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

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.

$$f = \Sigma(m1, m2, \dots mN)$$

$$then\ the\ array\ \ of\ On - Minterms\ (m)\ = [m1, m2, \dots, mN]$$

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

$$f = \sim((x1 \oplus x0) \oplus (x2 \& x3))$$

```
_____________SHANNON EXPANSION_____________
Expression : Not(Xor(x[0], x[1], And(x[2], x[3])))
Minterms : [12, 11, 10, 9, 7, 6, 5, 0]
```

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.

$$N_{bits} = ceil(\log_2(\max(m) + 1))$$

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

$$([\ supporting\ minterms], 'XX..X', tag)$$

- 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.***

$$\mathrm{P} = [\ \text{minterms with \#1} = 0, \text{minterms with \#1} = 1, \dots]$$

```
----- MintermBinaryTable -----
M ( 0 ) [[[0], '0000', 'Y']]
M ( 1 ) []
M ( 2 ) [[[5], '0101', 'Y'], [[6], '0110', 'Y'], [[9], '1001', 'Y'], [[10], '1010', 'Y'], [[12], '1100', 'Y']]
M ( 3 ) [[[7], '0111', 'Y'], [[11], '1011', 'Y']]
```

Fig. Sorted On-Minterms (P)

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

$$McTable = [P]$$

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.**

```
----- Mc Table Merged -----
McT ( 0 ) [[[[0], '0000', 'Y']], [], [[[5], '0101', 'N'], [[6], '0110', 'N'], [[9], '1001', 'N'], [[10], '1010', 'N'], [[12], '1100', 'Y']], [[[7], '0111', 'N'], [[11], '1011', 'N']]]
McT ( 1 ) [[], [], [[[5, 7], '01-1', 'Y'], [[6, 7], '011-', 'Y'], [[9, 11], '10-1', 'Y'], [[10, 11], '101-', 'Y']]]
McT ( 2 ) []
```

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.***

```
----- Prime Implicants -----
Pr ( 0 ) [[12], '1100', 'Y']
Pr ( 1 ) [[10, 11], '101-', 'Y']
Pr ( 2 ) [[9, 11], '10-1', 'Y']
Pr ( 3 ) [[6, 7], '011-', 'Y']
Pr ( 4 ) [[5, 7], '01-1', 'Y']
Pr ( 5 ) [[0], '0000', 'Y']
```

Fig. Prime Implicants

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

```
________PRIME IMPLICANT TABLE_________
____    |1100|  |101-|  |10-1|  |011-|  |01-1|  |0000|
<<<<    [12]    [10, 11]    [9, 11] [6, 7]  [5, 7]  [0]>>>>>
[000]       -       -       -       -       -       X
[005]       -       -       -       -       X       -
[006]       -       -       -       X       -       -
[007]       -       -       -       X       X       -
[009]       -       -       X       -       -       -
[010]       -       X       -       -       -       -
[011]       -       X       X       -       -       -
[012]       X       -       -       -       -       -
```

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

```
Essential Prime Implicants : ['0000', '01-1', '011-', '10-1', '101-', '1100']
```

Fig. Result

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

```
SOP: ['~x3.~x2.~x1.~x0', 'x3.x1.~x0', 'x2.x1.~x0', 'x3.~x1.x0', 'x2.~x1.x0', '~x3.~x2.x1.x0']
```

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