THE AMERICAN UNIVERSITY IN CAIRO
الجامعة الأمريكية بالقاهرة

School of Sciences and Engineering
Department of Computer Science and Engineering
Fall 2023

CSCE 230: Digital Design 1

Project 01: Quine McCluskey

Dr. Mohamed Shalan

Submitted By:

Farida Bey - 900212071
Hussein Elazhary - 900211733

Date: Nov 5, 2023

TABLE OF CONTENTS:

Program Description: 3

Program Design: 3

1. Read in and validate the Boolean expression: 3
2. Printing the truth table and the canonical SoP and PoS: 3
3. Generating and Printing Prime Implicants: 4
4. Determining Essential Prime Implicants: 4
5. Identifying Uncovered Minterms: 4
6. Minimized Boolean Expression: 5
7. Drawing Karnaugh Maps: 5
8. Logic Circuit Diagram: 5

Instructions for running the program: 6

Contributions: 6

Farida: 6
Hussein: 6

Program Description:

This program implements the Quine-McCluskey algorithm to simplify Boolean functions with up to 10 variables. It reads, validates, and simplifies Boolean functions, generating truth tables, canonical representations, prime implicants, and essential prime implicants. It also outputs minimized Boolean expressions and can draw K-Maps and logic circuit diagrams for functions with up to 4 and 10 variables, respectively.

Program Design:

1. *Read in and validate the Boolean expression:*

The first step is to read in a Boolean function of up to 10 variables. To ensure that the input is in the correct form, the program performs a series of validation checks. The program reads the User input Function in the main.

A. The 'balancedParentheses' function checks the balance of parentheses, ensuring that each opening parenthesis has a corresponding closing parenthesis.

B. The 'isSOP' function verifies if the input expression is in Sum of Products (SoP) form, further ensuring the correct structure.

C. The 'validation' function orchestrates the validation process by checking for various conditions:

   a. The expression must not be empty.

   b. Spaces are removed from the expression to facilitate validation.

   c. Valid operators and the count of variables are defined, and the program checks for too many variables (more than 10).

   d. Multiple negations (''') are calculated and replace with either a not or nothing to ensure validity.

   e. Character-by-character validation examines each character, ensuring they are alphabet characters or valid operators.

   f. Specific issues, such as consecutive operators or inappropriate use of not and brackets, are reported.

   g. The code verifies that the expression doesn't start or end with operators other than the parenthesis.

   h. Operator sequences are checked, and potential issues, like starting with '(' followed by '+', are identified.

If any of these validation checks fail, the program provides error messages, guiding the user toward resolving the issues.

2. *Printing the truth table and the canonical SoP and PoS:*

Helper Functions for Printing:

A. The 'transform' function converts the expression into a standard format. Within this format, '+' is replaced by '|' for logical OR, '!' is utilized for logical NOT, and '&' for logical AND.

B. The 'isOperator' and 'getPrecedence' ensure that logical operators ('!', '&', '|') are correctly processed.

Conversion of the Expression:

The 'convert_exp' function evaluates the transformed expression and derives the Boolean result.

Generating the Truth Table: The truth table generation process follows a well-defined series of steps:

1. Determine the number of variables in the expression, identifying the unique alphabet characters.
2. Sort the variables for consistency.

3. Calculate the number of rows in the truth table based on the number of variables.
4. Print the header row, presenting variable names for clarity.
5. Generate and print the truth table row by row, this is where 'convert_exp' function evaluates the output of the truth table.

Canonical SoP and PoS:

1. For each row in the truth table, the program evaluates the expression to determine whether it results in '1' (true) or '0' (false).
2. If the result is '1', a term is formulated for the canonical SoP form.
3. If the result is '0', a term is composed for the canonical PoS form.
4. The generated terms are collected into lists for SoP and PoS, respectively.
5. Finally the canonical sop and pos are printed.

3. *Generating and Printing Prime Implicants:*

Helper Functions:

A. The 'countOnes' function counts the number of '1's in a binary string.
B. 'compareByOnes' function used in 'sortbyones'.
C. 'sortbyones' sorts the minterms based on the number of '1's they include.
D. The 'calculateDecimalValue' function transforms binary strings into their decimal representations.

Generating Prime Implicants:

1. Sorting Minterms by the number of 1s.
2. Initializing Containers, 'primeImplicants' stores prime implicants, while 'marked' tracks minterms that have been combined during the process.
3. Mapping Minterms to Decimal to identify the minterms each implicant covers.
4. For each pair of minterms if only one bit differs, these minterms can be combined.
5. Keep track of combined minterms and document the minterms covered by prime implicants.
6. Repeat the steps for each column.
7. Removing Duplicates.
8. Returning Prime Implicants and minterms they cover.

4. *Determining Essential Prime Implicants:*

The function “generateEssentialPrimeImplicants” extracts the Essential Prime Implicants (EPI) by the following steps:

1. The function takes in a map of Prime Implicants (PI) with associated minterms
2. For each PI, the minterms are parced and incremented based on how many PIs cover them
3. If the mintermCount is 1, then this indicates unique coverage
4. Unique coverage PIs are added to the EPI vector.
5. The vector is then sorted and redundancies are removed
6. The function returns a vector of EPIs

5. *Identifying Uncovered Minterms:*

The function “generateMintermsNOTCoveredByEPI” identifies minterms that are not covered by any Essential Prime Implicants (EPI) through the following steps:

1. The function accepts a map of Prime Implicants (PI) with associated minterms, a vector of all minterms, and a vector of EPIs.
2. It iterates through the EPIs, parsing and extracting their associated minterms.

3. These minterms are collected in a vector to identify those covered by the EPIs.
4. The function then iterates through the original list of all minterms.
5. It checks if each minterm is absent in the EPI-covered minterms list.
6. Minterms not covered by any EPI are compiled into a separate vector.
7. This resulting vector of non-covered minterms is returned by the function.

6. *Minimized Boolean Expression:*

Helper Functions:

A. The 'splitMinterms' function splits the comma-separated minterms into individual minterms, storing them in a vector.
B. The 'countCoveredMinterms' function counts the number of uncovered minterms, covered by the PIs.
C. The 'removeCoveredMinterms,' updates the list of uncovered minterms.
D. The 'final_answer' function formats the minimized Boolean expression. (used in part 4,6)

Generating Minimized Expression:

1. Copying Essential Prime Implicants in the minimized expression vector.
2. Then select prime implicants until all minterms are covered. It identifies the "best candidate," which is a prime implicant that covers the most remaining uncovered minterms.
3. Constructing Minimized Expression, the constructed expression is in terms of 0s, 1s, and dashes.
4. The minimized expression is then given to the 'final_answer' function to format into a boolean expression.

7. *Drawing Karnaugh Maps:*

Helper Functions:

A. The `extractVar` function extracts the variables used in the Boolean expression.

Print K-Map:

1. The function starts by determining the dimensions of the K-Map based on the number of variables. K-Maps can have 2x2, 2x4, or 4x4 configurations depending on the number of variables.
2. The function then fills in the K-Map based on the provided list of minterms, indicating which cells are covered by these minterms.
3. The labels for rows and columns in the K-Map are determined by the gray code.
4. The function prints the K-Map along with variable labels.

Cover Visualization:

To implement the cover visualization, the program uses the prime implicants to do so as each cover is a prime implicant.

1. Extract the minterms the PIs cover. These minterms are then passed to the `print_KMap` function to visualize how each prime implicant covers specific regions in the K-Map.
2. The output includes a series of K-Maps, each representing a cover in the map.

8. *Logic Circuit Diagram:*

To draw the logic circuit diagram, the following functions were implemented:

1. **parseMinExp**
    a. The function takes in a string representing a minimized boolean expression.
    b. Iterates over the expression, splitting it at '+' characters to separate product terms.
    c. Compiles the product terms into a vector.
2. **termsToWaveDromJSON**
    a. Accepts a vector of product terms.

   b. Formats these terms into WaveDrom JSON syntax based on the number of terms and their composition.
   c. Handles cases for no terms, single terms, and multiple terms, incorporating logic gates and inversions where necessary.
   d. Completes the JSON structure and returns the WaveDrom JSON string.

3. **printDrawing**
   a. Utilizes "parseMinExp" to parse the simplified boolean expression into product terms.
   b. Converts these terms into WaveDrom JSON format using "termsToWaveDromJSON".
   c. Constructs HTML content embedding the WaveDrom scripts and the generated JSON.
   d. Writes the HTML content to an output file.
   e. Opens the output file in the default web browser to display the WaveDrom visualization of the simplified boolean expression.

Instructions for running the program:

1. Install a C++ Compiler
2. Choose a Code Editor/IDE
3. Clone the Program
4. Compile the C++ Program
5. Run the Compiled Program
6. Interact with the Program
7. Interact with the running program, provide input and review output.

This program will take a valid boolean expression given as:

a. A SoP (for example: abc + a'b'c), or

b. A PoS (for example: (a + c)(d' + e)

and output the truth table, canonical SoP & PoS, prime implicants, essential emplicants, minterms not covered by essential, the K-map, K-map covers and the logic circuit of the boolean expression.

Contributions:

Farida:

- Task 2 (Truth table, canonical SoP/PoS)
- Task 3 (PIs)
- Task 6 (Minimized boolean expression using 3-step heuristic)
- Task 7 (K-map representation)
- Bugg/issue fixes

Hussein:

- Task 1 (Validation)
- Task 4 (EPIs)
- Task 5 (minterms not covered by EPIs)
- Task 8 (Logic circuit)
- Bugg/issue fixes