Lab 2

Lab 2 assignment is due on Friday, February 28, by 9pm in your svn repository.

Binary computations

> *On June 4, 1996 an unmanned Ariane 5 rocket launched by the European Space Agency exploded just forty seconds after lift-off. The rocket was on its first voyage, after a decade of development costing $7 billion. The destroyed rocket and its cargo were valued at $500 million.*

> *On February 25, 1991, during the Gulf War, an American Patriot Missile battery in Dharan, Saudi Arabia, failed to intercept an incoming Iraqi Scud missile. The Scud struck an American Army barracks and killed 28 soliders.*

Source: http://www.ima.umn.edu/~arnold/455.f96/disasters.html

What do both events have in common? As it turns out, some basic errors in binary computation caused both of these tragedies. In this Lab, we'll explore the limitations of binary computation and see what went wrong in both of these cases.

More command line tricks

For this assignment, we'll be using a program written for it called *bincalc* to perform binary computations. Before we can start using it, we need to learn a few more command line tricks.

Program paths

Recall from Lab 1 that Unix terminal commands have the following syntax:

```
program [arg1 ...]
```

Recall that many programs take files and directories as arguments, which are written as *paths*. *Relative paths* describe how to get to the file from the current working directory, while *absolute paths* describe how to find the file from the *root directory*. What you may not be aware of is that program can also be a path. In fact, all programs are files that exist on the filesystem just like documents. You can see where a program is located with the "which" program:

```
which program1 ...
```

Try it with the "echo" program. When I run it, I get "/bin/echo". "bin" is short for "binary", since, as you'll soon discover in this class, programs are encoded in binary machine code. On Unix systems, a directory that contains executable (runnable) binaries is called "bin" by convention.

Remember that I said program can be a path? You can prove it:

```
/bin/echo 'Programs are paths too!'
```

Tilde expansion

The *shell* replaces paths starting with a "~" references your home directory. For example, to go to your ece120 directory, you can just use:

```
cd ~/ece120sp20
```

A "~" followed by another user's name represents their home directory. We can use the tilde to access the home directory of your classmates or teachers:

```
ls ~student
```

Introducing bincalc

Go you your home directory, make a new folder called bin:

```
cd
mkdir bin
```

Download bincalc program from here and save it in your newly created bin subfolder in your home directory.

> ⚠ bincalc is compiled to run on a 32-bit OS in your class Virtual Machine. If you are attempting to run it somewhere else, it will not work.

Change bincalc's permissions and run it:

```
cd ~/bin
chmod a+x bincalc
~/bin/bincalc
```

This is actually incorrect usage of the program, so it will print out some usage information. Read this information carefully. **You'll see that adding a "-v" flag will make the program verbose, and that you must always give it a mode to indicate what binary format to use (like s8 for signed 8-bit).**

Run the program in signed 16-bit mode. If you succeeded, you should see a different prompt that looks something like this:

```
>
```

This means the *shell* has launched a program that has taken control of the terminal. Once the program terminates, you'll be back at the shell's prompt again. Enter "exit" to terminate "bincalc" and see the shell prompt again.

Now launch bincalc again and put 2 and 2 together:

```
> 2 + 2
4 (x0004)
```

Notice that the binary encoding for the result is also printed in hex for your convenience. The "+" is called an *operator* and 2s are called *operands*. In this calculator, an operator performs a certain computation with its operands (addition, in this case) and produces a result.

The observant reader may ask: if we can somehow magically run "ls" without specifying the path, why can't we do the same with "bincalc"? The answer is that we can do it if the path of "bincalc" is set in the PATH environment variable (see Modifying your PATH section for more details). Try typing "bincalc" in the command line and check whether it works.

Lab 2 assignment

For this assignment, you'll be writing your answers to the questions below to a text file in your repository. First, you'll want to checkout the latest version of your repository with the subversion update command:

```
cd ~/ece120sp20
svn update
```

> ⓘ **If you encounter error with "svn update"**
>
> Remember to go to the ece120sp20 folder you worked with in Lab 1, not the one located in ~/bin. "pwd" command is useful here.

If the update went well, you should see a new lab2 directory. lab2/lab2.txt contains the questions in this exercise. Use your text editor to put your responses in that file. **Don't forget to commit to submit your work, and check the repository with 'svn status' to verify that your submission went through.** (Review Lab 1 for how to commit and check your submission.)

> ⚠ To receive a full credit for this lab, you only need to answer questions marked with ⭐.

Signed integer computation

In signed 16-bit mode, run the following computations with *bincalc*:

1. 100 - 199
2. 1000 + 243
3. 21000 + 22000

4. 32767 + 1
5. -32767 - 1

Answer the following questions in lab2.txt:

1. What is the largest value representable in 16-bit signed format? Smallest? ⭐
2. What is the result of the third computation? Why? ⭐
3. Why does the fourth computation overflow, but not the fifth? ⭐
4. If you were to start at 0, and increment repeatedly (add 1), what pattern would you see (in signed mode)?

Unsigned integer computation

In unsigned 16-bit mode, run the following computations:

1. 100 - 199
2. 21000 + 22000
3. 32767 + 1

Answer the following questions in lab2.txt:

1. What is the largest value representable in 16-bit unsigned format? Smallest? ⭐
2. What result do you get from the first computation? Why? ⭐
3. Why doesn't the second or third computation overflow?
4. If you were to start at 0, and increment repeatedly (add 1), what pattern would you see (in unsigned mode)?
5. What are the advantages and disadvantages of unsigned formats (compared to signed formats)?

Multiplication, division, modulus

In this calculator's language, "*" means multiply, "/" means divide, and "%" means modulus.

In signed 16-bit mode, run the following computations:

1. 22 * 33
2. 100 * 500
3. -100 * 500
4. 12 / 3
5. 11 / 3
6. 10 / 3
7. 9 / 3
8. 12 % 3
9. 11 % 3
10. 10 % 3
11. 9 % 3

Answer the following questions:

1. What result do you get in the second and third computations? Why? ⭐
2. What results do you get from computations 4-7? Why? (Hint: can signed 16-bit encoding represent fractions?) ⭐
3. Does the result from computations 4-7 get rounded to the nearest integer? If not, what actually happens? Where might this behavior be useful? (Hint: what if you wanted to divide 11 candies between 3 people?)
4. What does the modulus operator do for positive integers?
5. What happens when you divide by 0? Modulus with 0? What happens to the *binary-calculator* program? Why might this be a good thing? (Hint: remember the lab introduction?)
6. Excluding division by 0, characterize the behavior of the modulus operation for positive and negative divisors and dividends (for a total of 2x2 = 4 combinations).

Floating point

Run the following computations in single-precision (32-bit) floating-point mode:

1. 1000 + 1
2. 11 / 3
3. 100 - 0.25
4. 100 - 0.3
5. -100 + 0.3
6. 9000 + 0.0001 - 9005
7. 9000 - 9005 + 0.0001
8. 9000 + 0.01
9. 9000 * 0.01

Answer the following questions in lab2.txt:

1. Why is there an error in the fourth computation, but not the third? (Hint: how do you encode 0.25 and 0.3 in floating point?) ⭐
2. How does the result of the fifth computation compare to the fourth? Explain. (Hint: look at the hex representations of the results. How does the floating point format handle negative numbers?) ⭐
3. Mathematically, would you expect the same results in computations 6 and 7? Do you observe this result experimentally? Explain. (Hint: try stepping through each computation) ⭐
4. What happens if you try computation 6 in double-precision (64-bit) floating-point mode? Why?

5. Why is there noticeable error in computation 8, but not 9? (Hint: think of multiplying floating point numbers like multiplying two numbers in scientific notation, how do you do it?)
6. The root cause of the Ariane 5 rocket failure was isolated to the conversion of a floating point number, which stored the horizontal component of the rocket's velocity, to a 16-bit signed integer. What is the most likely cause of the failure? (Hint: this wasn't some small rounding error, the computed velocity was way off, causing the system to go haywire)

Tip: It might be useful to use the IEEE-754 Floating-point Conversion tool to better understand the representation of floating-point numbers.

Additional problems

Completion of these additional problems **is not required for lab2 credit.** However, problems similar to these may show up in a future exam.

Bitwise logic

In this calculator's language, a "&" is a bitwise logical AND, a "|" is a bitwise logical OR, a "^" is a bitwise logical XOR, and a "~" is a bitwise logical NOT. In the absence of parenthesis, bit shifting is always done before AND which in turn is always done before OR, in the same way that multiplication is always done before addition.

In 16-bit unsigned mode, run the following computations:

1. x01fa & 1
2. x3030 & 1
3. x2911 & 1
4. x0001 | x0080 | x0040
5. (x0001 | x0080 | x0040) & x0080
6. (x0001 | x0040) & x0080
7. x45ff & ~1
8. x1234 ^ x5678
9. x1234 & ~x5678 | ~x1234 & x5678

Answer the following questions:

1. What are computations 1-3 doing? (what does a function that computes x & 1 do?)
2. What are computations 4-6 doing? (hint: suppose each of the 16 bits bit represents a light switch that is either on or off)
3. Continuing with the light switch (or similar) analogy, what is computation 7 doing?
4. Why does computation 5 produce a nonzero result, but computation 6 produces zero?
5. What is computation 7 doing? (what does a function that computes x & ~1 do?)

Bit shifting

In this calculator's language, "<<" means shift left and ">>" means shift right. For example, *a* << *b* shifts the bits in *a* left *b* places.

In signed 16-bit mode, run the following computations:

1. 4 << 0
2. 4 << 1
3. 4 << 2
4. 4 << 3
5. 4 << 4
6. 9 << 14
7. 1 << 15
8. 10 >> 0
9. 10 >> 1
10. 10 >> 2
11. 10 >> 3
12. 10 >> 4

Answer the following questions:

1. Looking at the results from computations 1-5, do you see a pattern? What's going on? What does the left-shift operation effectively do?
2. Given this definition of "<<", what would be the result from computation #6? Whats happening?
3. Why is the result from computation #7 negative?
4. Looking at the results from computations 8-11, what does the right-shift operation effectively do? (Hint: Try doing integer divisions with the same divisor)
5. Thinking of the right-shift operation as a shift operation, there's a simple explanation for why the result of computation 12 is 0. What is it? (Hint: how does a computer encode decimal 10 in signed 16-bit?)

In unsigned 16-bit mode, run the following computations:

1. x01fa >> 4 & 1
2. x3040 >> 4 & 1
3. x2911 >> 4 & 1

Answer the following questions:

1. What are computations 1-3 doing? (hint: suppose each of the 16 bits bit represents a light switch that is either on or off)

Modifying your PATH

The search path is an *environment variable* that tells the operating system where to look for executables when you try to run a program by only its name (no slashes in program). *environment variables* are modifiable parameters associated with a *process*. We can see the contents of environment variables by telling the *shell* to expand them with the "$" symbol:

```
echo ${PATH}
```

As you may have guessed, $PATH is the *environment variable* that provides the search path; it's actually multiple paths separated by colons. If we wanted to add to it, we could do something like this:

```
PATH=${PATH}:~/bin
```

Now we can run "bincalc" by name. Try it. This environment won't be *inherited* by processes that are launched, however, without using an additional shell *built in* (program built into the shell) called "export":

```
export PATH=${PATH}:~/bin
```

Now all programs that we run in this shell will also be able to run binaries in ~/bin by name. This change is still ephemeral, however, because as soon as we close the terminal the *shell* is terminated and its $PATH will be reset the next time we run it. To modify the $PATH automatically on every startup of the *shell*, you'll need to modify your login script, which is probably ~/.bashrc. Scripts are simply a list of commands for the shell to run, one on each line. Just put the export command in your login script, and you should be able to run "bincalc" even after restarting your *shell*.