EE6094
CAD for VLSI Design
Programming Assignment 3 Report

Student Name:魏子翔

Student ID:107501019

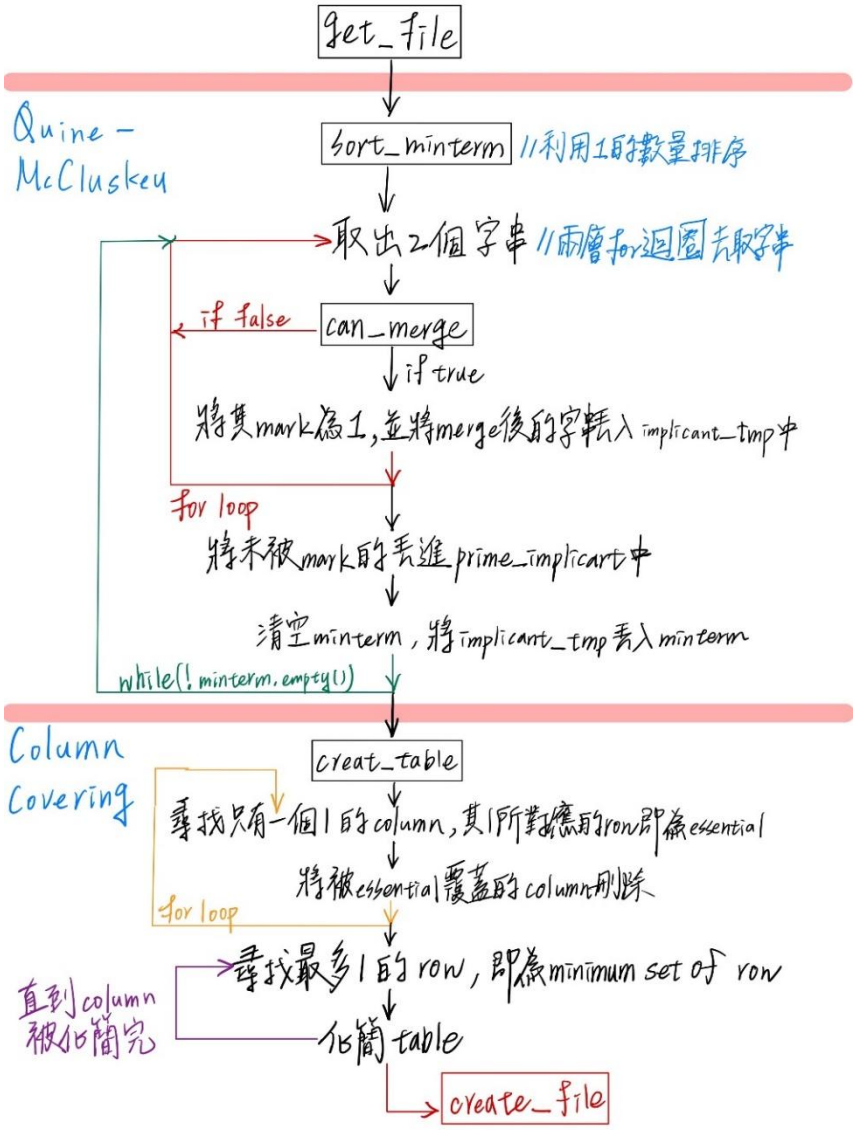
Abstract

Logic optimization 的目的在於去優化給定的 combinational circuit，以滿足性能或是面積上的限制，好讓 technology mapping 時，能將非電路形式的 Boolean networks，有效的轉換成最佳成本的給定限制之電路。

I. Problem Description

而本次 Programming Assignment 的目的在於 Two-level Logic Optimizarion，去化簡其中 Sum-of-Product 的數量。題目將給定一尚未被簡化的 Boolean function，我們將利用 C++寫出 Quine-McCluskey 演算法去簡化 Boolean function。

II. Program Structure



III. Data Structures / Algorithms Used

主要的資料結構：

```
vector<string> minterm;
vector<string> input;
string output;
vector<string> prime_implicant;
vector<string> on_set;
vector<vector<bool> > table;
vector<string> minimum_cover;
```

一、 當 testcase file 讀進來之後，testcase file 會分成 3 行：

1. 第一行的 INORDER 會利用 *vector<string> input;*去存取。
2. 第二行的 OUTORDER 會利用 *vector<string> output;*去存取。
3. 第三行的 Boolean function 會先利用 *vector<string> func_tmp;* 去存取，將其轉為二進位之後，再存進 *vector<string> minterm, on_set;*

二、 進行到 Quine_McCluskey 演算法：

1. 我們將化簡 *vector<string> minterm;*。
2. 根據化簡的結果會分成 *vector<string> prime_implicant;*及 *vector<string> implicant_tmp;*。
3. 將 *minterm* 中無法 merge 的丟進 *prime_implicant*。
4. 其餘的丟進 *implicant_tmp*，將 *minterm* 清空後，把 *implicant_tmp* 丟進 *minterm* 裡，從 2.開始周而復始，直到 *minterm.empty()*。

三、 執行 Column Covering

1. 此步驟主要的資料結構為 *vector<string> implicant_tmp;*、*vector<string> on_set;*以及根據 *implicant_tmp*、*on_set* 所建立的 *vector<bool> table;* 及儲存結果的 *vector<string> minimum_cover;*。
2. 先找出 *table* 中只有一個 1 的 column(*on_set*)，再將 1 所對應

的 row(*prime_implicant*)即為 essential，存入 *minimum_cover* 中。

3. 刪除 essential implicant 所覆蓋到的 column 來化簡 *table*。
4. 從化簡後的 *table* 中找出最多 1 的 row(*prime_implicant*)即為 minimum set of row，將其存入 *minimum_cover*，並將其覆蓋到的 column 刪除，以化簡 *table*，重複步驟 4.，直到 column(*on_set*)為空。
5. 將 *on_set*、*prime_implicant*、*table* 都 clear。

四、 創建 output file

1. 主要會使用的資料結構為 *vector<string> input;* 、*vector<string> output;*、 *vector<string> minimum_cover;*。
2. 先將 *input* 輸出到第一行。
3. 再將 *output* 輸出到第二行。
4. 利用 *input*、*minimum_cover* 將 output function 輸出。

演算法：

演算法主要分為兩個，一個為 Quine-McCluskey 演算法，另一個為 Column-Covering。

1. Quine-McCluskey：

```
void Quine_McCluskey_ALGO() //Quine McCluskey 演算法
{
    sort_minterm(); //先進行排序
```

```
    while(!minterm.empty()) //若 minterm 還未清空，則繼續執行演算法
    {
        bool merged_mark[minterm.size()];   //標記是否 merge 過
        for(int i=0; i<minterm.size(); i++)
            merged_mark[i] = 0;
        vector<string> implicant_tmp;   //將 prime implicant 以外的
implicant 先暫存起來
        for(int i=0; i<minterm.size(); i++) //開始進行演算
        {
            for(int j=i+1; j<minterm.size(); j++)
            {
                if(can_merge(minterm[i], minterm[j]) == MERGED)  //
確認是否可以 merge
                {
                    implicant_tmp.push_back(merge(minterm[i], minte
rm[j])); //將 merge 好的字串暫存進 implicant_tmp
                    merged_mark[i] = MERGED; //將其 mark 為 merge 過
                    merged_mark[j] = MERGED;
                }
            }
            if(merged_mark[i] == NO_MERGED) //將沒有 merge 過的丟進
prime implicant
            {
                prime_implicant.push_back(minterm[i]);
            }
        }
        if(!implicant_tmp.empty())
        {
            vector<string>::iterator it,it1;
            for(it=++implicant_tmp.begin(); it!=implicant_tmp.end()
;)
            {
                it1 = find(implicant_tmp.begin(), it, *it);
                if (it1 != it)  //將 implicant_tmp 中重複的的
implicant 刪除
                    it = implicant_tmp.erase(it);
                else
                    it++;
```

```
            }
        }
        minterm.clear();    //清空 minterm
        for(int i=0; i<implicant_tmp.size(); i++)   //將
implicant_tmp 丟進 minterm，再次執行演算
            minterm.push_back(implicant_tmp[i]);
    }
    FreeVector(minterm);    //釋放記憶體
}
```

2. Column-Covering。

```
void Column_Covering_ALGO() //Column Covering 演算法
{
    int on_set_amount = on_set.size();  //未被 cover 過的 on_set 數量
    create_table();
    column_has_single_1(&on_set_amount);
    minimum_set_of_row(&on_set_amount);
    //////演算完將沒用到的變數都釋放記憶體
    FreeVector(on_set);
    FreeVector(prime_implicant);
    FreeVector(table);
}
```

詳細演算步驟如同上述資料結構及 Program structure 所描述。

IV. Difficulty Encountered

這次 PA 的難點主要在於 Quine-McCluskey 的化簡及空間的使用，在寫 Quine-McCluskey 時沒有注意到 merge 後的 implicants 會有重複的問題，所以再生成 prime implicant 時一直出錯，研究一段時間後，在紙上自己畫過一遍 Quine-McCluskey 後，發現 merge 後的 implicants 會有重複的問題，在每跑一次演算時就將重複的刪掉就解決程式出錯的問題。而空間的使用比較擔心的問題就是，

當 testcase 一大，記憶體就會爆掉，所以在設變數時就盡量設容量小的，像是原本紀錄是否被 mark 是利用 vector<int>寫，之後就把他改成 vector<bool>去存，而原本的 minterm 也想用 unsigned vector<bool>去存，但是會發生無法存取 unknown '-'字元的問題，最後還是只能用 vector<string>去存。

V. Experimental Results

1. 執行 makefile：

```
[107501019@eda359_forclass ~/hw3]$ make clean
Removing objects
Removing executable file
[107501019@eda359_forclass ~/hw3]$ make all
Compiling: PA3_107501019.cpp -> PA3_107501019.o
Generating executable file: PA3_107501019.o -> go
[107501019@eda359_forclass ~/hw3]$ make run INPUT=test.eqn OUTPUT=out1.eqn
./go  test.eqn  out1.eqn
[107501019@eda359_forclass ~/hw3]$ make run INPUT=test2.eqn OUTPUT=out2.eqn
./go  test2.eqn  out2.eqn
[107501019@eda359_forclass ~/hw3]$ make run INPUT=test3.eqn OUTPUT=out3.eqn
./go  test3.eqn  out3.eqn
```

2. 用 abc 比對的結果：

```
[107501019@eda359_forclass abc-master]$ ./abc
UC Berkeley, ABC 1.01 (compiled May  2 2021 22:03:20)
abc 01> cec test.eqn out1.eqn
Networks are equivalent.  Time =     0.01 sec
abc 01> cec test2.eqn out2.eqn
Networks are equivalent.  Time =     0.00 sec
abc 01> cec test3.eqn out3.eqn
Networks are equivalent.  Time =     0.00 sec
```

VI. Reference

[1] 陳聿廣教授自編上課用講義

[2] C++ std::sort 排序用法與範例完整介紹
https://shengyu7697.github.io/std-sort/

[3] vector 心得整理

https://edisonx.pixnet.net/blog/post/34345257

[4] vector 記憶體分配和回收機制

https://codertw.com/%E7%A8%8B%E5%BC%8F%E8%AA%9E%E8%A8%80/605315/