Numbers, Representations, and Ranges

Ivor Page[1]

2.1 Unsigned Numbers

A k digit unsigned system in radix r has range 0 to $r^k - 1$.

For unsigned binary, the range is 0 to $2^k - 1$. The result of every arithmetic operation on these numbers is within this range. So, for example, if we perform the sum: $22 - 37$ in a 6 bit unsigned binary system, the result is as follows:

$$\begin{array}{rr} & 010110 \\ - & 100101 \\ \hline & 110001 \end{array}$$

[1]University of Texas at Dallas

The result has value 49. It should be -15, but the representation does not support negative values. As we shall see, the bit pattern of the result is exactly what we would get if the two values were in 2's complement representation.

Even though the answer above is, in a sense, meaningless within the representation (unsigned binary), if we add 41 to it, the result is correct:

$$\begin{array}{rr} & 110001 \\ - & 101001 \\ \hline & 011010 \end{array}$$

For any sequence of add and subtract operations in a k bit unsigned system in which the result is positive, the result will be correct in the k least significant digits of the result.

When viewed as bit-patterns, the results obtained from arithmetic operations on unsigned values are identical to those obtained when those operations are performed on the same values in the 2's complement system. Indeed the operation of the ALU is almost identical in both cases. The difference comes in how the overflow flag operates:

Unsigned operations do not set the overflow flag.

The programmer must understand this subtle difference and use these representations appropriately.

To put it more bluntly, if we have a program that performs only integer operations, there is no difference in the bit patterns of the results when we change the declarations from `int` (2's complement signed values) to `unsigned int` or vice-versa. Indeed, we can mix these types freely without changing the appearance of the results as binary bit patterns.

The difference comes in how the results are interpreted. If, for example, we print the results using the `cout` stream, the C++ language's run-time-system will take into account the types of the variables.

2.2 Signed Radix-Complement Systems

A Historical Example.

Early electro-mechanical calculators used decimal arithmetic. Punched cards were the main input devices and printers the output devices. Values were represented by the positions of gears. In mathematical terms these systems could represent decimal values of a certain number of digits, depending on the size of the counters (numbers of gear wheels) employed. Negative numbers were represented using the 10's complement system. To negate the 8 digit 10's complement value, 00,085,456, subtract it from the 9 digit value, 100,000,000, giving 99,914,544. By convention, any representation with a 9 in the left-most, or 10^7 position is a negative value.

If we add this representation of -85,456 to the representation of the positive value 115,145, we get 100,029,689. The most significant digit is beyond the range of the 8 digit number system. The answer that remains is 29,689, which is the correct answer. If, on the other hand, we add 71,103, the answer is 99,985,647. To see the magnitude of this negative value, we negate it by subtracting it from 100,000,000, giving 14,353. The value obtained represents -14,353. These are the correct answers.

There is an alternative way to negate a 10's complement number. Subtract each digit from 9 and write down the answer, digit at a time, then add 1 to the result.

What is going on here?

Any negative value with magnitude V is represented by $10^8 - V$. When we add this negative value to the positive value W, we get $10^8 - V + W$. If $W > V$ the answer is the positive quantity $W - V$ plus 10^8, which is discarded since it is beyond the range of the 8 digit system. If $W < V$ the answer is the negative value obtained by subtracting the positive value $V - W$ from 10^8. Note here that V and W are magnitudes.

The system described above is known as a *radix-complement system*. A 10's complement system with N digits, including the *sign digit*, has range -10^{k-1} to $+10^{k-1} - 1$.

Here are some examples of the representation:

Value	Representation in 8 digit 10's complement
-10,000,000	90,000,000
-9,999,999	90,000,001
-1	99,999,999
0	00,000,000
1	00,000,001
9,999,999	09,999,999

2.3 2's Complement Representation

Here are some examples of a 16 bit 2's complement system.

Value in Decimal	Representation in 16 bit 2's complement
-32768	1000,0000,0000,0000
-32767	1000,0000,0000,0001
-1	1111,1111,1111,1111
0	0000,0000,0000,0000
1	0000,0000,0000,0001
32766	0111,1111,1111,1110
32767	0111,1111,1111,1111

The range, for an N digit 2's complement system is -2^{k-1} to $+2^{k-1}-1$.

In any radix-complement system the most negative value is always one larger in magnitude than the most positive value.

2.3.1 Subtraction in 2's complement systems

The 2's complement system is the most widely used representation for signed numbers. Almost all modern computers use it.

Subtraction is easily performed by pre-negating the operand to be subtracted (the subtrahend) and adding it to the minuend. Negation requires inversion of all the bits of the representation, followed by the addition of *ulp* to the result. The term *ulp* means *unit in the least significant position*, or just 1×2^0 for integers. In general, *ulp* may mean 1×2^n, where the integer $n < 0$ for fractional systems. It is always, as its name suggest, the smallest value of the representation.

At first sight, it would seem that subtraction should take twice as long as addition because of the need to add *ulp* during the negation. This is not the case. We can easily accommodate the addition of 1 by setting the carry-in (of the least significant digit) of the adder to 1.

The subtraction is described mathematically as $A - B = A + B^{compl} + ulp$, where B^{compl} represents the *1's complement* of the bit-pattern of B. The subtraction operation proceeds exactly the same, independent of the sign of B.

2.3.2 2's Complement More Formally

It is important to distinguish the *value* of a number from its *representation*. We shall normally write the value of a number using decimal notation with a preceding plus or minus sign. The representation will be dependent on the number system we are discussing. In binary systems, the representation will normally be a bit vector.

A k bit 2's complement representation A has bits $a_{k-1}, a_{k-2}, \cdots, a_1, a_0$, where a_{k-1} is the sign bit and a_0 is the least significant bit.

The negative of A, as we have seen, is obtained by inverting all its bits and adding 1 to the result.

Negative of $A = A^{compl} + ulp$.

There are two mathematical ways to obtain the value of a k digit 2's complement number A:

$$\text{Value of } A = \begin{cases} \sum_{i=0}^{k-1} a_i 2^i & : \quad a_{k-1} = 0 \\ -2^k + \sum_{i=0}^{k-1} a_i 2^i & : \quad a_{k-1} = 1 \end{cases}$$

Alternatively,

$$\text{Value of } A = -a_{k-1} 2^{k-1} + \sum_{i=0}^{k-2} a_i 2^i$$

where a_{k-1} is the sign digit and a_i is the digit in the 2^i position.

These two forms are equivalent, independent of the sign of A.

2.4 Overflow

In any number system, the result of an arithmetic operation may not necessarily have a representation in that system. We saw an example of this when the result of a subtraction of two unsigned values was negative. Here is a table to show when overflow *can* occur with signed values:

$A + B$

	Sign of A	
	+	−
Sign of B		
+	Yes	No
−	No	Yes

$A - B$

	Sign of A	
	+	−
Sign of B		
+	No	Yes
−	Yes	No

Consider the following example using a 4 bit 2's complement representation:

$$\begin{array}{rr} & 0101 \\ + & 0011 \\ \hline & 1000 \end{array}$$

Here we are adding +5 and +3, but the answer is -8. Note that the bit pattern of the answer is the same as that of the correct answer in its 4 least significant bits. The correct answer is obtained by extending the representation to five bits by concatenating a zero (in this example) to the left of the above answer. This zero would become the new sign bit. In practice we cannot simply extend the representation of a number system by one bit, but see the module on multi-precision working.

In this example, it is easy to see that overflow has occurred because we have added two positive values and obtained a negative result. It would be equally simple if we added two negative values and got a positive result, or subtracted a negative value from a positive one and got a negative result. In any example where the sign of the result is independent of the magnitudes of the values concerned, overflow is easy to detect and these are the only cases where overflow can occur.

A much simpler way has been developed for detecting overflow in the 2's complement representation. Notice what happens to the carry into and out of the sign bit during these operations. If the values of those carries differ, then overflow has occurred. A simple EX-OR gate is suffucient to detect overflow in 2's complement addition. For subtraction, overflow cannot occur while complementing each bit of the subtrahend. Overflow can occur during the addition step, but the EX-OR gate is sufficient to detect all such cases.

Theorem: *An overflow has occurred in the addition of two 2's complement values iff the carry into and out of the sign digit differ.*

Proof:

By cases.

If we are adding two positive values, the two sign digits are zero, so the carry-out of the sign is always zero. The carry-into the sign digit is 1 iff the sum of the two values exceeds the range of the representation. Clearly this condition should generate overflow. If the carry-into the sign digit is zero, then the sum of the two values is within the range of the representation and overflow is not indicated.

If we are adding two negative values, the two sign digits are 1 and therefore there is always a carry-out from the sign digit. The carry-in to the sign digit is zero iff the result is beyond the range of the representation. Overflow must then be generated. The carry-in to the sign digit is 1 when the result is within the range of the representation. □

Note also that the simple operation of negating a value can cause overflow in any radix-complement system, since the negative range is always one larger than the positive range. If we try to negate the most negative value, the answer will not fit into the representation.

2.5 1's Complement Representation

The 1's complement system has rarely been used. It was used in the CDC 6600, but is not used in any modern computer systems. It is an example of a *Diminished Radix system* due to the way that the complement is formed:

To negate a number represented in 1's complement, we simply invert each bit:

Negative of $A = A^{compl} = 2^k - 1 - A$.

As with 2's complement, the most significant bit is the *sign bit*. Its value is 1 for negative numbers and zero for positive numbers. The value of a number in this representation has 2 mathematical forms:

$$\text{Value of } A = \begin{cases} \sum_{i=0}^{k-1} a_i 2^i & : \quad a_{k-1} = 0 \\ -2^k + 1 + \sum_{i=0}^{k-1} a_i 2^i & : \quad a_{k-1} = 1 \end{cases}$$

Alternatively,

$$\text{Value of } A = -a_{k-1}[2^{k-1} + 1] + \sum_{i=0}^{k-2} a_i 2^i$$

The range of a k bit 1's complements system is symmetrical: $-2^k - 1$ to $+2^k - 1$.

Here are some examples of 16 bit 1's complement numbers.

Value in Decimal	Representation in 16 bit 2's complement
-32767	1000,0000,0000,0000
-1	1111,1111,1111,1110
0	1111,1111,1111,1111
0	0000,0000,0000,0000
1	0000,0000,0000,0001
32766	0111,1111,1111,1110
32767	0111,1111,1111,1111

There are two representations of zero, often denoted 0_- and 0_+ (or -0 and $+0$). This anomaly creates a small problem for detection of zero, but also for the case where 1 is added to 0_- or 1 is subtracted from 0_+. One solution is to pass the result of every arithmetic operation through a simple logic filter that would change any occurrence of 0_- to the all-zeros pattern of 0_+.

A further complication in the 1's complement scheme arises during arithmetic. Since subtraction is performed by pre-negating the subtrahend, and adding the result to the minuend, we need only consider addition.

The sum of two 1's complement positive values proceeds as for unsigned numbers. If there is a carry into the sign digit, overflow has occurred. Consider the example below in a 6 bit 1's complement system:

	$5 + 9$
	000101
+	001001
	001110

The result is correct since the representations and the steps performed are exactly as for unsigned or 2's complement numbers. Now consider an example where both arguments are negative. We shall add the 1's complements of the values above:

	$-5 + -9$
	111010
+	110110
	110000

There is a carry-out of 1 from the most significant digit, which will be discarded since it cannot fit into the 6 bit number system. The 6 bits of the answer represent -15, not the value -14 expected.

To understand this, start with two positive values, A and B.

Next, we form $-A$ and $-B$.

$-A$ is represented by $2^k - 1 - A$, and

$-B$ is represented by $2^k - 1 - B$.

The sum of these is represented by $2^k - 1 - A + 2^k - 1 - B = 2^{k+1} - 2 - (A + B)$

The correct representation is $2^k - 1 - (A + B)$.

Since the values 2^k and 2^{k+1} are outside the range of the system, they do not show up in the answer. The only visible problem is the fact that the answer is a *ulp* smaller than it should be. The correction is simple.

Whenever there is a carry-out of 1 from the most significant position in 1's complement addition, *ulp* is added to the result.

This scheme is known as *end-around carry*. The carry-out of the most-significant digit of the adder is connected to the carry-in of the least significant digit.

As with any representational system, there is more than one way to ascribe meaning to values within the system. For example, if we think of the values in a k bit 1's complement system as representing positive numbers, their range would be $[0, 2^k - 2]$ (recall that there are 2 versions of zero, 000...0, and 111...1). The k bit 1's complement adder is then a $mod\ (2^k - 1)$ unsigned adder.

Consider the above examples again where the 1's complement bit patterns are interpreted as representing positive values. In the second example, (-5)+(-9), the sum would be interpreted as 58 + 54 and the bit pattern of the result, 110001, would represent 49, which is (58 + 54) mod 63.

A k bit 1's complement adder with end-around carry is a $mod\ (2^k - 1)$ adder of unsigned values that have range $[0, 2^k - 2]$.

Question: Since the end-around carry causes a cycle amongst the logic gates of a ripple-carry 1's complement adder, is there any potential for an add operation to continue for ever?

Answer: It is easy to show that the worst case time for addition is the same as for the 2's complement system if a ripple-carry adder is employed.

Negation of any value in the 1's complement system is faster than in 2's complement since the operation only requires inversion of all the bits of the number. Overflow cannot occur during negation.