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 leftmost bit of the n-bit signed $m - n$ times to the left of the n-bit number to create the m-bit number.

   So for example, we want to sign-extend 0b0101to an 8-bit number. Here $n = 4, m = 8$, and thus we copy the leftmost bit (which is 0) $m - n = 8 - 4 = 4$ times, giving us 0b<u>0000</u>0101.

   Similarly if we want to sign-extend 0b1101 to an 8-bit number, we would get 0b<u>1111</u>1101.

   Show that sign extension is value-preserving.

   Answer:

   If the leftmost bit is zero, this is straightforward, since padding more 0's to the left does not change the value.

   If the leftmost bit is one, then it has a weight of -2^{n-1}. After sign-extended to m bits, the leftmost $m - n$ bits of the m-bit number has the value $-2^{m-1} + 2^{m-2} + 2^{m-3} + \cdots + 2^n + 2^{n-1}$ which can be shown to be equal to -2^{n-1} by using sum of geometric progression.

   (Recall that sum of a geometric progression which initial value a, ratio r and T terms is $\frac{a(r^T-1)}{r-1}$. Here, $a = 2^{n-1}, r = 2$, and $T = (m-2) - (n-1) + 1 = m - n$.)

   $$-2^{m-1} + (2^{m-2} + 2^{m-3} + \cdots + 2^n + 2^{n-1})$$
   $$= -2^{m-1} + 2^{n-1}(2^{m-n} - 1) \text{ (sum of geometric progression)}$$
   $$= -2^{m-1} + 2^{m-1} - 2^{n-1}$$
   $$= -2^{n-1}$$

   Hence, sign extension is value preserving.

2. We generalize $(r-1)$'s-complement (also called radix diminished complement) of base r numbers to include fraction as follows:

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

   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 $r^n - 1 - X$ 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.

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

   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.

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

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

Answers

(a) 1.75 = **$(0001.110)_{2s}$**
Converting back: $(0001.110)_{2s}$ corresponds to 1.75. Representation is exact.

(b) -2.5

We begin with 2.5:	$(0010.100)_{2s}$
Then invert and +0.001:	**$(1101.100)_{2s}$**

Converting back: $(1101.100)_{2s}$ corresponds to -2.5. Representation is exact.

(c) 3.876

0.876 * 2 = 1.752
0.752 * 2 = 1.504
0.504 * 2 = 1.008
0.008 * 2 = 0.016

Note we do one more step and apply rounding for more accurate value.

So 0.876 = 0.111_{2s}
Hence 3.876 = **$(0011.111)_{2s}$**

Converting back: $(0011.111)_{2s}$ corresponds to 3.875. Representation is off by 0.001.

(d) 2.1

0.1 * 2 = 0.2
0.2 * 2 = 0.4
0.4 * 2 = 0.8
0.8 * 2 = 1.6

Note we do one more step and apply rounding for more accurate value.

So 0.1 = 0.0001_{2s} = 0.001_{2s}
Hence 2.1 = **$(0010.001)_{2s}$**

If we have done only 3 steps, we would have obtained 0.1 = 0.000_{2s}.

Converting back: $(0010.001)_{2s}$ corresponds to 2.125. Representation is off by 0.025.

Conclusion: 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**
Working:
$-0.078125 = -0.000101_2 = -1.01 \times 2^{-4}$
$\text{Exponent} = -4 + 127 = 123 = 01111011_2$
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
```

Please post on Canvas or QnA if you have any queries.