Representation and Operations on Numbers

Date of Submission: 27th February 2013 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.

Notational conventions: The representation of numbers as bit vectors will be denoted by letters at the beginning of the alphabet with arrows on top: $\vec{a}$, $\vec{b}$, ... Individual elements of the vector will be represented as a_i. n and m will be used for denoting sizes of vectors. Values will be denoted using letters towards the end of the alphabet x, y .... n and m will be used for denoting sizes of vectors. As examples:

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

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

$\vec{b}$ 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

In general, while you write functions representing n bit operations, do not pass n as a parameter; use n as a global variable defined at the beginning of the program. While this is bad programming practice, it will reduce some amount of clutter.

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 represents $2^7 + 2^5 + 2^4 + 2^2 = 180$

Problem 1: Write functions `(b2u-n a)` (standing for n-bit binary to unsigned) and `(u2b-n x)` to convert a n-bit vector to its unsigned value and its converse. The second function should check whether the input is in the right range. **Problems 1 and 2 are the only questions in which you are permitted to use the numeric operations of Scheme**.

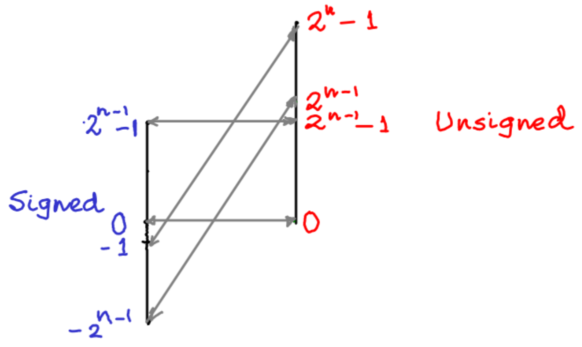
2 Signed Integer

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:

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

Problem 2: Write functions `b2s-n` and `s2b-n` 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:



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 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 $w_{m-1} \ldots w_0$. 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 a b)` and `(u-sub a b)` to perform unsigned addition of two n bit numbers, resulting in a n-bit number.

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 $x + y \equiv 0 \; mod \; 2^n$.

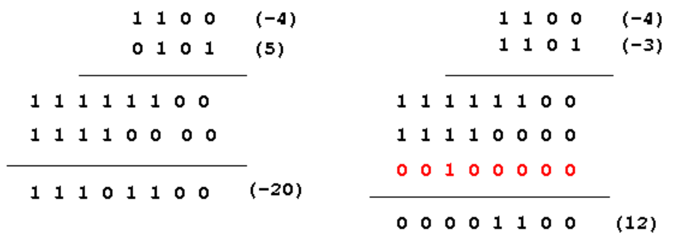
Signed addition is done exactly in the same way as unsigned addition, often using the same instruction. Thus `s-add` is the same as `u-add`. And, needless to say, a separate function for unsigned subtraction is not required. I hope you can see the consequences of this. In a 4-bit representation, what are (3 + 4), (-3 + -4) and (-3 + -9)?

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 x y)` which will take two n-bit unsigned integer `x` and `y` and compute their product in $2n$-bits. You should only use the previously defined `(u-add 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.



Problem 6: Write a function `(s-mult x y)` which will take two n-bit signed integer `x` and `y` and compute their product in $2n$-bits. You should only use the previously defined defined `(u-add x y)`, `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 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 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 x y)` and single bit boolean operations. The output format should be `(cons` 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 $2^3 + 2^2 + 2^0 + 2^{-2} + 2^{-4} = 13.3125$

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 a)` which will return the decimal value of a fixed-point representation, and a function `(d2fix a)` to do the converse In the case of `(d2fix)`, truncate if necessary.

We study a particular form of number representation called the IEEE 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 9: Consider a $1 + k + l$-bit representation of floating point numbers. Write a function `(float2d a)` which will return the decimal value of a floating-point representation, and a function `(d2float a)` to do the converse. In the case of `d2float`, truncate if necessary.

Addition and subtraction of two floating-point operands goes like this. Consider the representations of two floating point numbers $\vec{a} = 010101111$ and $\vec{b} = 010001100$. For clarity, we separate out the sign, the exponent and the significand by | and make explicit the decimal point and the normalization bit containing 1. After doing so we get $\vec{a} = 0|1010|1.111$ and $\vec{b} = 0|1000|1.100$. The sign, the exponent and the significand have been separated out by | and the decimal point and the normalization bit containing 1 has been made explicit.

First make the two exponents equal by increasing the exponent of the smaller number $\vec{b}$ with 2^2. According shift the significand of $\vec{b}$to the right by 2 to retain its value. Thus $\vec{b} = 0|1010|011$. Notice that the shift includes the normalization value of 1. Now add the two numbers to get 0|1010|10.110. Normalize to get 0|1010|1.011 dropping the |, normalization bit and the decimal point, we get 010101011.

On the other hand, subtracting b from a will result in 0|1010|0.001, which, when normalized, gives 0|1010|1.000. Dropping the |, normalization bit and the decimal point, we get 010101000.

For our programming problems, globally define the size of the exponent `k` to be 4 and the size of the significand `l` to be 3.

Problem 10: Write a function `(add a b)` which will add two $(1+k+l)$-bit floating point numbers. Similarly write a function `(sub a b)` which will subtract two $(1+k+l)$-bit floating point numbers. Truncate, if necessary, to fit the result in $(1+k+l)$-bits.

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

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

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 unnormalized. Therefore a single-bit right shift 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 product F_1/F_2 of the two significands can be in the range $1 < F_1/F_2 < 4$ and therefore unnormalized. Therefore a single-bit left shift may be necessary.

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