CSC 325 Lab 2

Lab 2: Data Representation and Arithmetic Operations

Table of Contents

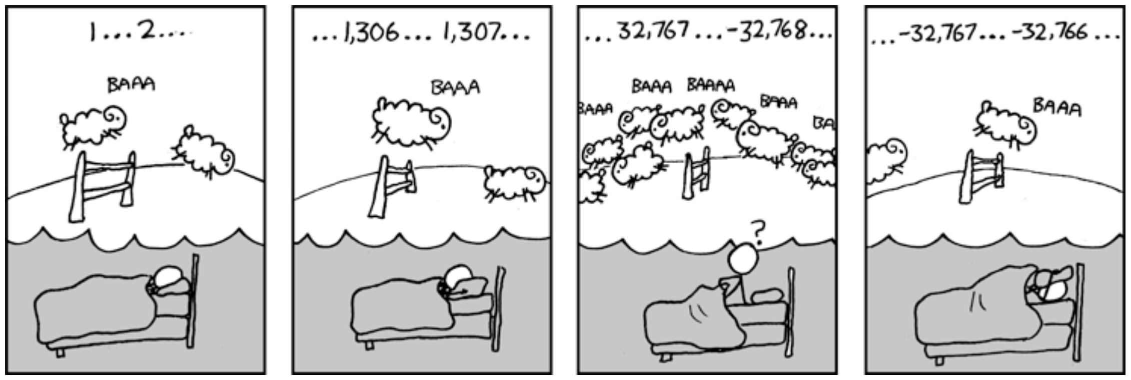

Figure 1. Can't Sleep (https://xkcd.com/571/). If androids someday DO dream of electric sheep, don't forget to declare sheepCount as a long int.

1. Lab Goals

- Explore the binary data representation of different types: `int`, `unsigned`, `float`, and `char`.
- Convert between decimal, binary, and hexadecimal representations.
- Apply arithmetic operations to binary numbers.
- Manipulate binary data with C bit-wise operators.
- Practice with C programming and the gdb debugger.

2. Lab Description

This lab assignment includes a written problem set (Part 1) and a programming exercise (Part 2).

3. Getting Lab 2 Starting Point Code

Download the starter files from Canvas.

- part1.txt
- Makefile
- lab2.c

4. Part 1. Problem Set

This part of the lab is a written assignment. You will write your solutions to the set of questions below in the `part1.txt` file.

If you would like to use vim, review the Vim lab.

```
$ vim part1.txt
```

Add your name and email address to the top of the `part1.txt` file.

The file contains some directions at the top for how to format certain numeric values, namely:

- for superscript use `^`, so `2^4` is 2^4 ("two to the fourth power").
- for subscripts use a number after a variable, so `X3` is X_3 ("X sub 3").
- for hexadecimal values, use the `0x` prefix, so `0x123a` is hexadecimal value "123a". Use lowercase letters when writing hexadecimal digits `a-f`.
- for binary values, use the `0b` prefix, so `0b101010` is binary value "101010".
- for decimal values, use no prefix, so `10431` is decimal value "10,431".

Line Wrap: Also, as you type in your answers, make sure to not have lines longer than 80 characters (explicitly hit the Enter key to start a new line). If you have lines longer than 80 characters, they will either appear to be cut off or wrapped strangely. I suggest making your editor window exactly 80 characters wide so that you can see when the current line is too long and starts to wrap around.

For these questions you will be graded on showing how you applied the operation or conversion method we described in class: you must show your work or explain how you got the result to receive credit. Check your answers for correctness by either writing a C program to do some of the computation and printing result values or by using gdb's print command. See lab2-supplement for details on using gdb.

Answer the questions below showing your work and/or explaining your answer.

- If a question specifies a number of bits, *your answer should be in a corresponding number of digits*.

    - For example, to add two 4-bit values together your answer should be a 4-bit value, not a 64-bit one!

- You can assume that **two's complement is used for signed values**, which makes the high-order bit an indication of the value's sign.
    - For example, `1000` should be interpreted as negative when treated as a *4-bit signed value* (high-order bit is `1`). But when interpreted as an *8-bit signed value* (`00001000`) it is positive (high-order bit is `0`).

Part 1 Questions

1. What is the largest positive value that can be represented with an **unsigned** 8-bit number? Explain.
2. What is the largest positive value that can be represented with a **2's complement** 8-bit number? Explain.
3. Convert the **unsigned** 8-bit binary value `0b10100110` to decimal. Show your work.
4. Convert the **signed** 8-bit binary value `0b10100110` to decimal. Show your work.
5. For the following **8-bit binary** values (show your work):

   ```
   value 1: 01011101
   value 2: 01100101
   ```

   a. Add the binary values together. What is the decimal representation of the result if it is interpreted as a **signed** 8-bit value?

   b. For the same addition, what is the decimal representation of the result if it is interpreted as an **unsigned** 8-bit value?

   c. What is the binary representation of the result of **adding** `value1` and `value2` together? Does this operation result in overflow? If so, under what interpretation of the values (signed/unsigned)?

   d. What is the binary representation of the result of **subtracting** `value2` from `value1`? Does this operation result in overflow? If so, under what interpretation of the values (signed/unsigned)?

6. Convert the following **2-byte binary numbers** to hexadecimal, indicating how each part is converted (the binary values are shown with spaces between each 4 digits just to make them easier to read):

   a. `0000 0110 0001 1111`

   b. `1100 0101 1110 0101`

   c. `1010 0111 1101 0110`

7. Convert the following **hexadecimal numbers to binary**, indicating how you converted each digit:

   a. `0x23`

   b. `0x852`

   c. `0xc1a6`

   d. `0xefab`

8. Convert the following **decimal values to 8-bit (2's complement)** binary and additionally convert the binary results to **hexadecimal**. Show your work:

   a. `12`

   b. `-36`

   c. `123`

   d. `-123`

9. Given the following **4-bit binary values**, show the results of each bit-wise operation, showing both the binary and decimal result value for each (list the unsigned decimal value):

   a. `0110 | ~(1010)`

   b. `~(0110 | 1010)`

   c. `0111 & ~(1001)`

   d. `(1010 | 0000) & 1111`

   e. `0011 ^ 1110`

   f. `0111 << 2`

   g. `0111 >> 2`

5. Part 2. C Programming

For this part, you will write a single C program that when run, prints out answers to each of the questions below. For each question, print out a string that is your answer to the question, and then print out some expressions and their results that support your answer to the question. For example, the beginning of a run of your program might look like this:

```
$ ./lab2
Question 1: my answer to question 1 is ...
```

```
    This can be verified by examining the result of the expression ...
    when x is the int value ... and y is ... the expression is ...
    when x is ... and y is ... the expression is ...

Question 2: my answer to question 2 is ...
    This can be verified by ...
```

Each answer should include printing out the result(s) of **computation(s)** that demonstrates your answer's correctness. **DO NOT** just print something like this:

```
printf("The answer to question 1, what 0x2 + 0x6, is 0x8\n");
```

Instead, have C code that computes the answer to show or to prove that your answer is correct:

```
unsigned x, y, z;
x = 0x2; y = 0x6;
z = x + y;
printf("The answer to question 1, what %x + %x is %x\n", x, y, z);
```

For some questions, the code proving your answer correct may be as simple as the example above. For others, however, you will have to think about how to construct some arithmetic expressions that demonstrate the correctness of your answer.

Answer these questions by writing a C program that prints out the answer and prints out example expression(s) that support your answer:

1. Given the following variable declaration and initialization, show how you can use **3 different format strings** in `printf` to display the value of `val` in different ways.

   Hint: If you aren't sure how to proceed, try taking a look at the list of format specifiers in the textbook.

   ```
   int val;
   val = 97;
   ```

2. What is the **maximum value** that can be stored in a C **unsigned int** variable (unsigned)?
3. What is the **maximum positive value** that can be stored in a C **int** variable (signed)?
4. What arithmetic operation is equivalent to left shifting an **unsigned int** value by 1?
5. What arithmetic operation is equivalent to left shifting an **unsigned int** value by 2?
6. What arithmetic operation is equivalent to right shifting an **unsigned int** value by 1?

6. Requirements

Part 1

- The answers to Part 1 should be entered in the `part1.txt` file.
- **Lines should be no longer than 80 characters** (use the "Enter" key to explicitly add line breaks, and run your editor in a window that is 80 chars wide):

Part 2

- The answers to Part 2 should be implemented in the `lab2.c` program, and when compiled and run, should output answers to each part.
- The answer to each question should be implemented as a separate function called by `main`. For example, your code and `main` might look like:

```
// function prototypes
void question1(void);
void question2(void);
void question3(void);
....
int main() {
  question1();  // call the question1 function
  question2();
  ...
}
..
void question1(void){
  ....
  //example print statement (not the actual statement you would right)
  printf("The sum of %d and %d is %d", x, y, x+y);
}
```

  You are welcome to add additional helper functions!

- Each answer should contain necessary and sufficient examples to support it. *Do not, for example, enumerate every possible `int` value*.
  - For most questions, it should be enough to **have 3 to 5 examples** to support your answer. *Do not have more than 10 for any one question*.
  - For questions 2 and 3, you don't really need more than one example if your example clearly demonstrates that your claim is true.
- **Examples in support of your answer must be computed by the C program.** For example, do not just print out the string "3 + 6 = 9" instead write C code that computes the expression and prints out its result, like this:

```
int x, y;
x = 3;
```

```
y = 6;
printf ("%d + %d = %d\n", x, y, (x+y));
```

- Your C program, when run, should print out the answer to each question in order, with supporting examples, in an easy-to-read format. Use formatted printf statements, and **do not print out lines that are longer than 80 characters** (break long output up into multiple lines).
  - Look at `printf` examples in the textbook here and here to use nice formatting for any tabular output your program might produce.
- Your code should be commented, modular, robust, and use meaningful variable and function names. This includes having a top-level comment describing your program and listing your name. *In addition, every function should include brief description of its behavior*.

7. Tips

- Remember that **type is important in C**!
  - If you give different formatting codes to `printf`, you can print out the same value as different types (for example, printing a value as `%c` looks different than printing it as `%d`).
  - If you are not seeing the values that you expect to see, check your `printf` format string and use `gdb` to examine your running program.

8. Submitting your Lab

Please remove any debugging output prior to submitting.

Zip your Lab2 folder containing part1.txt, lab2.c, and a screenshot showing the lab2 output. Submit the zip archive to Canvas by the deadline.