SAI
UNIVERSITY

Digital Logic Design Assignment 2 (Course Project)

Submitted by

Name	:	Bhavani Pediredla
Register Number	:	220200052
E-mail ID	:	Bhavani Pediredla
School of Study	:	SCDS
Year of Study	:	2nd year

Implementation:

The Quine-McCluskey method is a widely used and effective tabulation technique for simplifying Boolean functions in digital logic and circuit design. This approach is particularly useful when dealing with a large number of variables, where the Karnaugh map method becomes impractical. To implement this method, I have chosen the Python programming language due to its versatility, ease of use, and the availability of powerful libraries for handling Boolean functions and generating prime implicants. The provided code seamlessly integrates the logic for both sub-questions, enabling a unified solution. By leveraging Python's strengths, this implementation offers a flexible and efficient way to simplify Boolean expressions, making it a valuable tool for digital logic and circuit designers.

Here is the algorithm :

1. Import the combinations functions from the itertools module , which will be used to generate combinations of terms for finding prime implicants
2. Define the function generate_minterms that takes a name string as input and generates a list of minterms based on the ASCII codes of the characters in the name.
    - Convert each character in the name to its ASCII code.
    - For each ASCII code, add its individual digits to a set of minterms.
    - Convert the set of minterms to a list and return it.
3. Define the function binary_representation that takes a minterm value and the number of variables as input, and returns the binary representation of the minterm as a string.
    - Use the format function to convert the minterm to a binary string with the specified number of bits.
4. Define the function combine_terms that takes two binary terms as input and checks if they differ by exactly one bit.
    - If they differ by one bit, return a new term that combines the two terms, with the differing bit replaced by a '-' character.
    - If they differ by more than one bit, return None.
5. Define the function find_prime_implicants that takes a list of minterms and the number of variables as input, and returns a list of prime implicants.
    - Convert the minterms to their binary representations using the binary_representation function.
    - Initialize an empty set to store the prime implicants.
    - Iteratively combine pairs of terms that differ by one bit using the combine_terms function until no more combinations are possible.
    - The resulting terms that cannot be combined further are the prime implicants.
    - Return the list of prime implicants.

6. Define the function find_essential_prime_implicants that takes a list of minterms and a list of prime implicants as input, and returns a list of essential prime implicants.
    - Create a dictionary that maps each minterm to the list of prime implicants that cover it.
    - A prime implicant is considered essential if it is the only one that covers a particular minterm.
    - Return the list of essential prime implicants.
7. Define the function minimize_expression that takes a list of minterms, a list of essential prime implicants, the number of variables, and a string of variable names as input, and generates the minimized sum-of-products (SOP) expression.
    - For each essential prime implicant, generate a term using the given variable names and the binary representation of the implicant.
    - Combine the terms using the + operator and return the minimized SOP expression.

8. **For Part (a):**

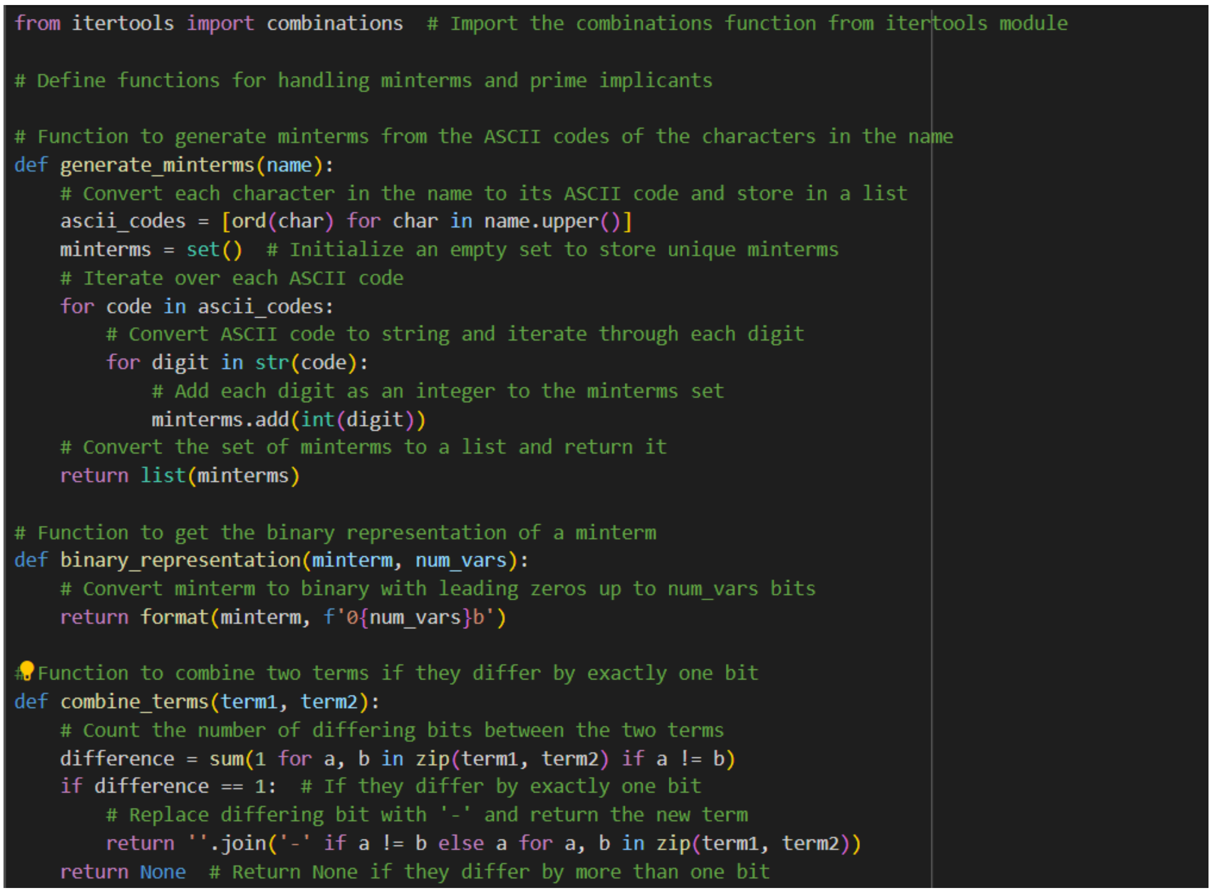
- **Define the input name "BHAVANI"**:
    - The input name is used to generate the minterms for the given problem.
- **Generate the minterms from the input name using the generate_minterms function**:
    - The generate_minterms function converts the input name into a list of minterms by extracting individual digits from the ASCII codes of the characters. Minterms are the decimal representation of the truth table rows where the output is 1.
- **Set the number of variables to 4 and the variable names to "ABCD"**:
    - This step defines the number of variables (4 in this case) and the variable names (ABCD) used in the minimization problem.
- **Find the prime implicants using the find_prime_implicants function**:
    - Prime implicants are essential for minimizing the sum-of-products (SOP) expression. They are the simplest product terms that cover one or more minterms and cannot be combined further.
- **Find the essential prime implicants using the find_essential_prime_implicants function**:
    - Essential prime implicants are those prime implicants that are necessary to include in the minimized expression because they cover at least one minterm that is not covered by any other prime implicant.
- **Minimize the expression using the minimize_expression function**:
    - The minimize_expression function generates the minimized SOP expression using the essential prime implicants and the given variable names.
- **Print the minterms, prime implicants, essential prime implicants, and the minimized SOP expression**:
    - This step displays the intermediate results (minterms, prime implicants, essential prime implicants) and the final minimized SOP expression for Part (a).

9. **For Part (b):**
1. **Define the minterms as [0, 5, 7, 8, 9, 12, 13, 23, 24, 25, 28, 29, 37, 40, 42, 44, 46, 55, 56, 57, 60, 61]**:
    - For Part (b), the minterms are given directly instead of being generated from a name.
2. **Set the number of variables to 6 and the variable names to "ABCDEF"**:
    - This step defines the number of variables (6 in this case) and the variable names (ABCDEF) used in the minimization problem for Part (b).
3. **Find the prime implicants using the find_prime_implicants function**:
    - the prime implicants are found for the given set of minterms.
4. **Find the essential prime implicants using the find_essential_prime_implicants function**:
    - The essential prime implicants are identified for the given minterms and prime implicants.
5. **Minimize the expression using the minimize_expression function**:
    - The minimized SOP expression is generated using the essential prime implicants and the given variable names.
6. **Print the minterms, prime implicants, essential prime implicants, and the minimized SOP expression**:
    - This step displays the intermediate results and the final minimized SOP expression for Part (b).

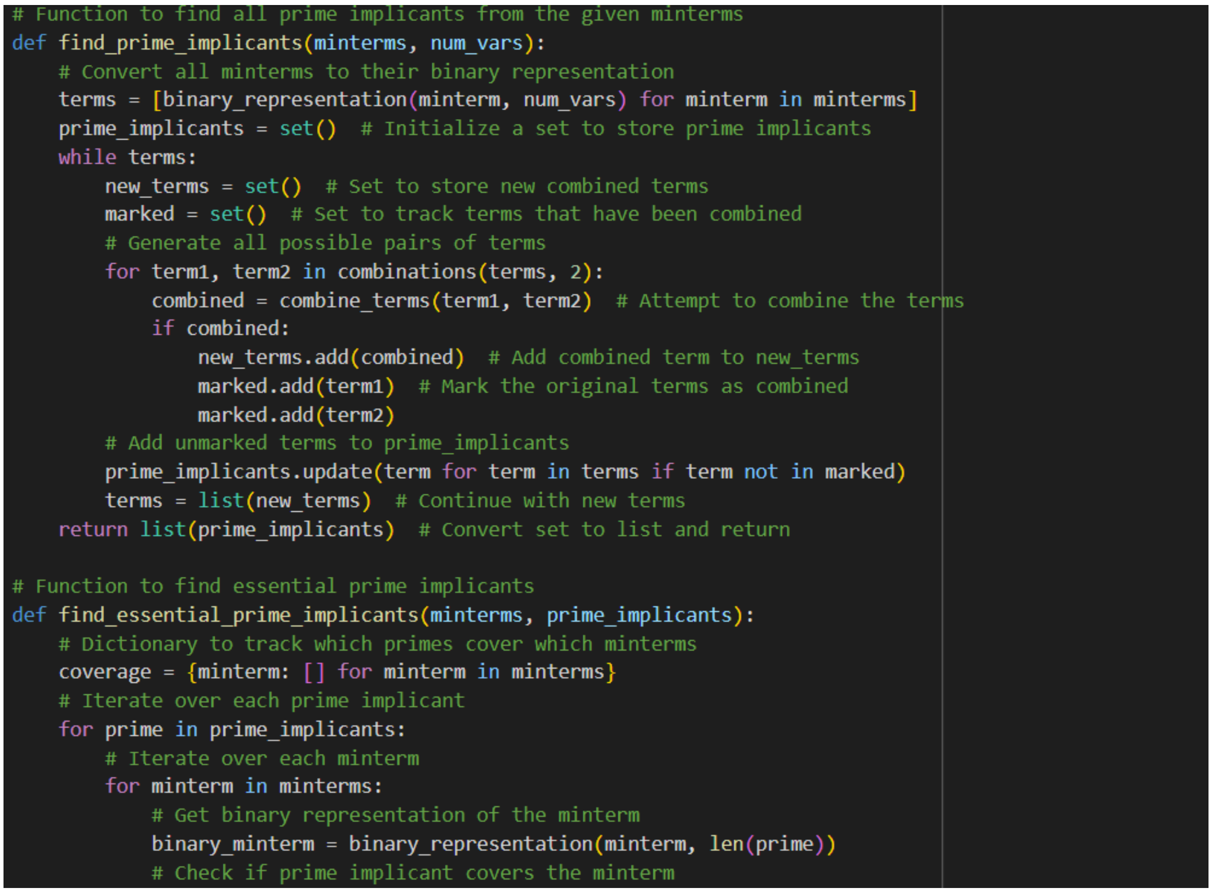
Add the 7th prime implicant to the minimized expression for Part (b):

1. **Find the prime implicant for the minterm 7 using the find_prime_implicants function**:
    - This step finds the prime implicant that covers the minterm 7, which was not included in the initial minimized expression for Part (b).
2. **Add the prime implicant to the list of essential prime implicants for Part (b)**:
    - The prime implicant for minterm 7 is added to the list of essential prime implicants for Part (b).
3. **Minimize the expression again using the minimize_expression function with the updated list of essential prime implicants**:
    - The expression is minimized again, considering the updated list of essential prime implicants, which now includes the prime implicant for minterm 7.
4. **Print the updated minimized SOP expression**:
    - The updated minimized SOP expression, which includes the prime implicant for minterm 7, is printed.

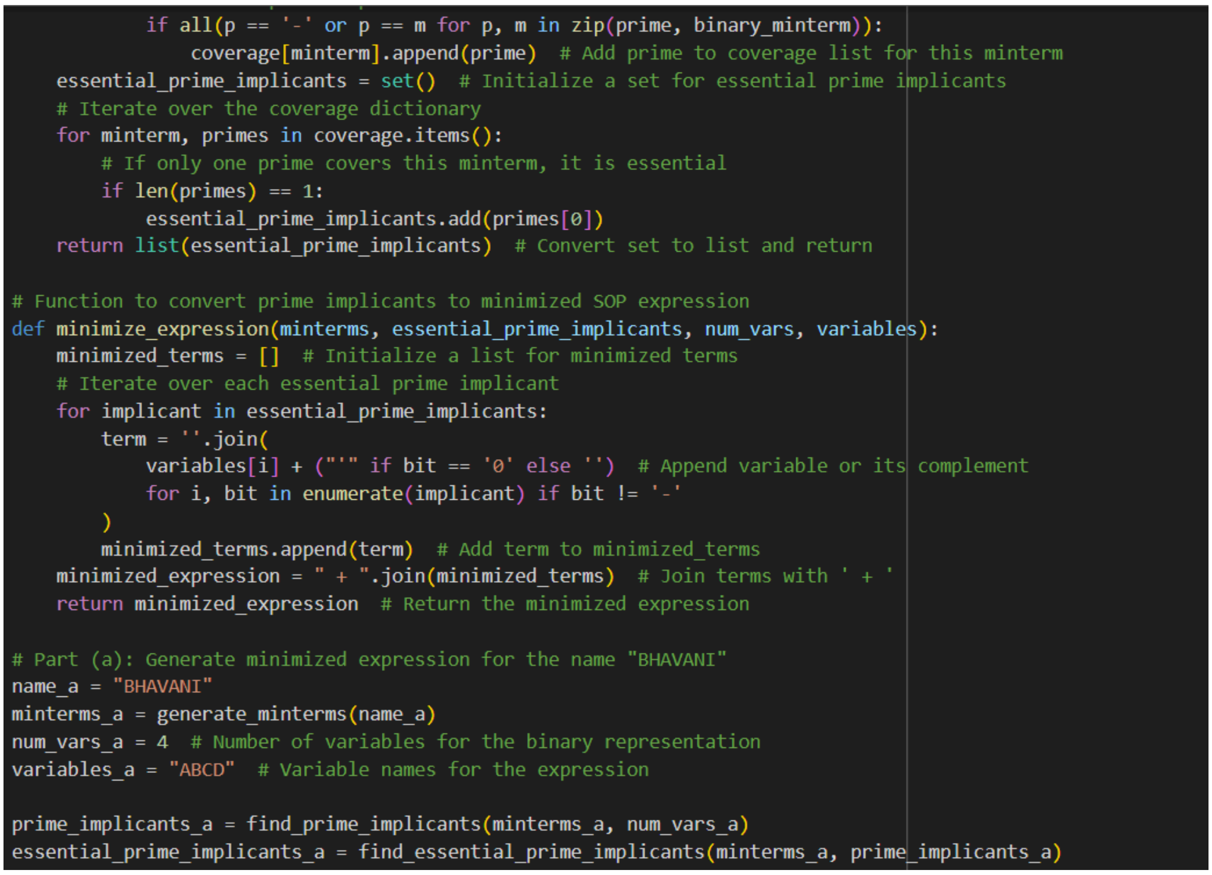
PYTHON CODE



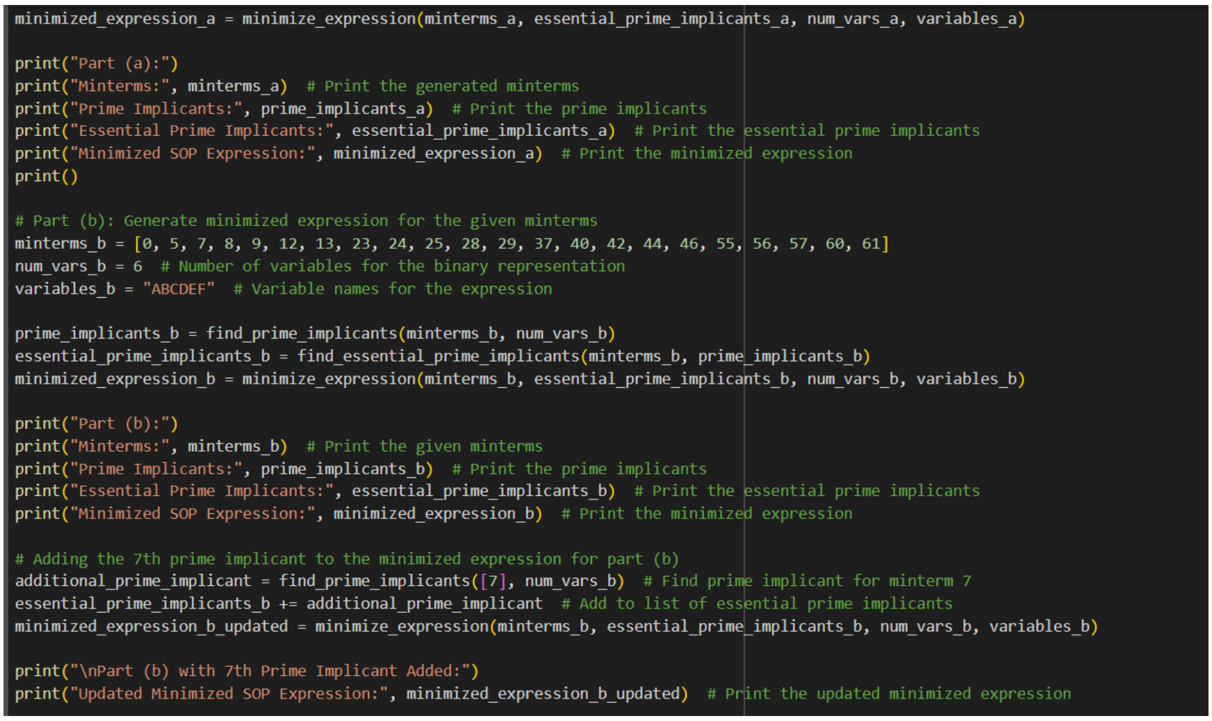
```python
from itertools import combinations  # Import the combinations function from itertools module

# Define functions for handling minterms and prime implicants

# Function to generate minterms from the ASCII codes of the characters in the name
def generate_minterms(name):
    # Convert each character in the name to its ASCII code and store in a list
    ascii_codes = [ord(char) for char in name.upper()]
    minterms = set()  # Initialize an empty set to store unique minterms
    # Iterate over each ASCII code
    for code in ascii_codes:
        # Convert ASCII code to string and iterate through each digit
        for digit in str(code):
            # Add each digit as an integer to the minterms set
            minterms.add(int(digit))
    # Convert the set of minterms to a list and return it
    return list(minterms)

# Function to get the binary representation of a minterm
def binary_representation(minterm, num_vars):
    # Convert minterm to binary with leading zeros up to num_vars bits
    return format(minterm, f'0{num_vars}b')

# Function to combine two terms if they differ by exactly one bit
def combine_terms(term1, term2):
    # Count the number of differing bits between the two terms
    difference = sum(1 for a, b in zip(term1, term2) if a != b)
    if difference == 1:  # If they differ by exactly one bit
        # Replace differing bit with '-' and return the new term
        return ''.join('-' if a != b else a for a, b in zip(term1, term2))
    return None  # Return None if they differ by more than one bit
```



```python
# Function to find all prime implicants from the given minterms
def find_prime_implicants(minterms, num_vars):
    # Convert all minterms to their binary representation
    terms = [binary_representation(minterm, num_vars) for minterm in minterms]
    prime_implicants = set()  # Initialize a set to store prime implicants
    while terms:
        new_terms = set()  # Set to store new combined terms
        marked = set()  # Set to track terms that have been combined
        # Generate all possible pairs of terms
        for term1, term2 in combinations(terms, 2):
            combined = combine_terms(term1, term2)  # Attempt to combine the terms
            if combined:
                new_terms.add(combined)  # Add combined term to new_terms
                marked.add(term1)  # Mark the original terms as combined
                marked.add(term2)
        # Add unmarked terms to prime_implicants
        prime_implicants.update(term for term in terms if term not in marked)
        terms = list(new_terms)  # Continue with new terms
    return list(prime_implicants)  # Convert set to list and return

# Function to find essential prime implicants
def find_essential_prime_implicants(minterms, prime_implicants):
    # Dictionary to track which primes cover which minterms
    coverage = {minterm: [] for minterm in minterms}
    # Iterate over each prime implicant
    for prime in prime_implicants:
        # Iterate over each minterm
        for minterm in minterms:
            # Get binary representation of the minterm
            binary_minterm = binary_representation(minterm, len(prime))
            # Check if prime implicant covers the minterm
```

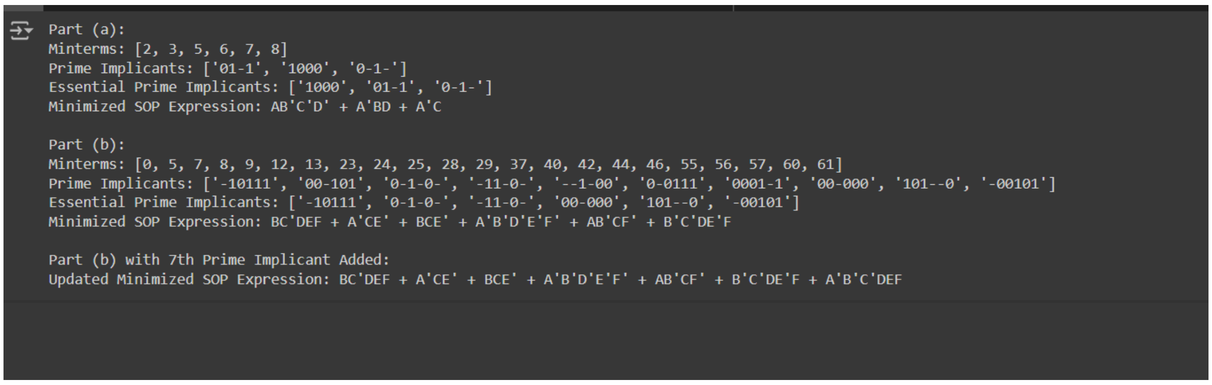


```python
            if all(p == '-' or p == m for p, m in zip(prime, binary_minterm)):
                coverage[minterm].append(prime)  # Add prime to coverage list for this minterm
    essential_prime_implicants = set()  # Initialize a set for essential prime implicants
    # Iterate over the coverage dictionary
    for minterm, primes in coverage.items():
        # If only one prime covers this minterm, it is essential
        if len(primes) == 1:
            essential_prime_implicants.add(primes[0])
    return list(essential_prime_implicants)  # Convert set to list and return

# Function to convert prime implicants to minimized SOP expression
def minimize_expression(minterms, essential_prime_implicants, num_vars, variables):
    minimized_terms = []  # Initialize a list for minimized terms
    # Iterate over each essential prime implicant
    for implicant in essential_prime_implicants:
        term = ''.join(
            variables[i] + ("'" if bit == '0' else '')  # Append variable or its complement
            for i, bit in enumerate(implicant) if bit != '-'
        )
        minimized_terms.append(term)  # Add term to minimized_terms
    minimized_expression = " + ".join(minimized_terms)  # Join terms with ' + '
    return minimized_expression  # Return the minimized expression

# Part (a): Generate minimized expression for the name "BHAVANI"
name_a = "BHAVANI"
minterms_a = generate_minterms(name_a)
num_vars_a = 4  # Number of variables for the binary representation
variables_a = "ABCD"  # Variable names for the expression

prime_implicants_a = find_prime_implicants(minterms_a, num_vars_a)
essential_prime_implicants_a = find_essential_prime_implicants(minterms_a, prime_implicants_a)
```



```python
minimized_expression_a = minimize_expression(minterms_a, essential_prime_implicants_a, num_vars_a, variables_a)

print("Part (a):")
print("Minterms:", minterms_a)  # Print the generated minterms
print("Prime Implicants:", prime_implicants_a)  # Print the prime implicants
print("Essential Prime Implicants:", essential_prime_implicants_a)  # Print the essential prime implicants
print("Minimized SOP Expression:", minimized_expression_a)  # Print the minimized expression
print()

# Part (b): Generate minimized expression for the given minterms
minterms_b = [0, 5, 7, 8, 9, 12, 13, 23, 24, 25, 28, 29, 37, 40, 42, 44, 46, 55, 56, 57, 60, 61]
num_vars_b = 6  # Number of variables for the binary representation
variables_b = "ABCDEF"  # Variable names for the expression

prime_implicants_b = find_prime_implicants(minterms_b, num_vars_b)
essential_prime_implicants_b = find_essential_prime_implicants(minterms_b, prime_implicants_b)
minimized_expression_b = minimize_expression(minterms_b, essential_prime_implicants_b, num_vars_b, variables_b)

print("Part (b):")
print("Minterms:", minterms_b)  # Print the given minterms
print("Prime Implicants:", prime_implicants_b)  # Print the prime implicants
print("Essential Prime Implicants:", essential_prime_implicants_b)  # Print the essential prime implicants
print("Minimized SOP Expression:", minimized_expression_b)  # Print the minimized expression

# Adding the 7th prime implicant to the minimized expression for part (b)
additional_prime_implicant = find_prime_implicants([7], num_vars_b)  # Find prime implicant for minterm 7
essential_prime_implicants_b += additional_prime_implicant  # Add to list of essential prime implicants
minimized_expression_b_updated = minimize_expression(minterms_b, essential_prime_implicants_b, num_vars_b, variables_b)

print("\nPart (b) with 7th Prime Implicant Added:")
print("Updated Minimized SOP Expression:", minimized_expression_b_updated)  # Print the updated minimized expression
```

Results



```
Part (a):
Minterms: [2, 3, 5, 6, 7, 8]
Prime Implicants: ['01-1', '1000', '0-1-']
Essential Prime Implicants: ['1000', '01-1', '0-1-']
Minimized SOP Expression: AB'C'D' + A'BD + A'C

Part (b):
Minterms: [0, 5, 7, 8, 9, 12, 13, 23, 24, 25, 28, 29, 37, 40, 42, 44, 46, 55, 56, 57, 60, 61]
Prime Implicants: ['-10111', '00-101', '0-1-0-', '-11-0-', '--1-00', '0-0111', '0001-1', '00-000', '101--0', '-00101']
Essential Prime Implicants: ['-10111', '0-1-0-', '-11-0-', '00-000', '101--0', '-00101']
Minimized SOP Expression: BC'DEF + A'CE' + BCE' + A'B'D'E'F' + AB'CF' + B'C'DE'F

Part (b) with 7th Prime Implicant Added:
Updated Minimized SOP Expression: BC'DEF + A'CE' + BCE' + A'B'D'E'F' + AB'CF' + B'C'DE'F + A'B'C'DEF
```

References

1. -T. Agarwal, “Quine Mccluskey Method : Algorithm, Example & Its Applications,” ElProCus- Electronic Projects for Engineering Students, Jul. 25, 2023. https://www.elprocus.com/quine-mccluskey-method/ (accessed May 19, 2024).-GeeksforGeeks, “Quine McCluskey Method,”
2. GeeksforGeeks, Apr. 22, 2022. Accessed: May 19, 2024. [Online]. Available: https://www.geeksforgeeks.org/quine-mccluskey-method/
3. M. Morris Mano, Digital Design. Pearson Educación, 2002.