Computing Gröbner Basis of Combinational Circuits using ZBDDs

Arjun Deb: u0736104@utah.edu
Rachitha Chandrashekar: u0942921@utah.edu
Sahana Sandeep:u0932424@utah.edu

Introduction

We aim to devise a method for computing the Gröbner basis of combinational circuits using ZBDDs. For computing Gröbner basis we are using Buchberger's algorithm. This algorithm hinges on repetitive division. At present, there are many schemes available to do this sort of computation. The pitfall of some of these schemes is the explosion of terms in the remainder as a result of repetitive division. This can lead to an increase in computational complexity. This problem can be alleviated by the use of ZBDDs. Since ZBDD nodes share terms, an increase in the size of remainders doesn't cause a proportional increase in the size of the ZBDD. Thereby reducing the time taken to perform various operations.

Previous Work

To further our understanding of ZBDDs we referred to the work done by Shin-ichi Minato, et al (see references). Zero suppressed binary Decision Diagrams provide an efficient way of solving problems expressed in terms of set theory. When BDDs are used with applications that work with sparse sets, the BDDs grow large making the processing inefficient or impossible.
A set is sparse if the number of elements in it is much smaller than the total number of elements that may appear in the set. The problem of prohibitively large BDD size of the sparse set representation can be remedied by introducing a different kind of decision diagrams called Zero suppressed binary Decision Diagrams.
ZBDDs are extremely useful in the representation of combination sets. A combination of n objects can be represented by an n bit binary vector, $(x_n x_{n-1} \ldots x_2 x_1)$, where each bit $x_k \varepsilon \{1, 0\}$, expresses whether the corresponding object is included in the combination or not. A set of combinations can be represented by a set of the n bit binary vectors. Combination sets can be regarded as subsets of the power set on n objects.

We can represent a combination set with a Boolean function by using n input variables for each bit of the vector. The output value of the function expresses whether each combination specified by the input variables are included in the set or not. Such Boolean functions are called characteristic functions. The operations on sets such as union, intersection and difference can be executed by logic operations on characteristic functions.

ZBDDs are reduced by eliminating all the nodes whose 1-edge points to the 0-terminal node and then connecting the edge to the other sub-graph directly. In a ZBDD, each path from the root node to the 1-terminal node corresponds to each combination vector in the set.

A cover can be constructed using the CUDD package using the Cudd_zddChange(), Cudd_zddUnateProduct() and Cudd_zddUnion() functions. It is constructed starting from the 1-terminal node. A ZBDD that just has the 1-terminal node represents the set of only the null combination. For example, if the characteristic function has three variables x,y and z, the ZBDD with a 1-terminal node represents the combination set (000). The function Cudd_zddChange (x) inverts x on each combination and thus this function can be used to represent the combination set (100) and so on. In this way, all the required combinations are generated and finally Cudd_zddUnion () is used to carry out a union of all the sets to obtain a cover which is basically represented as a ZBDD.

Algorithms

We developed the following algorithms to perform operations like determining leading terms, computing LCM and subtraction.

Algorithm 1 Find Leading Term

```
procedure FindLeadingTerm(node)
    if node is terminal 1 node then
        return node
    end if
    terms ← FindIndex(node)
    temp ← CreateMinTerms(terms)
    return temp
end procedure
```

Algorithm 2 Find Index of the Variable

```
procedure FindIndex(node)
    terms ← 0
    while node ≠ terminal 1 node do
        terms.insert(node*)
        node ← true child of node
    end while
    return terms
end procedure
```

Algorithm 3 Create minterms from index

```
procedure CreateMinTerms(terms)
    if size of terms = 1 then
        return node
    end if
    temp ← unate product of terms[0] and terms[1]
    for i = 2; i < size of terms; i++ do
        temp ← unate product of temp and terms[i]
    end for
    return temp
end procedure
```

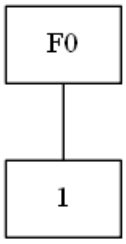
Algorithm 4 Subtraction of ZDDs

```
procedure ZddDiff(node1, node2)
    temp1 ← node1 ∩ node2
    temp2 ← node1 ∪ node2
    return Cudd_zddDiff(temp1, temp2)
end procedure
```

Results

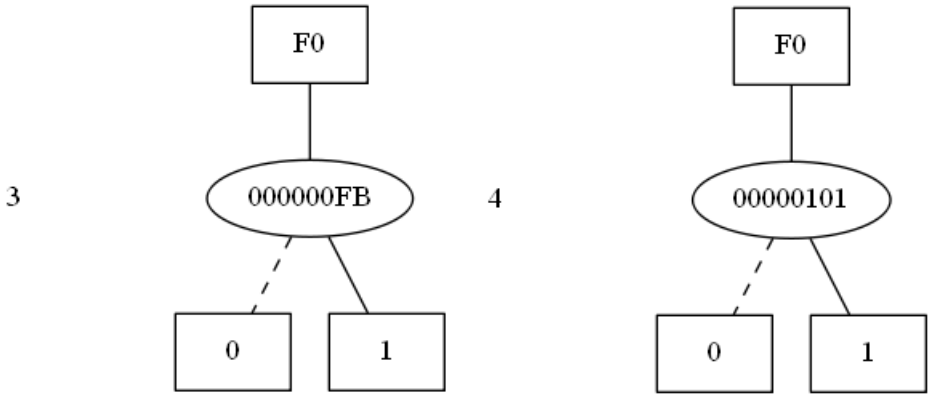
For validating our algorithm we used the following list of polynomials f[1]=ab+1, f[2]=ab+a+1, f[3]=ab+b+1, f[4]=ab+a+b, f[5]=xy+x+y, f[6]=yz+y+z. The reduced Gröbner basis for these polynomials is supposed to be 1. To illustrate the inner workings of our algorithms, we've collated outputs obtained from the CUDD package.

For constructing the polynomial ab+1 we first take a ZBDD which represents constant "1",

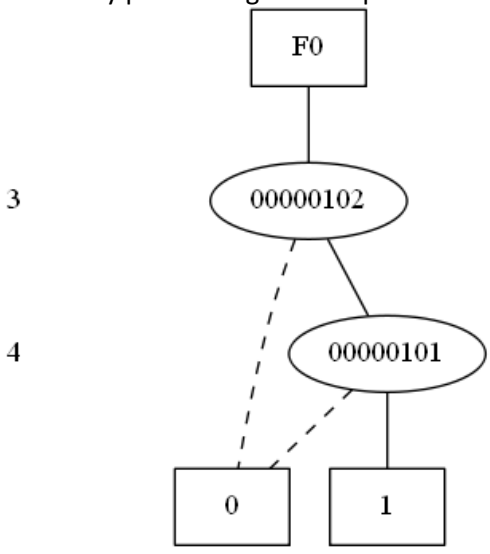


Then we apply the change function w.r.t "a" and "b", which give us the ZBDDs for "a" and "b".

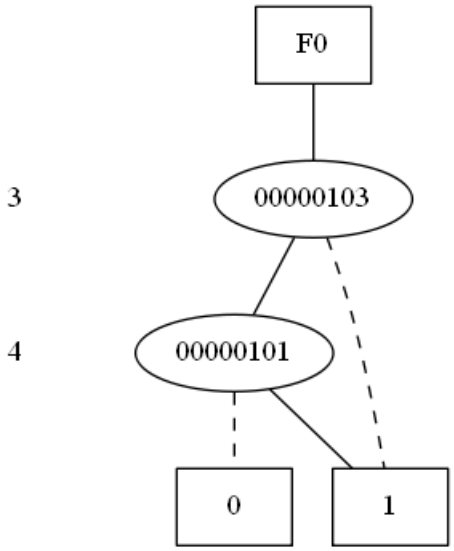
The term ordering that we've applied is x>y>a>b. The indices assigned by the CUDD package are in that order too, i.e. x =0,y=1,z=2,a=3 & b =4. The CUDD package uses the addresses of variables as node names, for the user's understanding it shows the index of the respective variable next to the node.



The ZBDD for "a.b" is obtained by performing a unate product of the ZBDDs of "a" and "b".

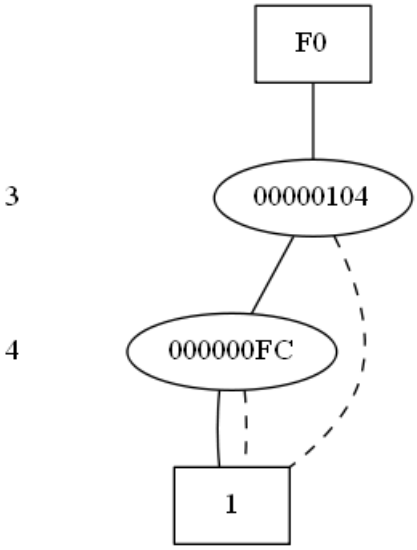


For obtaining the ZBDD of the final expression “ab+1” we perform a union of the ZBDD of “a.b” with the ZBDD of “1”.

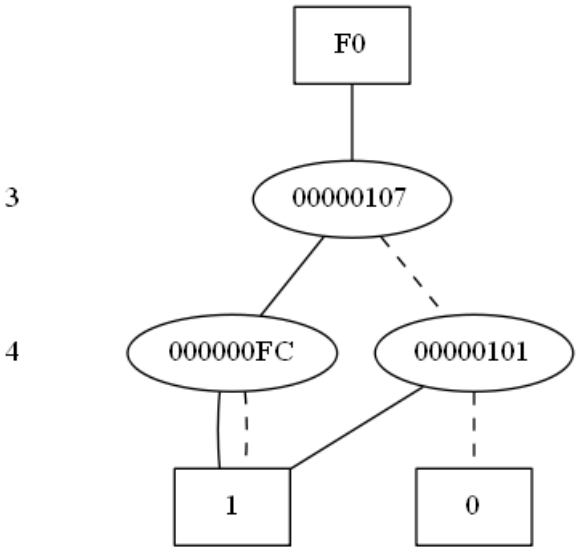


ZBDDs for the remaining polynomials obtained by the aforementioned algorithm, are as follows.

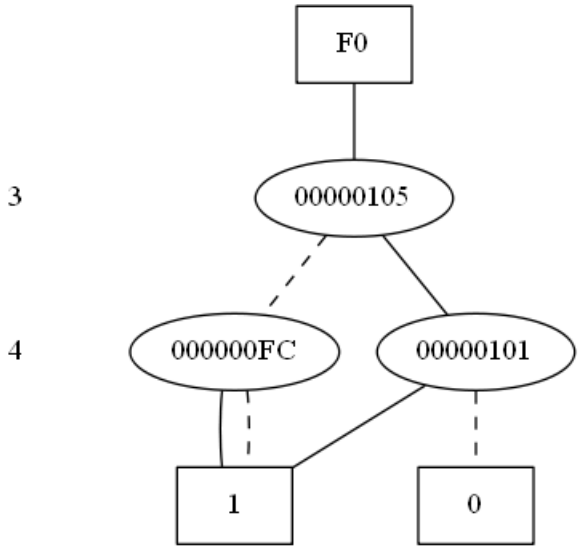
ab+a+1:



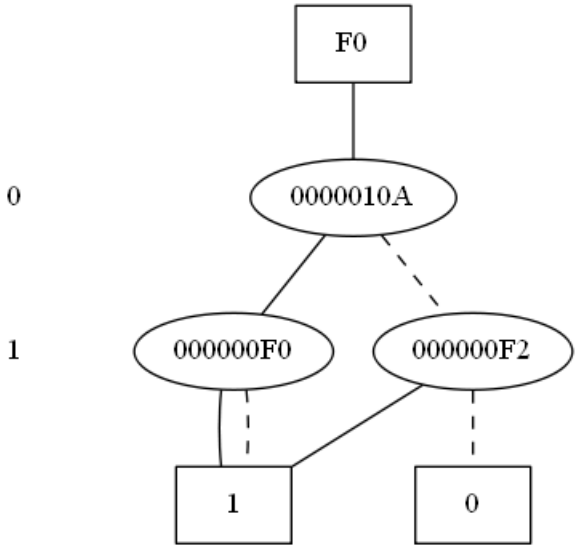
ab+a+b:


F0
3
00000107
4
000000FC
00000101
1
0

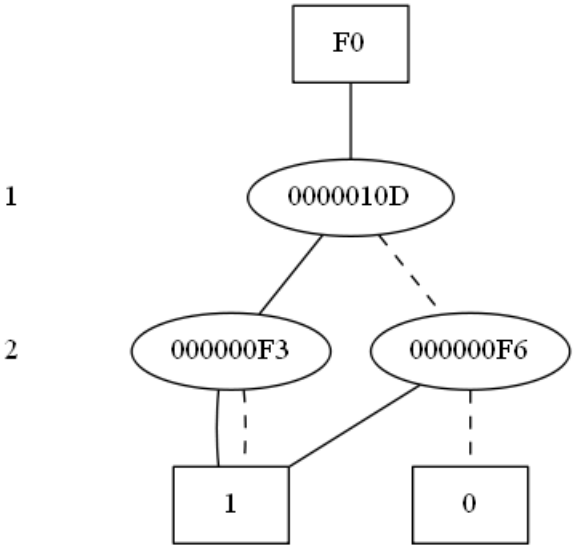
ab+b+1:


F0
3
00000105
4
000000FC
00000101
1
0

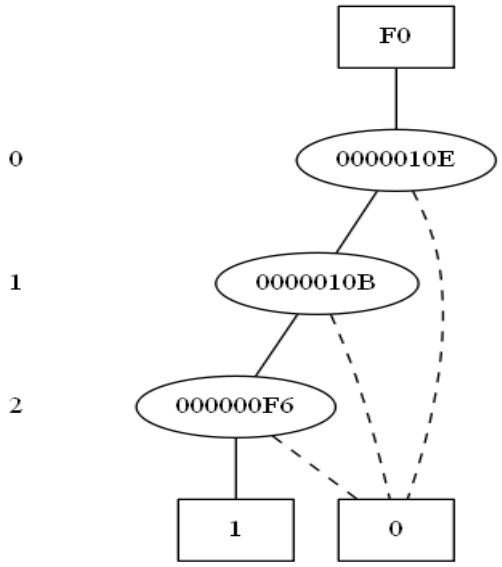
xy+x+y:


F0
0
0000010A
1
000000F0
000000F2
1
0

yz+y+z:

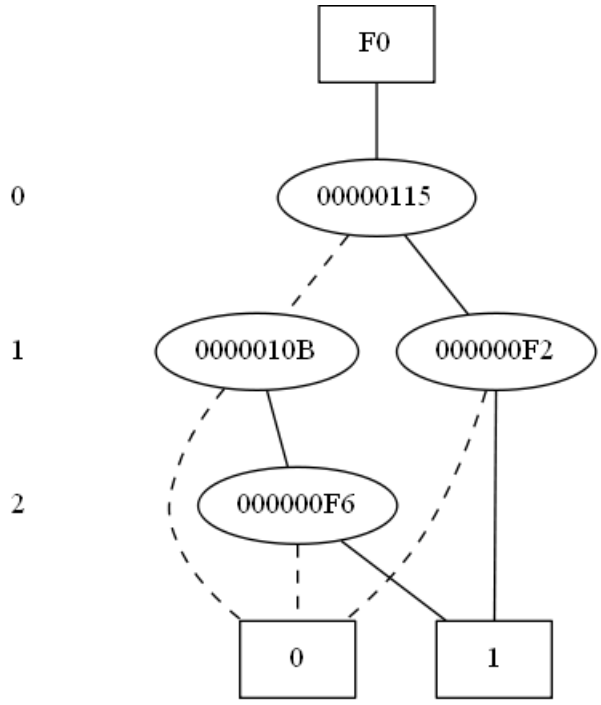

F0
1
0000010D
2
000000F3
000000F6
1
0

The next step is computing LCM using ZBDDs. To calculate the LCM of two terms, we just perform a unate product of the two. This example denotes the LCM of xy and yz, whose result is xyz.

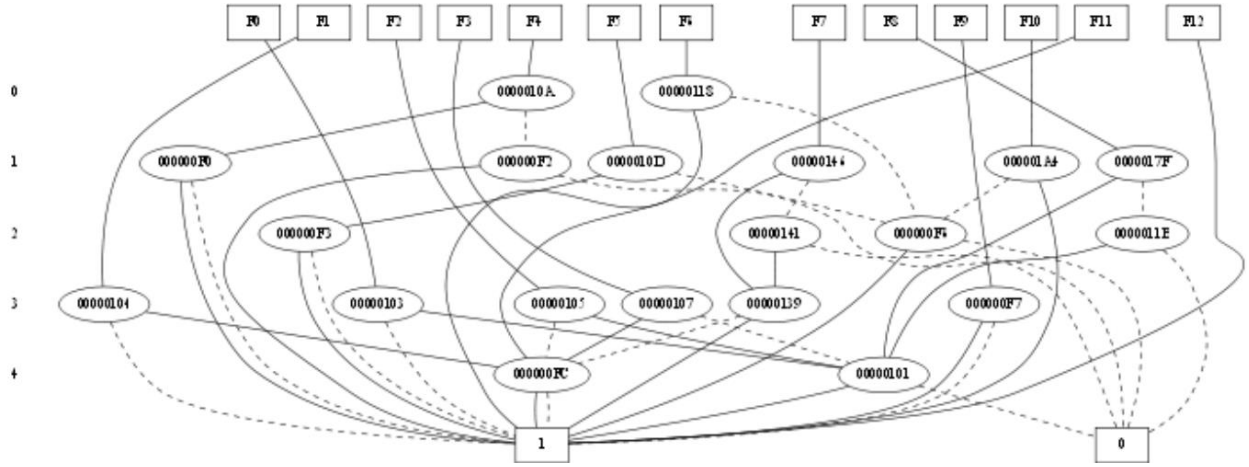


To calculate the S-polynomial, we perform all the operations listed in Buchberger's algorithm. For dividing leading terms we use the *Cudd_zddDivide* function, and for subtraction we use our own subtraction algorithm (see Algorithm 4).

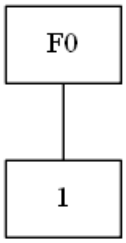
The following ZBDD shows the S-polynomial calculated for "xy+x+y" and "yz+y+z"



Finally to calculate Gröbner basis we need to perform multivariate division between the S-polynomial and all the polynomials in the list. For that purpose we need to compute to the leading term of each ZBDD, this can be done by traversing the positive edges of the ZBDD till the terminal “1” node is reached (see algorithm). The Gröbner basis computed for our list of polynomials is shown below:



The minimal and reduced Gröbner basis are both “1”for this list.



Conclusions

Our experiments add credence to the hypothesis that, computing Gröbner basis using ZBDDs is more efficient since the size of ZBDDs doesn’t increase drastically as the number of terms increase.

References

[1] M. Brickenstein and A. Dreyer, “Polybori: A Framework for Gröbner Basis Computations with Boolean Polynomials,” Journal of Symbolic Computation, vol. 44, no. 9, pp. 1326–1345, September 2009.
[2] Shin-ichi Minato, "Zero-suppressed BDDs for set manipulation in combinatorial problems", DAC '93: Proceedings of the 30th international conference on Design automation, 1993
[3] S. Minato: "Implicit Manipulation of Polynomials Using Zero-Suppressed BDDs", In Proc. of IEEE The European Design and Test Conference (ED&TC'95), pp. 449-454, Mar. 1995.
[4] Shin-ichi Minato, ”Zero-suppressed BDDs and Their Applications”, International Journal on Software Tools for Technology.
[5] CUDD Documentation: http://vlsi.colorado.edu/~fabio/CUDD/cuddIntro.html