# 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.

Although some questions have a written solutions below, for others it makes more sense to experiment in a hands-on manner: the C source code

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

and associated header

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

support such cases.

**S2.** a This is an open-ended question, but selected examples below show how each relevant operator works.

- The bit-wise NOT operator works as follows:

$$
\begin{array}{lll}
 & \texttt{\~{}} & \texttt{12} \\
\mapsto & \neg & 12_{(10)} \\
\mapsto & \neg & 00001100 \\
\mapsto & \neg & \langle 0,0,1,1,0,0,0,0 \rangle \\
\mapsto & & \langle 1,1,0,0,1,1,1,1 \rangle \\
\mapsto & & 11110011 \\
\mapsto & & -13_{(10)}
\end{array}
$$

- The bit-wise AND operator works as follows:

$$
\begin{array}{rrcr}
 & \texttt{-6} & \texttt{\&} & \texttt{12} \\
\mapsto & -6_{(10)} & \wedge & 12_{(10)} \\
\mapsto & 11111010 & \wedge & 00001100 \\
\mapsto & \langle 0,1,0,1,1,1,1,1 \rangle & \wedge & \langle 0,0,1,1,0,0,0,0 \rangle \\
\mapsto & \langle 0,0,0,1,0,0,0,0 \rangle & & \\
\mapsto & 00001000 & & \\
\mapsto & 8_{(10)} & &
\end{array}
$$

- The bit-wise OR operator works as follows:

$$
\begin{array}{rrcr}
 & \texttt{-6} & \texttt{|} & \texttt{12} \\
\mapsto & -6_{(10)} & \vee & 12_{(10)} \\
\mapsto & 11111010 & \vee & 00001100 \\
\mapsto & \langle 0,1,0,1,1,1,1,1 \rangle & \vee & \langle 0,0,1,1,0,0,0,0 \rangle \\
\mapsto & \langle 0,1,1,1,1,1,1,1 \rangle & & \\
\mapsto & 11111110 & & \\
\mapsto & -2_{(10)} & &
\end{array}
$$

- The bit-wise XOR operator works as follows:

$$\begin{array}{rrcr}
 & \texttt{-6} & \texttt{\^{}} & \texttt{12} \\
\mapsto & -6_{(10)} & \oplus & 12_{(10)} \\
\mapsto & 11111010 & \oplus & 00001100 \\
\mapsto & \langle 0,1,0,1,1,1,1,1\rangle & \oplus & \langle 0,0,1,1,0,0,0,0\rangle \\
\mapsto & \langle 0,1,1,0,1,1,1,1\rangle & & \\
\mapsto & 11110110 & & \\
\mapsto & -10_{(10)} & &
\end{array}$$

- The (arithmetic) left-shift operator works as follows

$$\begin{array}{rrcr}
 & \texttt{-6} & \texttt{<<} & \texttt{2} \\
\mapsto & -6_{(10)} & \ll & 2_{(10)} \\
\mapsto & 11111010 & \ll & 2_{(10)} \\
\mapsto & \langle 0,1,0,1,1,1,1,1\rangle & \ll & 2_{(10)} \\
\mapsto & \langle 0,0,0,1,0,1,1,1\rangle & & \\
\mapsto & 11101000 & & \\
\mapsto & -24_{(10)} & & \\
\mapsto & \texttt{-24} & &
\end{array}$$

  which might be a bit confusing because of the way we wrote the bit-sequence: think of the operator as moving bits so they are placed at more-significant indices (which are at the left-hand end if we write a binary literal), filling the less-significant indices with the zero.

- The (arithmetic) right-shift operator works as follows

$$\begin{array}{rrcr}
 & \texttt{-6} & \texttt{>>} & \texttt{2} \\
\mapsto & -6_{(10)} & \gg & 2_{(10)} \\
\mapsto & 11111010 & \gg & 2_{(10)} \\
\mapsto & \langle 0,1,0,1,1,1,1,1\rangle & \gg & 2_{(10)} \\
\mapsto & \langle 0,1,1,1,1,1,1,1\rangle & & \\
\mapsto & 11111110 & & \\
\mapsto & -2_{(10)} & & \\
\mapsto & \texttt{-2} & &
\end{array}$$

  which might be a bit confusing because of the way we wrote the bit-sequence: think of the operator as moving bits so they are placed at less-significant indices (which are at the right-hand end if we write a binary literal), filling the more-significant indices with the sign bit.

The main point to grasp for the bit-wise operators is that they are simply applying the associated Boolean operator to corresponding bits of the operands. Following notation from the lecture(s), a unary operator $\oslash$ such as NOT computes

$$R_i = \oslash X_i$$

while a binary operator $\ominus$ such as AND computes

$$R_i = X_i \ominus Y_i.$$

In each case the operators work element-wise, producing each $i$-th bit of the result $R$ from corresponding $i$-th bit(s) of operand(s) $X$ and $Y$. Given the elements are bits, the the name "bit-wise operator" should therefore be clear. That is, the C bit-wise operators do exactly the same: by combining the operands `x` and `y` via

```
r = x & y;
```

we find the $i$-th bit of `r` is computed by AND'ing together the $i$-th bits of `x` and `y`. So understanding the result in each case boils down to understanding the representation of each operand (and the result), then just using the right truth table.

A second point to grasp is the difference between so-called logical and arithmetic shifts. In C at least, right-shifting a signed operand implies arithmetic shift whereas an unsigned operand implies logical shift (whereas in Java, for example, there are distinct operators to do this). The operands here were signed, so the last examples is an arithmetic arithmetic: this is important, because we fill the more-significant indices with the sign bit (rather than zero, as would be the case for a logical right-shift) to preserve the sign.

b The way `rep` works boils down to understanding what the expression

$$\texttt{( x >> i ) \& 1}$$

does. The purpose of this expression is to extract (or isolate) the `i`-th bit of `x`: the function as a whole iterates through *all* such bits using the `for` loop, printing them to demonstrate the underlying representation. Note that in this case we know `x` is an 8-bit, signed integer because of the type; the function is more general however, since the `BITSOF` macro will force the loop to iterate the correct number of times for any `x`.

The expression works in two steps. First it right-shifts `x` by a distance of `i` bits; this has the effect of moving the `i`-th bit of the operand (here `x`) into the 0-th bit of the result. Then, we AND this result with `1` which means we end up with *just* the 0-th bit: all other bits will be zero (because $0 \wedge t \equiv 0$ for any $t$). Overall, we can therefore say that

$$\texttt{( x >> i ) \& 1} \;=\; \begin{cases} 1 & \text{if the } \texttt{i}\text{-th bit of } \texttt{x} \text{ is } 1 \\ 0 & \text{if the } \texttt{i}\text{-th bit of } \texttt{x} \text{ is } 0 \end{cases}$$

c

- Following the recommendation, imagine the type of `x` is changed to `uint8_t`. Now, if we call `rep` with a signed argument then it is coerced (i.e., an implicit conversion inserted by the compiler, in contrast with an explicit cast written by the programmer) to match the function parameter: think of this as the compiler automatically changing each call to read

$$\texttt{rep( ( uint8\_t )( t ) );}$$

so `t` is converted from the source type `int8_t` into the target type `uint8_t` before being used as the argument `x`.

The conversion here is simple though. Rather than changing the underlying representation, it is just reinterpreted by using the target rather than source type: when `rep` is called using

$$\texttt{t} \;=\; 11111111 \;\mapsto\; -1_{(10)}$$

the compiler reinterprets `t` to provide the argument

$$\texttt{x} \;=\; 11111111 \;\mapsto\; 255_{(10)}.$$

So to cut a long story short, the output may not differ the way you expected: the underlying representation is the same in both cases, but `printf` maps it to a different value based on the type (i.e., either signed or unsigned).

The goal of this question is to stress that one representation (a sequence of bits) can be interpreted as many values: it just depends on our interpretation. In this case the interpretation is determined by the variable type, and, as a result, you might argue that taking care with types in your program is crucial because *mis*interpretation could easily lead to a bug.

- A solution might be something like the following:

```
int main( int argc, char* argv[] ) {
  int n     = BITSOF( int8_t );

  int lower = -( ( 1 << ( n - 1 ) )     );
  int upper = +( ( 1 << ( n - 1 ) ) - 1 );

  for( int t = lower; t <= upper; t++ ) {
    rep( t );
  }
}
```

In general an $n$-bit, signed integer $x$ represented using two's-complement can take values in the range

$$-2^{n-1} \leq x \leq +2^{n-1} - 1.$$

We want to iterate through all such values. Provided with $n$ by using `BITSOF`, the function first computes the same upper- and lower-bounds, and iterates through the range by using a `for` loop: note that `t` is initialised to the lower-bound and is then incremented after each iteration, and that the loop terminates once `t` equals the upper-bound.

There are two features which might not be obvious at first:

i The variable `t` is of type `int` rather than `int8_t` as before: why is this? The first point to grasp is how the `for` loop works. The easiest way is to rewrite it as a `while` loop:

```
int t = 0;
while( t <= 127 ) {
  rep( t );
  t++;
}
```

This means exactly the same, but highlights where each of the initialisation, comparison and update expressions in the `for` loop version are executed.
Consider the iteration where `t` = 127. Having first initialised `t`, we test whether `t` <= 127: it is, so we execute the loop body then increment `t`. Then the process repeats again, in the sense we test whether `t` <= 127. Beforehand we had `t` = 127, but then we incremented it; the issue therefore boils down to understanding what the result of 127 + 1 is.
The obvious answer is of course 128, and if `t` is of type `int` this is exactly what we get. If we use the type `int8_t` instead (meaning a signed, 8-bit two's-complement integer) we cannot represent this value: 127 + 1 overflows (or "wraps around") to −128. As a result we find that −128 <= 127, so basically the loop never terminates! To sum up, there are various ways to resolve this problem, but using the type `int` is arguably the most straightforward: we can represent all values within the required range, plus we know from the question above that when used as an argument in a call to `rep` it will be coerced into the correct type automatically.

ii The way the bounds `lower` and `upper` are computed might look quite (even overly) complicated: again the idea is to think about the representation of values in terms of bits. Both expressions depend on the fact that left-shift by a distance of $k$ bits is the same as multiplying by $2^k$. This is true because the left-shift operation moves each $i$-th bit in the operand into the $(i + k)$-th bit in the result: having been weighted by $2^i$ beforehand, it will be weighted by $2^{i+k}$ afterwards. For example,

$$\begin{aligned} & \texttt{1 << 2} \\ \mapsto\ & 1_{(10)} \ll 2_{(10)} \\ \mapsto\ & 00000001 \ll 2_{(10)} \\ \mapsto\ & 00000100 \\ \mapsto\ & 4_{(10)} \end{aligned}$$

and of course equals $4 = 2^2$. So if we set $k = n - 1$, then $1_{(10)} \ll (n - 1)$ evaluates to $2^{n-1}$ and our bounds are therefore as required.

**S4[+].** a This is a tricky question, but a suitable expression is as follows:

```
#define POWEROFTWO(x) ( ( (x) & ( (x) - 1 ) ) == 0 )
```

Note that this works for any $n$, i.e., any unsigned integer type we could give `x`. To see why, note that because `x` ≠ 0, there *must* be some $l$-th bit which is non-zero. As such, `x` − 1 will flip every $i$-th bit for $0 \leq i < l$ from 0 to 1. There are two cases to consider:

- If `x` is an exact power-of-two, exactly one bit in `x` is equal to 1: when we AND `x` (which only has the $l$ bit equal to 1) with `x` − 1 (which only has the $i$-th bits for $0 \leq i < l$ equal to 1) the result is always zero; the comparison with zero thus yields **true**. For example, if

  $$\texttt{x} = 4_{(10)} = 00000100_{(2)}$$

  then

  $$\texttt{x} - 1 = 3_{(10)} = 00000011_{(2)}$$

  and hence AND'ing them produces zero; the comparison with zero thus yields **true**.
- If `x` is not an exact power-of-two, more than one bit in `x` is equal to 1: when we AND `x` (which has multiple bits equal to 1) with `x` − 1 (which has the $i$-th bits for $0 \leq i < l$ equal to 1, plus at least one other $j$-th bit where $l < j < n$) the result is always non-zero; the comparison with zero thus yields **false**. For example, if

  $$\texttt{x} = 12_{(10)} = 00001100_{(2)}$$

  then

  $$\texttt{x} - 1 = 11_{(10)} = 00001011_{(2)}$$

  and hence AND'ing them produces non-zero; the comparison with zero thus yields **false**.

This of course fails if `x` is zero, which of course is not a power-of-two. Since `x` is unsigned we get

$$\texttt{x} = 0_{(10)} = 00000000_{(2)}$$

and

$$\texttt{x} - 1 = 255_{(10)} = 11111111_{(2)}$$

which, when AND'ed, imply `x` *is* a power-of-two. Though the question allows us to ignore this case, we *could* easily deal with it via an additional term in the expression if need be, e.g.,

```
#define POWEROFTWO(x) ( ( (x) != 0 ) && ( (x) & ( (x) - 1 ) ) == 0 )
```

b This is a tricky question, but a suitable expression is as follows:

```
#define SIGNUM(x) ( -( (x) >> ( BITSOF( (x) ) - 1 ) ) | ( ( -(x) ) >> ( BITSOF( (x) ) - 1 ) ) )
```

Note that this works for any $n$, i.e., any unsigned integer type we could give `x`, but replacing `BITSOF( x )` with the constant 8 works for this specific question. To see why, consider the two terms in isolation:

- The expression `-( (x) >> ( BITSOF( (x) ) - 1 ) )` extracts the MSB of `x`, then negates it. If `x` is negative, the shift gives $00000001_{(2)}$ which is negated to give $11111111_{(2)}$. If `x` is zero, the shift gives $00000000_{(2)}$ which is negated to give $00000000_{(2)}$. If `x` is positive, the shift gives $00000000_{(2)}$ which is negated to give $00000000_{(2)}$.
- The expression `( ( -(x) ) >> ( BITSOF( (x) ) - 1 ) )` extracts the MSB of `x` negated. If `x` is negative, the negation gives a positive result so the shift gives $00000000_{(2)}$. If `x` is zero, the negation gives zero so the shift gives $00000000_{(2)}$. If `x` is positive, the negation gives a negative result so the shift gives $00000001_{(2)}$.

Finally we OR together each one of the cases, finding that

- if `x` is negative the result is

$$11111111_{(2)} \vee 00000000_{(2)} = 11111111_{(2)} \mapsto -1_{(10)},$$

- if `x` is zero the result is

$$00000000_{(2)} \vee 00000000_{(2)} = 00000000_{(2)} \mapsto 0_{(10)},$$

  and

- if `x` is negative the result is

$$00000000_{(2)} \vee 00000001_{(2)} = 00000001_{(2)} \mapsto +1_{(10)},$$

hence realising the original requirement.