CS:APP Web Aside DATA:TNEG:

Bit-Level Representation of Two’s Complement Negation

Randal E. Bryant

David R. O’Hallaron

August 27, 2016

Notice

The material in this document is supplementary material to the book Computer Systems, A Programmer’s Perspective, Third Edition, by Randal E. Bryant and David R. O’Hallaron, published by Prentice-Hall and copyrighted 2016. In this document, all references beginning with “CS:APP3e ” are to this book. More information about the book is available at csapp.cs.cmu.edu.

This document is being made available to the public, subject to copyright provisions. You are free to copy and distribute it, but you must give attribution for any use of this material.

* Introduction

There are several clever ways to determine the two’s complement negation of a value represented at the bit level. These techniques are both useful, such as when one encounters the value 0xfffffffa when debugging a program, and they lend insight into the nature of the two’s complement representation.

* Complement and Increment

One technique for performing two’s-complement negation at the bit level is to complement the bits and

then increment the result. In C, this can be written as ˜x + 1. To justify the correctness of this technique,

:

observe that for any single bit xi, we have ˜xi = 1 xi. Let ~x be a bit vector of length w and x = B2T w(~x) be the two’s-complement number it represents. By Equation CS:APP3e-2.3, the complemented bit vector ˜~x has numeric value

|  |  |  |
| --- | --- | --- |
| w 2 |  |  |
| Xi | xi)2i |  |
| B2T w(˜~x) = (1 xw 1)2w 1 + (1 |  |
| =0 |  |  |

Copyright c 2015, R. E. Bryant, D. R. O’Hallaron. All rights reserved.

1

2

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| ~x |  | ˜~x |  | incr(˜~x) | |
|  |  |  |  |  |  |
| [0101] | 5 | [1010] | 6 | [1011] | 5 |
| [0111] | 7 | [1000] | 8 | [1001] | 7 |
| [1100] | 4 | [0011] | 3 | [0100] | 4 |
| [0000] | 0 | [1111] | 1 | [0000] | 0 |
| [1000] | 8 | [0111] | 7 | [1000] | 8 |

Figure 1: **Examples of complementing and incrementing four-bit numbers.** The effect is to compute the two’s value negation.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| = | " 2w 1 + | w 2 | 2i# | " xw 12w 1 | + | w 2 xi2i# |
|  |  | X |  |  |  | Xi |
|  |  | i=0 |  |  |  | =0 |

* [ 2w 1 + 2w 1 1] B2T w(~x)
* 1 x

The key simplification in the above derivation is that Pw 2 2i = 2w 1 1. It follows that by incrementing

i=0

˜~x we obtain x.

:

To increment a number x represented at the bit-level as ~x = [xw 1; xw 2; : : : ; x0], define the operation incr

as follows. Let k be the position of the rightmost zero, such that ~x is of the form [xw 1; xw 2; : : : ; xk+1; 0; 1; : : : ; 1].

We then define incr(~x) to be [xw 1; xw 2; : : : ; xk+1; 1; 0; : : : ; 0]. For the special case where the bit-level

representation of x is [1; 1; : : : ; 1], define incr(~x) to be [0; : : : ; 0]. As illustrations, Figure 1 shows how complementing and incrementing affect the numeric values of several four-bit vectors.

To show that incr(~x) yields the bit-level representation of x +tw 1, consider the following cases:

:

1. When ~x = [1; 1; : : : ; 1], we have x = 1. The incremented value incr(~x) = [0; : : : ; 0] has numeric value 0.

2. When k = w 1, i.e., ~x = [0; 1; : : : ; 1], we have x = TMax w. The incremented value incr(~x) = [1; 0; : : : ; 0] has numeric value TMinw. From Equation CS:APP3e-2.13, we can see that TMax w +tw 1 is one of the positive overflow cases, yielding TMinw.

3. When k < w 1, i.e., x 6= TMax w and x 6= 1, we can see that the low-order k + 1 bits of incr(~x) has numeric value 2k, while the low-order k + 1 bits of ~x has numeric value Pk 1 2i = 2k 1. The

i=0

high-order w k + 1 bits have matching numeric values. Thus, incr(~x) has numeric value x + 1. In addition, for x 6= TMax w, adding 1 to x will not cause an overflow, and hence x +tw 1 has numeric value x + 1 as well.

Practice Problem 1:

Fill in the following table showing the effects of complementing and incrementing several five-bit vectors in the style of Figure 1. Show both the bit vectors and the numeric values.

3

|  |  |  |
| --- | --- | --- |
| ~x | ˜~x | incr(˜~x) |

[01101]

[01110]

[11000]

[11111]

[10000]

Practice Problem 2:

Show that first decrementing and then complementing is equivalent to complementing and then incre-menting. That is, for any signed value x, the C expressions -x, ˜x+1, and ˜(x-1) yield identical results. What mathematical properties of two’s-complement addition does your derivation rely on?

Complement Upper Bits

Another way to perform two’s complement negation of a number x is based on its bit-level representa-tion. Let k be the position of the rightmost 1, so that ~x, the bit-level representation of x has the form [xw 1; xw 2; : : : ; xk+1; 1; 0; : : : 0]. (This is possible as long as x 6= 0.) The negation is then written in binary form as [˜xw 1; ˜xw 2; : : : ˜ xk+1; 1; 0; : : : ; 0]. That is, we complement each bit to the left of bit position k. Showing that the resulting value indeed has value -t32 x is left as an exercise (Problem 6.)

We illustrate this idea with some 4-bit numbers, where we highlight the rightmost pattern 1; 0; : : : ; 0 in italics:

|  |  |  |  |
| --- | --- | --- | --- |
| x |  | x |  |
| [1100 ] | 4 | [0100 ] | 4 |
| [1000 ] | 8 | [1000 ] | 8 |
| [0101 ] | 5 | [1011 ] | 5 |
| [0111 ] | 7 | [1001 ] | 7 |

Practice Problem 3:

Show how the bit-level negation procedure applies to the examples of Problem 1. That is, 1) determine the bit position k of the rightmost 1, and 2) apply the rule of complementing the bits to the left of position

* .

Practice Problem 4:

You are given the task of writing a function with the following prototype:

/\*

* Generate mask indicating rightmost 1 in x.
* For example 0xFF00 -> 0x0100, and 0x6600 --> 0x0200.
* If x == 0, then return 0.

\*/

int rightmost\_one(unsigned x);

4

If argument x equals 0, this function returns 0. Otherwise, it returns a mask consisting of a single one in the same position as the least significant bit with value 1 in x.

Having just learned how to negate a number based on its bit-level representation, you realize this function can be written as a very simple expression having just two operations. Show the code.

Practice Problem 5:

We saw in Web Aside DATA:TMIN that trying to write TMin32 as an integer constant can expose some nuances of the C language, with results possibly depending on language version and word size.

Now that we are familiar with integer arithmetic, we can explore other possible ways of writing this constant. Below are six different expressions

|  |  |  |
| --- | --- | --- |
| 2147483647 | + 1 | /\* A. \*/ |
| 0x7FFFFFFF | + 1 | /\* B. \*/ |
| 2147483649 | - 1 | /\* C. \*/ |
| 0x80000001 | - 1 | /\* D. \*/ |
| -(2147483649 | - 1) | /\* E. \*/ |
| -(0x80000001 | - 1) | /\* F. \*/ |

Suppose that we compile the code on a machine that uses a 32-bit, two’s complement representation of data type int, and that the compiler implements ISO-C99. For each of these, determine:

1. What would be the resulting data type of the expression?
2. What would be the resulting numeric value?
3. Would we get TMin32 if we cast the value to type int?

Practice Problem 6:

We claimed that we could generate the two’s complement negation of a number x, having bit-level representation ~x, by finding bit position k such that ~x has the form xw 1; xw 2; : : : ; xk+1; 1; 0; : : : 0 The negation is then written in binary form as [˜xw 1; ˜xw 2; : : : ˜ xk+1; 1; 0; : : : ; 0].

Show that the resulting value indeed has value -t32 x.

Solutions to Problems

Problem 1 Solution: [Pg. 2]

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| ~x |  | ˜~x |  | incr(˜~x) | |
|  |  |  |  |  |  |
| [01101] | 13 | [10010] | 14 | [10011] | 13 |
| [01110] | 14 | [10001] | 15 | [10010] | 14 |
| [11000] | 8 | [00111] | 7 | [01000] | 8 |
| [11111] | 1 | [00000] | 0 | [00001] | 1 |
| [10000] | 16 | [01111] | 15 | [10000] | 16 |

5

Problem 2 Solution: [Pg. 3]

To show that ˜(x-1) equals -x, we make use of the property that two’s complement addition is associative, commutative, and has additive inverses. Using C notation, define y to be x-1. The complement and increment rule implies that ˜y+1 equals -y, and subtracting 1 from each side gives that ˜y equals -y-1. Substituting x-1 for y gives ˜(x-1) equals -(x-1)-1, which equals -x.

Problem 3 Solution: [Pg. 3]

Below, we highlight the pattern 1; 0; : : : ; 0 in bit vector ~x in italics.

|  |  |  |  |
| --- | --- | --- | --- |
| x |  | x |  |
| [01101 ] | 13 | [10011 ] | 13 |
| [01110 ] | 14 | [10010 ] | 14 |
| [11000 ] | 8 | [01000 ] | 8 |
| [11111 ] | 1 | [00001 ] | 1 |
| [10000 ] | 16 | [10000 ] | 16 |

Problem 4 Solution: [Pg. 3]

We have seen that x and -x have identical bit-level representations from the least significant bit up to the first bit having value 1, and beyond this they are complementary. Thus, the function can be written as:

/\*

* Generate mask indicating rightmost 1 in x.
* For example 0xFF00 -> 0x0100, and 0x6600 --> 0x200.
* If x == 0, then return 0.

\*/

int rightmost\_one(unsigned x) {

/\*

* Rightmost portions of x and -x
* are identical up to first 1

\*/

return (x & -x);

}

Problem 5 Solution: [Pg. 4]

1. Since 2147483647 is the decimal representation of TMax 32, the compiler would represent it as a value of type int. Adding 1 would cause an overflow to TMin32, and hence the expression yields type int and value 2,147,483,648. Casting this to int would have no further effect.
2. Since 0x7FFFFFFF is the hexadecimal representation of TMax 32, we would get the same result as in A: an int with value 2,147,483,648. Casting this to int would have no further effect.
3. Since 2,147,483,649 is larger than TMax 32, the compiler would find an alternate data type, select-ing data type long long as a result. We would therefore get data type long long and value +2,147,483,648, having hexadecimal representation 0x0000000080000000. Casting it to int would remove the 32 leading 0 bits, yielding TMin32.

6

1. 0x80000001 is the hexadecimal representation of 2,147,483,649. As with C, the compiler would find an alternate data type, selecting data type unsigned as a result. We would therefore get data type unsigned and value +2,147,483,648, having hexadecimal represntation 0x80000000. When we cast this to int, we have value TMin32.
2. As we saw in C, the expression within the parentheses generates data type long long and value +2,147,483,648, having hexadecimal representation 0x0000000080000000. Negating this value yields 2,147,483,648, having hexadecimal representation 0xFFFFFFFF80000000. Casting this to int yields removes the 32 leading 1 bits, yielding TMin32.
3. As we saw in D, the expression within the parentheses generates data type unsigned and value +2147483648 (hexadecimal representation 0x80000000). Negating this value yields 2,147,483,648, also having hexadecimal representation 0x80000000. Casting this to int yields TMin32.

Thus, we see that both expressions A and B are alternative ways to write int constant TMin32, and that all of them yield the correct value when cast to data type int.

Problem 6 Solution: [Pg. 4]

This problem requires elementary reasoning about bit-level representations.

The correctness of this technique follows directly from the complement-and-increment rule. Performing the bit-wise complement of ~x yields [˜xw 1; ˜xw 2; : : : ˜ xk+1; 0; 1; : : : ; 1]. Incrementing according to our rule incr then converts the low-order k + 1 bits to 1; 0; : : : ; 0.