# Representation and Operations on Numbers

**Date of Submission: 10th February 11:59:59pm**

The purpose of this assignment is twofold:

1. Understanding number representations and operations on these representations. As a computer scientist, you should know this very well.

2. Actually implementing number systems based on lists so that you get to do a fair bit of list programming.

**Note that the essence of this assignment is to manipulate numbers by doing operations on bit strings. You can use the numeric operations of Scheme only if you have to deal with decimal representations as in problems 1, 2 and 9, and some special cases that we discussed in the class**.

**Notational conventions:** The representation of numbers as bit vectors will be denoted by letters at the beginning of the alphabet with arrows on top: $\vec{a}$, $\vec{b}$, .... The $i$th element of this vector will be represented as $a_i$; the least significant bit is $a_0$. Decimal values will be denoted using letters towards the end of the alphabet $x$, $y$ etc. $n$ and $m$ will be used for denoting sizes of vectors. As examples:

$$x = 7, y = -5, z = 0$$

$$\vec{a} = 0111, \vec{b} = 1011, \vec{c} = 0000$$

As we shall see later, $\vec{b}$ is representation of 11 in 4-bit unsigned integer form and also the representation of -5 in 4-bit signed integer form. Often we shall say "a $n$-bit value" to mean a value whose range is limited by a $n$-bit representation (signed or unsigned). For the purposes of programming, a $n$-bit vector will be represented by a list of length $n$ consisting of 1's and 0's. For instance, the vector $\vec{a}$ is represented as `(0 1 1 1)`. Most of our examples will be based on a 4-bit or 8-bit representations.

## 1 Unsigned Integers

In general, the unsigned value of the vector $\vec{x}$ is given by:

$$b2u_n(\vec{x}) = \sum_{i=0}^{n-1} x_i * 2^i$$

As an example, the vector 10110100 in a 8-bit representation represents $2^7 + 2^5 + 2^4 + 2^2 = 180$

**Problem 1:** Write functions `(b2u n a)` (standing for binary to unsigned) and `(u2b n x)` to convert a `n`-bit unsigned number to its value and an unsigned value to its binary representation. The second function should check whether the input is in the right range.

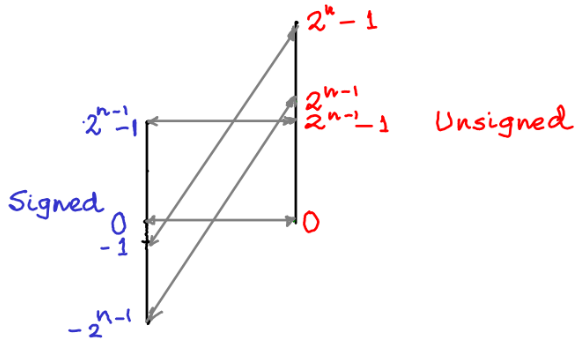
## 2 Signed Integer

In many situations we have to deal with negative integers. The most common signed integer representation is *2's complement* and in this assignment, we shall limit our attention to this form of signed integer representation only. In this representation, the leftmost bit represents the sign and, for 8-bit vector, has a weight of $-2^{8-1}$. As an example, 10110100 represents $-2^7 + 2^5 + 2^4 + 2^2 =$. In general:

$$b2s_n(\vec{x}) = -x_{n-1} * 2^{n-1} + \sum_{i=0}^{n-2} x_i * 2^i$$

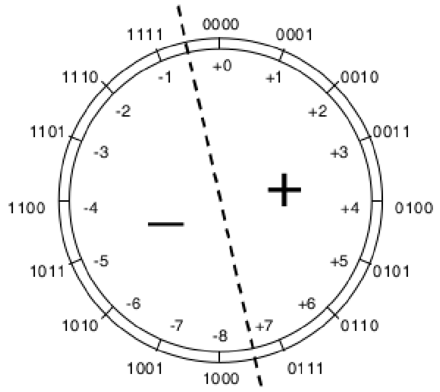
**Problem 2:** Write functions `b2s n a` and `s2b n x` to convert a `n`-bit vector to its signed value and its converse. The second function should check whether the input is in the right range.

Find out the highest and lowest representable numbers, and the representations of 1, and -1. What happens when we cast a `unsigned` to a signed integer `int` and vice-versa? The mapping from unsigned to signed integers can be visualized through the diagram shown below:



Given $n$ bits we can only store $2^n$ distinct numbers. In the unsigned case, these are numbers between $0$ and $2^n - 1$. What happens when an operation results in a number larger than this range. The usual practice is to retain the $n$ least significant bits of the result. This is effectively an operation modulo $2^n$.

In the case of signed integers, the space $0$ to $2^n - 1$ is used to store positive integers from up to $2^{n-1} - 1$, $0$ and negative integers down to $2^{n-1} - 1$. Here is another clock-face view of the correspondence, this time illustrating the modular nature of the representation.



How will you describe the function that maps an unsigned integer to the signed integer with the same representation? Similarly, how would you describe a function that maps a signed integer to the corresponding unsigned integer?

What happens when we go from a representation of one size to another. For instance, what happens when we go from a `short` to an `int`.

**Problem 3:** Write a function `(n2m n m a)` which will take a signed integer represented in $n$-bits and return the same number in m bits, $m > n$.

Suppose I take a signed integer value $x$ represented as a $m$-bit vector $\vec{w}$ and truncate it to $n$ bits by taking bits $w_{n-1} \dots w_0$. Interpret the resultant number as a signed integer of value $y$. What is the mathematical relation between $x$ and $y$?

## 3 Operations on Numbers

We start with unsigned addition. Clearly if we add two $n$-bit numbers, their sum can range from 0 to $2^{n+1} - 2$ and the resulting number may require $n + 1$ bits. The result has to be truncated to $n$ bits. Thus $n$ bit unsigned addition is addition modulo $2^n$.

**Problem 4:** Write functions `(u-add n a b)` and `(u-sub n a b)` to perform unsigned addition of two $n$ bit numbers, possibly resulting in a $n+1$-bit number. Truncate the result to $n$ bits before returning it. If `a` is less than `b`, `(u-sub a b)` should return `0`.

Given $n$-bit unsigned integer $x$, can you describe the function $u_{inv}$ that finds the additive inverse of $x$, i.e. an unsigned number $y$ such that $x + y \equiv_{2^n} 0$.

Signed addition is done exactly in the same way as unsigned addition, often using the same instruction. Thus `s-add` is the same as `u-add`. Also, a separate function for unsigned subtraction is not required— subtracting $b$ from $a$ is the same as adding $-b$ to $a$. I hope you can see the consequences of this. In a 4-bit representation, what are (3 + 4), (-3 + -4) and (-3 + -8)?

Unsigned multiplication can be done using a method which is exactly how you would multiply two numbers manually.

**Problem 5:** Write a function `(u-mult n x y)` which will take two $n$-bit unsigned integer `x` and `y` and compute their product in $2n$-bits. ~~The result is truncated before returning. You should only use the previously defined `(u-add n x y)`, the function `shift-l` to shift the contents of a list to the left and `n2m`.~~

Signed multiplication can be done exactly as you would do unsigned multiplication except that partial products have to be sign extended.

This shown in the figure below. Notice that while mutiplying $-4$ with $-3$, to obtain the last row (colored red) we have first multipled 1 with 1100 and shifted the resulting vector to the left three times yielding 1100000. This is then sign extended to give 11100000. We then obtain the 2's complement of this vector to give 00100000. You may want to think why.

```
          1 1 0 0   (-4)               1 1 0 0   (-4)
          0 1 0 1   (5)                1 1 0 1   (-3)
        ----------                   ----------
  1 1 1 1 1 1 0 0              1 1 1 1 1 1 0 0
  1 1 1 1 0 0 0 0              1 1 1 1 0 0 0 0
------------------             0 0 1 0 0 0 0 0
  1 1 1 0 1 1 0 0   (-20)   -------------------
                               0 0 0 0 1 1 0 0   (12)
```

**Problem 6:** Write a function (`s-mult n x y`) which will take two $n$-bit signed integer `x` and `y`, compute their product in $2n$-bits. ~~and truncate the result to $n$ bits before returning. You should only use the previously defined defined `u-add` , `shift-l` and `n2m`.~~

Unsigned division can also be done using a method which exactly resembles how you would divide two numbers manually.

**Problem 7:** Write a function (`u-div n x y`) which will take two $n$-bit unsigned integer `x` and `y` and compute their quotient in $n$-bits and the remainder in another $n$-bits. You should only use the previously defined defined (`u-sub x y`) and `shift-l`.

To do signed division, convert the numbers into positive, remembering the signs, do an unsigned division and restore the sign.

**Problem 8:** Write a function (`s-div n x y`) which will take two $n$-bit unsigned integer `x` and `y` and compute their quotient in $n$-bits and the remainder in another $n$-bits. You should only use the previously defined defined (`u-div n x y`) and single bit boolean operations. The output should put the quotient and the remainder in a list by using (`list quotient remainder`).

## 4 Floating Point Number Representation

Before we study floating point representations, we first look at fixed point representations. In such a representation, the point which denotes the beginning of the fractional part is assumed to be at a fixed location in the bit vector. Ignore the sign information for the moment. For, example, assume that in a 8-bit representation, the point is after the fourth bit. Then $1101.0101$ represents $2^3 + 2^2 + 2^0 + 2^{-2} + 2^{-4} = 13.3125$

In the questions that follow, you should always keep the numbers truncated to the specified length.

**Problem 9:** Consider a representation of non-negative fixed point numbers in which both the whole parts and the fractional parts are $n$-bits long. Write a function `(fix2d n a)` which will return the decimal value of a fixed-point representation, and a function `(d2fix n a)` to do the converse. The vector corresponding to the fixed point number is represented as a list of $2n$ bits.

In contrast fixed point numbers, the positon of the point in a floating point number is not fixed, and is part of the representation itself. We study a particular form of number representation called the IEEE 754 representation. In the IEEE standard, a floating point number consists of the following components:

1. The most significant bit $s$ is the sign bit which indicates whether the number is positive $s = 0$ or negative $s = 1$. 0 is handled as a special case.

2. There is a $k$-bit exponent field $\vec{e} = e_{k-1} \ldots e_0$ which weighs the number by a power of 2. Call the number represented by this as $E$. For single-precision IEEE standard, $k$ is 8. For our example, it is 4.

3. Then there is an $l$-bit significand field which $\vec{f} = f_{l-1} \ldots f_0$. This encodes a number between 1 and $2 - \epsilon$ or between $0$ and $1 - \epsilon$. Call the value represented by this field as $M$. For single-precision IEEE standard, $l$ is 23. For our example, it is 3.

The floating point number $V$ represented is $(-1)^s \times M \times 2^E$.

1. *$\vec{e}$ is not $0000$ or $1111$:* Then we get what are called *normalized numbers*. The way the bit-vector should be interpreted is as follows: Interpret $\vec{f}$ as a fractional number with a point before $f_{l-1}$ and call the resulting value $F$. Let the unsigned value of $\vec{e}$ be $E$. Further let $bias$ be $2^{4-1} - 1$ (in general $2^{k-1} - 1$. Then the value being represented is $(-1)^s \times (1 + F) \times 2^{E-bias}$.

2. *$\vec{e}$ is $0000$ and $\vec{f}$ is not $000$:* In this case the value being represented is $(-1)^s \times F \times 2^{1-bias}$.

3. *$\vec{e}$ is $0000$ and $\vec{f}$ is $000$:* Then, depending on the sign bit $s$, we get +0.0 or -0.0.

4. *$\vec{e}$ is $1111$ and $\vec{f}$ is $000$:* Once again, depending on the sign bit, we get $+\infty$ or $-\infty$.

5. *$\vec{e}$ is $1111$ and $\vec{f}$ is not $000$:* This is interpreted as "Not a Number" (NaN).

Here is a table which illustrates the rules above when $l = 3$ and $k = 4$. We assume that the sign bit is $0$ meaning the number is positive.

| $\vec{e}$ | $\vec{f}$ | $E$ | $1 - bias$ | $F$ | $0 + F$ | $Value$ |
|---|---|---|---|---|---|---|
| 0000 | 000 | 0 | -6 | 0 | 0 | 0 |
| 0000 | 001 | 0 | -6 | 1/8 | 1/8 | 1/512 |
| 0000 | 010 | 0 | -6 | 2/8 | 2/8 | 2/512 |
| 0000 | 011 | 0 | -6 | 3/8 | 3/8 | 3/512 |
| ... | ... | ... | ... | ... | | |
| 0000 | 110 | 0 | -6 | 6/8 | 6/8 | 6/512 |
| 0000 | 111 | 0 | -6 | 7/8 | 7/8 | 7/512 |
| $\vec{e}$ | $\vec{f}$ | $E$ | $E - bias$ | $F$ | $1 + F$ | $Value$ |
| 0001 | 000 | 1 | -6 | 0 | 8/8 | 8/512 |
| 0001 | 001 | 1 | -6 | 1/8 | 9/8 | 9/512 |
| ... | ... | ... | ... | ... | | |
| 0110 | 110 | 6 | -1 | 6/8 | 14/8 | 14/16 |
| 0111 | 000 | 7 | 0 | 0 | 8/8 | 1 |
| 0111 | 001 | 7 | 0 | 1/8 | 9/8 | 9/8 |
| ... | ... | ... | ... | ... | ... | ... |
| 1110 | 110 | 14 | 7 | 6/8 | 14/8 | 224 |
| 1110 | 111 | 14 | 7 | 7/8 | 15/8 | 240 |
| 1111 | 000 | ... | ... | ... | ... | $+\infty$ |
| 1111 | 001 | ... | ... | ... | ... | $NaN$ |
| 1111 | 010 | ... | ... | ... | ... | $NaN$ |
| ... | ... | ... | ... | ... | ... | ... |

**Problem 10:** Consider a $1 + k + l$-bit representation of floating point numbers. Write a function `(float2d k l a)` which will return the decimal value of a floating-point representation, and a function `(d2float k l a)` to do the converse. In the case of `d2float`, truncate if necessary. As an example, if the sign bit of a floating point number is 1, the 4-bit exponent is $0110$ and the 3-bit significand is $110$, the number should be represented as `(1 (0 1 1 0) (1 1 0))`

.

Here are some hints regarding how to approach the problem. Assume $k = 4$ and $l = 3$ so that $bias$ is 7. Define a canonical representation of a floating point number as $1.x \times 2^y$, where both $x$ and $y$ represented as binary (lists of bits in your implementation). An important issue here is the number of bits in $x$ and $y$. Since the minimum number that we can store is $0.001(binary) \times 2^{-6}$, $x$ need not contain more than 9 bits. Similarly since the largest number that we can store is $1.111(binary) \times 2^7$, $y$ need not contain more than 10 bits. While implementing `d2float`, we shall first convert the given decimal to its canonical representation, and then convert the canonical representation to the floating point representation. Similarly, while mutiplying or dividing two floating point numbers, we shall first convert them to their canonical form, perform the operation, and convert them back to floating point.

Let us discuss `d2float`. Here are some examples:

1. Consider `(d2float 4 3 84.8)`. First convert 84.8 to a a canonical representation giving $1.010100110 \times 2^6$. Notice that while .8 has an infinite expansion, the fractional part does not go beyond 9 places. This is clearly a normalized number. $\vec{f}$ is truncated to $010$ (remember that the 1. is implicit for normalized numbers) and $\vec{e}$ is $1011$ $(6 + bias)$.

2. Now consider `(d2float 4 3 1024.8)`. This case has to be treated specially. The canonical representation of this is $1.110011001 \times 2^{10}$. Since the highest number that we can store is $1.111 \times 2^7 = 540$, we round this to the representation of 540. Thus $\vec{f}$ is $111$ and $\vec{e}$ $1110$.

3. Now consider `(d2float 4 3 0.005859375)`. The canonical representation for this is $1.1 \times 2^{-8}$. This is clearly a denormalized number since the exponent is below $-6$. Writing the same number in the form $.011 \times 2^{-6}$, we clearly see that $\vec{f}$ is $011$ and $\vec{e}$ as $0000$.

4. Now consider `(d2float 4 3 0.005859375)`. The canonical representation for this is $1.1 \times 2^{-8}$. This is clearly a denormalized number since the exponent is below $-6$. Writing the same number in the form $.011 \times 2^{-6}$, we clearly see that $\vec{f}$ is $011$ and $\vec{e}$ as $0000$.

5. Finally, `(d2float 4 3 (/ 1 1024))` is below the minimum positive value that can be represented and approximates to 0 ($\vec{f}$ is $000$ and $\vec{e}$ as $0000$).

Floating-point multiplication and divsion are simpler than floating-point addition. It relies on the following identity.

$$(F_1 \times 2^{E_1}) \times (F_2 \times 2^{E_2}) = (F_1 \times F_2) \times 2^{E_1+E_2}$$

Here $F$ and $E$ are the unsigned values of $\vec{f}$ and $\vec{e}$. Postshifting may be needed, since the product $F_1 \times F_2$ of the two significands can be in the range $1 \leq F_1 \times F_2 < 4$ and therefore not in canonical form. Therefore a single-bit right shift of the significand of the product may be necessary.

Similarly division depends on the identity:

$$(F_1 \times 2^{E_1})/(F_2 \times 2^{E_2}) = (F_1/F_2) \times 2^{E_1-E_2}$$

Post-shifting may be needed, since the division $F_1/F_2$ of the two significands can be in the range $0.5 < F_1/F_2 < 2$ and therefore not in canonical form. Therefore a single-bit left shift of the value of the significand may be necessary. Also note that the division $F_1/F_2$ that we are talking of over here is not the (quot,rem) kind of integer division, but one involving decimals. For instance $0111/1001$ yields the truncated value $.110$ or $110 \times 2^{-3}$.

**Problem 11:** Write a function `(mult k l a b)` which will multiply two $(1+k+l)$-bit floating point numbers. Similarly write a function `(divide k l a b)` which will divide `a` by `b`. Truncate, if necessary, to fit the result in $(1+k+l)$-bits.