COMS12200 lab. worksheet: week #3

- We intend this worksheet to be attempted in the associated lab. session, which represents a central form of help and feedback for this unit.
- The worksheet is not *directly* assessed. Rather, it simply provides guided, practical exploration of the material covered in the lecture(s): the only requirement is that you archive your work (complete or otherwise) in a portfolio via the appropriate component at

  https://wwwa.fen.bris.ac.uk/COMS12200/

  This forms the basis for assessment during a viva at the end of each teaching block, by acting as evidence that you have engaged with and understand the material.
- The deadline for submission to your portfolio is the end of the associated teaching block (i.e., in time for the viva): there is no requirement to complete the worksheet in the lab. itself (some questions require too much work to do so), but there is an emphasis on *you* to catch up with anything left incomplete.
- To accommodate the number of students registered on the unit, the single 3 hour timetabled lab. session is split into two $1\frac{1}{2}$ hour halves. You should attend *one* half only, selecting as follows:
  1. if you have a timetable clash that means you *must* attend one half or the other then do so, otherwise
  2. execute the following BASH command pipeline

     ```
     id -n -u | sha1sum | cut -c-40 | tr 'a-f' 'A-F' | dc -e '16i ? 2 % p'
     ```

     e.g., log into a lab. workstation and copy-and-paste it into a terminal window, then check the output: 0 means attend the first half, 1 means attend the second half.

Q1. A JavaScript-based online quiz relating to number systems should be accessible directly at

http://www.cs.bris.ac.uk/home/page/teaching/cs-quiz/conf/convert.jsp

or via the unit web-page: it presents randomly generated questions, which require conversion from one representation to another and so on. Although the application itself is fairly rudimentary[1], it offers a chance for hands-on practice with this topic and, crucially, to get immediate help and feedback.

Your challenge is simple: take the quiz regularly (e.g., every week or so), and regularly get over 70%. The results are not collected or formally marked. If you cannot regularly get a good score however, this hints at a need to revise the material carefully: mastering this topic is important.

Q2.
- The C source code

  http://www.cs.bris.ac.uk/home/page/teaching/material/arch_new/sheet/lab-3.q.c

  reads as follows:

```c
#include "lab-3.q.h"

void rep( int8_t x ) {
  printf( "%4d_{(10)} = ", x );

  for( int i = ( BITSOF( x ) - 1 ); i >= 0; i-- ) {
    printf( "%d", ( x >> i ) & 1 );
  }

  printf( "_{(2)}\n" );
}

int main( int argc, char* argv[] ) {
  int8_t t;

  t =    0; rep( t );
  t =   +1; rep( t );
  t =   -1; rep( t );
  t = +127; rep( t );
  t = -128; rep( t );

  return 0;
}
```

  It includes two functions:

[1] The quiz seems to work best using a WebKit-based web-browser (e.g., Chrome or Safari): it at least functions in Firefox, but one or two of the UI elements seem to be rendered incorrectly.

  - `rep` takes one argument `x` whose type is `int8_t`. The purpose of `rep` is to print the representation used for `x`: this allows you to see how theory from the lecture(s) is actually used in practice.
  - `main` acts as the entry point (i.e., where execution starts), which simply calls `rep` with various test values (i.e., 0, +1, −1 and so on).

- The associated C header file

  http://www.cs.bris.ac.uk/home/page/teaching/material/arch_new/sheet/lab-3.q.h

  reads as follows:

```
#ifndef __LAB_3_Q_H
#define __LAB_3_Q_H

#include <stdbool.h>
#include  <stdint.h>
#include   <stdio.h>
#include  <stdlib.h>

#define SIZEOF(x) ( sizeof(x)     )
#define BITSOF(x) ( sizeof(x) * 8 )

#endif
```

  It first includes `stdio.h` etc. to allow use of various C standard library functions such as `printf`, then defines two macros

  - `SIZEOF`, which gives the number of bytes used to represent the operand `x`, and
  - `BITSOF`, which gives the number of bits used to represent the operand `x`.

Download or type in the program, then compile it using a command such as

```
gcc -std=gnu99 -o lab-3.q lab-3.q.c
```

Execute the result to make sure you get the output expected, then use it to explore the following challenges:

a Using Wikipedia for example, do some research of your own the C bit-wise and shift operators; where relevant, write some short functions to experiment with their behaviour.

b Now armed with your knowledge about the operators involved, try to explain how `rep` works: for instance, what is the purpose of the expression `( x >> i ) & 1` and how does it work?

c Alter the program to answer the following:

- In the function `rep`, change the argument `x` so it has a different (integer) type. For instance, how and why does using an unsigned type such as `uint8_t` change the behaviour?
- In the function `main`, each call to `rep` is made with a manually selected input `t`. Motivated by the need for more exhaustive testing, imagine we need to try *all* possible values of `t`: how could we change `main` to do this, ideally in as general a way as possible (i.e., hard-coding as little as possible)?

Q3. The following questions (of increasingly difficulty) challenge you to apply concepts encountered previously in your *own* programs. In each case, the solution is a short C function, of no more than 10 lines or so: verify the function works correctly by calling it from `main` (in a similar way to `rep`, as above).

a Implement a function whose signature is

```
int sign( int8_t x );
```

and that returns 0 if `x` is positive, or 1 if `x` is negative. Try to write the function *without* using any C comparison operators.

b Implement a function whose signature is

```
int8_t neg( int8_t x );
```

and that returns the negation of `x`. Try to write the function *without* using the C minus operator.

c Implement a function whose signature is

```
uint8_t mod( uint8_t x, int n );
```

and that returns `x` modulo 2^{n}. Try to write the function *without* using the C modulo operator.

d Implement each of the following:

i A function whose signature is

```
int int2seq( bool* X, int8_t x );
```

that should extract and then store each `i`-th bit of `x` in the `i`-th element of array `X`; it should return the total number of elements stored.

ii A function whose signature is

```
int8_t seq2int( bool* X, int n );
```

that should basically reverse `int2seq` by returning a result `x` whose `i`-th bit (of `n` in total) matches the `i`-th element of array `X`.

iii A function whose signature is

```
void add( bool* R, bool* X, bool* Y, int n );
```

that realises the algorithm for long-hand, binary addition we developed in the lecture(s). `X` and `Y` are arrays of `n` bits, produced using `int2seq`; the function should compute `R` so when converted back using `seq2int`, it represents their sum.

Q4[+]. This is an extended rather than core question: it caters for differing backgrounds and abilities by supporting study of more advanced topics, but is therefore significantly more difficult and open-ended. As such, you should *only* attempt the associated tasks having completed all core questions (which satisfy the unit ILOs) first; even then, there is no shame in ignoring the question, or deferring work on it until later. Note that a solution will not *necessarily* be provided.

a Write a 1-line C expression that determines whether or not an unsigned, 8-bit integer `x` is an exact power-of-two. Put another way, if `x` $= 2^k$ for some k then the expression should evaluate to a non-zero result, otherwise it evaluates to zero. You can assume `x` itself is non-zero.

b In a previous question, you were asked to implement a simple variant of the signum (which is Latin for sign) function; the more accurate definition of this function is

$$\mathrm{sign}(x) = \begin{cases} -1 & \text{if } x < 0 \\ 0 & \text{if } x = 0 \\ +1 & \text{if } x > 0 \end{cases}$$

Write a 1-line C expression to compute this function given a signed, 8-bit integer `x`.