CMPE-380: Applied Programming

Laboratory Exercise 10

Fixed-Point Representations (Q-Format)

Table of Contents

Pre-Lab – 10 pts 3

Floating Point Limitations 4

Qn Conversion 4

Interactive Examples – 30 pts 6

Qn Division 6

Qn Multiply 8

Qn Complex equations 9

Qn Performance 10

Assignment – 60 pts 11

Objective 11

Qn performance 11

Debugging and Timing the code 12

Makefile 12

Results and Analysis 14

Grading Criteria 14

Laboratory Grading Sheet 15

Fixed-Point Representations (Q-Format)

Pre-Lab – 10 pts

In this lab session, we will review Q format, an alternate data representation format between integers and floating point. Computers use a finite number of bits to represent an infinite range of numbers so by their very nature, some precision (data) will be lost. Modern high-end processors all include high-performance floating-point hardware, but most lower cost microcontrollers do not. Qn format numbers provide some fractional number support while only requiring integer hardware. Qn format numbers will work even on the lowest performance hardware.

Please review the QN format class slides and the provided **prelab/lab_a.c & exercise/lab_b.c** code and be prepared to answer how floating-point numbers are converted to/from Qn values. In the final step of this exercise you will append the output of your final **lab_a.c** code into your **prelab.txt** file

Floating Point Limitations

In this section you are going to show that traditional floating-point calculations have limitations and can lose precision (data).

1) Update the provided **lab_a.c** code section labeled "Prove floating point has limits"
2) Add the value "DBL_EPSILON" to the double precision floating point number 10.0 and clearly show that the mathematics either succeeded or failed.

   e.g: Proving floating point has limits by adding Epsilon

   Epsilon value '2.22045e-16' didn't add
3) Did the addition "work"? How did you prove that the addition worked, or not? What does the value "DBL_EPSILON" represent? Document your answers in **prelab.txt.**

Qn Conversion

In this section you are going to use the provided Qn conversion utilities to convert a floating-point value to the Qn format specified and then back to floating point to examine the binary and decimal representations of Qn numbers. You will also be able to determine if the conversion process introduces data loss. Use 16 bit precision for the binary printing function. All the output in this section should have the following style:

Started with **xxx.xx** converted to qn**x** = **xxx** decimal then back to **xxx.xx**

QN **xxx** decimal in binary is [**……**.]

1) Update the provided **lab_a.c** code section labeled "Examine the conversion utilities"
2) Convert **0.0** to **Qn0** and then convert back.
3) What is another name for a Qn0 number? Document your answers in **prelab.txt.**

4) Convert **12.25** to **Qn3** and then convert back.
5) Was there any data loss? Document your answers in **prelab.txt.**

6) Convert **12.0625** to **Qn3** and then convert back.
7) Was there any data loss? Document your answers in **prelab.txt.**

8) Convert **12.0625** to **Qn4** and then convert back.
9) Was there any data loss? Document your answers in **prelab.txt.**

Append the output of your final **lab_a.c** code into your prelab.txt file. Your TA will verify your **prelab.txt** file and conduct a brief oral quiz on the basic Qn format. You should be able to create a Qn number given some value and rules. You should be able to explain the basic issues behind Qn addition, subtraction multiplication and division. You should be able to explain how to convert a positive Qn number to a negative Qn number.

Interactive Examples – 30 pts

Each section in this code should be nicely formatted and clearly identify the test case. You will be documenting your comments in **exercise.txt**, be sure to clearly indicate what step you are commenting on. In the final step of this exercise you will append the output of your final **lab_b.c** code into your exercise.txt file. **Be sure to save backup copies of your exercise.txt file as you work.**

Qn Division

In this section you are going to repeatedly divide by 2 using normal floating-point division, integer division AND the Qn_DIVIDE() function until the resulting Qn value is zero. The purpose of this section is to demonstrate some of the dangers in coding simple mathematics functions and the dangers of integer operations. . All the output in this section should have the following style:

fnum	qnum1 (dec)	qnum2 (dec)	qnum1 (float)	qnum2 (float)
0.031250	8192	8192	0.031250	0.031250
0.015625	4096	4096	0.015625	0.015625

1) Update the provided **lab_b.c** code section labeled "Examine the division utilities"
2) Create two unique **Qn18** values from: **-.0625**
3) Write a while loop which will loop until one of the Qn values is zero.
4) The loop should divide the floating-point number by 2.0.
5) The loop should divide one of the Qn values by 2 using normal integer division.
6) The loop should divide the other Qn value by 2 using Qn_DIVIDE.
7) Print the results formatted as shown.
8) Did you have any trouble using the Qn_DIVIDE()? Why does integer division work? Document your answers in **exercise.txt.**

9) Repeat the previous example but now only print the absolute values of the floating-point value and Qn values.
10) Did you have any trouble printing absolute values? Document your answers in **exercise.txt**

Qn Multiply

In this section you are going to use the Qn_MULTIPLY() function. All the output in this section should have the following style:

> Multiplication test
>
> Product= xx.xxxx float qnx product= xxxx dec, converted back xx.xxxx float

1) Update the provided **lab_b.c** code section labeled "Examine the multiplication utilities"
2) Multiply the following two **Qn18** values: **64.125 and 0.755**
3) Print the results formatted as documented.
4) Was there any data loss? Document your answers in **exercise.txt**

Qn Complex equations

In this section you are going to use the Qn_MULTIPLY() function to implement "real" equations. All the output in this section should have the following style:

Multiplication test

Product= xx.xxxx float qnx product= xxxx dec, converted back xx.xxxx float

1) Update the provided **lab_b.c** code section labeled "Implement floating point scientific equations two different ways"
2) Implement the following **floating-point** decimal equation:

   F(x) = x**3 -.0001x**2 - 676X + .0676 Note: ** is exponentiation

3) Evaluate the equation using **1.0**
4) What is the resulting value? Document your answers in **exercise.txt**

5) Conduct the same test as above but using the following fractional equation instead of decimal equations:

   F(x) = x**3 -x**2/10000 - 676X + 169/2500

6) What is the resulting value? Did you have any implementation issues? Document your answers in **exercise.txt**

7) Update the provided **lab_b.c** code section labeled "Implement QN scientific equations" by adding a Qn evaluation of: **F(x) = x**3 - .0001x**2 - 676X + .0676**
8) Evaluate the equation using **1.0** as a **Qn18** number
9) What is the resulting value? Did you have any implementation issues? Is the result reasonably close to the floating-point implementation? Document your answers in **exercise.txt**

Qn Performance

In this section you are going to compare floating point addition and multiplication performance against Qn addition and multiplication performance. You will be running these calculations on high end processor, so the performance calculations are not indictive of lower end microcontrollers.

1) Update the provided **lab_b.c** code section labeled "Examine the performance"
2) Utilize the provided timing loops for your calculations.
3) Use the following floating-point values for your floating point and **Qn18** calculations:
   3.1415 & -674.9325
4) The first two timing loops verify floating point addition and Qn addition respectively
5) The last two-timing loops verify floating point multiplication and Qn multiplication respectively.
6) Clearly identify the number of clock ticks for each operation and clearly identify the faster operation for the group (addition and then multiplication).
7) Which format should be faster? Which format was faster on high end equipment? Document your answers in **exercise.txt**

8) Append the output from your **lab_b.c** to the end of your exercise.txt file and then show your results to your TA.

Assignment – 60 pts

Objective

Demonstrate the performance advantage of Qn integer calculations over floating point making use of the Bisection root finding method you implemented earlier. Please use the "ClassErrors.h", "rootfinding.h", and "Timers.h" from your previous works. You do NOT have to implement secant or newton methods.

Qn performance

A testing template program called **intTest.c** is included in this lab. You will rewrite your floating point bisection solver code using Qn format integer code and investigate the performance differences. Contact your professor if you don't have a functioning bisection implementation. Your implementation will use the "long" integer data type (64 bit on this machine). You will need to decide on the value of "n" to use, be sure to include your **analytic justification** in the analysis.txt file. You do NOT have to implement any command line parsing. An "iNAN" macro is provided to replace the "NAN" macro used in the floating point solution.

Rootfinding.h contains the #define QN 2, which is **wrong**. Modify this macro as you see fit. QN represents the fractional number of bits in the Qn format. The include file also includes the following Qn mathematical macros, use them in your solution. Your analysis must include a discussion on how you arrived at your Qn value. **Guessing** and trying different numbers **is not acceptable**, you need to identify an **analytic justification**.

FLOAT_TO_FIX(x)

FIX_TO_FLOAT(X)

Qn_DIVIDE(A,B)

MUL_FIX(A,B)

These macros utilize the value of "QN" in the include file.

The function you will be "finding the solution to" is **y = x**3 -.0001x**2 -676X +.0676**. The bounding guess and tolerance are: -25.0, 25.0 and 0.000001, respectively. The goal is to solve the equation using both your floating point bisection algorithm and your integer Qn format bisection algorithms, comparing the performance and accuracy. Recode your floating point bisection algorithm in the ibisection() function of intTest.c.

Modify the main() function in intTest.c to call both bisection algorithms and compare the performance and errors. Use your initial Qn guess and then try other larger and smaller values of Qn, provide a summary in your analysis.txt file. Why does changing the Qn value change the error? Try a Qn of 32, what happens? Why?

A true software floating point bisection implementation called **sw-float-Test** has been included in your lab. Run it and compare its' performance to your QN and HW bisection implantations. How much slower is the true SW floating point implementation than the QN implementation?

All program error messages should be printed to **stderr** (not to **stdout**). Any Verbose output is not an error and should be printed to **stdout**.

Debugging and Timing the code

Debug your module functions and make sure that you have an efficient implementation. Try different optimization levels, (**gcc** option **-O0 .. -O3**) and select the one that gives the "best" result. Record timing of your algorithms and save in your results file.

Makefile

Write a make file with the following targets: all, test, mem, clean, help. Your makefile should utilize a macro variable to enable verbose mode in your code and timing in your code. The default is timing enabled and verbose mode disabled.

"all"	- should make **intTest** with timing enabled and **intTest_mem** with timing disabled
"test"	- should run the application with timing enabled, output to **out.txt**
"mem"	- Run valgrind with timing disabled.

help, clean - should do the normal things

See results.txt for output formats.

Results and Analysis

Write in the file **analysis.txt** an explanation of the results obtained. Include your analysis from the Qn investigation. Submit a tarball with all C program(s) together with your analysis and Makefiles.

Grading Criteria

1. (35 points) Correct program(s)/algorithms.
2. (2 points) Memory leaks
3. (10 points) Makefile
4. (13 points) Analysis

Note:

1. Please note that you will use the Bisection implementation you coded in a previous lab!

Student Name: _____

Laboratory Grading Sheet

Lab 10 - Fixed-Point Representations – Q-format

Component	Point Value	Points Earned	Comments and Signatures
Pre-Lab: Floating Point Limitations	2		
Pre-Lab: Qn Conversion	4		
Pre-Lab: Quiz	4		
Interactive Exercises: Qn Division	7		
Interactive Exercises: Qn Multiply	3		
Interactive Exercises: Complex Equations	10		
Interactive Exercises: Qn Performance	10		
Total	40		

You must turn this signed sheet in at the end of lab to receive credit!