

Department of Computer Science and Engineering

Fall 2016

CSCE2301/230

Digital Design 1

Dr. Mohamed Shalan

Name: Amr Adel Gouhar

ID: 900153482

Name: Salma Hisham Eid

ID: 900141392

Project #1

1st of November

**Quine-McCluskey Logic Minimization**

**Procedure of QM method and algorithm for implementation**

**Objectives:**

1. Read in (and validate) a Boolean function using its minterms and don’t care  terms (as decimal numbers).
2. Generate and print all prime implicants using the Quine-McCluskey  tabulation method.
3. Using the prime implicants generated, obtain and print all the  essential prime implicants.
4. Print the optimized function based on the least number of literals and the least number of inversion.

**Code Description:**

**List of the functions used in this algorithm:**

20 functions in total

void verify();

void Input();

void convert();

string DtoB(int, int);

string fillinzeros(string);

void countones();

bool differences(string, string);

string showDiff(string, string);

void getprime();

void comparison(vector<forms>);

void compare();

void deleteprime();

void ConvertToVariables();

void diplayfunction();

void getessential();

bool checkexist(int, string);

void clearcover();

void cover();

void createcoverage();

void printfunction();

void header();

**Input:**

The user inputs the number of variables of the function, then the minterms of the function as decimal digits; to finish inputting minterms the user has to insert -1. The user then has to input don’t cares and then inserts -1 to finish the input.

**Procedures of QM step 1:**

All the minterms are converted into their binary form and arranged into different groups according to the number of 1s in the binary representation. These groups are put in ascending order in the Column 0.

Algorithm implementation:

The main building block of the code is a struct “minterm” (A global variable so it can be accessed by all the function)

struct minterm { // a struct for the minterm input by the user

int index = 0; // the user input (the number of the minterm)

int ones = 0; // the number of ones in the minterm

int include = 0; // the number of primes that include this minterm

string binary = ""; // the binary form of the minterm

bool dontcare = false; // if it's a dont care or not

bool includef = false; // if it's included in the optimized function or not

bool operator < (const minterm & rhs) const {

return (ones < rhs.ones);

}

The struct of F (referring to the function) includes 6 data elements to be able to sort and identify the minterms later on in the code. The user enters the mintern decimal value stored in “Index”. Then the user is asked to enter the don’t care terms; to set apart the don’t care terms the bool element “dontcare” its a default False; but set to true of the minterm is a don’t care.

All the minterms and the don’t-cares are then pushed into vector F which stores all of the minterms.

The function “***convert***” loops on all the minterms in vector F calling function “***DtoB***” to translate the decimal value “index” of each minterm to its binary equivalent of 0s and 1s stored in the string struct element “binary”.

The function “***countones***” counts the number of ones in the string “binary” and stores it in the struct element “ones”.

***sort*** is called, aided by the library #include<algorithm>, to sort the vector F based on the number of ones in each minterm.

**Procedures of QM step 2):**

A function compare is then called to apply the tabulation method and creates the table. The values of F are put in order into colb (the imaginary col 1).

In col 1 (colb) every two minterms from two adjacent groups of ones are paired for possible combination if they only differ in one variable. The digit being canceled is denoted by ‘-’. After the first round of pairing, a new set of terms that are smaller by one literal are produced and put in cola (the imaginary column 2). Terms in Column 1 are further paired for further reduction. The same procedures will be repeated until no terms can be paired anymore. In these columns, all the terms that have been paired will be ticked. All the terms that are not ticked are the prime implicants.

Algorithm implementation:

The comparison made by two minterns is done by the function “**comparison”.**

Two vectors are initially declared as global variables **colb** and **cola**.

The function “**comparison”** loops over colb and the result terms of the comparison (terms including ‘ - ‘) are stored in cola, colb is then cleared. Cola is coped in to colb then it is cleared. The same procedures will be repeated until no terms can be paired anymore (No terms with only one difference = cola being empty) this is done to save space as we don’t have to store the whole table.

More details about function “**comparison”:**

A new struct is used to represent the terms stored in colb and cola since some of the elements from struct ‘minterm’ are no longer needed and other elements are required to separate prime implicants (PI) and essential prime implicants (EPI).

struct forms {

string form; // the binary form of the merged minterms

bool ticked = false; // if it's ticked or not

int ones = 0; // the number of ones

};

“**comparison**” calls two functions “**differences**” and “**showDiff** ”. “**differences**” compares two minterms and determines whether or not they have only one difference. It returns True if they do, False otherwise.

It there is only one difference between terms “**showDiff** ” is then called to generate the result string with the difference between them denoted by '-'.

The result string from the comparison is then stored in each struct in the element ‘from’.

The function “**compare**” included all the functions mentioned above, this is done for organization of the code, and to make sure that the operation is repeated till all the prime implicants are deduced.

**Procedures of QM step 3):**

From all columns, the terms (which are not ticked) are PIs

Algorithm implementation:

Before clearing colb and working on the terms generated from the comparison in cola, “**compare**” calls “**getprime**” to extracts the unticked minterms as they are the PIs. While doing that the PIs are stored as a new form of struct ‘prime’ in a new vector ‘primes’.

struct prime { // a struct for the prime implicants

string form; // the binary form

string min; // the variable form; for instance : AC'D

bool mark = false; // marked as essential or not

bool care = false; // includes minterms or not (not only don't cares)

int included = 0; // the number of minterms included in the coverage table

bool includedf = false; // whether it's included in the optimized function or not

bool operator < (const prime & rhs) const {

return (included < rhs.included);}

};

This new form makes it easier to extract the EPIs and produce an optimized function later on in the code.

“**deleteprime**“ loops over the extracted primes and delete any repeated primes.

**Procedures of QM step 4):**

An ‘PI chart’ will be made. Each column represents a minterm (excluding don’t cares) and each row represents a PI. Column with only a single **X** indicates a minterm covered by only one PI, which means this PI is EPI.

Algorithm implementation:

For checking if PI is an EPI or not the function “**getessential**” is called.

1) It loops over each minterms, checking only the not-dontcare-minterms (as explained in the struct), for each minterm it loops over the PIs. Calling “**checkexist** checking if it exists in how many primes. If it is included the struct element ‘include’ in ‘minterm’ is incremented by 1.

2) Loops over all the minterms if it's only included in one prime (include == 1)

Then loop over the primes to find that prime then makes it as essential (mark = true). It also marks it to be included in the optimized function (includedf = true).

**Printing:**

“**ConvertToVariables**” is called to convert the string of 1s, 0s and –s to letter representations for printing using the ASCII. (string ‘form’ struct element is converted and stored in string ‘min’)

“**diplayfunction**” is the function used to print all the requirements.

It prints:

1. The variables used: Using the gobal vairble **n** (Total number of variables used) and ASCII code of the letter (char(65 + **n**))
2. The prime implicants and then the variable form (ex. AB'C’)

By looping over the ‘primes’ vector and printing the string ‘min’.

1. The essential prime implicants and their variable form

By also looping over the ‘primes’ vector and printing the string ‘min’ only if it is marked as essential (mark = true)

**Verifying the input:**

Function “**verify**” checks if:

1. The user input (minterm index) has a negative value.

If negative it informs the user of the Invalid input of minterm and gives him a choice: Press 1 to delete or 2 to change the invalid input.

1. The user input (minterm index) is greater than the possible limit of minterms 2^n where n is the number of variables.

If it’s greater than the limit, it informs the user of the Invalid input of the minterm and gives him a choice: Press 1 to delete or 2 to change the invalid input.

1. The user input (minterm index) is repeated

If repeated it informs the user of the Invalid input of minterm and gives him a choice: Press 1 to delete or 2 to change the invalid input.

**Printing the Function itself:**

The function “**createcoverage**” represents the imaginary coverage table and finds the optimized function by looping over the minterms (not don’t cares.) It checks ‘includef’ (a bool operator in the ‘minterm’ struct that indicates if the minterm is included in the optimized function or not). If any minterm is not yet included in the main function the counter(c) is increment by 1.

If the counter is 0 (all the minterms are included in the optimized function) then the function is printed and the program ends. Else “**createcoverage**” creates a new coverage table and calls the function that clears the coverage table “**clearcover**” and the function that reconstructs the new coverage table “**cover**” and then recheck.

“**clearcover**”: this function removes the used minterms and primes from the coverage table.

By looping over the vector ‘primes’. If the prime is included (includef = true) it loops over the minterms to find the corresponding minterms in the coverage table and assign them as included in the function (includef = true)

“**cover**”: this function creates the new coverage table after removing the included primes and minterms.

After including all the minterms “**Printfunction**” prints the optimized function. The optimization is based on including the best combination of minterms that has the least number of literals and the least number of inversions along with the essential prime

implicants.

**Challenges:**

1. Tracing the error and debugging was challenging with so many struct elements.

2. Having the code printing multiple prime implecants

3. Limiting the number of victors used for space efficiency

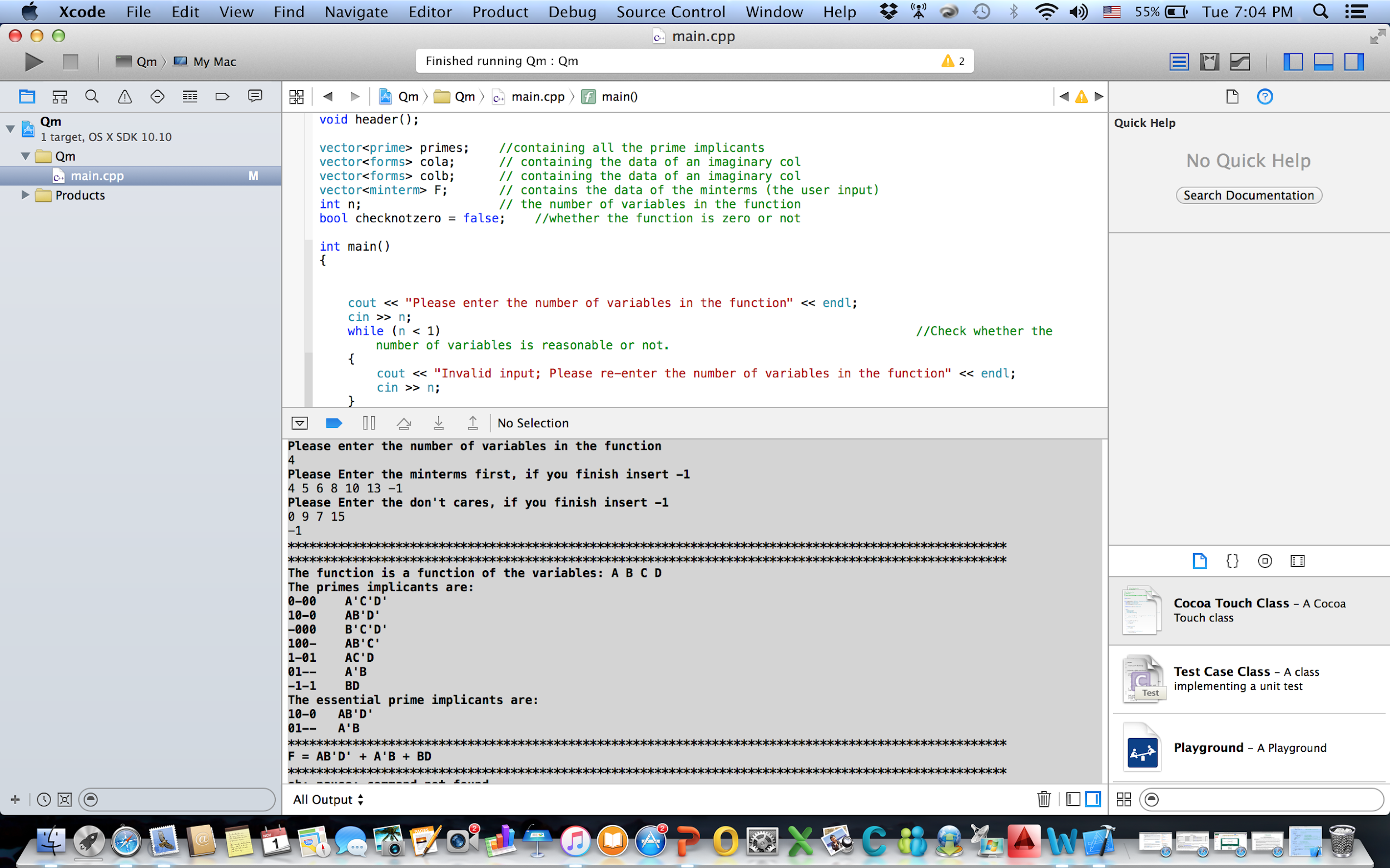
4. Printing the Function itself

**Limitations:**

1. It takes more time to output the prime impplicants and essential prime implicants compared to other similar codes using pointers and arrays.

**Test example:**

An example with 4 variables:



An example with 5:

