Problem Set 3 – Due Tuesday, October 12, at 5pm

1. The standard way to represent a number in base-r (also called radix-r) (with $\mathsf{r} \geq 2$ an integer) is as sequence of one or more *digits* (symbols), each digit c representing a value $v(c) \in \{0, 1, \ldots, \mathsf{r}-1\}$. The sequence of digits $a_n a_{n-1} \cdots a_2 a_1 a_0$ (subscripted by r if one wants to explicitly name the base) represents the number $v(a_n)\mathsf{r}^n + v(a_{n-1})\mathsf{r}^{n-1} + \cdots + v(a_2)\mathsf{r}^2 + v(a_1)\mathsf{r} + v(a_0)$.

   (a) Represent numbers 187_{10} and 119_{10} as binary (base-2) numbers. Then add up the binary numbers using the base-2 analog of grade-school addition. Finally, convert the sum back to decimal.

   (b) Represent numbers 187_{10} and 119_{10} in base-3. Then add up the base-3 representations using the base-3 analog of grade-school addition. Finally, convert the sum back to decimal.

   (b) The C/C++ programming language allows logical operators to be applied to pair of integers, applying the named operator bitwise (that is, to corresponding bit positions in the binary representations of the two numbers). What integer will result from taking the AND of integers 187 and 119 (written `187 & 119`)? From taking their OR (written `187 | 119`)?

2. *If you know Python:* We all know that $P \vee Q \equiv Q \vee P$. Still, write the simplest Python program you can where the line

   ```
   print(P() or Q())
   ```

   causes the program to print `False`, while replacing it with

   ```
   print(Q() or P())
   ```

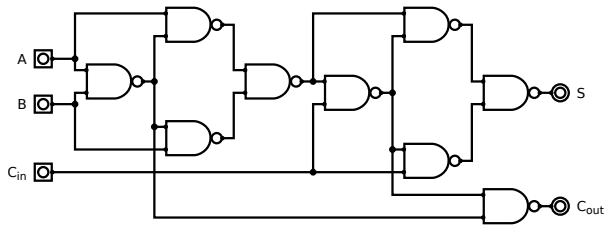
   causes it to print *True*. Unlike the example we gave in class, your program should never crash.

   If you don't know Python but know some other programming language, you can use that instead (adjusting the syntax as needed). If you don't know any programming language, you may say so and skip this problem.

   Try your program out. You can use an online python interpreter, if you like, at

   `https://www.onlinegdb.com/online_python_interpreter`

3. **(a)** In class we showed that the necessary functionality to add bits a, b, and c to generate a sum s and carry-out of c' is: $s = s(a, b, c) = a \oplus b \oplus c$ and $c' = c'(a, b, c) = ab \vee bc \vee ac$. Consider the following circuit "WFA":



   Does WFA compute the desired functions s and c'? (Treat labels A, B, $\mathsf{C_{in}}$, S, and $\mathsf{C_{out}}$ as synonyms for a, b, c, s, and c'). Explain how you made your determination.

   (b) Suppose we construct a 64-bit adder in the manner described in class using the circuit above. It adds two 64-bit values to generate a 65-bit sum. How many gates will your circuit employ, and what will be its *depth*? (The depth is the length of a longest path from an input wire to an output wire, measured in terms of the number of gates traversed.)

4. Chess is played on an 8×8 board. A *knight* placed on one square can move to any unoccupied square that is at a distance of two squares horizontally and one square vertically, or else two squares vertically and one square horizontally. The complete move therefore looks like the letter L (in some orientation). A knight cannot move off the board. Unlike other chess pieces, the knight can "jump over" other pieces in going to its destination.

   Consider a chess board on which we place any number $m \in \{0, 1, \ldots, 64\}$ of knights, at most one knight on each square. Call the configuration of knights *valid* if no knight can move to a square occupied by another knight.

   Carefully specify a Boolean formula ϕ over 64 Boolean variables $\mathcal{X}$ where the number of truth assignments to ϕ is exactly the number of valid knight configurations.

5. Write a BNF (Backus Normal Form) definition for a *number* (regarded as no more than a sequence of characters), the number written either in fixed point or scientific notation. Numbers should include values like

   ```
   0   314159   -27182   3.14159   -2.7182   0.0000000000000000000000013   6.0221515E23
   6.023e23   6.626068e-34   6.67300E-11   -1.234e-10   .29   -.999   100E100
   ```

   You can decide edge cases as you see fit—for example, whether or not `00` or `-000` should count as numbers. I suggest to do what's easiest.