Including Assembly Language Instructions in C and C++ Code

The GNU compiler has a feature by which you can explicitly include assembly language instructions in your C or C++ code. This might sound like a bizarre thing to do – after all, the whole point of a compiler is to be able to avoid worrying about the details of the underlying hardware – but there are occasions where it can be useful. Perhaps the most straightforward rational for the feature is to gain access to hardware features that the compiler knows nothing about. For example, the PowerPC architecture defines something called a "Time Base Register", which can be used for making timing measurements. There's nothing in the C or C++ languages that would allow you to access this register. But by embedding assembly language statements in your code you can take advantage of it. Of course, by embedding assembly language instructions in your code you are making it non-portable, so it's something that should be done with restraint.

The PowerPC Architecture

The examples in this document will be based on the PowerPC architecture. In case you're not familiar with this processor, it's a 32 bit RISC machine, with 32 general purpose registers (numbered 0 .. 31). When a subroutine is called the parameters are passed in registers R3, R4, R5, etc, and the return value is returned in register R3. There are also various special purpose registers, such as the time base register (TBR) mentioned above. Of course there's a lot more to it than that, but that's enough to understand the examples.

The "asm" Function

You embed assembly language instructions by using the "asm" builtin function. The parameters passed to asm can get extremely complicated. We'll start with very simple examples and work our way up. The simplest form is to just write the assembly language instruction you want as a quoted string. For example, suppose we just want to introduce a tiny delay with a "nop" (no-operation) instruction. This can be done with:

```
asm ( "nop" );
```

This will cause the compiler to insert a single PowerPC instruction ("nop") into the instruction stream it is generating. In general, other assembly instructions that have no arguments are just as simple. The PowerPC also has an instruction called "eieio" (yes, it really does), which is used to control how the processor orders load and store instructions. You can include one by doing:

```
asm ( "eieio" );
```

Inserting an instruction with fixed register references is just as simple. Suppose I want to add registers 20 and 21, and put the result in register 9. This would be done with:

```
asm ( "add r9,r20,r21" );
```

The last example brings us to our first complication. This line will be in some C/C++ file that is being compiled. How do you know what the compiler is doing? Maybe the compiler has saved something important in register 9, and now you've just blown it away with your inserted instruction! There are a large number of complicated options to the asm builtin, which allow the compiler to intelligently insert the instruction you want. This doesn't mean it's easy, but the options at least give you a fighting chance.

Adding Two Integers

To introduce the various extensions to the basic assembly instruction functionality of asm we'll use a standard example – a subroutine that takes two integers as arguments and returns their sum. In C, the code is:

```
int
add_example (int aa, int bb)
{
    return (aa + bb);
}
```

This will generate a subroutine with two instructions:

```
add r3,r3,r4
blr
```

Remember that on the PowerPC parameters are passed in via registers starting with register 3. Thus the value of aa will be in register 3, the value of bb will be in register 4, and the result will be passed back in register 3. The add instructions does what we need (registers 3 and 4 added together, sum placed in register 3). The blr instruction is how you return from a subroutine.

Now, rather than doing the add in C code, we can do it with an asm:

```
int
add_example (int aa, int bb)
{
    asm ( "add r3,r3,r4" );
}
```

This will generate exactly the same two instructions as the first example:

```
add r3,r3,r4
blr
```

Note that we didn't explicitly add the blr instruction. The compiler generated that itself just as it did in the first example, because that's what it outputs when it reaches the end of a subroutine.

Referring to registers directly is tricky – you have to know exactly what state the compiler is in to get it right. Wouldn't it be better if we could do things symbolically? Well we can! The string giving the assembly language instructions can (optionally) be followed by up to three other sections (separated by colons). Thus an asm builtin call can have up to four sections:

- ASSEMBLY_CONSTRUCT
- OUTPUT_INFO
- INPUT_INFO
- CLOBBER_INFO

All the examples we've shown so far have only had ASSEMBLY_CONSTRUCTS. The assembly constructs section gives the compiler the actual machine instructions to be included. The OUTPUT_INFO section gives the compiler information about outputs (things that will be written to). The INPUT_INFO section gives the compiler about inputs (things that will be read from). The CLOBBER_INFO informs the compiler about various side effects the assembly language instruction might produce, so it can factor that into its code generation. So symbolically a full asm call would look like:

```
asm( ASSEMBLY_CONSTRUCT : OUTPUT_INFO : INPUT_INFO : CLOBBER_INFO );
```

Only the ASSEMBLY_CONSTRUCT is required. All the others are optional. If you want to skip a section you leave it blank. For example, if we just want the ASSEMBLY_CONSTRUCT and INPUT_INFO sections, the format would be:

```
asm ( ASSEMBLY_CONSTRUCT : : INPUT_INFO );
```

We will be using the above format for our next example:

```
int
add_example (int aa, int bb)
{
    asm ( "add r3,%0,%1" : : "r" (aa), "r" (bb) );
}
```

One thing to note is that the content of the ASSEMBLY_CONSTRUCT has changed. For the two inputs we are now using %0 and %1 instead of explicit registers. This capability is why we refer to the first section as an ASSEMBLY_CONSTRUCT, rather than an ASSEMBLY_INSTRUCTION. Using "%", we can identify parts of the instruction that the compiler itself will fill in, based on the information in the other sections of the asm call.

The OUTPUT_INFO section is empty, and the INPUT_INFO section contains two items, separated by a comma. There is no CLOBBER_INFO section in this asm call. The general syntax used by the output, input, and clobber sections is that they consist of a series of one or more items, separated by commas. In the example above, the INPUT_SECTION has the two items:

```
"r" (aa)
"r" (bb)
```

Each item consists of two parts: 1) a quoted string, which gives a set of constraints on the item, and 2) the name of a C identifier in parenthesis. The constraint string in this case is just the letter "r", which is telling the compiler that it should use a general purpose register. The items in the output and input sections are correlated with the '%#' items in the assembly section by order – the first item corresponds to %0, the second to %1, etc. In this example, since there are no output items, we have aa associated with %0, and bb associated with %1.

What do we get when we compile this? Exactly the same code as before!

```
add r3,r3,r4
blr
```

This is because the compiler knows that the first parameter (aa) is located in r3, and that bb is located in r4. It may seem that we have gained nothing, but consider – suppose the calling convention changed so that instead of passing parameters in r3, r4, etc, we started passing then in r10, r11, etc. Our original coding, with explicit registers, is now broken. But the modified example will automatically track the fact the aa is now in r10, bb is in r11, and it will correctly generate the code:

```
add r3,r10,r11
```

To further drive home the association between the % items in the ASSEMBLY_CONSTRUCT and the items in the INPUT_INFO, suppose that instead of returning (aa + bb) we want to return (bb + aa). The two are equivalent as far as the result goes - we're just doing this for pedantic purposes. To do (bb + aa), we would code the asm as:

```
asm ( "add r3,%1,%0" : : "r" (aa), "r" (bb) );
```

and this would generate the assembly instruction:

```
add r3,r4,r3
```

Note that second and third registers are now reversed.

So far we've still been explicitly using r3 as the output register. We'd really like to do that symbolically as well, and we can. Here's the code in C:

```
int
add_example (int aa, int bb)
{
    int tmp;
    asm ( "add %0,%1,%2" : "=r" (tmp) : "r" (aa), "r" (bb) );
    return tmp;
}
```

And here's the code that gets generated:

```
add     r4,r3,r4
mr      r3,r4
blr
```

What's going on? First we've now added an OUTPUT_INFO item:

```
"=r" (tmp)
```

The constraint string is "=r". The '=' tells the compiler that this is an output item (which seems redundant, as this is already known because the item is in the OUTPUT_INFO section. None the less, all output items must have '=' in their constraint strings). As before, the 'r' says this item is talking about a register. And this output item is to be associated with the C variable named tmp.

We've now got three '%#' items in the ASSEMBLY_CONSTRUCT. %0 corresponds to tmp, %1 to aa, and %2 to bb.

So, what's with the first add instruction? The compiler is adding aa and bb (r3 and r4), which is what we want, but why is it putting the result in r4? Well, via the OUTPUT_INFO item we told the compiler that it should put the output in variable tmp, and apparently the compiler decided that r4 would be a good register to use for variable tmp! Our asm call is really just a complicated way of saying

```
tmp = aa + bb;
```

Since we've put the sum in a temporary variable called tmp, we had to add the explicit "return tmp" line. In the previous examples we were "cheating" by forcing the sum into register r3, where we knew the return value should be placed. Since the compiler knows that it used r4 for tmp, and that the return value must be in r3, it generates the "mv r3,r4" instruction to copy the sum from r4 to r3.

This is actually less efficient that the plain C code we started with, but this is not meant to be a realistic example of the use of asm. The purpose is to get you up to speed on the syntax of the asm call.

Is there something better than %0, %1, etc ?

In the previous section we saw how "%#" could be used as placeholders in an ASSEMBLY_CONSTRUCT. Just using numbers as identifiers is rather primitive – names would be better, and there is a way to do that. Instead of a percent sign followed by a number we can use "%[NAME]". Thus the last asm call:

```
asm ( "add %0,%1,%2"
      : "=r" (tmp)
      : "r" (aa), "r" (bb) );
```

could be rewritten as:

```
asm ( "add %[ASM_tmp],%[ASM_aa],%[ASM_bb]"
      : [ASM_tmp] "=r" (tmp)
      : [ASM_aa] "r" (aa), [ASM_bb] "r" (bb) );
```

Note that I've started splitting the call across multiple lines, which is often done for readability when the parameters start getting long and complicated. In the ASSEMBLY_CONSTRUCT I've replaced %0 with %[ASM_tmp], %1 with %[ASM_aa], and %2 with %[ASM_bb]. Now that the place holders are no longer numbers, we need another way to know which place holder is associated with which OUTPUT_INFO or INPUT_INFO item. We do this, as you can see above, by placing the placeholder name (in square brackets) ahead of the individual items. This, each OUTPUT_INFO or INPUT_INFO item can actually have up to three parts:

```
[ PLACE_HOLDER_NAME ] "CONSTRAINT_STRING" ( C_NAME )
```

Where the PLACE_HOLDER_NAME part is not needed if you are using the "%#" type of place holders. Note that the place holder names you use are arbitrary. For readability I like to keep them the same as the C name, with "ASM_" prepended.

Do you plan on clobbering anyone ?

Some assembly instructions have side effects that the compiler needs to know about. Knowledge about such things is given to the compiler via the CLOBBER_INFO section. This section consists of a series of quoted strings separated by commas.

If your instruction affects memory, include "memory" in the CLOBBERED_INFO section. This tells the compiler not to depend on values loaded from memory before the asm to still be valid after the asm.

If your instruction affects the condition code register, include "cc" in the CLOBBERED_INFO section.

Any registers you reference directly should be included in the CLOBBERED_INFO section. Thus, in general, you should not write an asm that looks like:

```
asm ( "add r9,r20,r21" );
```

Instead it should be:

```
asm ( "add r9,r20,r21"
     :       /* No output info            */
     :       /* No input info             */
     : "r9"  /* Things that are clobbered */
     );
```

So now the compiler knows that r9 is clobbered, and can make appropriate adjustments to the code it is generating. I did not include "r21" or "r22" in the list because those are only read, not modified, by the instruction.

Let's go through it one more time

This section is a review of what's been presented so far, and expands on some topics.

The general format of an asm call is:

```
asm( ASSEMBLY_CONSTRUCT : OUTPUT_INFO : INPUT_INFO : CLOBBER_INFO );
```

The ASSEMBLY_CONSTRUCT contains either a literal assembly language instruction, or an assembly language construct , which is an assembly language line with some symbols the compiler needs to "fill in" (such symbols start with '%').

The CLOBBER_INFO section tells the compiler what things the assembly instruction is going to touch in a "destructive" way, be it registers or memory.

The OUTPUT_INFO and INPUT_INFO sections map the symbols in the assembly construct (EG %0, %[FOOBAR], etc) to a C (or C++) name. Each mapping has either 2 or 3 elements, of the form:

```
CONSTRUCT_NAME(optional)   "CONSTRAINTS"   (C_NAME)
```

The CONSTRUCT_NAME is optional in the sense that it doesn't appear if you are using numeric symbols (%0, %1, %2, etc), but it is needed if you are using named symbols (%[ASM_in], %[ASM_out], etc).

To say that the last element is a "C_NAME" is actually incomplete. It can in fact be a C (or C++) name, but the syntax actually allows it to be a C (or C++) expression. This is the "thing" that is supposed to be substituted into your assembly construct at the place marked with the matching % symbol.

The constraint section gives the compiler additional "hints" so that it can properly match your construct symbol (%...) to the proper C (or C++) items in a correct and useful way.

Let's look at this from the compiler's point of view.

If it runs across an asm call with a simple assembly instruction (e.g. "nop") it's easy. It just inserts the instruction. If the instruction has any effect on registers, or memory, or whatever, the compiler will depend on you telling it so in the CLOBBER_INFO section so that it can adjust the code it is generating so that your asm code and the compiler's generated assembly instructions don't interfere with each other. The OUTPUT_INFO and INPUT_INFO sections are not needed.

If the compiler comes across an asm call with symbols (%...) in the assembly construct then it has more work to do. The % whatever is not part of a valid machine instruction, and the compiler knows it has to replace it with something else so that what you ultimately end up with *is* a valid assembly instruction. It will use the OUTPUT_INFO and INPUT_INFO sections to figure out what should replace the % symbol.

What we've looked at previously were cases where we just wanted to access a variable that we expected to be in a register. Hence we just had a constrict string of "r" (or "=r") and the C_NAME part was just the name of the C variable. Another possible constraint is "I", meaning an integer constant. Suppose we want to load the constant DELTA to register 5. Here's an asm to do it:

```
#define DELTA 20
asm ( "li r5,%[ASM_DELTA]" : : [ASM_DELTA] "i" (DELTA) : "r5" );
```

I've include the #define just for clarity. I'm using a "load immediate" (li) instruction. I've defined an input parameter which is an integer constant with the C-code name of "DELTA". I've also told the compiler that I will be clobbering register 5. The above asm generates the machine instruction:

```
li r5,20
```

As you can see, the compiler was able to map from the %[ASM_DELTA] in the ASSEMBLY_CONSTRUCT to the C symbol DELTA (by using the INPUT_INFO), and was finally able to match the C symbol DELTA to the value 20.

Just to reiterate the fact that the %[NAME] you use is arbitrary, the above asm could have been written as:

```
asm ( "li r5,%[xyz]" : : [xyz] "i" (DELTA) : "r5" );
```

And it would have worked exactly the same.

What you see ain't necessarily what you get.

Unfortunately, the assembly instructions inserted by asm calls are placed before optimization is complete. Thus, those instructions you've so carefully constructed and placed can get moved, or even deleted!

Preventing deletion is the easier of these two issues to deal with. The asm builtin can be followed by the optional keyword "volatile". For example:

```
asm volatile ( "nop" );
```

Volatile tells the compiler the instruction may have non-obvious side-effects, and should therefore not be deleted. (But note that if the compiler decides there's no way the code could ever reach your instruction, it might still be deleted as part of "dead code removal". However, in that case you've got other issues, as you obviously put the asm in the wrong place to start with!).

Now suppose we want the following three instructions to execute in the order given:

```
asm volatile ( "nop" );
asm volatile ( "add r9,0,25" );
asm volatile ( "nop" );
```

Just putting the three asm calls in a row, as I've done above, is no guarantee that they will stay in that order. The optimizer might decide to move them around. It turns out that you can include multiple instructions in a single ASSEMBLY_CONSTRUCT, and if that's viable that's the easiest way to guarantee that the order you want is preserved. Thus, the above should be written as:

```
asm volatile ( "nop\n\t"
               "r9,0,25\n\t"
```

```
                    "nop" );
```

I've split the string into three pieces on three lines for readability, but of course the compiler will combine all three strings into a single string (per the C standard). The newline and tab are needed so that the assembler will know that there are 3 separate lines. For some assemblers a semicolon (;) will work as well.

More On Constraints

So far we've seen two letters show up in the constraint string: 'r" and '='. . Here's a partial list of possible constraints & what they mean:

```
m    memory
o    offsetable memory address
V    non-offsetable memory address
<    memory address with autodecrement
>    memory address with autoincrement

i    immediate integer operand
n    immediate integer operand with known value
s    immediate integer operand that is not an explicit value

E    immediate floating operand
F    immediate floating operand

r    general register

g    any general register, memory or immediate integer

X    any type of operand is allowed

=    item is an output
+    item is both input and output
&    used with '=', means output operand may not overlap with any
input
%    declares that this operand and the next commute
```

If '=' or '+' are used, they must be the first character in the constraint string. Some constraints are also machine specific. For further discussion of constraints see:

http://gcc.gnu.org/onlinedocs/gcc/Simple-Constraints.html#Simple-Constraints
http://gcc.gnu.org/onlinedocs/gcc/Modifiers.html#Modifiers
http://gcc.gnu.org/onlinedocs/gcc/Machine-Constraints.html#Machine-Constraints

Some Additional Points

You can use the asm call to assign a variable to a particular register. In the following example we get the compiler to put integer variable foo in register 10:

```
int foo asm ( "r10" );
```

We can even initialize it to a value:

```
int foo asm ( "r10" ) = 3;
```

We don't even have to generate an assembly instruction. We can just tell the compiler that it shouldn't depend on anything loaded to memory before the asm call to still be there after the asm call:

```
asm ( "" : : : "memory" );
```

The above call is telling the compiler that at that point in the code memory may change in unexpected ways. Thus even though it may still have a value it loaded from memory before the asm call, once the asm call is crossed it shouldn't use the saved value, but should reread the value from memory.

Making the Assembler Happy

If you've tried any of the above examples you've probably discovered something significant – they don't work! The problem is that by default the PowerPC assembler expects registers to be referenced by numbers only. Instead of

```
add r3,r3,r4
```

It wants:

```
add 3,3,4
```

You can of course change your code, but I prefer having my register names start with "r". There is a way around this. The PowerPC assembler has an option, "-mregnames", which tells it to accept register identification of the form "r#". To pass that option thru gcc to the assembler we add the following to the gcc command line:

```
-Xassembler –mregnames
```

Order is important! The "-Xassembler" just tells gcc that the next option should be passed through to the assembler.

Note that this particular issue is peculiar to the PowerPC, and maybe a few other processors. However, other processors and their assemblers might have other gotcha's, so this is an area where you might run into issues that require some research (and/or some trial and error experimentation).

One general-purpose bit of knowledge is that if you need to pass through a '%' in your ASSEMBLY_CONSTRUCT you need to use '%%' (basically the same way '%' is handled in printf format strings).

Checking Things Out

As you should have gathered by now, getting your assembly instructions exactly as needed is non-trivial. You certainly cannot assume that the generated code is going to be what you want, no matter how careful you are. It's necessary to look at the actual code being produced. There are two ways to do this. One is to use the compiler –S switch, which causes gcc to generate an assembly language listing (with a .s suffix). The other is to us objdump on the generated .o file (EG "objdump -d –S example.o") and examine the machine instructions. Also remember that changes to the set of compiler switches used (EG the optimization level) might change the code that gets generated. So be aware that changing compiler options can have unforeseen and unwanted effects on what your asm calls end up doing!

References

http://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html
http://gcc.gnu.org/onlinedocs/gcc/Simple-Constraints.html#Simple-Constraints
http://gcc.gnu.org/onlinedocs/gcc/Modifiers.html#Modifiers
http://gcc.gnu.org/onlinedocs/gcc/Machine-Constraints.html#Machine-Constraints
http://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html