Design Verification and Test of Digital VLSI Circuits NPTEL Video Course

Module-III

Lecture-I, II and III

Two level Boolean Logic Synthesis

Introduction

•In the last two modules, we discussed that in case of digital VLSI design we start with high-level system specifications, which are transformed into optimal Register Transfer Level (RTL) circuits using High Level Synthesis (HLS) algorithms.

•Once the RTL circuit is available, we need to transform it to gate level design, which can then be processed by backend algorithm; this process is called Logic Synthesis. Formally speaking, Boolean logic synthesis is a process by which an abstract form of desired circuit behavior, typically RTL, is transformed into a design implementation in terms of logic gates and flip-flops. This module is dedicated to logic synthesis of combinational and sequential circuits.

•

• It may be noted that two levels of logic are minimum required to implement an arbitrary Boolean function. Generally, we assume that the primitives are AND and OR gates and Inverters. AND gates are used at the first level and OR gates are used in the second level. Inverters may be present at some inputs of the gates of the first level, but it is not considered as an additional level.

Introduction

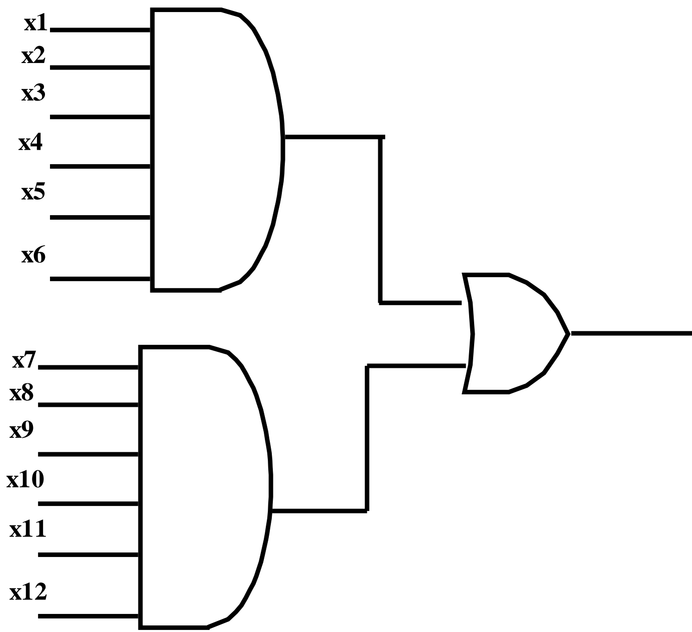
•Many other choices are also possible namely, using OR gates at first level and AND gates in the second, using NOR and NAND gates etc.

•It is also possible to implement a circuit in more than two levels, however, it is more complex procedure.

•In this triple lecture, we will first discuss two level logic synthesis procedures. Latter in this module, we will discuss multilevel synthesis.

•There are two main reasons why we may want to implement a circuit in two levels, rather than multiple levels namely, speed of operation and simplicity of the algorithms. However, in practical cases two level implementation may not be possible. Reducing the number of levels increase the fanin and fanout counts of gates. Gates having high fanins and fanouts are slow. Therefore, design libraries do not generally have gates with more than four fanins; this requires multiple level synthesis.

Introduction



(a) Two level Implementation

(a) Multi-level Implementation

we have a Boolean function as $f = x1.x2.x3.x4.x5.x6 + x7.x8.x9.x10.x11.x12$.

Introduction

•If we want to have two level implementation, then we need an AND gate having 6 fanins. If we have gates with 3 maximum fanins, a multilevel implementation is required

• However, two-level implementation is important to be studied. Two-level implementations are easier to design and analyze because the solution space is greatly restricted.

Further, before the development of CMOS logic gates, Boolean functions were realized using Programmable Logic Array (PLAs) and Programmable Array Logic (PALs).

•These programmable arrays can implement any combinational logic circuit. Broadly speaking, they have a set of programmable AND planes, which connect to a set of programmable OR planes; this arrangement is two level AND-OR realization that can implement functions in terms of sum of products. In addition, the outputs could be conditionally complemented when required. As PLAs and PALs worked on "product of sum (POS)/sum of product (SOP)" based representation, the algorithms for the optimum implementation of two-level functions were developed in the fifties.

Introduction

- With the introduction of CMOS based standard cell and semi-custom design methodologies, there was a decline in the popularity of PLAs and PALs.

- When implementing a circuit with standard cells, it is customary to use multi-level implementation because generally a CMOS gate has a maximum of 4 fanins.

- The cost in terms of area or speed of a multi-level implementation is not directly related to the cost of an equivalent two-level circuit. However, the role of the two-level techniques is still important, because optimization of multilevel logic involves a network whose nodes represent functions, which are represented as two-level circuits.

- Therefore, in this (triple) lecture we discuss two-level Boolean logic synthesis (i.e., optimized two level implementation of a circuit for a given Boolean function). Following that, latter in this module we will also discuss multilevel logic synthesis.

Representation: Sums of Products and Products of Sums

We know that input/output of an RTL circuit can be represented by a Boolean function.

In logic synthesis, we need to design a circuit to implement the Boolean function. It may be noted that more the number of terms (will be defined precisely latter) in the function, more the number of gates in the circuit.

Therefore, the primary objective of logic synthesis is to determine a minimal gate representation of the function; this is called "minimizing Boolean functions". From the context of two level implementation, our objective is to find the simplest two-level formula that represents a given function. Simplicity is measured, in terms of the number of gates and gate inputs of the circuit.

Representation: Sums of Products and Products of Sums

Now we formally define a two-level formulae. Formula consists of constants, variables, parentheses and operators. A letter is a constant or a variable. A literal is a letter or its complement. For example, for $0,1,x,y$ are letters and $0,1,x,x',y'$ are literals; $0,1$ are constant literals and x',y' variable literals. The following definitions are introduced.

Definition 1: A product term is a formula of one of the following forms:

1. 1
2. a variable literal
3. a conjunction of variable literals where no letter appears more than once.

Definition 2: A sum term is a formula of one of the following forms:

1. 0;
2. a variable literal
3. a disjunction of variable literals where no letter appears more than once.

Representation: Sums of Products and Products of Sums

For example, $x.y'$ is a product term, $x+y'$ is a sum term and x' is both. On the other hand, $x.x'$ is neither product terms nor sum term, because the letter x appears twice and the term reduces to 0.

Definition 3: A Sum of Products (SOP) formula is one of the following:
0;
a product term;
a disjunction of product terms.

Example, $f = x.y'+yz$

The cost of a SOP formula is determined by the number of product terms and the number of literals. Broadly, speaking, number of product terms determine the number of AND gates and number of literals determine the number of inputs of a gate. $f = x.y'+yz$ has two product terms and four literals.

Representation: Sums of Products and Products of Sums

Definition 4: A Product of Sums (POS) formula is one of the following:

1. 1;
2. a sum term;
3. a conjunction of sum terms.

Example, $f = (x + y').(y + z)$

The cost of a POS formula is determined by the number of sum terms and the number of literals. Broadly, speaking, number of sum terms determine the number of OR gates and number of literals determine the number of inputs of a gate. $f = (x + y').(y + z)$ has two sum terms and four literals.

A two-level formula is either a SOP or a POS. The two forms are, one is the dual of the other, which can be shown by De-Morgan's theorem. In this module, we will restrict all our discussions on SOP based representation of a Boolean formula.

Prime Implicants

There are two basic steps for minimizing Boolean functions namely, determining prime implicants and then finding subset such implicants that cover all product terms of a function. In this section, we introduce the concept of prime implicants and schemes to determine prime implicants.

Definition 5: An implicant of a function is a product term that is included in the function.

For instance, xyz is an implicant of $f(x,y,z) = xy$; $xy = \{xyz + xyz'\}$.

Definition 6: A prime implicant of a function is an implicant that is not included in any other implicant of the function.

For instance, xyz is not a prime implicant of $f(x,y,z) = xy + x'y'z'$ because xyz is contained in xy. xy is a prime implicant of $f(x,y,z)$ because it is not contained in $x'y'z'$. So, if is an implicant is not prime, then it is possible to obtain prime implicant of by removing some literals from it.

Prime Implicants

Definition 7: If a prime implicant includes a minterm that is not included in any other prime implicant, then that prime implicant is essential.

For example, $f(x,y,z) = xy + x'y'z'$ has two prime implicants namely, xy and $x'y'z'$. Prime implicant xy is essential because xy contains xyz and xyz' which are not contained in any other prime implicant (i.e., $x'y'z'$). In another function $f(x,y,z) = xy + xy' + xz'$, prime implicants are xy, xy' and xz'. Among them, xz' is not essential because $xz' = \{xy'z', xyz'\}$ and $xy'z'$ is in xy' and xyz' is in xy.

Determining Prime Implicants

Theorem 1: If cost of a Boolean function depends on literals then a minimal SOP must always consist of a sum of prime implicants.

Proof: Let us assume that f is an SOP which is minimal and one (product) term is non-prime. Another SOP formula $f1 \equiv f$ exists, that can be obtained by replacing the non-prime implicant by a prime implicant that contains it. The cost does not increase and the formula is equivalent to the original one. So, we reach a contradiction i.e., f is an SOP which is minimal and all terms are prime implicants.

For example, let SOP representation of $f(x, y)$ be $x + x'y$. A prime implicant of f is y. The SOP representation can be changed as $x + y$, where we have replaced $x'y$ with y; y includes $x'y$. So the new SOP is rewritten by saving one literal.

Determining Prime Implicants

Theorem 1: If cost of a Boolean function depends on literals then a minimal SOP must always consist of a sum of prime implicants.

Proof: Let us assume that f is an SOP which is minimal and one (product) term is non-prime. Another SOP formula $f1 \equiv f$ exists, that can be obtained by replacing the non-prime implicant by a prime implicant that contains it. The cost does not increase and the formula is equivalent to the original one. So, we reach a contradiction i.e., f is an SOP which is minimal and all terms are prime implicants.

For example, let SOP representation of $f(x, y)$ be $x + x'y$. A prime implicant of f is y. The SOP representation can be changed as $x + y$, where we have replaced $x'y$ with y; y includes $x'y$. So the new SOP is rewritten by saving one literal.

Determining Prime Implicants by Tabular Method

Definition 8: Given a Boolean function in SOP form $Xy + Xy'$, where X is a product term not having variable y, then consensus can be applied on the two terms Xy, Xy' to generate $X(= Xy + Xy')$; X is the consensus term containing both Xy, Xy'. This is also called distance-1 merging.

In other words, pairs of terms that differ in exactly one letter, which must appear complemented in one term and un-complemented in the other, are used for consensus.

Determining Prime Implicants by Tabular Method

Given a Boolean formula in SOP form we need to determine the prime implicants. The steps of tabular method are given below.

1. Express the function in minterm canonical form

2. Consider all pairs of adjacent terms, i.e., the pairs of terms to which consensus can be applied. The consensus terms are implicants of the function though not necessarily prime. All terms that form these new consensus terms are included in the new terms, and hence they are not prime. We mark the old terms as non-prime and are not used for further consensus.

3. The new terms are added to the SOP and steps 1,2 are repeated till no more consensus terms can be found

4. All terms that are absorbed (or contained) by the new terms are marked as non-primes. Finally, the terms that are not marked constitute all the prime implicants of the function.

Determining Prime Implicants by Tabular Method

Now we illustrate the scheme using an example.

Consider the SOP: $f(w,x,y,z) = x'y' + wxy + x'yz'$.

The function in minterm canonical form is as follows:

$$w'x'y'z' + w'x'y'z + wx'y'z' + wx'y'z + wxyz' + wxyz + w'x'yz' + wx'yz'.$$

- Now we construct a table, where the minterms appearing in the canonical form are entered.

- The column is divided into four parts based on number of complemented letters of the terms.
- The first group consists of minterms with no un-complemented literals.

- In general, some groups may be empty. So, for consensus, we to need to compare terms of immediately successive groups.

- It may be noted that we need not consider the minterms in the immediately preceding group, because this would only cause us to repeat comparisons.

Determining Prime Implicants by Tabular Method

$w'x'y'z'$
$w'x'y'z$ $wx'y'z'$ $w'x'yz'$
$wx'y'z$ $wx'yz'$
$wxyz'$
$wxyz$

Minterms partitioned based on number of complemented alphabets

Determining Prime Implicants by Tabular Method

In the example, if we compare the term from the first row ($w'x'y'z'$) with the top most term of second row ($w'x'y'z$), their consensus term is $w'x'y'$.

Both $w'x'y'z'$ and $w'x'y'z$ are marked as non-prime because there exists another implicant $w'x'y'$ that contains them.

A second column is created in the table where in the first row the consensus term $w'x'y'$ is placed;

Determining Prime Implicants by Tabular Method

In the example, if we compare the term from the first row ($w'x'y'z'$) with the top most term of second row ($w'x'y'z$), their consensus term is $w'x'y'$.

Both $w'x'y'z'$ and $w'x'y'z$ are marked as non-prime because there exists another implicant $w'x'y'$ that contains them.

A second column is created in the table where in the first row the consensus term $w'x'y'$ is placed;

Determining Prime Implicants by Tabular Method

$w'x'y'z'$ (not prime)	$w'x'y'$
$w'x'y'z$ (not prime) $wx'y'z'$ $w'x'yz'$	
$wx'y'z$ $wx'yz'$	
$wxyz'$	
$wxyz$	

Entry after consensus of $w'x'y'z'$ and $w'x'y'z$

Determining Prime Implicants by Tabular Method

$w'x'y'z'$ (not prime)	$w'x'y'$ $x'y'z'$ $w'x'z'$
$w'x'y'z$ (not prime) $wx'y'z'$(not prime) $w'x'yz'$(not prime)	$x'y'z$ $wx'y'$ $wx'z'$ $x'yz'$
$wx'y'z$(not prime) $wx'yz'$(not prime)	wyz'
$wxyz'$(not prime)	wxy
$wxyz$(not prime)	

Entry after consensus of all terms of first table

Determining Prime Implicants by Tabular Method

$w'x'y'z'$ (not prime)	$w'x'y'$ (notprime) $x'y'z'$ (notprime) $w'x'z'$ (notprime)	$x'y'$ $x'z'$
$w'x'y'z$ (not prime) $wx'y'z'$(not prime) $w'x'yz'$(not prime)	$x'y'z$ (not prime) $wx'y'$ (not prime) $wx'z'$ (not prime) $x'yz'$ (not prime)	
$wx'y'z$(not prime) $wx'yz'$(not prime)	wyz'	
$wxyz'$(not prime)	wxy	
$wxyz$(not prime)		

Entry after consensus of all terms of column 2

So here, we have four prime implicants as $wyz', wxy, x'y', x'z'$.

Determining Prime Implicants by Tabular Method

•It my be noted that the function considered in the example was completely specified, i.e., there was no don't care terms.

•If the function is incompletely specified then we follow the same procedure discussed above (for completely specified functions), however, we drop the mean terms that contain only don't care minterms.

•Specifically, a product term is a prime implicant of an incompletely specified function if it is a prime implicant of the function and contains at least one minterm which is not don't care.

•Now we will consider the following example, which illustrates tabular method for incompletely specified functions.

Determining Prime Implicants by Tabular Method

Consider the function written in SOP form $f(x,y,z) = x'z' + xyz' + d(xy'z' + xy'z)$.

The function in minterm canonical form is as follows

$x'y'z' + x'yz' + xyz' + d(xy'z' + xy'z)$.

The table for consensus of the SOP is shown in next Table

$x'y'z'$ (not prime)	$x'z'$(not prime) $y'z'$(not prime)	z'
$x'yz'$(not prime) $xy'z'....don't$(not prime)	yz'(not prime) xz'(not prime) $xy'.....don't$	
xyz'(not prime) $xy'z.....don't$(not prime)		

Determining Prime Implicants by Tabular Method

The prime implicates are xy' and z'. However, we do not consider xy' as prime implicate because it comprises only don't care terms.

It may be noted that in case of merging, if both the terms in consensus are don't care, then the new term is also marked don't care.

In this example, the term xy' in second column is marked don't care because both the terms in consensus $xy'z', xy'z$ are don't cares.

Thank You

Design Verification and Test of Digital VLSI Circuits NPTEL Video Course

Module-III

Lecture-II

Two level Boolean Logic Synthesis

Determining Prime Implicants by Iterated Consensus

- The tabular method discussed above is based on the application of the Quine's theorem.

- The tabular method is simple, however, it requires the minterm canonical form to start with, which is exponential in number of input variables (alphabets).

- To cater to this issue, iterated consensus method does not expand the product terms (of SOP) to minterms. Instead, consensus is made between all pairs of product terms.
  - The consensus terms of all pairs of terms that are not contained in some other term are added to the equation.

  - The new terms are compared to the existing terms and among themselves to see if more new consensus terms can be generated.

  - All terms that are contained in some other term are removed.

  - Once no more new terms can be generated, the SOP has only prime implicants.

Determining Prime Implicants by Iterated Consensus

Definition 9: An SOP formula is complete sum if it comprises all the prime implicants of the function it represents.

Theorem 2 : A SOP formula is a complete sum if and only if:

1. No term includes any other term.
2. The consensus of any two terms of the formula either does not exist or is contained in some term of the formula.

We only highlight the basic philosophy behind the proof.

Suppose a SOP representing a function is not a complete sum, because there is one prime implicant of the function that does not appear in the SOP (and all other terms are prime implicants). Therefore, the remaining prime implicant must be covered by two or more of the prime implicants in the SOP. Suppose for simplicity let us assume that there are two such prime implicants p_1 and p_2. If we add the consensus term of p_1 and p_2, we get another term that covers the missing prime implicant.

Determining Prime Implicants by Iterated Consensus

The theorem suggests a simple procedure (steps given below) to generate all the primes of a function, called iterated consensus.

1. Start from an arbitrary SOP formula and add the consensus terms of all pairs of terms that are not contained in some other term.

2. The new terms are compared to the existing terms and among themselves to see if new consensus terms can be generated (and added). All terms that are contained in some other term are removed.

3. Repeat step 2 until no more new terms are created.

The steps generate a complete sum (i.e., all prime implicants)

Determining Prime Implicants by Iterated Consensus

As an example, consider the SOP for the function $f(x, y, z) = x'z' + xyz' + xy'z' + xy'z$ (it is the same function used in the last section, however, now the don’t care terms are made minterms).

Steps of iterative consensus for the function are discussed below.

1. Consensus of $x'z'$ and xyz' (1st two terms) generates yz', which contains xyz'. By eliminating xyz' and adding yz', we have $f = x'z' + xy'z' + xy'z + yz'$
2. Consensus of $x'z'$ and $xy'z'$ will generate $y'z'$, which contains $xy'z'$. By eliminating $xy'z'$ and adding $y'z'$, we have $f = x'z' + xy'z + yz' + y'z'$.

Determining Prime Implicants by Iterated Consensus

3. $x'z'$ and $xy'z$ do not have consensus

4. $x'z'$ and yz' do not have consensus; $xy'z$ and yz' do not have consensus

5. yz' and $y'z'$ generates z'. yz', $y'z'$ and $x'z'$ are in z'. By eliminating yz', $y'z'$ and $x'z'$ and adding z', we have $f = xy'z + z'$.

6. Consensus between $xy'z$ and z' generates xy', which contains $xy'z$. Eliminating $xy'z$ and adding xy' we get $f = xy' + z'$.

Now $f = xy' + z'$ is the complete sum as no more new terms can be created. It may be noted that xy' and z' are the prime implicants; same result was determined using tabular method, as discussed in the last sub-section (however, in that case xy' comprised only don't cares).

Selecting a Subset of Primes

- In the last Lecture, we discussed techniques to generate prime implicants from a given SOP Boolean equation.

- Now, we need to select a subset of the prime implicants that cover all the minterms.

- The approach of minimizing a SOP formula based on computing all primes and then selecting some of them to form a cover is called Quine-McCluskey procedure.

- In this lecture, we will discuss Quine-McCluskey procedure and illustrate the same using examples.

Selecting a Subset of Primes

Let us consider the following SOP formula:

$$f(x,y,z) = yz + x'y + y'z' + xyz + x'z' + x'y'z'.$$

The complete sum for the function (i.e., in terms of prime implicants) is

$$f(x,y,z) = x'y + x'z' + y'z' + yz.$$

- The condition that any subset of primes must satisfy the Boolean formula is that, each minterm (for which the function is 1) is to be included in at least one prime implicant, which is in the subset.

- A subset of (prime) implicants that satisfies this requirement is called a SOP cover of the function, or simply a cover.

- The concept of "cover" can be represented by a constraint matrix.

- Each column of the constraint matrix corresponds to a prime implicant and each row corresponds to a minterm.

Let C be the constraint matrix and let c_{ij} be the element in row i and column j;

$c_{ij} = 1\,(0)$, if the j-th prime covers (does not cover) the i-th minterm.

Selecting a Subset of Primes

In the example, let us represent the prime implicants as follows.

$$x'y \ldots\ldots\ldots \text{PI1}$$
$$x'z' \ldots\ldots\ldots \text{PI2}$$
$$y'z' \ldots\ldots\ldots \text{PI3}$$
$$yz \ldots\ldots\ldots \text{PI4}$$

Now the matrix C is as follows:

	PI1	PI2	PI3	PI4
$x'y'z'$	0	1	1	0
$x'yz'$	1	1	0	0
$x'yz$	1	0	0	1
xyz	0	0	0	1
$xy'z'$	0	0	1	0

Selecting a Subset of Primes

Given the constraint matrix, we need to find a subset of columns of minimum cost that *covers* all the rows.

In other words, for every row there must be at least one selected column with a 1 in that row.

In the example, we note that, columns PI3 and PI4 must be part of every solution, because the last two rows are singletons. If a row is a singleton, there is only one column that may cover it and that column must be selected.

It may also be noted that prime implicates corresponding to PI3 (and PI4) are essential because they comprise minterm xyz (and $xy'z'$) which is not in any other prime implicant.

Selecting a Subset of Primes

When we select some columns, we simplify the constraint matrix accordingly, by eliminating the selected columns and the rows covered by them; if we select PI3 and PI4 then the resultant constraint matrix is as follows.

	PI1	PI2
$xy'z$	1	1

From the resultant matrix, we can easily see that a complete solution may be obtained by adding either PI1 or PI2 to PI3 and PI4.

In the first case we obtain $f(x,y,z) = x'y + y'z' + yz$ in the second, we obtain $f(x,y,z) = x'z' + y'z' + yz$. In this case, both the solutions involve same number of literals. However, in a general case, we select the solution involving minimum literals.

Selecting a Subset of Primes

Let us consider an arbitrary constraint matrix as shown below.

	PI1	PI2	PI3	PI4
minterm1	1	1	0	0
minterm2	0	1	1	0
minterm3	0	0	1	1
minterm4	1	0	0	1

1. A function has a cyclic core if we cannot identify columns of the constraint that must be part of the solution or that can be eliminated.

2. In the arbitrary constraint matrix, each row is covered by exactly two columns and each column covers exactly two rows. There is no essential primes. There is no apparent reason to prefer one column over another. For this matrix we must proceed by choosing one column arbitrarily and finding the best solution subject to the assumption that the column is selected. We must then assume that the column is not in the solution and find another solution. This process is repeated till the whole solution space is explored.

Selecting a Subset of Primes

•From the two previous examples, we note that two important mechanisms are involved in determining the cover (of minterms using prime implicants)

1. Reduction of the constraint matrix by selecting columns that cover singleton rows (i.e., essential primes)
2. Exploring solution space by branching in case of cyclic cores.

•We will discuss algorithms to perform the above two tasks. Before that, however, we formulate the covering problem as a constraint matrix, formally. It may be noted that once the matrix is created, it becomes a problem of determining minimum cost columns that cover all rows, which can be thought independently of Boolean functions, prime implicants, minterms etc.

Selecting a Subset of Primes

One may readily see that the rows of the constraint matrix in our first example can be written as the switching function as follows

- (PI2+PI3) for the 1st row
- (PI1+PI2) for the 2nd row.
- (PI1+PI4) for the 3rd row.
- PI3 for the 4th row
- PI4 for the 5th row

Selecting a Subset of Primes

The switching function $(\mathrm{PI2}+\mathrm{PI3})$ for the 1st row evaluates to one if either $\mathrm{PI2}=1$ or $\mathrm{PI3}=1$ or both $\mathrm{PI2}=1, \mathrm{PI3}=1$.

We interpret $\mathrm{PI_i}=1$ as "column *i* is selected," and all *j* "rows covered" for which the matrix has $c_{ij}=1$.

We can proceed similarly for the other rows. The expressions thus obtained are switching functions that must all be 1 for a solution to be valid. Hence, their product must be 1. We can therefore write the following equation as an equivalent to the constraint matrix $(\mathrm{PI2}+\mathrm{PI3}).(\mathrm{PI1}+\mathrm{PI2}).(\mathrm{PI1}+\mathrm{PI4}).\mathrm{PI3}.\mathrm{PI4}$

Selecting a Subset of Primes

- This equation is called the constraint equation of the covering problem (represented by the constrained matrix). The covering problem can be stated in this setting as the problem of finding an assignment of zeroes and ones to the variables that is a solution to the constraint equation and that is of minimum cost. Cost is predefined on the variables depending on the literals in the corresponding prime implicant.

- We may note that all variables in the constraint equation are un-complemented. This is not a coincidence, but rather a direct consequence of the way the equation is built; we do not put the concept “not considering a column” in the equation. A formula where no letter appears with both phases is said unate. A non-unate formula is called binate. Because of the form of the constraint equation that we get, the covering problem we are dealing with is sometimes called unate covering.

Unate Covering Problem

The Unate Covering Problem (UCP) can be defined in terms of both a constraint matrix and a constraint equation. As we will solve the UCP using constraint matrix, we will define the problem only in that from

Definition 10: Let M be a matrix of m rows and n columns, for which M_{ij} is either 0 or 1. The unate covering problem involves finding a minimum cardinality column subset S, such that for all such that for all $S' \; \exists_{j \in S'} M_{ij} = 1, \forall i \in \{1, 2, \ldots m\} \rightarrow |S| \leq |S'|$.

In other words, the columns in the set S "cover" M in the sense, that every row of M contains a 1-entry in at least one of the columns of S, and there is no smaller set S' which also covers M.

Unate Covering Problem

To solve the unate covering problem, the constraint matrix can be simplified by considering three factors namely,

- Columns that have singleton rows are mandatory to be taken in the cover. All rows covered by such columns can be deleted and so do the columns. The unate covering can be then carried out in the reduced matrix. The procedure is called elimination of rows covered by “essential columns”. For example, in the matrix given below, PI4 is an essential column.

	PI1	PI2	PI3	PI4
minterm1	1	1	0	0
minterm2	0	1	1	0
minterm3	0	0	1	1
minterm4	0	0	0	1

Unate Covering Problem

PI4 also covers minterm3. The resultant simplified matrix (after elimination of rows covered by essential column PI4) is give below.

	PI1	PI2	PI3
minterm1	1	1	0
minterm2	0	1	1

If a row r_i of the constraint matrix has all the ones of another row r_j, then r_i is covered whenever r_j is covered. We say that r_i dominates r_j. In the example matrix below, minterm1 has all 1s of another row, minterm2. So minterm1 dominates minterm2.

	PI1	PI2	PI3	PI4
minterm1	1	1	1	0
minterm2	0	1	1	0
minterm3	0	0	1	1
minterm4	0	0	0	1

Unate Covering Problem

All dominating rows (minterm1, in this example) can be eliminated from the constraint matrix. This is because, if a column is taken which covers the dominated row then by virtue of the "common 1", the dominating row is also covered. In the example, if we select PI2 to cover minterm2, then by virtue of common 1 (i.e., c_{12}) minterm1 is also covered. By applying "eliminating dominating rows", the reduced matrix is as follows

	PI1	PI2	PI3	PI4
minterm2	0	1	1	0
minterm3	0	0	1	1
minterm4	0	0	0	1

Unate Covering Problem

If a column c_i of the constraint matrix has all the ones of another column c_j, then c_i is said to dominate c_j. In such a case we may say that then c_i covers all rows that are covered by c_j. Also, if the cost of prime implicant (in terms of number of literals) corresponding to c_i is not more than that corresponding to c_j, then we can say that c_i is not inferior to c_j, in that it covers all the rows that c_j covers, at a cost that is not larger. This means that we can eliminate c_j and simplify the matrix, without giving up the possibility of finding an optimum solution. The process is called elimination of columns through "column dominance".

Unate Covering Problem

In the matrix given below, row dominance are--PI2 dominates PI1 and PI3 dominates PI1 and PI2.

	PI1	PI2	PI3	PI4
minterm1	1	1	1	0
minterm2	0	1	1	0
minterm3	0	0	1	1
minterm4	0	0	0	1

If cost of PI3 is not higher than PI1 and PI2 then, by eliminating PI1 and PI2, the simplified matrix is given below.

	PI3	PI4
minterm1	1	0
minterm2	1	0
minterm3	1	1
minterm4	0	1

It may be noted that if the matrix is cyclic then none of the above three simplification procedures can be applied.

Design Verification and Test of Digital VLSI Circuits NPTEL Video Course

Module-III

Lecture-III

Two level Boolean Logic Synthesis

Branch-and-Bound Algorithm for selecting cover

The unate covering problem can be solved efficiently using Branch-and-Bound Algorithm. Now we will present the scheme and then illustrate the same with examples.

- There may be multiple optimum solutions to the problem and our intension to find one of them.
- The search space can be defined as a subset of selected columns.; if there are n columns then solution space is $O(2^n)$.
- In the branch and bound procedure for this problem the solution space is enumerated in form of a binary search tree.
- A node corresponds to a column and the left (right) edge represents the column being (not being) considered in the cover.
- It may be noted that for all columns there may not be a node in the tree, because considering some column may result in elimination of some other due to dominance or inclusion due to essentiality.

Branch-and-Bound Algorithm for selecting cover

Therefore, if we can determine that a given part of the search space (i.e., sub-tree rooted at a non-leaf node) does not contain any solution better than the one we have found so far, then we can avoid exploring that part of the search space altogether.

We start searching another branch by considering a column that was considered not to be taken in the cover. In addition, we may search other branch by not considering a column that was considered to be taken in the cover. Therefore, in branch and bound algorithm for unate covering problem, we resort to two basic ideas—organize the search space in from of a binary search tree and explore the branches of the tree.

Branch-and-Bound Algorithm for selecting cover

•For efficiency, we need some scheme to estimate the lower bound cost of a solution (given a constraint matrix) without fully exploring the search space.

•When we find that exploring a given path would lead to more expensive solution than expected, we retract to other braches without exploring the path under question.

•In other words, at any given node of the search tree, we have selected and rejected some columns. These columns are identified by the path from the root of the tree to that node. Hence, at that node we have a partial solution. If the cost of a partial solution (from that node) exceeds the expected solution at a node, clearly we can abandon that path and track back.

Branch-and-Bound Algorithm for selecting cover

- Branch and Bound then tries to establish whether a new best solution can still be found by proceeding from the current node in a different branch.

- The way of computing the lower bound depends on the particular problem. It is obvious that a careful choice of the lower bound criterion is important. Ideally, the criterion should provide an accurate estimate of the real minimum cost incurred in completing the current solution. At the same time, the computation of the bound should be fast. So, most of the lower bound estimation algorithms are fast heuristics, that may not provide a sharp bound.

- So the branch and bound algorithm for unate covering problem generally provides near optimal solutions. However, due to the lower bound estimation heuristics the execution time is low.

Branch-and-Bound Algorithm for selecting cover

Before we turn our attention to the computation of the lower bound for the unate covering problem, we now look at the steps for the branch-and-bound algorithm for the unate covering problem.

1. Execute the lower bound computation algorithm on the constraint matrix. The "Initial Lower Bound" is the Lower Bound. Current Cost is set to 0.

2. Select or de-select a column PI (i.e., a path in the solution space corresponding PI=1 or PI=0). If a column PI is selected then add PI to the cover and add cost 1 to the Current Cost. Apply reduction techniques (essential columns, row and column dominance) to the constraint matrix. For all essential columns, add them to the cover and increase Current Cost by the number of essential columns. If the problem is now reduced to a terminal case (because of selection of a column), then check whether the solution thus found is better than (or at least at par) the "Initial Lower Bound" (i.e., Initial Lower Bound >= Current cost). If so, the cover is returned and algorithm is terminated.

Branch-and-Bound Algorithm for selecting cover

3. On the reduced matrix, again compute the lower bound. Add the lower bound value to the Current Cost. If there is still a chance of getting an optimal solution, identify a column to be selected/de-selected from the reduced matrix and go to Step-2; execute step-2 by selecting/de-selecting the column being marked.

4. If the value of lower bound + Current Cost is higher than Initial Lower Bound, then there is no point in exploring that path. Undo selection or de-selection of the column done last (in Step-2), consequent selection/de-selection of columns (due to matrix reduction) and addition in Current cost. Go to step-2 and traverse another path by selecting or de-selecting an alternative column.

Computation of the Lower Bound

As discussed in the last sub-section, the branch and bound algorithm needs a procedure that can provide a quick estimate of the minimum number of columns required for a cover. This lower bound helps the algorithm to retract (to other branches by selecting appropriate column) if the estimated solution cost is higher than the expected one, without actually exploring the full path.

We now address the problem of computing a lower bound approximation to the cost of covering a constraint matrix in terms of number of columns. Before the discussion, some definitions and a theorem are introduced.

Definition 11: In a given covering matrix *C*, suppose that two rows have no nonzero columns in common, we say these two rows are independent (that is, column-disjoint).

Computation of the Lower Bound

It is obvious that we need two different columns to cover these two rows.

Generalizing this argument, if a matrix has *n* rows that are disjoint (pair wise), we need at least *n* columns to cover the whole matrix.

In this case, the rows are said to form an independent set of rows.

In the constraint matrix given below, rows corresponding to minterm1, minterm3 and minterm5 are independent as they do not have a common column that has 1. Similarly, rows corresponding to minterm2, minterm4 and minterm6 are independent. So, we need at least three columns to cover the matrix. The lower bound in this case is 3.

Computation of the Lower Bound

	PI1	PI2	PI3	PI4	PI5	PI6
minterm1	1	0	0	0	0	1
minterm2	1	1	0	0	0	0
minterm3	0	1	1	0	0	0
minterm4	0	0	1	1	0	0
minterm5	0	0	0	1	1	0
minterm6	0	0	0	0	1	1

For cyclic matrices, lower bound is 2 or more, as shown in the following theorem.

Theorem 3: The lower bound for a cyclic matrix is at least 2.
There is only one case when the lower bound is 1; constraint matrix contains a full column of ones. In this case, the matrix cannot be cyclic because the column with all ones dominates all the others. The matrix can therefore be reduced to one column, which is obviously essential. Thus, such a matrix is not cyclic.

Computation of the Lower Bound

There are several heurists for computing the lower bound for the coving problem that have been proposed. Steps are as follows.

1. Add a field w to each row, whose value is equal to the number of 1's in the row
2. Choose the row with minimum w. Let it be r_i. If there are multiple rows with same value, choose the one from the top.
3. Delete all rows r_j such that r_i and r_j have at least one column where both of them have a 1. Also, delete r_i.
4. Repeat step 2 and 3 until no more rows remain.

The number of rows selected in Step-2 is the lower bound of the cover.

Computation of the Lower Bound

The key feature of this algorithm is it just chooses the "shortest" row, that is, the row with the fewest nonzero columns, and breaking ties in ascending order.

The basic motivation of choosing the "shortest" row first, comes from the fact-- shorter the row is, higher is the probability of involving a column to cover it. If there is a row of *w*=1, then one column is mandatory to cover it. All rows with at least one common column of 1 are deleted because, all the rows can be covered with the common columns. This makes the solution faster because we eliminate many rows quickly.

However, this also leads to inaccuracy explained as follows. Let row1 be selected to be deleted. Let row2 has common column as column1 (with row1) and row3 has common column as column2 (with row1). So the lower bound estimate in this case is 1; deletion of row1 will also eliminate row2 and row3 because of common columns having 1. Also, let row2 and row3 be singletons. To cover row1 and row2 if we select column1, then selection of column2 is mandatory for row3. Similarly, to cover row1 and row3 if we select column2, then selection of column1 is mandatory for row2. Therefore, we require two columns in this case, which is not reflected in the estimate.

Computation of the Lower Bound

Let is consider the constraint matrix given below. The weights w are also shown in the matrix.

	PI1	PI2	PI3	PI4	PI5	PI6	
minterm1	1	0	0	0	1	0	$w=2$
minterm2	1	1	0	1	0	0	$w=3$
minterm3	0	1	1	0	0	0	$w=2$
minterm4	0	0	0	1	1	1	$w=3$
minterm5	0	0	1	1	0	0	$w=2$
minterm6	0	1	0	0	0	1	$w=2$

In this case, we start with row1 (minterm1) which has the minimum weight and occurs first in chronological order among all other rows having equal weight. Now, rows minterm2 and minterm4 are also eliminated because of common columns column1 (PI1) and column5 (PI5), respectively. In the next iteration, row3 is selected. Now, rows minterm6 and minterm5 are also eliminated because of common columns column2 (PI2) and column3 (PI3), respectively. As there are no more rows, Lower Bound = 2 and comprises {1, 3}. It may be noted that in this case the bound is not sharp, as we require at least three columns in the cover.

Example of Branch and Bound applied to Unate covering

Let us consider the constraint matrix given below. It may be noted that this matrix cannot be cannot be further simplified. If we apply the quick lower bound estimate , we get Lower Bound = 4 comprising {1, 3,5,7}.

	PI1	PI2	PI3	PI4	PI5	PI6	PI7	PI8	PI9	PI10	PI11
minterm1	1	1	0	0	0	0	0	0	0	0	0
minterm2	0	1	1	0	0	0	0	0	0	0	0
minterm3	0	0	1	1	0	0	0	0	0	0	0
minterm4	1	0	0	1	0	0	0	0	0	0	0
minterm5	0	0	0	0	1	1	0	0	0	1	0
minterm6	0	0	0	0	0	1	1	0	1	0	0
minterm7	0	0	0	0	0	0	1	1	0	0	0
minterm8	0	0	0	0	0	1	0	1	0	1	1
minterm9	0	0	0	0	1	0	0	0	1	1	1
minterm10	0	0	0	0	1	0	0	1	1	0	0
minterm11	0	0	0	0	1	0	1	0	0	0	1
minterm12	1	0	0	0	0	0	0	0	0	0	1

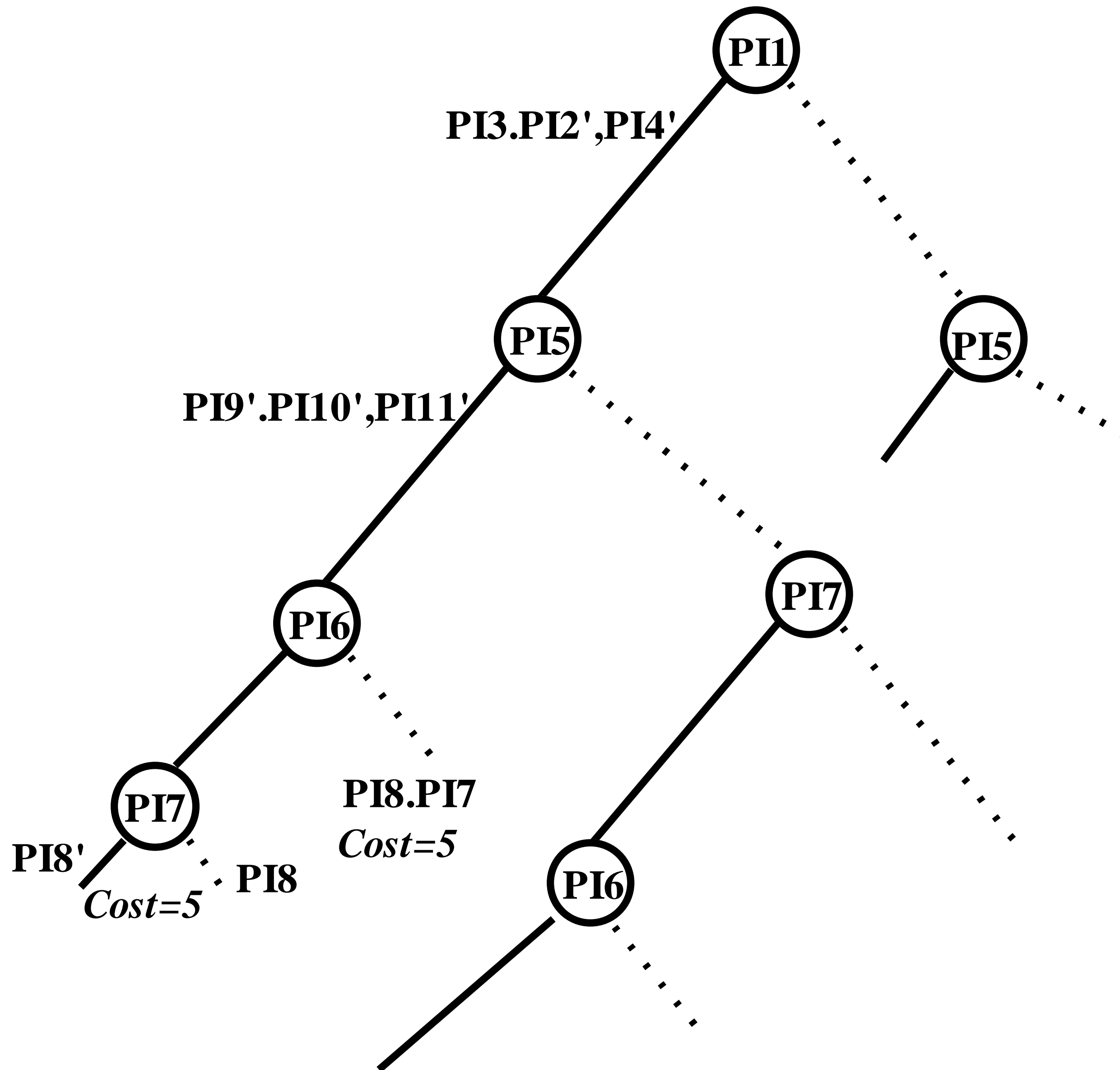
Example of Branch and Bound applied to Unate covering

Now we start exploring the solution space. Let is start with selecting PI1 (i.e., PI1=1). With PI1 being taken, the following happens in the matrix

- rows minterm1, minterm4, and minterm112 are covered,
- columns PI2 (by PI3) and PI4 (by PI3) are dominated
- column PI3 becomes essential.

The Binary search tree illustrating the search in the solution space by the branch and bound algorithm is shown in next figure. The left edge (right dotted edge) indicates the column being (not being) considered.

Example of Branch and Bound applied to Unate covering



Example of Branch and Bound applied to Unate covering

The left edge from root note indicates PI1 being taken. Also in the edge it is marked that "PI3, PI2',PI4"—this indicates that because of taking PI1, PI3 has to be considered and PI2 and PI4 get eliminated.

After reduction (i.e., taking PI1, PI3 and eliminating PI2, PI3) we get the following matrix. As of now, the cost of the solution is 2 (PI1, PI3).

	PI5	PI6	PI7	PI8	PI9	PI10	PI11
minterm5	1	1	0	0	0	1	0
minterm6	0	1	1	0	1	0	0
minterm7	0	0	1	1	0	0	0
minterm8	0	1	0	1	0	1	1
minterm9	1	0	0	0	1	1	1
minterm10	1	0	0	1	1	0	0
minterm11	1	0	1	0	0	0	1

Example of Branch and Bound applied to Unate covering

In this matrix Lower Bound = 2 comprising {5,7}. So if we explore on this matrix the solution cost lower bound is 2+2=4. As this value is equal to that of the Lower Bound on the initial matrix, we explore on the search space. Let us now consider column PI5. With PI5 being taken, the following happens in the matrix

- rows minterm9, minterm10, and minterm11 are covered,
- columns PI9 (by PI6), PI10 (PI6) and PI11 (by PI6) are dominated

The left edge from root note indicates PI5 being taken (tree). Also in the edge it is marked that "PI9', PI0',PI1''"—this indicates that because of taking PI5, PI9, PI10 and PI11 get eliminated.

After reduction (i.e., taking PI5 and eliminating PI92, PI10, PI1) we get the following matrix. As of now the cost of the solution is 3 (PI1, PI3,PI5).

	PI6	PI7	PI8
minterm6	1	1	0
minterm7	0	1	1
minterm8	1	0	1

Example of Branch and Bound applied to Unate covering

- In this matrix Lower Bound = 1 comprising {6}. So if we explore on this matrix the solution cost lower bound is 3+1=4. As this value is equal to that of the Lower Bound on the initial matrix, we explore on the search space. If we consider PI6, the following happens in the matrix
    - row minterm8 is covered,
    - Column PI7 or PI8 needs to be taken in the cover.

- This is the terminal case as all rows are covered. This is solution is {PI1,PI3,PI5,PI6,PI7} (left edge of PI7) in the tree or {PI1,PI3,PI5,PI6,PI8} (right edge of PI7).

- In both the solutions, the cost is 5, which is higher than the initial expected lower bound. Therefore, we need to retract. We may select an alternative path where, PI6=0 (right edge of PI6). It is easy to observe that in this case the solution is {PI1,PI3,PI5,PI7,PI8}; as the cost is 5 we retract. We may select an alternative path where, PI5=0 (right edge of PI5).

Example of Branch and Bound applied to Unate covering

- Similarly, the whole tree can be created. In this example, if the whole tree is created, it may be noted that the cost at all branches (solution space) is 5.

- Therefore, the whole tree will be explored and in the end, it will be concluded that the lower bound (=4) given by the estimate is not sharp and a solution (of cost 5) will be taken as cover.

- However, it is not always the case. In the Question and Answer part of the lecture we will provide a modified lower bound estimate algorithm and show how, in the same example (being considered in this sub-section) some paths need not be explored

Question and Answers

- **Question:** The lower bound estimate algorithm discussed in this lecture may not always give a sharp bound. Suggest suitable modifications and show that better bounds can be obtained. Also show using an example, how benefit is achieved in branch and bound algorithm using the modification .

Answer

The following 4 steps were present in the lower bound estimate algorithm discussed in the lecture. The modification is highlighted as bold.

1. Add a field w to each row, whose value is equal to the number of 1's in the row
2. Choose the row with minimum w. Let it be r_i. If there are multiple rows with same value, choose the one from the top.
3. Delete all rows r_j such that r_i and r_j have at least one column where both of them have a 1. Also, delete r_i. **If, after deletion no more rows remain, then check if there exists a column PI say, such that PI has all 1s. If there is no such column then 1 needs to be added the lower bound.**
4. Repeat step 2 and 3 until no more rows remain.

Question and Answers

The motivation of the extension is explained by the following matrix.

	PI1	PI2	PI3
minterm1	1	1	0
minterm2	0	1	1
minterm3	1	0	1

In the above matrix, if we apply the lower bound estimate algorithm discussed in the lecture, then we get the answer as 1; the algorithm stops after 1 iteration because row1 is selected and row2 and row3 get eliminated. However, it may be noted one column cannot cover the three rows because no column PI exists, such that PI has all 1s. Therefore, in this case, we increment the cost by 1; two rows can cover. It may be noted that if the matrix has more rows and columns, then the addition required in the cost may be more than 2. However to keep the algorithm simple we compromise on accuracy and add just 1; calculating more accurate value to be added, requires more computation steps.

Question and Answers

Let us again consider the same constraint matrix example. If we apply the modified lower bound estimate algorithm, we get Lower Bound = 4 comprising {1, 3,5,7} (same as the original algorithm of the lecture).

	PI1	PI2	PI3	PI4	PI5	PI6	PI7	PI8	PI9	PI10	PI11
minterm1	1	1	0	0	0	0	0	0	0	0	0
minterm2	0	1	1	0	0	0	0	0	0	0	0
minterm3	0	0	1	1	0	0	0	0	0	0	0
minterm4	1	0	0	1	0	0	0	0	0	0	0
minterm5	0	0	0	0	1	1	0	0	0	1	0
minterm6	0	0	0	0	0	1	1	0	1	0	0
minterm7	0	0	0	0	0	0	1	1	0	0	0
minterm8	0	0	0	0	0	1	0	1	0	1	1
minterm9	0	0	0	0	1	0	0	0	1	1	1
minterm10	0	0	0	0	1	0	0	1	1	0	0
minterm11	0	0	0	0	1	0	1	0	0	0	1
minterm12	1	0	0	0	0	0	0	0	0	0	1

Question and Answers

After third iteration of the modified lower bound estimate algorithm (i.e., deleting rows corresponding to 1,3 and 5), we have the following matrix

	PI1	PI2	PI3	PI4	PI5	PI6	PI7	PI8	PI9	PI10	PI11
minterm7	0	0	0	0	0	0	1	1	0	0	0

After eliminating minterm7, we exhaust all rows, but need not add 1 to the cost because there are two rows PI7 and PI8, which have all 1s.

Question and Answers

As discussed in the lecture, we start exploring the solution space by selecting PI1 (i.e., PI1=1). With PI1 being taken, the following happens in the matrix

- rows minterm1, minterm4, and minterm112 are covered,
- columns PI2 (by PI3) and PI4 (by PI3) are dominated
- column PI3 becomes essential.

After reduction (i.e., taking PI1, PI3 and eliminating PI2, PI3) we get the following matrix. As of now, the cost of the solution is 2 (PI1, PI3).

	PI5	PI6	PI7	PI8	PI9	PI10	PI11
minterm5	1	1	0	0	0	1	0
minterm6	0	1	1	0	1	0	0
minterm7	0	0	1	1	0	0	0
minterm8	0	1	0	1	0	1	1
minterm9	1	0	0	0	1	1	1
minterm10	1	0	0	1	1	0	0
minterm11	1	0	1	0	0	0	1

Question and Answers

In this matrix Lower Bound = 2 comprising {5,7}; this is same, using the original lower bound computation algorithm as well as the modified one. So if we explore on this matrix the solution cost lower bound is 2+2=4. As this value is equal to that of the Lower Bound on the initial matrix, we explore on the search space. Let us now consider column PI5. With PI5 being taken, the following happens in the matrix

- rows minterm9, minterm10, and minterm11 are covered,
- columns PI9 (by PI6), PI10 (PI6) and PI11 (by PI6) are dominated

After reduction (i.e., taking PI5 and eliminating PI92, PI10, PI1) we get the following matrix. As of now the cost of the solution is 3 (PI1, PI3,PI5).

	PI6	PI7	PI8
minterm6	1	1	0
minterm7	0	1	1
minterm8	1	0	1

Question and Answers

In this matrix Lower Bound = 2 comprising {5,7}; this is same, using the original lower bound computation algorithm as well as the modified one. So if we explore on this matrix the solution cost lower bound is 2+2=4. As this value is equal to that of the Lower Bound on the initial matrix, we explore on the search space. Let us now consider column PI5. With PI5 being taken, the following happens in the matrix

- rows minterm9, minterm10, and minterm11 are covered,
- columns PI9 (by PI6), PI10 (PI6) and PI11 (by PI6) are dominated

After reduction (i.e., taking PI5 and eliminating PI92, PI10, PI1) we get the following matrix. As of now the cost of the solution is 3 (PI1, PI3,PI5).

	PI6	PI7	PI8
minterm6	1	1	0
minterm7	0	1	1
minterm8	1	0	1

Question and Answers

In this matrix Lower Bound = 1 comprising {6}; this is the value obtained using the using the original lower bound computation algorithm of discussed in the lecture.

Using the modified algorithm the Lower Bound = 2 (in this case 1 is added to the cost because there is no column with all 1s.)

Therefore, if we explore on this matrix the solution cost lower bound is 3+2=5. As this value is higher than the Lower Bound on the initial matrix, we do not explore on the search space.

As shown in the lecture, exploring in path PI1=1, PI5=1….., would have the cost of 5. As this value is available without actually exploring the path, when we use the modified bound computation algorithm, we save computation time in the branch and bound algorithm.

Design Verification and Test of Digital VLSI Circuits NPTEL Video Course

Module-III

Lecture-IV

Heuristic Minimization of Two-Level Circuits

Introduction

•In the last lecture, we have discussed schemes to minimize Boolean functions for two level implementations.

•However, the major problem in exact minimization is the potentially very large number of minterms and prime implicants.

•exact minimization procedure involves two basic steps namely, finding prime implicants and determining a minimal subset of prime implicants, which covers the Boolean function.

•For a circuit with *n* inputs there may be of the order of 2^n minterms, thereby making the tabular method of finding prime implicants, an exponentially complex method. Similarly, determining a minimal subset of prime implicants also requires minterms, making the procedure complex. Broadly speaking, for most functions with more than, say, 15 inputs, there are too many primes and minterms, rendering exact minimization schemes prohibitive.

Introduction

• To cater to this problem, heuristic methods for Boolean function minimization have been proposed which avoid computing all the primes and all the minterms in an attempt to avoid the computational cost.

• The idea is to successively modify a given initial cover of the function, until a suitable stopping criterion (in terms of number of literals) is met. The drawback is the inability to guarantee the cheapest solution, though a careful choice of the algorithms may actually provide solutions very close to the optimum (when the optimum is known) in a very reasonable time.

• In this lecture, we will discuss a simplified version of such a heuristic called ESPRESSO.

Local Search

•Heuristic based logic minimization techniques are based on local search.

•The algorithms that start with an initial solution and try to find a better solution by applying successive modifications are called local search algorithms.

•Local search algorithms are very common in optimization domain and logic minimization is an important optimization problem in VLSI area.

• In the "Selecting a Subset of Primes" problem, the search space consists of a subset of primes that cover all the minterms of the Boolean equation.

•A point in the search space is a subset of primes. The subsets, which covers all the minterms are valid solutions and the ones which do not cover all the minterms are not the valid solutions. The set of all points of the search space that are valid solutions form the so-called feasible region. The complement is the infeasible region.

Local Search

•Local search relies on distance between two points in the search space.

•In case of the "Selecting a Subset of Primes" problem, the distance between two points would be the number of minterms that are covered in one subset of primes, but not in the second or vice versa (i.e., the cardinality of the symmetric difference of the two sets).

•Different problems have generally different definitions of distance, and it is also possible to define different distances for the same problem.

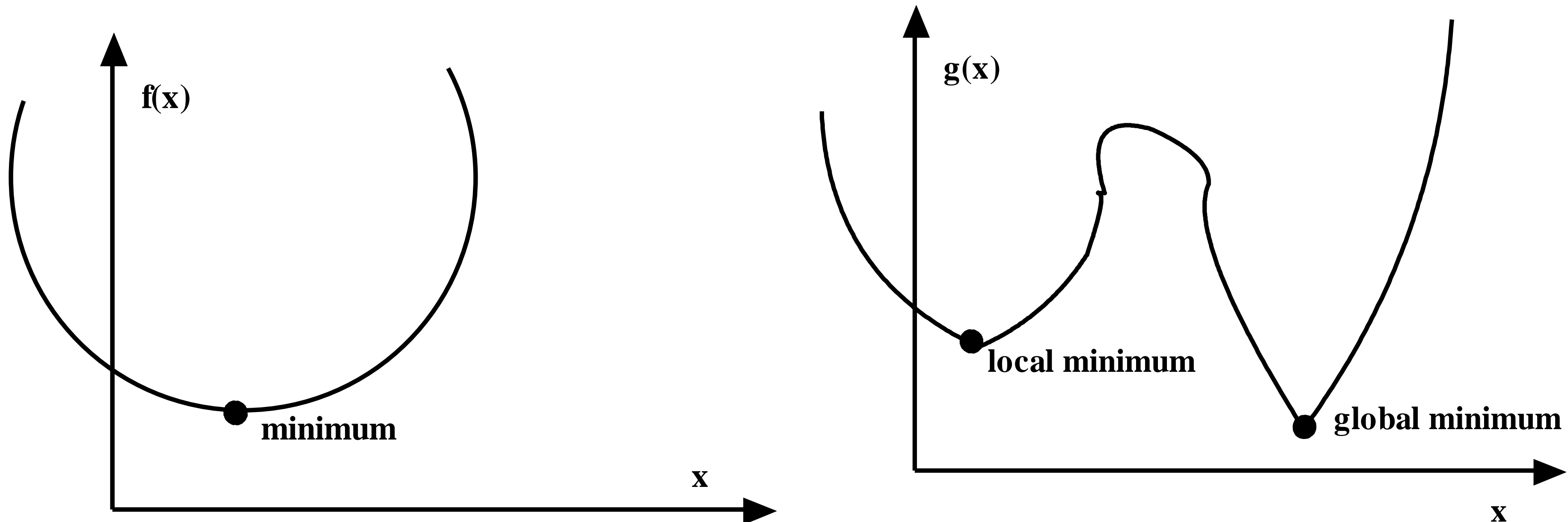
•Once a definition of distance is given, we can define the neighborhood of a point *p* as all points that are less that some distance *k* from *p*.

•Starting from an initial random valid solution, the algorithm examines its neighborhood for a feasible point whose cost is lower than the current cost. If one is found, it is assumed as the new starting point and the process is repeated until a stopping criterion is met.

Local Search

Depending on the problem and the definition of neighborhood, various degrees of optimality of the solution can be guaranteed.

For example, if we are interested in the minimum of a convex function of a real variable as the one shown in next figure, local search optimization consists of moving downward along the curve and is guaranteed to find the minimum value. In this case, local search finds the solution which is the best.



Local Search

However, if the function is not convex, then local search may get stuck in a local minimum.

Once the local search reaches the local optimum point it terminates because it cannot see a better solution in the neighborhood. However, if it takes a worse solution compared to the local optimum then there is a chance that we may reach the global optimum.

In the first case, we say that the solution is minimum, whereas in the second case we say that the solution is minimal.

In general, one interesting property of the solution of a local search algorithm is local optimality. A solution is locally optimal if its neighborhood does not contain any solution of lower cost. In order to guarantee local optimality, it is sufficient to use the stopping criterion--when there is no cheaper solution in the neighborhood of the current solution then terminate.

Local Search applied to Logic Minimization

•Local search based logic optimization starts with a random subset of the primes, which is a cover. The cost of a solution is the number of cubes.

•If two solutions have the same number of cubes then number of literals can be used as a secondary cost.

•The neighborhood of a solution is the subset of primes that are obtained from the solution by adding (or removing) exactly one literal to (or from) one of the cubes. A new cover thus obtained is a valid solution in the search space (and considered) if it represents the same function (i.e., covers the same minterms).

•The solutions with more cubes than the current cover, are not considered.

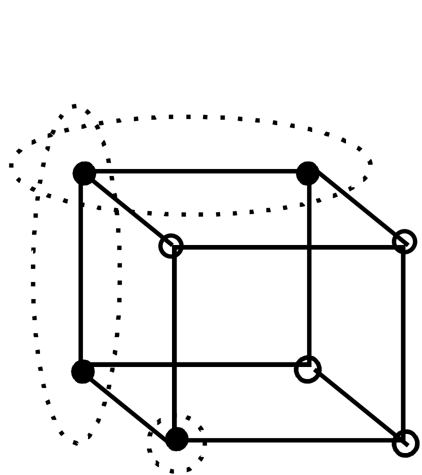
Boolean function $f(xyz) = \bar{x}\bar{y}\bar{z} + \bar{x}y\bar{z} + \bar{x}yz + xyz$ and the dotted lines represent the initial cover. The cubical representation of the function is as follows

xyz	$f(x,y,z)$
000	1
01–	1
–11	1

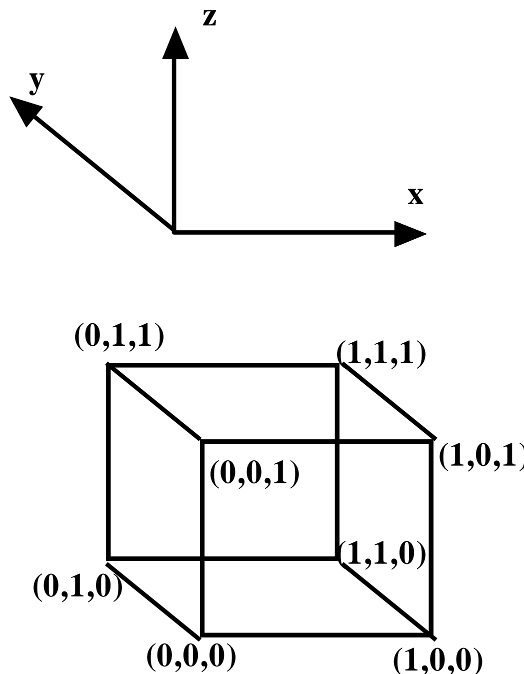
Local Search applied to Logic Minimization

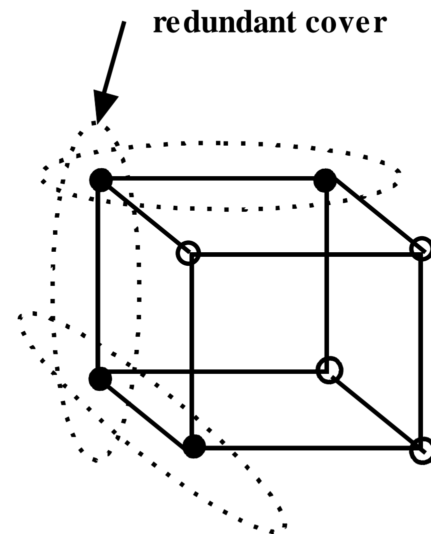
Boolean function $f(xyz) = \bar{x}\bar{y}\bar{z} + \bar{x}y\bar{z} + \bar{x}yz + xyz$ and the dotted lines represent the initial cover. The cubical representation of the function is as follows

xyz	$f(x,y,z)$
000	1
01–	1
–11	1



(a) Initial cover





(c) Cover after expansion of an implicant

Local Search applied to Logic Minimization

- Now let us remove the literal "y" from cube $\overline{x}\,\overline{y}\,\overline{z}$ leading to $\overline{x} - \overline{z}$.
- It may be noted that we can do so because $\overline{x} - \overline{z}$ comprises minterms $\overline{x}\,\overline{y}\,\overline{z}$ and $\overline{x}y\overline{z}$, which are to be covered in the function under question.
- After removing the literal "y" from cube we get the cover as shown in last figure (b). Now we can remove the redundant cover, i.e., remove the output literal of $\overline{x}y-$, thus effectively removing the cube from the cover; we have the cover as follows.
- Here, cover is the optimum solution.

xyz	$f(x,y,z)$
$0-0$	1
-11	1

Local Search applied to Logic Minimization

The function is as follows

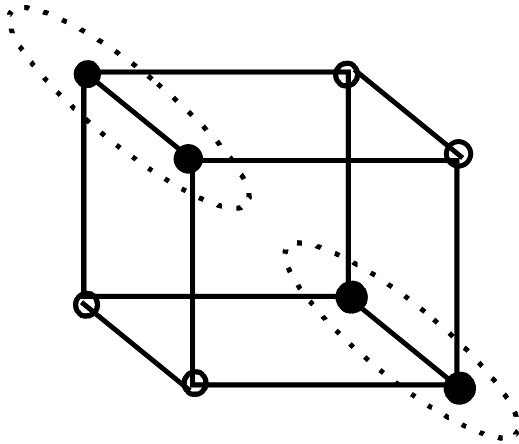
$$f_1(xyz) = \bar{x}\bar{y}z + \bar{x}yz + x\bar{y}\bar{z} + xy\bar{z}; \qquad f_2(xyz) = \bar{x}\bar{y}\bar{z} + \bar{x}\bar{y}z + x\bar{y}\bar{z} + \bar{x}yz + xyz$$

The cubical representation is

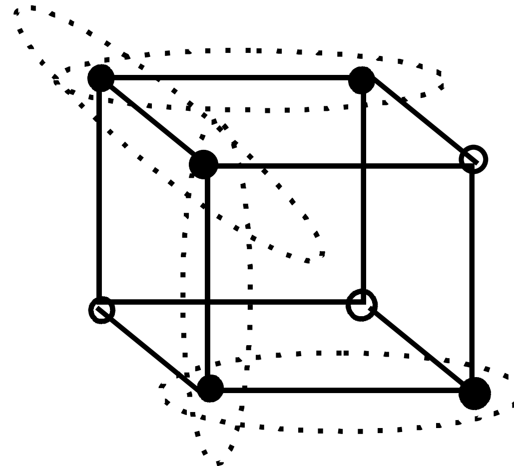
xyz	$f_1 f_2$
$0-1$	10
$1-0$	10
$00-$	01
-00	01
-11	01

The initial cover of the two outputs of the function is illustrated in next Figure.

Local Search applied to Logic Minimization



(a) Initial Cover f_1



(a) Initial Cover f_2

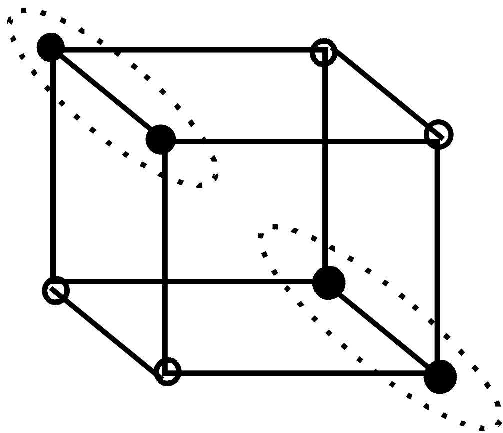
xyz	$f_1 f_2$
0–1	11
1–0	10
00–	01
–00	01
–11	01

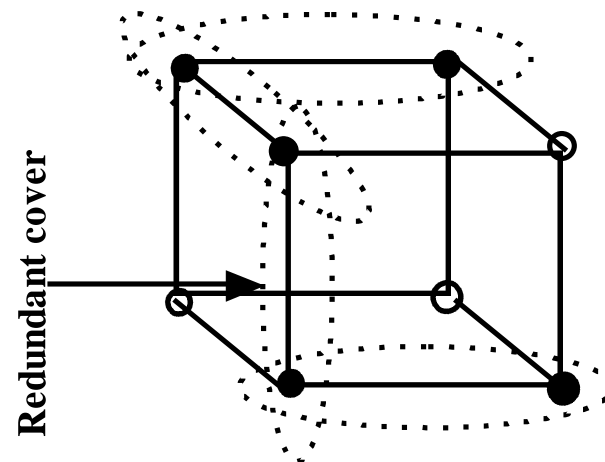
It may be noted that there is no way to improve the solution by expanding the input parts of the cubes or reducing their output parts. However, if we expand the output part of the first cube (i.e., 0–1|10), we obtain the cover above.

Local Search applied to Logic Minimization

The cover is represented in next Figure. We see that 00–|01 is now redundant and can be eliminated, yielding the following cover.

xyz	$f_1 f_2$
$0-1$	11
$1-0$	10
-00	01
-11	01





It may be noted that the second cover (after expansion of the output of the first cube) has one cube less than the initial cover.

A Simple Local Search Algorithm for Logic Minimization

- The examples in the last section suggest that a procedure based only on the simple moves that immediately decrease the cost is effective.

- However, in the real scenario we need to consider a wider variety of moves, and combine them into sequences of moves that eventually lead to the desired reductions in cost. In other words, we may also need to consider moves that may not immediately reduce the cost.

- Now we discuss an algorithm which applies simple local search algorithm for logic minimization.

A Simple Local Search Algorithm for Logic Minimization

The algorithm has the following basic phases:

1. EXPAND: This step expands implicants to their maximum size. Implicants covered by an expanded implicant are removed from further consideration. Quality of result depends on order of implicant expansion. Heuristic methods are used to determine order.

2. IRREDUNDANT COVER : Irredundant cover (i.e., no proper subset is also a cover) is extracted from the expanded primes.

3. REDUCE: There might exist another cover with fewer terms or fewer literals. Shrink prime implicants to smallest size that still covers the minterms

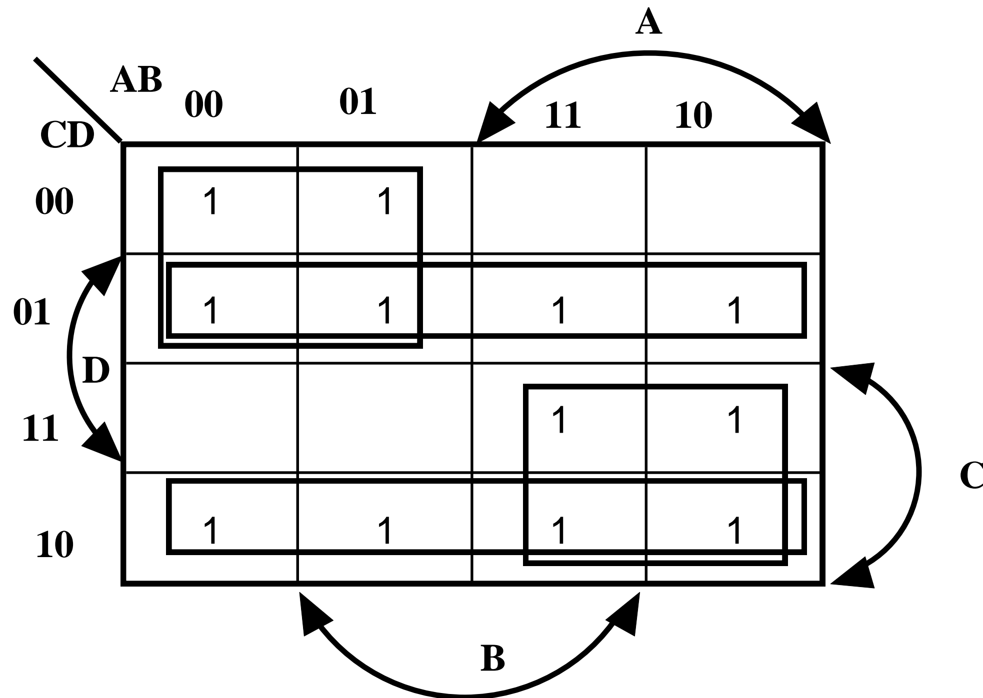
4. Repeat the sequence REDUCE/EXPAND/IRREDUNDANT COVER to find alternative solutions. Keep doing this as long as new cover improve on last solution

A Simple Local Search Algorithm for Logic Minimization: Example

Let us consider the function $f(A,B,C,D)$ having minterms as $\{0,1,2,4,5,6,9,10,11,13,14,15\}$.

Let the initial cover be the ones shown in the boxes of the K-map -- $\overline{A}\overline{C}, \overline{C}D, AC, C\overline{D}$

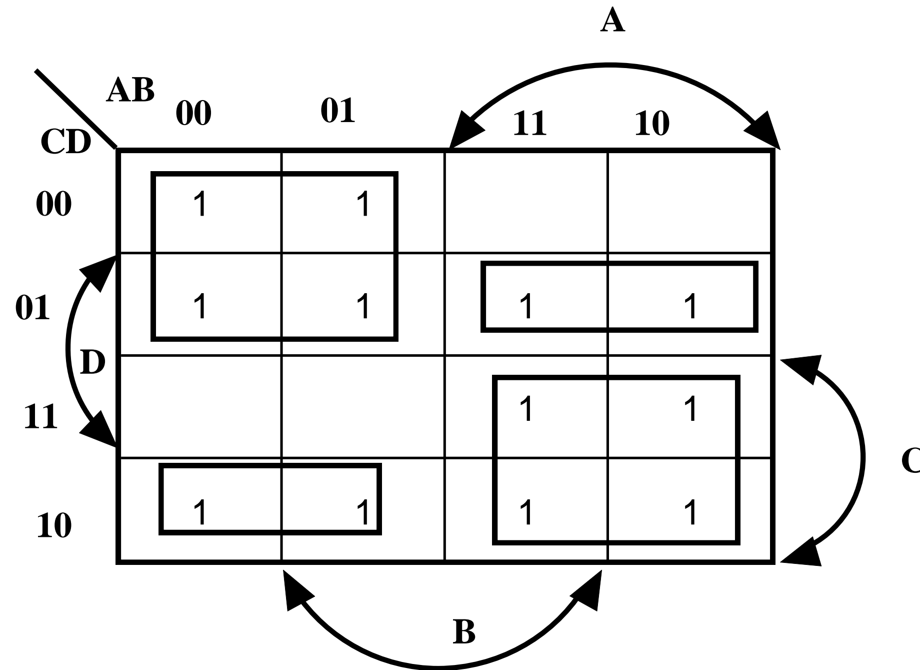
.



A Simple Local Search Algorithm for Logic Minimization: Example

Now we reduce the covers, however, keeping in mid that all the minterms are covered

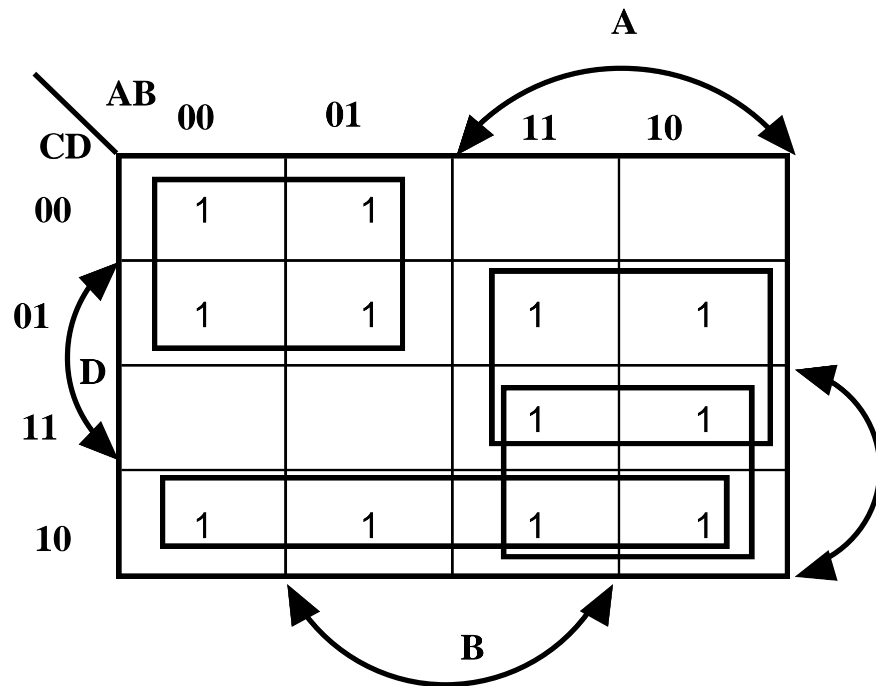
We use the literal A for the reduction making the cover as $\overline{A}\,\overline{C}, A\overline{C}D, AC, \overline{A}C\overline{D}$; $\overline{C}D$ is changed to $A\overline{C}D$ and $C\overline{D}$ is changed to $\overline{A}C\overline{D}$.

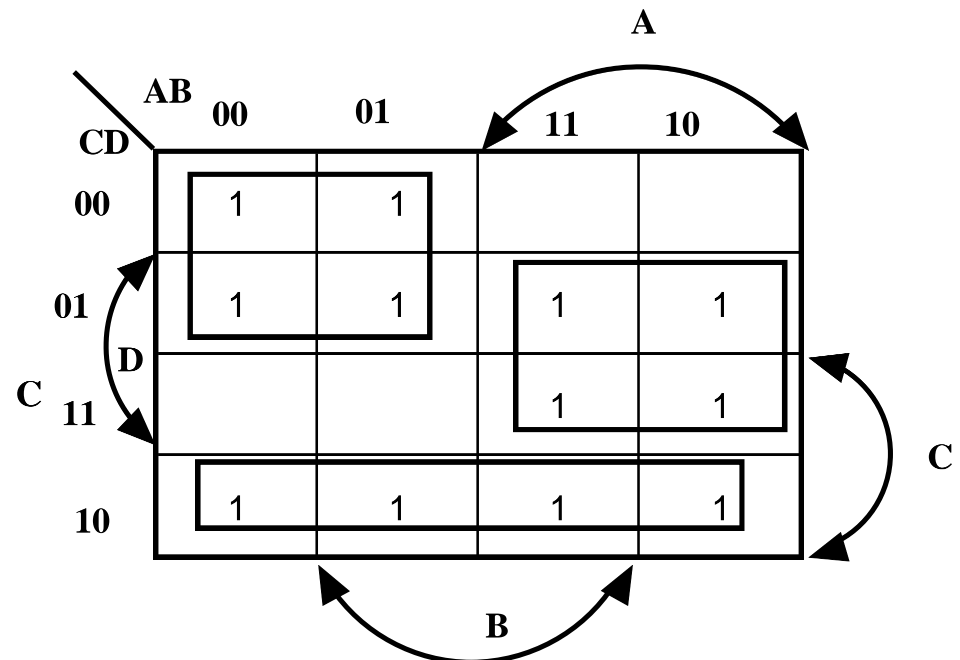


A Simple Local Search Algorithm for Logic Minimization: Example

Now we expand $A\overline{C}D$ using literal C resulting in cover AD and $\overline{A}C\overline{D}$ using literal A resulting in cover $C\overline{D}$.

It may be now noted that cover AC is redundant can be eliminated. The cover after one iteration. The same sequence is repeated till the cost of solution is as expected.





Question and Answer

Question: Why is the “Simple Local Search Algorithm for Logic Minimization” called anytime algorithm. ?

Answer:

“Simple Local Search Algorithm for Logic Minimization” called anytime algorithm because at any iteration of the algorithm, even at the initial state, we have a valid solution. One need not wait for any (k, say) number of iterations for a valid solution, only the quality of solution improves.

Design Verification and Test of Digital VLSI Circuits NPTEL Video Course

Module-III

Lecture-V

Finite State Machine Synthesis

Introduction

•In the last four lectures, we discussed two level minimization of Boolean functions that directly map to combinational circuits. In this lecture, we will present techniques for Finite State Machine (FSM) synthesis.

An FSM of Mealy type is a 6-tuple $\langle I, S, \delta, S_0, O, \lambda \rangle$, where

I is the input alphabet, i.e., a finite, non-empty set of input values;

S is the (finite, non-empty) set of states

$\delta : S \times I \rightarrow S$ is the next-state function

$S_0 \subseteq S$ is the set of initial (reset) states

O is the output alphabet

$\lambda : S \times I \rightarrow O$ is the output function.

Introduction

•For a Moore type machine, the outputs do not depend on the present inputs, i.e., the outputs depend only on the state value.

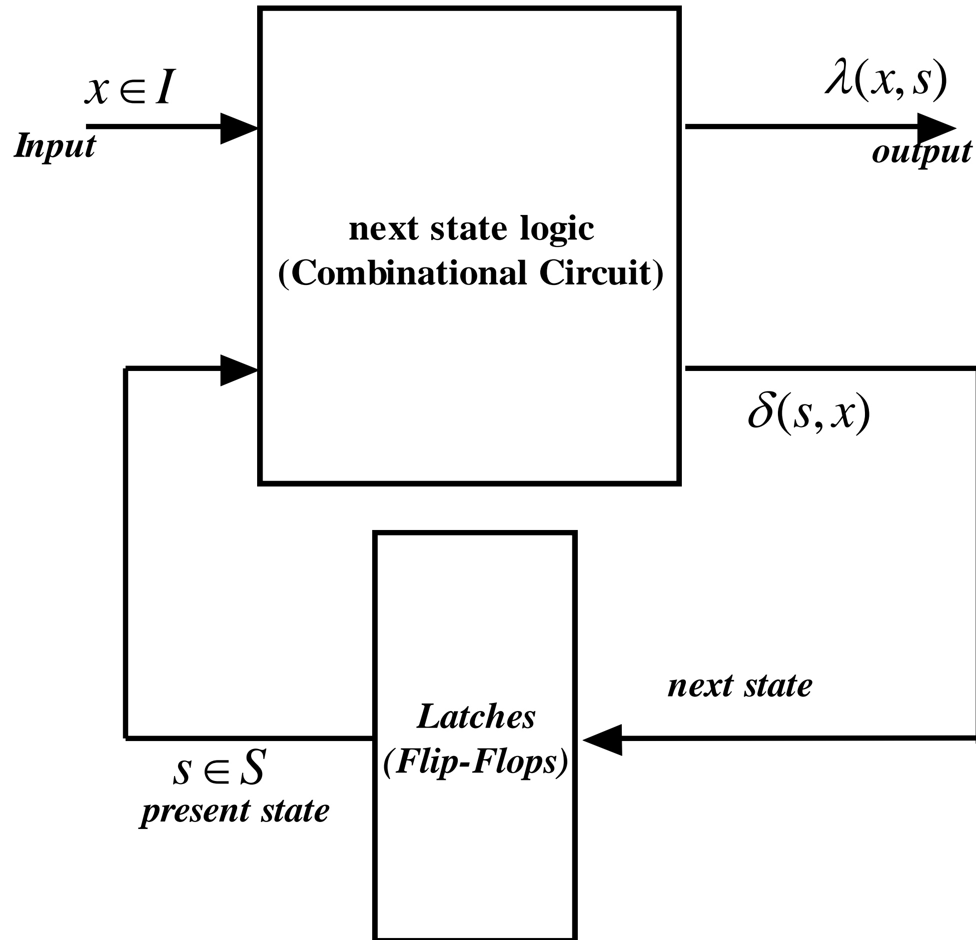
•In this lecture, we shall mostly deal with Mealy machines, because they are more general.

•The algorithms we shall discuss will work equally well for both types of machines.

FSM synthesis involves four basic steps

1. State minimization by merging equivalent states
2. State encoding
3. Determination of Boolean functions for representing next state and output.
4. Two level minimization of these functions.

Introduction



The block diagram of a sequential circuit in terms of $\langle I, S, \delta, S_0, O, \lambda \rangle$.

An example: Illustration of FSM synthesis

•Various representations of FSMs are in use like State Transition Graph (STG), flow table, cube table etc. and STG is one of the most widely used scheme for pictorial representation.

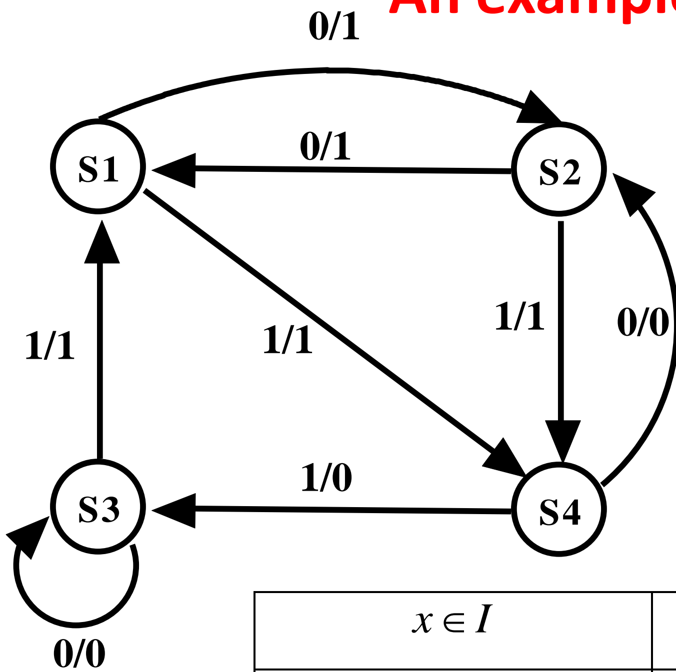
•Next Figure illustrates a simple STG representation of an FSM.

•In STG, every node corresponds to a state and every arc corresponds to a transition. The label on the arc indicates the input that enables the transition and the output produced when the transition takes place.

•The cube table representation is suitable for algorithms used for FSM synthesis.

•The cube table has inputs and present state versus next state and output.

An example: Illustration of FSM synthesis



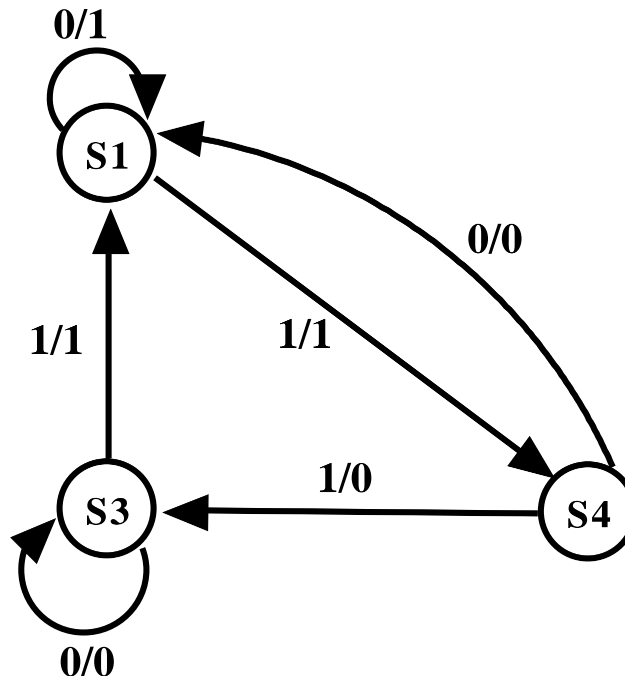
$x \in I$	Present state $s \in S$	Next state $s' \in S$	Output $o \in O$
0	S1	S2	1
1	S1	S4	1
0	S2	S1	1
1	S2	S4	1
0	S3	S3	0
1	S3	S1	1
0	S4	S2	0
1	S4	S3	0

An example: Illustration of FSM synthesis

- If we closely observer the STG, we may note that state S1 and S2 are equivalent.

- We will define state equivalence and algorithms to determine such states formally, in subsequent sections of this lecture.

- For the time, we may broadly say that if two equivalent states are merged there is no change in input/output behavior of the FSM. If we merge states S1 and S2 of, we obtain a reduced FSM.



An example: Illustration of FSM synthesis

•Once we obtain a minimized STG, we need to assign encodings to the states. Now we will see two different encodings and the effect in the cost of the Boolean functions implementing them.

As there are three states, we need two encoding bits as $s_0 s_1$. Let us consider the encoding as

$$s_0 = 0 s_1 = 0 : S1$$
$$s_0 = 1 s_1 = 0 : S3$$
$$s_0 = 0 s_1 = 1 : S4$$

$x \in I$	Present state $s_0 s_1$	Next state $s'_0 s'_1$	Output $o \in O$
0	00	00	1
1	00	01	1
0	10	10	0
1	10	00	1
0	01	00	0
1	01	10	0

An example: Illustration of FSM synthesis

The Boolean functions representing the next state bits and the output in terms of minterms are as follows.

$$s'_0 : f_0(x, s_0, s_1) = x\overline{s_0}s_1 + \overline{x}s_0\overline{s_1}$$

$$s'_1 : f_1(x, s_0, s_1) = x\overline{s_0}\,\overline{s_1}$$

$$O : f_O(x, s_0, s_1) = \overline{x}\,\overline{s_0}\,\overline{s_1} + x\overline{s_0}\,\overline{s_1} + xs_0\overline{s_1}$$

As state-encoding 11 is unused we have two don't cares as $\overline{x}s_0s_1, xs_0s_1$. By applying the two-level minimization techniques discussed in the last few lectures, on the above Boolean functions, we get the following

$$s'_0 : f_0(x, s_0, s_1) = xs_1 + \overline{x}s_0$$

$$s'_1 : f_1(x, s_0, s_1) = x\overline{s_0}\,\overline{s_1}$$

$$O : f_O(x, s_0, s_1) = \overline{s_0}\,\overline{s_1} + x\overline{s_1}$$

The above-mentioned state encoding involves 11 literals.

An example: Illustration of FSM synthesis

Let us consider another encoding as

$$s_0 = 0 s_1 = 1 : S1$$

$$s_0 = 1 s_1 = 1 : S3$$

$$s_0 = 1 s_1 = 0 : S4$$

$x \in I$	Present state $s_0 s_1$	Next state $s'_0 \, s'_1$	Output $o \in O$
0	01	01	1
1	01	10	1
0	11	11	0
1	11	01	1
0	10	01	0
1	10	11	0

An example: Illustration of FSM synthesis

The Boolean functions representing the next state bits and the output in terms of minterms are as follows.

$$s'_0 : f_0(x, s_0, s_1) = x\overline{s_0}s_1 + \overline{x}s_0s_1 + xs_0\overline{s_1}$$

$$s'_1 : f_1(x, s_0, s_1) = \overline{x}\,\overline{s_0}s_1 + \overline{x}s_0s_1 + xs_0s_1 + \overline{x}s_0\overline{s_1} + xs_0\overline{s_1}$$

$$O : f_O(x, s_0, s_1) = \overline{x}\,\overline{s_0}s_1 + x\overline{s_0}s_1 + xs_0s_1$$

As state-encoding 00 is unused we have two don't cares as $\overline{x}\,\overline{s_0}\,\overline{s_1}, x\overline{s_0}\,\overline{s_1}$. By applying the two-level minimization techniques discussed in the last few lectures, on the above Boolean functions, we get the following

$$s'_0 : f_0(x, s_0, s_1) = \overline{s_0} + x\overline{s_1}$$

$$s'_1 : f_1(x, s_0, s_1) = x + s_0 + s_1$$

$$O : f_O(x, s_0, s_1) = \overline{s_0}s_1 + xs_1$$

The above-mentioned state encoding involves 10 literals. Therefore, the second encoding is better than the first.

State minimization by merging equivalent states

First, we define the concept of equivalence between two states of a given STG of an FSM, and then show how this can be generalized into a procedure for state minimization.

Definition 1: Consider two states s_1 and t_1 of a given FSM, and a *k*-string (sequence of input symbols) $\langle x_0, x_2, \ldots x_{k-1} \rangle$. Suppose the string $\langle x_0, x_2, \ldots x_{k-1} \rangle$ produces one run (i.e., state consecution corresponding to the string) $s = \langle s_1, s_2, \ldots s_{k-1} \rangle$ and another run $t = \langle t_0, t_2, \ldots t_{k-1} \rangle$. Let $os = \langle o_{s0}, o_{s2}, \ldots o_{s(k-1)} \rangle$ and $ot = \langle o_{t0}, o_{t2}, \ldots o_{t(k-1)} \rangle$ be the corresponding output strings. The string $\langle x_0, x_2, \ldots x_{k-1} \rangle$ is said to be a length-*k* distinguishing sequence for states x_1 and t_1 (starting states) if and only if $os \neq ot$ (there is one output value which is not same).

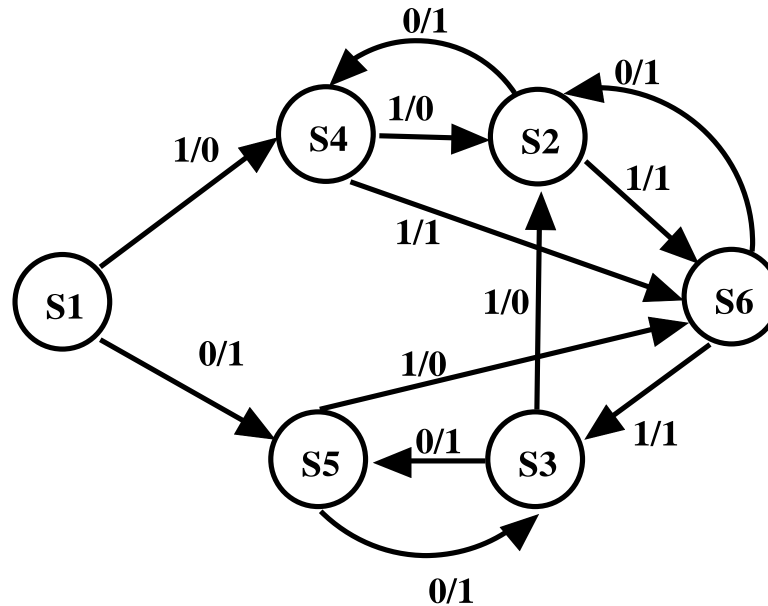
State minimization by merging equivalent states

Definition 2: Two states s_1 and t_1 are *k*-equivalent, written as $s_1 \equiv_k t_1$ if and only if there does not exist a distinguishing sequence of length *k* or less for these states. Two states s_1 and t_1 are equivalent if and only if they are *n*-equivalent, where $n = |S|$.

Definition 3: A binary relation $\Re^k(s_1, t_1) \subset S \times S$ is defined as $\Re^k = \{(s_1, t_1) \mid s_1 \equiv_k t_1\}$. So the relation is an equivalence relation on the set of states *S* and partitions the set into the disjoint equivalence classes. We also denote the equivalence classes of the relation $\Re^k$ by $E_1^k, E_2^k, \ldots E_l^k$.

State minimization by merging equivalent states

Now we will discuss and illustrate the scheme of finding equivalent states using the example of next Figure.



State minimization by merging equivalent states

1. Initially we consider all the states of the STG; let the order be $\langle S1, S2, S3, S4, S5, S6 \rangle$. Now we apply 1 and 0 as input to these states and the runs are $\langle S4, S6, S2, S6, S6, S3 \rangle$ and $\langle S5, S4, S5, S2, S3, S2 \rangle$, respectively. In other words, if we apply 1 (0) to states $S1, S2, S3, S4, S5, S6$ the next sates are $S4, S6, S2, S6, S6, S3$, respectively ($S5, S4, S5, S2, S3, S2$, respectively). The outputs corresponding to 1 and 0 are $\langle 0,1,0,1,0,1 \rangle$ and $\langle 1,1,1,1,1,1 \rangle$, respectively. So we have $S1 \equiv_1 S3 \equiv_1 S5$ and $S2 \equiv_1 S4 \equiv_1 S6$. So, $E_1^1 = \{S1, S3, S5\}$ and $E_2^1 = \{S2, S4, S6\}$.

State minimization by merging equivalent states

2. As the second stage we will try to determine $E_1^2, E_2^2, E_3^2 \ldots$. So, we apply 1 and 0 as input to the equivalent classes $E_1^1 = \{S1, S3, S5\}$ and $E_2^1 = \{S2, S4, S6\}$.

   (i) The runs corresponding to 1 and 0 when applied to $E_1^1 = \{S1, S3, S5\}$ are $\langle S4, S2, S6 \rangle$ and $\langle S5, S5, S3 \rangle$, respectively. The outputs corresponding to 1 and 0 when applied to $E_1^1 = \{S1, S3, S5\}$ are $\langle 0,0,0 \rangle$ and $\langle 1,1,1 \rangle$, respectively. So, $E_1^2 = \{S1, S3, S5\}$; two conditions are satisfied—(i) outputs corresponding to 1 (and 0) are same i.e. all 0s (all 1s), (ii) runs for 1 (and 0), go to states which are in an equivalent class for $\Re^1$.

State minimization by merging equivalent states

(ii) The runs corresponding to 1 and 0 when applied to $E_2^1 = \{S2, S4, S6\}$ are $\langle S6, S6, S3\rangle$ and $\langle S4, S2, S2\rangle$, respectively. The outputs corresponding to 1 and 0 when applied to $E_2^1 = \{S2, S4, S6\}$ are $\langle 1,1,1\rangle$ and $\langle 1,1,1\rangle$, respectively. So, $E_2^2 = \{S2, S4\}, E_2^3 = \{S6\};$ —(i) outputs corresponding to 1 (and 0) are same, (ii) runs for 1 (and 0), from states $S2, S4$ go to states $S6, S6$ (and $S4, S2$) which are in an equivalent class for $\Re^1$. On the other hand, run for 1, from state $S6$ goes to state $S3$, which is in an equivalence class different from the class comprising $S6, S6$.

State minimization by merging equivalent states

4. As the third stage we will try to determine $E_1^3, E_2^3, E_3^3...$. So, we apply 1 and 0 as input to the equivalent classes $E_1^2 = \{S1, S3, S5\}$, $E_2^2 = \{S2, S4\}, E_2^3 = \{S6\}$. Similar, to the procedure of Step 2, we can determine that, there will be no further partitioning in $E_2^2 = \{S2, S4\}, E_2^2 = \{S6\}$, thereby generating $E_1^3 = \{S2, S4\}, E_2^3 = \{S6\}$.

   The runs corresponding to 1 and 0 when applied to $E_1^2 = \{S1, S3, S5\}$ are $\langle S4, S2, S6 \rangle$ and $\langle S5, S5, S3 \rangle$, respectively. The outputs corresponding to 1 and 0 when applied to $E_1^1 = \{S1, S3, S5\}$ are $\langle 0,0,0 \rangle$ and $\langle 1,1,1 \rangle$, respectively. So, $E_3^3 = \{S1, S3\}, E_4^3 = \{S5\}$; run for 1, from state $S5$ goes to state $S6$, which is in an equivalence class different from the class comprising $S4, S2$ (run for 1, from state $S1, S3$ reaches $S4, S2$, respectively) as per $\Re^2$.

State minimization by merging equivalent states

4. If we repeat the procedure (Step-2 or Step-3) on $E_1^3, E_2^3, E_3^3, E_4^3$, we will see that there is no change in the equivalence classes. So we obtain $S1 \equiv_4 S3$ and $S2 \equiv_4 S4$. As there is no change in the classes applying Step-4 (i.e., *k=4*) we terminate and consider the classes of Step-3 as final.

So, finally we have $S1 \equiv_2 S3 \equiv_2 S5$, $S2 \equiv_2 S4 \equiv_2 S6$, $S1 \equiv_3 S3 \not\equiv_3 S5$ and $S2 \equiv_3 S4 \not\equiv_3 S6$.

Note: In this example, we found a terminating condition. In general, we get a terminating condition for the following cases

- There is no change in the equivalence classes after an iteration
- All the equivalence classes have single element; in this case, no state is equivalent with any other.
- After n[th] iteration where $n = |S|$

State minimization by merging equivalent states

So, in this example, we note that there is single run 0 (single input 0) that can distinguish between states $S1, S3, S5$ with $S2, S4, S6$; on giving input 0 to $S1, S3, S5$ we get 0 as output, while on giving input 0 to $S2, S4, S6$ we get 1 as output.

However, there are no single runs that can distinguish between states $S1, S3, S5$; $S1, S3, S5$ cannot generate distinguishing outputs if input is either 0 or 1 (and same in case of $S2, S4, S6$).

Similarly, we can determine that there are no double runs that can distinguish between states $S1, S3, S5$.

State minimization by merging equivalent states

However, there is a double run that can distinguish states $S2, S4$ with $S6$.

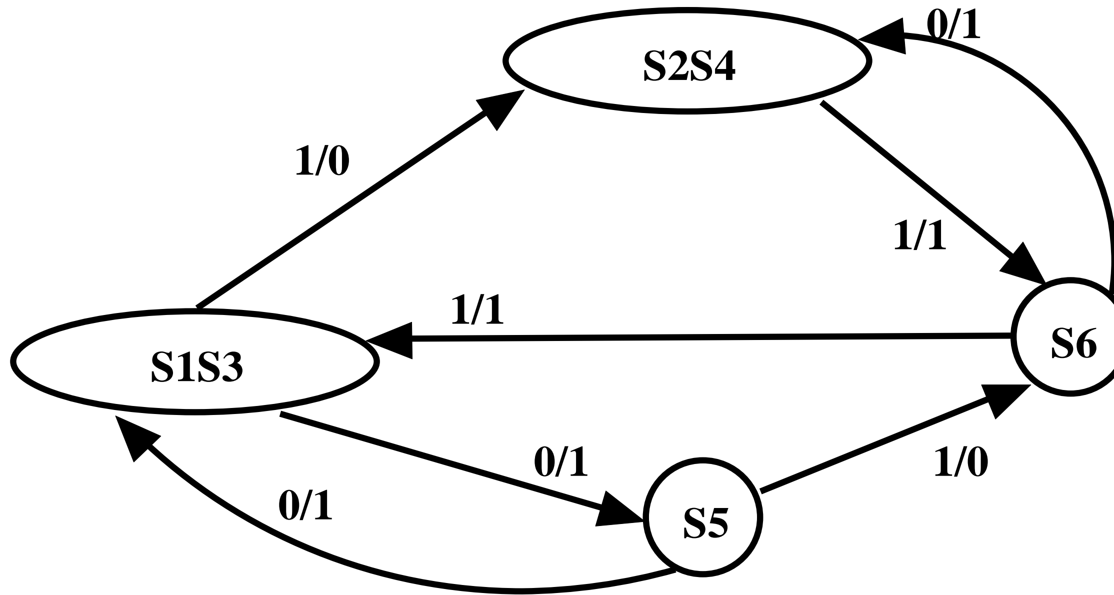
Example: Starting from state $S2$ ($S6$) if we apply 1, we go to state $S6$ ($S3$) generating output 1 (1). Now if we apply input 1, to state $S6$($S3$), we go to state $S3$($S2$) generating output 1 (0)—different output. So run with input "11" from state $S2$ and $S6$ give outputs as "11" and "10", respectively, thereby differentiating $S2$ from $S6$. Similarly, we may verify that is a double run that can differentiate $S4$ from $S6$.

There are triple runs that can distinguish between states $S1, S3$ with $S5$.

Example: Starting from state $S1$ ($S5$) if we apply 1, we go to state $S4$ ($S6$) generating output 0 (0). Now if we apply input 1, to state $S4$ ($S6$), we go to state $S6$($S3$) generating output 1 (1). Now if we apply input 1, to state $S6$($S3$), we go to state $S3$($S2$) generating output 1 (0)—different output. So run with input "111" from state $S1$ and $S5$ give outputs as "011" and "010", respectively, thereby differentiating $S1$ from $S5$. Similarly, we may verify that there are triple runs (no single or double runs) that can differentiate $S3$ from $S5$.

State minimization by merging equivalent states

Therefore, we see that in this example, $S1 \equiv S3$ and $S2 \equiv S4$. We merge $S1$ with $S3$ and $S2$ with $S4$. Next figure represents modified STG when the equivalent sates (i.e., $S1 \equiv S3$ and $S2 \equiv S4$) are merged.



State Encoding

- Once the state minimized STG is obtained after merging equivalent states, we focus on the problem of state assignment, where the primary objective is the reduction of the cost of the implementation.

- As discussed, cost of implementation depends on state encoding.

- It is well understood that the number of all possible assignments is very high. If one uses k bits to encode n states, there are ${}^{2^k}P_n$ possible assignments. Therefore, encoding methods used in practical scenarios are heuristics.
- In this lecture we shall discuss the scheme called MUSTANG.

State Encoding

•Broadly speaking, MUSTANG (and most of the other similar techniques) works by trying to identify pairs of states that should receive adjacent codes.

•Two codes are adjacent if they only differ in one bit, like 01 and 01.

•However, it is clear that providing adjacent codes to all pairs of states would require hot one coding which would involve extremely large number of bits for encoding.

•So those states pairs which have transitions leading to one state on same input are assigned adjacent codes. This leads to lower implementation cost.

State Encoding

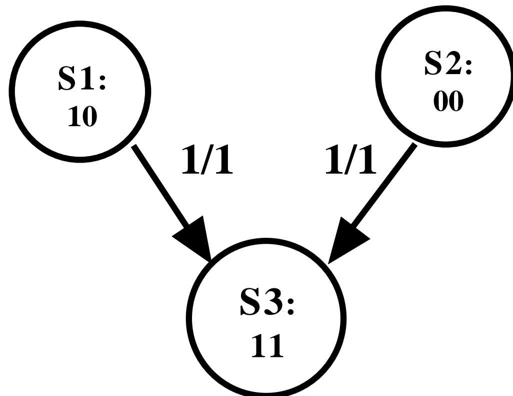
There are two states *S1* and *S2* which lead to the same state *S3* on the same input 1. So it is better to provide adjacent codes to *S1* and *S2*.

The Boolean function representing the next state bits ($s'_0 s'_1$) and the output (o) in terms of input (x) and present state bits ($s_0 s_1$) are as follows; the dotted part represents the product terms of the part of the STG not shown in Figure.

$$s'_0 : f_{S_0}(x, s_0, s_1) = \ldots\ldots. + x s_0 \overline{s_1} + x \overline{s_0}\,\overline{s_1} + \ldots..$$

$$s'_1 : f_{s_1}(x, s_0, s_1) = \ldots\ldots. + x s_0 \overline{s_1} + x \overline{s_0}\,\overline{s_1} + \ldots..$$

$$O : f_O(x, s_0, s_1) = \ldots\ldots. + x s_0 \overline{s_1} + x \overline{s_0}\,\overline{s_1} + \ldots..$$



State Encoding

The equations can be simplified (by low level minimization) as given below.

$$s'_0 : f_{S_0}(x, s_0, s_1) = ... + x\overline{s_1} + ...$$

$$s'_1 : f_{s_1}(x, s_0, s_1) = ... + x\overline{s_1} + ...$$

$$O : f_O(x, s_0, s_1) = ... + x\overline{s_1} + ...$$

It may be noted that the reduction is possible only because of assigning adjacent codes to *S1* and *S2*.

State Encoding

MUSTANG works on this principle by building an attraction graph.

Whenever two states, *S1* and *S2* have a common fanout state, the weight of the edge $\langle S1, S2 \rangle$ of the attraction graph is increased.

Once the graph of the attractions is found, we try to assign adjacent codes to pairs of states that have strong (i.e., higher weights) attractions. It is true that always it may not be possible to assign adjacent codes to all the state pairs. In that case, we start with the state pairs with the edge that has the highest weight and assign adjacent codes.

Following that we move to the edge with the next highest weight and assign adjacent codes, if possible. We repeat, till all the states are assigned a code.

State Encoding

To obtain the weights we build two matrices:

- the first with one row for each present state and one column for each next state (called *St*) and
- the second with one row for each present state and one column for each output (called *Ot*).

- For the STG of the example, we obtain the following matrices, where the superscripts *p* and *n* stand for present state and next state, respectively.

$$St = \begin{array}{cccc} & S1^n & S3^n & S4^n \\ S1^p & 1 & 1 & 0 \\ S3^p & 1 & 0 & 1 \\ S4^p & 1 & 0 & 1 \end{array} \qquad Ot = \begin{array}{cc} & z \\ S1^p & 1 \\ S3^p & 1 \\ S4^p & 0 \end{array}$$

State Encoding

A 1 in $S1^p$ row and $S1^n$ column indicates that there is one self loop from S1 to itself.

Similarly, a 1 in $S1^p$ row and z column indicates that there is an arc going out of $S1$ that asserts that output z should be 1.

In general, the entries of the matrix are non-negative integers that give the number of arcs connecting two states or the number of arcs going out of a state and asserting a given output.

Let St_i be the ith row of St and Zt_i the ith row of Zt. Let also k be number of encoding bits. Then the attraction between states Si and Sj is given by:

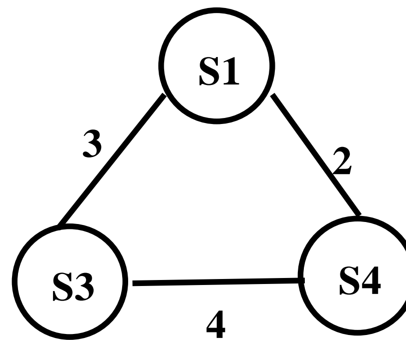
$$k.St_i.St_j^T + Zt_i.Zt_j^T$$

where the operations are on integers and T means transpose.

State Encoding

In our example, assuming *k=2* (as there are three states we need two bits for encoding) the attraction of states S3 and S4 is computed as:

$$W(S3, S4) = 2.[101].\begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} + [1].[0] = 4$$

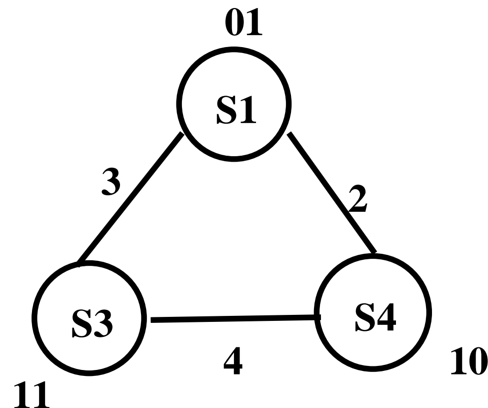


State Encoding

Based on the weight of the attraction graph, we are now ready to assign codes to states.

We select first the state for which the sum of the weights of the incident edges is maximum; in our case, state S3 has the maximum weight of 7. We arbitrarily assign a code 11 to S3 and adjacent codes to the states connected by the heaviest edges; 10 is assigned to S4 and 01 is assigned to S1.

In our example, this completes the process.



Question and Answer

Question:

If there is a STG with n states then we require $\lceil \log_2 n \rceil$ bits to encode the states. After state reduction using "state equivalence" let there be n-e states, where $\lceil \log_2 n \rceil = \lceil \log_2 (n-e) \rceil$. In such a case we do not gain advantage in number of bits to encode the states i.e., there is no reduction in the number of flip-flops required to implement the STG. What is the advantage of "state equivalence" in such a condition?

Answer:

It is true that under the case given above, we do not save on flip-flops. However, as we have reduced number of states, we have fewer conflicts in assigning adjacent codes to the states. More the number of states with adjacent codes, lower are the number of literals in the Boolean expression implementing the next state bits and the output bits.

Design Verification and Test of Digital VLSI Circuits NPTEL Video Course

Module-III

Lecture-VI

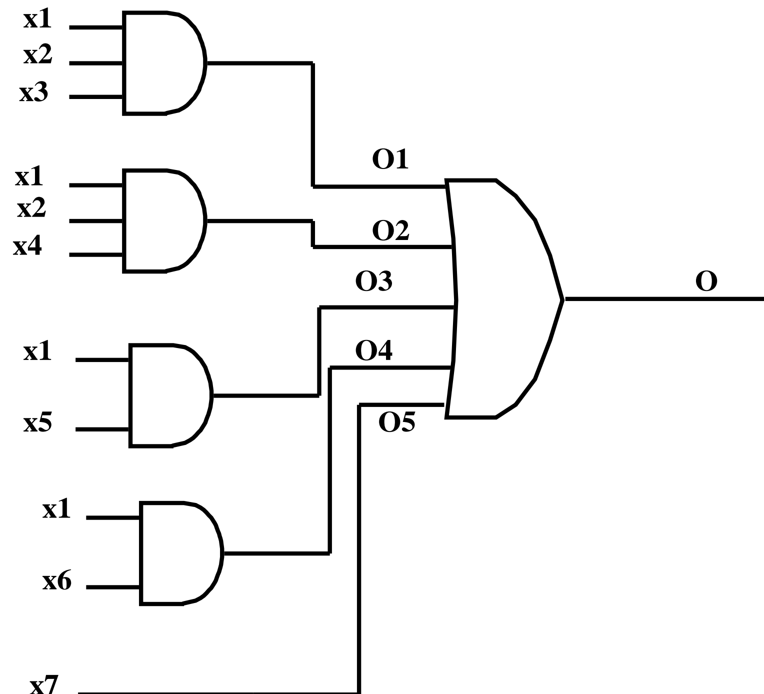
Multilevel Implementation

Introduction

•In all the lectures of this module, we discussed digital synthesis from the context of two-level implementation. Algorithms for optimization in two-level implementation are quite matured (in terms of area), be it exact algorithms like Quine-mccluskey or heuristics like Espresso. However, the major problem in two-level implementation is the huge number of fanins and fanouts of the gates. Let us consider the SOP of the equation:

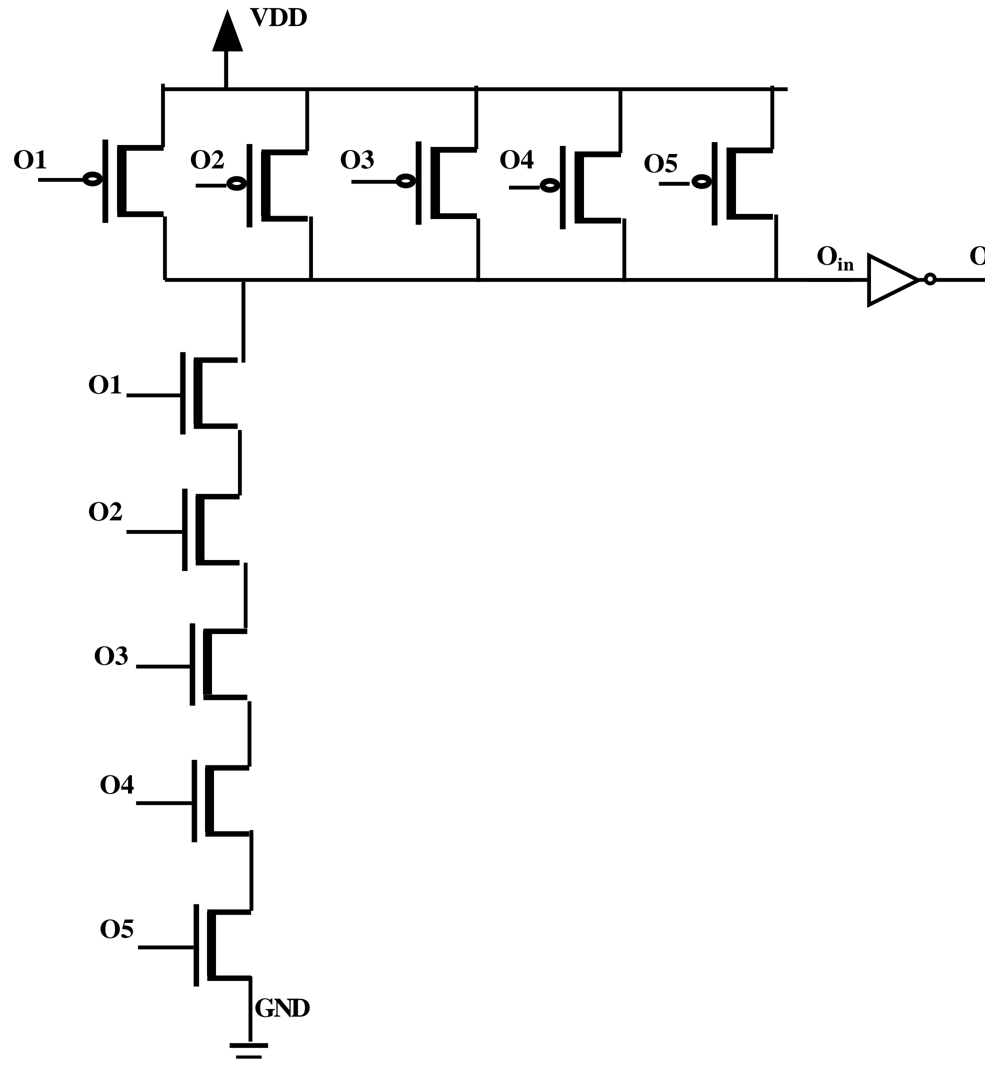
$$f(x1,x2,x3,x4,x5,x6,x7)=x1x2x3+x1x2x4+x1x5+x1x6+x7$$

It may be noted that the SOP is already minimized.



Introduction

It may be noted that the OR gate has five fanins. The CMOS transistor level realization of the gate is shown in next Figure.



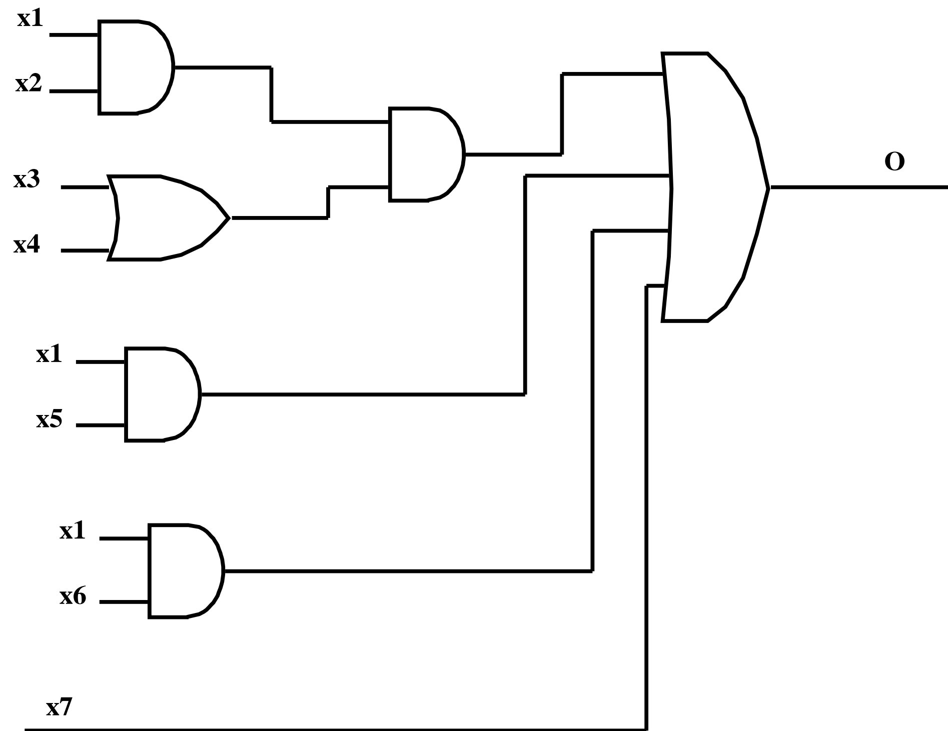
Introduction

It may be noted that there are five NMOS transistors in series for pull down i.e., draining the voltage from O_{in} .

More the number of transistors in series in pull down (pull up) more time the gate will require to go from 1 to 0 (0 to 1). In case of AND gate, PMOS transistors are in series for the pull up.

Therefore, in VLSI implementation, gates having more than four fanins are slow and are generally not used. So to cater to this problem, SOP forms are factorized into multilevel implementation. For example, the SOP can be factorized as $x1x2(x3+x4)+x1x5+x1x6+x7$ and implemented as shown in next Figure.

Introduction



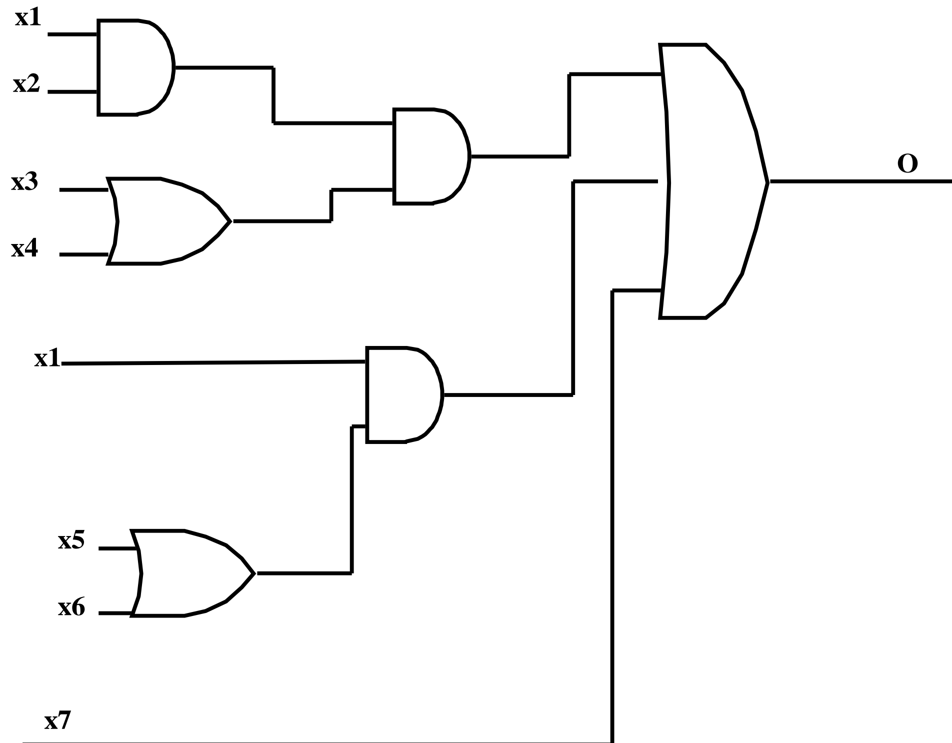
Multilevel implementation of $x1x2(x3+x4)+x1x5+x1x6+x7$

Introduction

- It may be noted that none of the gates have fanins more than four.
- Among all the gates of the circuit the OR gate has the maximum fanin of four, making it the slowest gate of the circuit (assuming same driving capabilities of the transitions used in the gates).
- If for a certain case of meeting the timeline, delay of the OR gate may need to be reduced (by decreasing the fanins). While doing so we need to be careful that we do not increase delay for some other gate which may lead to timing violations.

- The factorized expression $x1x2(x3+x4)+x1x5+x1x6+x7$ can be further factored as $x1x2(x3+x4)+x1(x5+x6)+x7$ leading to circuit shown in next Figure.

Introduction



Multilevel implementation of $x1x2(x3+x4)+x1(x5+x6)+x7$

Introduction

- It may be noted that all gates have fanins of three; also, the number of gates remain same.
- It may be noted that the factorized form $x1x2(x3+x4)+x1(x5+x6)+x7$, is minimal for the SOP $x1x2x3+x1x2x4+x1x5+x1x6+x7$.
- So given a minimal SOP form we need to determine the minimal factorized from, which realizes an efficient multilevel implementation.

Factoring an SOP

An alternative representation (to SOP form) of a logic function, which is closer to the physical multi-level implementation, is the factored form. It is the generalization of SOP form allowing nested parenthesis. For example, each of the following is a factored form:

$$x1$$

$$x1'$$

$$x1x2'x3'$$

$$x1x2' + x3'x4$$

$$(x1 + x2')(x3 + x1' + x4x5) + x6$$

where $x1, x1', x2...$ are called literals of the factored form. Factored forms are generally derived from (minimized) SOP forms. The SOP expression $x1x3x5 + x1x4x5 + x2x3x5 + x2x4x5 + x5'$ can be factorized as $x5(x1 + x2)(x3 + x4) + x5'$. Broadly speaking, a factored form is a SOP of SOP…. , of arbitrary depth.

Factoring an SOP

Definition 1 A factored form is defined recursively by the following rules:

1. A product is either a single literal or a product of factored forms.
2. A sum is either a single literal or a sum of factored forms
3. A factored form is either a product or a sum.

For example, each of the following is a factored form,

$$x1$$

$$x1'$$

$$x1x2'x3$$

$$x1x2 + x3'x4$$

$$(x1 + x2)(x3 + x1' + x4x5) + x6$$

where the first two are literals, the third is a product, the fourth is a sum, and the last one is a sum of products of sums of ....

Factoring an SOP

According to Definition 1, the following are not factored forms

$(x1+x2)'$

$\overline{x1x2}$

because they complement internally, which is not allowed by the definition.

Also it might be noted that like two level SOPs, factored forms are in general not unique, as illustrated by the following two equivalent factored forms.

$x1x2+x3(x1+x2)$

$x2x3+x1(x2+x3)$

Definition 2. An algebraic expression $f=C_1+C_2+...C_n$ is one in which no cube contains another, that is $C_i \subsetneq C_j, i \neq j$. An expression that is not algebraic is called Boolean.

For example, expression $x1+x2x3x4$ is algebraic, and expression $x1+x1x3x4$ is non-algebraic because $\{x1\} \subsetneq \{x2,x3,x4\}$ and $\{x1\} \subsetneq \{x1,x3,x4\}$.

Factoring an SOP

Definition 3. A factored form F is said to be algebraic if the SOP expression obtained by multiplying F out directly (i.e., without using $x1x1'=0$ and $x1x1=x1$ and single-cube containment) is algebraic. F is a Boolean factored form if it is not algebraic.

For example, each of the following is an algebraic factored form.

$$x1+x2x3$$

$$(x1+x2)(x3+x4x5)+x6$$

and each of the following is a Boolean factored form.

$$x1+x1x2x3$$

$$(x1+x2)(x1'+x3+x4x5)+x6$$

As disused in the introduction section, there are many equivalent factored forms of a given SOP expression. Also, the difference in number of literals of these equivalent factored forms can be significant. This point is illustrated using an example in Question and Answers section.

Factoring an SOP

For a given SOP expression we may have different factored forms having different number of literals.

Therefore, algorithms need to be designed which can generate factored forms having minimal number of literals and that would lead to multilevel circuit implementation that is efficient in terns of area.

In others words, we need schemes for obtaining "factored forms which are maximally factored" for efficient multilevel circuit implementation.

Definition 4. A factored form is maximally factored, if
For every sum of products, there are no two syntactically equivalent factors in the products.

For every product of sums, there are no two syntactically equivalent factors in the sums.

Factoring an SOP

For example, the following factored forms are not maximally factored

$$x1x2 + x1x3$$

$$(x1 + x2)(x1 + x3)$$

because they contain trivial syntactically equivalent factors $x1$ in their products and sums, respectively. Hence, they can be further factored to

$$x1(x2 + x3)$$

$$x1 + x2x3$$

The first transformation is obvious. The second is obtained as follows

$$(x1 + x2)(x1 + x3)$$

$$x1x1 + x2x1 + x1x3 + x2x3$$

$$x1 + x1(x2 + x3) + x2x3$$

$$x1(1 + x2 + x3) + x2x3$$

$$x1 + x2x3$$

Division for factoring

We start this subsection with some definitions.

Definition 5. The product of two cubes A and B is a cube defined by

$$AB = \begin{cases} \varnothing \ \mathit{if} \ \exists x (x \in A \cup B \text{ and } x' A \cup B) \\ \mathrm{A} \cup \mathrm{B}, \text{ else} \end{cases}$$

Definition 6. AB is an algebraic product if A and B have disjoint variable sets, otherwise AB is a Boolean product.

For example, $(x1+x2)(x3+x4)$ is an algebraic product because $A=(x1+x2), B=(x3+x4)$ and they have disjoint variable sets i.e., $\{x1+x2\} \cup \{x3+x4\} = \varnothing$. $(x1+x2)(x2+x4)$ is an example of Boolean product i.e., $\{x1+x2\} \cup \{x2+x4\} = x2$.

Division for factoring

Definition 7. Division of SOP is an operation where, given two SOP expressions F and D, it generates SOP expressions Q and R such that $F = DQ + R$.

If DQ is an algebraic product, the operation is an algebraic division; otherwise if DQ is a Boolean product and the operation is called a Boolean division. If $R = \varnothing$ is then D is a factor; otherwise, D is a divisor.

Given a SOP form of Boolean expression F, to convert it into maximally factored from (for efficient multilevel circuit implementation) we need the following steps

1. Find a good candidate divisor D (there are exact and heuristics algorithms), such that $F = DQ + R$, where R has as few cubes as possible
2. Perform the division F/D to generate Q and R. This step is simple and explained with examples as follows

Division for factoring

Let $F = x1x4 + x1x2x3 + x2x3x4$ and $D = x1 + x2x3$; the processing of determining D will be discussed latter in this lecture. For each cube of D, d_i we look for cubes of F, f_j such that f_j has all the literals of d_i.

In this example, for $d_1 = x1$ we get $f_1 = x1x4 = d_1x4$ and $f_2 = x1x2x3 = d_1x2x3$; we write $V^{d1} = \{x4, x2x3\}$. For $d_2 = x2x3$ we get $f_2 = x1x2x3 = x1d_2$ and $f_3 = x2x3x4 = d_2x3$; we write $V^{d2} = \{x1, x4\}$. It may be noted that $x4$ multiplies both d_1 and d_2 because $V^{d1} \cap V^{d2} = \{x4\}$ Hence, we can factor F using $x4$ and write $F = (x1 + x2x3)x4 + x1x2x3$. So, we get $R = x1x2x3$.

Division for factoring

Let $F = x1x3 + x1x4 + x2x3 + x2x4 + x5$ and $D = x1 + x2$. For $d_1 = x1$ we get $V^{d1} = \{x3, x4\}$. For $d_2 = x2$ we get $V^{d2} = \{x3, x4\}$. It may be noted that $x3$ and $x4$ multiplies both d_1 and d_2 because $V^{d1} \cap V^{d2} = \{x3, x4\}$ Hence, we can factor F using $x3$ and $x4$ and write $F = (x1 + x2)(x3 + x4) + x5$. So, we get $R = x5$.

From the above discussion we many understand that once a good divisor is determined division procedure is straight forward. However, if a good divisor is not used than both Q and R can be factorized again (i.e., Q and R may not have minimum cubes). In such a case both Q and R are factorized again and the process is repeated till no more factorization is possible. A typical generic recursive factoring algorithm is given below.

Division for factoring

FACTOR(*F*, DIVISOR, DIVIDE) **BEGIN**

if (*F* cannot be factored) return (F)

D = DIVISOR(F) // selected based on heuristics or exact algorithms [1,2,3]

(*Q*, *R*) = F/D // Division as explained by two examples above

return (FACTOR(*Q*) FACTOR(*D*) + FACTOR(*R*))

END

Given an expression *F* in SOP form, Procedure DIVISOR (F) used to find a candidate divisor, *D*, which, when substituted into *F*, can simplify the expression. Then, the quotient *Q* is found by dividing *D* into *F* using procedure *DIVIDE(F,D)*. Now, the function can be represented as a partial factored form $F = QD + R$ where *R* is the remainder. The algorithm then proceed to factor *Q*, *D*, and *R* separately using the same method.

Division for factoring

Let us consider the example given below.

$$F = x1x2x3 + x1x2x4 + x1x5 + x1x6 + x7$$

$$D = x3 + x4$$

$$Q = x1x2$$

$$R = x1x5 + x1x6 + x7$$

$$F = x1x2(x3 + x4) + x1x5 + x1x6 + x7$$

In this case D is considered to be $x3 + x4$ and after the division $Q = x1x2$ and $R = x1x5 + x1x6 + x7$. We may note that Q cannot be factorized but R can be further factorized as $= x1(x5 + x6) + x7$. So the maximally factored form is $F = x1x2(x3 + x4) + x1(x5 + x6) + x7$. So when we take $D = x3 + x4$, we reach the maximum factored form in two steps having 8 literals.

Division for factoring

Now if we take the same F but consider $D = x1$, then $Q = x2x3 + x2x4 + x5 + x6$ and $R = x7$ (shown below) .

$$F = x1x2x3 + x1x2x4 + x1x5 + x1x6 + x7$$
$$D = x1$$
$$Q = x2x3 + x2x4 + x5 + x6$$
$$R = x7$$
$$F = (x2x3 + x2x4 + x5 + x6)x1 + x7$$

Here Q can be further factorized but R cannot be. So the maximally factored form (after factorizing $x2x3 + x2x4 + x5 + x6$) is $F = x1(x2(x3 + x4) + x5 + x6) + x7$. So when we take $D = x1$ we reach the maximum factored form in two steps having 7 literals.

Division for factoring

Let us consider the example now with $D = x2$.

$$F = x1x2x3 + x1x2x4 + x1x5 + x1x6 + x7$$
$$D = x2$$
$$Q = x1x3 + x1x4$$
$$R = x1x5 + x1x6 + x7$$

In this case after the division $Q = x1x3 + x1x4$ and $R = x1x5 + x1x6 + x7$. We may note that Q can be factorized as $x1(x3 + x4)$ and R can be further factorized as $= x1(x5 + x6) + x7$. So the maximally factored form is $F = x1x2(x3 + x4) + x1(x5 + x6) + x7$. So when we take $D = x2$, we reach the maximum factored form in two steps having 8 literals.

Division for factoring

Let us consider another example given below.

$$F = x1x3x5 + x1x4x5 + x2x3x5 + x2x4x5 + x3x6 + x4x6$$
$$D = x3 + x4$$
$$Q = x1x5 + x2x5 + x6$$
$$R = \varnothing$$
$$F = (x3 + x4)(x1x5 + x2x5 + x6) + \varnothing$$

In this case D is considered to be $x3 + x4$ and after the division $Q = x1x5 + x2x5 + x6$ and $R = \varnothing$.

We may note that Q can be further factorized as $x5(x1 + x2) + x6$.

So the maximally factored form is $F = (x3 + x4)(x5(x1 + x2) + x6)$. So when we take $D = x3 + x4$, we reach the maximum factored form in two steps (basically one and a half step as $R = \varnothing$) having 6 literals.

Division for factoring

Now if we take the same F but consider $D = x1 + x2$, then $Q = x3x5 + x4x5$ and $R = x3x6 + x4x6$ (shown below) .

$$F = x1x3x5 + x1x4x5 + x2x3x5 + x2x4x5 + x3x6 + x4x6$$
$$D = x1 + x2$$
$$Q = x3x5 + x4x5$$
$$R = x3x6 + x4x6$$
$$F = (x1 + x2)(x3x5 + x4x5) + x3x6 + x4x6$$

Here Q and R can be further factorized leading to $F = x5(x1 + x2)(x3 + x4) + x5(x3 + x4)$. So when we take $D = x1 + x2$ we reach the maximum factored form in two steps having 8 literals.

Division for factoring

•From the discussion we may note that computation efficiency (i.e., number of steps) and area of the multi-level circuit (i.e., number of literals) depend on the divisor *D* being selected.

•There are several exact algorithms and heuristics for selection of the divisor.

•It may be noted that due to the extremely large number of all possible divisors exact algorithms are highly time complex. To cater to this issue heuristic algorithms are proposed.

•For example, very simple heuristic is the one were the divisor is a single cube and involves literals that occur in most of the cubes of *F*.

Question and Answer

Question: Give an example to show that Boolean division provides better solution that Algebraic division.

Question and Answer

Answer:

For example, given the SOP expression

$$x1x2x7 + x1x3x7 + x1x4x6 + x1x5x6 + x1x6x7 + x2x4 + x3x5 + x2x5 + x3x4$$

the following are three equivalent factored forms, where the first has 12 literals, the second has 11 literals and the third has 8 literals.

$$(x2 + x3)(x4 + x5) + ((x4 + x5 + x7)x6 + (x2 + x3)x7)x1$$

$$(x2 + x3)(x4 + x5 + x1x7) + (x4 + x5 + x7)x1x6$$

$$(x1x6 + x2 + x3)(x1x7 + x4 + x5)$$

The first two of these equivalent factored forms are algebraic, whereas the third one is Boolean.

Thank You