CS2100 Computer Organization
Tutorial 1
C and Number Systems
SUGGESTED SOLUTIONS

1. In 2's complement representation, "sign extension" is used when we want to represent an *n-bit* signed integer as an *m-bit* signed integer, where *m* > *n*. We do this by copying the MSB (most significant bit) of the *n-bit* number *m* – *n* times to the left of the *n-bit* number to create the *m*-bit number.

   For example, we want to sign-extend 0110_{2s} to an 8-bit number. Here *n = 4*, *m = 8*, and thus we copy the MSB bit 0 four (8 – 4) times, giving 00000110_{2s}.

   Similarly, if we want to sign-extend 1010_{2s} to an 8-bit number, we would get 11111010_{2s}.

   Show that IN GENERAL, sign extension is value-preserving. For example, $00000110_{2s} = 0110_{2s}$ and $11111010_{2s} = 1010_{2s}$.

Answer:

Let X be the *n*-bit signed integer and Y be the *m*-bit signed integer which is the sign-extended version of X.

If the MSB of X is zero, this is straightforward, since padding more 0's to the left adds nothing to the final value. If the MSB of X is one, then it is trickier to prove. In the original *n-bit* representation, the MSB has a weight of -2^{n-1} giving us

$$X = -2^{n-1} + b_{n-2} \cdot 2^{n-2} + b_{n-3} \cdot 2^{n-3} + \cdots + b_0.$$

Let $Z = b_{n-2} \cdot 2^{n-2} + b_{n-3} \cdot 2^{n-3} + \cdots + b_0$, then $X = -2^{n-1} + Z$.

In the new *m*-bit representation Y where *m* > *n*, the MSB of Y has a weight of -2^{m-1}, and since we copy the MSB (i.e. the leftmost bit) of X a total of *m* – *n* times, we get

$$Y = -2^{m-1} + 2^{m-2} + 2^{m-3} + \cdots + 2^{n} + 2^{n-1} + Z.$$

For $Y = X$, it suffices to show that $-2^{m-1} + 2^{m-2} + 2^{m-3} + \cdots + 2^{n} + 2^{n-1} = -2^{n-1}$.

Recall that the sum of a Geometric Progression with N terms, initial value a and ratio r is given by: $\frac{a(r^N-1)}{r-1}$. We will use this formula to calculate $2^{m-2} + 2^{m-3} + \cdots + 2^{n} + 2^{n-1}$, which has $N = (m-2) - (n-1) + 1 = m - n$; $a = 2^{n-1}$ and $r = 2$.

$$\begin{aligned}
&-2^{m-1} + (2^{m-2} + 2^{m-3} + \cdots + 2^{n} + 2^{n-1}) \\
&= -2^{m-1} + \frac{a(r^N - 1)}{r - 1} \\
&= -2^{m-1} + 2^{n-1}(2^{m-n} - 1) \\
&= -2^{m-1} + 2^{m-1} - 2^{n-1} \\
&= -2^{n-1}
\end{aligned}$$

Therefore, $Y = X$.

2. We generalize (r – 1)'s-complement (also called *radix diminished complement*) to include fraction as follows:

$$(r - 1)\text{'s complement of } N = r^n - r^{-m} - N$$

where n is the number of integer digits and m the number of fractional digits. (If there are no fractional digits, then m = 0 and the formula becomes $r^n - 1 - N$ as given in class.)

For example, the 1's complement of 011.01 is $(2^3 - 2^{-2})$ – 011.01 = (1000 – 0.01) – 011.01 = 111.11 – 011.01 = 100.10. (Since 011.01 represents the decimal value 3.25 in 1's complement, this means that -3.25 is represented as 100.10 in 1's complement.)

Perform the following binary subtractions of values represented in 1's complement representation by using addition instead. (Note: Recall that when dealing with complement representations, the two operands must have the same number of digits.)

(a) 0101.11 – 010.0101
(b) 010111.101 – 0111010.11

Is sign extension used in your working? If so, highlight it.

Check your answers by converting the operands and answers to their actual decimal values.

Answers:

(a) 0101.1100 – 0010.0101 ➔ 0101.1100 + 1101.1010 ➔ **0011.0111_{1s}**
(Check: 5.75 – 2.3125 = 3.4375)

(b) 0010111.101 – 0111010.110 ➔ 0010111.101 + 1000101.001 ➔
1011100.110_{1s} = -0100011.001_2
(Check: 23.625 – 58.75 = –35.125)

Note that sign-extension is used above.

Note that two trailing zeroes are added. (This is not sign extension.)

3. Convert the following numbers to fixed-point binary in 2's complement, with 4 bits for the integer portion and 3 bits for the fraction portion.

(a) 1.75 (b) -2.5 (c) 3.876 (d) 2.1

Using the binary representations you have derived, convert them back into decimal. Comment on the compromise between range and accuracy of the fixed-point binary system.

Answers:

(a) 1.75
$(0001.110)_{2s}$

(b) -2.5
Begin with 2.5: $(0010.100)_{2s}$. Invert and add 0.001: $(1101.100)_{2s}$

(c) 3.876
0.876 × 2 = 1.752
0.752 × 2 = 1.504
0.504 × 2 = 1.008
0.008 × 2 = 0.016 (why perform 4 steps instead of 3?)
So $0.876_{10} = 0.1110_{2s} = 0.111_{2s}$
Answer: $(0011.111)_{2s}$

(d) 2.1

$0.1 \times 2 = 0.2$

$0.2 \times 2 = 0.4$

$0.4 \times 2 = 0.8$

$0.8 \times 2 = 1.6$ (why perform 4 steps instead of 3?)

So $0.1_{10} = 0.0001_{2s} = 0.001_{2s}$

Putting it together we have: $2.1_{10} = (0010.001)_{2s}$

The first two will convert back exactly to 1.75 and -2.5, so that's ok.

For (c), the fraction part is $0.111_2 = 0.5 + 0.25 + 0.125 = 0.875$, which is just off the target of 0.876 by 0.001. Not bad.

For (d), the fraction part is $0.001_2 = 0.125$. This is off the actual value of 0.1 by 0.025, quite a lot.

Comment: Not all values can be represented exactly, and the precision depends on the number of bits in the fraction part. In this case 3 bits is too little to even represent 0.1, because the smallest fraction it can represent is 0.125.

4. [AY2010/2011 Semester 2 Term Test #1]
How would you represent the decimal value –0.078125 in the IEEE 754 single-precision representation? Express your answer in hexadecimal. Show your working.

Answer: **B D A 0 0 0 0 0**

$-0.078125 = -0.000101_2 = -1.01 \times 2^{-4}$

Exponent = $-4 + 127 = 123 = 01111011_2$

1 01111011 0100000…

1011 1101 1010 0000 …

B D A 0 0 0 0 0

5. Given the partial C program shown below, complete the two functions: **readArray()** to read data into an integer array (with at most 10 elements) and **reverseArray()** to reverse the array. For **reverseArray()**, you are to provide two versions: an iterative version and a recursive version. For the recursive version, you may write an auxiliary/driver function to call the recursive function.

```
#include <stdio.h>
#define MAX 10

int readArray(int [], int);
void printArray(int [], int);
void reverseArray(int [], int);

int main(void) {
   int array[MAX], numElements;

   numElements = readArray(array, MAX);
   reverseArray(array, numElements);
   printArray(array, numElements);

   return 0;
}

int readArray(int arr[], int limit) {

   // ...
   printf("Enter up to %d integers, terminating with a negative
integer.\n", limit);
   // ...
}

void reverseArray(int arr[], int size) {

   // ...
}

void printArray(int arr[], int size) {
   int i;

   for (i=0; i<size; i++) {
      printf("%d ", arr[i]);
   }
   printf("\n");
}
```

Answers:

```
int readArray(int arr[], int limit) {
   int i, input;

   printf("Enter up to %d integers, terminating with a negative
integer.\n", limit);
   i = 0;
   scanf("%d", &input);
   while (input >= 0) {
      arr[i] = input;
      i++;
      scanf("%d", &input);
   }
   return i;
}
```

```
// Iterative version
// Other solutions possible
void reverseArray(int arr[], int size) {
   int left=0, right=size-1, temp;

   while (left < right) {
      temp = arr[left]; arr[left] = arr[right]; arr[right] = temp;
      left++; right--;
   }
}
```

```
// Recursive version
// Auxiliary/driver function for the recursive function
// reverseArrayRec()
void reverseArrayV2(int arr[], int size) {
   reverseArrayRec(arr, 0, size-1);
}

void reverseArrayRec(int arr[], int left, int right) {
   int temp;

   if (left < right) {
      temp = arr[left]; arr[left] = arr[right]; arr[right] = temp;
      reverseArrayRec(arr, left+1, right-1);
   }
}
```

6. Trace the following program manually (do not run it on a computer) and write out its output. When you present your solution, draw diagrams to explain.

```
#include <stdio.h>

int main(void) {
   int a = 3, *b, c, *d, e, *f;

   b = &a;
   *b = 5;
   c = *b * 3;
   d = b;
   e = *b + c;
   *d = c + e;
   f = &e;
   a = *f + *b;
   *f = *d - *b;

   printf("a = %d, c = %d, e = %d\n", a, c, e);
   printf("*b = %d, *d = %d, *f = %d\n", *b, *d, *f);

   return 0;
}
```

Answers:

```
a = 55, c = 15, e = 0
*b = 55, *d = 55, *f = 0
```