**EE677 VLSI CAD**

(Course Project)

Conversion of Logic Expressions into Canonical Form

-by-  
Ajinkya Gorad (140110033)  
Shruti Hiray  
Himani Prajapati

Contents

[Conversion of Logic Expressions into Canonical Form 3](#_Toc467222609)

[Implementation 3](#_Toc467222610)

[Generation of Prime Implicants 3](#_Toc467222611)

[Minimization by Quine-McCluskey 3](#_Toc467222612)

[Code 5](#_Toc467222613)

[Generation of Prime Implicants 5](#_Toc467222614)

[Minimization of Quine-McCluskey 5](#_Toc467222615)

[File ‘TestMcCluskey.py’ 5](#_Toc467222616)

[File ‘helper.py’ 6](#_Toc467222617)

[Interconversion between SOP, POS and ESOP forms 9](#_Toc467222618)

# **Conversion of Logic Expressions into Canonical Form**

The task is to be able to reduce the given logical expression in the tree format to its canonical forms of Sum of Product (SOP), Product of Sums (POS) or Exclusive-Or Sum of Products (ESOP) partly using PYEDA (Python Electronic Design Automation Tool.

It involves 3 stages of implementation

* Efficient generation of on Minterms(Shannon Product Method)
* Minimization by Quine-McCluskey tabular algorithm to get SOP form
* Interconvert SOP expressions to POS and ESOP forms

# Implementation

## Generation of Prime Implicants

## Minimization by Quine-McCluskey

Minimization by Quine McCluksey is done without using PyEDA facility. Input to the algorithm are the on-Minterms generated above. For more detailed self-explanation follow code.

In this example we have used Boolean Function f as a pyEDA expression:

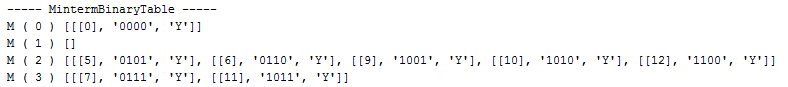
  
Fig. Input

Each minterm is converted into its string binary representation with number of bits (Nbits) calculated by maximum value in On-Minterms by routine ***minterm2BinString.*** In our example it is 4.

The format of representing a minterm or any of its reduced form is

* Supporting minterms says what ‘XX..X’ is made of
* X’s are the binary values and can be ‘1’, ‘0’, or ‘-‘ (don’t care)
* Tag depicts the necessity of minterm in the Prime Implicant (marking checked or required minterms for the next iteration. ***By default these are ‘N’***

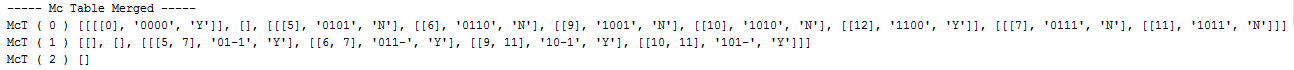
The Function ***McCluskey*** is an overall implementation of algorithm with all routines packed in one method. After minterms are converted to string they are sorted in number of their ones (‘1’s) in the table called ***P*** by the routine ***sortNumOne.***



*Fig. Sorted On-Minterms (P)*

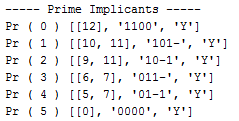
***dispX*** displays the sorted table for convenience. Now we create the ***McTable*** for our reduction using tabular method.

***mergeTable*** implements the laborious task of checking the minterms or sum of minterms (terms) which were previously merged and creates the entire table and updates result in *McTable.* Note that the ***mergeTable***is a ***recursive algorithm*** and time complexity of this algorithm. Now the minterms marked with ***‘Y’*** are the required prime implicants, however may not be essential. This is done by ***getPrimes*** method in **primes.**



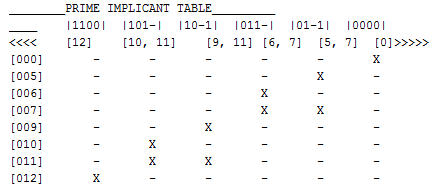
*Fig. Merged evaluated Table*

Now to construct the Prime Implicant table for finding the Essential Prime Implicants, we first sort the reduced required terms with their composite On-Minterms. This creates map from On-Minterm to prime Implicants. This is done through routine ***getS.***



*Fig. Prime Implicants*

***displayPrimeImplicantTable*** displays the Prime Implicant table generated in pretty format like:



*Fig. Prime Implicant Table*

Now to find the essential Prime Implicants, ***tableReduce*** reduces the Prime Implicant table and also displays the progress on Table on Each Iteration. It continues to reduce until the table gets to NULL. It does not use recursion. For reduction of table it uses Row-Column Dominance method. Firstly a Dominating Row is removed and then Dominated Column, and this is done by methods ***removeDominatingRow*** & ***removeDominatedColumn.*** The removed entries from table are appended into a list of essential prime implicants. Note that it gives only a set of essential prime implicants and not all the possibilities, hence the solution may or may not be an optimal solution.

Now we have our essential Primes in binary format

  
*Fig. Result*

The result is converted into readable format using routine ***prime2Str*** , Here’s the Final Result

  
Fig. Final Result

# Code

## Generation of Prime Implicants

## Minimization of Quine-McCluskey

### File ‘TestMcCluskey.py’

|  |
| --- |
| **from** pyeda.inter **import** \* **from** helper **import** \* **from** math **import** \* **def** McCluskey(m,Nbits):   print(**"\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Mc CLUSKEY\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_"**)  print(**"MINTERM INPUT (BITS :"**,Nbits,**"): "**,m)  mintermBinary = minterm2BinString(m,Nbits) *# convert minterms into their binary strings* P = sortNumOne(mintermBinary) *# create the table P with minterm ( STORE PRIME TO MINTERM MAPPING)* dispX(P,**"M"**,**"MintermBinaryTable"**)  McTable = [P]  mergeTable(McTable,0) *# merge and find all possible minterms* dispX(McTable,**"McT"**,**"Mc Table Merged"**)  primes = getPrimes(McTable) *# get the primes from the table ( values marked as Y)* dispX(primes,**"Pr"**,**"Prime Implicants"**)  S = getS(m,primes) *# sort the inverse of STORE MINTERM TO PRIME MAPPING* dispX(S,**"S"**,**"MINTERM TO PRIME MAPPING (INPUT)"**)  print(**"\_\_\_\_\_\_\_\_PRIME IMPLICANT TABLE\_\_\_\_\_\_\_\_\_"**)  displayPrimeImplicantTable(m, primes)  essentialPrimes = tableReduce(m, primes)  print(**"Essential Prime Implicants :"**,essentialPrimes)  **return** essentialPrimes **def** getEssentialImplicants(f,X):  f\_bar = ~f *# complement the function to get the minterms* n = len(X) *# number of variables  # SHANNON METHOD FOR FINDING MINTERMS* m=**''** M =[]  shannon(f\_bar,X,0,n,m,M)  minterms = [ int(x,2) **for** x **in** M]  print(**"\_\_\_\_\_\_\_\_\_\_\_\_\_\_SHANNON EXPANSION\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_"**)  print(**"Expression :"**,f)  print(**"Minterms :"**,minterms)  m = minterms  *# MCCLUSKY ALGORITHM FOR REDUCING ON MINTERMS AND FINDING PRIME IMPLICANTS TO ESSENTIAL PRIME IMPLICANTS* m = list(set(m)) *# make sure all minterms are unique* Nbits = ceil(log(max(m)+1,2)) *# Get number of Bits to work with* essentialPrimes = McCluskey(m,Nbits) *# call McCluksy Function for getting essential prime implicants* **return** essentialPrimes   *#-----------SHANNON--------------* n =4 X = exprvars(**'x'**, n)  f = expr(**'~((x[1]^x[0])^(x[2]&x[3]))'**) ep = getEssentialImplicants(f,X)  print(**'SOP:'**,prime2Str(ep)) |

### File ‘helper.py’

|  |
| --- |
| **from** pyeda.inter **import** \* zero = expr(0) one = expr(1) *#-------------DEFINATIONS------------* **def** displayPrimeImplicantTable(m,P):  **if**(len(P)==0):  print(**"EMPTY TABLE"**)  **return** line =**"\_\_\_\_"** fStr = **'{0:03d}'  for** k **in** range(0,len(P)):  line =line+ **"\t|"**+P[k][1]+**"|"** print(line)  line = **"<<<<"  for** k **in** range(0,len(P)):  line = line+**"\t"**+str(P[k][0])  line = line+**">>>>>"** print(line)  **for** y **in** range(0,len(m)):  line = **"["** + fStr.format(m[y]) + **"]"  for** x **in** range(0,len(P)):  line = line+**"\t\t"  if**(m[y] **in** P[x][0]):  line = line+**"X"  else**:  line = line+**"-"** print(line) **def** getS(m,P):  **return** list(zip(m, primesOfminterm(m, P))) **def** getP(S):  **return** mintermsOfprime(S) **def** getm(S):  **return** [s[0] **for** s **in** S] **def** removeDominatingRow(m,P):  S = getS(m,P)  print(**"Sr"**)  dispP(S)  **for** s **in** S:  **for** s2 **in** S:  **if**(s!=s2):  **if**(set(s[1])>set(s2[1])):  **if**(s **in** S):  S.remove(s)  P = getP(S)  m = getm(S)  **return** m,P **def** removeDominatedCol(m,P):  dispP(P)  **for** p **in** P:  **for** p2 **in** P:  **if** (p != p2):  **if** (set(p[0]) >= set(p2[0])):  P.remove(p)  **return** m,P   **def** tableReduce(m,P):  S = getS(m, P) *# S gets modified in loop* essentialPrimes = []  iter = 0  print(**"Prime Implicant Table"**)  displayPrimeImplicantTable(m,P)  **while**(len(P)!=0):  iter = iter+1  print(**"[ITERATION TABLE REDUCE : "**,iter,**"]"**)  *#print(S)* **for** s **in** S:  **if**(len(s[1])==1):  essentialPrimes.append(s)  **if**(s[0] **in** m):  m.remove(s[0])  sTrash=[s **for** s **in** S **for** ep **in** essentialPrimes **if** ep[1][0] **in** s[1] ] *# remove the minterm common primes  # remove minterms corresponding to particular* S = [s **for** s **in** S **if** s **not in** sTrash]  P = getP(S)  m = getm(S)  print(**"Removed essential Prime Implicants"**)  displayPrimeImplicantTable(m,P)  m,P=removeDominatingRow(m,P)  print(**"Removed Dominating Rows in Prime Implicant Table"**)  displayPrimeImplicantTable(m,P)  m,P=removeDominatedCol(m,P)  print(**"Removed Dominated Columns"**)  displayPrimeImplicantTable(m,P)  S = getS(m,P)  ep = [e[1][0] **for** e **in** essentialPrimes]  **return** ep **def** primesOfminterm(m,primes):  S = []  **for** mi **in** m:  S.append([minterm[1] **for** minterm **in** primes **if** mi **in** minterm[0]])  **return** S **def** mintermsOfprime(S):  m = [ s[0] **for** s **in** S ]  primes = []  **for** s **in** S:  **for** p **in** s[1]:  **if**(p **not in** primes):  primes.append(p)  P =[]  **for** p **in** primes:  P.append( [[s[0] **for** s **in** S **if** p **in** s[1]],p,**'R'**] )  **return** P *# renamed from mintermType to mineterm2BinString* **def** minterm2BinString(m,Nbits):  formatString = **'{0:0'** + str(Nbits) + **'b}'** onMinterm = []  **for** minterm **in** m:  bin = formatString.format(minterm)  onMinterm.append([[minterm], bin, **'Y'**]) *# mark all tags as yes initially* **return** onMinterm **def** checkDominance(a,b):  **if**(a==b): **return "="** a\_ = [int(x) **for** x **in** list(a)]  b\_ = [int(x) **for** x **in** list(b)]  c = [ x\*y **for** x,y **in** zip(a\_,b\_)]  **if** c==b\_ : **return ">"  if** c==a\_ : **return "<"  return None def** dispP(P):  **for** ip **in** range(0, len(P)):  print(**"P("**, ip, **")"**, P[ip]) **def** dispT(T):  **for** it **in** range(0,len(T)):  **for** ip **in** range(0,len(T[it])):  print(**"T("**,it,**","**,ip,**")|"**,T[it][ip]) **def** dispX(X,L,title):  print(**"-----"**,title,**"-----"**)  **for** ip **in** range(0, len(X)):  print(L,**"("**, ip, **")"**, X[ip]) **def** ifDifferByOne(m1,m2):  **if**(len([(x,y) **for** x,y **in** zip(m1[1],m2[1]) **if** x!=y])==1):  **return True  return False def** getDifferingIndex(m,mn):  **for** i **in** range(0,len(m[1])):  **if**(m[1][i]!=mn[1][i]):  **return** i  **return None def** mergeTable(table,id):  *#print("called MergeTable with table:",id)  #dispT(table)* P1 = table[id]  Pnew = []  found = **False  for** ip **in** range(0,len(P1)-1):  **for** m **in** P1[ip]:  **for** mn **in** P1[ip+1]:  **if**(ifDifferByOne(m,mn)):  diff = getDifferingIndex(m,mn)  **if**((m[1][diff]!=**'-'**)|(mn[1][diff]!=**'-'**)):  found = **True** m\_new =list(m)  m\_new[0] = list(m[0])+list(mn[0])  m\_new[0] = list(set(m\_new[0]))  m\_new[1] = m\_new[1][0:diff]+**'-'**+m\_new[1][diff+1:]  m\_new[2] = **'Y'** *# a newly generated term always as 'Y'* m[2]=**'N'** *# check them as 'N' as they are included at least once* mn[2]=**'N'  if** m\_new **not in** Pnew: *# add only if it doesn't exist* Pnew.append(m\_new)  *#print(m,"+",mn,"=",m\_new)* Pnew = sortNumOne(Pnew)  table.append(Pnew)  **if**(found):  mergeTable(table,id+1) *# use recursion* **return def** getPrimes(table):  primes = []  **for** col **in** table:  **for** minterms **in** col:  **for** minterm **in** minterms:  **if**(minterm[2]==**'Y'**):  minterm[0].sort()  primes.append(minterm)  **return** sortSet(primes) *# return the minterms haveing number of ones* **def** getMinterms(minterms,NumOne):  mi = []  **for** m **in** minterms:  **if**(m[1].count(**'1'**)==NumOne):  mi.append(m)  **return** mi **def** uniq(lst):  last = object()  **for** item **in** lst:  **if** item == last:  **continue  yield** item  last = item **def** sortSet(l):  **return** list(uniq(sorted(l, reverse=**True**))) **def** sortNumOne(minterms):  P=[]  NumOne = 0;  **while**(len(minterms)):  mi = getMinterms(minterms,NumOne)  *#print('mi : ',mi)* P.append(mi)  minterms = [m **for** m **in** minterms **if** m **not in** mi] *# remove sorted entries  #print("minterms:",len(minterms),"|",minterms)* NumOne = NumOne+1  **return** P   **def** shannon(f,X,i,n,m,M):  **if**(i == n):  **return** f1\_bar = f.restrict({X[i]:zero})  f1 = f.restrict({X[i]:one})  **if**((f1\_bar == one)&(i==n-1)):  M.append(m+**'1'**)  **else**:  shannon(f1\_bar|expr(~X[i]),X,i+1,n,m+**'1'**,M)  **if**((f1 == one)&(i==n-1)):  M.append(m+**'0'**)  **else**:  shannon(f1|expr(X[i]),X,i+1,n,m+**'0'**,M)  **return** *# To Convert the essential primes to string equivalent* **def** prime2Str(p):  SOP = []  **for** prime **in** p:  n = len(prime)  primeStr=**''  for** i **in** range(0,n):  **if**(prime[i]==**'1'**):  primeStr = **'x'**+str(i)+**'.'**+primeStr  **if**(prime[i]==**'0'**):  primeStr = **'~x'**+str(i)+**'.'**+primeStr  primeStr = primeStr[:-1]  SOP.append(primeStr)  **return** SOP |

## Interconversion between SOP, POS and ESOP forms