Week05 Laboratory Exercises

Objectives

- to practice using C's bitwi
- to understand how integer ed
- to practice manipulating d
- to explore working with bi alues
- to explore arbitrary precision integer arithmetic

Preparation

Before the lab you should re-read the relevant lecture slides and their accompanying examples.

Getting Started

Set up for the lab by creating a new directory called `lab05` and changing to this directory.

```
$ mkdir lab05
$ cd lab05
```


There are some provided files for this lab which you can fetch with this command:

```
$ 1521 fetch lab05
```

If you're not working at CSE, you can download the provided files as a zip file or a tar file.

EXERCISE — INDIVIDUAL:

Convert 16 Binary Digits to A Signed Number

Download sixteen_in.c, or copy it to your CSE account using the following command:

```
$ cp -n /web/cs1521/24T2/activities/sixteen_in/files.cp/sixteen_in.c sixteen_in.c
```

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

```
//
// given a string of binary digits ('1' and '0')
// return the corresponding signed 16 bit integer
//
int16_t sixteen_in(char *bits) {

    // PUT YOUR CODE HERE

    return 0;
}
```


Add code to the function `sixteen_in` which takes a sixteen-character string containing an ASCII positional representation of a binary number, and returns the corresponding signed integer. For example:

```
$ ./sixteen_in 0000000000000000
0
$ ./sixteen_in 1111111111111111
-1
$ ./sixteen_in 0011001100110011
13107
$ ./sixteen_in 1111000011110000
-3856
```


HINT:

Write a loop which, for each character, sets the appropriate bit of a `int16_t`-type result variable, using the bitwise operators `|` and `<<`.

NOTE:

`sixteen_in` can assume it is given a string of exactly sixteen characters, and every character is either `'0'` or `'1'`.

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

You are not permitted to call any functions from the C library.

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

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

```
$ 1521 autotest sixteen_in
```

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

```
$ give cs1521 lab05_sixteen_in sixteen_in.c
```

You must run `give` before **Monday 08 July 12:00 (midday)** (2024-07-08 12:00:00) to obtain the marks for this lab exercise. Note that this is an individual exercise, the work you submit with `give` must be entirely your own.

EXERCISE — INDIVIDUAL:

Convert a 16-bit Signed Number to Binary Digits

Download sixteen_out.c, or copy it to your CSE account using the following command:

```
$ cp -n /web/cs1521/24T2/activities/sixteen_out/files.cp/sixteen_out.c sixteen_out.c
```


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

```
// given a signed 16 b
// return a null-termi                binary digits ('1' and '0')
// storage for string                 malloc
char *sixteen_out(int1

    // PUT YOUR CODE H

}
```


Add code to the function `sixteen_out` so that, given a 16-bit signed integer it returns a string containing sixteen binary digits (`'0'` or `'1'`). For example:

```
$ ./sixteen_out 0
0000000000000000
$ ./sixteen_out -1
1111111111111111
$ ./sixteen_out 13107
0011001100110011
$ ./sixteen_out -3856
1111000011110000
```


HINT:

Write a loop which, for each bit (determined using the bitwise operators `&` and `<<`) sets the corresponding character in the string to a `'0'` or `'1'` . This should be structurally very similar to `sixteen_in` .

`sixteen_out` returns a string, whose storage space must be allocated using *malloc*. A string is a NUL-terminated character array; remember to allocate enough space for all the characters *and* the terminating NUL byte.

NOTE:

`sixteen_out` can assume its input is a value between 32767 and -32768 inclusive.

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

You are not permitted to call any functions from the C library, other than *malloc*.

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

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

```
$ 1521 autotest sixteen_out
```

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

```
$ give cs1521 lab05_sixteen_out sixteen_out.c
```

You must run `give` before **Monday 08 July 12:00 (midday)** (2024-07-08 12:00:00) to obtain the marks for this lab exercise. Note that this is an individual exercise, the work you submit with `give` must be entirely your own.

EXERCISE — INDIVIDU

Convert a 2 dig e to an Integer

Download bcd.c, or copy it to y ng the following command:

```
$ cp -n /web/cs1521/24T2 les.cp/bcd.c bcd.c
```

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


```
// given a  BCD encoded value between 0 .. 99
// return corresponding integer
int bcd(int bcd_value) {

    // PUT YOUR CODE HERE

    return 0;
}
```


Add code to the function `bcd` so that, given a 2 digit Binary-Coded Decimal (BCD) value, it returns the corresponding integer.

In binary-coded decimal format, each byte holds 1 decimal value (0 to 9), so each byte contains 1 decimal digit. For example:


```
$ ./bcd 0x07
7
$ ./bcd 0x0102              # note: 0x0102 == 258 decimal
12
$ ./bcd 0x0402              # note: 0x0402 == 1026 decimal
42
```

> HINT:
>
> Use the bitwise operators `&` and `>>` to extract each BCD digit.

> NOTE:
>
> `bcd` should return an integer value between 0 and 99 inclusive.
>
> You may define and call your own functions if you wish.
>
> You are not permitted to call any functions from the C library.
>
> You are not permitted to change the `main` function you have been given, or to change `bcd` 's prototype (its return type and argument types).

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

```
$ 1521 autotest bcd
```

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

```
$ give cs1521 lab05_bcd bcd.c
```


You must run `give` before **Monday 08 July 12:00 (midday)** (2024-07-08 12:00:00) to obtain the marks for this lab exercise. Note that this is an individual exercise, the work you submit with `give` must be entirely your own.

EXERCISE — INDIVIDU

Convert an 8 di BCD Value to an Integer

Download packed_bcd.c, or copy it to your CSE account using the following command:

```
$ cp -n /web/cs1521/24T2/activities/packed_bcd/files.cp/packed_bcd.c packed_bcd.c
```


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


```
// given a packed BCD encoded value between 0 .. 99999999
// return the corresponding integer
uint32_t packed_bcd(uint32_t packed_bcd_value) {

    // PUT YOUR CODE HERE

    return 0;
}
```


Add code to the function `packed_bcd` so that, given an eight-digit packed binary-coded decimal value, it returns the corresponding integer.

In packed binary-coded decimal format, each 4 bits holds 1 decimal value (0 to 9), so each byte contains 2 decimal digits. For example:

```
$ ./packed_bcd 0x42          # note: 0x42 == 66 decimal
42
$ ./packed_bcd 0x9999        # note: 0x9999 == 39321 decimal
9999
$ ./packed_bcd 0x42424242    # note: 0x42424242 == 1111638594 decimal
42424242
```

HINT:

Write a loop which extracts each BCD digit using the bitwise operators `&` and `>>` .

NOTE:

`packed_bcd` should return an integer value between 0 and 99999999 inclusive.

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

You are not permitted to call any functions from the C library.

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

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

```
$ 1521 autotest packed_bcd
```

When you are finished working, you must submit your work by running `give` :

```
$ give cs1521 lab05_packed_bcd
```

You must run `give` before **Monday (midday)** (2024-07-08 12:00:00) to obtain the marks for this lab exercise. Note that this is a [illegible] the work you submit with `give` must be entirely your own.

CHALLENGE EXERCISE — INDIVIDUAL:

Add 2 Arbitrary Length BCD Values

Download bcd_add.c, or copy it to your C85 account using the following command:

```
$ cp -n /web/cs1521/24T2/activities/bcd_add/files.cp/bcd_add.c bcd_add.c
```


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

```
big_bcd_t *bcd_add(big_bcd_t *x, big_bcd_t *y) {

    // PUT YOUR CODE HERE

}
```


Add code to the function `bcd_add` so that, given 2 arbitrary length binary-coded decimal numbers, it returns their sum. For example:

```
$ ./bcd_add 123456789123456789 123456789123456789
246913578246913578
$ ./bcd_add 999999999999999999 1
1000000000000000000
$ ./bcd_add 7777777777777777777777777777777777 8888888888888888888888888888888888
16666666666666666666666666666666665
$ ./bcd_add 98765432198765432198765432 98765987659876598765
98765442075364198186425308 6
```

HINT:

Use *realloc* if you need to grow an array.

You will be working with pointers to `struct` s with a field that is a pointer to an array. It would be wise to revise pointers, structs, typedefs, arrays, and dynamic memory allocation. The relevant videos in this playlist may help.

You can use Python (for example) to check what the sum of any two integers is:

```
$ python3 -i
Python 3.7.3 (default, Apr  3 2019, 05:39:12)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> 123456789123456789 + 123456789123456789
246913578246913578
```


NOTE:

You may define and call y[illegible]you wish.

You are not permitted to [illegible] the C library, other than *malloc*, *calloc* and *realloc*.

You are not permitted to [illegible]ction, to change any other functions you have been given, to change the type `big_bcd_t`, or to change the return type or argument types of `bcd_add`.

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

```
$ 1521 autotest bcd_add
```


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

```
$ give cs1521 lab05_bcd_add bcd_add.c
```


You must run `give` before **Monday 08 July 12:00 (midday)** (2024-07-08 12:00:00) to obtain the marks for this lab exercise. Note that this is an individual exercise, the work you submit with `give` must be entirely your own.

CHALLENGE EXERCISE — INDIVIDUAL:

MIPS NUXI

We have two 32 bit values which the bytes have placed in an unknown order.

Fortunately we know the 4 bytes of the first value originally contained the ASCII values `"UNIX"`, and the two values were shuffled in an identical manner.

e.g. if the first value was `"IXUN"`, and the second value was `"PSMI"`, then the second value correctly ordered would be `"MIPS"`.

Write a MIPS program `nuxi.s` which read the two values and prints the second value with its bytes correctly ordered.

For example:

```
$ 1521 mipsy nuxi.s
1481199189
-2023406815
-2023406815
$ 1521 mipsy nuxi.s
1431193944
-2023406815
558065031
$ 1521 mipsy nuxi.s
1230525774
-559038737
-1377898562
$ 1521 mipsy nuxi.s
1229871189
305419896
1444033656
```


HINT:

This challenge should not require the use of any bitwise operations.

A major part of this challenge is understanding the problem. You should get started by printing the signed integers above in hexadecimal, and try to match up the first input with the letters `UNIX` on the ASCII table, and see how their shuffling has effected the second input.

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

```
$ 1521 autotest nuxi
```


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

```
$ give cs1521 lab05_nuxi nuxi.s
```


You must run `give` before **Monday 08 July 12:00 (midday)** (2024-07-08 12:00:00) to obtain the marks for this lab exercise. Note that this is an individual exercise, the work you submit with `give` must be entirely your own.

CHALLENGE EXERCISE — INDIVIDUAL:

Subtract, Multiply and Divide 2 Arbitrary Length BCD Values

Download bcd_arithmetic.c, or copy it to your CSE account using the following command:

```
$ cp -n /web/cs1521/24T2/activities/bcd_arithmetic/files.cp/bcd_arithmetic.c bcd_arithmetic.c
```

Add code to the functions `bcd_add` , `bcd_subtract` , `bcd_multiply` , and `bcd_divide` so that, given two arbitrary-length binary-coded decimal (BCD) numbers, they return the result of the corresponding arithmetic operation. For example:

```
$ ./bcd_arithmetic 1123456789123456789 - 1123456789123456788
1
$ ./bcd_arithmetic 123456789123456789 '*' 123456789123456789
15241578780673678515622620750190521
$ ./bcd_arithmetic 15241578780673678515622620750190521 / 123456789123456789
123456789123456789
$ ./bcd_arithmetic 123456789 '*' 987654321 +  987654321 / 1234
121932631113435637
$ ./bcd_arithmetic 14 /
2
```


HINT:

The code you are given a '*', and '/', and calls `bcd_add`, `bcd_subtract`, `bcd_multiply`, and `bcd_divide` respectively. You don't need to understand this code to do the exercises, though you will find it interesting to read. You only need to implement the four arithmetic functions.

Use *realloc* to grow an array.

You could use Python (for example) to check what the value of an expression is, (though you don't need to quote the `*`, and you need to use `//` to get integer division):


```
$ python3 -i
Python 3.7.3 (default, Apr  3 2019, 05:39:12)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> 123456789 * 987654321 + 987654321 // 1234
121932631113435637
```


NOTE:

You can assume the results of subtraction are non-negative.

You can assume that divisors will be non-zero.

Integer division should yield only the integer part of the result. In other words, truncate towards zero; do not round.

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

You are not permitted to call any functions from the C library, other than *malloc*, *calloc* and *realloc*.

You are not permitted to change the `main` function, to change any other functions you have been given, to change the type `big_bcd_t`, or to change the return type or argument types of `bcd_add`, `bcd_subtract`, `bcd_multiply`, or `bcd_divide`.

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

```
$ 1521 autotest bcd_arithmetic
```

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

```
$ give cs1521 lab05_bcd_arithmetic bcd_arithmetic.c
```

You must run `give` before **Monday 08 July 12:00 (midday)** (2024-07-08 12:00:00) to obtain the marks for this lab exercise. Note that this is an individual exercise, the work you submit with `give` must be entirely your own.

Submission

When you are finished each ex u submit your work by running `give` .

You can run `give` multiple tim omission will be marked.

Don't submit any exercises you

If you are working at home, you venient to upload your work via give's web interface.

Remember you have until **Week 7 Monday 12:00:00 (midday)** to submit your work without receiving a late penalty.

You cannot obtain marks by e-mailing your code to tutors or lecturers.

You check the files you have submitted here.

Automarking will be run by the lecturer several days after the submission deadline, using test cases different to those `autotest` runs for you. (Hint: do your own testing as well as running `autotest` .)

After automarking is run by the lecturer you can view your results here. The resulting mark will also be available via give's web interface.

Lab Marks

When all components of a lab are automarked you should be able to view the the marks via give's web interface or by running this command on a CSE machine:


```
$ 1521 classrun -sturec
```

COMP1521 24T2: Computer Systems Fundamentals is brought to you by
the School of Computer Science and Engineering
at the University of New South Wales, Sydney.
For all enquiries, please email the class account at cs1521@cse.unsw.edu.au