# EE6094

# CAD for VLSI Design

# Programming Assignment III Report

Student Name:李庭緯

Student ID:106303010

## Logic Optimizer using Quine-McCluskey and Petrick_Method

## Introduction

A Lib base on

1. Quine–McCluskey algorithm

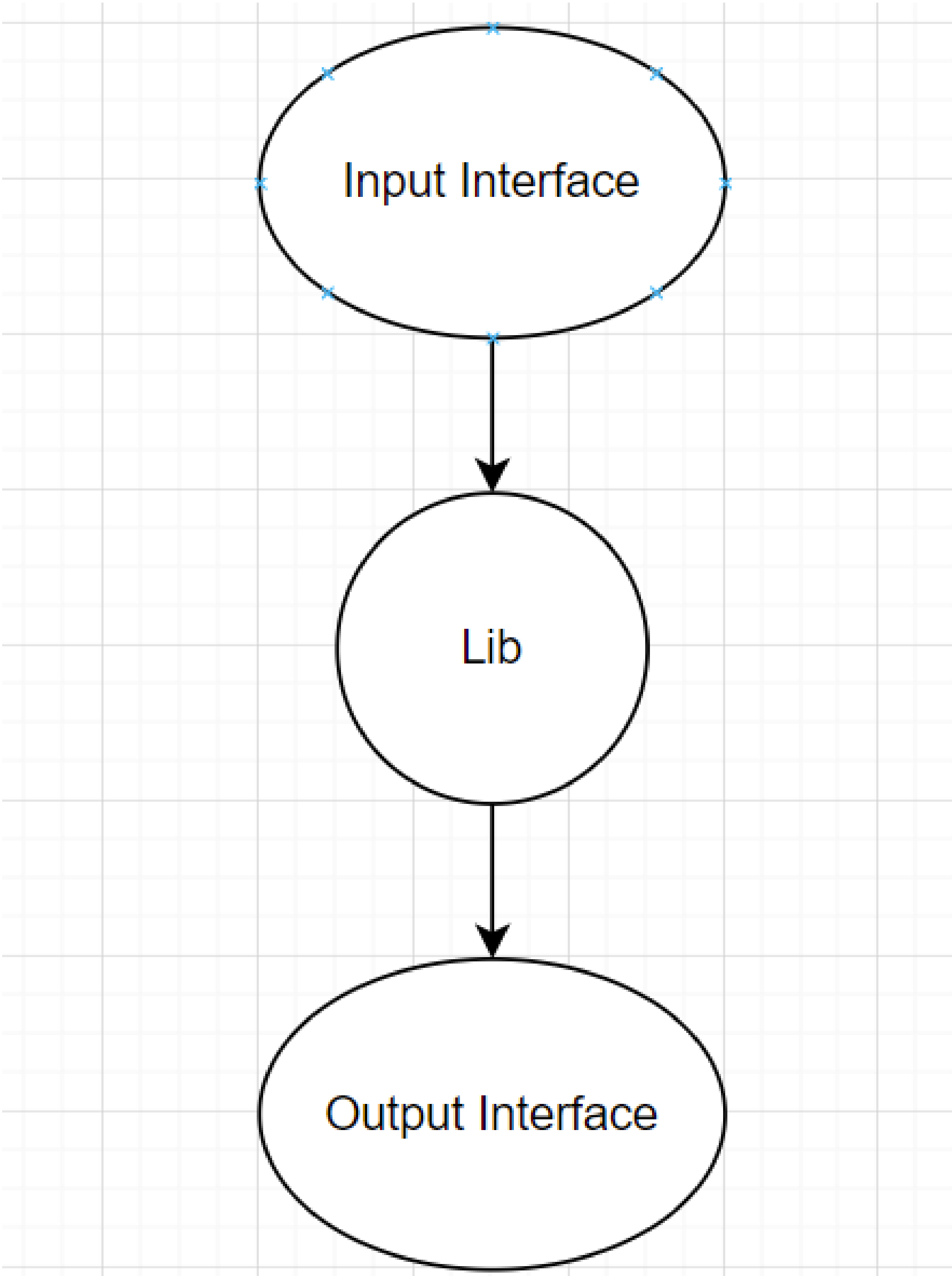
2. Petrick's method

I optimize the prime_implicant finding part,And it can be validated in simple way. check Implicant

Result in workstation:

```
[106303010@eda359_forclass Logic_Optimizer]$ ./main testcase/test.eqn ans1.eqn
[106303010@eda359_forclass Logic_Optimizer]$ ./main testcase/test2.eqn ans2.eqn
[106303010@eda359_forclass Logic_Optimizer]$ ./main testcase/test3.eqn ans3.eqn
[106303010@eda359_forclass Logic_Optimizer]$ ./abc
UC Berkeley, ABC 1.01 (compiled Apr 30 2021 13:34:44)
abc 01> cec testcase/test.eqn ans1.eqn
Networks are equivalent.  Time =     0.01 sec
abc 01> cec testcase/test2.eqn ans2.eqn
Networks are equivalent.  Time =     0.00 sec
abc 01> cec testcase/test3.eqn ans3.eqn
Networks are equivalent.  Time =     0.00 sec
abc 01>
```

### *IO*Interface

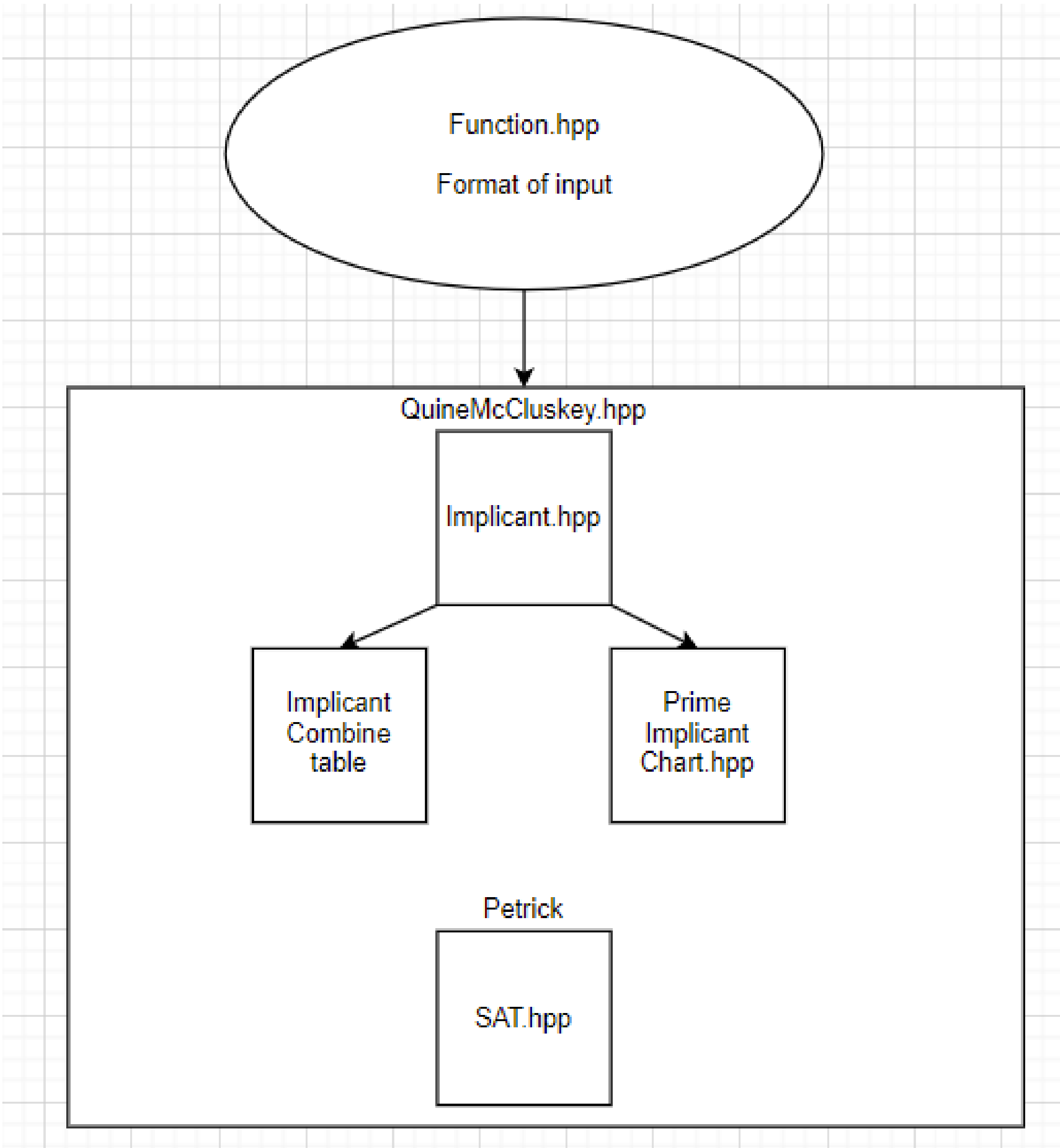



You can use your own I/O interface if you want.I write example of both, check Input_Interface

Input_Example

and Output_Interface
Output_Example

**LibArchitecture**



## HowToUse

Main.cpp has two example in img

```
$ make
g++  -std=c++11 -c lib/Implicant.cpp -o lib/Implicant.o
g++  -std=c++11 -c lib/QuineMcCluskey.cpp -o lib/QuineMcCluskey.o
g++  -std=c++11 -c lib/Prime_Implicant_Chart.cpp -o lib/Prime_Implicant
_Chart.o
g++  -std=c++11 -c lib/SAT.cpp -o lib/SAT.o
g++  -std=c++11 lib/Implicant.o lib/QuineMcCluskey.o lib/Prime_Implican
t_Chart.o lib/SAT.o main.cpp -o main
```

```
$ ./main.exe
(a*!d)+(b*!c*!d)+(a*c)

(!a*!b*!c*!d)+(b*d)+(b*c)+(a*!c*d)+(a*c*!d)
```

## The Flow

If you have don't care : (Example1)

```
 //---------------------------------------------------INPUT------------
------------------------------------------------
 unsigned int Fan_in  = 4;
 std::string F_str = "(!a*b*!c*!d)+(a*!b*!c*!d)+(a*!b*c*!d)+(a*!b*c*d)+
(a*b*!c*!d)+(a*b*c*d)";
 std::string Dont_Care_str = "(a*!b*!c*d)+(a*b*c*!d)";
 Function F = get_function(F_str,Fan_in);
 Function Dont_Care = get_function(Dont_Care_str,Fan_in);
 ////---------------------------------------------------QuineMcCluskey-
-----------------------------------------------------------
 auto implicants = QuineMcCluskey(F,Dont_Care);

 ///---------------------------------------------------Output----------
---------------------------------------------------------
 print(implicants,Fan_in);
```

NO Don't care Example (Example2)

```
unsigned int Fan_in2 = 4;
std::string F2_str = "(!a*!b*!c*!d)+(!a*b*!c*d)+(!a*b*c*!d)+(a*!b*!c*d)
+(a*!b*c*!d)+(a*b*!c*d)+(a*b*c*!d)+(a*b*c*d)+(!a*b*c*d)";
Function F2 = get_function(F2_str,Fan_in2);

auto implicants2 = QuineMcCluskey(F2);

print(implicants2,Fan_in2);
```

## FolderStructure

```
Logic_Optimizer
|
|----Makefile
|----main.cpp
|----README.md
|----include
|        |----Input/Input.hpp
|        |----Output/Output.hpp
|        |----QuineMcCluskey/QuineMcCluskey.hpp
|
|----src
|      |----Function.hpp
|      |----Implicant.hpp
|      |----Implicant.cpp
|      |----Prime_Implicant_Chart.hpp
|      |----Prime_Implicant_Chart.cpp
|      |----QuineMcCluskey.cpp
|      |----SAT.hpp
|      |----SAT.cpp
|
|
|----tests
|       |----Makefile
|       |----Input/Input.cpp
|       |----Output/Output.cpp
|       |----Prime_generate/Prime_generate.cpp
|       |----Prime_Implicant_Chart/Prime_Implicant_Chart.cpp
|       |----QuineMcCluskey/QuineMcCluskey.cpp
|       |----SAT/SAT.cpp
|       |----README.md
|       |----Demo_Example.png
|
|----doc
|      |----CAD_PA3_Logic_Opt.pdf
|      |----Optimization of the Quine-McCluskey Method for the Minimizat
ion.pdf
|
|----img
```

## DataStructure

### Function

A standard function format in all Lib

```
using Function = std::vector<std::vector<unsigned int>>;
```

The format example:

```
Function F = {
        {},     // has zero 1 , if empty,still need a {}
        {4,8},    // has one 1
        {9,10,12},  //has two 1
        {11,14},  //has three 1
        {15} };  //has four 1
```

| Number of 1s | Minterm | Binary Representation |
|---|---|---|
| 1 | m4 | 0100 |
| | m8 | 1000 |
| 2 | (m9) | 1001 |
| | m10 | 1010 |
| | m12 | 1100 |
| 3 | m11 | 1011 |
| | (m14) | 1110 |
| 4 | m15 | 1111 |

## Implicant

A kernel part in generating prime implicant by combination.

Constructor

```
 Implicant(type v,type dash_part = 0)
       :val{v},cover{dash_part}
      {}
```

Important data members

```
type val; // val without consider dash_part, for example :  0b10-1  mea
ns   0b1001 + 0b00-0(dash-part), and val = 0b1001 = 9
type cover; // use to store dash part, for example  0b10-1 means cover
= 0b0010(dash-part) = 2
```

take the same example :

| Number of 1s | Minterm | 0-Cube | Size 2 Implicants | | Size 4 Implicants | |
|---|---|---|---|---|---|---|
| 1 | m4 | 0100 | m(4,12) | -100* | m(8,9,10,11) | 10--* |
| | m8 | 1000 | m(8,9) | 100- | m(8,10,12,14) | 1--0* |
| | — | — | m(8,10) | 10-0 | — | |
| | — | — | m(8,12) | 1-00 | — | |
| 2 | m9 | 1001 | m(9,11) | 10-1 | m(10,11,14,15) | 1-1-* |
| | m10 | 1010 | m(10,11) | 101- | — | |
| | — | — | m(10,14) | 1-10 | — | |
| | m12 | 1100 | m(12,14) | 11-0 | — | |
| 3 | m11 | 1011 | m(11,15) | 1-11 | — | |
| | m14 | 1110 | m(14,15) | 111- | — | |
| 4 | m15 | 1111 | | — | — | |

```
m4    0100  : val = 0100 (base part:4)  , cover = 0000 (dash part : 0)
               so it can cover m4
m(4,12)     -100* : val = 100 (base part : 4) , cover : -000 (dash part
 : 8)                so it can cover m4,m12 ( base + 1000)
m(8,9,10,11)      10--   :  val = 1000 (base part : 8) , cover : 00-- (d
ash part : 3)     so it can cover m8,m9(base + 0001),m10(base + 0010),m1
1(base + 0011)
```

Implicants can combine each other iff

1. differ by 1 bit.

2. has same dash part : cover must be same.

```
bool diff_one_bit(const Implicant&I1,const Implicant&I2)
{
    if(I1.get_cover()!=I2.get_cover())return false;//different cover

    unsigned int diff = I1.get_val() ^ I2.get_val();//do xor , we can g
et difference

    return ((diff)&(diff-1))==0;//diff is power of 2
}
```

With cover , we can compare two implicants without iterating all characters and can check fastly.

**We can use cover and the val to combination all min term this implicant can cover.**
See Implicant.cpp-Implicant::get*cover*terms()

### Implicant*Combine*table

It is defined in QuineMcCluskey.hpp

```
using Implicant_Combine_table = std::vector<std::map <Implicant,bool>>;
//use in phase 1 : find prime_implicants
```

| Number of 1s | Minterm | 0-Cube | Size 2 Implicants | | Size 4 Implicants | |
|---|---|---|---|---|---|---|
| 1 | m4 | 0100 | m(4,12) | -100* | m(8,9,10,11) | 10--* |
| | m8 | 1000 | m(8,9) | 100- | m(8,10,12,14) | 1--0* |
| | — | — | m(8,10) | 10-0 | — | |
| | — | — | m(8,12) | 1-00 | — | |
| 2 | m9 | 1001 | m(9,11) | 10-1 | m(10,11,14,15) | 1-1-* |
| | m10 | 1010 | m(10,11) | 101- | — | |
| | — | — | m(10,14) | 1-10 | — | |
| | m12 | 1100 | m(12,14) | 11-0 | — | |
| 3 | m11 | 1011 | m(11,15) | 1-11 | — | |
| | m14 | 1110 | m(14,15) | 111- | — | |
| 4 | m15 | 1111 | | — | — | |

I use std::map <Implicant,bool> for two reasons

```
1. The new implicant durring combining may duplicate.
2. The implcant in table need a record to indentify whether it is  a pr
ime implicant (*).
```

### Prime*Implicant*Chart

This chart describe the relations between min_terms in function and prime implicants which be generated in phase1.

minterms

prime implicants

| | 4 | 8 | 10 | 11 | 12 | 15 | ⇒ | A | B | C | D |
|---|---|---|---|---|---|---|---|---|---|---|---|
| m(4,12)* | ✓ | | | | ✓ | | ⇒ | — | 1 | 0 | 0 |
| m(8,9,10,11) | | ✓ | ✓ | ✓ | | | ⇒ | 1 | 0 | — | — |
| m(8,10,12,14) | | ✓ | ✓ | | ✓ | | ⇒ | 1 | — | — | 0 |
| m(10,11,14,15)* | | | ✓ | ✓ | | ✓ | ⇒ | 1 | — | 1 | — |

With Implicant::get_cover_terms(),It is easy to list the minterms that can be cover of each prime implicant.

See Prime*Implicant*Chart::draw

Three important data members

1. std::vector<min_term>Min_term_vec;//each minterm save the prime_index that can cover this min_term.
2. std::vector<std::vector<int>>Prime_vec;//each prime save the Min_term index that can be cover by this prime.
3. std::map<unsigned int ,unsigned int>term_index_mapping;

//term_val in function f is not start from 1, and it may be unorderd,so we need a way to record the min_term's index.
//use to map min_term's value into index in Min_term_vec

The Prime*Implicant*Chart's constructor do two things.

```
Prime_Implicant_Chart(const Function &f,const std::vector<Implicant>&prime)
    {
        init_table(f,prime);
        draw(prime);
    }
```

1. init_table:construct all minterms and prime implicants in this table

2. draw : draw the relations between minterms and prime implicants using Implicant::get_cover_terms()

After constructor , important member functions are

private:
1. Find_Essential()   : scan the table and return the Essential prime implicants (ESPIs)    It execute only when std::vector<unsigned int> Essential_prime is empty.

public:
2. cover_terms_by_ESPI()  : automatically call Find_Essential() and marked the min_terms coverd by these ESPIs.
3. const std::vector<unsigned int>& get_Essential_prime(); :automatically call Find_Essential() and return Essentail prime implicants.
4. std::vector<min_term>get_un_converd_Min_term()const  : Return the min_terms which are not marked.

After calling `cover_terms_by_ESPI()` , we can use `get_un_converd_Min_term()` to get the remain minterms and change it to SAT problem(Petrick's method).

## SAT_interface

After `cover_terms_by_ESPI()`,the next step is solve the SAT problem of the remaining minterms.
In this example

| | 4 | 8 | 10 | 11 | 12 | 15 | ⇒ | A | B | C | D |
|---|---|---|---|---|---|---|---|---|---|---|---|
| m(4,12)* | ✓ | | | | ✓ | | ⇒ | — | 1 | 0 | 0 |
| m(8,9,10,11) | | ✓ | ✓ | ✓ | | | ⇒ | 1 | 0 | — | — |
| m(8,10,12,14) | | ✓ | ✓ | | ✓ | | ⇒ | 1 | — | — | 0 |
| m(10,11,14,15)* | | | ✓ | ✓ | | ✓ | ⇒ | 1 | — | 1 | — |

P0 : m(4,12)
P1 : m(8,9,10,11)
P2 : m(8,10,12,14)
P3 : m(10,11,14,15)

P0 and P3 are ESPIs. we need choose both of them.

SO,the problem covering m8 can be changed into the problem
(P1 + P2) = 1

So,We need to

1. generate all the bracket
2. Use backtracking to solve this problem

In order to generate all the bracket and make the same literal's val(true/false) can be change at same time "in all brackets",we need to use global vars.
Instead of using global vars, SAT use `std::vector<literal>literals;` to store literals(P0,P1,P2,P3) and using `bracket = std::vector<int>;` to store the litera's index in `std::vector<literal>literals;`

The interface:

```
void SAT::add_bracket(const bracket &br)
{
    bracket new_br;
    new_br.reserve(br.size());
```

```
    for(auto l : br)
    {
        if(lit_id.find(l)==lit_id.end())//l is a new literal
        {
            lit_id.insert({l,literals.size()});
            literals.push_back({l});
        }
        new_br.push_back(lit_id[l]);//put index of l in std::vector<lit
eral>literals into bracket. used in bool SAT::evaulate_one_bracket(cons
t bracket& br).
    }
    brackets.push_back(new_br);
}
```

How to use this interface :
std::vectorPetrick*Method(Prime*Implicant*Chart &table,size*t remain*prime*num,size*t max*bracket_num)

```
std::vector<int>Petrick_Method(Prime_Implicant_Chart &table,size_t rema
in_prime_num,size_t max_bracket_num)
{

    SAT sat{max_bracket_num,remain_prime_num};
    for(auto &m : table.get_un_converd_Min_term())
    {
        //change each minterm into one bracket form.

        SAT::bracket br;
        br.reserve(m.get_prime_index().size());
        for(auto p_i : m.get_prime_index())
        {
            br.push_back(p_i);
        }

        sat.add_bracket(br);//add this min_term's bracket into SAT prob
lem
    }

    return sat.min_cover_SAT();//use sat to solve this problem.each ele
ment in return is a prime_implicant's index.
}
```

## Tests

tests

$$f(A,B,C,D) = \sum m(4,8,10,11,12,15) + d(9,14).$$

```
Function F = {
        {},     // has zero 1 , if empty,still need a {}
        {4,8},    // has one 1
        {10,12},  //has two 1
        {11},  //has three 1
        {15} };  //has four 1

Function Dont_care = {
        {},
        {},
        {9},
        {14}
};
```

## Prime_generate

```
Term value|Cover

         4|    0
         8|    0

         9|    0
        10|    0
        12|    0

        11|    0
        14|    0

        15|    0

Term value|Cover

         4|    8
         8|    1
         8|    2
         8|    4

         9|    2
        10|    1
        10|    4
        12|    2
```

```
        11|    4
        14|    1

Term value|Cover

         8|    3
         8|    6

        10|    5

Prime implicants :
val = 4 , cover = 8
m(12,4)
val = 8 , cover = 3
m(11,9,10,8)
val = 8 , cover = 6
m(14,10,12,8)
val = 10 , cover = 5
m(15,11,14,10)
```

| Number of 1s | Minterm | 0-Cube | Size 2 Implicants | | Size 4 Implicants | |
|---|---|---|---|---|---|---|
| 1 | m4 | 0100 | m(4,12) | -100* | m(8,9,10,11) | 10--* |
| | m8 | 1000 | m(8,9) | 100- | m(8,10,12,14) | 1--0* |
| | — | — | m(8,10) | 10-0 | — | |
| | — | — | m(8,12) | 1-00 | — | |
| 2 | m9 | 1001 | m(9,11) | 10-1 | m(10,11,14,15) | 1-1-* |
| | m10 | 1010 | m(10,11) | 101- | — | |
| | — | — | m(10,14) | 1-10 | — | |
| | m12 | 1100 | m(12,14) | 11-0 | — | |
| 3 | m11 | 1011 | m(11,15) | 1-11 | — | |
| | m14 | 1110 | m(14,15) | 111- | — | |
| 4 | m15 | 1111 | | — | — | |

## Prime*Implicant*Chart

```
prime implicants are
0 : m(12,4)
1 : m(11,9,10,8)
2 : m(14,10,12,8)
3 : m(15,11,14,10)

create table :
m4 : 0
m8 : 1 2
m10 : 1 2 3
m12 : 0 2
m11 : 1 3
m15 : 3

ESPI index : 0 3

After use ESPI to cover terms :
m8 : 1 2


Use SAT to choose remain prime implicants
choose 2
The final ans is :
p2 p0 p3
```

| | 4 | 8 | 10 | 11 | 12 | 15 | ⇒ | A | B | C | D |
|---|---|---|---|---|---|---|---|---|---|---|---|
| m(4,12)* | ✓ | | | | ✓ | | ⇒ | — | 1 | 0 | 0 |
| m(8,9,10,11) | | ✓ | ✓ | ✓ | | | ⇒ | 1 | 0 | — | — |
| m(8,10,12,14) | | ✓ | ✓ | | ✓ | | ⇒ | 1 | — | — | 0 |
| m(10,11,14,15)* | | | ✓ | ✓ | | ✓ | ⇒ | 1 | — | 1 | — |

## SAT

```
std::vector<SAT::bracket> brs = {
        {1,6},   //(p1+p6)
        {6,7},   //(p6+p7)
        {6},     //(p6)
        {2,3,4}, //(p2+p3+p4)
        {3,5},   //(p3+p5)
        {4},    //(p4)
        {5,7}   //(p5+p7)
    };
```

```
the minimum solution is :
p6 = 1
p4 = 1
p5 = 1
```

## QuineMcCluskey

```
    Function F = {
            {},     // has zero 1 , if empty,still need a {}
            {4,8},    // has one 1
            {10,12},  //has two 1
            {11},  //has three 1
            {15} };  //has four 1

    Function Dont_care = {
            {},
            {},
            {9},
            {14}
    };
```

```
implicants :
val = 8 dash = 6 can cover :m(14,10,12,8)
val = 4 dash = 8 can cover :m(12,4)
val = 10 dash = 5 can cover :m(15,11,14,10)
ans is correct!!
```