****
EE/CS 119A Homework 2
Edward Speer
October 16, 2024
****

1. **Problem 1**: *Codebar Decoding*

   In order to simplify the logic of the codebar decoding, I began by developing a Python package which implements the Quine-McCluskey algorithm to find the prime implicants table of a given boolean function. I then wrote a script which used that package to find the prime implicant tables of each signal in the decoder. The code for that package and the script is attached at the back of the assignment. Use the chart to find the minimal sum of products representation of the function to cover all of the minterms, and reduce as much as possible.

```
Signal: S
===============================
| I6...I0 |      Minterms     |
-------------------------------
| ...1.1. | x ||   || x || x |
-------------------------------
| .1.1... |   || x ||   ||   |
===============================
```

   Trivially from the chart $\boxed{S = I_3 + I_1}$

```
Signal: V3
========================================================
| I6...I0 |                 Minterms                   |
--------------------------------------------------------
| ....1.1 |   ||   ||   ||   ||   || x ||   ||   || x |
--------------------------------------------------------
| ...1..1 | x ||   ||   ||   ||   ||   ||   ||   ||   |
--------------------------------------------------------
| ...11.. |   ||   ||   || x ||   ||   ||   ||   ||   |
--------------------------------------------------------
| ..1.... |   || x || x ||   || x ||   || x || x || x |
--------------------------------------------------------
| .0...01 | x ||   ||   ||   ||   || x || x ||   || x |
--------------------------------------------------------
| .0..10. |   ||   ||   || x ||   || x ||   || x || x |
--------------------------------------------------------
| .0.0.0. |   ||   ||   ||   ||   || x || x || x || x |
--------------------------------------------------------
| 0...0.0 |   || x || x ||   || x ||   ||   ||   ||   |
--------------------------------------------------------
| 0..1... | x ||   ||   || x || x ||   ||   ||   ||   |
--------------------------------------------------------
| 00...0. | x ||   ||   || x || x ||   ||   ||   || x |
--------------------------------------------------------
| 1.....1 |   ||   ||   ||   ||   || x || x ||   ||   |
--------------------------------------------------------
| 1...1.. |   ||   ||   ||   ||   || x ||   || x ||   |
========================================================
```

   Here, one of the possible minimum covering sets may be made up of the one single term prime implicant plus the two double term prime implicants in rows 9 and 12. Thus $\boxed{V_3 = I_4 + I_6' I_3 + I_6 I_2}$

```
Signal: V2
=======================================================
| I6...I0 |                 Minterms                  |
-------------------------------------------------------
| ....1.1 |   ||   ||   ||   ||   || x ||   ||   || x |
-------------------------------------------------------
| ...1..0 |   ||   || x || x || x ||   ||   ||   ||   |
-------------------------------------------------------
| ...11.. |   ||   ||   || x ||   ||   ||   ||   ||   |
-------------------------------------------------------
| ..0.0.0 | x || x || x ||   ||   ||   ||   ||   ||   |
-------------------------------------------------------
| ..1...1 |   ||   ||   ||   ||   ||   || x ||   || x |
-------------------------------------------------------
| ..1.1.. |   ||   ||   ||   ||   ||   ||   || x || x |
-------------------------------------------------------
| ..11... |   ||   ||   ||   || x ||   ||   ||   ||   |
-------------------------------------------------------
| .0...00 |   ||   || x || x || x ||   ||   || x ||   |
-------------------------------------------------------
| .0..10. |   ||   ||   || x ||   || x ||   || x || x |
-------------------------------------------------------
| .0.0.0. |   ||   ||   ||   ||   || x || x || x || x |
-------------------------------------------------------
| .01..0. |   ||   ||   ||   || x ||   || x || x || x |
-------------------------------------------------------
| 1...... | x || x || x ||   ||   || x || x || x ||   |
=======================================================
```

Here the minimum covering set may be made up of the one single term prime implicant plus the two double term prime impicants located in rows 1 and 2 of the table. Therefore $\boxed{V_2 = I_6 + I_3 I_0' + I_2 I_0}$

```
Signal: V1
==============================
| I6...I0 |      Minterms    |
------------------------------
| ....10. |   ||   || x ||   |
------------------------------
| ...0.0. | x || x || x || x |
------------------------------
| ..1..0. |   ||   ||   || x |
------------------------------
| .1..... | x || x || x || x |
------------------------------
| 0....00 |   ||   || x || x |
==============================
```

Trivially from the table $\boxed{V_1 = I_5}$

```
Signal: V0
==============================
| I6...I0 |      Minterms    |
------------------------------
| ......1 | x || x || x ||   |
------------------------------
| ...1... |   || x ||   || x |
------------------------------
| .0...0. |   || x ||   || x |
------------------------------
| 0.0.0.. | x || x || x ||   |
==============================
```

Here the mimimal covering set may be formed by the two single term implicants in rows 1 and 2. Therefore $\boxed{V_0 = I_3 + I + 0}$

Signal: G

I6...I0	Minterms																			
0.01001			x															x		
000.011	x																		x	
000.110		x																		x
00010.1			x																x	
00011.0											x									x
001.010					x												x			
0010101																x				
00110.0												x					x			
010.001							x											x		
0100100								x												
0110000									x											
1000010						x														
1000101													x							
1001000										x										
1010001														x						
1010100															x					
1100000				x																

From the chart above, there is only one prime implicant which may be eliminated since almost all prime implicants are responsible for individual minterms. Row 1 may be eliminated. This results in:

$$G = I_6'I_5'I_4'I_2'I_1I_0 + I_6'I_5'I_4'I_2I_1I_0' + I_6'I_5'I_4'I_3I_2'I_0 + I_6'I_5'I_4'I_3I_2I_0' + I_6'I_5'I_4I_2'I_1I_0' + I_6'I_5'I_4I_3'I_2I_1'I_0$$

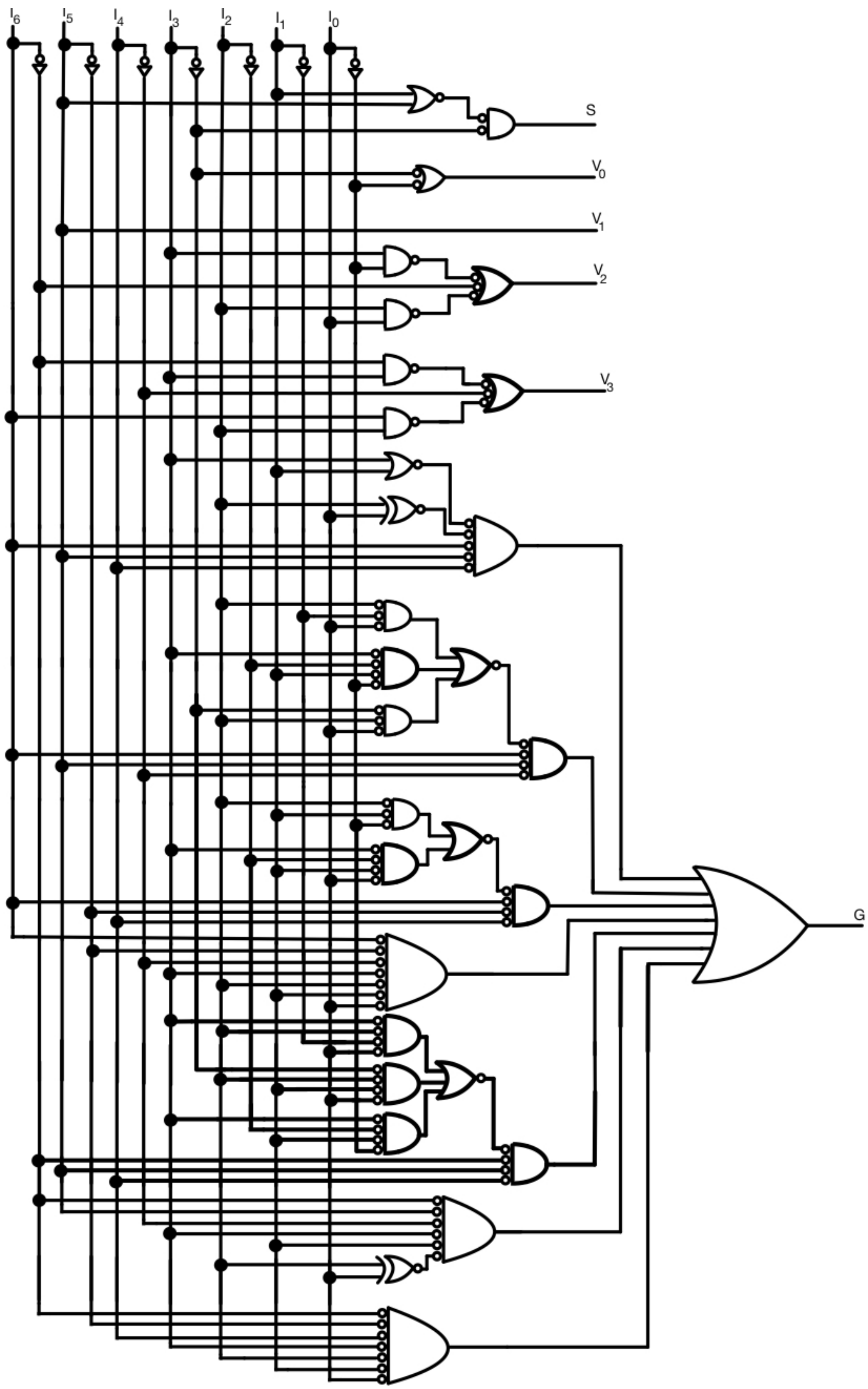
$$+I_6'I_5'I_4I_3I_2'I_0' + I_6'I_5I_4'I_2'I_1'I_0 + I_6'I_5I_4'I_3'I_2I_1'I_0' + I_6'I_5I_4I_3'I_2'I_1'I_0' + I_6I_5'I_4'I_3'I_2'I_1I_0' + I_6I_5'I_4'I_3'I_2I_1'I_0+$$
$$I_6I_5'I_4'I_3I_2'I_1'I_0' + I_6I_5'I_4I_3'I_2'I_1'I_0 + I_6I_5'I_4I_3'I_2I_1'I_0' + I_6I_5I_4'I_3'I_2'I_1'I_0$$

Combining terms and simplifying slightly:

$$G = I_6'I_5'I_4'(I_2 \oplus I_0)(I_1 + I_3) + I_6'I_5'I_4(I_2'I_1I_0' + I_3'I_2I_1'I_0 + I_3I_2'I_0') + I_6'I_5I_4'(I_2'I_1'I_0 + I_3'I_2I_1'I_0')+$$
$$I_6'I_5I_4I_3'I_2'I_1'I_0' + I_6I_5'I_4'(I_3'I_2'I_1I_0' + I_3I_2'I_1'I_0' + I_3'I_2I_1'I_0) + I_6I_5'I_4I_3'I_1'(I_2 \oplus I_0)+$$
$$I_6I_5I_4'I_3'I_2'I_1'I_0'$$

The logic for each of these signals is implemented in 61.5 gates in the following circuit, which was checked using a python script also attached at the back of the assignment.

Note that for the circuit drawn, in order to reduce gate count using negative logic, the inverters on each of the inputs have both matched and unmatched bubbles. Since they were being used across multiple inputs, it was not possible to match bubbles correctly without using additional inverters. Therefore I placed all of the bubbles on the input inverters in a consistent manner, and then ensured the bubble rules were followed throughout the rest of the design.


I6
I5
I4
I3
I2
I1
I0
S
V0
V1
V2
V3
G

2. **Problem 2**: *Glitches*

   In order to have a potential glitch in the circuit, we need a case where the transition of a signal impacts the output of multiple gates concurrently. In this circuit, we have 3 gates, meaning we have 3 pairings of gates to consider.

   - Consider the first and second product terms. The first contains C while the second contains C'. This means there is a potential for a glitch on the transition of C if we can produce an input for which the output of the whole circuit relies on the value of C in gates 1 and 2. Consider the input 1110. In this case, the second product term will start out high, while the first and third are low. When C transitions to low, the output should remain high. However, the output of the second gate will transition to low before the output of the first gate goes high due to the propagation delay of C through the inverter, meaning that the output of the circuit would temporarily glitch low even though it should stay high.
   - Consider the first and third product terms. The first term holds B while the third holds B'. This means there is a potential for a glitch on the transition of B if we can produce an input for which the output of the whole circuit relies on the value of B in gates 1 and 3. Consider the input 1101. In this case, the first product term will start out high, while the second and third are low. When B transitions to low, the output should remain high. However, the output of the first gate will transition to low before the output of the third gate goes high due to the propagation delay of B through the inverter, meaning that the output of the circuit would temporarily glitch low even though it should stay high.
   - Consider the second and third product terms. The second term is actually the NOT of the third term, meaning that no one input could cause a swap of the outputs of these gates. This means that there is no potential for a glitch between these two gates with a single signal transition.

   Therefore, the potential values for which the circuit may glitch when a signal transitions are

   $\boxed{(1110 \leftrightarrow 1100) \text{ and } (1101 \leftrightarrow 1001)}$

   To resolve these glitches, we can collect the common terms in the glitch conditions into a redundant term for each glitch condition, therefore placing an extra gate which will reamin high on the transition and thus mask the glitch by holding the output high.

   - For the transitions $1110 \leftrightarrow 1100$, this means we should add the product term $AB\overline{D}$ which will remain high regardless of the transition of C.
   - For the transitions $1101 \leftrightarrow 1001$, this means we should add the product term $A\overline{C}D$ which will remain high regardless of the transition of B.

   This gives us a final circuit with 5 product terms, described by the following equation:

   $$\boxed{Y = (A * B * \overline{C}) + (B * C * \overline{D}) + (\overline{B} * \overline{C} * D) + (A * B * \overline{D}) + (A * \overline{C} * D)}$$

3. **Problem 3**: *Gate counts*

Count the number of gates in the circuit part by part.

- U29 is a 74LS00, a quad 2-input NAND gate, giving 4 gates.
- U19, U20, and U21 are 74LS04s, hex inverters, giving $6 * 0.5 = 3$ gates, times 3 is 9 gates.
- U33 is a 74AHCT1G04, a single half-gate inverter.
- U9-U15 and U32 are 74LS08, quad 2-input AND gates, giving $4 * 1.5 = 6$ gates, time 8 is 48 gates.
- U25-U28 are 74LS11, triple 3-input AND gates, giving $2 * 3 = 6$ gates, times 4 is 24 gates.
- U31 is a 74LS20, a dual 4-input NAND gate, giving 4 gates.
- U22-U24 are 74LS27, triple 3-input NOR gates, giving $1.5 * 3 = 4.5$ gates, times 3 is 13.5 gates.
- U16-U18 are 74LS32, quad 2-input OR gates, giving $1.5 * 4 = 6$ gates, times 3 is 18 gates.
- U6 is a 74LS83, given in handout as 31 gates.
- U30 is a 74LS85, given in handout as 22.5 gates.
- U1 and U5 are 74LS92, given in handout as 21 and 27 gates respectively.
- U7 and U8 are 74LS175, quad D flip-flops with clear. These yield 4 DFFs * 6 gates per DFF, plus an additional gate for an inverter and a buffer each, giving 25 gates per package, times 2 is 50 gates.
- Now, subtract away the unused gates, a two input AND from U14, a two input AND from U32, and a two input OR from U18. This is -4.5 gates.

In total, this is $4 + 9 + 0.5 + 48 + 24 + 4 + 13.5 + 18 + 31 + 22.5 + 21 + 27 + 50 - 4.5 = \boxed{\text{268 gates}}$

```
################################################################################
# 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                      #
################################################################################

################################################################################
# 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)
    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)
```

```
################################################################################
# Assignment 2, Problem 1                                                      #
# EECS 119, Fall 2024                                                          #
#                                                                              #
# This script uses the Quine-McCluskey algorithm to find simplified boolean    #
# expressions from the truth table for the Codebar decoder.                    #
#                                                                              #
# Author: Edward Speer                                                         #
# Revision History:                                                            #
# 10/9/2024  - Initial revision                                                #
# 10/14/2024 - Fill in remaining signals to solve                              #
# 10/15/2024 - Remove unnecessary printing                                     #
################################################################################

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

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

# Input values for each logic problem for the decoder
INPUTS = {
    "S":  {"mutex":    False,
           "active":   ["0011010", "0101001", "0001011", "0001110"],
           "inactive": ["0000011", "0000110", "0001001", "1100000", "0010010",
                        "1000010", "0100001", "0100100", "0110000", "1001000",
                        "0001100", "0011000", "1000101", "1010001", "1010100",
                        "0010101"]},
    "V3": {"mutex":    False,
           "active":   ["0001001", "0010010", "0110000", "0001100", "0011000",
                        "1000101", "1010001", "1010100", "0010101"],
           "inactive": ["0000011", "0000110", "1100000", "1000010", "0100001",
                        "0100100", "1001000"]},
    "V2": {"mutex":    False,
           "active":   ["1100000", "1000010", "1001000", "0001100", "0011000",
                        "1000101", "1010001", "1010100", "0010101"],
           "inactive": ["0000011", "0000110", "0001001", "0010010", "0100001",
                        "0100100", "0110000"]},
    "V1": {"mutex":    False,
           "active":   ["1100000", "0100001", "0100100", "0110000"],
           "inactive": ["0000011", "0000110", "0001001", "0010010", "1000010",
                        "1001000"]},
    "V0": {"mutex":    False,
           "active":   ["0000011", "0001001", "0100001", "1001000"],
           "inactive": ["0000110", "1100000", "0010010", "1000010", "0100100",
                        "0110000"]},
    "G":  {"mutex":    True,
           "active":   ["0000011", "0000110", "0001001", "1100000", "0010010",
                        "1000010", "0100001", "0100100", "0110000", "1001000",
                        "0001100", "0011000", "1000101", "1010001", "1010100",
```

```
                        "0010101", "0011010", "0101001", "0001011", "0001110"]}
}

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

import sys

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


"""
main()

Main function for the script. This function runs the Quine-McCluskey algorithm
on the input values for the logic problem and prints the most simplified prime
implicants for the truth table.
"""
def main():

    for input_name in INPUTS:
        print(f"Signal: {input_name}")
        active = INPUTS[input_name]["active"]
        mutex = INPUTS[input_name]["mutex"]
        if not mutex:
            inactive = INPUTS[input_name]["inactive"]
        else:
            inactive = []

        lp = qm.LogicProblem(active, inactive, mutex=mutex)
        lp.run_qm()
        lp.get_table()


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

```python
###############################################################################
# This module is used for checking the correctness of my logic design for     #
# problem 1. It contains logic to run gate functions and check that the       #
# outputs are correct.                                                        #
#                                                                             #
# Author: Edward Speer                                                        #
# Revision History:                                                           #
# 10/14/2024  - Initial revision                                              #
###############################################################################

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

# Translate indices in the binary input to the index of the corresponding signal
# in an input string
IND = {0: 6, 1: 5, 2: 4, 3: 3, 4: 2, 5: 1, 6: 0}

# Inputs resulting in an active G output
G_ACTIVE   = ["0000011", "0000110", "0001001", "1100000", "0010010", "1000010",
              "0100001", "0100100", "0110000", "1001000", "0001100", "0011000",
              "1000101", "1010001", "1010100", "0010101", "0011010", "0101001",
              "0001011", "0001110"]

# Inputs resulting in an inactive G output
G_INACTIVE = [format(i, '07b') for i in range(128) if format(i, '07b') not in
              G_ACTIVE]

INPUTS = {
    "S":  {"active":   ["0011010", "0101001", "0001011", "0001110"],
           "inactive": ["0000011", "0000110", "0001001", "1100000", "0010010",
                        "1000010", "0100001", "0100100", "0110000", "1001000",
                        "0001100", "0011000", "1000101", "1010001", "1010100",
                        "0010101"]},
    "V3": {"active":   ["0001001", "0010010", "0110000", "0001100", "0011000",
                        "1000101", "1010001", "1010100", "0010101"],
           "inactive": ["0000011", "0000110", "1100000", "1000010", "0100001",
                        "0100100", "1001000"]},
    "V2": {"active":   ["1100000", "1000010", "1001000", "0001100", "0011000",
                        "1000101", "1010001", "1010100", "0010101"],
           "inactive": ["0000011", "0000110", "0001001", "0010010", "0100001",
                        "0100100", "0110000"]},
    "V1": {"active":   ["1100000", "0100001", "0100100", "0110000"],
           "inactive": ["0000011", "0000110", "0001001", "0010010", "1000010",
                        "1001000"]},
    "V0": {"active":   ["0000011", "0001001", "0100001", "1001000"],
           "inactive": ["0000110", "1100000", "0010010", "1000010", "0100100",
                        "0110000"]},
    "G":  {"active":   G_ACTIVE,
           "inactive": G_INACTIVE}
}
```

```
################################################################################
# LOGIC FUNCTIONS                                                              #
################################################################################

"""
INV(x: str) -> str:

Invert a bus of signals of any length.
"""
def INV(x: str) -> str:
    new_x = ""
    for i in range(len(x)):
        new_x += '1' if x[i] == '0' else '0'
    return new_x


"""
AND(x: str) -> str:

Perform a logical AND operation on a bus of signals of any length.
"""
def AND(x: str) -> str:
    return '1' if '0' not in x else '0'


"""
OR(x: str) -> str:

Perform a logical OR operation on a bus of signals of any length.
"""
def OR(x: str) -> str:
    return '0' if '1' not in x else '1'


"""
XOR(x: str) -> str:

Perform a logical XOR operation on a bus of signals of any length.
"""
def XOR(x: str) -> str:
    return '1' if x.count('1') % 2 == 1 else '0'


"""
NAND(x: str) -> str:

Perform a logical NAND operation on a bus of signals of any length.
"""
def NAND(x: str) -> str:
    return INV(AND(x))
```

```
"""
NOR(x: str) -> str:

Perform a logical NOR operation on a bus of signals of any length.
"""
def NOR(x: str) -> str:
    return INV(OR(x))


"""
XNOR(x: str) -> str:

Perform a logical XNOR operation on a bus of signals of any length.
"""
def XNOR(x: str) -> str:
    return INV(XOR(x))


################################################################################
# LOGIC DESIGN                                                                 #
################################################################################

"""
S(x: str) -> str:

Custom logic for output signal S.
"""
def S(x: str) -> str:
    return NOR(INV(x[IND[3]]) + NOR(x[IND[5]] + x[IND[1]]))


"""
V3(x: str) -> str:

Custom logic for output signal V3.
"""
def V3(x: str) -> str:
    return NAND(NAND(INV(x[IND[6]]) + x[IND[3]]) + INV(x[IND[4]]) +
                NAND(x[IND[6]] + x[IND[2]]))


"""
V2(x: str) -> str:

Custom logic for output signal V2.
"""
def V2(x: str) -> str:
    return NAND(NAND(x[IND[3]] + INV(x[IND[0]])) + INV(x[IND[6]]) +
                NAND(x[IND[2]] + x[IND[0]]))
```

```
"""
V1(x: str) -> str:

Custom logic for output signal V1.
"""
def V1(x: str) -> str:
    return x[IND[5]]


"""
V0(x: str) -> str:

Custom logic for output signal V0.
"""
def V0(x: str) -> str:
    return NAND(INV(x[IND[3]]) + INV(x[IND[0]]))


"""
G(x: str) -> str:

Custom logic for output signal G.
"""
def G(x: str) -> str:
    P1 = NOR(
        NOR(x[IND[3]] + x[IND[1]]) +
        XNOR(x[IND[2]] + x[IND[0]]) +
        x[IND[6]] + x[IND[5]] + x[IND[4]]
    )

    P2 = NOR(
            NOR(
                NOR(x[IND[2]] + INV(x[IND[1]]) + x[IND[0]]) +
                NOR(x[IND[3]] + x[IND[1]] + INV(x[IND[2]] + x[IND[0]])) +
                NOR(INV(x[IND[3]]) + x[IND[2]] + x[IND[0]])
            ) +
            x[IND[6]] + x[IND[5]] + INV(x[IND[4]])
    )

    P3 = NOR(
            NOR(
                NOR(x[IND[2]] + x[IND[1]] + INV(x[IND[0]])) +
                NOR(x[IND[3]] + INV(x[IND[2]]) + x[IND[1]] + x[IND[0]])
            ) +
            x[IND[6]] + INV(x[IND[5]]) + x[IND[4]]
    )

    P4 = NOR(x[IND[6]] + INV(x[IND[5]] + x[IND[4]]) + x[IND[3]] + x[IND[2]] +
             x[IND[1]] + x[IND[0]])
```

```
    P5 = NOR(
            NOR(
                NOR(x[IND[3]] + x[IND[2]] + INV(x[IND[1]]) + x[IND[0]]) +
                NOR(INV(x[IND[3]]) + x[IND[2]] + x[IND[1]] + x[IND[0]]) +
                NOR(x[IND[3]] + INV(x[IND[2]]) + x[IND[1]] + INV(x[IND[0]]))
            ) +
            INV(x[IND[6]]) + x[IND[5]] + x[IND[4]]
    )

    P6 = NOR(
            INV(x[IND[6]]) + x[IND[5]] + INV(x[IND[4]]) + x[IND[3]] + x[IND[1]]
            + XNOR(x[IND[2]] + x[IND[0]])
    )

    P7 = NOR(INV(x[IND[6]] + x[IND[5]]) + x[IND[4]] + x[IND[3]] + x[IND[2]] +
             x[IND[1]] + x[IND[0]])

    return OR(P1 + P2 + P3 + P4 + P5 + P6 + P7)


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

"""
check_signal(signal: function, active: List[str], inactive: List[str]) -> bool:

Check if a signal is correct for a given set of active and inactive inputs.
"""
def check_signal(signal, active, inactive) -> bool:
    for a in active:
        if signal(a) != '1':
            print(f"Signal {signal.__name__} failed for active input {a}")
    for i in inactive:
        if signal(i) != '0':
            print(f"Signal {signal.__name__} failed for inactive input {i}")

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

"""
main() -> None

The main method of this module checks the correctness of the logic design for
every signal in the design for problem 1.
"""
def main() -> None:
    # All logic functions to check
    logic_functions = [S, V3, V2, V1, V0, G]
```

```python
    # Check all signals
    for signal in logic_functions:
        check_signal(signal, INPUTS[signal.__name__]["active"],
                             INPUTS[signal.__name__]["inactive"])

    print("Finished checking logic design for problem 1")


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