

# ENSC 254 Lab Assignment 1

## Important Logistics:

- Some general grading logistics and lab computer access instructions have been posted on our course website.
- Lab 1 weighs 4% of the final marks. It is done and graded individually. It includes 3 parts and 25 points in total. If you get X points out of 25 points, it will be scaled as  $X/25 * 4\%$  of the final marks. This is a warmup lab to prepare you for lab 2 to 5, and also serves as a reference point for you to find your teammate for lab 2 to 5.
- Please make sure your code can compile and run correctly on the lab computers (i.e., FASRLA Linux computers). If your code cannot compile on the lab computers, then you get 0 mark for lab 1. If your code can compile, but cannot run correctly, then you only get the points where your code runs correctly on the lab computers.

## Introduction of Lab 1:

The purpose of this assignment is to refresh your memory on bit-level representations and operations of integer numbers (part 1), struct and union data structures (part 2), pointer-based data structure and debugging using gdb (part 3), which you should have already learned from CMPT 128 or ENSC 151, ENSC 251 and ENSC 252. TA also gave a tutorial on these topics. This lab would help warm you up for the upcoming lab 2 to 5, and provide a reference point for you to find your teammate for lab 2 to 5.

To start the assignment, download lab1.zip from the course website and put it into a Linux machine.

1. `unzip lab1.zip` //decompress the zip file
2. `cd lab1` //go to the lab1 folder
3. `ls` //you can see three subfolders under lab1: part1, part2, and part3

## Part 1: Bit-Level Operations [16 points]

In part 1, you will solve a series of programming puzzles. **The only file you will modify is bits.c in the part1 folder.** The bits.c file contains a skeleton for each of the 8 programming puzzles. Your assignment is to complete each function skeleton using only *straightline* code for the integer puzzles and a limited number of C arithmetic and logical operators. Specifically,

- The only allowed data type is *int*, no type casting allowed
- You are *only* allowed to use the following eight operators: `! ~ & ^ | + << >>`
- A few of the functions further restrict the above list
- Also, you are not allowed to use any constants longer than 8 bits
- No loops or conditionals (e.g., if/else, switch/case) or sub-functions are allowed
- **Read the comments in bits.c for detailed rules and a discussion of desired coding style**

The table below lists the puzzles in rough order of difficulty from easiest to hardest. The “**Rating**” field gives the difficulty rating, i.e., **the number of points**, for the puzzle, and the “Max ops” field gives the maximum number of operators you are allowed to use to implement each function. See the comments in bits.c for more details on the desired behavior of the functions. You may also refer to the test functions in tests.c. These are used as reference functions to express the correct behavior of your functions, although they don’t satisfy the coding rules for your functions.

Name	Description	Rating /Points	Max ops
tmax	Return maximum two's complement integer 0x7fffffff	1	4
isZero(x)	Return 1 if x == 0, and 0 otherwise	1	2
bitXor(x,y)	$x \oplus y$ using only ~ and &	1	14
isNotEqual(x, y)	Return 0 if x == y, and 1 otherwise	2	6
sign(x)	Return 1 if positive, 0 if zero, and -1 if negative	2	10
conditional(x, y, z)	Same as $x ? y : z$	3	16
replaceByte(x, n, c)	Replace byte n in x with c. Bytes numbered from 0 (Least Significant Bytes) to 3 (Most Significant Byte)	3	10
rotateRight(x, n)	Rotate x to the right by n	3	25

### Test and grading for part1:

For each puzzle, you need to pass both the correctness check and max ops coding check. If you pass both checks, you get all its points of that puzzle; otherwise, you get 0 point for that puzzle.

#### 1. Correctness check using *btest*:

**btest** checks the functional correctness of the functions (i.e., puzzles) in bits.c. To build and use it, type the following two commands (inside the part1 folder) on a lab computer:

1. make
2. ./btest

**Note that you must rebuild *btest* each time you modify your bits.c file.**

You’ll find it helpful to work through the functions one at a time, testing each one as you go. You can use the -f flag to instruct **btest** to test only a single function:

1. ./btest -f bitXor

You can feed it specific function arguments using the option flags -1, -2, and -3:

1. ./btest -f bitXor -1 4 -2 5

**Note although btest gives you the points, they are not the final ones that you get. You must also pass the max ops coding check using dlc to get the corresponding points for a puzzle.**

#### 2. Max ops coding check using *dlc*:

**dlc** is a modified version of an ANSI C compiler from the MIT CILK group that you can use to check for compliance with the coding rules for each puzzle. The typical usage is:

1. `./dlc bits.c`

The **dlc** check runs silently unless it detects a problem, such as an illegal operator, too many operators, or non-straightline code in the integer puzzles. **Note if dlc detects a problem for your puzzle, you get 0 point for that puzzle.**

To print the number of operators used by each function, you can run **dlc** with the `-e` switch:

1. `./dlc -e bits.c`

## Part 2: Struct and Union [6 points]

In part 2, you will convert an *unsigned int* into an *Instruction* object. **The only file you will modify is test.c in the part2 folder.**

*Instruction* is a 32-bit long **union** data type defined in types.h, which is shown below.

```
typedef union {
```

```
    /* access rtype with: instruction.rtype.(opcode|rd|funct3|rs1|rs2|funct7) */
```

```
    struct {
```

```
        unsigned int opcode : 7;
```

```
        unsigned int rd : 5;
```

```
        unsigned int funct3 : 3;
```

```
        unsigned int rs1 : 5;
```

```
        unsigned int rs2 : 5;
```

```
        unsigned int funct7 : 7;
```

```
    } rtype;
```

```
    /* access itype with: instruction.itype.(opcode|rs|rt|imm) */
```

```
    struct {
```

```
        unsigned int opcode : 7;
```

```
        unsigned int rd : 5;
```

```
        unsigned int funct3 : 3;
```

```
        unsigned int rs1 : 5;
```

```
        unsigned int imm : 12;
```

```
    } itype;
```

```
} Instruction;
```

The content of an *Instruction* object (i.e., variable) can be interpreted as either a *rtype struct* value (representing a register-type instruction, which we will use in lab assignments 2 and 3) or an *itype struct* value (representing an immediate-type instruction, which we will use in lab assignments 2 and 3), depending on its opcode (i.e., the lowest 7 bits, bits 0 to 6).

As shown in the code above, a *rtype struct* object (i.e., variable) is 32-bit long and has six member fields (we will explain the detailed meaning of these fields in lab 2 and 3).

- The lowest 7 bits are for *opcode*
- The next higher 5 bits are for *rd*
- The next higher 3 bits are for *funct3*
- The next higher 5 bits are for *rs1*
- The next higher 5 bits are for *rs2*
- The highest 7 bits are for *funct7*

Similarly, an *itype struct* object (i.e., variable) is also 32-bit long but has five member fields. Its first four member fields are the same as those in *rtype struct*. Its fifth member field, which is the highest 12 bits, is for *imm*.

In test.c, two print functions *print\_rtype* and *print\_itype* are already provided to print out an Instruction object as a *rtype* object and an *itype* object.

Your assignment is to complete two function skeletons (*parse\_rtype* and *parse\_itype*) to parse an *unsigned int* input, extract its corresponding bits, and fill in the *Instruction* object's member fields (*rtype* and *itype struct*, respectively), and return the *Instruction* object. Please do NOT change any other code besides these two function bodies. There is no limitation on the allowed operators.

## Test and grading for part2:

To build and test it, type the following two commands (inside the part2 folder) on a lab computer:

1. `make`
2. `./test //` This will print out the parsed Instruction objects

The golden output is provided in golden.txt. To verify the correctness of your part2 code, redirect the output to output.txt and compare it against golden.txt, which should exactly match with each other.

1. `./test >& output.txt`
2. `diff output.txt golden.txt`

The first two lines of output are for rtype instructions, which counts for **3 points if both lines match (otherwise, 0 point)** between output.txt and golden.txt. Similarly, the last two lines of output are for itype instructions, which counts for **3 points if both lines match (otherwise, 0 point)** between output.txt and golden.txt.

## Part 3: Pointers and Debugging [3 points]

In part 3, an example linked list code is provided and there is a segmentation fault bug if you build and run the code, by typing the following two commands (inside the part3 folder) on a lab computer:

1. `make`

## 2. ./test

Your assignment is to use gdb to locate the bug and then fix the bug. Note the bug is NOT in the main function and you are NOT allowed to change the main function. In your final submitted code, extra printf/cout statements are NOT allowed. After your fix, your code should run correctly (no error message) and you will get 3 points; otherwise, you get 0 point.

### Assignment Submission:

Your lab assignment 1 will be submitted electronically. You will need to submit a single ***YOUR\_SFU\_COMPUTING\_ID.zip*** file. This format makes it easy for TA to do auto-grading. Failure to comply with this format will result in a **ZERO** point for this assignment. To zip your files in Linux,

1. Go to your lab1 folder
2. `cd part1`; make clean **//make sure you clean your files in part1 sub-folder**
3. `cd ../part2`; make clean **//make sure you clean your files in part2 sub-folder**
4. `cd ../part3`; make clean **//make sure you clean your files in part3 sub-folder**
5. `cd ../../` //go two levels up
6. `mv lab1 YOUR_SFU_COMPUTING_ID` //rename your lab1 folder using your SFU Computing ID
7. `zip -r YOUR_SFU_COMPUTING_ID.zip YOUR_SFU_COMPUTING_ID` //zip all your lab1 files into a single .zip

### Submission Deadline:

You need to meet the deadline: every 10 minutes late for submission, you lose 10% of the points; that is, 100 minutes late, you will get zero for this lab.

### Lab Demonstration:

To finalize your lab 1 points graded through part 1 to part 3, you will have to demo your lab code to your TA in the lab sessions that you enrolled. Due to the large enrollment, TA will **randomly** select some students (about 30% students) to do the lab 1 demo. If you are NOT selected to do the demo, the auto-graded points are your final points.

For those who are selected to do the demo, each student has around 10 minutes to explain their code to the TA. Only code from your submission is allowed in the lab demo. If you fail to do the demo (without a medical note), you will be awarded zero on this lab assignment. **If it is determined that you do not understand the code being evaluated, you will be awarded zero and considered as CHEATING on this lab assignment.** Also please show up in the demo day on time (TA will send out your scheduled time), otherwise you will lose 10% of the lab 1 points.