operand 1 says that it must occupy the same location as operand 0. A digit in constraint is allowed only in an input operand, and it must refer to an output operand.

Only a digit in the constraint can guarantee that one operand will be in the same place as another. The mere fact that 'foo' is the value of both operands is not enough to guarantee that they will be in the same place in the generated assembler code. The following would not work:

```
asm ("combine %2,%0"
     : "=r" (foo)
     : "r" (foo), "g" (bar));
```

Various optimizations or reloading could cause operands 0 and 1 to be in different registers; GNU CC knows no reason not to do so. For example, the compiler might find a copy of the value of 'foo' in one register and use it for operand 1, but generate the output operand 0 in a different register (copying it afterward to 'foo''s own address). Of course, since the register for operand 1 is not even mentioned in the assembler code, the result will not work, but GNU CC can't tell that.

Some instructions clobber specific hard registers. To describe this, write a third colon after the input operands, followed by the names of the clobbered hard registers (given as strings). Here is a realistic example for the Vax:

```
asm volatile ("movc3 %0,%1,%2"
              : /* no outputs */
              : "g" (from), "g" (to), "g" (count)
            : "r0", "r1", "r2", "r3", "r4", "r5");
```

If you refer to a particular hardware register from the assembler code, then you will probably have to list the register after the third colon to tell the compiler that the register's value is modified. In many assemblers, the register names begin with '%'; to produce one '%' in the assembler code, you must write '%%' in the input.

If your assembler instruction can alter the condition code register, add 'cc' to the list of clobbered registers. GNU CC on some machines represents the condition codes as a specific hardware register; 'cc' serves to name this register. On other machines, the condition code is handled differently, and specifying 'cc' has no effect. But it is valid no matter what the machine.

If your assembler instruction modifies memory in an unpredictable fashion, add 'memory' to the list of clobbered registers. This will cause GNU CC to not keep memory values cached in registers across the assembler instruction.

You can put multiple assembler instructions together in a single 'asm' template, separated either with newlines (written as '\n') or with semicolons if the assembler allows such semicolons. The GNU assembler allows semicolons and all Unix assemblers seem to do so. The input operands are guaranteed not to use any of the clobbered registers, and neither will the output operands' addresses, so you can read and write the clobbered registers as many times as you like. Here is an example of multiple instructions in a template; it assumes that the subroutine '_foo' accepts arguments in registers 9 and 10:

```
asm ("movl %0,r9;movl %1,r10;call _foo"
        : /* no outputs */
        : "g" (from), "g" (to)
        : "r9", "r10");
```

Unless an output operand has the '&' constraint modifier, GNU CC may allocate it in the same register as an unrelated input operand, on the assumption that the inputs are consumed before the outputs are produced. This assumption may be false if the assembler code actually consists of more than one instruction. In such a case, use '&' for each output operand that may not overlap an input. *Note Modifiers::.

If you want to test the condition code produced by an assembler instruction, you must include a branch and a label in the 'asm' construct, as follows:

```
asm ("clr %0;frob %1;beq 0f;mov #1,%0;0:"
        : "g" (result)
        : "g" (input));
```

This assumes your assembler supports local labels, as the GNU assembler and most Unix assemblers do.

Speaking of labels, jumps from one 'asm' to another are not supported. The compiler's optimizers do not know about these jumps, and therefore they cannot take account of them when deciding how to optimize.

Usually the most convenient way to use these 'asm' instructions is to encapsulate them in macros that look like functions. For example,

```
#define sin(x)       \
 ({ double __value, __arg = (x);    \
    asm ("fsinx %1,%0" \
        : "=f" (__value) \
        : "f" (__arg));  \
    __value; })
```

Here the variable '__arg' is used to make sure that the instruction operates on a proper 'double' value, and to accept only those arguments 'x' which can convert automatically to a 'double'.

Another way to make sure the instruction operates on the correct data type is to use a cast in the 'asm'. This is different from using a variable '__arg' in that it converts more different types. For example, if the desired type were 'int', casting the argument to 'int' would accept a pointer with no complaint, while assigning the argument to an 'int' variable named '__arg' would warn about using a pointer unless the caller explicitly casts it.

If an 'asm' has output operands, GNU CC assumes for optimization purposes that the instruction has no side effects except to change the output operands. This does not mean that instructions with a side effect cannot be used, but you must be careful, because the compiler may eliminate them if the output operands aren't used, or move them out of loops, or replace two with one if they constitute a common subexpression. Also, if your instruction does have a side effect on a variable that otherwise appears not to change, the old value of the variable may be reused later if it happens to be found in a register.

You can prevent an ‘asm’ instruction from being deleted, moved significantly, or combined, by writing the keyword ‘volatile’ after the ‘asm’. For example:

```
#define set_priority(x)  \
 asm volatile ("set_priority %0"   \
              : /* no outputs */  \
              : "g" (x))
```

An instruction without output operands will not be deleted or moved significantly, regardless, unless it is unreachable.

Note that even a volatile ‘asm’ instruction can be moved in ways that appear insignificant to the compiler, such as across jump instructions. You can't expect a sequence of volatile ‘asm’ instructions to remain perfectly consecutive. If you want consecutive output, use a single ‘asm’.

It is a natural idea to look for a way to give access to the condition code left by the assembler instruction. However, when we attempted to implement this, we found no way to make it work reliably. The problem is that output operands might need reloading, which would result in additional following “store” instructions. On most machines, these instructions would alter the condition code before there was time to test it. This problem doesn't arise for ordinary “test” and “compare” instructions because they don't have any output operands.

If you are writing a header file that should be includable in ANSI C programs, write ‘__asm__’ instead of ‘asm’. *Note Alternate Keywords::.

3.0 Brennan's Guide to Inline Assembly

by Brennan "Bas" Underwood

Document version 1.1.2.2

3.1 Introduction

Ok. This is meant to be an introduction to inline assembly under DJGPP. DJGPP is based on GCC, so it uses the AT&T/UNIX syntax and has a somewhat unique method of inline assembly. I spent many hours figuring some of this stuff out and told Info that I hate it, many times.

Hopefully if you already know Intel syntax, the examples will be helpful to you. I've put variable names, register names and other literals in bold type.

3.2 The Syntax

So, DJGPP uses the AT&T assembly syntax. What does that mean to you?

* Register naming: Register names are prefixed with "%". To reference eax:

AT&T: %**eax** Intel: **eax**

* Source/Destination Ordering: In AT&T syntax (which is the UNIX standard, BTW) the source is always on the left, and the destination is always on the right. So let's load **ebx** with the value in **eax**:

AT&T: **movl** %**eax**, %**ebx** Intel: **mov ebx**, **eax**

* Constant value/immediate value format: You must prefix all constant/immediate values with "$". Let's load **eax** with the address of the "C" variable **booga**, which is static.

AT&T: **movl** $**_booga**, %**eax** Intel: **mov eax**, **_booga**

Now let's load **ebx** with **0xd00d**:

AT&T: **movl** $**0xd00d**, %**ebx** Intel: **mov ebx**, **d00dh**

* Operator size specification: You must suffix the instruction with one of b, w, or l to specify the width of the destination register as a byte, word or longword. If you omit this, GAS (GNU assembler) will attempt to guess. You don't want GAS to guess, and guess wrong! Don't forget it.

AT&T: **movw** %**ax**, %**bx** Intel: **mov bx**, **ax**

The equivalent forms for Intel is **byte ptr**, **word ptr**, and **dword ptr**, but that is for when you are...

* Referencing memory: DJGPP uses 386-protected mode, so you can forget all that real-mode addressing junk, including the restrictions on which register has what default segment, which registers can be base or index pointers. Now, we just get 6 general purpose registers. (if you use ebp, but be sure to restore it yourself or compile with -fomit-frame-pointer.) Here is the canonical format for 32-bit addressing:

AT&T: immed32(basepointer,indexpointer,indexscale)

Intel: [basepointer + indexpointer*indexscale + immed32]

You could think of the formula to calculate the address as:

immed32 + basepointer + indexpointer * indexscale

You don't have to use all those fields, but you do have to have at least 1 of immed32, basepointer and you MUST add the size suffix to the operator! Let's see some simple forms of memory addressing:

o Addressing a particular C variable:

AT&T: **_booga** Intel: [**_booga**]

Note: the underscore ("_") is how you get at static (global) C variables from assembler. This only works with global variables. Otherwise, you can use extended asm to have variables preloaded into registers for you. I address that farther down.

o Addressing what a register points to:

AT&T: (%**eax**) Intel: [**eax**]

o Addressing a variable offset by a value in a register:

AT&T: **_variable**(%**eax**) Intel: [**eax + _variable**]

o Addressing a value in an array of integers (scaling up by 4):

AT&T: **_array**(,%**eax**,**4**) Intel: [**eax*4 + array**]

o You can also do offsets with the immediate value:

C code: ***(p+1)** where p is a char *

AT&T: **1**(%**eax**) where **eax** has the value of **p**

Intel: [**eax + 1**]

o You can do some simple math on the immediate value:

AT&T: **_struct_pointer+8**

I assume you can do that with Intel format as well.

o Addressing a particular char in an array of 8-character records: eax holds the number of the record desired. ebx has the wanted char's offset within the record.

AT&T: **_array**(%**ebx**,%**eax**,**8**) Intel: [**ebx + eax*8 + _array**]

Whew. Hopefully that covers all the addressing you'll need to do. As a note, you can put **esp** into the address, but only as the base register.

3.3 Basic inline assembly

The format for basic inline assembly is very simple, and much like Borland's method.

asm ("statements");

Pretty simple, no? So

asm ("**nop**");

will do nothing of course, and

asm ("**cli**");

will stop interrupts, with

asm ("**sti**");

of course enabling them. You can use **__asm__** instead of **asm** if the keyword **asm** conflicts with something in your program.

When it comes to simple stuff like this, basic inline assembly is fine. You can even push your registers onto the stack, use them, and put them back.

asm ("**pushl** %**eax**\n\t" "**movl** $**0**, %**eax**\n\t" "**popl** %**eax**");

(The \n's and \t's are there so the .s file that GCC generates and hands to GAS comes out right when you've got multiple statements per **asm**.) It's really meant for issuing instructions for which there is no equivalent in C and don't touch the registers.

But if you do touch the registers, and don't fix things at the end of your **asm** statement, like so:

asm ("**movl** %**eax**, %**ebx**"); **asm** ("**xorl** %**ebx**, %**edx**"); **asm** ("**movl** $**0**, **_booga**");

then your program will probably blow things to hell. This is because GCC hasn't been told that your **asm** statement clobbered **ebx** and **edx** and **booga**, which it might have been keeping in a register, and might plan on using later. For that, you need:

3.4 Extended inline assembly

The basic format of the inline assembly stays much the same, but now gets Watcom-like extensions to allow input arguments and output arguments.

Here is the basic format:

asm (“statements” : output_registers : input_registers : clobbered_registers);

Let’s just jump straight to a nifty example, which I’ll then explain:

asm (“**cld**\n\t” “**rep**\n\t” “**stosl**” : /* no output registers */ : “c” (**count**), “a” (**fill_value**), “D” (**dest**) : “%**ecx**”, “%**edi**”);

The above stores the value in **fill_value count** times to the pointer **dest**.

Let’s look at this bit by bit.

asm (“**cld**\n\t”

We are clearing the direction bit of the flags register. You never know what this is going to be left at, and it costs you all of 1 or 2 cycles.

“**rep**\n\t” “**stosl**”

Notice that GAS requires the rep prefix to occupy a line of its own. Notice also that **stos** has the **l** suffix to make it move longwords.

: /* no output registers */

Well, there aren’t any in this function.

: “c” (**count**), “a” (**fill_value**), “D” (**dest**)

Here we load **ecx** with **count**, **eax** with **fill_value**, and **edi** with **dest**. Why make GCC do it instead of doing it ourselves? Because GCC, in its register allocating, might be able to arrange for, say, **fill_value** to already be in **eax**. If this is in a loop, it might be able to preserve **eax** thru the loop, and save a **movl** once per loop.

: “%**ecx**”, “%**edi**”);

And here’s where we specify to GCC, “you can no longer count on the values you loaded into **ecx** or **edi** to be valid.” This doesn’t mean they will be reloaded for certain. This is the *clobberlist*.

Seem funky? Well, it really helps when optimizing, when GCC can know exactly what you’re doing with the registers before and after. It folds your assembly code into the code it generates (whose rules for generation look remarkably like the above) and then optimizes. It’s even smart enough to know that if you tell it to put (**x+1**) in a register, then if you don't clobber it, and later C code refers to (**x+1**), and it was able to keep that register free, it will reuse the computation. Whew.

Here's the list of register loading codes that you'll be likely to use:

a eax b ebx c ecx d edx S esi D edi I constant value (0 to 31) q,r dynamically allocated register (see below) g eax, ebx, ecx, edx or variable in memory A eax and edx combined into a 64-bit integer (use long longs)

Note that you can't directly refer to the byte registers (**ah**, **al**, etc.) or the word registers (**ax**, **bx**, etc.) when you're loading this way. Once you've got it in there, though, you can specify **ax** or whatever all you like.

The codes have to be in quotes, and the expressions to load in have to be in parentheses.

When you do the clobber list, you specify the registers as above with the %. If you write to a variable, you must include "memory" as one of The Clobbered. This is in case you wrote to a variable that GCC thought it had in a register. This is the same as clobbering all registers. While I've never run into a problem with it, you might also want to add "**cc**" as a clobber if you change the condition codes (the bits in the flags register the **jnz**, **je**, etc. operators look at.)

Now, that's all fine and good for loading specific registers. But what if you specify, say, **ebx**, and **ecx**, and GCC can't arrange for the values to be in those registers without having to stash the previous values. It's possible to let GCC pick the register(s). You do this:

asm ("**leal** (%1,%1,4), %0" : "=r" (**x**) : "0" (**x**));

The above example multiplies **x** by 5 really quickly (1 cycle on the Pentium). Now, we could have specified, say **eax**. But unless we really need a specific register (like when using **rep movsl** or **rep stosl**, which are hardcoded to use **ecx**, **edi**, and **esi**), why not let GCC pick an available one? So when GCC generates the output code for GAS, %0 will be replaced by the register it picked.

And where did "q" and "r" come from? Well, "q" causes GCC to allocate from **eax**, **ebx**, **ecx**, and **edx**. "r" lets GCC also consider **esi** and **edi**. So make sure, if you use "r" that it would be possible to use **esi** or **edi** in that instruction. If not, use "q".

Now, you might wonder, how to determine how the %n tokens get allocated to the arguments. It's a straightforward first-come-first-served, left-to-right thing, mapping to the "q"'s and "r"'s. But if you want to reuse a register allocated with a "q" or "r", you use "0", "1", "2"... etc.

You don't need to put a GCC-allocated register on the clobberlist as GCC knows that you're messing with it.

Now for output registers.

asm (“**leal** (%1,%1,4), %0” : “=r” (**x_times_5**) : “r” (**x**));

Note the use of = to specify an output register. You just have to do it that way. If you want 1 variable to stay in 1 register for both in and out, you have to respecify the register allocated to it on the way in with the “0” type codes as mentioned above.

asm (“**leal** (%0,%0,4), %0” : “=r” (**x**) : “0” (**x**));

This also works, by the way:

asm (“**leal** (%%**ebx**,%%**ebx**,4), %%**ebx**” : “=b” (**x**) : “b” (**x**));

2 things here:

* Note that we don’t have to put **ebx** on the clobberlist, GCC knows it goes into **x**. Therefore, since it can know the value of **ebx**, it isn’t considered clobbered.

* Notice that in extended asm, you must prefix registers with %% instead of just %. Why, you ask? Because as GCC parses along for %0’s and %1’s and so on, it would interpret %**edx** as a %**e** parameter, see that that’s non-existent, and ignore it. Then it would bitch about finding a symbol named **dx**, which isn’t valid because it’s not prefixed with % and it’s not the one you meant anyway.

Important note: If your assembly statement must execute where you put it, (i.e. must not be moved out of a loop as an optimization), put the keyword **volatile** after **asm** and before the ()’s. To be ultra-careful, use

__asm__ __volatile__ (...whatever...);

However, I would like to point out that if your assembly’s only purpose is to calculate the output registers, with no other side effects, you should leave off the **volatile** keyword so your statement will be processed into GCC’s common subexpression elimination optimization.

3.5 Some useful examples

```
#define disable() __asm__ __volatile__ ("cli");
#define enable() __asm__ __volatile__ ("sti");
```

Of course, libc has these defined too.

```
#define times3(arg1, arg2) \
 __asm__ ( \
   "leal (%0,%0,2),%0" \
   : "=r" (arg2) \
   : "0" (arg1) );

#define times5(arg1, arg2) \
 __asm__ ( \
   "leal (%0,%0,4),%0" \
   : "=r" (arg2) \
   : "0" (arg1) );

#define times9(arg1, arg2) \
 __asm__ ( \
   "leal (%0,%0,8),%0" \
   : "=r" (arg2) \
   : "0" (arg1) );
```

These multiply arg1 by 3, 5, or 9 and put them in arg2. You should be ok to do:

times5(**x**,**x**);

as well.

```
#define rep_movsl(src, dest, numwords) \
 __asm__ __volatile__ ( \
   "cld\n\t" \
   "rep\n\t" \
   "movsl" \
   : : "S" (src), "D" (dest), "c" (numwords) \
   : "%ecx", "%esi", "%edi" )
```

Helpful Hint: If you say **memcpy**() with a constant length parameter, GCC will inline it to a **rep movsl** like above. But if you need a variable length version that inlines and you're always moving dwords, there ya go.

```
#define rep_stosl(value, dest, numwords) \
 __asm__ __volatile__ ( \
   "cld\n\t" \
   "rep\n\t" \
   "stosl" \
   : : "a" (value), "D" (dest), "c" (numwords) \
   : "%ecx", "%edi" )
```

Same as above but for **memset**(), which doesn't get inlined no matter what (for now.)

```
#define RDTSC(llptr) ({ \
 __asm__ __volatile__ ( \
         ".byte 0x0f; .byte 0x31" \
         : "=A" (llptr) \
         : : "eax", "edx"); })
```

Reads the TimeStampCounter on the Pentium and puts the 64 bit result into llptr.

3.6 The End

"The End"?! Yah, I guess so.

If you're wondering, I personally am a big fan of AT&T/UNIX syntax now. (It might have helped that I cut my teeth on SPARC assembly. Of course, that machine actually had a decent number of general registers.) It might seem weird to you at first, but it's really more logical than Intel format, and has no ambiguities.

If I still haven't answered a question of yours, look in the Info pages for more information, particularly on the input/output registers. You can do some funky stuff like use "A" to allocate two registers at once for 64-bit math or "m" for static memory locations, and a bunch more that aren't really used as much as "q" and "r".

Alternately, mail me, and I'll see what I can do. (If you find any errors in the above, please, e-mail me and tell me about it! It's frustrating enough to learn without buggy docs!) Or heck, mail me to say "boogabooga."

It's the least you can do.

Related Usenet posts:

* local labels * fixed point multiplies

-- Thanks to Eric J. Korpela <korpela@ssl.Berkeley.EDU> for some corrections. -- Have you seen the DJGPP2+Games Page? Probably. Page written and provided by Brennan Underwood. Page created with vi, God's own editor.

4.0 A Brief Tutorial on GCC inline asm (x86 biased)

colin@nyx.net, 20 April 1998

I am a great fan of GCC's inline asm feature, because there is no need to second-guess or outsmart the compiler. You can tell the compiler what you are doing and what you expect of it, and it can work with it and optimize your code.

However, on a convoluted processor like the x86, describing just what is going on can be quite a complex job. In the interest of a faster kernel through appropriate usage of this powerful tool, here is an introduction to its use.

4.1 Extended asm, an introduction.

In a nice clean register-register RISC architecture, accessing an occasional "foo" instruction is quite simple. You just write:

```
asm("foo %1,%2,%0"
    : "=r" (output)
    : "r" (input1), "r" (input2));
```

The part before the first colon is very much line the semi-standard asm() feature that has been in many C compilers since the K&R days. The string is pasted into the compiler's assembly output at the current location.

However, GCC is rather cleverer. What will actually appear in the output of "gcc -O -S foo.c" (a file named "foo.s") is:

```
#APP          foo r17,r5,r9 #NO_APP
```

The "#APP" and "#NO_APP" parts are instructions to the assembler that briefly put it into normal operating mode, as opposed to the special high-speed "compiler output" mode that turns off every feature that the compiler doesn't use as well as a lot of error-checking. For our purposes, it's convenient becuase it highlights the part of the code we're interested in.

Between, you will see that the "%1" and so forth have turned into registers. This is because GCC replaced "%0", "%1" and "%2" with registers holding the first three arguments after the colon.

That is, r17 holds input1, r5 holds input2, and r9 holds output.

It's perfectly legal to use more complex expressions like:

```
asm("foo %1,%2,%0"
    : "=r" (ptr->vtable[3](a,b,c)->foo.bar[baz])
    :
    : "r" (gcc(is) + really(damn->cool)), "r" (42));
```

GCC will treat this just like:

```
register int t0, t1, t2;
t1 = gcc(is) + really(damn->cool);
t2 = 42;
asm("foo %1,%2,%0"
    : "=r" (t0)
    : "r" (t1), "r" (t2));
ptr->vtable[3](a,b,c)->foo.bar[baz] = t0;
```

The general form of an asm() is

```
asm( "code" : outputs : inputs : clobbers);
```

Within the "code", %0 refers to the first argument (usually an output, unless there are no outputs), %1 to the second, and so forth. It only goes up to %9. Note that GCC prepends a tab and appends a newline to the code, so if you want to include multi-line asm (which is legal) and you want it to look nice in the asm output, you should separate lines with "\n\t". (You'll see lots of examples of this in the Linux source.) It's also legal to use ";" as a separator to put more than one asm statement on a line.

There are option letters that you can put between the % and the digit to print the operand specially; more on this later.

Each output or input in the comma-separated list has two parts, "constraints" and (value). The (value) part is pretty straightforward. It's an expression. For outputs, it must be an lvalue, i.e. something that is legal to have on the left side of an assignment.

The constraints are more interesting. All outputs must be marked with "=", which says that this operand is assigned to. I'm not sure why this is necessary, since you also have to divide up outputs and inputs with the colon, but I'm not inclined to make a fuss about it, since it's easy to do once you know.

The letters that come after that give permitted operands. There are more choices than you might think. Some depend on the processor, but there are a few that are generic.

"r", as example "rm" means a register or memory. "ri" means a register or an immediate value. "g" is "general"; it can be anything at all. It's usually equivalent to "rim", but your processor may have even more options that are included. "o" is like "m", but "offsettable", meaning that you can add a small offset to it. On the x86, all memory operands are offsettable, but some machines don't support indexing and displacement at the same time, or have something like the 680x0's autoincrement addressing mode that doesn't support a displacement.

Capital letters starting with "I" are usually assigned to immediate values in a certain range. For example, a lot of RISC machines allow either a register or a short immediate value. If our machine is like the DEC Alpha, and allows a register or a 16-bit immediate, you could write

```
asm("foo %1,%2,%0"
    : "=r" (output)
    : "r" (input1), "rI" (input2));
```

and if input2 were, say, 42, the compiler would use an immediate constant in the instruction.

The x86-specific constraints are defined later.

4.2 A few notes about inputs

An input may be a temporary copy, but it may not be. Unless you tell GCC that you are going to modify that location (described later in “equivalence constraints”), you must not alter any inputs.

GCC may, however, elect to place an output in the same register as an input if it doesn’t need the input value any more. You must not make assumptions either way. If you need to have it one way or the other, there are ways (described later) to tell GCC what you need.

The rule in GCC’s inline asm is, say what you need and then get out of the optimizer’s way.

4.3 x86 assembly code

The GNU tools used in Linux use an AT&T-developed assembly syntax that is different from the Intel-developed one that you see in a lot of example code. It’s a lot simpler, actually. It doesn’t have any of the DWORD PTR stuff that the Intel syntax requires.

The most significant difference, however, is a major one and easy to get confused by. While Intel uses “op dest,src”, AT&T syntax uses “op src,dest”. DON’T FORGET THIS. If you’re used to Intel syntax, this can take quite a while to get used to.

The easy way to know which flavour of asm syntax you’re reading is to look for all the % synbols. AT&T names the registers %eax, %ebx, etc. This avoids the need for a kludge like putting _ in front of all the function and variable names to avoid using perfectly good C names like esp. It’s easy enough to read, but don’t forget it when writing.

The other major difference is that the operand size is clear from the instruction. You don’t have just “inc”, you have “incb”, “incw” and “incl” to increment 8, 16 or 32 bits. If the size is clear from the operands, you can just write “inc”, (e.g. “inc %eax”), but if it’s a memory operand, rather than writing “inc DWORD PTR foo” you just wrote “incl foo”. “inc foo” is an error; the assembler doesn’t try to keep track of the type of anything. Writing “incl %al” is an error which the assembler catches.

Immediate values are written with a leading $. Thus, “movl foo,%eax” copies the contents of memory location foo into %eax. “movl $foo,%eax” copies the address of foo. “movl 42,%eax” is a fetch from an absolute address. “movel $42,%eax” is an immediate load.

Addressing modes are written offset(base,index,scale). You may leave out anything irrelevant. So (%ebx) is legal, as is -44(%ebx,%eax), which is equivalent to -44(%ebx,%eax,1). Legal scales are 1, 2 4 and 8.

4.4 Equivalence constraints

Sometimes, especially on two-address machines like the x86, you need to use the same register for output and for input. Although if you look into the GCC documentation, you'll see a useful-looking "+" constraint character, this isn't available to inline asm. What you have to do instead is to use a special constraint like "0":

```
asm("foo %1,%0"
    : "=r" (output)
    : "r" (input1), "0" (input2));
```

This says that input2 has to go in the same place as the output, so %2 and %0 are the same thing. (Which is why %2 isn't actually mentioned anywhere.) Note that it is perfectly legal to have different variables for input and output even though they both use the same register. GCC will do any necessary copying to temporary registers for you.

4.5 Constraints on the x86

The i386 has *lots* of register classes, designed for anything remotely useful. Common ones are defined in the "constraints" section of the GCC manual. Here are the most useful:

g - general effective address m - memory effective address r - register i - immediate value, 0..0xffffffff n - immediate value known at compile time. ("i" would allow an address known only at link time)

But there are some i386-specific ones described in the processor-specific part of the manual and in more detail in GCC's i386.h:

q - byte-addressible register (eax, ebx, ecx, edx) A - eax or edx a, b, c, d, S, D - eax, ebx, ..., esi, edi only

I - immediate 0..31 J - immediate 0..63 K - immediate 255 L - immediate 65535 M - immediate 0..3 (shifts that can be done with lea) N - immediate 0..255 (one-byte immediate value) O - immedaite 0..32

There are some more for floating-point registers, but I won't go into those. The very special cases like "K" are mostly used inside GCC in alternative code sequences, providing a special-case way to do something like ANDing with 255.

But something like "I" is useful, for example the x86 rotate left:

```
asm("roll %1,%0"
    : "=g" (result)
    : "cI" (rotate), "0" (input));
```

(See the section on “x86 assembly syntax” if you wonder why the extra “l” is on “rol”.)

4.6 Advanced constraints

In the GCC manual, constraints and so on are described in most detail in the section on writing machine descriptions for ports. GCC, not surprisingly, uses the same constaints mechanism internally to compile C code. Here’s a summary.

= has already been discussed, to mark an output. No, I don’t know why it’s needed in inline asm, but it’s not worth “fixing”.

+ is described in the gcc manual, but is not legal in inline asm. Sorry.

% says that this operand and the next one may be switched at the compiler’s convenience; the arguments are commutative. Many operations (+, *, &, |, ^) have this property, but the options permitted in the instruction set may not be as general. For example, on a RISC machine which lets the second operand be an immediate value (in the “I” range), you could specify an add instruction like:

```
asm("add %1,%2,%0"
    : "=r" (output)
    : "%r" (input1), "rI" (input2));
```

, separates a list of alternative constraints. Each input and output must have the same length list of alternatives, and one element of the list is chosen. For example, the x86 permits register-memory and memory-register operations, but not memory-memory. So an add could be written as:

```
asm("add %1,%0"
    : "=r,rm" (output)
    : "%g,ri" (input1), "0,0" (input2));
```

This says that if the output is a register, input1 may be anything, but if the output is memory, the input may only be a register or an immediate value. And input2 must be in the same place as the output, although you can swap things and place input1 there instead.

If there are multiple options listed and the compiler has no preference, it will choose the first one. Thus, if there’s a minor difference in timing or some such, list the faster one first.

? in one alternative says that an alternative is discouraged. This is important for compiler-writers who want to encourage the fastest code, but is getting pretty esoteric for inline asm.

& says that an output operand is written to before the inputs are read, so this output must not be the same register as any input. Without this, gcc may place an output and an input in the same register even if not required by a “0” constraint. This is very useful, but is mentioned here because it’s specific to an alternative. Unlike = and %, but like ?, you have to include it with each alternative to which it applies.

Note that there is no way to encode more complex information, like "this output may not be in the same place as *that* input, but may share a ragiater with that *other* input". Each output either may share a register with any input, or with none.

In inline asm, you usually specify this with every alternative, since you can't chnage the order of operations depending on the option selected. In GCC's internal code generation, there are provisions for producing different code depending on the register alternative chosen, but you can't do that with inline asm.

One place you might use it is when you have the possibility of the output overlapping with input two, but not input one. E.g.

```
asm("foo %1,%0; bar %2,%0"
     : "=r,&r" (out)
     : "r,r" (in1), "0,r" (in2));
```

This says that either in2 is in the same register as out, or nothing is. However, with more operands, the number of possibilities quickly mushrooms and GCC doesn't cope gracefully with large numbers of alternatives.

4.7 Clobbers

Sometimes an instruction knocks out certain specific registers. The most common example of this is a function call, where the called function is allowed to do whatever it likes with some registers.

If this is the case, you can list specific registers that get clobbered by an operation after the inputs. The syntax is not like constraints, you just provide a comma-separated list of registers in string form. On the 80x86, they're "ax", "bx", "si" "di", etc.

There are two special cases for clobbered values. One is "memory", meaning that this instruction writes to some memory (other than a listed output) and GCC shouldn't cache memory values in registers across this asm. An asm memcpy() implementation would need this. You do *not* need to list "memory" just because outputs are in memory; gcc understands that.

The second is "cc". It's not necessary on all machines, and I havem't figured it out for the x86 (I don't think it is), but it's always legal to specify, and means that the instructions mess up the condition codes.

Note that GCC will not use a clobbered register for inputs or outputs. GCC 2.7 would let you do it anyway, specifying an input in class "a" and saying that "ax" is clobbered. GCC 2.8 and egcs are getting pickeri, and complaining that there are no free registers in class "a" available. This is not the way to do it. If you corrput an input register, include a dummy output in the same register, the value of which is never used. E.g.

```
int dummy;
asm("munge %0"
     : "=r" (dummy)
     : "0" (input));
```

4.8 Temporary registers

People also sometimes erroneously use clobbers for temporary registers. The right way is to make up a dummy output, and use “=r” or “=&r” depending on the permitted overlap with the inputs. GCC allocates a register for the dummy value. The difference is that GCC can pick a convenient register, so it has more flexibility.

4.9 const and volatile

There are two optimization hints that you can give to an asm statement.

asm volatile(...) statements may not be deleted or significantly reordered; the volatile keyword says that they do something magic that the compiler shouldn’t play with too much.

GCC will delete ordinary asm() blocks if the outputs are not used, and will reorder them slightly to be convenient to where the outputs are. (asm blocks with no outputs are assumed to be volatile by default.)

asm const() statements are assumed to produce outputs that depend only on the inputs, and thus can be subject to common subexpression optimization and can be hoisted out of loops. The most common example of an output that does *not* depend only on an input is a pointer that is fetched. *p may change from time to time even if p does not change. Thus, an asm block that fetches from a pointer should not include a const.

An example of something that is good is a coprocessor instruction to compute sin(x). If GCC knows that two calls have the same value of x, it can compute sin(x) only once.

For example, compare:

```
int foo(int x);    {
   int i, y, total;
   total = 0;
   for (i = 0; i < 100; i++) {
      asm volatile("foo %1,%0"
                   : "=r" (y)
                   : "g" (x));
      total += y;
   }
   return total;
}
```

then try changing that to "const" after the asm. The code (on an x86) looks like:

```
func1:
    xorl %ecx,%ecx
    pushl %ebx
    movl %ecx,%edx
    movl 8(%esp),%ebx
    .align 4
.L7:
#APP
    foo %ebx,%eax
#NO_APP
    addl %eax,%ecx
    incl %edx
    cmpl $99,%edx
    jle .L7
    movl %ecx,%eax
    popl %ebx
    ret
```

which then changes to (in the const case):

```
func2:
    xorl %edx,%edx
#APP
    foo 4(%esp),%ecx
#NO_APP
    movl %edx,%eax
    .align 4
.L13:
    addl %ecx,%edx
    incl %eax
    cmpl $99,%eax
    jle .L13
    movl %edx,%eax
    ret
```

I'm still not completely thrilled with the code (why put the loop counter in %eax instead of total, which gets returned), but you can see how it improves.

4.10 Alternate keywords

__asm__() is a legal alias for asm(), and it is legal (and produces no warnings) even when in strict-ANSI mode or when warning about non-portable constructs. Otherwise, it is equivalent.

4.11 Output substitutions

Sometimes you want to include a value in an asm statement in an unusual way. For example, you could use the lea instruction to do something hairy like

```
asm("lea %1(%2,%3,1<<%4),%0"
    : "=r" (out)
    : "%i" (in1), "r" (in2), "r" (in3), "M"(logscale));
```

this looks like a way to generate a legal lea instruction with all the possible bells and whistles. There's only one problem. When GCC substitutes the immedaites "in1" and "logscale", it's going to produce something like:

```
lea $-44(%ebx,%eax,1<<$2),%ecx
```

which is a syntax error. The $ on the constants are not useful in this context. So there are modifier characters. The one applicable in this context is “c”, which means to omit the usual immediate value information. The correct asm is

```
asm("lea %c1(%2,%3,1<<%c4),%0"
     : "=r" (out)
     : "%i" (in1), "r" (in2), "r" (in3), "M"(logscale));
```

which will produce

```
lea -44(%ebx,%eax,1<<2),%ecx
```

as desired. There are a few others mentioned in the GCC manual as generic:

%c0 substitutes the immediate value %0, but without the immediate syntax. %n0 substitutes like %c0, but the negated value. %l0 substitutes lile %c0, but with the syntax expected of a jump target. (This is usually the same as %c0.)

And then there are the x86-specific ones. These are, unfortunately, only listed in the i386.h header file in the GCC source (config/i386/i386.h), so you havr to dig a bit for them.

%k0 prints the 32-bit form of an operand. %eax, etc. %w0 prints the 16-bit form of an operand. %ax, etc. %b0 prints the 8-bit form of an operand. %al, etc. %h0 prints the high 8-bit form of a register. %ah, etc. %z0 print opcode suffix coresponding to the operand type, b, w or l.

By default, when %0 prints a register in the form corresponding to the argument size. E.g. asm(“inc %0” : “=r” (out) : “0” (in)) will print as “inc %al”, “inc %ax” or “inc %eax” depending on the type of “out”.

For example, byte-swapping on a non-486:

```
asm("xchg %b0,%h0; roll $16,%0; xchg %b0,%h0"
     : "=q" (x)
     : "=" (x));
```

This says that x must be in a byte-addressible register and proceeds to swap the bytes to big-endian form.

It’s legal to use the %w and %b forms on objects that aren’t registers, it just makes no difference. Using %b and %h on non-byte addressible registers tends to make the compiler abort, so don’t do that.

%z is rather cool. For example, consider the following code:

```
#define xchg(m, in, out)       \
    asm("xchg%z0 %2,%0"   \
        : "=g" (*(m)), "=r" (out)   \
        : "1" (in))

int bar(void *m, int x)    {
     xchg((char *)m, (char)x, x);
     xchg((short *)m, (short)x, x);
     xchg((int *)m, (int)x, x);
     return x;
}
```

This produces, as assembly output,

```
.globl bar
    .type bar,@function
bar:
    movl 4(%esp),%eax
    movb 8(%esp),%dl
#APP
    xchgb %dl,(%eax)
    xchgw %dx,(%eax)
    xchgl %edx,(%eax)
#NO_APP
    movl %edx,%eax
    ret
```

(Re-using x is a way to make sure that nothing got optimized away.)

It's not really needed here because the size of the %2 register lets you get away with just "xchg", but there are situations where it's nice to have an operand size.

4.12 Extra % patterns

Some % substitutions don't specify an argument. The most common one is %%, which comes out as a single %.

The second is %=, which generates a unique number for each asm() block. (Each time it is used if inlined or used in a macro.) This can be used for temporary labels and so on.

4.13 Examples

Some code that was in include/asm-i386/system.h:

```
#define _set_tssldt_desc(n,addr,limit,type) \
 __asm__ __volatile__ ("movw %3,0(%2)\n\t" \
    "movw %%ax,2(%2)\n\t" \
    "rorl $16,%%eax\n\t" \
    "movb %%al,4(%2)\n\t" \
    "movb %4,5(%2)\n\t" \
    "movb $0,6(%2)\n\t" \
    "movb %%ah,7(%2)\n\t" \
    "rorl $16,%%eax" \
    : "=m"(*(n)) \
    : "a" (addr), "r"(n), "ri"(limit), "i"(type))
```

It's obvious that the writer didn't know how to take optimal advantage of this (admittedly complex, but x86 addressing *is* complex) facility. This could be rewritten to use any register instead of %eax:

```
#define _set_tssldt_desc(n,addr,limit,type) \
__asm__ __volatile__ ("movw %w3,0(%2)\n\t" \
    "movw %w1,2(%2)\n\t" \
    "rorl $16,%1\n\t" \
    "movb %b1,4(%2)\n\t" \
    "movb %4,5(%2)\n\t" \
    "movb $0,6(%2)\n\t" \
    "movb %h1,7(%2)\n\t" \
    "rorl $16,%1" \
    : "=m"(*(n))
    : "q" (addr), "r"(n), "ri"(limit), "ri"(type))
```

You notice here that *n is listed as an output, so GCC knows that it's modified, but actually addressing it is done relative to n as an input register everywhere because of the need to compute an offset.

The problem is that there is no syntactic way to encode an offset from a given address. If the address is "40(%eax)" then an offset of 2 can be made by prepending "2+" to it. But if the address is "(%eax)" then "2+(%eax)" is not valid. Tricks like "2+0" fall flat because "040" is taken as octal and gets translated into 32.

BUT THERE'S NEWS (19 April 1998): gas will actually Do The Right Thing with "2+(%eax)", just emit a warning. Having seen this, a gas maintainer (Alan Modra) decided to make the warning go away in this case, so in some near future version you will be able to do it.

With this fix (or putting up with the warning), you could write the above as:

```
#define _set_tssldt_desc(n,addr,limit,type) \
__asm__ __volatile__ ("movw %w2,%0\n\t" \
    "movw %w1,2+%0\n\t" \
    "rorl $16,%1\n\t" \
    "movb %b1,4+%0\n\t" \
    "movb %3,5+%0\n\t" \
    "movb $0,6+%0\n\t" \
    "movb %h1,7+%0\n\t" \
    "rorl $16,%1" \
    : "=o"(*(n))
    : "q" (addr), "ri"(limit), "i"(type))
```

The "o" constraint is just like "m", except that it's "offstable"; adding a small value to it leaves a valid address. On the x86, there is no distinction, so it's not really necessary, but on the 68000, for example, you can't add an offset to a postincrement addressing mode.

If neither the warning nor waiting is acceptable, a fix is to list each possible offset as a different output (here we're using the fact that n is a char *):

```
__asm__ __volatile__ ("movw %w7,%0\n\t" \
     "movw %w6,%1\n\t" \
     "rorl $16,%6\n\t" \
     "movb %b6,%2\n\t" \
     "movb %b8,%3\n\t" \
     "movb $0,%4\n\t" \
     "movb %h6,%5\n\t" \
     "rorl $16,%6" \
     : "=m"(*(n)), \
       "=m"((n)[2]), \
       "=m"((n)[4]), \
       "=m"((n)[5]), \
       "=m"((n)[6]), \
       "=m"((n)[7]) \
     : "q" (addr), "g"(limit), "iqm"(type))
```

Although, as you can see, this gets a bit ugly when you have lots of offsets, but it works just the same.

4.14 Conclusion

I hope this has been of use to some folks. GCC's inline asm features are really cool becuase you can just do the little bit that you want and let the compiler optimize the rest.

This has the unfortunate side effect that you have to learn how to explain to the compiler what's going on. But it's worth it, really!