Digital Design I

Project 1 Report

Sarah Elsamanody 900212915
Nour Abdalla 900213024
Ali Elkhouly 90021

Program Design

This program is a fully developed C++ program with html scripting to run WaveDrom to display circuits. The way this program works is that the set of requirements were divided upon subtasks and distributed equally using the grades of each requirement to guarantee fair and equal distribution. The requirements were written down in separate header files and all combined in one source file by including the headers.

The Process of Each Requirement

A. Input & Validation

In the first part, it was asked to take input as either SOP or POS format. This type of input has many restrictions which made it difficult as there were many test cases to consider. After planning out all possible test cases that are considered invalid, we reached the point that we had to check:

1. If any character other than small latin characters, '+' & ' were written.
2. Check the values following +.
3. Check the values following ' .
4. Checking the parentheses as it differs from SOP to POS.

After ensuring the validation, the next functions required modification to the string entered by the user and these modifications included:

- Removing the spaces.
- Removing double negations.

Conflicts Faced:

- Writing an efficient method to check the invalid options.
- Coming up with the sequence of what to check first.

Conflict Resolution:
We tried spotting patterns to find the right sequence to come up with the right approach to the algorithm.
Complexity: O(n) ; n is length of input.

B. Truth Table & Canonical SOP & POS

In the second question of the programming part, we had to build a truth table representing the minterms and maxterms of the function that we get. The conflicts that we had to conquer were as follows:

1. Size (number of rows) of the Truth Table.
2. Knowing what our variables are and their number (as in index).
3. How to make the Truth Table show the minterms as 0000, 0001 and etc.
4. Setting our output as 1 at the correct minterm.

First of all, we chose a map of character and vector of integers as our truth table; the character representing the variable name and being the key index in the map, and its vectors being the frequency to show when it is ON or OFF in the table. We used bit manipulation to get the number of rows in the Truth Table, by left-shifting 2 by the number of inputs subtracted by 1. So if there is only 1 input for example, we would get 2 rows. 4 rows for 2 inputs etc. We chose bit manipulation as it takes less complexity rather than using the power function in C. Moreover, in the first function, we used a STL Set of characters to store the variables, as there will be no repeated variables in the set, and it'll also sort them. Then by filling the Truth Table (excluding the output for now) we noticed that with each 0, a certain number of rows away, there becomes a 1. Such sequence changes to each column, so we implemented an algorithm accordingly. To set the output accordingly, we added a new column at the Truth Table labeling it as 'X'. We used an STL vector of strings prepared at the input that holds each Sum/Product as an element. Furthermore, we prepared a boolean variable that represents whether the input is in the form of Sum of Products or Product of Sums. If the input was a Sum of Products, we first iterate through the Products we have, and then loop over each row to check if the output should become ON or OFF. We first check if the Output is already ON or not, since if it is already ON we do not want to change it. We check each character in the Product, and we also check if there is a /' after it, to check if it is negated or not. If the variable is the same as in the Truth Table, in other terms if it is negated in the Product and it is 0 in the Truth Table, then we use an AND operation with an integer variable called final that is initialized as 1. After the Product, we change the output accordingly, by equalizing the Output with the variable "final". For the Product of Sums, we first set all my Outputs as ON (1) and then do the same iteration, however we invert the conditions. To show the Canonical Sum of Products, we had to iterate through the table and check whenever the Output is 1. If the Output is 1 (or true/On), we iterate over the characters in the Set and if the variable is 1, we add it into a string, if it is 0, we add it with an inverter (\') behind it. After that we insert the string to a vector of strings that shows the Sum of Products. For the Product of Sums, we iterate through the Truth Table searching for 0 in the output, if we do that we do practically the same; we loop through the row and check if the variable is ON, we add it to a string with a + after it. If it is OFF we add an inverter. Then we print them all using Loops.

Conflicts Faced:

- I used to change the Output even though it was already true.
- When a product would produce a true, another product would be false at the exact minterm, so it would overwrite the original value to 0 instead of 1, even though it should be true.
- Iterating over the set of characters.
- Sets cannot be accessed through indexes and square brackets, and can only be accessed through iterator pointers.

Conflict Resolution:

- I check if Output is already 0 or 1 before altering it using a simple if statement.
- Learned the syntax of iterating through sets online and trained on how to iterate through sets.

Complexity:

- Time: O(m * 2n-1); m being the number of products and n being the number of variables.
- Space: O(n*2n-1); n being the number of variables.

C. *Prime Implicant Generation*

In the third question of the programming part. We had to generate all the Prime Implicants. I had to do the following:

1. Write the minterms currently as binary expressions.
2. Sort them by the number of 1's.
3. Divide them into groups depending on the number of 1's in each.
4. Comparing each expression in each group by all the expressions in the following group.
5. Remove the use of expressions /covers.
6. Track and print the minterms each PI covers.

We started with initializing an STL vector of string called “terms” to store the first binary expressions as a form of strings. We used string so that we can add ‘-’ in the binary expression/cover. We iterated through the whole Truth Table, and whenever the Output was 1, we then added the minterm in a string and then added it to the vector. We also track the index and add the index in a vector of int, to represent the minterm each cover represents.

After that, we added the terms into a vector of sets (called sets), each set representing the expressions depending on their number of 1’s. For example, sets[0] would contain all the expressions carrying no 1’s. We filtered through the terms and added them according to the number of 1’s in them. Into the comparison, we iterated through the vector of sets, comparing each set with its following. We just used the simple “i+1” trick to compare each. However, we had to insert a new set each time insert the product of the comparison into. We then kept iterating over each element in the set with each element in its following set, checking if the logical distance is only 1, we then change it to ‘-’. If we find 2 expressions and we add the outcome of their comparison, we push both of these expressions in another vector of strings to remove them from the final list in the end.

In the end, I had to delete the redundant empty sets for less memory usage. Empty sets are the product of creating a set in case there is similarity between 2 existing sets, but turns out there isn’t. After that, an iteration is performed to remove all the used implicants, as they are not prime. We then add all the covers into a vector of strings (It was requested in the 4th function for simplicity). Finally, we compare them with the original terms, to check which minterms they cover, and add them into a vector of vectors of integers. The first index represents which cover, and 2nd index represents the minterms.

Conflicts Faced:

- Deleting the used covers.
- It would crash very often, and I couldn’t remove the implicant right away after using it, as I might combine it with another implicant in the group.
- Checking what minterms each Implicant covers.
- It was difficult to track the minterms each implicant covered in the iteration. In further context, when combining covers, the minterms would also combine and checking their indexes would become tiresome and difficult.

Conflict Resolution:

- Stored all the used implicants in the end and removed it from the final list.
- Compared all the final Prime Implicants with the original terms that each represented only 1 minterm. If the only difference was ‘-’, then that PI would also cover that term.

Complexity:

- Time: O(nn); n being the number of rows in the truth table.
- Space: O(nn); n being the number of rows in the truth table.

D. Essential Prime Implicant Generation

In the fourth part, we were told to generate the EPIs from the PIs generated. This part was relatively easy as it only required the truth table, PIs, the minterms and variables used. However, merging them with the algorithm in mind was challenging..

Conflicts faced:

- Manipulation of the data structure .
- Frequently checking for number of 1’s to output the correct EPIS
- Converting the binary input to boolean expression

Conflict Resolution:

- I had to have my date accessible in multiple ways.
- I used minterm frequency which is a map that shows how many times a specific minterm is covered by multiple covers.
- I checked if it was one cover then took it to be my EPI.
-

E. Minterms Not Covered

For this part in the question, I noticed that it can be combined with the previous and that is by checking the frequency as well and double checking that the PIs covering it do not overlap with the EPIs I had just selected.

Conflicts Faced:

- Looping was tricky in order to avoid EPIs.
- Creating a covered chart needed by my colleague for the next requirement.

Conflict Resolution:

- I fixed it through a data structure map < *int* , vector<string> >.
- that shows PIs and their corresponding minterms.
- Learned syntax to deal with vectors of pairs.

F. Minimized Expression

In part 6, we were asked to take the coverage chart and return the minimized function. To do this we had to implement an algorithm that:

1- Detects and removes the dominating columns
2- Detects and removes the dominated rows
3- Removes and stores the single number of one's in each column in a vector
4- Use the given EPIs from previous functions and combine them with the remaining PIs of the coverage chart
5-Transfer the combined vector to the correct format used in the K-maps generator.

I implemented the needed functionalities using 6 different procedures and worked on the chart using a vector<pair<string, vector<int>> .This coverage chart had its first pair as a vector<string> representing the keys and the second pair vector<int> representing the minterms covered by each row of these keys. I was able to output the required minimized expression by manipulating over this coverage chart using loops and conditions.

Conflicts faced:

- Looping over vector<pair<string, vector<int>> columns and rows correctly to find which are dominated/dominating
- Coming up with a simple algorithm that returns if rows/columns are dominated/dominating
- Correctly erase and count for duplicates as well as looping backward
- Repeatedly check for single ones after removing columns/rows to output the correct minimized dunction

G. K-map Generation

In part seven we were asked to output the K-map and the covers of the minimized expression. To do this we have to:

1-Combine both EPIS and PIs in one vector to contain the needed covers
2-Transform the EPIs & PIS to strings of characters(variables) to be able to print the covers using K-maps
3-Storing all possible covers that could be found on K-map.
3-Accommodate for different numbers of variables as they would output distinct k-maps sizes and covers.

Conflicts faced:

- Thinking of all possibilities and listing them correctly.
- Merging with existing different vectors to use the correct one for the K-map covers.
- Algorithm is compatible for any number of variables even if the number of variables in minimized functions < the number of variables in the original input.

H. Circuit Generation

In part eight we had to draw the minimized function from function 6 using the wavedorm application, To do this we had to :

1-Use the string given from minimized expression and write it in the javascript file to output the right logic circuit

2-Making it more efficient by linking waveform to HTML in order for the code to automatically output the circuit instead of opening the application and loading a file.

3-Use the javascript code which represents the circuit and write it to the HTML code to produce the circuit and link to waveform successfully.

Conflicts faced:

- Linking wavedorm to html where one has to add the directory of the file on their local computer.
- Writing the right syntax in the javascript file from c++ code for diverse logic circuits.
- Taking into consideration different numbers of inputs to be written and outputted.
- Reading, writing & updating between the c++, html & javascript codes

Test Cases

1-aaa+a' -To make sure that it runs correctly for duplicates
Minimized function to be outputted : a+a'
2-a'''+ab -To handle the number ' and accommodate it accordingly
Minimized function to be outputted : b+a'
3-(a+b)+(b c) -To make sure it takes either the correct sop or pos format only.
Minimized function to be outputted : No output since it is invalid
3-ab'+20 -Only takes valid inputs from users
Minimized function to be outputted : No output since it is invalid
4-1001 -Will not accept binary inputs
Minimized function to be outputted : No output since it is invalid
5-(a+b)(c+d') - Validating Pos
Minimized function to be outputted : ac+ad'+bc+bd'
6-c'd+cd'+c'd'+cd - Works even when boolean expression is equal to 1
Minimized function to be outputted : 1
7-a'b'c'd'e'f' + ab'c'd'e'f' + a'bc'd'e'f' + abc'd'e'f' + a'b'cd'e'f' + ab'cd'e'f' + a'bcd'e'f' - Works and does not print K-map
Minimized function to be outputted : c'd'e'f'+b'd'e'f'+a'd'e'f'
8-abc+bcd' - Normal test
Minimized function to be outputted : abc+bcd'
9-a'bc'd'+ab'c'd'+a'bcd'+ab'cd'+abcd'+abc'd'+abcd+abcd'+a'bcd'+abcd -Only EPIS are minimized function
Minimized function to be outputted : abc+ad'+bd'
10-a- Test for logic circuit when it is directly connected with output.
Minimized function to be outputted : a
11- 0 -to see when the output is 0

Contributions

The whole project was built on teamwork and great communication. Each individual was responsible for tasks with equal points.

Sarah Elsamanody took A, D, E.
Ali Elkhouly took B, C.
Nour Abdalla took F, G, H.& test cases.

The overall merging of the functions into main.cpp file was done by all contributing members and we all combined our brainstorming to come up with the final product.