Binary and C Intro Assignment (Evaluation)

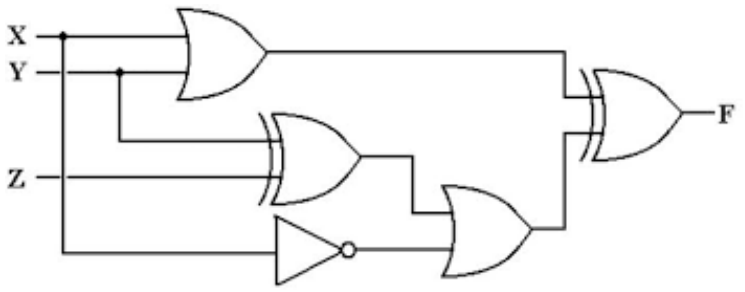
CSCI 389: Computer Systems

Fall 2022

This assignment is designed to test your understanding of binary and early c. Feel free to collaborate wih others and use resources, but all code and writeups must be your own. For submitting your code, you can convert the files to pdf (through screenshots or other methods) or provide a link to a repository or other codebase.

Due Date: Friday, September 23rd at 12:00 pm.

1. (12 points) **Converting Bases.** Convert the following numbers to the specified base.
   (a) (2 points) Convert 93_{10} to binary.
   (b) (2 points) Convert 215_{10} to hexadecimal.
   (c) (2 points) Convert 11000101_2 to decimal.
   (d) (2 points) Convert 1011010_2 to hexadecimal.
   (e) (2 points) Convert $C14B_{16}$ to decimal.
   (f) (2 points) Convert $5A6D_{16}$ to binary.
2. (4 points) **Binary Addition.** Show how to add 10111001_2 and 00111100_2 using binary arithmetic.
3. (4 points) **Binary Multiplication.** Show how to multiply 110100 and 10110 using binary arithmetic.
4. (4 points) **Circuits.** Create the truth table for the following circuit:



5. (16 points) **Galaxy Explorers (C).**

   Congratulations! you are now the big-shot CTO of Masstronaut, Inc., a hot new startup developing the next massive multiplayer space exploration game, "Galaxy Explorers".

   The current prototype is implemented in Python in the file `update_locations.py` available on Moodle, but this prototype turns out to scale poorly to many players. After some profiling, you realize that one of the slowest pieces of code in your game server is the one that updates the position of all of the objects in 3D space, every turn: spaceships, planets, asteroids, space junk, astronauts taking a space walk, you name it.

A 10
B 11
C 12
D 13

256	16	1
16^2	16^1	16^0

	1	0	1	1	1	0	1
128	64	32	16	8	4	2	1
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

1. a) $93_{10} \rightarrow$ binary = 1011101

b) $215_{10} \rightarrow$ hex = D7

c) $11000101_2 \rightarrow$ decimal = 197

d) $1011010_2 \rightarrow$ hex = 5A

e) $C14B_{16} \rightarrow$ dec = 49483

f) $5A6D_{16} \rightarrow$ binary = 101101001101101

2.

```
   111
  10111001
+ 00111100
----------
  11110101
```

3.

```
        110101
 x       10110
--------------
      1 000000
       1101010
      11010100
     000000000
    1101010000
--------------
   10010001110
```

4.

X	Y	Z	F
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

(a) (5 points) **Benchmarking.** You'll start with a micro-benchmark in Python that encapsulates the code that updates locations (see `update_locations.py`). This benchmark takes two parameters: the number of objects in space, and how many iterations (game turns) to simulate. It updates all objects every turn, and measures the mean time to update a single coordinate. Your first task is to understand the scaling properties of this code.

Run this code for multiple object counts (say from 28 to 224 in powers of two). For a small number of objects, you'll need to run for many iterations (turns) to get a statistically stable measurement. For larger objects, many turns would take too long, and may not be necessary. But you'll probably still see some variation from run to run. Once you decide on good choices for the number of iterations, run the benchmark for all these sizes and collect the results. Note that the checksum printed should remain the same between consecutive runs with the same arguments (as a sanity check).

Plot a line graph of mean update time per coordinate as a function of total number of objects (first parameter). Explain what you chose as your testing parameters and how you chose them. If you're using a script or modified python file, submit that on gradescope. There is no single correct answer here, I just want to see your reasoning.

(b) (8 points) **Port to C.** Port this benchmark to C, as closely as possible to the Python version. You may use any standard library functions. Repeat the above measurements and create a new graph that adds the C line to the Python line.

You may find that you can run many more iterations now, and perhaps larger object counts. Explain any parameter changes you make and demonstrate how it's better (or worse) than before.

To time a function in C, you can use `clock_gettime` (with the `CLOCK_MONOTONIC` argument). Since you're using a different random number generator now, you'll probably have different initial values and therefore a different checksum than the Python version. That's fine, as long as you're getting the same checksum from run to run.

(c) (3 points) **Comparing Types.** Compare your C implementation using different C types: float, double, int8_t, int16_t, int32_t, int64_t for the coordinate and velocity types in terms of performance. Report your findings.

Note: the checksum will probably not be computed correctly for the integral types due to precision loss. That's OK, though: we're only interested in the performance trade-offs.