- Remember to register your attendance using the UoB Check-In app. Either
  1. download, install, and use the native app[a] available for Android and iOS, or
  2. directly use the web-based app available at

     https://check-in.bristol.ac.uk

  noting the latter is also linked to via the `Attendance` menu item on the left-hand side of the Blackboard-based unit web-site.
- The hardware *and* software resources located in the MVB Linux lab(s). (e.g., MVB-1.15 or MVB-2.11) are managed by the Faculty IT Support Team, a subset of IT Services. If you encounter a problem (e.g., a lab. workstation that fails to boot, an error when you try to use some software, or you just cannot log into your account), they can help: you can contact them, to report then resolve said problem, via

  https://www.bristol.ac.uk/it-support
- The lab. worksheet is written *assuming* you work in the lab. using UoB-managed and thus *supported* equipment. If you need or prefer to use your own equipment, however, various *unsupported*[b] alternatives available: for example, *you* could 1) manually install any software dependencies yourself, *or* 2) use the unit-specific Vagrant[c] box by following instructions at

  https://cs-uob.github.io/COMS10015/vm
- The questions are roughly classified as either C (for core questions, that *should* be attempted within the lab. slot), A (for additional questions, that *could* be attempted within the lab. slot), or R (for revision questions). Keep in mind that we only *expect* you to attempt the C-class questions: the other classes are provided *purely* for your benefit and/or interest, so there is no problem with nor penalty for totally ignoring them.
- There is an associated set of solutions is available, at least for the C-class questions. These solutions are there for you to learn from (e.g., to provide an explanation or hint, or illustrate a range of different solutions and/or trade-offs), rather than (purely) to judge your solution against; they often present *a* solution vs. *the* solution, meaning there might be many valid approaches to and solutions for a question.
- Keep in mind that various mechanisms exist to get support with and/or feedback on your work; these include both in-person (e.g., the lab. slot itself) *and* online (e.g., the unit forum, accessible via the unit web-site) instances.

[a] https://www.bristol.ac.uk/students/support/it/software-and-online-resources/registering-attendance

[b] The implication here is that such alternatives are provided in a best-effort attempt to help you: they are experimental, and so *no* guarantees about nor support for their use will be offered.

[c] https://www.vagrantup.com

# COMS10015 lab. worksheet #2: integer representation and arithmetic

Before you start work, download (and, if need be, unarchive[a]) the file

https://assets.phoo.org/COMS10015_2025_TB-4/csdsp/sheet/lab-02_q.tar.gz

somewhere secure[b] in your file system; from here on, we assume `${ARCHIVE}` denotes a path to the resulting, unarchived content. The archive content is intended to act as a starting point for your work, and will be referred to in what follows.

[a] For example, you could 1) use `tar`, e.g., by issuing the command `tar xvfz lab-02_q.tar.gz` in a terminal window, 2) use `ark` directly: use the `Activities` desktop menu item, search for and execute `ark`, use the `Archive`→`Open` menu item to open `lab-02_q.tar.gz`, then extract the contents via the `Extract` button, or 3) use `ark` indirectly: use the `Activities` desktop menu item, search for and execute `dolphin`, right-click on `lab-02_q.tar.gz`, select `Open with`, select `ark`, then extract the contents via the `Extract` button.

[b] For example, the `Private` sub-directory within your home directory (which, by default, cannot be read by another user).

## §1. C-class, or core questions

▷ **Q1[C].** The archive provided includes a C program, divided into two parts:

- Figure 1 illustrates the header file `rep.h`. 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`.
- Figure 2 illustrates the source code `rep.c`. It includes two functions:
  - `rep` takes one argument `x` whose type is `int8_t`. The purpose of `rep` is to print the representation of `x`.
  - `main` acts as the entry point (i.e., where execution starts), which simply calls `rep` with various test cases (i.e., values of `t` equal to 0, +1, −1, and so on).

You can use the program via an edit-compile-execute style design cycle, which, more concretely, means using a terminal[1] to execute the following commands:

a. Fix the working directory:

```
cd ${ARCHIVE}
```

b. Build the executable, i.e., compile the source code, using the `Makefile`[2] provided:

```
make rep
```

c. Execute the executable:

```
./rep
```

Notice that the left-hand side of the output produced shows the decimal *value* of some `x`, whereas the right-hand side shows the *representation*. Put simply, then the idea is that `rep` illustrates how theory from the lecture slot(s) is actually used in practice: looking at the relationship between left- and right-hand side, it should be clear that a given`x` is represented internally (i.e., "within the computer") as a sequence of 8 bits using two's-compliment. By using this starting point, you should be able to explore each of the following tasks/challenges:

a. Using Wikipedia, for example, improve your understanding of the C bit-wise and (both arithmetic and logical) shift operators; where relevant, write some short functions to experiment with their behaviour.

b. Now armed with your understanding of the operators involved, try to explain how `rep` works. For example, consider the expression `( x >> i ) & 1`: what does it do, and why (i.e., what purpose does it have)?

c. Alter the program to answer the following:

   - In the function `rep`, change the argument `x` so it has a different (integer) type. How and why does using an unsigned type such as `uint8_t` change the behaviour, for example?

[1] That is, within a BASH shell (or prompt, e.g., a terminal window) or similar: see, e.g., https://en.wikipedia.org/wiki/Unix_shell.
[2] https://en.wikipedia.org/wiki/Make_(software)

```
#ifndef __REP_H
#define __REP_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
```

**Figure 1:** *rep.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;
}
```

**Figure 2:** *rep.c*.

- 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 that we need to try *all* possible values of `t`: how could we change `main` to do this, ideally using as general an approach as possible?
- In the function `rep`, the right-hand side of the output, i.e., the representation of an `x` is expressed as a binary sequence. Imagine we want to express it by using hexadecimal, which acts as a "short-hand" for the same binary sequence: how could we change `rep` to do this, ideally using as general an approach as possible?

▷ **Q2[C].** The following questions challenge you to take various concepts encountered previously, and apply them in your *own* programs. In each case, the solution should be a short C function which applies the concept of "bit manipulation" (i.e., explicitly manipulating bits in some low-level representation to realise some higher-level function); take care to verify your solution works by using an appropriate call from `main`, as with `rep` above.

a. Implement a function whose prototype 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 prototype is

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

and that returns the (arithmetic) negation of `x`, such that `neg( x ) + x` = 0. Try to write the function *without* using the C negation or minus operators.

c. Implement a function whose prototype 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.

## §2. R-class, or revision questions

▷ **Q3[R].** There is a (large) set of questions available at

https://assets.phoo.org/COMS10015_2025_TB-4/csdsp/sheet/misc-revision_q.pdf

There are far too many for the lab. slot(s) alone, but, in the longer term, the idea is simple: attempt to answer the questions, applying theory covered in the lecture(s) to do so, as a means of revising and thereby *ensuring* you understand the material. Note that the set of questions is general-purpose, in the sense it covers *all* topics covered by the lecture(s): you will need to be selective, and only consider questions related to this lab. slot.

▷ **Q4[R].** An online, JavaScript-based quiz relating to number systems should be accessible directly at

https://assets.phoo.org/COMS10015_2025_TB-4/csdsp/sheet/quiz/jsq-conf_convert.html

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 *extremely* rudimentary[3] and somewhat in development, it provides hands-on practice and, perhaps more crucially, immediate (albeit automated) help and feedback with this topic.

The idea is simple: take the quiz regularly (e.g., every week or so), and regularly score at least 70%. Although the results are not collected or assessed, doing so acts as a means of revising and thereby ensuring you understand the material.

## §3. A-class, or additional questions

▷ **Q5[A].** In the lecture slot(s), we explored an algorithm for binary addition; the questions below task you with developing a concrete implementation of this algorithm in C. Taking care to test your functions after each step, implement each of the following:

a. A function whose prototype 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.

b. A function whose prototype 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`.

c. A function whose prototype is

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

that should compute binary addition of the `n` element sequences `X` and `Y`, producing their sum in `R`.

In combination, you should be able to write something like

```
int main( int argc, char* argv[] ) {
  bool X[ 8 ], Y[ 8 ], R[ 8 ];

  int x = 107;
  int y =  14;

  int2seq( X, x );
  int2seq( Y, y );

  add_seq( R, X, Y, 8 );

  int r = seq2int( R, 8 );

  printf( "add(%d,%d) = %d, %d + %d = %d", x, y, r, x, y, x + y );

  return 0;
}
```

[3] The application has only been tested at all with Chrome and Firefox, and even then in a fairly limited manner; if you identify a problem with the questions (e.g., it generates one that makes no sense, or has no answer) *or* the UI (e.g., some element is rendered incorrectly), we would be glad to know so we can potentially improve it.

and find r = 121 as expected, thereby explaining in enormous detail how the significantly easier and more obvious way to compute the same result (i.e., `r = x + y`) actually works!

▷ **Q6[A].** A **bit-set**[4] is a set data structure. Imagine you have a universe of $n$ objects: a bit-set $X$ captures set membership (resp. non-membership) of each $i$-th object by setting $X_i$, i.e., the $i$-th bit of $X$, to 1 (resp. 0). Put another way,

$$X_i = \begin{cases} 0 & \text{then object number } i \text{ is } \notin X \\ 1 & \text{then object number } i \text{ is } \in X \end{cases}$$

Unlike alternative set data structures (e.g., using a list of objects), a bit-set therefore captures the set *meaning* versus the *content*: it assumes we can give each object a number (i.e., an integer index), and that the objects themselves are stored in some *other* data structure. Although this is sometimes disadvantageous, the clear advantage is that it allows a) very compact representation, and b) very efficient operations on the set, by focusing at a low-level.

a. Implement

- a bit-set data structure, i.e., a structure `bs_t` which can represent sub-sets of a fixed sized, $n$-object universe.
- a function

```
void bs_rep( bs_t* X );
```

which prints a human-readable version of a bit-set represented by `X`.

Note that, ideally, if $w$ is the processor word size (e.g., $w = 32$ or $w = 64$) then your data structure should support a choice of $n > w$.

b. Implement functions for some standard set access operations, e.g.,

```
void bs_add ( bs_t* X, int i );
```

and

```
void bs_remove( bs_t* X, int i );
```

which add and remove object `i` from bit-set `X`, and

```
bool bs_is_member( const bs_t* X, int i );
```

which tests whether `i` is a member of bit-set `X`.

c. Implement functions for some standard set arithmetic operations, e.g.,

```
void bs_union( bs_t* R, const bs_t* X, const bs_t* Y );
```

which would compute `R`, the union of bit-sets `X` and `Y`.

▷ **Q7[A].** In the lecture slot(s), we discussed carry and overflow conditions when computing integer addition: both are errors, in the sense that the sum computed is incorrect, relating to the fixed range of representable values given any fixed $n$. Implement two functions whose prototypes are

```
uint8_t add_flag_u( uint8_t x, uint8_t y );
```

and

```
int8_t add_flag_s( int8_t x, int8_t y );
```

and that both return the sum of `x` and `y`. In addition, they should set a global variable called `flag` to signal whether said addition was correct, or produced a carry or overflow respectively.

Various approaches are possible, but, since this is an optional question, you may be interested to explore use of inline assembly language[5]. Each time an addition instruction is executed by the processor, it will update a set of (hardware) flags to reflect carry and overflow conditions. Although they cannot be accessed directly from C, you *can* access them from an alternative based on assembly language: embedding suitable instructions in your C program (i.e., inline vs. stand-alone) can be an attractive compromise versus writing whole programs in assembly language, and a useful topic to know something about more generally.

[4] https://en.wikipedia.org/wiki/Bit_array
[5] See, e.g., https://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html.