

## Assignment 1

Objectives:

- Conversion
  - Unsigned and signed arithmetic operations and overflow
  - C programming, endian and bit-level manipulation
- 

Submission:

- When creating your assignment, first include the question itself and its number then include your answer, keeping the questions in its original numerical order.
  - Submit your assignment electronically on CourSys. For this assignment, you need to submit 2 documents:
    - Your document called **Assignment\_1.pdf**, which must include your answers to all of the questions in Assignment 1 as well as a copy of your program **Assn1\_Q3.c**, i.e., your answer to the programming question.
      - Add your full name and student number at the top of the first page of your document **Assignment\_1.pdf**.
    - Your program **Assn1\_Q3.c**, i.e., your answer to the programming question.
      - Add your full name and student number in a header comment block at the top of your program.
      - Here is an example of a header comment block:

```
/*
 * Name: Anne Lavergne
 * Student number: 123456789
 */
```
    - You do not have to submit **Assn1\_main.c** or the **makefile**.
- 

Due:

- Thursday, Jan. 23 at 3pm
  - Late assignments will receive a grade of 0, but they will be marked in order to provide the student with feedback.
-

Requirements:

- **Show your work** (as illustrated in lectures).
- Whenever you are asked to write a C program in this course, your program must follow the following C standard:
  - Variables, constants and functions must be descriptively named.
  - Your code must be commented and well-spaced such that others (i.e., the instructor and the TA's) can read your code and understand it easily.
  - You cannot use the `goto` statement.

Marking scheme:

This assignment will be marked as follows:

- Questions 1 and 2 will be marked for correctness.
- The program for Question 3 will be tested for correctness, robustness and whether all the requirements were satisfied.

The amount of marks for each question is indicated as part of the question.

A solution will be posted after the due date.

## 1. [6 marks] Conversion

- a. Convert each of the **unsigned** decimal values below into its corresponding binary value ( $w = 8$ ), then convert the binary value into its corresponding hexadecimal value.
  - I.  $157_{10}$
  - II.  $248_{10}$
- b. Convert each of the **signed** decimal values below into its corresponding **two's complement** binary value ( $w = 8$ ), then convert the binary value into its corresponding hexadecimal value.
  - I.  $123_{10}$
  - III.  $-74_{10}$
- c. Interpret each of the binary values below first as an **unsigned** decimal value, then as a **signed** decimal value (using the **two's complement** encoding scheme).
  - I.  $11101001_2$
  - II.  $10010110_2$

- d. Convert  $247_{10}$  into a **signed** value directly, without converting it first to its corresponding binary value ( $w = 8$ ).
- e. Convert  $-112_{10}$  into a **unsigned** value directly, without converting it first to a binary number ( $w = 8$ ).

2. [6 marks] Unsigned and signed arithmetic operations and overflow

For **a.** below, convert each of the operands (**unsigned** decimal values) into its corresponding binary value ( $w = 8$ ).

For **b.** below, convert each of the operands (**signed** decimal values) into its corresponding **two's complement** binary value ( $w = 8$ ).

For **a.** and **b.** below, perform both the decimal addition and the binary addition and indicate the **true sum** and the **actual sum** and whether they are the same or different.

For the binary addition, clearly label all **carry in bits** (by using the label "carry in") and the **carry out bit** (by using the label "carry out").

Finally, indicate whether or not an overflow occurred (for **signed** values, specify whether the overflow is positive or negative). If an overflow occurred, explain how addition overflow can be detected ...

1. at the bit level, and
2. using the decimal operands.

**a.** Unsigned addition:

- I.  $74_{10} + 63_{10}$
- II.  $123_{10} + 157_{10}$

**b.** Signed (two's complement) addition:

- I.  $28_{10} + -74_{10}$
- II.  $-117_{10} + 126_{10}$
- III.  $74_{10} + 63_{10}$
- IV.  $-119_{10} + -105_{10}$

3. [8 marks] C Code, endian and bit-level manipulation

Download and extract **Assn1-files** and open **Assn1\_Q3.c**, **Assn1\_main.c** and **makefile** in a text editor. Read and understand their content. Using the makefile, compile and execute the program.

Requirements:

- While answering this question, you must not change the prototype of the functions given. The reason is that these functions will be tested using a test driver built based on these function prototypes.
- a. Modify the `printf` statement of the `show_bytes(...)` function such that it first prints the memory address of each byte then the content of the byte itself. Here is an example:
- ```
0x7ffe5fb887cc  0x80
```
- where `0x7ffe5fb887cc` is the memory address of a byte which contains the value `0x80`. Compile and test your program.
- b. Looking at the output of this program, would you say that the CSIL computer you are using is a **little endian** or a **big endian** computer? Justify your answer by including some of the output of your program in your answer.
- c. Modify the loop of the `show_bytes(...)` function such that, instead of using array notation to access each element of the array `start`, it uses pointer notation to access each of these elements. The output of your program should remain the same as in **a.** above.
- d. Write a function called `show_bits( ... )`. This function must have the following prototype:

```
void show bits(int);
```

This function must print the bit pattern of the parameter of type `int`. Compile and test your program. Here are two test cases (data and expected results) to illustrate the behaviour of this function:

**Test Case 1:** If the parameter (int) is 12345, then show `bits( ... )` prints:

000000000000000000000011000000111001

**Test Case 2:** If the parameter (int) is -12345, then show `bits( ... )` prints:

1111111111111111100111111000111

Add this function to **Assn1 Q3.c**.

- e. Write a function called `mask_LSBits( ... )`. This function must have the following prototype:

```
int mask LSbits( int n );
```

This function creates (returns) a mask with the  $n$  least significant bits set to 1. For example, if  $n$  is 2, the function returns 3 (i.e., `0x00000003`) and if  $n$  is 15, the function returns 32767 (i.e., `0x00007fff`). What happens when  $n \geq w$  or when  $n \leq 0$ ? When  $n \geq w$ , your function must return a mask of all 1's. When  $n \leq 0$ , your function must return a mask of all 0's, i.e., 0.

Requirements (for e.):

- In creating this function, when you create the mask, you cannot use division, modulus, multiplication, conditional or iterative statements (such as loops) or call other function(s).
- You can use conditional statements in your function when you validate the parameter.

Add this function to **Assn1\_Q3.c**.

**Testing:** For part e., make sure you test your code with valid and invalid test cases.

- Here is an example of a valid test case: calling `mask_LSBits(4)`.
- Here is an example of an invalid test case: calling `mask_LSBits(0)`.

In order to test your program, you can put your test cases in **Assn1\_main.c**.

---