CS:APP3e Web Aside ASM:EASM: Combining Assembly Code with C Programs[*]

Randal E. Bryant
David R. O'Hallaron

December 29, 2014

Notice

1 Combining Assembly Code with C Programs

In the early days of computing, most programs were written in assembly code. Even large-scale operating systems were written without the help of high-level languages. This becomes unmanageable for programs of significant complexity. Since assembly code does not provide any form of type checking, it is very easy to make basic mistakes, such as using a pointer as an integer rather than dereferencing the pointer. Even worse, writing in assembly code locks the entire program into a particular class of machine. Rewriting an assembly language program to run on a different machine can be as difficult as writing the entire program from scratch.

Aside: Writing large programs in assembly code.
Frederick Brooks, Jr., a pioneer in computer systems wrote a fascinating account of the development of OS/360, an early operating system for IBM machines [1] that still provides important object lessons today. He became a devoted believer in high-level languages for systems programming as a result of this effort. **End Aside.**

Early compilers for higher-level programming languages did not generate very efficient code and did not provide access to the low-level data representations, as is often required by systems programmers. Programs

[*]Copyright © 2015, R. E. Bryant, D. R. O'Hallaron. All rights reserved.

requiring maximum performance or requiring low-level access to data structures were still often written in assembly code. Nowadays, however, optimizing compilers have largely removed performance optimization as a reason for writing in assembly code. Code generated by a high quality compiler is generally as good or even better than what can be achieved manually. The C language has largely eliminated low-level data structure access as a reason for writing in assembly code. The ability to access low-level data representations through unions and pointer arithmetic, along with the ability to operate on bit-level data representations, provide sufficient access to the machine for most programmers. For example, almost all of the code for modern operating systems, including Linux, Windows, and MacOS, are written in C.

Nonetheless, there are times when writing in assembly code is the only option. This is especially true when implementing an operating system. For example, there are a number of special registers storing process state information that the operating system must access. There are either special instructions or special memory locations for performing input and output operations. Even for application programmers, there are some machine features, such as the values of the condition codes, that cannot be accessed directly in C.

The challenge then is to integrate code consisting mainly of C with a small amount written in assembly language. In this document, we will describe two such mechanisms. The first is to write a few key functions in assembly code, using the same conventions for argument passing and register usage as are followed by the C compiler. The linker then serves to combine the two forms of code into a single program. This approach is often feasible for simple functions, and it does not require any GCC-specific constructs. An alternative to writing an entire function in C is to embed assembly code within a C program. GCC supports *inline assembly* via the `asm` directive. Inline assembly allows the user to insert assembly code directly into the code sequence generated by the compiler. Features are provided to indicate to the compiler how to interface to the inserted code. The resulting code, of course, only runs on a specific class of machines, but we will see, for example, that it is often possible to have inline assembly that compiles properly on both IA32 and x86-64 machines. The `asm` directive is also specific to GCC, creating an incompatibility with many other compilers. Nonetheless, this can be a useful way to keep the amount of machine-dependent code to an absolute minimum.

Our presentation is drawn from the GCC documentation [2]. This is the authoritative reference, but it is fairly terse and contains very few examples.

2 Program Example

For our presentation, we will develop several implementations of functions with the following prototypes. These examples provide real-life cases where gaining access to the condition codes will enable us to monitor the status of a computation.

```
/* Multiply x and y.  Store result at dest.
   Return 1 if multiplication did not overflow
*/
int tmult_ok(long x, long y, long *dest);


/* Multiply x and y.  Store result at dest.
   Return 1 if multiplication did not overflow
```

```
*/
int umult_ok(unsigned long x, unsigned long y, unsigned long *dest);
```

Each function is to compute the product of arguments `x` and `y` and store the result in the memory location specified by argument `dest`. As return values, they should return 0 when the multiplication overflows, requiring more than 64 bits to represent the true product, and 1 when it does not. We have separate functions for signed and unsigned multiplication, since their overflow conditions differ. We examined ways to determine whether a multiplication has overflowed using C (see CS:APP2eProblem 2.35 and 2.36), but all of these methods require performing additional operations to check the result of a multiplication.

Examining the documentation for the x86 multiply instructions `mull` and `imull`, we see that both set the carry flag `CF` when they overflow. Thus, by inserting code that checks this flag after performing a multiplication of `x` and `y`, we should be able to easily test for multiplicative overflow.

3 Handwritten Assembly-Code Functions

Although writing complex programs entirely in assembly code is a daunting task, we can often narrow the amount of functionality that needs to be expressed in assembly code to a small amount and then write this code as functions in a separate file. The compiled C code is combined with the assembled assembly code by the linker. For example, if file `p1.c` contains C code and file `p2.s` contains assembly code, then the compilation command

```
linux> gcc -o p p1.c p2.s
```

will cause file `p1.c` to be compiled, file `p2.s` to be assembled, and the resulting object code to be linked to form an executable program `p`.

In order for the assembler to generate information required by the linker about a function, we must declare the function to be *global*. Whereas in C, any function is global unless it is declared to be static, the assembler assumes that a file is only locally available to functions within the same file, unless it is declared global. If we have assembly code for a function `fun` in a file, then we should precede it with the declaration

```
.globl fun
```

We have found that even when writing functions in assembly code, it is best to let GCC do as much of the work as possible. Toward that end, it often helps to write a function in C similar to the desired functionality and then generate an initial version of the assembly code by running GCC with the command-line option '`-S`.' This code provides a good starting point for fetching arguments, allocating and deallocating the stack frame, and so forth. It is much easier to edit existing assembly code than to start from scratch.

As an example, consider the following approximation to our function `tmult_ok_asm`:

```
/* Starter function for tmult_ok */
/* Multiply arguments and indicate whether it did not overflow */
int tmult_ok_asm(long x, long y, long *dest) {
    long p = x*y;
    *dest = p;
    return p > 0;
}
```

This function does much of what we want for `tmult_ok`—it multiplies arguments `x` and `y`, stores the product at `dest`, and returns a 0 or 1 based on the result. Its only shortcoming is that it checks the wrong property, but this is just one part of the overall computation.

GCC generates the following assembly code for the initial function We show this code in the exact form it appears in the file, rather than the more stylized way we have presented assembly code in the book, since we will want to edit the assembly code.

```
1          .globl  tmult_ok_asm
2  tmult_ok_asm:
3          imulq   %rdi, %rsi
4          movq    %rsi, (%rdx)
5          testq   %rsi, %rsi
6          setg    %al
7          movzbl  %al, %eax
8          ret
```

Note the presence of the `.globl` declaration in this code. Only two lines of this code need to be changed. Line 5 sets condition codes based on the 64-bit product of `x` and `y`. We want to eliminate this instruction and instead rely on the condition-code values set by the `imulq` instruction. Line 6 sets the low-order byte of register `%eax` based on the zero and sign flags. We want to set the byte based on the carry flag.

Examining CS:APP3e Figure 3.14, we see that the instruction `setae` can be used to set the low-order byte of a register to 0 when the carry flag is set and to 1 otherwise. We can therefore make small edits to the assembly code to get the desired function:

```
# Hand-generated code for tmult_ok
.globl tmult_ok_asm
# int tmult_ok_asm(long x, int y, long *dest);
# x in %rdi, y in %rsi, dest in %rdx
tmult_ok_asm:
        imulq   %rdi, %rsi
        movq    %rsi, (%rdx)
# Deleted code
#       testq   %rsi, %rsi
#       setg    %al
# Inserted code
        setae   %al             # Set low-order byte
# End of inserted code
        movzbl  %al, %eax
        ret
```

As this example shows, it helps to add annotations to the assembly code as documentation. Anything to the right of the symbol '#' is treated as a comment by the assembler.

Unfortunately, this approach of prototyping and modifying does not work for the implementation of `umult_ok_asm`, where unsigned arithmetic is required. We attempt to create an approximate version with the C code

```
/* Multiply arguments and indicate whether it did not overflow */
int umult_ok_asm(unsigned long x, unsigned long y, unsigned long *dest) {
    unsigned long p = x*y;
    *dest = p;
    return p > 0;
}
```

GCC generates the following assembly code for the initial function:

```
        .globl  umult_ok_asm
umult_ok_asm:
        imulq   %rdi, %rsi
        movq    %rsi, (%rdx)
        testq   %rsi, %rsi
        setne   %al
        movzbl  %al, %eax
        ret
```

The code is nearly identical to that for signed data! In particular, it still uses the signed multiply instruction `imulq`. As discussed in CS:APP3e Chapter 2, the low-order 64 bits of the product of two 64-bit products is identical, and so this code computes the correct product, but it does not set the condition codes in a way that will let us detect overflow.

Instead, we must write code that uses the unsigned multiply instruction `mulq`, but this instruction is only available in a one-operand format (CS:APP3e Figure 3.12). With the one-operand form, the other argument to the multiplication must be in register `%rax`, and the product is stored in registers `%rdx` (high-order 8 bytes) and `%rax` (low-order 8 bytes). Using this instruction requires writing code that looks very different from that generated by GCC:

```
# Hand-generated code for umult_ok
.globl umult_ok_asm
# int umult_ok_asm(unsigned long x, unsigned long y, unsigned long *dest);
# x in %rdi, y in %rsi, dest in %rdx
umult_ok_asm:
        movq    %rdx, %rcx      # Save copy of dest
        movq    %rsi, %rax      # Copy y to %rax
        mulq    %rdi            # Unsigned multiply by x
        movq    %rax, (%rcx)    # Store product at dest
        setae   %al             # Set low-order byte
        movzbl  %al, %eax
        ret
```

In both of these examples, we can see that our study of assembly code gives us the ability to write our own assembly-code functions.

4 Basic Inline Assembly

Using inline assembly lets GCC do more of the work in integrating hand-generated assembly code into a program.

The basic form of inline assembly is to write code that looks like a procedure call:

`asm(` *code-strings* `);`

The term *code-strings* denotes an assembly code sequence given as one or more quoted strings (with no delimiters between them.) The compiler will insert these strings verbatim into the assembly code being generated, and hence the compiler-supplied and the user-supplied assembly will be combined. The compiler does not check the string for errors, and so the first indication of a problem might be an error report from the assembler.

In an attempt to use the least amount of both assembly code and detailed analysis, we attempt to implement `tmult_ok` with the following code:

```
/* First attempt.  Does not work */
int tmult_ok1(long x, long y, long *dest)
{
    long result = 0;
    *dest = x*y;
    asm("setae %al");
    return result;
}
```

The strategy here is to exploit the fact that register `%eax` is used to store the return value. Assuming the compiler uses this register for variable `result`, our intention is that the first line of the C code will set the register to 0. The inline assembly will insert code that sets the low-order byte of this register appropriately, and the register will be used as the return value.

Unfortunately, the generated code does not work as desired. In running tests, it returns 0 every time it is called. On examining the generated assembly code for this function, we find the following:

```
   int tmult_ok1(long x, long y, long *dest)
   x at %rdi, y at %rsi, dest at %rdx
1 tmult_ok1:
2   imulq   %rsi, %rdi         Compute x * y
3   movq    %rdi, (%rdx)       Store at dest
   Code generated by asm
4   setae %al                  Set low-order byte of %eax
   End of asm-generated code */
5   movl    $0, %eax           Set %eax to 0
6   ret
```

GCC has its own ideas of code generation. Instead of setting register `%eax` to 0 at the beginning of the function, the generated code does so at the very end (line 5), and so the function always returns 0. The fundamental problem is that the compiler has no way to know what the programmer's intentions are, and how the assembly code should interface with the rest of the generated code. Clearly, more sophisticated mechanisms are required to embed assembly code within C code.

5 Extended Form of `asm`

GCC provides an extended version of `asm` that allows the programmer to specify which program values are to be used as operands to an assembly code sequence, and which registers are overwritten by the assembly code. With this information the compiler can generate code that will correctly set up the required source values, execute the assembly instructions, and make use of the computed results. It will also have information it requires about register usage so that important program values are not overwritten by the assembly code instructions.

The general syntax of an extended `asm` directive is

`asm(` *code-strings* [`:` *output-list* [`:` *input-list* [`:` *overwrite-list*]]] `);`

where the square brackets denote optional arguments. As with basic `asm`, the extended form contains one or more strings giving stylized versions of the lines of assembly code. These are followed by optional lists of *outputs* (i.e., results generated by the assembly code), *inputs* (i.e., source values for the assembly code), and registers that are overwritten by the assembly code. These lists are separated by the colon (‘`:`’) character. As the square brackets show, we only include lists up to the last nonempty list.

The syntax for a code string is reminiscent of that for the format string in a `printf` statement. It consists an assembly code instruction, but the operands are written in a symbolic form, with references to the operand expressions in the output and input lists. Within the assembly code, we give names to the operands. Earlier versions of GCC required these names to be of the from `%0`, `%1`, and so on, up to `%9`, based on the order of the operands in the two lists. Since GCC version 3.1, a more descriptive naming convention is supported, where names are written with the notation `%[`*`name`*`]`. Register names such as `%eax` must be written with an extra ‘`%`’ symbol, such as `%%eax`.

If multiple instructions are given, it is critical that return characters be inserted between them. The conventional method of doing this is to finish all but the final string with the sequence `\n\t`, so that the generated assembly code follows the normal formatting conventions for assembly code.

The output input list is a comma-separated list, with each list element containing three components in the form `[`*`name`*`]` *`tag`* `(`*`expr`*`)`, giving the name of the operand, the output constraint, and the C expression indicating the destination for the instruction result. The field *`tag`* specifies constraints on the use and location of the output operand, in the form of one of the following quoted strings:

Constraint	Meaning
`"=r"`	Update value stored in a register
`"+r"`	Read and update value stored in a register
`"=m"`	Update value stored in memory
`"+m"`	Read and update value stored in memory
`"=rm"`	Update value stored in a register or in memory
`"+rm"`	Read and update value stored in a register or in memory

The expression *`expr`* can be any assignable value (known in C as an *lvalue*). The compiler will generate the necessary code sequence to perform the assignment.

The input list entries have the same format, except that the tags are of the form tag "r", "m", or "rm", indicating that the operand will be read from a register, memory, or either, respectively. Each input operand can be any C expression. The compiler will generate the necessary code to evaluate the expression. The overwrite list consists of a comma-separated list of register names, with each in double quotes.

As an illustration, the following is a better implementation of `tmult_ok` using the extended assembly directive to indicate to the compiler that the assembly code generates the value for the variable `result`:

```
int tmult_ok2(long x, long y, long *dest)
{
    int result;

    *dest = x*y;
    asm("setae %%bl          # Set low-order byte\n\t"
        "movzbl %%bl,%[val]  # Zero extend to be result"
        : [val] "=r" (result)  /* Output    */
        :                      /* No inputs */
        : "%bl"                /* Overwrites */
        );
    return result;
}
```

We see that the `asm` directive has two code strings: one for the `setae` instruction, and one to zero-extend the low-order byte to form the result. We can see that we have chosen register `%bl` as the destination of the `setae` instruction and as the source of the `movzbl` instruction. The register operand must be written as `%%bl`. Observe also that we can add comments to our code, and that all but the last lines are terminated with `\n\t`. We have given the name `val` to indicate the final value generated by the code. This is shown in the output list as corresponding to program variable `result`. We also indicate that our code will overwrite register `%bl`.

When we compile this code for x86-64, GCC generates the following assembly code:

```
  int tmult_ok2(long x, long y, long *dest)
  x at %rdi, y at %rsi, dest at %rdx
1 tmult_ok2:
2   pushq   %rbx                Save %rbx
3   imulq   %rsi, %rdi          Compute x * y
4   movq    %rdi, (%rdx)        Store at dest
  Code generated by asm
5   setae %bl                   Set low-order byte
6   movzbl %bl,%eax             Zero extend %eax
  End of asm-generated code
7   popq    %rbx                Restore %rbx
8   ret
```

We see that the program saves the value of `%rbx` on the stack (line 2) and restores it at the end (line 7.) Since this register is a callee-saved register, and we have indicated that our code will overwrite its low-order byte (register `%bl`), GCC takes the necessary steps to preserve its value.

As a further refinement, we can simplify the code even more and make use of GCC's ability to work with different data types. GCC uses the type information for an operand when determining which register to substitute for an operand name in the code string. In the version given as function `tmult_ok2`, it used a 32-bit register `%eax`, based on the fact that variable `result` has data type `int`. Instead, we can use a variable `bresult` of type `unsigned char` in the output list, and have this operand be the destination of the `setae` instruction:

```
/* Uses extended asm to get reliable code */
int tmult_ok3(long x, long y, long *dest)
{
    unsigned char bresult;
    *dest = x*y;

    asm("setae %[b]             # Set result"
        : [b] "=r" (bresult)  /* Output    */
        );

    return (int) bresult;
}
```

The compiler will use a single-byte register identifier as the destination for the `setae` instruction, and then use this register as the source operand of a `movzbl` instruction to implement the casting of `bresult` to data type `int`. This simplified form also avoids the need for us to make use of a specific register, and hence we need not specify any overwritten registers.

Writing inline assembly to implement `umult_ok` is more involved. We must set up, use, and obtain the result from the one-operand `mulq` instruction, as follows:

```
int umult_ok(unsigned long x, unsigned long y, unsigned long *dest)
{
    unsigned char bresult;

    asm("movq %[x],%%rax      # Get x\n\t"
        "mulq %[y]            # Unsigned long multiply by y\n\t"
        "movq %%rax,%[p]      # Store low-order 8 bytes at dest\n\t"
        "setae %[b]           # Set result"
        : [p] "=m" (*dest), [b] "=r" (bresult) /* Outputs    */
        : [x] "r"  (x), [y] "r" (y)            /* Inputs     */
        : "%rax", "%rdx"                       /* Overwrites */
        );

    return (int) bresult;
}
```

This code makes use of many of the features of the extended `asm` directive. The two output operands are given symbolic names `p` (the product) and `b` (the status byte), while the two input operands have symbolic names `x` and `y`. We can see that output operand `p` is associated with the expression `*dest` and is specified as being stored to memory, while `b` is associated with local variable `bresult` and is specified as being held in a register. We need to list both registers `%rax` and `%rdx` on the overwrite list.

To see how the compiler generates code in connection with an `asm` directive, here is the code generated for `umult_ok`:

```
int umult_ok(unsigned long x, unsigned long y, unsigned long *dest)
x at %rdi, y at %rsi, dest at %rdx
umult_ok:
  movq    %rdx, %rcx      Save dest
Code generated by asm
  movq %rdi,%rax          Get x
  mulq %rsi               Unsigned long multiply by y
  movq %rax,(%rcx)        Store low-order 8 bytes at dest
  setae %dil              Set low-order byte
End of asm-generated code
  movzbl  %dil, %eax      Zero-extend result
  ret
```

We can see that GCC has chosen the following register allocations for the operands: `%rdi` for `x`, `%rsi` for `y`, `%rdi`, and `%dil` for `b`.

6 Concluding Remarks

We have explored two ways to combine assembly code with C code to generate a program. Writing a complete function in assembly code as a separate file has the advantage that it uses existing and familiar technology: the assembler and the linker. Using the facility for GCC to insert assembly code directly in a C function has the advantage that we can greatly limit the amount of machine-specific code.

Although the syntax of the `asm` directive is somewhat arcane, and its use makes the code less portable, it can be very useful for writing programs that access machine-level features using a minimal amount of assembly code. We have found that a certain amount of trial and error is required to get code that works. The best strategy is to compile the code with the `-S` command-line option and then examine the generated assembly code to see if it will have the desired effect. It is important to note that in processing an `asm` directive, GCC has no real understanding of the assembly code it is generating. It merely follows a set of syntactic rules for replacing the symbolic names of operands with different register identifiers and memory references. The code should be tested with different settings of switches such as with different levels of optimization.

Practice Problem 1:

Use extended `asm` to write a version of `tmult_ok` that includes the `imulq` instruction as part of the assembly code. (It can be argued that this approach will be more robust across different versions of GCC and optimization levels, since it will guarantee that the correct form of multiplication is being used.) Try to write the `asm` code in a way that compiles to machine code that does not involve any unnecessary movement of data among registers.

Practice Problem 2:

Use the `asm` directive to implement an x86-64 function with the prototype

```
/* Multiply two 64-bit numbers to get 128-bit result,
*/
void umult_full(unsigned long x, unsigned long y, unsigned long *dest);
```

This function should compute the full 128-bit product of its arguments and store the result in the destination array, with `dest[0]` having the low-order eight bytes and `dest[1]` having the high-order eight bytes.

Practice Problem 3:

As is described in CS:APP3e Chapter 13, a program consisting of multiple independent threads of control requires special attention when these threads read and write a single program variable. This can lead to *data races*, where the different threads interfere with one another.

For example, suppose a program has t threads running, each of which invokes the following function n times, and where the argument `ptr` for every thread points to a single global variable `cnt`:

```
void bad_incr(int *ptr)
{
    (*ptr)++;
}
```

Assuming `cnt` is initially 0, one would expect its final value to be $t \cdot n$. Instead, it is typically less, because not all of the updates have the effect of incrementing `cnt`.

Executing the `addl` instruction in the function involves three steps:

Read: Read the value stored in memory into the CPU

Modify: Increment the value

Write: Store the incremented value back to memory

Suppose that two threads both execute their respective `addl` instructions around the same time. Then they may both read the same value C of `cnt`. Each will then compute $C + 1$ and store this value back to `cnt`. So, even though two `addl` instructions have been executed, the net effect is to only increment `cnt` by 1.

CS:APP3e Chapter 13 describes how to use special synchronization primitives to ensure that only one thread has access to shared program state at a time. These primitives involve calls to the operating system and incur a significant overhead.

A more direct solution is to take advantage of a feature implemented in x86 processors that allows some instructions, including `addl` to be executed *atomically*, guaranteeing that the read-modify-write steps of the instruction take place without any other thread being able to read or write the memory location while these steps are taking place. The implementation of this atomicity is done completely in hardware and so is significantly faster than using synchronization primitives.

To specify that an instruction should be executed atomically, a special lock prefix byte (with code `0xF0`) is included in the machine code immediately before the instruction. This is written in assembly by writing `lock` as if it were an instruction on a separate line before the instruction that is to be executed atomically. Use the extended `asm` in a function `lock_incr` such that the generated assembly code will be exactly like that for `bad_incr`, except that the `addl` instruction will be executed atomically.

Practice Problem 4:

X86 machines have a parity flag PF as one of the condition codes. Every arithmetic or logical operation sets this flag when the low-order eight bits of a result have even parity, meaning that they contain an even number of ones. Whether the operation computes an 8-bit result or a larger one, the parity flag depends only on the low-order 8 bits.

Consider the following function prototype:

```
int odd_parity(unsigned long x);
```

This function should determine whether its argument has odd parity, meaning that it contains an odd number of ones. We want this function to operate correctly when compiled for either IA32, in which case the argument is 32 bits, or for x86-64, in which case the argument is 64 bits.

Write a C function including an `asm` directive to implement this function, using the parity flag to compute the parity 8 bits at a time.

Practice Problem 5:

Extended `asm` can be used for inserting floating-point instructions into a program. In the constraint field, the character ‘`x`’ is used to indicate an XMM register as either input or output.

Write code for the following function:

```
/* Return minimum of x and y */
double dmin(double x, double y);
```

Your code should use the `vminsd` instruction (see CS:APP3e Figure 3.48) to perform the computation.

Practice Problem 6:

Write code for the following function:

```
/* Return sqrt(x) */
double dsqrt(double x);
```

Your code should use the `sqrtsd` instruction (see CS:APP3e Figure 3.48) to perform the computation.

Solutions to Practice Problems

Problem 1 Solution: [Pg. 10]

The trick to writing this in a way that doesn’t involve additional data movement is to designate either `x` or `y` as the destination of the multiplication operation, and use the constraint `"+r"`:

```
/* Uses extended asm with explicit inclusion of imulq instruction */
int tmult_ok4(long x, long y, long *dest)
{
    unsigned char bresult;

    asm("imulq %[x],%[y]    # Compute y = x * y\n\t"
        "setae %[b]         # Set result"
        : [y] "+r" (y), [b] "=r" (bresult) /* Outputs */
        : [x] "r" (x)                      /* Inputs */
        );
    *dest = y;
    return (int) bresult;
}
```

Problem 2 Solution: [Pg. 11]

This function bears many similarities to the function `umult_ok`, except that we want to store both the high and the low-order words of the product:

```
void umult_full(unsigned long x, unsigned long y, unsigned long *dest)
{
    asm("movq %[x],%%rax       # Get x\n\t"
        "mulq %[y]             # Unsigned multiply by y\n\t"
        "movq %%rax,%[lo]      # Store low-order  8 bytes\n\t"
        "movq %%rdx,%[hi]      # Store high-order 8 bytes"
        : [lo] "=m" (dest[0]), [hi] "=m" (dest[1]) /* Outputs    */
        : [x] "r"  (x), [y] "r" (y)                /* Inputs     */
        : "%rax", "%rdx"                           /* Overwrites */
        );
}
```

Problem 3 Solution: [Pg. 11]

This problem demonstrates a case where inline assembly can tap into a feature of the hardware that cannot be done with normal C code. In our experiments with two threads, we found that using hardware locking slows down execution by 6.4X compared to `bad_incr`. Using synchronization primitives slows down execution by 31X or more.

Our solution uses the constraint `"+m"` to specify that the value of `*ptr` will be both read and used as an output.

```
void lock_incr(int *ptr)
{
    asm("lock             # Insert lock prefix\n\t"
        "addl $1, %[p]    # Increment *ptr"
        : [p] "+m" (*ptr) /* Output */
        );
}
```

Problem 4 Solution: [Pg. 12]

There are many solutions to this problem. This one is perhaps the simplest. It simply shifts the successive bytes of argument `x` into the low-order byte and thereby tests the parity of each non-zero byte. We use the `testb` instruction to test each byte, since the parity flag depends only on the low-order byte, and this makes the code portable between IA32 and x86-64.

```
/* Using ASM to access parity flag */
int odd_parity(unsigned long x) {
    int result = 0;
    while (x != 0) {
        char bresult;
        unsigned char bx = x & 0xff;
        asm("testb %[bx],%[bx] # Test value of low-order byte\n\t"
            "setnp %[v]        # Set if odd parity"
            : [v] "=r" (bresult)  /* Output */
            : [bx] "r" (bx)       /* Input  */
            );
        result ^= (int) bresult;
        x = x >> 8;
    }
    return result;
}
```

Problem 5 Solution: [Pg. 12]

This is another demonstration of tapping into hardware features of the processor in ways that cannot be done with normal C code.

```
/* Return minimum of x and y */
double dmin(double x, double y) {
    double result;
    asm("vminsd %[x], %[y], %[r]   # Compute r = min(x,y)\n\t"
        : [r] "=x" (result)         /* Outputs */
        : [x] "x" (x), [y] "x" (y)  /* Inputs */
        );
    return result;
}
```

Problem 6 Solution: [Pg. 12]

Normally, this operation is implemented in C by calling the library function `sqrt`. This function uses the `sqrtsd` instruction to perform the computation.

```
/* Return sqrt(x) */
double dsqrt(double x) {
    double result;
    asm("sqrtsd %[x], %[r]   # Compute r = sqrt(x)\n\t"
        : [r] "=x" (result)         /* Output */
        : [x] "x" (x)               /* Input */
        );
    return result;

}
```

References

[1] Jr. F. P. Brooks. *The Mythical Man-Month, Second Edition*. Addison-Wesley, 1995.

[2] *GCC Online Documentation*. Available at `http://gcc.gnu.org/`.