Merna Hisham-900221976
Sabry Wael-900214356
Adham Hassan- 900211924
Digital Design I – Fall 2023

Dr. Mohamed Shaalan

Project 1

Project Description:

The project is intended to do the following:

1. Take from the user and validate a Boolean function that is either in the form of SOP or POS using its minterms as decimal numbers.

2. Generate and print the corresponding K-map.

3. Generate and print the simplified Boolean expression

1. Introduction

In this report, we will discuss the design, development, testing, and instructions for a Boolean function minimization program. The program's primary objective is to simplify Boolean expressions and find essential prime implicants using the Quine-McCluskey Logic Minimization algorithm.

The program accepts Boolean expressions, generates truth tables, identifies minterms and maxterms, computes prime implicants and essential prime implicants, and produces minimized expressions. This report provides insights into the design, challenges faced during development, testing procedures, and instructions for building and running the program.

2. Program Design

2.1. High-Level Overview

The Boolean function minimization program consists of several key components:

- Expression Parser: Parses and validates Boolean expressions to check whether the expressions input by the user are valid or invalid SOP or POS expressions according to the format of them..

- Truth Table Generator: Generates truth tables based on the input expression. The number of columns generated in the truth table depends on the number of variables entered by the user in the expression. This was managed by adding a variable called number of variables to control this part of the program.

- Prime Implicants Calculator: Identifies prime implicants through having a function that takes the truth table as a parameter then calculates the prime implicants according to the minterms which are shown as ‘1’ in the result of the truth table.

- Essential Prime Implicants Finder: Determines essential prime implicants using the Quine-McCluskey algorithm by constructing a cover chart using ‘X’ then applying the algorithm to get the essential prime implicants from the cover chart..

For the first step, we used the regex (regular expression) library to validate a boolean expression. In order to validate a boolean expression and make sure it is in either pos form or sop form, we made sure that the only characters that are valid are characters from a-z, a plus sign, a multiplication sign, white space, apostrophes, or parentheses. We made sure to check for balanced parentheses and for valid operation usage (no double + or *). Then we used regex match to check if the expression inputted is in the form of sop of pos by limiting the accepted form products + products or sums * sums.

We have a variety of functions including ostream& operator<<(ostream& os, const vector<string>& vector) which allows a vector to be displayed using cout without a loop. We also have an evaluateExpression(const string& expression, const vector<bool>& inputValues) function which returns 0 or 1 depending on the expression inputted. This function finds the output of an expression using stacks. The generateTruthTable(const string& expression, int numVariables) function generates a truth table, as evident from the name, by using 2-dimensional vectors. The number of rows is decided by the number of variables (2^n). The truth table is unconventional since it is read from right to left, and the most significant bit is the rightmost one instead of the leftmost one. It also cannot read the apostrophe as inversion, which is a flaw in our program.

Next we have the getCanonicalSOP(const vector<vector<bool>>& truthTable, int numVariables) function which returns a string with the canonical sum of products by checking if the result is true, and if it is, then include all the variables in that row. The getCanonicalPOS(const vector<vector<bool>>& truthTable, int numVariables) function is very similar, except it checks for when the result is false to include all the variables in the row. The get_minterms(const std::vector<std::vector<bool>>& truth_table) function returns a string vector that contains the minterms. It checks if the last column is true (the result column) to decide whether or not to add the current term. The get_maxterms(const std::vector<std::vector<bool>>& truth_table) is similar, except it checks for when the result is false to combine the variables.

The differByOneBit(const std::string& term1, const std::string& term2) function checks if two strings differ by only one bit to see if they can be combined or not. If the difference is exactly 1, it returns true. The getPrimeImplicants(const std::vector<std::string>& minterms) function returns the prime implicants in the form of a string vector. It loops over the terms and checks if they've been used or not. To know this, we have a boolean vector named used that is initially false for the size of the current terms. There is another loop that checks if the current term and the next term differ by only one bit. If they do, then set the used vector to true for both terms. The current term is stored inside a string called combined terms. There is another loop that checks if the kth character in the first term and the kth character in the next term are equal to each other, and if they're not, replace it with a dash since that is the one differing character. The combined term is then added to the nextTerms vector. A separate for loop is needed inside the while loop to check for any minterms that have not been used. If they aren't used, then we push them into the primeImplicants vector. Afterwards, the nextTerms are placed inside the current terms and then it gets erased for the loop to continue.

The displayMinterms(const vector<string>& primeImplicants) function shows the minterms in letter-format. The binaryToLetters(const string& binaryRepresentation) and lettersToBinary(const string& lettersExpression) functions convert either binary results to variables or the opposite. Afterwards we have the covers(const string& minterm, const string& prime_implicant) function which checks if an implicant covers a minterm which is used by the create_coverchart(const vector<string>& prime_implicants, const vector<string>& minterms) function. The create_coverchart function fills in the chart based on coverage and checks to see if the prime implicant covers the minterm. If it does, then an "X" is placed inside the cell. The getEssentialPrimes(const vector<vector<string>>& coverChart) function uses the cover chart to check if the minterms are covered. It returns the essential primes in a string vector. The print_uncovered_minterms function uses the covers2 function to print the minterms not covered by essential prime implicants. These functions are automatically called in the main once the user types in an expression.

2.2. Data Structures

The program employs data structures such as vectors for storing truth tables, prime implicants, and cover charts. The program also uses stacks and sets to store data in the memory. These structures are efficiently used to manage and manipulate data throughout the program.

2.3. Algorithms

- Quine-McCluskey Algorithm: The algorithm is employed to identify prime implicants and determine essential prime implicants.

- Truth Table Generation: A systematic approach is used to create truth tables for Boolean expressions.

2.4. User Interface

The program offers a user-friendly command-line interface. Users input Boolean expressions, and the program processes the data to deliver minimized expressions.

3. Problems and Challenges

3.1. Issues Faced

During the development of the program, several challenges were encountered:

- Handling negations in Boolean expressions

- Accurate identification of essential prime implicants

- Generating and displaying cover charts effectively

- Time constraints to finish the rest of the algorithm, especially steps 6,7 and 8.

3.2. Solutions

- To address negations, we modified the expression parsing logic to handle 'a' and 'a" (NOT a) correctly.

- We refined the essential prime implicants determination process by fine-tuning the Quine-McCluskey algorithm.

- The cover chart display was enhanced to improve comprehensibility and correctness.

4. Program Testing

4.1. Testing Methodology

Testing of the program was comprehensive and included the following:

- Unit Testing: Each module of the program was tested individually to ensure correct functionality.

- Integration Testing: Interactions between program components were tested to guarantee seamless operation.

- Test Cases: A wide range of test cases (10), including different Boolean expressions and edge cases, were designed to validate the program's correctness.

4.2. Test Cases

The 10 test cases that were used are:

- Input: "a*b". -> This is a correctly entered expression according to the format of SOP or POS expressions

- Input: "a+b". -> This is a correctly entered expression according to the format of SOP or POS expressions

- Input:"a*b+c". -> This is a correctly entered expression according to the format of SOP or POS expressions

- Input:"a*b*c*d*e". -> This is a correctly entered expression according to the format of SOP or POS expressions

- Input:"(ab+c)". -> This should produce an error due to having the product of 2 variables in the sum of a POS form.

- Input: "a'+b". -> This is a correctly entered expression according to the format of SOP or POS expressions

- Input:"a++b". -> This should produce an error due to having 2 plus signs.

- Input:"(a+b". -> This should produce an error due to the missing parenthesis at the end of the expression

- Input:"a+b*c+d*e+f". -> This is a correctly entered expression according to the format of SOP or POS expressions

- Input:"a+b*c". -> This is a correctly entered expression according to the format of SOP or POS expressions

4.3. Results

The testing process identified minor issues with the program, primarily related to expression parsing and cover chart generation and due to time constraints step 6,7 and 8 are incomplete. However, the code correctly validates the expressions entered by the user by checking if it is a correct SOP or POS expression or not. Moreover, the code produces the correct truth table of the expression along with having the canonical SOP and POS according to the expression input by the user. Also, the code calculates the prime implicants and essential prime implicants using the Quine-McCluskey Logic Minimization algorithm.