Computer Systems Organization
CSCI-UA.0201-001 Spring 2022

Homework 1
ANSWERS

1. (a) Write one line of C code that sets the value of a 32-bit unsigned integer variable `x` such that the pattern 1011 is repeated 8 times. That is, if you printed out the bits of `x`, you'd see `10111011101110111011101110111011`. You should assume that the compiler will not let you express constants in binary.

   Answer: Since 1011 is B hex, repeating B in a hex number will give the desired bit pattern.

   ```
   unsigned x = 0xBBBBBBBB;
   ```

   (b) Write one line of C code that writes into a 32-bit unsigned integer variable `y` a value that, if it were interpreted as a float, would be 117.625 That is, after your line of code, the following would print "117.625" (perhaps with some trailing zeros).

   ```
   printf("%f\n", *((float *) &y));
   ```

   To solve this, you should figure out the bit pattern for 117.625 in IEEE floating point.

   Answer:
   117.625 decimal = 64 + 32 + 16 + 4 + 1 + 1/2 + 1/8 = 1110101.101 x 2^0 = 1.110101101 x 2^6.

   Since the actual mantissa is 1.110101101, the stored 23-bit mantissa, starting at the leftmost bit will be 110101101 followed by 14 zeros (to fill out the 23 bits). This bit pattern, 110 1011 0100 0000 0000 0000, when expressed as a hex number is 6B4000.

   Since the actual exponent is 6, the stored 8-bit exponent is 6 + 127 = 133 = 128 + 4 + 1 = 10000101 binary = 85 hex.

   Since 117.65 is positive, the sign bit is 0. We shift the stored exponent left by 23 bits and OR in the mantissa bits, so the one line of code is:

   ```
   unsigned y = (0x85 << 23) | 0x6B4000;
   ```

2. (a) Show how the numbers -14.5 and 27.125, represented as 32-bit IEEE floating point numbers, are added together to form 12.625. That is, show what the sign, exponent, and fraction fields are for -14.5 and 27.125, then show each step of the addition that leads to 12.625. Be sure to keep everything in binary (i.e. no decimal arithmetic).

   Answer:
   Since -14.5 is -(8+4+2+(1/2)), it is -1110.1 x 2^0 = -1.1101 x 2^3 in normalized binary scientific notation. Thus, in IEEE floating point, the sign is 1, the 23-bit stored mantissa is 1101 followed by 19 zeros, and the stored exponent is 3 + 127 = 130 = 10000010 binary.

   Similarly, 27.125 = 16+8+2+1+(1/8) = 11011.001 x 2^0 = 1.1011001 x 2^4. Thus,the stored sign bit is 0, the stored exponent is 4 + 127 = 131 = 10000011 binary.

To perform the addition, the exponents must be the same. To make the exponents the same, we increase the exponent of the number with the smaller exponent, namely -14.5, shifting the mantissa to the right. Therefore, -14.5 becomes -0.11101 x 2^4.
Storing the mantissa in 32 bits (you can use <u>fewer</u> bits in your answer), where the point occurs to the left of bit 22, the mantissas would be:
27.125: 000000001.10110010000000000000000 (obviously the point isn't stored)
-14.5 : 000000000.11101000000000000000000

Since the signs are different, we subtract the mantissa of the smaller number (in magnitude) from the mantissa of the larger number. We perform the subtraction by negating the smaller mantissa (by flipping the bits and adding one) and then performing addition. Negating the above mantissa for -14.5 by flipping the bits and adding 1, we get:
111111111.00010111111111111111111 + 1 = 111111111.00011000000000000000000

Now performing the addition, we get:
000000001.10110010000000000000000
<u>111111111.00011000000000000000000</u>
000000000.11001010000000000000000

Ignoring the leading zeros before the point and the trailing zeros after the point, the mantissa of the sum is 0.1100101 (unnormalized), the exponent is 4. Normalizing, the mantissa becomes 1.100101 and the exponent becomes 3. So, the result is 1.100101 x 2^3. To convert back to decimal, it can be written as 1100.101 x 2^0 = 8 + 4 + (1/2) + (1/8) = 12.625.

(b) Convert the decimal numbers 17 and 11 to binary. Then, multiply the two numbers together in binary, showing every step of the binary multiplication. Then, convert the result back to decimal.

Answer:
17 = 16 + 1 = 10001 binary
11 = 8 + 2 + 1 = 1011 binary

Here's the multiplication:

```
    10001
     1011
    -----
    10001
   10001
       0
 10001
 --------
 10111011
```

The result is 10111011 binary = 128 + 32 + 16 + 8 + 2 + 1 = 187 decimal.

(c) Convert the decimal numbers 97 and 5 to binary. Then divide the binary version of 97 by the binary version of 5, showing every step of the binary division. The result (quotient) can be a whole binary number, with a binary remainder.

Answer:
97 = 64 + 32 + 1 = 1100001 binary. 5 = 4 + 1 = 101 binary.
Here is the division:

```
      10011
101 ) 1100001
     -101
       1000
      -101
        111
       -101
         10
```

The quotient is 10011 binary = 16 + 2 + 1 = 19 decimal.
The remainder is 10 binary = 2 decimal.

3. Given the following C code,

```
int foo()
{
  int a[5] = {2,4,6,8,10};
  int sum = 0;
  // FILL IN CODE HERE
  printf("The sum is %d\n", sum);
}
```

write a loop where it says "`// FILL IN CODE HERE`", such that the loop computes the sum of the elements of `a`, in the variable `sum`, and such that your code does not use square brackets (i.e. "`[`" or "`]`"). Feel free to define any other variables you like and to have statements that are outside the loop.

Answer:

```
void foo()
{
  int a[5] = {2,4,6,8,10};
  int sum = 0;
  int *p = a;
  for(int i = 0; i < 5; i++) {
    sum += *p;
    p++;
  }
  printf("The sum is %d\n", sum);
}
```

4. (a) Define in C a struct type `NODE`, using `typedef`, that can be used in a tree in which each node has a `value1` field of type `int`, a `value2` field of type `int`, and three children, `left`, `middle`, and `right`, where each child is a pointer to another `NODE` (or NULL).
Answer:

```
typedef struct node {
  int value1;
  int value2;
  struct node *left;
  struct node *middle;
  struct node *right;
} NODE;
```

(b) What would `sizeof(NODE)` return on your computer? Explain.

Answer: Since the two integer fields are each 4 bytes and the three pointer fields are each 8 bytes, then assuming the fields are all adjacent in memory, that's a total of 32 bytes (which is what `sizeof(NODE)` returns on my Mac).

(c) Write a function that, given a tree consisting of NODEs, rotates the children of each node in the tree. That is, for each node, the left child becomes the middle child, the middle child becomes the right child, and the right child becomes the left child. As usual for trees, it will be easiest to do this using recursion.

Answer:

```
void rotate(NODE *tree)
{
  if (tree == NULL)
    return;
  // recursive calls
  rotate(tree->left);
  rotate(tree->middle);
  rotate(tree->right);
  // now rotate this node
  NODE *temp = tree->right;
  tree->right = tree->middle;
  tree->middle = tree->left;
  tree->left = temp;
}
```

It doesn't matter if the recursive calls are before or after the rotation.

5. Assuming that the register `%rcx` contains the address of an array of 32-bit integers and the register `%rdx` contains the size of the array, write some x86-64 assembly code that returns the index of the largest element of the array. The result should be placed in the `%eax` register. For example, if the array `A` pointed to by `%rcx` contains the values 32, 15, 46, 0, 5 and 10 and `%rdx` contains 6 because there are 6 elements, then your code should place 2 in `%eax` since `A[2]` is the largest value.

Answer:

```
        movq    $0,%rdi           #i in %rdi
        movq    $0,%rax           # index of largest element in %rax
        movl    (%rcx,%rax,4),%r8d      #largest value in %r8d
TOP:                              # top of loop
        cmpq    %rdx,%rdi         # compare i to size
```

```
        jge      DONE             # if i >= size, jump out of loop.
        cmpl     %r8d,(%rcx,%rdi,4)   # compare a[i] to %r8d (32 bits)
        jle      NEXT             # if a[i] <= %rd8, continue to next element
        movq     %rdi,%rax        # otherwise, put i in %rax
        movl     (%rcx,%rdi,4),%r8d  # and put a[i] in %r8d (32 bits)
NEXT:
        incq     %rdi         # i++
        jmp      TOP          # jump to top of loop

DONE:
```

Note that the result has been placed in %rax. Since %eax is the lower half of %rax, the result will also be in %eax.

6. (a) Assume a computer supports 8-bit signed integers represented using two's complement. What is the most positive number that can be represented? What is the most negative number? Give your answers in <u>both</u> binary and decimal.

   Answer: The largest 8-bit postive signed number will be 01111111, which is $2^7 - 1 =$ **127. The largest 8-bit negative number will be 10000000, which is** $-(2^7) =$ **-128.**

   (b) What is the formula for converting a signed two's complement binary number to a decimal number (this is the formula with the summation that I wrote in class)? Give a brief intuitive description of how a negative number is represented, based on that formula.

   Answer: Here is the formula from the lecture:

The value of an N-bit
two's complement (signed) number

$b_{n-1} b_{n-2} \cdots b_1 b_0$

most negative number

is

$$\left(b_{n-1} \times -2^{n-1}\right) + \sum_{i=0}^{n-2} \left(b_i \times 2^i\right)$$

leftmost bit

An n-bit negative number, whose leftmost bit is 1, is represented as the largest possible negative number, namely -2^{n-1}**, plus the positive number represented by the rest of the bits (namely bits 0 through n-2).**