CMSC 311 Computer Organization

Jolly Numbers Notes
or Representing Numbers in the Computer

Conventions

An essential skill in computer science is the ability to understand the different interpretations of 8-bit, 16-bit, 32-bit, and 64-bit long strings of binary digits (0's and 1's), since all computing control decisions and data processing operations involve such constructs.

The ability to convert between binary and decimal representations of integers and reals, without digital or mechanical assistance, is a required skill. In fact, memorizing as many powers of 2 as you can never hurts. Neither does learning the binary and decimal values of 1K, 1M, 1G, and 1T.

Unless the problem requires a specific base, please feel free to work in powers of 2, 8, 10, or 16, for MY convenience. Whatever makes you more comfortable. Why? Because no calculators will be permitted on exams. If it excites you to use one when you work problems , or to check your homework, be my guest. However, you will be required to have mastered these conversions your self and to demonstrate such ability. So, showing your work will help you receive credit for knowing what to do, even if the numbers don't quite dance correctly below your writing implement.

Unless otherwise stated, the rightmost bit is also the the least significant bit, corresponding to the value in the 2^0 position in the base-2 interpretation of the number. And, since I can't count anyway, for long strings of 0's and 1's, your grouping of digits into "fours" is fine, as is using octal (base 8) or hex (base 16). Again, this is allowed for your personal work as long as you get the requested format for your answer.

For example, the 32-bit binary string

```
(11110000111101011010111100001111)
```

is much easier for me to read in any of the following forms:

- In base 2, but grouped into 3-bit chunks and converted, becomes base 8, or octal

```
(11 110 000  111 101 011 010 111 100 001 111)  (base 2)

(36 075 327 617)          (base 8)
```

- In base 2, but grouped into 4-bit chunks, becomes base 16 or hexadecimal.

```
(1111 0000 1111 0101 1010 1111 0000 1111)       (base 2)

(F0F5 AF0F) (base 16 or H)
```

Add this to the rules and regs for the rest of the semester, and keep in mind in doing labs and in taking exams. Oh, by the way, absolutely no calculators for exams. **No exceptions**.

Notation

Given $k+1$ bits, with bits numbered $b_k b_{k-1} b_{k-2} \ldots b_2 b_1 b_0$, we assume that bit b_0 corresponding to the least significant bit. Then the sequence of bits on the left is the binary number equivalent of the decimal number represented by the summation on the right. Since each bit can only take values from the alphabet {0,1}, a string of $k+1$ bits so numbered can represent up to 2^{k+1} unique patterns. The type representation adopted for this string of bits determines how the string will be interpreted, as described below.

Unsigned Binary

The simplest representation assumes that all numbers are positive integers, with bit b_0 giving the coefficient of 2^0, b_1 giving the coefficient of 2^1, and so on, up to b_k as the coefficient of 2^k.

More formally, we have:

$$(b_k b_{k-1} b_{k-2} \dots b_2 b_1 b_0)_2 = (\sum_{i=0}^{k} b_i 2^i)_{10}$$

For example, the binary version of 2087 is shown below:

$$(1000\ \ 0010\ \ 0101)_2 = (1 \times 2^{11} + 1 \times 2^5 + 1 \times 2^2 + 1 \times 2^0)_{10}$$

Rule of Thumb: An n-bit string of all ones has the value of $2^n - 1$.

Sign(ed) Magnitude

In sign magnitude (or signed magnitude), the uppermost or most significant bit (msb) is referred to as the **sign bit**. The remaining $(k-1)$ bits are the size or magnitude of the number, with this bit string interpreted as a $(k-1)$ bit unsigned binary number. If the sign bit is zero, and all the other bits are zero, the the number is zero. If the sign bit is zero, and at least one other bit is non-zero, than the number is positive. If the sign bit is one, and at least one other bit is one, then the number is negative. The bit pattern with msb=1 and all other bits zero, is NOT negative zero; some machines adopt a “large negative” value for it, other machines make it illegal. We will treat it as an illegal bit pattern.

Using the notation from the previous section, we have:

$$(b_k b_{k-1} b_{k-2} \dots b_2 b_1 b_0)_2 = ((-1)^{b_k} \sum_{i=0}^{k-1} b_i 2^i)_{10}$$

The number 2087_{10} from the previous section requires 13 bits now: twelve to capture the size or **magnitude** of the number, and one, the msb, set to 0, to signify that the number is positive.

$$(0\ \ 1000\ \ 0010\ \ 0101)_2 = (-1)^0\ \ (1 \times 2^{11} + 1 \times 2^5 + 1 \times 2^2 + 1 \times 2^0)_{10}$$

The negative of 2087_{10}, given by -2087_{10} matches the previous value in every bit but the msb, which is now 1 because we are representing a negative number.

$$(1\ \ 1000\ \ 0010\ \ 0101)_2 = (-1)^1 (1 \times 2^{11} + 1 \times 2^5 + 1 \times 2^2 + 1 \times 2^0)_{10}$$

While this is a nice, logical way of implementing signed integers, if you attempt to add the representations for positive and negative 2087 in base 2, you don’t get “0”, which is the answer in base 10. So, when we want to do arithmetic, we need to move to yet another way of representing integers.

Sign(ed) 2’s Complement

Signed 2’s complement (or sign 2’s complement) (s2c) is a modification of the sign-magnitude form in which addition and subtraction work the way that you expect them to. The price we pay is that we can’t read a negative number directly. The high order bit is still the sign bit, and a “1” still indicates a negative number. Furthermore, zero is represented by all zeros and, for this course, a “1” followed by all zeros is still an illegal value for us.

The relationship between a number and its s2c equivalent is formalized in the following, and explained below.

$$(b_k b_{k-1} b_{k-2} \dots b_2 b_1 b_0)_2 = ((-1)^{b_k} (b_k + \sum_{i=0}^{k-1} (b_k \oplus b_i) 2^i)_{10}$$

The s2c form of a positive integer is identical to the sign-magnitude form of the positive integer. So, our "example" number 2087 is still given by 13 bits: twelve to capture the magnitude of the number, and one, the msb, set to 0, to signify that the number is positive.

$$(0\ \ 1000\ \ 0010\ \ 0101)_2 = (-1)^0\ \ (1 \times 2^{11} + 1 \times 2^5 + 1 \times 2^2 + 1 \times 2^0)_{10}$$

In order to encode or determine the s2c form of a negative integer, we first write down the s2c form of its positive form. Then, we bit-wise complement all bits of that number (the zeros become ones, the ones become zeros). This is called forming the one's complement. Once we've done that, we form the 2's complement by adding 1 to the bit string that we have. The result is the s2c form.

So, for -2087, we start with

$$+2087_{10} = (0\ \ 1000\ \ 0010\ \ 0101)_2$$

We complement all the bits, yielding the intermediate expression

$$(1\ \ 0111\ \ 1101\ \ 1010)$$

Then, we add 1, to get

$$-2087_{10} = (1\ \ 0111\ \ 1101\ \ 1011)$$

To "decode" a number in s2c form, first check the sign bit. If the sign bit is zero, the conversion is identical to that shown for sign-magnitude positive numbers above. If the sign bit is one, then we know the number is negative. Now that we know the sign, we must decode to get the magnitude. This is done by complementing all the bits and adding one. The magnitude of the resulting number will be that of the negative number you started with.

For example, consider the following 13-bit number in s2c form:

$$1111111111110$$

Because the msb is "1", we know that the number is negative. It remains to determine the magnitude of the number. First, we take the complement of all the bits, referred to as computing the 1's complement.

$$0000000000001$$

Next, we add "1" to the above value to get:

$$0000000000010$$

Since the above value is positive 2 in binary, the value we started with, all ones except for the least significant bit, must be the s2c form of -2.

Note: Suppose you are asked to compute a s2c equivalent of either positive or negative 2087 above, but the representation must be longer than 13 bits, say 16. All you have to do is extend the sign bit! If the number is positive, fill to the new msb with zeros. If the number is negative, fill to the new msb with ones. This **sign extension** works for any s2c value.

Fixed Point?–Integral in Our Book

Since the discussion above considers only whole numbers, why is this the fixed point worksheet? Well, technically the *binary point* is to the right of the least significant bit (digit) in the representations shown above. Furthermore, you can express a subset of the rational numbers by moving the binary point to the left in any of the above representations. Note that in fixed point notations, the position of the binary point is assumed–that is, no space needs to be devoted to it because the number of places or digits to the right of the binary point is predefined–literally, fixed.

For example, a 32-bit fixed point number might have 8 bits representing the binary fractional part of the number, and the remaining 24 bits will be the integer part. The only difference from the previous expressions is that, like in base 10, the first position to the right of the binary point corresponds to 2^{-1}; the next is 2^{-2} and so on, down to 2^{-8} for 8 bits. We don't see fixed point like this very often anymore, because with the exception of machines with fixed accuracy (such as cash registers, where the the smallest difference between values is 1 cent.

regardless, as long as you know where the binary point is, and it never moves, technically you are dealing with fixed point notation.

Conversion: Signed Magnitude to 2's Complement

The book is very helpful to a mathematician, and explains how to do it from a software-like function standpoint. Mechanically, if the number is positive, you are done. If the number is negative, then you retain the sign bit, complement all the other bits and add a "1" in the least significant bit position.