CS:APP Web Aside DATA:TNEG: Bit-Level Representation of Two's Complement Negation*

Randal E. Bryant
David R. O'Hallaron

July 9, 2009

Notice

The material in this document is supplementary material to the book Computer Systems, A Programmer's Persepctive, Second Edition*, by Randal E. Bryant and David R. O'Hallaron, published by Prentice-Hall and copyrighted 2010. The supplementary material is being made available to the public at no charge. More information about the book is available at* `www.csapp.cs.cmu.edu`*.*

1 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.

2 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 x_i, we have $\tilde{}\, x_i = 1 - x_i$. Let $\vec{x}$ be a bit vector of length w and $x \doteq B2T_w(\vec{x})$ be the two's-complement number it represents. By Equation CS:APP-**??**, the complemented bit vector $\tilde{}\,\vec{x}$ has numeric value

$$B2T_w(\tilde{}\,\vec{x}) \quad = \quad -(1 - x_{w-1})2^{w-1} + \sum_{i=0}^{w-2}(1 - x_i)2^i$$

$\vec{x}$		$\sim\vec{x}$		$incr(\sim\vec{x})$	
$[0101]$	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.

$$\begin{aligned} &= \left[-2^{w-1} + \sum_{i=0}^{w-2} 2^i\right] - \left[-x_{w-1}2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i\right] \\ &= [-2^{w-1} + 2^{w-1} - 1] - B2T_w(\vec{x}) \\ &= -1 - x \end{aligned}$$

The key simplification in the above derivation is that $\sum_{i=0}^{w-2} 2^i = 2^{w-1} - 1$. It follows that by incrementing $\sim\vec{x}$ we obtain $-x$.

To increment a number x represented at the bit-level as $\vec{x} \doteq [x_{w-1}, x_{w-2}, \ldots, x_0]$, define the operation $incr$ as follows. Let k be the position of the rightmost zero, such that $\vec{x}$ is of the form $[x_{w-1}, x_{w-2}, \ldots, x_{k+1}, 0, 1, \ldots, 1]$. We then define $incr(\vec{x})$ to be $[x_{w-1}, x_{w-2}, \ldots, x_{k+1}, 1, 0, \ldots, 0]$. For the special case where the bit-level representation of x is $[1, 1, \ldots, 1]$, define $incr(\vec{x})$ to be $[0, \ldots, 0]$. As illustrations, Figure 1 shows how complementing and incrementing affect the numeric values of several four-bit vectors.

To show that $incr(\vec{x})$ yields the bit-level representation of $x +^{\mathrm{t}}_w 1$, consider the following cases:

1. When $\vec{x} = [1, 1, \ldots, 1]$, we have $x = -1$. The incremented value $incr(\vec{x}) \doteq [0, \ldots, 0]$ has numeric value 0.

2. When $k = w - 1$, i.e., $\vec{x} = [0, 1, \ldots, 1]$, we have $x = TMax_w$. The incremented value $incr(\vec{x}) = [1, 0, \ldots, 0]$ has numeric value $TMin_w$. From Equation CS:APP-**??**, we can see that $TMax_w +^{\mathrm{t}}_w 1$ is one of the positive overflow cases, yielding $TMin_w$.

3. When $k < w - 1$, i.e., $x \neq TMax_w$ and $x \neq -1$, we can see that the low-order $k + 1$ bits of $incr(\vec{x})$ has numeric value 2^k, while the low-order $k + 1$ bits of $\vec{x}$ has numeric value $\sum_{i=0}^{k-1} 2^i = 2^k - 1$. The high-order $w - k + 1$ bits have matching numeric values. Thus, $incr(\vec{x})$ has numeric value $x + 1$. In addition, for $x \neq TMax_w$, adding 1 to x will not cause an overflow, and hence $x +^{\mathrm{t}}_w 1$ 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.

$\vec{x}$	$\tilde{}\vec{x}$	$incr(\tilde{}\vec{x})$
$[01101]$		
$[01110]$		
$[11000]$		
$[11111]$		
$[10000]$		

Practice Problem 2:

Show that first decrementing and then complementing is equivalent to complementing and then incrementing. 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 representation. Let k be the position of the rightmost 1, so that $\vec{x}$, the bit-level representation of x has the form $[x_{w-1}, x_{w-2}, \ldots, x_{k+1}, 1, 0, \ldots 0]$. (This is possible as long as $x \neq 0$.) The negation is then written in binary form as $[\tilde{}x_{w-1}, \tilde{}x_{w-2}, \ldots \tilde{} x_{k+1}, 1, 0, \ldots, 0]$. That is, we complement each bit to the left of bit position k. Showing that the resulting value indeed has value $-_{32}^{t} x$ is left as an exercise (Problem 6.)

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

x		$-x$	
$[1\mathit{100}]$	-4	$[0\mathit{100}]$	4
$[\mathit{1000}]$	-8	$[\mathit{1000}]$	-8
$[010\mathit{1}]$	5	$[101\mathit{1}]$	-5
$[011\mathit{1}]$	7	$[100\mathit{1}]$	-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 k.

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);
```

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 $TMin_{32}$ 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 $TMin_{32}$ 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 $\vec{x}$, by finding bit position k such that $\vec{x}$ has the form $x_{w-1}, x_{w-2}, \ldots, x_{k+1}, 1, 0, \ldots 0$ The negation is then written in binary form as $[\sim x_{w-1}, \sim x_{w-2}, \ldots \sim x_{k+1}, 1, 0, \ldots, 0]$.

Show that the resulting value indeed has value $-_{32}^{\mathrm{t}} x$.

Solutions to Problems

Problem 1 Solution: [Pg. 2]

$\vec{x}$		$\sim\vec{x}$		$incr(\sim\vec{x})$	
$[01101]$	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

Problem 2 Solution: [Pg. 3]

This problem provides a chance to rework the proof that complementing and incrementing performs negation.

We make use of the property that two's complement addition is associative, commutative, and has additive inverses. Using C notation, if we define `y` to be `x-1`, then we have `~y+1` equal to `-y`, and hence `~y` equals `-y+1`. Substituting gives the expression `-(x-1)+1`, which equals `-x`.

Problem 3 Solution: [Pg. 3]

Below, we highlight the pattern $1, 0, \ldots, 0$ in bit vector $\vec{x}$ in italics.

x		$-x$	
$[0110\mathit{1}]$	13	$[1001\mathit{1}]$	-13
$[011\mathit{10}]$	14	$[100\mathit{10}]$	-14
$[1\mathit{1000}]$	-8	$[0\mathit{1000}]$	8
$[1111\mathit{1}]$	-1	$[0000\mathit{1}]$	1
$[\mathit{10000}]$	-16	$[\mathit{10000}]$	-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]

A. 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 $TMin_{32}$, and hence the expression yields type `int` and value $-2{,}147{,}483{,}648$. Casting this to `int` would have no further effect.

B. 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.

C. Since 2,147,483,649 is larger than $TMax_{32}$, the compiler would find an alternate data type, selecting 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 $TMin_{32}$.

D. `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 $TMin_{32}$.

E. 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 $TMin_{32}$.

F. 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 $TMin_{32}$.

Thus, we see that both expressions A and B are alternative ways to write `int` constant $TMin_{32}$, 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 $\vec{x}$ yields $[\tilde{}\, x_{w-1}, \tilde{}\, x_{w-2}, \ldots \tilde{}\, x_{k+1}, 0, 1, \ldots, 1]$. Incrementing according to our rule $incr$ then converts the low-order $k+1$ bits to $1, 0, \ldots, 0$.