COMS10015 lab. worksheet #2

Although some questions have a written solution below, for others it will be more useful to experiment in a hands-on manner (e.g., using a concrete implementation). The file

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

supports such cases.

§1. C-class, or core questions

▷ **S1[C].** 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:

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

- 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:

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

- 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

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

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 slot(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

```
( 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

```
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 lo = -( ( 1 << ( n - 1 ) )     );
  int hi = +( ( 1 << ( n - 1 ) ) - 1 );

  for( int t = lo; t <= hi; 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 $-128 \leq 127$, 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 `lo` and `hi` 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,

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

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.

- A solution might be something like the following

```
char itox( int  x ) {
  if     ( x >=  0 && x <=  9 ) {
    return ( char )( ( int )( '0' ) + x       );
  }
  else if( x >= 10 && x <= 15 ) {
    return ( char )( ( int )( 'A' ) + x - 10  );
  }

  return '?';
}

void rep( int8_t x ) {
  printf( "%4d_{(16)} = ", x );

  for( int i = ( BITSOF( x ) - 4 ); i >= 0; i -= 4 ) {
    printf( "%c", itox( ( x >> i ) & 0xF ) );
  }

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

which can be explained via two changes to the `for` loop in `rep`. Both changes are motivated by the fact that a sub-sequence of 4 bits can be expressed using 1 hexadecimal digit: since there are $2^4 = 16$ possible 4-bit sequences and each hexadecimal digit is between 0 and 15, we have that

$$\begin{array}{rcl} \{0,0,0,0\} & \mapsto & 0_{(16)} \\ \{1,0,0,0\} & \mapsto & 1_{(16)} \\ & \vdots & \\ \{1,0,0,1\} & \mapsto & 9_{(16)} \\ \{0,1,0,1\} & \mapsto & A_{(16)} \\ \{1,1,0,1\} & \mapsto & B_{(16)} \\ \{0,0,1,1\} & \mapsto & C_{(16)} \\ \{1,0,1,1\} & \mapsto & D_{(16)} \\ \{0,1,1,1\} & \mapsto & E_{(16)} \\ \{1,1,1,1\} & \mapsto & F_{(16)} \end{array}$$

First, the loop counter initialisation and increment expressions is changed so that it decrements in steps of 4 rather than 1: if `BITSOF( x )` $= 8$ for example, we now have `i` $\in \{4,0\}$ rather than `i` $\in \{7,6,5,4,3,2,1,0\}$. Second, the loop body is changed so that 4-bit rather than 1-bit sub-sequence of `x` is extracted in each iteration. This sub-sequence is converted into a hexadecimal digit via a call to `itox`, then, finally, printed as before.

▷ **S2[C].** An associated solution, i.e., a (documented) implementation, for this question can be found in the archive provided. Beyond this, however, the `mod` function has a fairly wide range of potential solutions: some seem worthy of more detailed discussion, which is the goal of the following. To give a consistent example throughout, consider the case where

$$\begin{array}{rcr} \texttt{x} & = & 15_{(10)} \\ & \mapsto & 00001111_{(2)} \\ \texttt{n} & = & 2_{(10)} \end{array}$$

so we know `mod` should compute

$$\texttt{x} \bmod 2^{\texttt{n}} = 15 \bmod 2^2 = 3.$$

- If you look at the binary values involved, i.e.,

$$\begin{array}{rcr} \texttt{x} & = & 15_{(10)} \\ & \mapsto & 00001111_{(2)} \\ \texttt{x} \bmod 2^{\texttt{n}} & = & 3_{(10)} \\ & \mapsto & 00000011_{(2)} \end{array}$$

it becomes clear that computing x modulo 2^n, basically means retaining the n LSBs and "killing off" or zero'ing out all remaining MSBs of x. For example, here we want to retain the n = 2 LSBs. So, with this in mind, you *might* reasonably think that a good solution would be a function such as

```
uint8_t mod( uint8_t x, int n ) {
  return ( x << ( BITSOF( x ) - n ) ) >> ( BITSOF( x ) - n );
}
```

Why *might* this work? Given BITSOF(x) = 8 in this case (since uint8_t is the type of x), we know

$$\texttt{BITSOF(x) - n} = 8 - 2 = 6$$

and *should* find, in theory, that

$$\begin{array}{rcl} \texttt{x} & = & 00001111_{(2)} \\ \texttt{x} \ll 6 & = & 11000000_{(2)} \\ (\ \texttt{x} \ll 6\ ) \gg 6 & = & 00000011_{(2)} \end{array}$$

So by first left-shifting we kill off the BITSOF(x) - n, i.e., 6 MSBs, and the right-shift the result back into place to get the result we want. In practice, however, it fails to work: using the same example computes a result of 15, which is confusing.
The reason is quite subtle, but highlights the value of understanding the C type system. We can start by instead looking at a simpler expression, namely

```
x << ( BITSOF( x ) - n )
```

which we may as well write as

```
x << 6
```

given our example. If look at the C specification, e.g.,

https://webstore.ansi.org/RecordDetail.aspx?sku=INCITS/ISO/IEC+9899-1999+(R2005)

then Section 6.5.7 tells us what the semantics (or meaning) of left- and right-shift should be: for example, it states that

> Integer promotions are performed on each of the operands. The type of the result is that of the promoted left operand.

What does that mean? Well, Section 6.3.1.1 tells us

> If an int can represent all values of the original type, the value is converted to an int; otherwise, it is converted to an unsigned int.

and that, in a nutshell, explains the problem: whereas we wrote

```
x << 6
```

the compiler is interpreting this more like

```
( int )( x ) << 6
```

because the right-hand operand is (implicitly) cast, or converted into an int type before the shift happens; the type of the result is also int. So we *actually* compute

$$\begin{array}{rcr} \texttt{x} & = & 00001111_{(2)} \\ \texttt{(int)(x)} & = & 00000000000000000000000000001111_{(2)} \\ \texttt{(int)(x)} \ll 6 & = & 00000000000000000000001111000000_{(2)} \\ (\ \texttt{(int)(x)} \ll 6\ ) \gg 6 & = & 00000000000000000000000000001111_{(2)} \end{array}$$

and hence end up with the result 15 not 3 because the MSBs are not killed off as we expected. In fact, we can check this using

```
uint8_t mod_v1( uint8_t x, int n ) {
  return             ( x << ( BITSOF( x ) - n ) ) >> ( BITSOF( x ) - n );
}

uint8_t mod_v2( uint8_t x, int n ) {
  return ( uint8_t )( x << ( BITSOF( x ) - n ) ) >> ( BITSOF( x ) - n );
}
```

where the second function (explicitly) casts the right-hand operand of the right-shift into a `uint8_t`. You can see the impact of this additional cast using the `-S` flag when compiling via `gcc`: it produces assembly language as output, rather than an executable. Removing (a lot of) detail for clarity, we get

```
mod_v1: movl   $8,   %ecx
        movzbl %dil, %eax
        subl   %esi, %ecx
        sall   %cl,  %eax
        sarl   %cl,  %eax
        ret

mod_v2: movl   $8,   %ecx
        movzbl %dil, %eax
        subl   %esi, %ecx
        sall   %cl,  %eax
        movzbl %al,  %eax
        sarl   %cl,  %eax
        ret
```

Notice the only difference is an extra `movzbl` instruction, which can be read as "move zero-extended byte to long": it takes the least-significant byte of register `eax`, and zero-extends it (i.e., adds 24 zeros to the more-significant end). Either way, we get the correct result because

$$\begin{array}{rcr}
\texttt{x} & = & 00001111_{(2)} \\
(\ \texttt{int}\ )(\ \texttt{x}\ ) & = & 00000000000000000000000000001111_{(2)} \\
(\ \texttt{int}\ )(\ \texttt{x}\ ) \ll 6 & = & 00000000000000000000001111000000_{(2)} \\
(\ \texttt{uint8\_t}\ )(\ (\ \texttt{int}\ )(\ \texttt{x}\ ) \ll 6\ ) & = & 11000000_{(2)} \\
(\ \texttt{uint8\_t}\ )(\ (\ \texttt{int}\ )(\ \texttt{x}\ ) \ll 6\ ) \gg 6 & = & 00000011_{(2)}
\end{array}$$

So there we are: subtle, potentially quite annoying because the idea behind the solution seems sound, but actually correct behaviour per the C specification!

- What if we want to *avoid* the cast, and produce a solution which is more intuitive? We can (re)use the same reasoning to some extent, and consider this solution

```
uint8_t mod( uint8_t x, int n ) {
  return x ^ ( ( x >> n ) << n );
}
```

instead. At face value it *looks* similar; there are still two shifts, for example. Taking the same approach, i.e., inspecting the binary values at each step, we can show why this works:

$$\begin{array}{rcl}
\texttt{x} & = & 00001111_{(2)} \\
\texttt{x} \gg 2 & = & 00000011_{(2)} \\
(\ \texttt{x} \gg 2\ ) \ll 2 & = & 00001100_{(2)} \\
\texttt{x}\ \oplus\ (\ (\ \texttt{x} \gg 2\ ) \ll 2\ ) & = & 00000011_{(2)}
\end{array}$$

Put another way, we first kill off the *LSBs* (rather than MSBs as in the previous approach) by first right- and then left-shifting by `n`. We then use that result, in which only the original MSBs remain, to kill off the MSBs we wanted to in the first place: the XOR of said result with `x` achieves this, noting we *could* have used subtraction as an alternative.

Although the same reasoning to the above applies wrt. casting of the short operands, we are saved from the same problem by right-shifting as the first step: the value cannot "grow" to the right as it can if we extend the length via a cast to `int`, so we can be confident the LSBs are killed off as intended.

§2. R-class, or revision questions

▷ **S3[R].** There is a set of solutions available at

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

§3. A-class, or additional questions

▷ **S5[A].** An associated solution, i.e., a (documented) implementation, for this question can be found in the archive provided.

▷ **S6[A].** An associated solution, i.e., a (documented) implementation, for this question can be found in the archive provided.

▷ **S7[A].** An associated solution, i.e., a (documented) implementation, for this question can be found in the archive provided.