Lab 05 Specification – Explore usage of Pointers and Automating Binary Divison
Due (via your git repo) no later than 8 a.m., Friday, 26th October 2018.
50 points
2 weeks Individual Lab

Lab Goals

- Practice binary divison using the restore method
- Practice usage of Pointers

You are required to work individually on this lab. Lab work may be done offline. But the expectation is that every student should attend at least the first 20 minutes of the lab sessions. As you may be aware that, I go around and check with the students in the lab after the first 15 minutes of the lab session (that is, around 2:45 PM precisely) for clarification questions on the lab requirement. I expect every student should be in attendance, and be present in the lab till that time. This will also help the student to interact with other students, teaching assistants, and the Professor.

The key for doing well in this lab, is that don't wait till the last week to complete this lab. There is a purpose behind making this lab a two weeks lab. A good amount of coding is required to successfully finish the lab and to receive full points for this lab. So, work gradually throughout the two weeks window, to properly implement all the requirements of this lab.

Attendance is not required for next week lab 10/19/2018 as such, but it is recommended that you would come to the lab to ask any questions regarding the lab. Both the TA's and I will be available to answer your questions next week during the entire lab period.
So, you should take advantage of our time.

Suggestions for Success

- Take a look at the suggestions for successfully completing the lab assignment, which is available at: `https://www.cs.allegheny.edu/sites/amohan/resources/suggestions.pdf`

Learning Assignment

To do well on this assignment, you should also read

- **PH** - chapter 03: [3.1 - 3.4];
- **KR** - chapter 05: [5.1-.5.2];
- Class discussion notes and slides.

Assignment Details

In this lab assignment, you will work on automating the binary division process. Towards achieving this goal, you will also implement automation of binary addition and subtraction process. It is very natural to include the following lines in a high level language code such as C, C++, and Java:

```
int a = 12;
int b = 5;
int z = a/b;
print(z);
```

So the question here is what happens behind the scene, when **"/"** is being used in the high level language program? Let us try to address this question through this lab exercise. I strongly recommend taking this task seriously. Besides getting lab points, there is also a good chance that "Binary Division" might be included in the final exam. This lab work is aimed to prepare you to do well in the manual binary division process.

Part 1: Binary Addition)

Write a program called **add.c**, with a function called "addBinaryNos", that automate the binary addition process based on the rules outlined below:

1. 0 + 0 = 0; Carry = 0
2. 1 + 0 = 1; Carry = 0
3. 0 + 1 = 1; Carry = 0
4. 1 + 1 = 0; Carry = 1

It is very natural to include the following lines in a high level language code such as C, C++, and Java:

```
int a = 12;
int b = 5;
int z = a+b;
print(z);
```

So the question here is what happens behind the scene, when **"+"** is being used in the high level language program? Let us try to address this question through this part of the lab exercise.

Let us suppose that, the user provide two inputs. In contrast to the in-class practical activity, where we assumed the user inputs are binary, for this lab, we are assuming that the user inputs are provided by the user in its decimal form. So how do we convert the decimal provided by the user into its binary equivalent? Ah, does it strike that we had implemented this in the previous lab work? **Truth Table Generator**. So the code reuse from previous lab work is required. If the user typed in 23 as input then your truth table generator is supposed to generate an array (let us assume truth[]) with 32 rows. By accessing, truth[23], the binary form of decimal 23 would be generated.

At this point, the user input values are nicely converted into the binary form. After getting the binary equivalent, store it into the corresponding arrays X and Y. Note: Array X and Y should always be equal in array size, you could force this to happen in your code by adding leading 0's to the smaller array so that to match the size on both arrays.

I will give you some code as a sample, to implement binary addition. In this part of the code, we simply take two arrays (let us say r and s) and iterate through the array right to left. Then, the elements in both arrays are processed bit by bit and the output is stored in an array called sum. The rules are as follows:

- If the bit from array r = 0, If the bit from array s = 0, and If carry = 0; then update the corresponding bit in array sum = carry by setting carry = 0;
- If the bit from array r = 1, If the bit from array s = 0, and carry = 0; then update the corresponding bit in array sum = 1 by setting carry = 0;
- If the bit from array r = 1, If the bit from array s = 0, and carry = 1; then update the corresponding bit in array sum = 0 by setting carry = 1;
- If the bit from array r = 0, If the bit from array s = 1, and carry = 0; then update the corresponding bit in array sum = 1 by setting carry = 0;
- If the bit from array r = 0, If the bit from array s = 1, and carry = 1; then update the corresponding bit in array sum = 0 by setting carry = 1;
- If the bit from array r = 1, If the bit from array s = 1, and carry = 0; then update the corresponding bit in array sum = 0 by setting carry = 1;
- If the bit from array r = 1, If the bit from array s = 1, and carry = 1; then update the corresponding bit in array sum = 1 by setting carry = 1;

A typical implementation of the binary addition use case is as follows:

```
/* assuming equal sized array...*/
void addBinaryNos(int *larray, int *rarray, int lsize, int rsize, int *sum){
int carry = 0;
for (int i = lsize-1; i >= 0; i--){
if (larray[i] == 0 && rarray[i] == 0){
sum[i+1] = carry;
carry = 0;
}
else if (larray[i] == 0 && rarray[i] == 1){
if (carry == 0){
sum[i+1] = 1;
}
else{
sum[i+1] = 0;
carry = 1;
}
}
else if (larray[i] == 1 && rarray[i] == 0){
if (carry == 0){
sum[i+1] = 1;
}
else{
sum[i+1] = 0;
carry = 1;
}
}
else if (larray[i] == 1 && rarray[i] == 1){
if (carry == 0){
sum[i+1] = 0;
carry = 1;
}
else{
```

```
sum[i+1] = 1;
carry = 1;
}
}


}
sum[0] = carry;
}
```

You are required to either use this code directly or indirectly, within your add.c code to handle the requirement of binary addition. A critical part of this requirement is that you are **required to use pass by reference while calling the function to manipulate the values in the array**. If you have not used pass by reference, then there will be point deductions for not handling the requirements correctly. Additional details about the point deductions policy will be provided in the graded report sheet.

After completing the process of binary addition, the program above simply stores the binary form of the output in the array **sum**. In your code, you are required to display the decimal equivalent along with the binary equivalent in the console. In order to find the decimal equivalent, once again the use of the array (truth[]) from the truth table generator code is required. By iterating through the array with an index value **"i"**, if there is match in the truth[] array with that of the binary number, the **"i"** value during that iteration is simply equates to the decimal equivalent! So, the expected output while executing add.c is:

```
Enter the first no: 11
Enter the second no: 4
The result of binary addition is: 15 (1111)
```

Part 2: Binary Subtraction)

Write a program called **sub.c**, with a function called "subBinaryNos", that automate the binary subtraction process based on converting the given negative number into its 2's complement form. After converting the 2's complement form, binary addition of the two numbers is applied to complete the implementation. It is worth noting that, subtraction is just a variation of addition in binary.

It is very natural to include the following lines in a high level language code such as C, C++, and Java:

```
int a = 12;
int b = 5;
int z = a-b;
print(z);
```

So the question here is what happens behind the scene, when **"-"** is being used in the high level language program? How does **a - b** equate to a + b? Let us try to address this question through this part of the lab exercise.

Let us suppose that, the user provide two inputs. In contrast to the in-class practical activity, where we assumed the user inputs are binary, for this lab, we are assuming that the user inputs are provided by the user in its decimal form. So how do we convert the decimal provided by the user into its binary equivalent? Ah, does it strike that we had implemented this in the previous lab work? **Truth Table Generator**. So the code reuse from previous lab work is required. If the user typed in 23 as input then your truth table generator is supposed to generate an array (let us assume truth[]) with 32 rows. By accessing, truth[23], the binary form of decimal 23 would be generated.

At this point, the user input values are nicely converted into the binary form. After getting the binary equivalent, store it into the corresponding arrays X and Y. Note: Array X and Y should always be equal in array size, you could force this to happen in your code by adding leading 0's to the smaller array so that to match the size on both arrays.

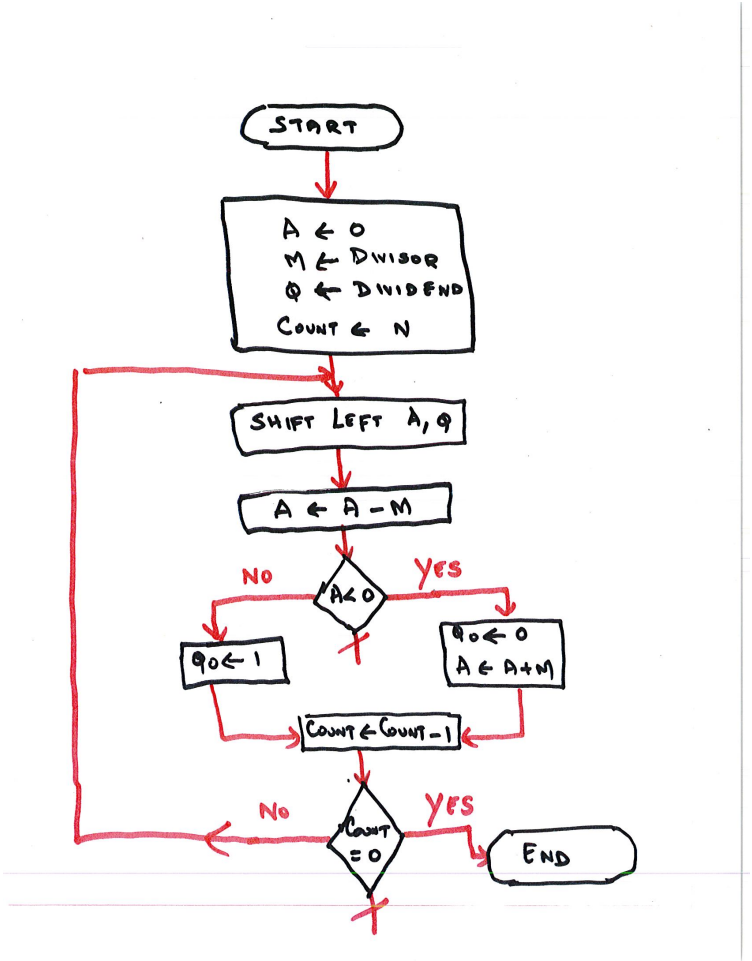
You are required to use your function from part 1 ”addBinaryNos”, within your sub.c code to handle the requirement of binary subtraction. You are recommended to copy and paste your code from the previous part into this part of the lab code. In addition, you are required to create a function called ”twosComplement”, which will create the 2’s complement form of any given binary number. A critical part of this requirement is that you are **required to use pass by reference while calling the function to manipulate the values in the array**. If you have not used pass by reference, then there will be point deductions for not handling the requirements correctly.

After completing the process of binary subtraction, the function is expected to store the binary form of the output in the array **diff**. In your code, you are required to display the decimal equivalent along with the binary equivalent in the console. In order to find the decimal equivalent, once again the use of the array (truth[]) from the truth table generator code is required. By iterating through the array with an index value **”i”**, if there is match in the truth[] array with that of the binary number, the **”i”** value during that iteration is simply equates to the decimal equivalent! So, the expected output while executing sub.c is:

```
Enter the first no: 11
Enter the second no: 4
The result of binary subtraction is: 7 (0111)
```

Part 3: Binary Divison)

Write a program called div.c, with a function called ”divBinaryNos”, that automate the binary division process based on the rules outlined in the flowchart below:



It is very natural to include the following lines in a high level language code such as C, C++, and Java:

```
int a = 12;
int b = 5;
int z = a/b;
print(z);
```

So the question here is what happens behind the scene, when **"/"** is being used in the high level language program? Let us try to address this question through this part of the lab exercise.

Let us suppose that, the user provide two inputs, namely the divisor and the dividend. In contrast to the in-class practical activity, where we assumed the user inputs are binary, for this lab, we are assuming that the user inputs are provided by the user in its decimal form. So how do we convert the decimal provided by the user into its binary equivalent? Ah, does it strike that we had implemented this in the previous lab work? **Truth Table Generator**. So the code reuse from previous lab work is required. If the user typed in 23 as input then your truth table generator is supposed to generate an array (let us assume truth[]) with 32 rows. By accessing, truth[23], the binary form of decimal 23 would be generated.
At this point, the user input values are nicely converted into the binary form. After getting the binary equivalent, store it into the corresponding arrays M and Q. Note: Array M always have an additional bit to represent the sign and to accommodate the usage of 2's complement form for binary subtraction in the algorithm.

There are three major stages in implementing this part of the lab:

1. Initialize
2. Left-Shift
3. Subtraction
4. Addition

During the initialization phase, an array A, that is of same size as M, is initialized to all 0's

```
A = {0, 0, 0, ...}
```

A critical part of this implementation is to develop the necessary functionality to implement the Left-Shift operation.

Let us suppose, we are given two arrays A and Q:

```
A = {0,0,0,0,0}
Q = {1,1,0,0}
```

After performing left shift operation, the new values in the arrays A and Q are as follows:

```
A = {0,0,0,0,1}
Q = {1,0,0,-1}
```

The value in the last bit of Q (Q_0) is temporarily set to -1. This value will change based on how the algorithm works. Refer to the flowchart shown at the start of part 03 description.

In your implementation, the different iterations need to be set based on the count value. The count value is simply the total number of bits in Q.

You are required to use your function from part 1 ”addBinaryNos” and from part 2 ”subBinaryNos” within your div.c code to handle the requirement of binary division. You are recommended to copy and paste your code from the previous part into this part of the lab code. In addition, you are required to create a function called ”leftShift”, which will shift the bits in the given arrays, as outlined in the process above.

A critical part of this requirement is that you are **required to use pass by reference while calling the function to manipulate the values in the array**. If you have not used pass by reference, then there will be point deductions for not handling the requirements correctly.

After completing the process of binary division, the function is expected to store the binary form of the output in the array **A** and array **Q**. In your code, you are required to display the decimal equivalent along with the binary equivalent in the console. In order to find the decimal equivalent, once again the use of the array (truth[]) from the truth table generator code is required. By iterating through the array with an index value **”i”**, if there is match in the truth[] array with that of the binary number, the **”i”** value during that iteration is simply equates to the decimal equivalent! So, the expected output while executing div.c is:

```
Enter the first no: 12
Enter the second no: 5
The result of binary division is: Quotient is 2 (0010); Remainder is 2 (0001)
Do you like to display the intermediate results?(y/n): y
```

```
Operation Type      A         Q         M         Count
--------------    ------    ------    -------    ---------
Init              00000      1100      00101       4
LeftShift         00001      100-1     00101       4
Subtract          11100      1000      00101       4
Add               00001      1000      00101       4
----
----
----
Subtract          11101      0010      00101       1
Add               00010      0010      00101       1
```

The solution in the above result, shows only a few steps from the entire intermediate steps. For the detailed manual solution to the problem above, please refer the division-solution.pdf file in lecture 09 folder. You are expected to match the formatting produced in the sample output shown above. If the user types ”n” for the question ”Do you like to display the intermediate results?”, then the intermediate results should not be displayed and the program should terminate with a greeting instead.

```
Enter the first no: 12
Enter the second no: 5
The result of binary division is: Quotient is 2 (0010); Remainder is 2 (0001)
Do you like to display the intermediate results?(y/n): n
Thank you for trying the binary division calculator. Bye!
```

Submission Details

1. Your lab assignment solution file must be uploaded to your repository by the due date and time.
2. Questions about the lab? Bring them to class on Tuesday morning!

Before you turn in this assignment, you must ensure that the course instructor has read access to your git repository that is named according to the convention `cs200F2018-<your user name>`.

Grading Rubric

1. If you complete Part 1 fully, as per the requirement outlined above, you will receive a total of 15 points.
2. If you complete Part 2 fully, as per the requirement outlined above, you will receive a total of 15 points.
3. If you complete Part 3 fully, as per the requirement outlined above, you will receive a total of 20 points.
4. Failure to upload the lab assignment code to your git repo, will lead you to receive ”0” points given for the lab submission.
5. There will be a partial credit (30% of the points allocated for the part) given if your code in (Part 1, 2, 3) compiles correctly but not able to produce the expected result. Note: there will be no partial credit awarded for the extra credit question.
6. There will be no partial credit awarded if your code doesn’t compile correctly. It is highly recommended to validate if the correct version of the code is being submitted before due date and make sure to follow the honor code policy described in the syllabus. If it is a late submission, then it is the student’s responsibility to let the professor know about it after the final submission in github. In this way, an updated version of student’s submission will be used for grading. If the student did not communicate about the late submission, then automatically, the most updated version before the submission deadline will be used for grading purposes. If the student had not submitted any code, then in this case, there is 0 points to the student automatically for the lab work.
7. If you needed any clarification on your lab grade, talk to the instructor. The lab grade may be changed if deemed appropriate.