ECE120: Introduction to Computer Engineering

Notes Set 1.2 The 2's Complement Representation

This set of notes explains the rationale for using the 2's complement representation for signed integers and derives the representation based on equivalence of the addition function to that of addition using the unsigned representation with the same number of bits.

1.2.1 Review of Bits and the Unsigned Representation

In modern digital systems, we represent all types of information using binary digits, or **bits**. Logically, a bit is either 0 or 1. Physically, a bit may be a voltage, a magnetic field, or even the electrical resistance of a tiny sliver of glass. Any type of information can be represented with an ordered set of bits, provided that *any given pattern of bits corresponds to only one value* and that *we agree in advance on which pattern of bits represents which value.*

For unsigned integers—that is, whole numbers greater or equal to zero—we chose to use the base 2 representation already familiar to us from mathematics. We call this representation the **unsigned representation**. For example, in a 4-bit unsigned representation, we write the number 0 as 0000, the number 5 as 0101, and the number 12 as 1100. Note that we always write the same number of bits for any pattern in the representation: *in a digital system, there is no "blank" bit value.*

1.2.2 Picking a Good Representation

In class, we discussed the question of what makes one representation better than another. The value of the unsigned representation, for example, is in part our existing familiarity with the base 2 analogues of arithmetic. For base 2 arithmetic, we can use nearly identical techniques to those that we learned in elementary school for adding, subtracting, multiplying, and dividing base 10 numbers.

Reasoning about the relative merits of representations from a practical engineering perspective is (probably) currently beyond your ability. Saving energy, making the implementation simple, and allowing the implementation to execute quickly probably all sound attractive, but a quantitative comparison between two representations on any of these bases requires knowledge that you will acquire in the next few years.

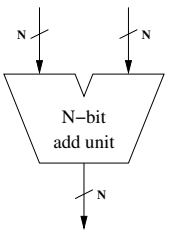
We can sidestep such questions, however, by realizing that if a digital system has hardware to perform operations such as addition on unsigned values, using the same piece of hardware to operate on other representations incurs little or no additional cost. In this set of notes, we discuss the 2's complement representation, which allows reuse of the unsigned add unit (as well as a basis for performing subtraction of either representation using an add unit!). In discussion section and in your homework, you will use the same idea to perform operations on other representations, such as changing an upper case letter in ASCII to a lower case one, or converting from an ASCII digit to an unsigned representation of the same number.

1.2.3 The Unsigned Add Unit

In order to define a representation for signed integers that allows us to reuse a piece of hardware designed for unsigned integers, we must first understand what such a piece of hardware actually does (we do not need to know how it works yet—we'll explore that question later in our class).

The unsigned representation using N bits is not closed under addition. In other words, for any value of N, we can easily find two N-bit unsigned numbers that, when added together, cannot be represented as an N-bit unsigned number. With $N = 4$, for example, we can add 12 (1100) and 6 (0110) to obtain 18. Since 18 is outside of the range $[0, 2^4 - 1]$ representable using the 4-bit unsigned representation, our representation breaks if we try to represent the sum using this representation. We call this failure an **overflow** condition: the representation cannot represent the result of the operation, in this case addition.

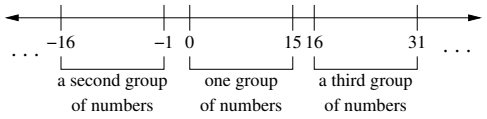
Using more bits to represent the answer is not an attractive solution, since we might then want to use more bits for the inputs, which in turn requires more bits for the outputs, and so on. We cannot build something supporting an infinite number of bits. Instead, we choose a value for N and build an add unit that adds two N-bit numbers and produces an N-bit sum (and some overflow indicators, which we discuss in the next set of notes). The diagram to the right shows how we might draw such a device, with two N-bit numbers entering at from the top, and the N-bit sum coming out from the bottom.



The function used for N-bit unsigned addition is addition modulo 2^N. In a practical sense, you can think of this function as simply keeping the last N bits of the answer; other bits are simply discarded. In the example to the right, we add 12 and 6 to obtain 18, but then discard the extra bit on the left, so the add unit produces 2 (an overflow).

```
  1100 (12)
+ 0110 (6)
 10010 (2)
```

Modular arithmetic defines a way of performing arithmetic for a finite number of possible values, usually integers. As a concrete example, let's use modulo 16, which corresponds to the addition unit for our 4-bit examples.



Starting with the full range of integers, we break the number line into contiguous groups of 16 integers, as shown to the right. The numbers 0 to 15 form one group. The numbers -16 to -1 form a second group, and the numbers from 16 to 31 form a third group. An infinite number of groups are defined in this manner.

We then define 16 **equivalence classes** consisting of the first numbers from all groups, the second numbers from all groups, and so forth. For example, the numbers $\ldots, -32, -16, 0, 16, 32, \ldots$ form one such equivalence class. Mathematically, we say that two numbers A and B are equivalent modulo 16, which we write as

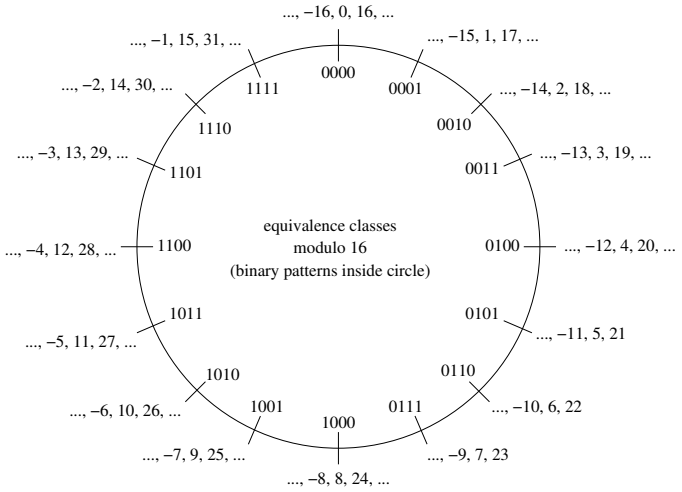
$$\begin{aligned} (A &= B) \bmod 16, \qquad \text{or sometimes as} \\ A &\equiv B \pmod{16} \end{aligned}$$

if and only if $A = B + 16k$ for some integer k.

Equivalence as defined by a particular modulus distributes over addition and multiplication. If, for example, we want to find the equivalence class for $(A + B)$ mod 16, we can find the equivalence classes for A (call it C) and B (call it D) and then calculate the equivalence class of $(C + D)$ mod 16. As a concrete example of distribution over multiplication, given $(A = 1,083,102,112 \times 7,323,127)$ mod 10, find A. For this problem, we note that the first number is equivalent to 2 mod 10, while the second number is equivalent to 7 mod 10. We then write $(A = 2 \times 7)$ mod 10, and, since $2 \times 7 = 14$, we have $(A = 4)$ mod 10.

1.2.4 Deriving 2's Complement

Given these equivalence classes, we might instead choose to draw a circle to identify the equivalence classes and to associate each class with one of the sixteen possible 4-bit patterns, as shown to the right. Using this circle representation, we can add by counting clockwise around the circle, and we can subtract by counting in a counterclockwise direction around the circle. With an unsigned representation, we choose to use the group from $[0, 15]$ (the middle group in the diagram markings to the right) as the number represented by each of the patterns. Overflow occurs with unsigned addition (or subtraction) because we can only choose one value for each binary pattern.



In fact, we can choose any single value for each pattern to create a representation, and our add unit will always produce results that are correct modulo 16. Look back at our overflow example, where we added 12 and 6 to obtain 2, and notice that $(2 = 18) \bmod 16$. Normally, only a contiguous sequence of integers makes a useful representation, but we do not have to restrict ourselves to non-negative numbers.

The 2's complement representation can then be defined by choosing a set of integers balanced around zero from the groups. In the circle diagram, for example, we might choose to represent numbers in the range $[-7, 7]$ when using 4 bits. What about the last pattern, 1000? We could choose to represent either -8 or 8. The number of arithmetic operations that overflow is the same with both choices (the choices are symmetric around 0, as are the combinations of input operands that overflow), so we gain nothing in that sense from either choice. If we choose to represent -8, however, notice that all patterns starting with a 1 bit then represent negative numbers. No such simple check arises with the opposite choice, and thus an N-bit 2's complement representation is defined to represent the range $[-2^{N-1}, 2^{N-1}-1]$, with patterns chosen as shown in the circle.

1.2.5 An Algebraic Approach

Some people prefer an algebraic approach to understanding the definition of 2's complement, so we present such an approach next. Let's start by writing $f(A, B)$ for the result of our add unit:

$$f(A, B) = (A + B) \bmod 2^N$$

We assume that we want to represent a set of integers balanced around 0 using our signed representation, and that we will use the same binary patterns as we do with an unsigned representation to represent non-negative numbers. Thus, with an N-bit representation, the patterns in the range $[0, 2^{N-1} - 1]$ are the same as those used with an unsigned representation. In this case, we are left with all patterns beginning with a 1 bit.

The question then is this: given an integer k, $2^{N-1} > k > 0$, for which we want to find a pattern to represent $-k$, and any integer $m \geq 0$ that we might want to add to $-k$, can we find another integer $p > 0$ such that

$$(-k + m = p + m) \bmod 2^N \quad ? \tag{1}$$

If we can, we can use p's representation to represent $-k$ and our unsigned addition unit $f(A, B)$ will work correctly.

To find the value p, start by subtracting m from both sides of Equation (1) to obtain:

$$(-k = p) \bmod 2^N \tag{2}$$

Note that $(2^N = 0) \bmod 2^N$, and add this equation to Equation (2) to obtain

$$(2^N - k = p) \bmod 2^N$$

Let $p = 2^N - k$. For example, if $N = 4$, $k = 3$ gives $p = 16 - 3 = 13$, which is the pattern 1101. With $N = 4$ and $k = 5$, we obtain $p = 16 - 5 = 11$, which is the pattern 1011. In general, since $2^{N-1} > k > 0$, we have $2^{N-1} < p < 2^N$. But these patterns are all unused—they all start with a 1 bit!—so the patterns that we have defined for negative numbers are disjoint from those that we used for positive numbers, and the meaning of each pattern is unambiguous. The algebraic definition of bit patterns for negative numbers also matches our circle diagram from the last section exactly, of course.

1.2.6 Negating 2's Complement Numbers

The algebraic approach makes understanding negation of an integer represented using 2's complement fairly straightforward, and gives us an easy procedure for doing so. Recall that given an integer k in an N-bit 2's complement representation, the N-bit pattern for $-k$ is given by $2^N - k$ (also true for $k = 0$ if we keep only the low N bits of the result). But $2^N = (2^N - 1) + 1$. Note that $2^N - 1$ is the pattern of all 1 bits. Subtracting any value k from this value is equivalent to simply flipping the bits, changing 0s to 1s and 1s to 0s. (This operation is called a **1's complement**, by the way.) We then add 1 to the result to find the pattern for $-k$.

Negation can overflow, of course. Try finding the negative pattern for -8 in 4-bit 2's complement.

Finally, be aware that people often overload the term 2's complement and use it to refer to the operation of negation in a 2's complement representation. In our class, we try avoid this confusion: 2's complement is a representation for signed integers, and negation is an operation that one can apply to a signed integer (whether the representation used for the integer is 2's complement or some other representation for signed integers).