COMP 211: Lab 3
Fall, 2014

To get the code for this lab, go to the Assignments page and save the file `lab03.c0` to your COMP211 directory.

1 Words

Decimal notation and hex notation are two ways of writing words (32-bit numbers), which are the representation of the C0 type `int`. Numbers can be entered in either notation, but by default they are printed in decimal:

```
--> 15;
15 (int)
--> 0x0000000F;
15 (int)
```

The function `printhex` defined in the lab file prints an `int` in hex notation. Of course, you can write the argument in either decimal or hex notation.

```
--> printhex(0);
0x00000000
--> printhex(1);
0x00000001
--> printhex(15);
0x0000000F
--> printhex(16);
0x00000010

--> printhex(0x00000010);
0x00000010
```

You can use this to help with the tasks below.

1.1 Converting between different bases

Task 1.1 Write the number 255 in hex.

Task 1.2 Write the number 256 in hex.

Task 1.3 Write number `0x00000BED` in decimal.

Task 1.4 Write number `0x00000BED` in binary.

1.2 Bitwise And

There is an operation of "and" on two bits that is 1 when both bits are 1, and 0 otherwise:

```
0 & 0 == 0
0 & 1 == 0
1 & 0 == 0
1 & 1 == 1
```

More generally, the `&` operation can be applied to two words (`int`s); for each bit (`0` through `31`), this applies `&` to the two bits in that position from the two words. For 4-bit numbers, the general formula is

```
  b3     b2      b1      b0
& c3     c2      c1      c0
--------------------------------
 (b3&c3) (b2&c2) (b1&c1) (b0&c0)
```

So, for example,

```
  1010
& 1100
------
  1000
```

Task 1.5 What is `0x0000000A & 0x0000000C` (answer in hex)? Write out the binary to explain why.

One use of bitwise-and is to "select" parts of a hex number. For example, the expression `0x000000FF & p` keeps the last two hex digits of `p` the same, but turns the first 6 digits into 0. This can be thought of as "selecting" the last two digits. For example:

```
--> printhex(0x000000FF & 0x12345678);
0x00000078
```

This works because `0 & b = 0` and `1 & b = b`.

Task 1.6 Write an expression that "selects" the first two digits of a word.

Task 1.7 Write an expression that selects both the first two digits and the last two digits of a word.

1.3 Bitwise Or

There is an operation of "or" on two bits that is 1 when *either* bit is 1 (or both are), and 0 otherwise:

```
0 | 0 == 0
0 | 1 == 1
1 | 0 == 1
1 | 1 == 1
```

More generally, the `|` operation can be applied to two words (`int`s); for each bit (`0` through `32`), this applies `|` to the two bits in that position from the two words. For 4-bit numbers, the general formula is

```
  b3     b2      b1      b0
& c3     c2      c1      c0
--------------------------------
 (b3|c3) (b2|c2) (b1|c1) (b0|c0)
```

So, for example,

```
  1010
| 1100
------
  1110
```

Task 1.8 What is `0x0000000A | 0x0000000C` (answer in hex)? Write out the binary to explain why.

Because `1 | b = 1` and `0 | b = b`, we can use bitwise or (`|`) to "set" zeroes in a hex number to something else, keeping the other digits the same. For example, the expression `0x00000021 & p` keeps the first six hex digits of `p` the same, but turns the last two into `21`, assuming they were 0 to begin with. For example:

```
--> printhex(0x00000021 | 0x87654300);
0x87654321
```

Task 1.9 Write an expression that sets the first two digits of a word to `D4`, assuming they are 0 to begin with. For example, this should turn `0x00345678` into `0xD4345678`.

Task 1.10 Using both bitwise or and bitwise and, write an expression that changes the first two hex digits of a word to `D4`, keeping the rest the same. For example, this should turn `0x12345678` into `0xD4345678` and `0x876544321` into `0xD46544321`.

1.4 Shifts

The *left shift* operation `e << k` shifts each bit in the word `e` to the left by `k` positions, adding `k` 0s on the right, and dropping the leftmost `k` bits.

Task 1.11 What is the value of `0x12345678 << 4` (answer in hex)?

Task 1.12 What is the value of `0x12345678 << 8` (answer in hex)?

The *right shift* operation `e >> k` shifts each bit in the word `e` to the right by `k` positions and dropping the rightmost `k` bits. If the leftmost bit of `e` is a 0, 0s are added on the left; if the leftmost bit of `e` is a 1, 1s are added on the left.

Task 1.13 What is the value of `0x12345678 >> 4` (answer in hex)?

Task 1.14 What is the value of `0xE2345678 >> 4` (answer in hex)?

Task 1.15 Write a shift that turns `0x12345678` into `0x00000012`.

Task 1.16 Write a shift that turns `0x00000012` into `0x12000000`.

Task 1.17 Using shifts and bitwise and/or, turn `0xD4345678` into `0x000000D4`.

Task 1.18 Using shifts and bitwise and/or, select the third and fourth hex digits of a number and move them to be the last two digits. E.g. this will turn `0x12345678` into `0x00000034`.

2 Testing

In `lab03.c0`, you will find a function `addOne` that is supposed to add 1 to each element of an array.

Task 2.1 Create a function

```
void test_addOne()
```

that creates an array and tests this `addOne` function.

Use the command `assert(e);`, which, when `e` has type `bool`, evaluates `e` and then does nothing if `e` is `true`, or stops the program with an error if `e` is false.

The `addOne` code has a problem, so when you run your test function, you will see:

```
--> test_addOne();
lab03.c0:18.3-18.25: assert failed
Last position: lab03.c0:18.3-18.25
               test_addOne from <stdio>:1.1-1.11
```

This indicates that the test on line 18 fails. (The line may be different for you.)

Fix the problem with `addOne`. Then, your tests should succeed, and you will see:

```
--> test_addOne();
(void)
```

3 Nested Loops

An *increasing run* of numbers is a sequence of consecutive numbers, such as $1, 2, 3$ or $2, 3, 4, 5$. The *length* of an increasing run is the number of numbers in it (so $2, 3, 4, 5$ has length 4), and the *start* of an increasing run is the first number in it.

In this problem you will write a function

```
int[] increasing_runs(int num_runs, int run_length)
```

That makes an array containing `num_runs` increasing runs of consecutive numbers, each of length `run_length`, and each starting one number higher than the previous one, with the first one starting at 0.

For example, `increasing_runs(2,2)` should produce an array with two increasing runs, each of length 2, where the first starts with 0 (so $0, 1$), and the second starts with 1 (so $1, 2$):

0	1	1	2

`increasing_runs(3,2)` should produce an array with three increasing runs, each of length 2, the first starting with 0 (so $0, 1$) then 1 (so $1, 2$) then 2 (so $2, 3$):

0	1	1	2	2	3

`increasing_runs(2,3)` should produce an array with two increasing runs, each of length 3, starting with 0 (so $(0, 1, 2)$) then 1 (so $1, 2, 3$).

0	1	2	1	2	3

Task 3.1 In terms of `run_length` and `num_runs`, how big should the result of `increasing_runs` be? Put an appropriate `alloc_array` in `increasing_runs`.

Task 3.2 Fill in the array. Hint: Use a `for` loop inside a `for` loop, like this:

```
for (int i = 0; i < ... ; i = i + 1) {
  for (int j = 0; j < ...; j = j + 1) {
    ...
  }
}
```

Task 3.3 Use the function

```
// length should be the length of A
string show_array(int[] A, int length)
```

to test.