# GCC inline assembler code notes for PowerPC

Reference: GCC docs. A lot of the material here seems to have been written assuming that the inline assembly consists of a single instruction which reads inputs and then writes outputs (possibly overwriting inputs).

Inline assembler code for the memory management package uses the GNU extension of named operands, e.g., "%[foo]". We generally use the "&" constraint for output operands; otherwise, the compiler assumes that the output operands are set as the very last actions of the code, after all input operands have been used. In such a case the compiler may use the same register for an input operand and an output operand. Using "&" marks the output operand as "early clobber",i.e., the register is altered early on, perhaps before all the inputs have been used.

We also use the trick of creating a register variable to use as an output only ("=") or input-output ("+") operand when we want the compiler to pick which register to use. Sometimes when we do that we want the compiler to avoid r0 because it can't be used as a base register. Then we use the "b" constraint instead of "r", where "b" means "register that can be a base register".

Even if you use "volatile" the compiler will analyze the input operands, output operands and clobber list of each asm() and may reorder two asm() sequences if it determines that they don't depend on one another. The "memory" clobber list item will stop that since it means "updates memory in an unpredictable way". When you throw syncs's and isync's around it's even true. The other way to prevent reordering is to put everything into a single asm().

You should have a memory operand constraint if you change a known location; "known" in the sense that the compiler can deduce the relationship between it and other values visible in the local scope. For example if there is a pointer p and you load or modify *p you'll need "m"(*p) or "=m"(*p), respectively. You need to do this even if you don't use that constraint number (or name) in your assembly code. On a PowerPC some memory reference instructions use two registers to calculate an effective address instead of an offset and a register, so you'll need both a register constraint for the base register and a memory constraint to tell the compiler the effect of the instruction, e.g.:

asm("lwarx %0,0,%1" : "=r"(result) : "r"(p), "m"(*p))

The memory-ref constraint tells the compiler that the value of *p is needed, preventing it from possibly optimizing away a prior store to some location known to be aliased to *p.

If you can't write a simple constraint for an access to memory then you should have "memory" on the clobber list. Yes, even if you don't change memory. This situation would arise if you were to add a run-time offset to a pointer and dereference the result. If the compiler has been keeping memory values cached in registers it will have to flush them all before a read or write to an unpredictable location on the chance that one of the cached locations may be the target. If you do change memory and need to use the memory-clobber constraint then you should use "asm volatile" instead of just "asm"; "volatile" keeps optimization from removing the asm code which is important if the code has some crucial side effect that the compiler can't deduce from the constraints. You will also need "volatile" if your assembly code depends on a voltile data source such as an I/O register. Remember that the compiler doesn't analyze the actual instructions in inline assembly code.

**Example 1.** Here we alter the output "msr" early on (in the first instruction, in fact) so we use the ampersand. We also mark it as output-only ("=") instead of in-out ("+") because we don't care what the value is before the asm. The use of "volatile" and "memory" makes sure that the compiler won't move instructions across the inline assembler code during the course of optimization. We need that because the inline code is changing the Machine State Register and hence the context in which instructions are executed; the isync instruction just prevents reordering by the processor not by the compiler. The compiler doesn't look at the instructions inside the asm, just at the lists of inputs, outputs and clobbers. (The "\n\t" at the end of each instruction just makes the assembly code output generated with -S look nice.)

```
register unsigned msr;
asm volatile
  (
   // Make sure that address translation is disabled.
   "mfmsr  %[msr]                \n\t"
   "rlwimi %[msr],%[zero],0,26,27 \n\t" // Clear translation bits (26-27) in the PPC405 MSR.
   "mtmsr  %[msr]                \n\t"
   // Perform a context sync since we need to wait for translation
   // to turn off. This will also purge any shadowed TLB entries
   // which must be done before we turn translation back on.
   "isync \n\t"
   // Clear the zone-protection and PID registers.
   "mtzpr %[zero] \n\t"
   "mtpid %[zero] \n\t"
   // Mark all TLB entries as invalid.
   "tlbia"
   : [msr]"=&r"(msr)
   : [zero]"r"(0)
   : "memory"
   );
```

**Example 2.** This code changes a specific memory location so it uses the "m" constraint; the compiler will replace "%[saved]" in the instruction with the correct addressing mode, e.g., base + offset. Note that if necessary the compiler will generate extra code to load the value of "this" into a base register.

```
register unsigned oldmsr;
register unsigned newmsr;
asm volatile
  (
   // Save the exception-enabling bits of the current PPC405 MSR value.
   "mfmsr %[oldmsr]                  \n\t"
   "and   %[newmsr],%[oldmsr],%[mask] \n\t"
   "stw   %[newmsr],%[save]          \n\t"
   // Reset those enabling bits in the MSR.
   "andc  %[newmsr],%[oldmsr],%[mask] \n\t"
   "mtmsr %[newmsr]                  \n\t"
   // Perform a context sync since the MSR value may have changed.
   "isync"
   :[save]"=m"(this->mSavedExcBits), [oldmsr]"=&r"(oldmsr), [newmsr]"=&r"(newmsr)
   :[mask]"r"(ExceptionBitMask)
   :
   );
```

**Example 3.** Here's a case where we explicitly specify the contents of a base register and so have to use the "b" constraint to keep the compiler from picking r0. We also put a block of memory on the output list by dereferencing a pointer to the block; in this case we're altering the entire object so we use "*this".

```
register unsigned tmp;
asm volatile
  (// Read and store a Translation Lookaside Buffer entry from the PPC405 MMU.
    // High part.
   "tlbrehi %[tmp],%[idx]      \n\t"
   "stw     %[tmp],0(%[self])  \n\t"
    // Low part.
   "tlbrelo %[tmp],%[idx]      \n\t"
   "stw     %[tmp],4(%[self])"
   : "=m"(*this), [tmp]"=&r"(tmp)
   : [idx]"r"(this->index), [self]"b"(this)
   :
   );
```

The "I" constraint lets you specify signed 16-bit constants so the above could also have been written like the following, assuming that the data members affected were named "foo" and "bar" in the Plain Old Data structure MyPod:

```
register unsigned tmp;
asm volatile
  (
   "tlbrehi %[tmp],%[idx]      \n\t"
   "stw     %[tmp],%[foo](%[self])  \n\t"
   "tlbrelo %[tmp],%[idx]      \n\t"
   "stw     %[tmp],%[bar](%[self])"
   : "=m"(*this), [tmp]"=&r"(tmp)
   : [idx]"r"(this->index), [self]"b"(this),[foo]"I"(offsetof(MyPod,foo)),[bar]"I"(offsetof(MyPod,bar))
   :
   );
```

**Counterexample.** The following code is meant to atomically remove the head item from a linked list and return the pointer the item. The programmer meant for %2 to always contain the address of the variable _head; that variable contains a pointer to the first item on the list. %2 must not be changed by this code because it will be used to update _head later; it will also be needed to read _head again if the `stwcx` instruction doesn't store and the processor takes the branch. However, the output argument "tmp" (%1) is not marked early-clobber with "&" so the compiler is in principle free to make %1 the same register as one of the inputs. In one case the compiler used r0 for both %2 and %1 so that the `lwzx` instruction destroyed the contents of %2.

```
Flink<T>* head;
Flink<T>* tmp;
asm volatile ("1:;"
              "lwarx  %0,0,%2;"
              "lwzx   %1,0,(%0);"
              "stwcx. %1,0,%2;"
              "bne-   1b;"
              : "=&r" (head), "=r" (tmp) : "r" (&_head) : "cc", "memory");
return static_cast<T*>(head);
```

Now consider the effect of the above instructions when %2 is the same as %1 and the list has one member (and is terminated by a special "empty" node), and assuming that the stwcx instruction always stores:

_head -> first -> empty

| Instruction | _head | first | %2==%1 | %0 |
|---|---|---|---|---|
| Setup | &first | &empty | &_head | ? |
| lwarx | &first | &empty | &_head | &first |
| lwzx | &first | &empty | &empty | &first |
| stwcx. | &first | &empty | &empty | &first |

This is what was intended (to get it the constraint for "tmp" should be "=&r" instead of just "=r"):

| Instruction | _head | first | %2 | %1 | %0 |
|---|---|---|---|---|---|
| Setup | &first | &empty | &_head | ? | ? |
| lwarx | &first | &empty | &_head | ? | &first |
| lwzx | &first | &empty | &_head | &empty | &first |
| stwcx. | &empty | &empty | &_head | &empty | &first |