A guide to inline assembly for C and C++

Basic, intermediate, and advanced concepts

Salma Elshatanoufy
William O'Farrell

November 01, 2011

First, the authors describe basic usage syntax for inline assembly (inline asm) embedded within C and C++ programs. Then they explain intermediate concepts, such as addressing modes, the clobbers list, and branching stanzas, as well as more advanced topics, such as memory clobbers, the volatile attribute, and locks are discussed for those who want to use inline asm in multithreaded applications.

In this article, we discuss several use scenarios for inline assembly, also called inline asm. For beginners, we introduce basic syntax, operand referencing, constraints, and common pitfalls that new users need to be aware of. For intermediate users, we discuss the clobbers list, as well as branching topics that facilitate the use of branch instructions within inline asm stanzas in their C/C++ code. Lastly, we discuss memory clobbers and the volatile attribute for advanced users who use inline asm to optimize their code. We conclude with an example of multithreaded locking with inline asm.

Basic inline asm

In the asm block shown in code Listing 1, the addc instruction is used to add two variables, op1 and op2. In any asm block, assembly instructions appear first, followed by the inputs and outputs, which are separated by a colon. The assembly instructions can consist of one or more quoted strings. The first colon separates the output operands; the second colon separates the input operands. If there are clobbered registers, they are inserted after the third colon. If there are no clobbered inputs for the asm block, the third colon can be omitted, as Listing 2 shows.

Listing 1. Opcodes, inputs, outputs, and clobbers

```c
int res=0;
int op1=20;
int op2=30;

asm ( " addc.    %0,%1,%2        
  : "=r"(res) 
  : "b"(op1), "r"(op2) 
  : "r0"       );
```

```c
int res=0;
int op1=20;
int op2=30;
asm ( " addc.    %0,%1,%2        
  : "=r"(res) 
  : "b"(op1), "r"(op2) 
  : "r0"       );
```
Listing 2. No clobbered inputs for the asm block, so third colon omitted

```
asm ( " addc.    %0,%1,%2        
     : "=r"(res)       
     : "b"(op1), "r"(op2) );
```

Note:
The clobberers list is discussed later in this section.

Each instruction "expects" inputs and outputs to be passed in a certain format. In the previous example, the addc. instruction expects its operands to be passed through registers, hence op1 and op2 are passed into the asm block with the "b" and "r" constraints. For a complete listing of all legal asm constraints for the IBM XL C and C++ compiler, see the compiler language reference.

Register constraints on variable declarations

In some programs, you will want to tie variables to certain hardware registers. This is done at the variable declaration. The following example ties the variable res to GPR0 throughout the life of the program:

```
int register res asm("r0")=0;
```

When the variable type is not matched with the type of target hardware register, you will receive a compilation error notice.

After a variable is tied to a specific register, it is not possible to use another register to hold the same variable. For example, the following code will cause a compilation error, the variable res is associated at declaration time with GPR0, but in the asm block, the user attempts to use any register but GPR0 to pass in res.

Listing 3. Compilation error when conflicting constraints are used on a variable

```
int register res asm("r0")=0;
asm ( " addc.    %0,%1,%2        
     : "=b"(res)         
     : "b"(op1), "r"(op2) : "r0" );
```

In the example in Listing 4, there is no output operand for the stw instruction, hence the outputs section of the asm is empty. None of the registers is modified, so they are all input operands, and the target address is passed in with the input operands. However, something is modified: the addressed memory location. But that location is not explicitly mentioned in the instruction, so the output of the instruction is implicit rather than explicit.

Listing 4. Instructions with no output operands
int res [] = {0,0};
int a=45;
int *pointer = &res[0];
asm ( " stw    %0,%1(%2)        
:   : "r"(a), "i"(sizeof(int)),"r"(pointer));

Listing 5. Instructions with preserved operands

int res [] = {0,0};
int a=45;
asm ( " stw    %0,%1(%2)        
: +r"(res[0])
:  "r"(a), "i"(sizeof(int)),"r"(pointer));

In listing 5, if you want to preserve the initial value of a result variable that is not necessarily modified by the asm block, then you need to use the + (plus sign) constraint to preserve the initial value of that variable, as is shown with res[0].

Target memory addresses in inline asm

If an instruction specifies two of its arguments in a form similar to D(RA), where D is a literal value and RA is a general register, then this is taken to mean that D+RA is an effective address. In this case, the appropriate constraints are "m" or "o". Both "m" and "o" refer to memory arguments. Constraint "o" is described as an offsettable memory location. But in the IBM® POWER® architecture, nearly all memory references require an offset, so "m" and "o" are equivalent. In this case, you can use a single constraint to refer to two operands in the instruction. Listing 6 is an example.

Listing 6. A single constraint to refer to two operands in the instruction

int res [] = {0,0};
int a=45;
asm ( " stb %1,%0          
:   : "=m"(res[1])
: "r"(a));

The form of the instruction stb (from the assembly language reference) is: stb RS,D(RA).

Although the stb instruction technically takes three operands (a source register, an address register, and an immediate displacement), the asm description of it uses only two constraints. The "=m" constraint is used to notify the compiler that the memory address of res is to be used for the result of the store instruction (The "sync" instruction is often used for this purpose, but there are others available, as described in the POWER ISA See Resources for a link.) The "=m" indicates that the operand is a modified memory location. You do not need to know the address of the target location beforehand, because that task is left to the compiler. This allows the compiler to choose the right register (r1 for an automatic variable, for instance) and apply the right displacement automatically. This is necessary, because it would generally be impossible for an asm programmer
to know what address register and what displacement to use. In other instances, you can also override this behavior by manually calculating the target address as in the following example.

**Listing 7. Manually calculating the target address**

```c
int res[] = {0, 0};
int a=45;
asm ( "stb %0,%1(%2) \n" : "r"(a), "i"(sizeof(int)),"r"(&res));
```

In this code, the specification `%1(%2)` represents a base address and an offset, where `%2` represents the base address, and `res[0]` and `%1` represent the offset, `sizeof(int)`. As a result, the store is performed at the effective address, `res`.

**Note:**
For some instructions, `GPR0` cannot be used as a base address. Specifying GPR0 tells the assembler not to use a base register at all. To ensure that the compiler does not choose `r0` for an operand, you can use the constraint "b" rather than "r".

**Addressing modes for POWER and PowerPC instructions**

The IBM POWER architecture type is RISC. Instructions typically operate either with three register arguments (two registers for source arguments, one register to hold a result) or with two registers and an immediate value (one register and one immediate value for the source arguments, and one register to hold the result). There are exceptions to this pattern, but mostly it is true.

Among the instructions that take two registers and an immediate value, there are two special subclasses: load instructions and store instructions. These instructions use the immediate value as an offset to the value in the source register to form an "effective address." The offset value is typically an offset onto the stack (r1 is the stack pointer), or it is an offset to the TOC (Table of Contents -- r2 is the TOC pointer). The TOC is used to promote the construction of position-independent code, which enables efficient dynamic loading of shared libraries on these machines.

When using inline asm, you do not have to use specific registers nor manually construct effective addresses. The argument constraints are used to direct the compiler to choose registers or construct effective addresses appropriate to the requirements of the instructions. Thus, if a general register is required by the instruction, you could use either the "r" or "b" constraint. The "b" constraint is of interest, because many instructions use the designation of register 0 specially -- a designation of register 0 does not mean that r0 is used, but instead a literal value of 0. For these instructions, it is wise to use "b" to denote the input operands to prevent the compiler from choosing r0. If the compiler chooses r0, and the instruction takes that to mean a literal 0, the instruction would produce incorrect results.

**Listing 8. r0 and its special meaning in the stbx instruction**
char res[8]={'a','b','c','d','e','f','g','h'};
char a='y';
int index=7;
asm (" stbx %0,%1,%2 
     : "r"(a), "r"(index), "r"(res) );

Here, the expected result string is abcdefgy, but if the compiler chose \texttt{r0} for \%1, then the result would incorrectly be ybcdefgh. To prevent this from happening, use "b" as in Listing 9 shows.

**Listing 9. Using "b" constraint to signify non-zero GPR**

Another example is in the following ASM block. While it appears that the asm block below does res=res+4, that is not the actual functional behavior of the code.

**Listing 10. Meaning of \texttt{r0} in the second operand with addi opcode**

int register res asm("r0")=5;
int b=4;
asm (" addi %0,%0,%1 
      : "r"(res) 
      : "i"(b) 
      : "r0" );

where:
addi %0(result operand),%0(input operand res),%3(immediate operand b)

Because \texttt{res} is tied to \texttt{r0}, the translation of the asm code in assembly looks becomes:

\texttt{addi \_0,\_0,4}

The second operand does not translate to register zero. Instead, it translates to the immediate number zero. In effect, the following is the result of the addi operation:

\texttt{res=0+4}

This case is special to the \texttt{addi opcode}. If, instead, \texttt{res} was tied to \texttt{r1}, then the original intended behavior would have been obtained:

\texttt{res=res+4}

**Clobbers list**

**Basic clobbers list**

In cases when registers that are not directly tied to the inputs/outputs are used within the asm block, the user must specify such registers within the clobbers list.
The *clobbers list* is used to notify the compiler that the registers contained within the list can potentially have their values altered. Hence, they should not be used to hold other data other than for the instructions that they are used for.

In the example in Listing 11, registers 8 and 7 are added to the clobbers list because they are used in the instructions but are not explicitly tied to any of the input/output operands. Also, condition register field zero is added to the clobbers list for the same reason. Although it is not present in the input/output operands, the `mfocrf` instruction reads that bit from the condition register and moves the value in register 8.

**Listing 11. Clobbers list example**

```asm
asm (" addc. %0,%2,%3              
" mfocrf   8,0x1                
" andi.   7,8,0xF                
" stw    7,%1                    
: "=r"(res),"=m"(c_bit)
  : "b"(a), "r"(b)
  "r0","r7","r8","cr0"      );        clobbers list
```

If, instead, the `mfocrf` instruction read from condition register field 1 (cr1), then that field would need to be added to clobbers list instead. Also, the period [full stop] at the end of the `addc.` and `andi.` instructions means their results are compared to zero, and the result of the comparison is stored in condition register field 0.

When clobbered registers are omitted from the clobbers list, the results from the asm operations might not be correct. This is because such clobbered registers might be reused to hold intermediate values for other operations. Unless the compiler detects that those registers are clobbered, the intermediate data can be used to perform the programmer's instructions, with inaccurate results. Also, the user's asm instructions may clobber values used by the compiler.

**Exceptions to the clobbers list**

Nearly all registers can be clobbered, except for those listed in Table 1.

**Table 1. Registers that cannot be clobbered**

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td>stack pointer</td>
</tr>
<tr>
<td>r2</td>
<td>toc pointer</td>
</tr>
<tr>
<td>r11</td>
<td>environment pointer</td>
</tr>
<tr>
<td>r13</td>
<td>64 bit mode thread local data pointer</td>
</tr>
<tr>
<td>r30</td>
<td>often used by the compiler as a stack frame pointer, pointer to constant area</td>
</tr>
<tr>
<td>r31</td>
<td>often used by the compiler as a stack frame pointer, pointer to constant area</td>
</tr>
</tbody>
</table>
Memory clobbers
Memory clobber implies a fence, and it also impacts how the compiler treats potential data aliases. A memory clobber says that the asm block modifies memory that is not otherwise mentioned in the asm instructions. So, for example, a correct use of memory clobbers would be when using an instruction that clears a cache line. The compiler will assume that virtually any data may be aliased with the memory changed by that instruction. As a result, all required data used after the asm block will be reloaded from memory after the asm completes. This is much more expensive than the simple fence implied by the "volatile" attribute (discussed later).

Remember, because the memory clobber says anything might be aliased, everything that is used needs to be reloaded after the asm, regardless of whether it had anything to do with the asm. A memory clobber can be added to the clobbers list by simply using the "memory" word instead of a register name.

Branching
Basic branching
Branching can be tricky with inline asm, this is because you need to know the address of the instruction to which to branch before compile time. Although this is not possible, you can use labels. Using labels, the branch-to address can be designated with a unique identifier that can be used as a target branch address.

Within a single source file, labels cannot be repeated within an inline asm block, nor within neighboring asm blocks within the same source. In a given program, each label is unique. There is an exception to this rule, however, and this is if you use relative branching (more on this later). With relative branching, more than one label with the same identifier can be found within the same program and within the same asm block.

Note: Labels cannot be used in asm to define macros because of possible namespace clashes.

In the example in Listing 12, the branch occurs when the LT bit, bit 0, of the condition register is set. If it is not set, then the branch is not taken.

Listing 12. Example of branch taken when LT bit of CR0 is set (0x80000000)

```asm
" addic. %0,%2,%4 
" bc  0xC,0,here 
" there: add %1,%2,%3 
" here: mul %0,%2,%3 : "r"(res),"r"(res2) : "r" (a),"r"(b),"r"(c) : "cr0" );
```

Likewise, a branch would occur if the GT bit (bit 1) of the condition register is set, as in the code in Listing 13.

Listing 13. Example of branch taken when GT bit of CR0 is set (0x40000000)
With inline asm, it is perfectly legal to branch within the same asm block; however, it is not possible to branch between different asm blocks, even if they are contained within the same source.

Relative branching

As discussed earlier, relative branching allows you to reuse the name of a label more than once within the same program. It is predominantly used, however, to dictate the position of the target address relative to the branch instruction. These are examples of the relative branch codes that can be used:

- F - forward
- B - backward

Note:
That they must be suffixed to numeric labels to be syntactically correct.

In this example (Listing 14), notice that the target address is referenced as "Hereb". In this case, we use the label of the target address appended with a suffix that dictates where this label is located relative to the branch instruction itself. The label "Here" is located before the branch instruction, hence the use of the "b" suffix in "Hereb."

Listing 14. Needs caption

```
asm (     
          "  addic. %0,%2,%4       \n"     
          "  bc     0xC,1,here     \n"     
          "  there: add %1,%2,%3   \n"     
          "  here:  mul %0,%2,%3   \n"     
          :  ":r"(res),":r"(res2)      
          :  ":r"(a),":r"(b),":r"(c)     
          :  "cr0" );
```

The condition register

The condition register is used to capture information on results of certain instructions.

For non-floating point instructions with period (.) suffixes that set the CR, the result of the operation is compared to zero.

- If the result is greater than zero, then bit 1 of the CR field is set (0x4).
- If it is less than zero, then bit 0 is set (0x8).
- If the result is equal to zero, then bit 2 is set (0x2).
For all compare instructions, the two values are compared, and any CR field can be set (not just CR0). Table 2 lists the bits and their corresponding meanings (there are eight such sets of 4 bits in the condition register, called "cr0, cr1, cr2 ... cr7").

Table 2. Bits of a CR field and the meanings of different settings

<table>
<thead>
<tr>
<th>Bit</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LT</td>
<td>RA &lt; 0</td>
</tr>
<tr>
<td>1</td>
<td>GT</td>
<td>RA &gt; 0</td>
</tr>
<tr>
<td>2</td>
<td>EQ</td>
<td>RA = 0</td>
</tr>
<tr>
<td>3</td>
<td>U</td>
<td>Overflow for integer operations.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Unordered, for floating point operations</td>
</tr>
</tbody>
</table>

Note: For floating point instructions with a period suffix, CR1 is set to the upper 4 bits of the FPSCR.

Blocking the Volatile attribute

Making an inline asm block "volatile" as in this example, ensures that, as it optimizes, the compiler does not move any instructions above or below the block of asm statements.

```
asm volatile(" addic. %0,%1,%2
":"r"(res):"=r"(a),"r"(a))
```

This can be particularly important in cases when the code is accessing shared memory. This will be illustrated in the next section on multithreaded locking.

Multithreaded locking

One of the most common uses of inline asm is in writing short segments of instructions to manage multithreaded locks. Because of the loose memory model on the POWER architecture, constructing such locks requires careful use of a pair of instructions:

- One instruction that loads the lock word and creates a "reservation"
- Another that updates the lock word if the reservation hasn't been lost in the interim

Note:
If the reservation has been lost, a loop can be used to retry repeatedly.

Listing 15 shows a basic inline function that attempts to acquire a lock (there are several problems with this code, which we discuss after these examples).

Listing 15. Example of Acquire lock function coded in asm
inline bool acquireLock(int *lock){
    bool returnvalue = false;
    int lockval;
    asm {
        "0: lwarx %0,0,%2  \n" //load lock and reserve
        " cmpwi 0,%0,0   \n" //compare the lock value to 0
        " bne 1f         \n" //not 0 then exit function
        " ori %0,%0,1    \n" //set the lock to 1
        " stwcx. %0,0,%2 \n" //try to acquire the lock
        " bne 0b         \n" //reservation lost, try again
        " ori  %1,%1,1   \n" //set the return value to true
        "1:                \n" //didn't get lock, return false
        : "+r" (lockval), "+r" (returnvalue)
        : "r"(lock)       //parameter lock is an address
        : "cr0" );               //cmpwi, stwcx both clobber cr0
    return returnvalue;
}

Listing 16 is an example of how this inline function could be used.

**Listing 16. Example of how the acquireLock function can be used**

```c
if (acquireLock(lockWord)){
    //begin to use the shared region
    temp = x + 1;
    .  .  .
}
```

Because the function is inline, the resulting code won’t have an actual call in it. Instead, it will precede the use of the shared region `x` with the instructions to acquire the lock.

The first problem to notice with this code is the lack of a synchronization instruction. One of the key performance enhancements enabled by the loose memory model of the POWER architecture is the ability of the machine to reorder loads and stores to make more efficient use of internal pipelines. However, there are times when the programmer needs to curtail this reordering to some degree to properly access shared storage. In the case of a lock, you would not want a load of data from the shared region (“x” in the case above) to be reordered so that it occurs before the lock on the region is acquired. For this reason, a synchronization instruction should be inserted to tell the machine to limit reordering in this case. The `sync` instruction is often used for this purpose, but there are others available, as described in the POWER ISA (see **Resources**). In the code example in Listing 17, we inserted `sync` instruction to prevent reordering of loads of "x" (this is called an "import barrier"):  

**Listing 17. Sync example**
inline bool acquireLock(int *lock) {
    bool returnvalue = false;
    int lockval;
    asm {
        "0: lwarx %0,0,%2 \n" //load lock and reserve
        " cmpwi 0,%0,0 \n" //compare the lock value to 0
        " bne 1f \n" //not 0 then exit function
        " ori %0,%0,1 \n" //set the lock to 1
        " stwcx. %0,%0,2 \n" //try to acquire the lock
        " bne 0b \n" //reservation lost, try again
        " sync \n" //import barrier
        " ori %1,%1,1 \n" //set the return value to true
        "1: \n" //didn't get lock, return false
            
            : "+r" (lockval), "+r" (returnvalue)
            : "r"(lock) //parameter lock is an address
            : "cr0" ); //cmpwi, stwcx both clobber cr0
        return returnvalue;
    }
}

In that asm block, the sync will prevent any subsequent loads from occurring until after it is known which way the preceding branch went. That way the variable x will not be loaded unless the branch was not taken and the acquireLock returns true.

So, are we set now? Unfortunately not. We still have to worry what the compiler might do.

Modern optimizing compilers can be very aggressive in moving code around -- and even removing it completely -- if it appears that the changes might make the program run faster without changing the semantics of the code. However, compilers typically aren't aware of the complexities involved with accessing shared memory. For example, a compiler might move the statement temp = x + 1; to a place higher in the program if it determines that the result would be scheduled more efficiently (and it assumes that the "if" is usually taken). Of course, that would be disastrous from the viewpoint of accessing shared data. To prevent the movement of any loads (or any instructions at all) from below the inline asm to a location above it, you can use the keyword "volatile" (also known as the volatile attribute) to modify the asm block, as Listing 18 shows.

**Listing 18. Volatile keyword example**

inline bool acquireLock(int *lock) {
    bool returnvalue = false;
    int lockval;
    asm volatile {
        "0: lwarx %0,0,%2 \n" //load lock and reserve
        " cmpwi 0,%0,0 \n" //compare the lock value to 0
        " bne 1f \n" //not 0 then exit function
        " ori %0,%0,1 \n" //set the lock to 1
        " stwcx. %0,%0,2 \n" //try to acquire the lock
        " bne 0b \n" //reservation lost, try again
        " sync \n" //import barrier
        " ori %1,%1,1 \n" //set the return value to true
        "1: \n" //didn't get lock, return false
            
            : "+r" (lockval), "+r" (returnvalue)
            : "r"(lock) //parameter lock is an address
            : "cr0" ); //cmpwi, stwcx both clobber cr0
        return returnvalue;
    }
}

When you do this, an internal fence is placed before and after the asm block that prevents instructions from being moved past it. And remember that this asm block is inlined, so it will prevent the access to x from being moved above the asm-implemented lock.
Memory clobbers in multithreaded locking

The discussion of multithreaded locking would not be complete without a mention of memory clobbers. The keyword memory is often added to the clobber list in such situations, although it is not always clear why it would be needed. The use of memory in the clobbers list means that memory is altered unpredictably by the asm block.

However, memory modifications in the locking example given are quite predictable. Although the variable lock is a pointer (that points to a lock location), that isn't any more unpredictable that the expression "*lock" in a C program. In that case, a well-behaved compiler would likely associate the expression "*lock" with all variables of the appropriate type, and so would correctly reload any affected variables after the pointer was used for modifying data. Nonetheless, the use of memory clobbers appears to be a pervasive practice, which is probably driven by an abundance of caution when dealing with shared regions. Programmers should be aware, though, of the performance penalties involved and of alternative approaches.

When an inline asm includes "memory" in the clobbers list, it means that any variable in the program might have been modified by the asm, so it must be reloaded before it is used. This requirement can pretty much put a sledgehammer to optimization efforts by the compiler. A potentially lighter-weight approach would be to make the shared region volatile (in addition to the asm block itself). Making a variable volatile means its value must be reloaded before it is used in any given expression. If the shared region in question is a data structure, such as a list or queue, this will ensure that the updated structure is reloaded after the lock is acquired. However, all of the non-shared data accesses can enjoy the full complement of compiler optimizations.

Tip:

If the shared data structure is accessed by a pointer (say *p), be sure to declare the pointer so that you indicate that it's the object pointed to that is volatile, not the pointer itself. For example, this declares that the list pointed to by p is volatile:

```c
volatile list *p
```

Acknowledgments

Thank you Ian McIntosh, Christopher Lapkowski, Jim McInnes, and Jae Broadhurst. You've each played an important role in publishing this article.

© Copyright IBM Corporation 2011
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)