A Brief Tutorial on GCC inline asm (x86 biased)

by colin@nyx.net

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.

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 more clever. 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 because 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 auto-increment 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 could use an immediate constant in the instruction.

The x86-specific constraints are defined later.

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.

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 % symbols. 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. *movl $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.

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.

To reiterate what was said in the section on inputs above, this asm is also promising **not** to alter input1, unless GCC assigns it to the same register as the output. (Which could only happen if input1 and input2 ended up in the same register, which is entirely legal if they happen to be the same value.)

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-addressable register (eax, ebx, ecx, edx)
A - eax or edx
a, b, c, d, S, D - eax, ebx, ecx, edx, esi, edi respectively

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 - immediate 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*.)

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 constraints 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 register 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 change 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.
```

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 haven'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 picky, and complaining that there are no free registers in class *a* available. This is not the way to do it. If you corrupt 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));
```

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.

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.

As an example of where it is relevant, consider the ntohl() functions that switch around byte ordering. y=bswap(x) depends only on the bits in x, and two calls with the same x can be optimized down to one.

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
```

The code could get better yet, but you can see how it improves.

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.

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 immediates *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 like %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 have 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 corresponding 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) : "0" (x));
```

This says that x must be in a byte-addressable 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 addressable 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.

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.

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 hacker (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, if the gas maintainer accepts his hacks (something that is not at all certain)

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 post-increment 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.

Conclusion

I hope this has been of use to some folks. GCC's inline asm features are really cool because 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!