****

EE/CS 119A Homework 1

Edward Speer

October 20, 2024

****

1. **Problem 1**: *7-Segment Decimal Counter*

Begin by creating the truth table for the 7-segment display. In this table, x_1 will represent a signal currently stored in the DFFs, and x_2 will represent desired next signal to be stored in the DFFs.

a_1	b_1	c_1	d_1	e_1	f_1	g_1	a_2	b_2	c_2	d_2	e_2	f_2	g_2
1	1	1	1	1	1	0	0	1	1	0	0	0	0
0	1	1	0	0	0	0	1	1	0	1	1	0	1
1	1	0	1	1	0	1	1	1	1	1	0	0	1
1	1	1	1	0	0	1	0	1	1	0	0	1	1
0	1	1	0	0	1	1	1	0	1	1	0	1	1
1	0	1	1	0	1	1	0	0	1	1	1	1	1
0	0	1	1	1	1	1	1	1	1	0	0	0	0
1	1	1	1	1	1	1	1	1	1	0	0	1	1
1	1	1	0	0	1	1	1	1	1	1	1	1	0

I then leveraged the Quine-McCluskey software that I wrote for HW2 (attached) to write a script (also attached) that would generate the prime implicants table for each of the segments in the display. From the prime implicants table, the equations for each segment may be derived. To generate the equation for each signal, I combined the results from the prime implicants chart with the reset mechanism. For reset, I take the OR of all the segments that are on to display the number 0 with the RESET signal, and the AND of the only segment that's off in displaying 0 with the negation of the RESET signal. This way, when reset is high, the display will be forced to show 0. Call the reset signal r.

For a:

```
Signal: a
=============================================
| .....00 | x ||   ||   ||   || x ||   ||   |
---------------------------------------------
| ....0.0 | x ||   ||   ||   || x ||   ||   |
---------------------------------------------
| ....1.1 |   || x ||   || x ||   || x ||   |
---------------------------------------------
| ....10. |   || x ||   ||   ||   ||   ||   |
---------------------------------------------
| ...0... | x ||   || x ||   || x ||   || x |
---------------------------------------------
| ..0.... |   || x ||   ||   ||   ||   ||   |
---------------------------------------------
| .0..1.. |   ||   ||   || x ||   ||   ||   |
---------------------------------------------
| .1...11 |   ||   || x ||   ||   || x || x |
---------------------------------------------
| .1..01. |   ||   || x ||   ||   ||   || x |
---------------------------------------------
| 0...... | x ||   || x || x ||   ||   ||   |
---------------------------------------------
=============================================
```

From the prime implicants table, rows 3 and 5 form a minimal covering set for the minterms so that $\boxed{a = eg + d' + r}$

For b:

```
Signal: b
===================================================
| ......0 | x || x ||   ||   ||   || x ||   ||   |
---------------------------------------------------
| .....0. |   || x || x || x ||   || x ||   ||   |
---------------------------------------------------
| ....1.. | x ||   || x ||   || x ||   || x ||   |
---------------------------------------------------
| ..0.... |   ||   || x ||   ||   ||   ||   ||   |
---------------------------------------------------
| .1.1... | x ||   || x || x ||   ||   || x ||   |
---------------------------------------------------
| 0..1... |   ||   ||   ||   || x ||   ||   ||   |
---------------------------------------------------
| 00..... |   ||   ||   ||   || x ||   ||   ||   |
---------------------------------------------------
| 1..0... |   ||   ||   ||   ||   || x ||   || x |
---------------------------------------------------
| 11..... | x ||   || x || x ||   || x || x || x |
---------------------------------------------------
===================================================
```

There are many possible covering sets for b, but one using the fewest total signals would be rows 2, 3, and 9. This yields $\boxed{b = ab + e + f' + r}$

For c:

```
Signal: c
========================================================
| ......1 |   || x || x || x || x || x ||   || x || x |
--------------------------------------------------------
| .....1. | x ||   ||   || x || x || x ||   || x || x |
--------------------------------------------------------
| ....1.. | x || x ||   ||   ||   || x ||   || x ||   |
--------------------------------------------------------
| ...1... | x || x || x ||   || x || x ||   || x ||   |
--------------------------------------------------------
| ..0.... |   || x ||   ||   ||   ||   ||   ||   ||   |
--------------------------------------------------------
| .0..... |   ||   ||   ||   || x || x ||   ||   ||   |
--------------------------------------------------------
| 1...... | x || x || x ||   || x ||   || x || x || x |
--------------------------------------------------------
========================================================
```

c has a clear minimal covering set from rows 1 and 7, yielding $\boxed{c = a + g + r}$

for d:

```
Signal: d
=========================================
| .....00 | x ||   ||   ||   || x ||   |
-----------------------------------------
| ....0.0 | x ||   ||   ||   || x ||   |
-----------------------------------------
| ....01. |   ||   || x || x ||   || x |
-----------------------------------------
| ....10. |   || x ||   ||   ||   ||   |
-----------------------------------------
| ...0... | x ||   || x ||   || x || x |
-----------------------------------------
| ..0.... |   || x ||   ||   ||   ||   |
-----------------------------------------
| .0..0.. |   ||   ||   || x ||   ||   |
-----------------------------------------
| 0.....0 | x ||   ||   ||   ||   ||   |
-----------------------------------------
| 0....0. | x ||   ||   ||   ||   ||   |
-----------------------------------------
| 0...0.. | x ||   || x ||   ||   ||   |
-----------------------------------------
| 01..... | x ||   || x ||   ||   ||   |
-----------------------------------------
| 10..... |   ||   ||   || x ||   ||   |
-----------------------------------------
=========================================
```

d has many possible covering sets, but the one using the fewest total signals is rows 5, 6, and 12. This yields $\boxed{d = ab' + c' + d' + r}$

for e:

```
Signal: e
===============================
| .....00 | x ||   || x ||   |
-------------------------------
| ....0.0 | x ||   || x ||   |
-------------------------------
| ...0..0 | x ||   || x ||   |
-------------------------------
| ...0.0. | x ||   || x ||   |
-------------------------------
| ...101. |   || x ||   ||   |
-------------------------------
| .0..0.. |   || x ||   ||   |
-------------------------------
| 0.....0 | x ||   ||   ||   |
-------------------------------
| 0....0. | x ||   ||   ||   |
-------------------------------
| 1...01. |   || x ||   || x |
-------------------------------
| 1..0... |   ||   || x || x |
-------------------------------
| 10..... |   || x ||   ||   |
-------------------------------
===============================
```

e has multiple possible covering sets, but I chose the one using rows 1 and 9. This yields $\boxed{e = f'g' + ae'f + r}$

for f:

```
Signal: f
========================================
| ....0.1 | x || x || x ||   ||   || x |
----------------------------------------
| ....01. |   || x || x ||   ||   || x |
----------------------------------------
| ...0..1 |   || x ||   ||   ||   || x |
----------------------------------------
| ...0.1. |   || x ||   ||   ||   || x |
----------------------------------------
| ...10.. | x ||   || x ||   ||   ||   |
----------------------------------------
| ..1..01 | x ||   ||   ||   ||   ||   |
----------------------------------------
| ..11.0. | x ||   ||   ||   ||   ||   |
----------------------------------------
| .0..0.. |   ||   || x ||   ||   ||   |
----------------------------------------
| .1...11 |   || x ||   ||   || x || x |
----------------------------------------
| .11...1 | x || x ||   ||   || x || x |
----------------------------------------
| 01....1 |   || x ||   ||   ||   ||   |
----------------------------------------
| 01...1. |   || x ||   ||   ||   ||   |
----------------------------------------
| 1....00 |   ||   ||   || x ||   ||   |
----------------------------------------
| 1....11 |   ||   || x ||   || x || x |
----------------------------------------
| 1...0.. | x ||   || x || x ||   || x |
----------------------------------------
| 1..0... |   ||   ||   || x ||   || x |
----------------------------------------
| 1.1...1 | x ||   || x ||   || x || x |
----------------------------------------
| 1.1..0. | x ||   ||   || x ||   ||   |
----------------------------------------
| 10..... |   ||   || x ||   ||   ||   |
----------------------------------------
========================================
```

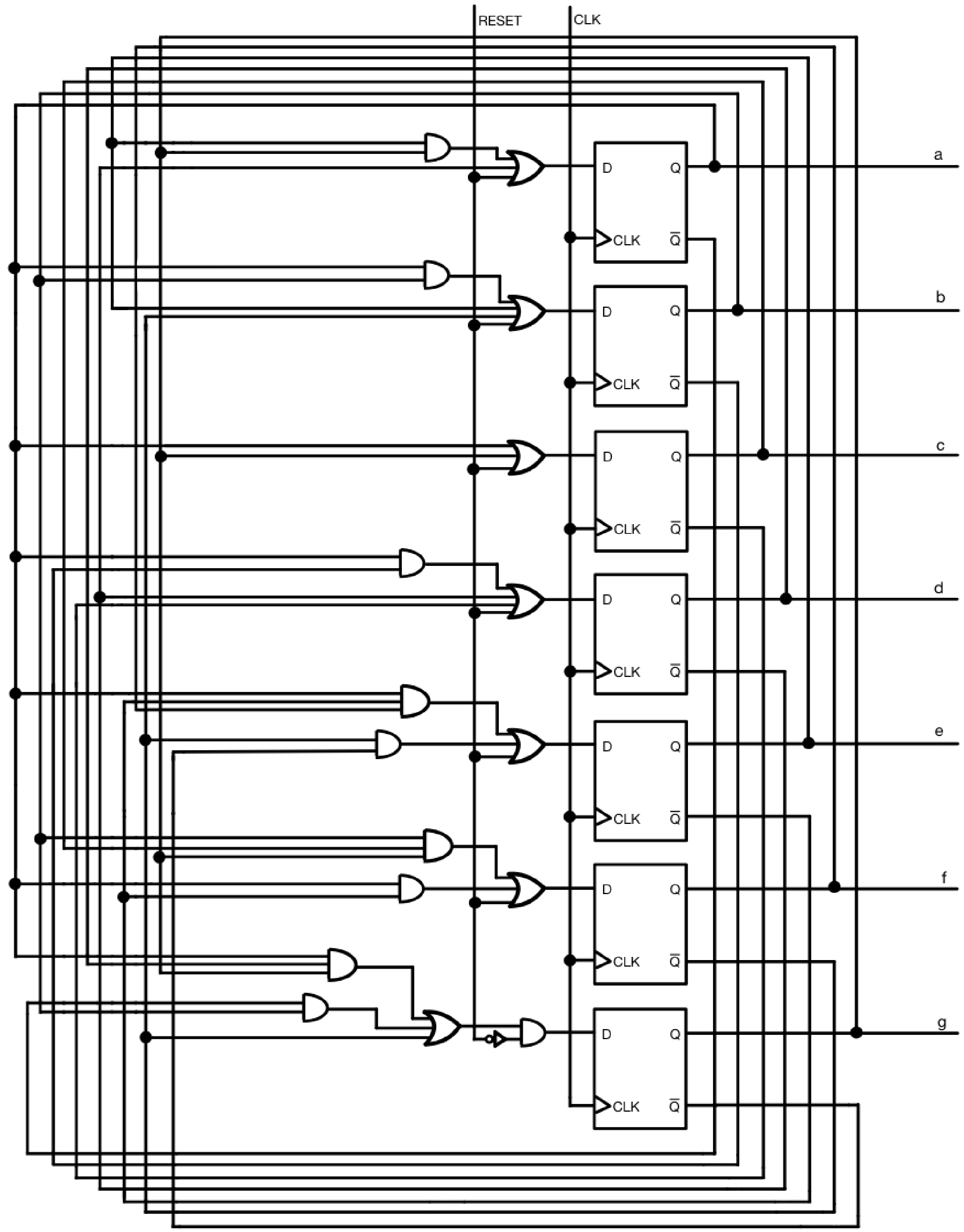
f has several possible covering sets, but the one using the least total signals is rows 15 and 9. This yields $\boxed{f = ae' + bcg + r}$

for g:

```
Signal: g
==============================================
| .....0. | x || x || x ||   ||   || x ||   |
----------------------------------------------
| ....0.0 | x ||   ||   ||   ||   || x ||   |
----------------------------------------------
| ...0..0 | x ||   ||   ||   ||   || x ||   |
----------------------------------------------
| ...10.. |   ||   || x ||   || x ||   ||   |
----------------------------------------------
| ..0.... |   || x ||   ||   ||   ||   ||   |
----------------------------------------------
| .0..0.. |   ||   ||   ||   || x ||   ||   |
----------------------------------------------
| .1..1.1 |   || x ||   ||   ||   ||   || x |
----------------------------------------------
| .1.1..1 |   || x || x ||   ||   ||   || x |
----------------------------------------------
| 0.....0 | x ||   ||   ||   ||   ||   ||   |
----------------------------------------------
| 0...0.. | x ||   ||   || x ||   ||   ||   |
----------------------------------------------
| 0..0... | x ||   ||   || x ||   ||   ||   |
----------------------------------------------
| 01..... | x ||   ||   || x ||   ||   ||   |
----------------------------------------------
| 1...1.1 |   || x ||   ||   ||   ||   || x |
----------------------------------------------
| 1..1..1 |   || x || x ||   || x ||   || x |
----------------------------------------------
| 10..... |   ||   ||   ||   || x ||   ||   |
----------------------------------------------
==============================================
```

g has multiple possible covering sets, but the one using the least total signals is from rows 1, 14, and 12. This yields $\boxed{g = r'(a'b + adg + f')}$

These equations are implemented in AND/OR/INVERT logic in the following circuit:

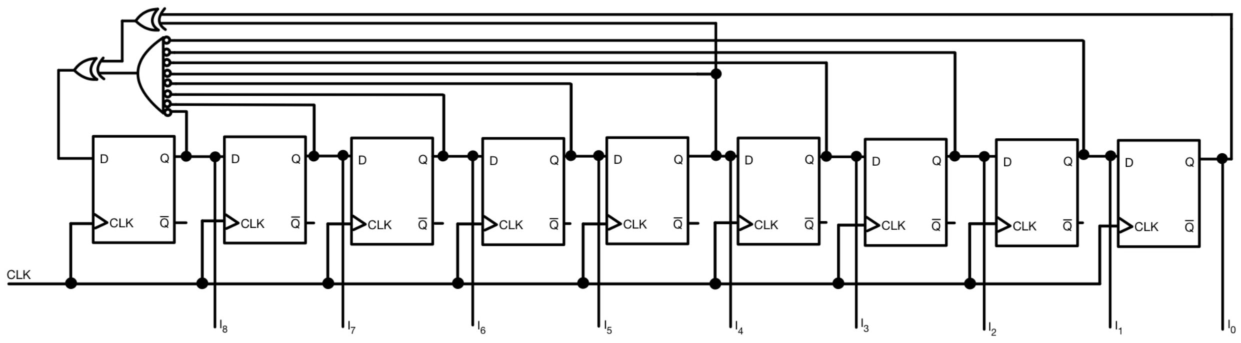

RESET
CLK
D
Q
CLK
Q̄
a
b
c
d
e
f
g

2. **Problem 2**: *LFSR Design*
   In order to design the LFSR which will count every single 9 bit value, begin with the 9 bit LFSR that produces the maximum length sequence excluding zero. This is an LFSR using feedback bits 8, and 4, so a sequence of DFFs in series with Q outputs from bits 4 and 8 fed into an XOR gate which feeds the D input of the first DFF in the series.

   To include zero in the sequence of the LFSR, we must both force zero into the shift register, and out of zero. To do this, consider how we can force the register OUT of the all zero state. This is simple, essentially add a NOR gate on all of the Q outputs of the DFFs and OR the result with the output of the XOR gate. This way, if the LFSR is in the all zero state, the new gate will drive the input to the first DFF to 1, and the LFSR will begin its sequence from 0b100000000.

   To force the zero value into the LFSR, we need to be conscious of where in the sequence to insert 0. We would like to insert 0 into the output sequence right before the LFSR would output ob100000000, so that the sequence will be exactly the same as before with a zero inserted, but with a zero inserted. The value which would normally precede 0b100000000 is 0b000000001, since the 1 at bit 8 would drive the XOR gate high and result in shifting in a 1 to a register which is otherwise all 0. So to include 0, we need to force the transition from 0b000000001 to 0b000000000. This is simple. Only allow the XOR output to drive the first DFF high when bit 8 is not the only high bit in the register. We are already checking for the all zero state with the NOR gate. Remove bit 8 from the AND gate and then XOR the output of the NOR gate with the output of the XOR gate. Now, if all signals other than bit 8 are low, the first XOR gate and the NOR both be high. Taking the XOR will result in a low signal, loading a 0 into the first DFF so that the sequence will be 0b000000000. We still will be able to recover from the all zero state, since on the next clock cycle, the XOR will be low and the NOR will be high, meaning we will get the sequence 0b100000000 and jump back into the normal sequence.

   This described LFSR is implemented in the following circuit, which will ouput the enumerated values on $I_8 ... I_0$:



   Notice that the gate count here is the count of the 9 DFFs, plus 10. This should be around 64 gates. Compare this to a 9 bit synchronous counter, which should require 1 DFF, 1 XOR gate, and 1 AND gate per bit, giving roughly 94 to 95 gates. This LFSR uses around 30 less gates, making it massively more efficient!

3. **Problem 3**: *Range Counter*

The design for this problem is fairly simple. Begin by creating a standard 7 bit counter. This is a bunch of flip flops where each output is fedback through an XOR gate to its own input, and an AND gate to the next input. The result is that every bit is high if either all previous bits are high, or the bit is high and not all previous bits are high.

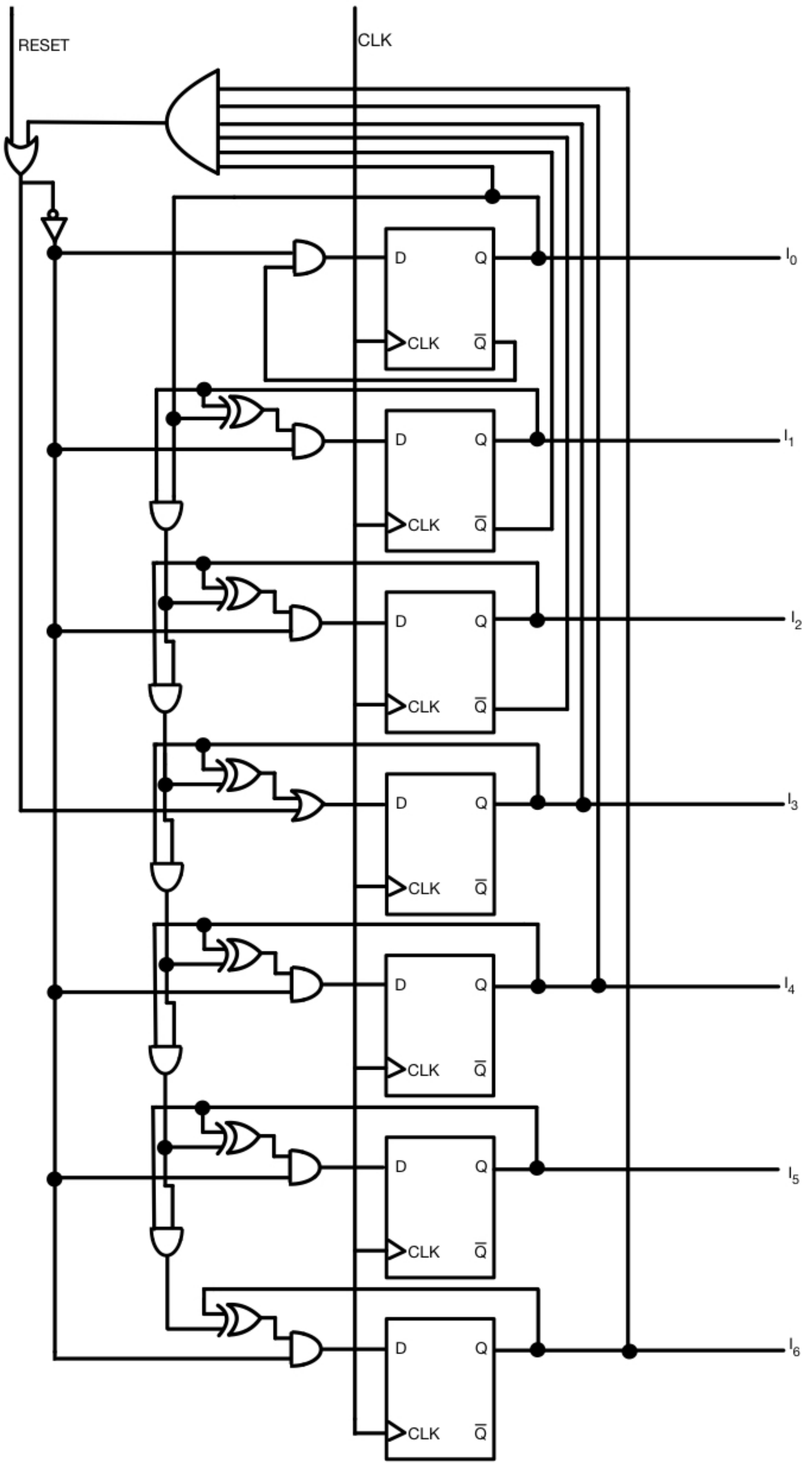
We can simplify this a bit. Simply connect the D-input on bit 0 to its own $\overline{Q}$ output so that it toggles on every clock cycle. Then bit 6 also doesn't need to output to an AND gate since it we aren't prompted to include a carry out signal.

Now, we need to restrict the range and include the RESET signal. To detect the top end of the range, place an AND gate out the outputs which detects the exact top end of the range. We can ignore bit 5 since it will never be high concurrently with bit 6 given the range restriction. We need to force the circuit to output 8 if either this AND gate is high, or the reset signal is high. Therefore, feed the RESET signal and the output of the AND gate into an OR gate. Take the resulting signal and OR it with the logic on the D-input of bit 3 so that bit 3 will go high if the range tops out or if the reset signal is high. Then, negate the output of the OR gate and AND the negation with all of the D-inputs of the other bits. This way, if either RESET is high or the range tops out, only bit 3 will be high and we will get the desired output of 8.

This results in the following set of equations:

$$\boxed{\begin{array}{l} R = I_6 I_4 I_3 I_2' I_1' I_0 + RESET \\ I_0 = R' I_0' \\ I_1 = R'(I_0 \oplus I_1) \\ I_2 = R'(I_0 I_1 \oplus I_2) \\ I_3 = R + (I_0 I_1 I_2 \oplus I_3) \\ I_4 = R'(I_0 I_1 I_2 I_3 \oplus I_4) \\ I_5 = R'(I_0 I_1 I_2 I_3 I_4 \oplus I_5) \\ I_6 = R'(I_0 I_2 I_3 I_4 I_5 \oplus I_6) \end{array}}$$

The resulting logic is implemented in the following circuit, which outputs the counter values on $I_6 ... I_0$:


RESET
CLK
D
Q
CLK
$\overline{Q}$
I_0
I_1
I_2
I_3
I_4
I_5
I_6

4. **Problem 4**: *Synchronous Design*

   To determine the rule violation of synchronous design rules, it is useful to enumerate the synchronous design rules. These are:

   (a) Use only D flip-flops are storage elements
   (b) Non-synchronous presets and clear are not allowed (other than gglobal system reset)
   (c) All DFFs need to be on the same global clock signal
   (d) Sys clock may not be gated
   (e) External and non-synchronous signals must be synchronized to the clock.

   The violations of these rules that occur within the Craps game design are as follows:

   (a) Packages U1 and U5 are 74LS92 counters. These counters use JK flip-flops as storage elements, which are not D flip-flops, and therefore violate rule a listed above.
   (b) The system clock signal is gated by part U18C on the way into U1, violating rule d listed above.
   (c) Part U7 is a 74LS175 containing 4DFFS. However its clock signal is connected to combinational logic, not the system clock line, unlike its counterpart DFFs in U8 which are on the clock generated by X1. This violates rule c listed above.
   (d) Roll is an external signal generated off of switch S1, which is not synchronized to the system clock. This violates rule e listed above.

```
################################################################################
# qm.py                                                                        #
#                                                                              #
# This file contains a Quine-McCluskey algorithm implementation.               #
# Quine-McCluskey outputs a simplified boolean expression given a truth table. #
#                                                                              #
# Author: Edward Speer                                                         #
#                                                                              #
# Revision History:                                                            #
# 10/9/2024  - Initial revision                                                #
# 10/14/2024 - Debug repeated implicants in recursion                          #
# 10/15/2024 - Add nice printing of prime implicant chart                      #
# 10/19/2024 - Print column numbers in PI tables                               #
################################################################################

################################################################################
# IMPORTS                                                                      #
################################################################################

from typing import List, Dict
from itertools import combinations, product
from re import findall

################################################################################
# HELPER FUNCTIONS                                                             #
################################################################################

"""
check_merge(minterm1: str, minterm2: str) -> bool

Check if two minterms can be merged. Minterms may be merged if they differ by a
single bit.
"""
def check_merge(minterm1: str, minterm2: str) -> bool:
    # Check that all dashes in minterms are in the same position
    if not [i for i, c in enumerate(minterm1) if c == '-'] == \
            [i for i, c in enumerate(minterm2) if c == '-']:
        return False

    # Obtain int representations of the minterms from binary strings
    minterm1_int, minterm2_int = list(map(lambda x:int(x.replace('-',
                                            '0'), 2), [minterm1, minterm2]))

    # To merge, only one bit can be twiddled
    xor = minterm1_int ^ minterm2_int
    return xor != 0 and xor & (xor - 1) == 0

"""
merge(minterm1: str, minterm2: str) -> str

Merge two minterms. The minterms must be able to be merged. The function returns
a new minterm with a dash in the position of the single twiddled bit from the
input minterms.
"""
def merge(minterm1: str, minterm2: str) -> str:
    return "".join(list(map(lambda x, y: x if x == y else '-', minterm1,
                            minterm2)))

################################################################################
# Logic Problem Class                                                          #
################################################################################

"""
LogicProblem

This class represents a logic problem. It contains a set of minterms for the
problem, and holds methods for finding prime implicants and creating a prime
implicant chart. This will allow for the creation of a simplified boolean
expression.
"""
class LogicProblem():

    # CONSTRUCTORS

    """
    __init__(self, active: List[str], inactive: List[str], mutex: bool=False)
```

```
Given a list of active and inactive minterms, create a new LogicProblem
object. The active minterms are the minterms that are true in the truth
table, and the inactive minterms are the minterms that are false in the
truth table. The false minterms are used to collect the don't care terms
for the problem. If mutex (mutally exclusive) is set to True, the inactive
set will automatically be set to the complement of the active set.
"""
def __init__(self, active: List[str], inactive: List[str],
             mutex: bool=False):
    self.minterms = active
    self.vars = len(active[0])

    if mutex:
        self.super = self.minterms
    else:
        self.super = list(set([''.join(bits) for bits in product('01',
                      repeat=self.vars)]) - set(inactive))

# METHODS

"""
prime_implicants() -> None

This function takes a list of minterms and returns a list of fully merged
prime implicants. The implementation is recursive, calling itself until a
traversal of the minterms has been completed without any further merging.
"""
def get_prime_implicants(self, minterms=None) -> None:

    # If this is the first iteration, start from all active + don't cares
    if minterms is None:
        minterms = self.super

    # list of prime implicants
    p_is = []

    # List of minterms that have been merged (one bool per minterm)
    merged = {minterm: False for minterm in minterms}

    # Merge any possible minterms from all combinations and count merges
    merge_cnt = list(map(
        lambda t: (p_is.append(merge(t[0], t[1])) or
                ((lambda i, j: merged.update({i: True}) or
                    merged.update({j: True}) or False)(t[0], t[1])) or
                True) if check_merge(t[0], t[1]) and merge(t[0], t[1]) not
                in minterms else False,
                list(combinations(minterms, 2)))).count(True)

    # Add any unmerged terms to the prime implicants list
    p_is += list(set([minterm for minterm in minterms if not
                      merged[minterm]]))

    # If no merges were made, return the prime implicants list. Otherwise,
    # call the function recursively.
    self.p_is = list(set(p_is))
    if merge_cnt != 0:
        self.get_prime_implicants(minterms=self.p_is)

"""
p_i_chart() -> None

Create a prime implicant chart. The chart is a dictionary with the prime
implicants as keys and the minterms that the prime implicant covers as
values.
"""
def p_i_chart(self) -> None:
    self.p_i_dict = {p_i.replace("-", "."): "" for p_i in self.p_is}
    for minterm in self.minterms:
        for p_i in self.p_i_dict:
            if findall(p_i, minterm):
                self.p_i_dict[p_i] += "1"
            else:
                self.p_i_dict[p_i] += "0"
    self.p_i_dict = {k: self.p_i_dict[k] for k in sorted(self.p_i_dict) if
                    "1" in self.p_i_dict[k]}

"""
```

```
run_qm() -> None

Run the Quine-McCluskey algorithm on the logic problem. This function will
call the get_prime_implicants() and p_i_chart() functions, and will print
the prime implicant chart.
"""
def run_qm(self)->None:
    self.get_prime_implicants()
    self.p_i_chart()

"""
get_table() -> None

Print the prime implicant chart in a table format.
"""
def get_table(self)->None:
    table_len = (5 * len(self.minterms) + self.vars + 4)
    print("=" * table_len)
    print(f"| {' ' * len(self.minterms[0])} ", end="")
    for i in range(1, len(self.minterms) + 1):
        if (i < 10):
            print(f"| {i} |", end="")
        else:
            print(f"|{i} |", end="")
    print("")
    print("-" * table_len)
    for p_i in self.p_i_dict:
        coverage = ["| x |" if c == "1" else "|   |" for c in
                    self.p_i_dict[p_i]]
        print(f"| {p_i} " + "".join(coverage))
        print("-" * table_len)
    print("=" * table_len)
```

```
################################################################################
# p1.py                                                                        #
#                                                                              #
# This file uses the Quine-McCluskey algorithm implemented in the Q-M module   #
# to solve for the prim implicants chart for each segment in the 7 segmnent    #
# display for problem 1 of HW3 in EECS119a.                                    #
#                                                                              #
# Author: Edward Speer                                                         #
# Revision History:                                                            #
# 10/19/2024 - Initial Revision                                                #
################################################################################

################################################################################
# CONSTANTS                                                                    #
################################################################################

# Path to Quine-McCluskey algorithm implementation
QM_PATH = "../Q-M"

# Input values for each logic problem for the 7 segment display counter
INPUTS = {
    "a": {
        "active":   ["0110000", "1101101", "0110011", "0011111", "1110000",
                     "1111111", "1110011"],
        "inactive": ["1111110", "1111001", "1011011"]},
    "b": {
        "active":   ["1111110", "0110000", "1101101", "1111001", "0011111",
                     "1110000", "1111111", "1110011"],
        "inactive": ["0110011", "1011011"]},
    "c": {
        "active":   ["1111110", "1101101", "1111001", "0110011", "1011011",
                     "0011111", "1110000", "1111111", "1110011"],
        "inactive": ["0110000"]},
    "d": {
        "active":   ["0110000", "1101101", "0110011", "1011011", "1110000",
                     "1110011"],
        "inactive": ["1111110", "1111001", "0011111", "1111111"]},
    "e": {
        "active":   ["0110000", "1011011", "1110000", "1110011"],
        "inactive": ["1111110", "1101101", "1111001", "0110011", "0011111",
                     "1111111"]},
    "f": {
        "active":   ["1111001", "0110011", "1011011", "1110000", "1111111",
                     "1110011"],
        "inactive": ["1111110", "0110000", "1101101", "0011111"]},
    "g": {
        "active":   ["0110000", "1101101", "1111001", "0110011", "1011011",
                     "1110000", "1111111"],
        "inactive": ["1111110", "0011111", "1110011"]}
}

################################################################################
# IMPORTS                                                                      #
################################################################################

import sys

# Add the path to the Quine-McCluskey algorithm to the system path
sys.path.append(QM_PATH)
import qm

################################################################################
# MAIN                                                                         #
################################################################################

"""
main()

The main function for this script runs the Quine-McCluskey on each of the
defined input tables in the INPUTS dict. The resulting prime implicant charts
printed on stdout.
"""
def main():
    for segment in INPUTS:
        print(f"Signal: {segment}")
        active = INPUTS[segment]["active"]
        inactive = INPUTS[segment]["inactive"]
        lp = qm.LogicProblem(active, inactive)
        lp.run_qm()
        lp.get_table()

if __name__ == "__main__":
    main()
```