Digital Design I

Project 1 – Report

Alaa Anani

Jomana Abd El-Rahman

Summary:

The code for this project, when it runs, asks the user for the number of minterms they wish to use. Next, it asks the user what these minterms are. After all the data is collected from the user, the program then output the corresponding K-Map. Note that the code is optimized for 3 variable K-Maps, and therefore the maximum number of minterms is 8. After the K-Map is generated, the code then generates the simplest possible Boolean expression from the K-Map, how this is done will be explained, in detail, below.

To start off, it must first be noted that the program uses 2 different structs and 12 different functions. Each of these functions carries out an important role when generating the K-Map and producing the simplest Boolean expression from the K-Map.

Structs:

The two structs were called KMapElement and Implicant, respectively. The struct "Implicant" has a vector of type KMapElement called minterms, which stores the cells included within each implicant.

Data Structures used:

A 2D array of type KMapElement is created. A vector of type Implicants is created to store all implicants. Exactly 5 vectors are created using the struct KMapElement, these are all the possible combinations of minterms over a single cell containing "1".

Functions and algorithm:

The set of functions serve several purposes, sequentially. First, there's the function named GetInput to obtain the input from the user: number of minterms, what the minterms are in decimal and it keeps looping if the input is not valid from the user (If the number of minterms and the decimal minterm is more than 8). Next, the functions called IntializeKMap and IntializeBinaryMintermsKMap intialize the 2D array that will later be used as K-Map. The IntializeKMap function initializes the KMap with the corresponding decimal values of each slot, state of each slot (0 or 1) and number of inclusion of each slot to 0 at the beginning.

IntializeBinaryMintermsKMap initializes binary minterms to each cell. The function PrintKMap then outputs the K-Map as if it were a 2D array. The AlgorithmFindImplicants is called next,

what it does is basically find the prime implicants from all the possible implicants in the K-Map. To put it differently, it loops over the K-Map and compares a single cell to all adjacent cells that can be combined together. It works with 5 temporary vectors (V1, V2, V3, V4, V5) which cover all possible implicants that can be formed over a single cell containing "1". Each vector is for a certain possible implicant, if it occurred, then the minterms (cells) are pushed into it. Then after getting the possible combinations, the 5 vectors are passed to a function to get the vector of maximum size (Which is the implicant of maximum size that could be formed over this one cell – prime implicant.) This vector of maximum size is pushed into the vector AllImplicants that should have the maximum size implicant (prime implicant) for each cell. The inclusion for each cell is incremented by 1 each time it is pushed into an implicant.

AssignInclusionTimesToImplicants is called to initialize the final implicants' cells with its inclusions. RemoveDuplicateImplicantsTrial2 is then called to remove all duplicate implicants and leave only one of them. SortAllImplicantsBySizeInAscendingOrder is called to sort the vectors created before (the vectors of all possible combinations) in ascending order, this is to prioritize removing implicants of smaller size and have minterms that are all included more than once (Which means they are unnecessary since their minterms are already covered by other implicants (possibly prime).)

Now that the implicant sizes are sorted, after the duplicate implicants are removed, GetEssentialPrimeImplicants is called to isolate the EPIs (Essential Prime Implicants) as they are needed to figure out the Boolean expression. This is done through looping over the implicants and checking the inclusion of their minterms, if an implicant has minterms that are all included more than once, then it is not an essential prime implicants and it is removed. The cells covered by the removed implicant will have their inclusion decreased by one. Finally, GetFinalExpression is called to extract the Boolean expression from the Essential Prime Implicants obtained before that remain in AllImplicants vector. It loops over each implicant and access its vector of minterms. The first minterm 3 characters are compared to other minterms. If a character is not the same in any of the minterms included in the ESP, it doesn't get added to the product. If it is the same in all minterms, then check if it is a zero or one. If it is a one, add char(i+65) to the product. Here "i" is the index of the character and 65 is the decimal equivalent to A in ASCII table. If it is zero, then add "~"+"char(i+65) to the product. Then the product is

formed after reaching the last character. It is added to the final string expressing after adding a "+" after it. It continues looping over all implicants and does the same. In the end, the sum of products final expression is formed with an extra "+" which gets erased in the string (FinalExpression.erase(FinalExpression.length() -1, 1)).

Conclusion:

Although it would have much easier to generate the K-Map and the Boolean expression after using Quine-McCluskey as it is more systematic that the K-Map method, the K-Map method turned out to be just as viable. Our code covers all possible cases for the K-Map and finds the simplest Boolean expression successfully.