Week 03 Weekly Test Sample Answers

Test Conditions

These questions must be completed under self-administered exam-like conditions. You must time the test yourself and ensure you comply with the conditions below.

- You may complete this test in CSE labs or elsewhere using your own machine.
- You may complete this test at any time before **Wed Oct 07 21:00:00 2020**.
- Weekly tests are designed to act like a past paper - to give you an idea of how well you are progressing in the course, and what you need to work on. Many of the questions in weekly tests are from past final exams.
- Once the first hour has finished, you must submit all questions you've worked on.
- You should then take note of how far you got, which parts you didn't understand.
- You may choose then to keep working and submit test question anytime up to Wed Oct 07 21:00:00 2020
- However the maximum mark for any question you submit after the first hour will be 50%

You may access this **language documentation** while attempting this test:

- C quick reference
- MIPS quick reference

You may also access manual entries (the `man` command).

Any violation of the test conditions will results in a mark of zero for the entire weekly test component.

Set up for the test by creating a new directory called `test03`, changing to this directory, and fetching the provided code by running these commands:

```
$ mkdir test03
$ cd test03
$ 1521 fetch test03
```

Or, if you're not working on CSE, you can download the provided code as a zip file or a tar file.

WEEKLY TEST QUESTION:

Swap the bytes of a 16-bit Value

Your task is to add code to this function in **short_swap.c**:

```c
// given uint16_t value return the value with its bytes swapped
uint16_t short_swap(uint16_t value) {
    // PUT YOUR CODE HERE

    return 42;
}
```

Add code to the function `short_swap` so that, given a `uint16_t` value, it returns the value with its bytes swapped. You should use bitwise operators to do this.

For example:

```
$ ./short_swap 0x1234
short_swap(0x1234) returned 0x3412
$ ./short_swap 0xfade
short_swap(0xfade) returned 0xdefa
$ ./short_swap 0x01080
short_swap(0x1080) returned 0x8010
```

Use *make* to build your code:

```
$ make    # or 'make short_swap'
```

NOTE:

You may define and call your own functions if you wish.

DANGER:

You are not permitted to call any functions from the C library, or to use division (/), multiplication (*), or modulus (%)

You are not permitted to change the `main` function you have been given, or to change `short_swap`'s prototype (its return type and argument types).

When you think your program is working you can `autotest` to run some simple automated tests:

```
$ 1521 autotest short_swap
```

When you are finished working on this exercise you must submit your work by running **give**:

```
$ give cs1521 test03_short_swap short_swap.c
```

Sample solution for `short_swap.c`

```c
// Sample solution for COMP1521 lab exercises
// Swap bytes of a short

#include <stdint.h>
#include <stdlib.h>
#include <assert.h>

// given uint16_t value return the value with its bytes swapped
uint16_t short_swap(uint16_t value) {
    uint16_t bottom_byte = value & 0xFF;
    uint16_t top_byte = value >> 8;
    return (bottom_byte << 8) | top_byte;
}
```

WEEKLY TEST QUESTION:

Count the 1 Bits of a 64-bit Value

Your task is to add code to this function in **bit_count.c**:

```c
// return how many 1 bits value contains
int bit_count(uint64_t value) {
    // PUT YOUR CODE HERE

    return 42;
}
```

Add code to the function `bit_count` so that, given a `uint64_t` value, it returns how many 1 bits it contains. You should use bitwise operators to do this.

For example:

```
$ ./bit_count 0x123456789abcdef0
bit_count(0x123456789abcdef0) returned 32
$ ./bit_count 0xffffffffffffffff
bit_count(0xffffffffffffffff) returned 64
$ ./bit_count 0x0000000000000000
bit_count(0x0000000000000000) returned 0
$ ./bit_count 0x0000000400000000
bit_count(0x0000000400000000) returned 1
$ ./bit_count 0x8000000000000001
bit_count(0x8000000000000001) returned 2
```

Use *make* to build your code:

```
$ make    # or 'make bit_count'
```

NOTE:

You may define and call your own functions if you wish.

DANGER:

You are not permitted to call any functions from the C library, or to use division (`/`), multiplication (`*`), or modulus (`%`).

You are not permitted to change the `main` function you have been given, or to change `bit_count`'s prototype (its return type and argument types).

When you think your program is working you can `autotest` to run some simple automated tests:

```
$ 1521 autotest bit_count
```

When you are finished working on this exercise you must submit your work by running **give**:

```
$ give cs1521 test03_bit_count bit_count.c
```

Sample solution for `bit_count.c`

```c
// Sample solution for COMP1521 lab exercises
// count bits in a uint64_t

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

// return how many 1 bits value contains
int bit_count(uint64_t value) {
    int count = 0;
    while (value) {
        count += value & 1;
        value >>= 1;
    }
    return count;
}
```

Alternative solution for `bit_count.c`

```c
// COMP1521 19t3 ... test03_bit_count: count bits in a uint64_t
// An intrinsically better solution.
//
// 2019-09-28   Jashank Jeremy <jashank.jeremy@unsw.edu.au>

#include <assert.h>
#include <stdint.h>
#include <stdlib.h>

//
// If I allow myself the luxury of calling a function, this exercise is
// very simple.  But ... this _isn't_ a function! :-)
//
// Many compilers contain what are called 'intrinsic' operations: those
// which are implemented in hardware, sometimes.  In GCC and Clang,
// there are several ways to invoke intrinsic operations, but the safest
// and most portable is to use a '__builtin_'-prefixed function name.
//
// The compiler, as it's doing code generation, can replace calls to
// these intrinsics with real hardware instructions, if they exist, or a
// software implementation if they don't.
//
// Here, I invoke the 'popcountl' intrinsic, which "returns the number
// of 1-bits in x", and where "the argument type is unsigned long".
// That sounds like exactly what I'm looking for!
//
// So, if I compile this 'bit_count' function on the AMD64 processor
// architecture, the result is 27 instructions long, and most of them
// implement the intrinsic by doing bitwise hackery.  (How that
// implementation works is left an as exercise to the reader.)
//
// But if I allow my compiler (here, Clang) to use an architecture
// extension, SSE4::
//
//   :jaenelle: clang -msse4 -O1 -c bit_count.1.c
//   :jaenelle: objdump -d bit_count.1.o | sed -ne '/<bit_count>:/,$ {p}'
//   0000000000000060 <bit_count>:
//     60:  f3 48 0f b8 c7          popcnt %rdi,%rax
//     65:  c3                      retq
//
// And that's pretty cool!
//
// So how did I boil 27 instructions down to just two?  Because the SSE4
// extension adds an instruction, 'POPCNT', which does exactly what the
// intrinsic wants: "return the count of number of bits set to 1":
//   <https://www.felixcloutier.com/x86/popcnt>
//
// It's definitely worth having a look through a list of compiler
// intrinsics to see what else your compiler can do.  Some won't be
// obvious in their use -- often, they're extremely specialised, or
// perform operations that you almost certainly don't want to invoke.
//
// GCC's builtins and intrinsics list:
//   <https://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html>
// Clang's builtins and intrinsics list:
//   <https://clang.llvm.org/docs/LanguageExtensions.html#builtin-functions>
//
int bit_count(uint64_t value)
{
    return __builtin_popcountl (value);
}
```

Alternative solution for `bit_count.c`

```c
#include <assert.h>
#include <stdint.h>
#include <stdlib.h>

/*
Compiles to one instruction

popcnt  rax, rdi
ret
*/
int bit_count(uint64_t value) {
    int count = 0;
    while (value) {
        count++;
        value &= (value - 1); // fun bit twiddling
    }
    return count;
}
```

WEEKLY TEST QUESTION:

Swap Pairs of Bits of a 64-bit Value

Your task is to add code to this function in **bit_swap.c**:

```c
// return value with pairs of bits swapped
uint64_t bit_swap(uint64_t value) {
    // PUT YOUR CODE HERE

    return 42;
}
```

Add code to the function `bit_swap` so that, given a `uint64_t` value, it returns the same value with each pair of bits swapped: bit 0 should be swapped with bit 1, bit 2 should be swapped with bit 3, bit 4 should be swapped with bit 5, and so on. You should use bitwise operators to do this.

For example:

```
$ ./bit_swap 0x1111111111111111
bit_swap(0x1111111111111111) returned 0x2222222222222222
$ ./bit_swap 0x1111888855553333
bit_swap(0x1111888855553333) returned 0x22224444aaaa3333
$ ./bit_swap 0x0000000400000000
bit_swap(0x0000000400000000) returned 0x0000000800000000
$ ./bit_swap 0x8000000000000001
bit_swap(0x8000000000000001) returned 0x4000000000000002
$ ./bit_swap 0x123456789abcdef0
bit_swap(0x123456789abcdef0) returned 0x2138a9b4657cedf0
```

Use *make* to build your code:

```
$ make    # or 'make bit_swap'
```

NOTE:

You may define and call your own functions if you wish.

DANGER:

You are not permitted to call any functions from the C library, or to use division (`/`), multiplication (`*`), or modulus (`%`).

You are not permitted to change the `main` function you have been given, or to change `bit_swap`'s prototype (its return type and argument types).

When you think your program is working you can `autotest` to run some simple automated tests:

```
$ 1521 autotest bit_swap
```

When you are finished working on this exercise you must submit your work by running **give**:

```
$ give cs1521 test03_bit_swap bit_swap.c
```

Sample solution for `bit_swap.c`

```c
// COMP1521 19t3 ... test03_bit_swap: sample solution
// swap pairs of bits of a 64-bit value, using bitwise operators

#include <assert.h>
#include <stdint.h>
#include <stdlib.h>

// return value with pairs of bits swapped
uint64_t bit_swap(uint64_t value) {
    uint64_t new_value = 0;
    for (int i = 0; i < 64; i += 2) {
        uint64_t lower_bit = value & (((uint64_t)1) << i);
        uint64_t upper_bit = value & (((uint64_t)1) << (i + 1));
        new_value |= lower_bit << 1;
        new_value |= upper_bit >> 1;
    }
    return new_value;
}
```

Submission

When you are finished each exercise make sure you submit your work by running **give**.

You can run **give** multiple times. Only your last submission will be marked.

Don't submit any exercises you haven't attempted.

If you are working at home, you may find it more convenient to upload your work via give's web interface.

Remember you have until **Wed Oct 07 21:00:00 2020** to complete this test.

Automarking will be run by the lecturer several days after the submission deadline for the test, using test cases that you haven't seen: different to the test cases `autotest` runs for you.

(Hint: do your own testing as well as running `autotest`)

Test Marks

After automarking is run by the lecturer you can view it here the resulting mark will also be available via via give's web interface or by running this command on a CSE machine:

```
$ 1521 classrun -sturec
```

The test exercises for each week are worth in total 1 marks.

The best 6 of your 8 test marks for weeks 3-10 will be summed to give you a mark out of 9.