Application Report
SPRA683 - June 2000

How to Write Multiplies Correctly in C Code

Brett Huber *Digital Signal Processing Solutions*

ABSTRACT

Writing multiplication expressions in C code so that they are both correct and efficient can be confusing, especially when technically illegal expressions can, in some circumstances, generate the code the user wanted in the first place. This document will help you chose the correct expression for your algorithm.

Contents

1 Introduction

Writing multiplication expressions in C code so that they are both correct and efficient can be confusing, especially when technically illegal expressions can, in some circumstances, generate the code the user wanted in the first place. This document will help you chose the correct expression for your algorithm.

In short, the correct expression for a 16x16–>32 multiply is (with appropriate typedefs):

INT32 res = (INT32)(INT16)src1 * (INT32)(INT16)src2;

According to the C arithmetic rules, this is actually a 32x32–>32 multiply, but the compiler will notice that both operands fit in 16 bits, so it will issue a single-instruction multiplication.

[The notation “16x16” means a 16-bit multiply. “16x16–>32” means a 16-bit multiply with a 32-bit result, an operation which does not directly exist in the C language, but does exist on most of our DSPs, and is vital for MAC-like algorithm performance.]

The C issues involved in explaining why the above expression is the correct one include:

- Signed vs. unsigned arithmetic
- Signed arithmetic overflow
- Undefined behavior
- Integral promotions
- Arithmetic conversions
- Implicit vs. explicit conversions

2 Signed vs. Unsigned Arithmetic, Overflow

Although signed and unsigned artithmetic typically appear identical at the assembly code level, they are not equivalent operations at the C level. They differ mainly in treatment of arithmetic overflow, or when the result of an expression is larger than can be represented by the type of the expression.

The standard mandates that unsigned arithmetic operations be performed using modulo arithmetic (results are truncated to the size of the type). This means that when a 16x16 unsigned multiply overflows, the operation must be performed as if it were a 16x16–>32 multiply, followed by truncation to the lower 16 bits.

Ex: (with 16-bit int): assert(0x8001 * 2 == 2);

On the other hand, the standard dictates that when signed arithmetic overflows, “undefined behavior” is invoked. This generally allows the compiler to produce more efficient code, as it does not need to explicitly check for or handle overflow. However, this means that the programmer must be more careful when using signed arithmetic, or the program might fail.

3 Undefined Behavior

Undefined behavior is a certain class of erroneous program behavior for which the compiler does not guarantee any particular result when the program is executed. There are many instances in C in which a program might invoke undefined behavior, such as signed arithmetic overflow, or dereferencing a NULL pointer.

At the point in time undefined behavior is invoked, all aspects of the program become undefined, which means that the program might begin doing anything from rebooting the computer to doing what the programmer expected in the first place.

It is the programmer’s responsibility to avoid invoking undefined behavior; it’s generally impossible for the compiler to determine whether a program invokes undefined behavior, particularly those which accept input or process data.

4 Integral Promotions, Arithmetic Conversions

In C, all arithmetic operands of size less than int are converted to int or unsigned int before computing the expression. This is called the “integral promotions.”

Most arithmetic operators (including ’*’) require that both operands be of the same type. This means that one or both of the operands may need to be converted to a larger type to compete the operation, which the compiler will do for you automatically. For example, the expression “(int)x * (long)y” will be converted to “(long)(int)x * (long)y,” and then the operation will proceed as a long operation. The rules become somewhat strange when mixing signed and unsigned types. This step is called the “arithmetic conversions.”

Both of these can be confusing, even for expert C programmers, so it is probably best to avoid the problem by making sure that the operand types agree, possibly with a cast operator.

5 Implicit vs. Explicit Conversions

Explicit conversions are cast operators. Other conversions, such as the arithmetic conversions and conversion upon assignment, are implicit conversions. Implicit conversions are equivalent to explicit conversions of the same type, but are performed automatically.

For instance, these two statements are equivalent:

```
long x = (short)y; /* implicit conversion to long */
long x = (long)(short)y; /* explicit conversion to long */
```

A common misconception about multiplication is the idea that multiplication takes place in infinite precision, and is then truncated to the result type. This is not so; the multiply will take place in the *narrowest* legal type (according to the types of the operands), and will then be *converted* to the result type, which may be a narrowing or widening conversion.

For example, the following statement is an “int” multiply, regardless of the fact that the destionation is long. The int result of the multiply is then converted to long.

```
long res = (int)src1 * (int)src2;
```

This statement is equivalent to:

```
long res = (long)((int)src1 * (int)src2);
```

If (src1 > INT_MAX / src2), this expression will overflow, invoking undefined behavior.

6 Putting It All Together

Multiplication expressions must peform the operation legally according to the C semantics, preferably with just one assembly instruction, if possible. For a 16x16–>32 multiply, this means the operation must be done as INT32 at the C level, but INT16 at the assembly level. With just a little help from the programmer, the TI compilers are smart enought to accomplish this. Both operands must be 32 bits:

```
 INT32 res = (INT32)src1 * (INT32)src2;
```

but if it is known that both operands are only 16-bit values, the compiler should be told so:

```
INT32 res = (INT32)(INT16)src1 * (INT32)(INT16)src2;
```

For this statement (with appropriate typedefs), the TI C compilers generate a single 16x16–>32 multiply, rather than the more expensive 32x32–>32 multiplication.

IMPORTANT NOTICE

Texas Instruments and its subsidiaries (TI) reserve the right to make changes to their products or to discontinue any product or service without notice, and advise customers to obtain the latest version of relevant information to verify, before placing orders, that information being relied on is current and complete. All products are sold subject to the terms and conditions of sale supplied at the time of order acknowledgment, including those pertaining to warranty, patent infringement, and limitation of liability.

TI warrants performance of its semiconductor products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements.

Customers are responsible for their applications using TI components.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards must be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance or customer product design. TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used. TI's publication of information regarding any third party's products or services does not constitute TI's approval, warranty or endorsement thereof.