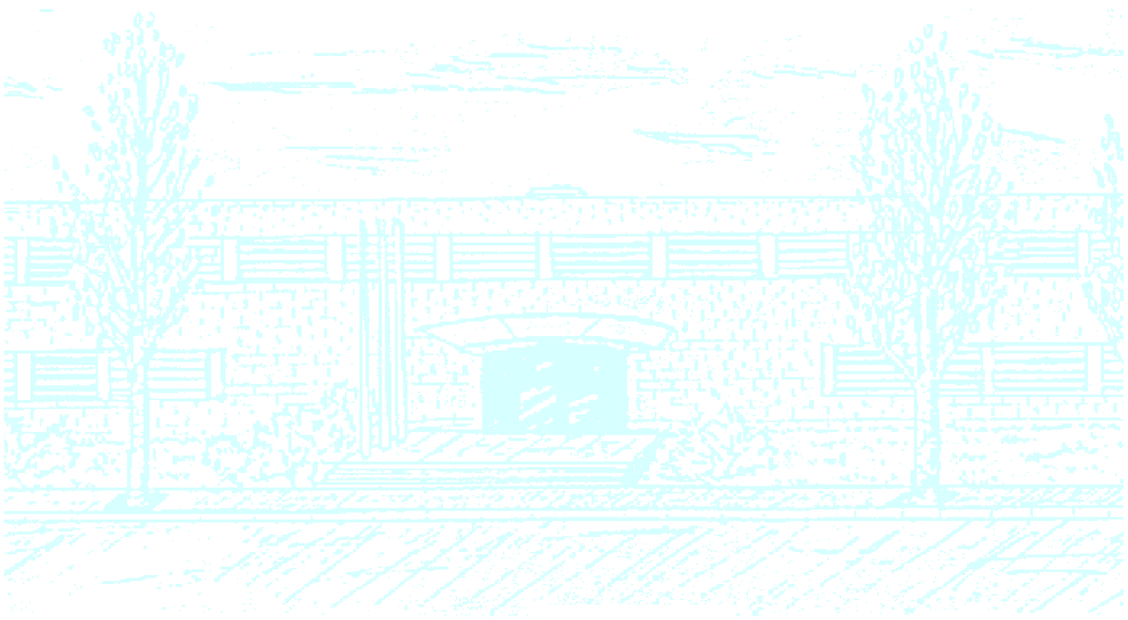
Title: Library-Free Technology Mapping for VLSI Circuits with Regular Layouts

Author: Alex Alvarez Ruiz

Advisor: Jordi Cortadella Fortuny, Sachin Sapatnekar

Department: Llenguatges I Sistemes Informàtics

Academic year: 2013-2014



Universitat Politècnica de Catalunya
Facultat de Matemàtiques i Estadística

Master's Degree Thesis

Library-Free Technology Mapping for VLSI Circuits with Regular Layouts

Alex Alvarez Ruiz

Advisor: Jordi Cortadella Fortuny, Sachin Sapatnekar

Departament de Llenguatges i Sistemes Informàtics

To my family, advisors and everyone that has made this thesis possible

If you find that you're spending almost all your time on theory, start turning some attention to practical things; it will improve your theories. If you find that you're spending almost all your time on practice, start turning some attention to theoretical things; it will improve your practice.

Donald Knuth

Abstract

Keywords: algorithms, technology mapping, graphs, optimization

MSC2000: 200068W35, 68R10

Technology mapping is the task to transform a technology independent logic network into a mapped network using gates from a library, optimizing some objective function such as total area, delay or power consumption. As stated, the problem is completely intractable. Therefore, different techniques are applied to solve the problem, such as using different simplified representations.

The usual approach consists of using a fixed library, typically partially handmade. Designing such libraries is a costly process, specially because of the lack of good automatic techniques to perform it. This is the main motivation for this work and the goal is to move from fixed-library technology mapping to library-free technology mapping.

The idea of this is simple: design a new and specific library for each network instead of using a fixed library for all of them and then perform the technology mapping. Although simple, this idea presents several difficulties to overcome. As technology mapping is a hard problem, libraries have to be small enough to be tractable and the models used to generate them need to be both efficient and accurate enough. Finally, networks may have sizes of about billions nowadays, which means that the algorithms for all the process have to be very efficient.

This thesis presents different algorithms and techniques to move to library-free technology mapping for area and for delay optimization. The structure of this document is as follows. The first chapter contains an introduction to the problem and to the physical design of gates using transistors. Chapter 2 is focused on area optimization while chapter 3 covers delay optimization. Last chapter presents the results obtained using those methods and some conclusions.

Contents

Chapter 1. Introduction 1
1. The design of gates using transistors 1

Chapter 2. Area Optimization 7
1. Cut Extraction 8
2. Factoring Algorithm 10
3. Boolean Matching 11
4. Binary Decision Diagram 13
5. Area Estimation 14
6. Minimization of the Eulerian Decomposition 16

Chapter 3. Delay Optimization 19
1. Delay Estimation 19
2. Elmore Delay Model 20
3. TILOS algorithm 23

Chapter 4. Results and Conclusions 27
1. Area Results 27
2. Delay Results 28
3. Conclusions and future work 28

References 29

Chapter 1
Introduction

This thesis is centered in the study of library-free technology mapping. The standard problem of technology mapping is the following: given a library of gates L and a Boolean network G, represent G as a network constructed using just the gates in L. This new network is called mapped network. But the technology mapping does not consist only in finding such a representation, but in minimizing some objective function (typically area, delay or power consumption).

Technology mapping usually uses a fixed library, which in most cases is either totally or partially handmade designed. The goal of this thesis is overcoming those two properties of libraries: instead of using fixed libraries, library-free technology mapping techniques are presented. The term library-free means that the library is generated specially for the circuit to be designed. Therefore, that obliges to develop automatic methods for constructing those libraries instead of designing them by hand.

The next sections include a brief introduction to the design of gates at physical level, as this is a key point to understand the rest of the work. The next two chapters will be focused on the minimization of area (Chapter 2) and delay (Chapter 3). Chapter 4 presents the results along with conclusions on future work.

1. The design of gates using transistors

In the project, circuits are going to be present at every moment, starting at the level of gates. Therefore, it is important to have a basic idea on how gates are designed and as the main element is the MOS transistor, a brief introduction to their types and how they work is presented here.

A MOSFET (Metal-Oxide-Semiconductor Field-Effect Transistor) is the most common transistor in circuits nowadays. A cross section of a general MOSFET is shown in Figure 1. W.l.o.g. MOS transistors can be seen as switches.

Depending on how they work, there are two different types of MOS transistors: pMOS and nMOS. The difference is based on the type of doping. Pure semiconductors are called intrinsic. Doping refers to the process of intentionally adding impurities to the semiconductors to modify their electrical properties, obtaining extrinsic semiconductors. n-type semiconductors are those doped to have a larger electron concentration than hole concentration while p-type semiconductors have a larger hole concentration than electron concentration.

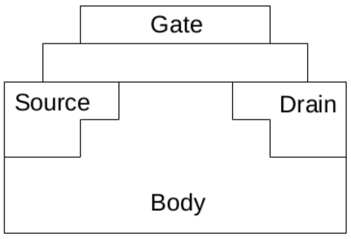


Fig. 1. Cross section of a MOS transistor

Figure 2 shows a cross section of a pMOS and an nMOS transistor. In the case of an nMOS, both the source and the drain are n+ regions (the "+" sign is just a notation to indicate that the semiconductor is highly doped) and the body is a p region. Therefore, by applying a positive voltage to the gate, the electrons of the body are attracted, thus forming a channel from source to drain. pMOS transistors work just the opposite, as shown in the figure.

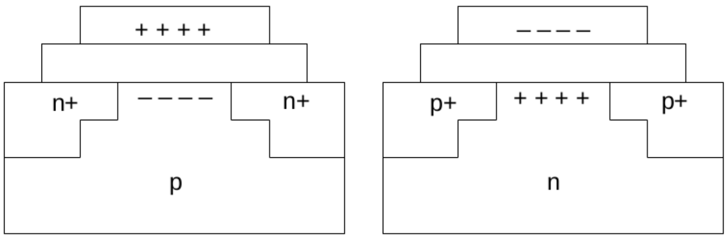


Fig. 2. Cross section of an nMOS transistor (left) and a pMOS transistor (right)

Thus, both types can be seen as complementary switches: when an nMOS transistor receives positive voltage at the gate, the channel is formed and then it acts as a closed switch, otherwise it acts as an open switch. pMOS work the other way around.

CMOS (Complementary MOS) is a technology for constructing integrated circuits, based on combining transistors of types p and n. For a simple example on how to design a NOT gate see Figure 3.

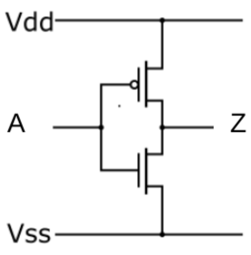


Fig. 3. NOT logic gate

The transistor in the upper part is of type p and the other of type n. Vdd is the power supply voltage (logic 1) and Vss stands for the logic 0 or ground (also noted as GND). Now let us analyze what happens with the value of Z depending on the value of A. If $A = 1$, the n transistor acts as a closed switch, but the p transistor acts as an open one. Then, the logic 0 propagates to Z. If $A = 0$, then the state of the transistors is reversed, thus propagating a 1.

This scheme can be used to produce any logic function f in which every input is negated and there are no negations affecting anything apart from single variables. De Morgan's laws are useful to put a function in that way.

THEOREM 1 (De Morgan's laws). *If x and y are boolean variables , then*

$$\begin{aligned}(xy)' &= x' + y' \\ (x+y)' &= x'y'\end{aligned}$$

where the $'$ operator just denotes the negation of the Boolean formula affected.

Then, consider $f(x_1, \ldots, x_n)$ such that x_i appears always negated in the function for every i. f can be recursively constructed using pMOS transistors, as Figure 4 shows:

- $f(x_1, \ldots, x_n) = x_i'$. Then a single p transistor with input x_i encodes that.
- $f(x_1, \ldots, x_n) = f_1(x_1, \ldots, x_n) f_2(x_1, \ldots, x_n)$. Then connect in series the circuits for f_1 and f_2.
- $f(x_1, \ldots, x_n) = f_1(x_1, \ldots, x_n) + f_2(x_1, \ldots, x_n)$. Then connect in parallel the circuits for f_1 and f_2.

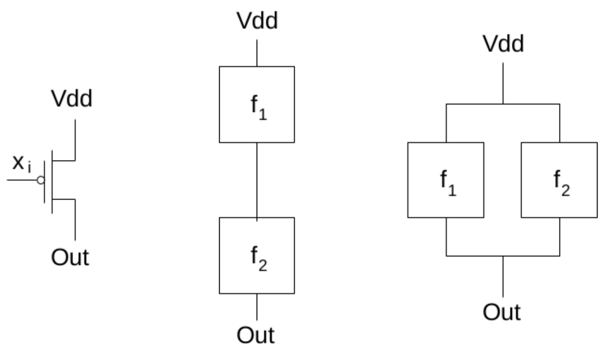


FIG. 4. Construction of a function f. At the left, the single variable function, in the center the product and at the right, the addition.

That construction leads to a circuit which connects the logic 1 to the output when the function evaluates to one and acts as an open switch otherwise.

Considering f' and by applying De Morgan's laws, a function with no negations is obtained and therefore, with every variable appearing with positive sign. With a similar procedure a circuit for f' using nMOS transistors can be constructed: the rules for the addition and the product are equivalent, and the case of a single variable (which now is not negated) is designed with a single n transistor.

The circuit for f is called the pull-up network and the circuit for f' is called the pull-down network and one is the dual of the other. Notice that as they are complementary, for each possible combination of values for the inputs exactly one of them acts as an open switch and the other as a closed switch. The circuit obtained by connecting the outputs of f and f' realizes the function f. In Figure 5 there are a couple of examples constructing a NAND and a NOR function of two inputs each.

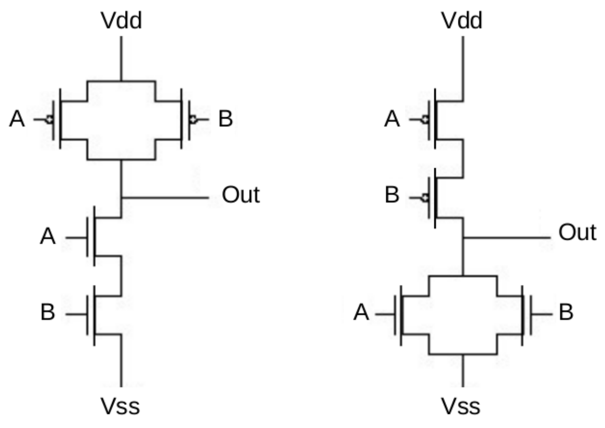


FIG. 5. An example of CMOS logic functions. The circuit at the left is the NAND of A and B and the one at the right is the NOR of A and B.

The next natural step is to see how those CMOS circuits are constructed. But before that, let us introduce the graph associated to the pull-up and the pull-down of a CMOS gate. The construction is the same for both pull-up and pull-down and is very simple: every node represents a junction of wires and the transistors are the edges connecting them. Therefore the graph is a series-parallel graph. Figure 6 shows the graphs for the pull-ups of the examples in Figure 5.

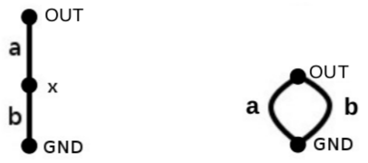


FIG. 6. The graphs of the pull-downs of the examples in Figure 5. NAND at the left, NOR at the right.

With the series-parallel graph, it is easy to present the final form of the constructed gate and argue why the graph is important. To construct the gate the transistors need to be placed in some order, connected as the graph shows. Figure 7 shows possible implementations for the graphs in Figure 6, that can be seen as paths or cycles inside the graphs. For a complete design of a gate, see Figure 8.

But as the reader may notice, such a path will not always exists. Therefore, sometimes is needed to add diffusion breaks to isolate a couple of chains of transistors. Figure 9 shows an example of such a case.

Therefore, the properties of the graph have a direct impact in the area of the physical implementation of the circuit. This will be covered in the section about area estimation in the next chapter.

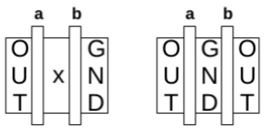


FIG. 7. Implementations of the graphs in Figure 6. NAND at the left, NOR at the right.

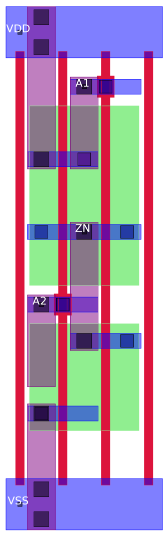


FIG. 8. NAND gate with two inputs A1 and A2. This design also shows the connections between the different regions.

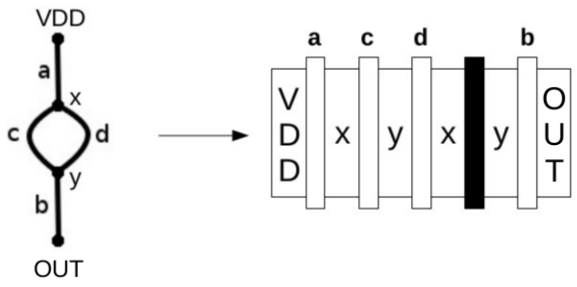


FIG. 9. A possible implementation of the graph at the left. As no single transistor chain can realize that graph, a diffusion break is needed to isolate the two transistor chains.

This chapter has presented the process of technology mapping and how gates are physically designed. To finish this introduction, let us end with the design of a real circuit in terms of gates, i.e. a mapped network, as shown in Figure 10.

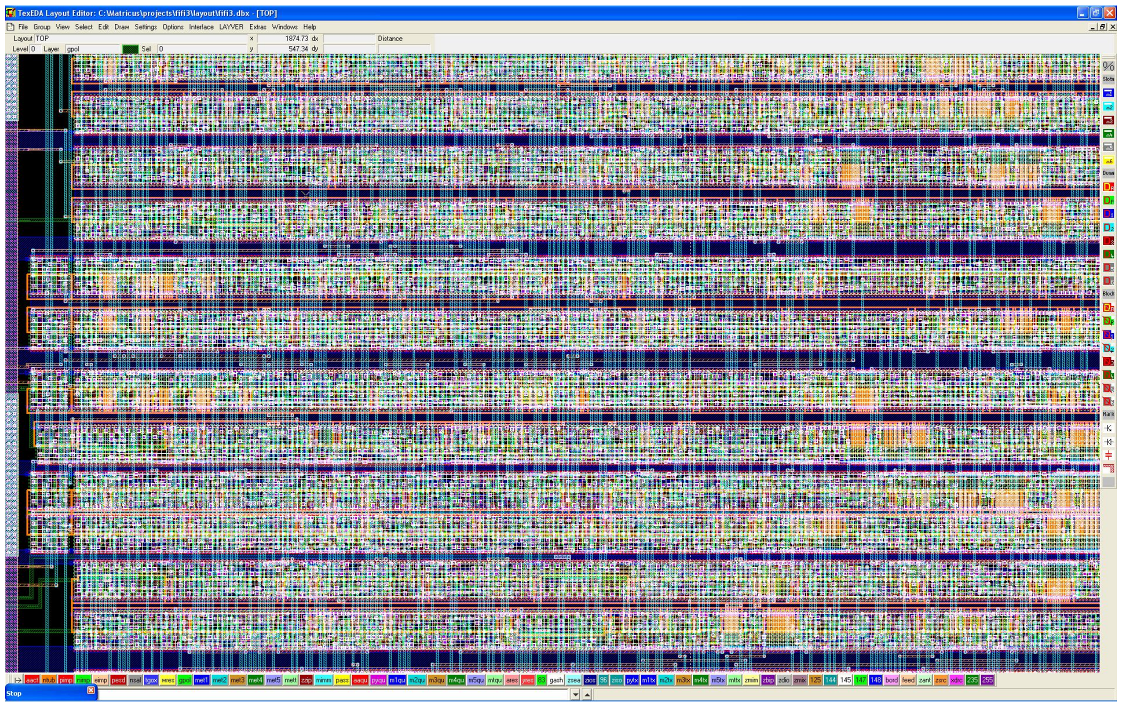

TexEDA Layout Editor: C:\Matricus\projects\fifi3\layout\fifi3.dbx - [TOP]
File Group View Select Edit Draw Settings Options Interface LAYVER Extras Windows Help
Layout TOP
Level 0 Layer gpol Sel 0
x 1874.73 dx
y 547.34 dy
Distance
Stop

Fig. 10. An example of circuit with standard cells.
From `http://www.laytools.com/images/StandardCells.jpg`.

Chapter 2
Area Optimization

This chapter is focused on area optimization. The goal is to know whether using library-free technology mapping optimizes the area and if it does, how much gain is obtained.

In order to test that, the following method is proposed: given a circuit to be designed, construct an specific library for it, and then perform the technology mapping. Figure 1 shows the main steps involved in the process.

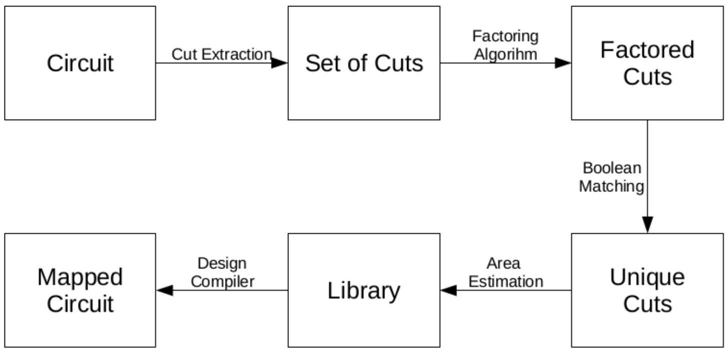


FIG. 1. The main steps for the Area Optimization process.

The idea behind the whole process is to create a new library formed by gates that appear in the circuit to be designed. This process of extracting those candidates to gates, called cuts, is called Cut Extraction.

Once it has been performed, a multiset of cuts is obtained. Notice the word multiset, as a cut can (and in general may) appear more than once in the network. Therefore, to avoid redundancy in the library that is going to be created, is important to remove equivalent cuts under an equivalence relation called NPN. To do that, the cuts are first factored to reduce their size and therefore, become more tractable. Then, they are classified into equivalence classes, thus obtaining different cuts and removing those that are equivalent.

The last step for the creation of a library is estimating the area of the remaining cuts to construct a new library with them. Once the library is finished, the technology mapping is performed. This chapter covers these steps in several sections.

1. Cut Extraction

The first step for building the library is extracting cuts from the circuit up to a certain size. Let us then define formally what a cut is. But to be able to define a cut, it is needed to previously introduce And Inverter Graphs (AIGs).

DEFINITION 1. *Let (V, A), $A \subseteq V \times V$ be a directed acyclic graph (DAG) where each node either has no incoming arcs or exactly two incoming arcs. Let inv be a function from A to $\{0, 1\}$. Then $G = (V, A, inv)$ is called an AIG.*

The nodes with no incoming edges are called inputs, except for one special node called Zero. Every other node is just an AND node that computes the AND of its incoming arcs. If $\text{inv}(a) = 1$, then it means that the arc a is inverted. Figure 2 shows an example of AIG.

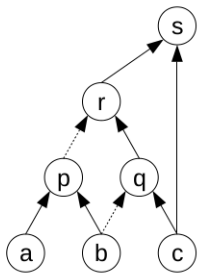


FIG. 2. Example of AIG. The Zero node is omitted. Nodes a, b and c are inputs and the rest are AND nodes. Solid lines represent non-inverted arcs and dashed lines represent inverted arcs.

DEFINITION 2. *Let G be an AIG and u a node of G. A cut C of u is a set of nodes such that every path from an input node to u includes a node in C and such that the set is minimal (there are no redundant vertices). The size of the cut is just $|C|$.*

AIGs were first presented in [**8**]. It is easy to see that they can represent any function. Besides, it is also efficient finding cuts in an AIG.

Using the size of a cut, is natural to talk about k-cuts, referring to cuts of size k or less. In particular, there is always the trivial cut of size 1 formed by just each node itself. To make it clearer, consider the AIG in Figure 2. The following table presents all its 3-cuts:

Node	**3-cuts**
a	{{a}}
b	{{b}}
c	{{c}}
p	{{p}, {a, b}}
q	{{q}, {b, c}}
r	{{r}, {p, q}, {p, b, c}, {q, a, b}, {a, b, c}}
s	{{s}, {r, c}, {p, q, c}, {p, b, c}, {a, b, c}}

Notice that for this AIG, every cut of size larger than 3 is redundant and therefore, those are all the cuts.

Therefore, cuts represent parts of the function that is going to be designed and seem to be a good choice for extracting new gates for the library generation. The method proposed consists in finding all the cuts up to a certain size and computing the function they represent to obtain a set of cuts that are candidates for the gates in the new library. Notice that once a cut is found, computing its function is as easy as traversing in topological order[1] from the nodes of the cut to the node. From now on, the term "cut" is going to be used both to represent the set and the function that it represents.

Let A, B be two sets of k-cuts. Define

$$A *_k B = \{a \cup b : a \in A, b \in B, |a \cup b| \leq k\} \tag{1}$$

Now define,

$$f : V(G) \to 2^{2^V} \tag{2}$$

such that for a node it returns its k-cuts.

Definition 3. *Let G be an AIG and u a node in G and v, w its two inputs (if they exist). Define*

$$g(u) = \begin{cases} \{\{u\}\} & \textit{if } u \textit{ is an input node} \\ (g(v) *_k g(w)) \cup \{\{u\}\} & \textit{otherwise} \end{cases} \tag{3}$$

Theorem 2. *Let G be an AIG and u a node in G. Then, $f(u) \subseteq g(u)$.*

Proof. By induction on the position in a topological order of G. If u is an input, equality holds and the theorem is obviously true.

Otherwise, consider a k-cut C of u. Then every path from an input to u passes through a node in the cut, so it passes through one of the two inputs of u, call them u_1, u_2. Thus, C induces cuts C_1, C_2 for the inputs, that by induction hypothesis are included in $g(u_1), g(u_2)$ respectively. Thus, the inclusion for u holds. □

Notice that the equality is not true. This procedure may produce sets that are actually not cuts because they are not minimal (they may have redundant nodes). In fact, considering again the AIG in Figure 2, one can check that for $k = 4$ then $f = g$ restricted to the set of nodes $\{a, b, c, p, q, r\}$. But $\{c\} \in g(c)$ and $\{q, a, b\} \in g(r)$, which means that $\{q, a, b, c\} \in g(s)$. As noticed before, every cut of size 4 is redundant (here q can be removed) and therefore $g(s) \neq f(s)$.

Even though, that definition gives a simple yet very efficient algorithm once that some speed-ups are applied. The idea is to just traverse the AIG in topological order and keep computing the cuts for every node: first by using the previous recurrence and then erasing the sets that are not valid. This algorithm is based in [**9**].

To make the algorithm efficient, implementation details include the use of hashing to quickly discard lots of candidates to cuts, similar to the well-known Bloom filters[**10**]. The implementation used for the project is the one included in ABC[**1**], with some modifications.

Therefore, cuts have been defined and this section has presented an algorithm to obtain them, using AIGs as a data structure to represent the network.

[1]Recall that a topological sorting of a directed acyclic graph is a permutation of the nodes such that arcs always go forward, i.e. there are no arcs (i, j) such that $\sigma(j) < \sigma(i)$, where σ is the permutation.

2. Factoring Algorithm

For a given cut there are many equivalent forms to represent it as a formula. There are two important reasons to try to find a form that is as small as possible:

- As said in the beginning of the chapter, many cuts may be repeated, appearing even with different forms. Besides, there are many cuts that are also equivalent to others (this notion of equivalence is presented in the following section). Therefore, is interesting to obtain a simple form for every cut to be able to work as efficiently as possible with them.
- The introduction describes how gates are designed, and the area of the gate heavily depends on the form chosen for the function that it represents. In the section about area estimation, an algorithm to minimize the area for a given form is presented, but it is important that the initial form is as small as possible to lead to the smallest possible area.

Therefore, a factoring algorithm is used to find the smallest possible form for the cut. That form is called factored form and can be easily represented by a tree in a unique way, up to permutations of the subtrees of some node.

The idea of this representation is simple: a variable is just a tree with one node, otherwise the node encodes one operation and its children are factored forms representing the operands affected by the node (two or more). Notice that each internal node has as children all of its operands and then going down over the tree means reaching a leaf or the opposite operator. In Figure 3 an example of the tree associated to a factored form is presented. From now on, the term "factored form" is going to be used for both the tree and the formula, as they are equivalent representations.

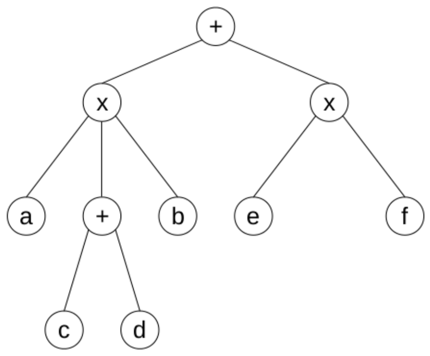


FIG. 3. A tree representation of the factored form $a(c + d)b + ef$.

Although there are several approaches for factoring, almost all of them rely on finding divisors. This process of finding divisors can be reduced to a very general problem in combinatorial optimization: Rectangle Covering. For the process, ABC's implementation[**1**] has been used. To find more about the reduction and the problem itself, check [**6**] or [**7**].

3. Boolean Matching

Once all the cuts have been extracted from the original circuit, there are lots of equivalent cuts in general. Technology mapping is a time-consuming process, so it is important to remove those repeated cuts and only then construct the library to do the mapping.

Definition 4. *Given two cuts c_1, c_2, then $c_1 \sim c_2$ if c_2 can be obtained by negating and/or permuting the inputs of c_1 and/or negating the output. This is called NPN equivalence.*

By having every cut represented with a factored form, the assumption that two equivalent forms have the same number of inputs can be almost[2] always made.

Theorem 3. *$\sim$ is an equivalence relation.*

Proof. Reflexivity is obvious: just take as a permutation the identity and do not invert the inputs nor the output.

For a given cut c_1 of size k, write it as $c_1(a_1, \ldots, a_k)$. If $c_1 \sim c_2$, then

$$c_2(a_1, \ldots, a_k) = i_{out}(c_1(i_1(a_{\sigma(1)}), \ldots, i_k(a_{\sigma(k)}))) \tag{4}$$

for some permutation σ and for $k+1$ functions $i_1, \ldots, i_k, i_{out}$ such that they are the identity or an inverter. Therefore, just inverting everything the required symmetry property is achieved.

Using the same idea, by composing it can be obtained that the relation is also transitive. □

Therefore, the goal is to classify the found cuts under NPN equivalence and just take one as a representative of each class. As the technology mapper can connect the outputs to any input and can place inverters in between, this equivalence is fully justified.

This has been the most time-consuming step of the whole area optimization process. Therefore, the key point was to be able to perform this classification as fast as possible and, on top of that, to design an algorithm as parallelizable as possible.

There are mainly two approaches to do this process known as Boolean matching:

- Comparison-based: Based on taking two functions and check whether they are equivalent directly.
- Canonization-based: Based on taking only one function to compute a canonical form. Once a canonical form has been found for each function, it is straightforward to quickly compare two canonized functions.

Comparing two circuits is faster than canonizing them to just compare them after that: the reason is that there are lots of easy-to-check properties that are able to quickly discard the equivalence, which are called filters.

But having as many cuts as the cut extraction process obtains and taking into account that those cuts are small, for the Boolean matching is faster to just find a canonical form for every cut and then just compare the canonical forms.

[2]Not always because the Rectangle Covering Problem is an NP-Complete problem and hence the optimal solution is not always found by the algorithms.

For the canonizing algorithm, a unique string will be associated to different equivalence classes, thus once they are calculated the classes can be found fast enough by just comparing strings.

Therefore, the important step in this matching process is the canonizing step. There are plenty of articles talking about this, check for example [**11**]. The following definition presents the canonical form chosen to represent functions.

DEFINITION 5. *Given a cut c of size k, consider its truth table as a binary string of length 2^k. The canonical form of c is defined as the lexicographically smaller truth table under NPN modifications.*

Notice that this is well-defined as a canonical form: if two cuts are equivalent, then both have the same canonical form because of the properties of the relation.

The initial algorithm consist on just checking the $k!2^k$ possibilities without taking into account the output negation because once a permutation and negation pattern for the inputs have been found, the output phase that produces a smaller truth table can be found in constant time.

This algorithm fulfills one of the two purposes: it is really parallelizable (each function is an independent computation) but it fails in terms of time. In order to make it more efficient, symmetries of the cuts are exploited.

DEFINITION 6. *For a given cut c, inputs i and j are symmetric if transposing them does not alter the output, independently of the permutation applied to the other inputs.*

It is well-known that symmetries are very usual, so the idea is to exploit them to avoid checking as many permutations of the inputs. For that, the following theorem is required.

THEOREM 4. *The symmetry relation is an equivalence relation.*

PROOF. Reflexivity and symmetry are obvious. It is also easy to see the transitivity. If i and j are symmetric and so they are j and k, w.l.o.g. we can assume that their relative order is (i, j, k). Swapping i and j does not affect, and then the order (j, i, k) maintains the function invariant. Now swap i and k, with the same result again to obtain (j, k, i). Finally, swap j and k to obtain (k, j, i). Along the process the function is unchanged and we have swapped i and k. □

Therefore, the algorithm can be restricted to just the permutations that are ordered inside each symmetry class and that prunes a lot the number of permutations that it checks. Empirically, there is a 10-factor speed-up, which makes the algorithm fast enough to perform the matching. The only remaining thing is to find a way to compute whether two inputs are symmetric.

THEOREM 5. *Given a function f and inputs i and j, they are symmetric if and only if $f_{i'j} = f_{ij'}$, where f_x denotes the cofactor of f with respect to x.*

PROOF. The function f can b write as $f = i'j'f_{i'j'} + i'jf_{i'j} + ij'f_{ij'} + ijf_{ij}$. Therefore, it is now straightforward to see that f remains the same after a swap of i and j if and only if $f_{i'j} = f_{ij'}$. □

That property can be computed efficiently using Binary Decision Diagrams, which are presented in the next section. Hence, the Boolean Matching step can be performed efficiently enough.

4. Binary Decision Diagram

A Binary Decision Diagram (BDD) is a data structure used to represent Boolean functions. A BDD is a rooted directed graph in which there are two different kind of nodes: the terminal vertices and the non-terminal vertices.

Terminal vertices can be either the 0 function or the 1 function. Each non-terminal vertex u is associated to a variable of the function, which will be refered as $\text{index}(u) \in \{1, \dots, n\}$, where n is the number of variables in the function.

Besides, each non-terminal node u is connected to two nodes called $\text{low}(u)$ and $\text{high}(u)$. Thus, every node u represents a function f_u, which is either 1 or 0 for a terminal node and for a non-terminal node such that $\text{index}(u) = i$ it satisfies the following property:

$$f_u(x_1, \dots, x_n) = x_i' f_{\text{low}(u)}(x_1, \dots, \hat{x_i}, \dots, x_n) + x_i f_{\text{high}(u)}(x_1, \dots, \hat{x_i}, \dots, x_n) \tag{5}$$

Imposing also that the variables in a tree traversal appear in order, i.e. $\text{index}(u) < \text{index}(\text{high}(u))$ and $\text{index}(u) < \text{index}(\text{low}(u))$ whenever it makes sense, then the data structure is called an Ordered Binary Decision Diagram (OBDD).

THEOREM 6. *Every OBDD is a directed acyclic graph.*

The theorem is just a consequence of defining an order among the nodes. Figure 4 contains an example of OBDD.

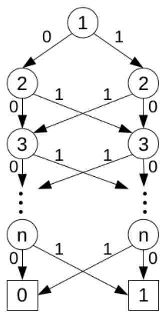


FIG. 4. An OBDD representing the parity function on n variables.

But even with a OBDD issues as in Figure 5 can happen. Notice that both nodes with index 3 are identical. In those cases, the OBDD can be reduced by deleting one of the two repeated nodes. After that, notice that the node with index 2 with both edges pointing to the node with index 3 is useless, and then it can be actually removed too.

DEFINITION 7. *Two OBDDs G_1 and G_2 are isomorphic if there exists a bijection*

$$\sigma : V(G_1) \longrightarrow V(G_2) \tag{6}$$

such that for any vertex u with $\sigma(u) = v$, then either both u and v are terminal and they have the same value or both are non-terminal, have the same index and satisfy that $\sigma(low(u)) = low(v)$ and $\sigma(high(u)) = high(v)$.

That presents a natural way to define reduced OBDDs.

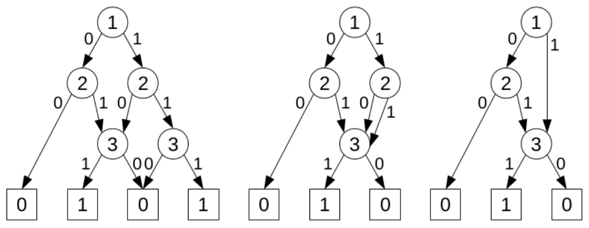


FIG. 5. Successive steps to obtain a ROBDD from the OBDD at the left of the figure.

DEFINITION 8. *An OBDD is reduced if for every vertex u we have low(u) $\neq$ high(v) and it does not contain two different vertices such that the subgraphs rooted at them are isomorphic. In that case it is called a Reduced Ordered Binary Decision Diagram (ROBDD).*

The importance of ROBDDs as data structures for representing Boolean functions is that apart from having a reasonable size in many cases, they serve as a canonical form for the function, as it states the following theorem.

THEOREM 7. *For any Boolean function f, there is a unique ROBDD representing f, up to isomorphism. Besides, any other OBDD representing f contains more vertices.*

For a proof of the theorem, check [**12**]. The general idea is to use induction on the number of inputs.

Using the theorem and the fact that it is easy to check whether two ROBDDs are isomorphic (a simple linear-time check), it is a powerful tool for canonize Boolean functions.

Besides, ROBDDs support several operations. It is straightforward to complement the BDD or to cofactor it. With some more work, it can also be applied any Boolean operation between two ROBDDs. Therefore, with that structure the operations needed for the Cut Matching step can be efficiently performed. For the implementation of BDDs the project uses the CUDD package[**3**].

5. Area Estimation

For a certain gate, its area is usually obtained by designing it and computing the area from the design. That is a valid approach when working for a finite fixed set of cells, but this is not the case here. The set of possible gates is not fixed at all, but it depends on the network to be mapped. Therefore, this classical way is not reasonable and a new method to approximate the area is needed.

As said in the first chapter, the construction of a given factored form depends basically on the chosen transistor chains, thus determining the number of diffusion breaks needed to isolate different transistor chains. Hence, the width of the cell depends on the number of diffusion breaks, i.e., it depends on the number of transistor chains.

DEFINITION 9. *Given a factored form f, its area is defined as the minimum number of transistors chains to cover it plus the number of transistors needed (i.e. the number of inputs of the factored form, counted with multiplicity).*

That definition is just counting the minimum possible number of sections in the cell. Notice that both the pull-up and the pull-down networks have to be covered, and then the area is just the maximum of the two values.

Under the model defined, the area is directly related to the number of Eulerian paths in an Eulerian decomposition of the graph associated to the circuit, because a transistor chain is just a path that does not repeat edges along a CMOS graph of the factored form.

THEOREM 8 (Euler). *For a given undirected graph G, if the graph contains either zero (two) nodes with odd degree, then it contains an Eulerian cycle (path).*

This theorem is well-known, but consider now a more general version that is needed in order to compute the area defined above and it is also known.

THEOREM 9. *For a given undirected graph G with $k > 2$ nodes with odd degree, it can be decomposed into $k/2$ edge-disjoint paths covering every edge.*

PROOF. By induction on k. For $k = 1$ is the previous theorem. Suppose $k > 2$ and let s be a node in G with odd degree. Consider a path starting at s and continue picking arbitrary edges until it reaches a node with odd degree other than s.

Notice that this algorithm is well-defined: every time the path reaches a node with even degree, it has at least one unused edge to continue the path and if it reaches s at some moment, as it has odd degree it must also have an unused edge. Therefore, as the number of edges is finite, this algorithm ends in a node with odd degree.

Let us now remove the edges of this path of G. The number of vertices with odd degree has decreased by two. Therefore, a valid decomposition can be obtained by applying the induction hypothesis. □

This decomposition will be called an Eulerian decomposition of the graph and with a little abuse, each of those paths will be called Eulerian paths.

There are different CMOS graph representations associated to a certain factored form that lead to different Eulerian decompositions. For example, consider the formula $f' = ab(c+d)$. In Figure 6 there are two different graphs associated to this factored form but with different number of odd nodes, i.e. with Eulerian decompositions of different size. Therefore, it is crucial to determine the best such graph representation in order to minimize the number of odd nodes and hence the area of the cell.

Before starting with the algorithm to minimize the number of Eulerian paths in a decomposition, the two different models used to measure the final area after the technology mapping are presented:

- Simple model: The area of the mapping is just the sum of the areas corresponding to each cell.
- Complex model: This model tries to take into account the fact that sometimes it can be done better than in the simple model. In the simple mode the assumption is that consecutive cells have a diffusion break to isolate the last section of the previous cell and the first section of the following one.
  Suppose that some cell ends in a VDD section (in the pull-up) and the next cell also begins with a VDD section. Then there is no reason to not merge both sections into only one.

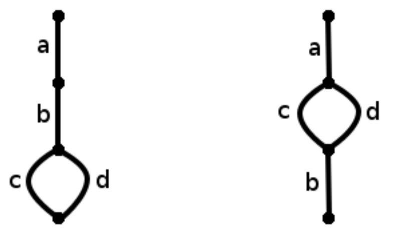


FIG. 6. Two different graph representations for the pull-up of the formula f = ab(c + d). The first graph contains only two odd nodes while the second has four.

To approximate that, the area definition is slightly modified. Thus, if in a pull-up the cell ends and starts with something different than the VDD, the area is increased by 1. If it ends or (exclusive or) starts with the VDD, the area is increased only by 0.5 and in case it starts and ends with VDD the area remains unchanged. The idea is analogous for the pull-down and the GND.

With that, the whole model for the area estimation has been covered. The next step, presented in the following section, is to design an algorithm to minimize that area.

6. Minimization of the Eulerian Decomposition

Given a formula, it is computationally impossible to consider every possible factored form associated to it. Therefore, in order to minimize the area of a cell, the only considered factored form is the one obtained during the cut extraction by applying the factoring algorithm mentioned before.

The goal now is to find a graph associated to the factored form such that it minimizes the number of vertices with odd degree. Being the factored form fixed, the only possibility left is permuting the subtrees of a given node in the tree associated to the factored form. The first result about that follows from the definition of the CMOS graph associated to the tree.

CLAIM 1. *Permuting the subtrees of a sum node does not affect to the degree of the vertices.*

Therefore it is only needed to consider permuting the subtrees of product nodes. That assumption does not make the problem easier: backtracking for every possibility is still impossible as at each product node there are $r!$ possible choices, where r is the number of subtrees.

Consider a subtree of the factored form. For an associated graph of that factored form, there is a natural bijection between a subtree of the factored form and a subgraph of the associated graph with only two nodes connected to the rest of the graph (see Figure 7 for clarity). In particular, the root of the tree maps to the whole graph.

DEFINITION 10. *Given a node u of the tree and a graph representation G, consider the subgraph H associated to u in G and let v_1, v_2 be the two special nodes that are connected to the rest of the graph. We define the parity of u as the pair $(d_H(v_1), d_H(v_2))$, where d_H denotes the parity of the degree in H (0 if even, 1 if odd).*

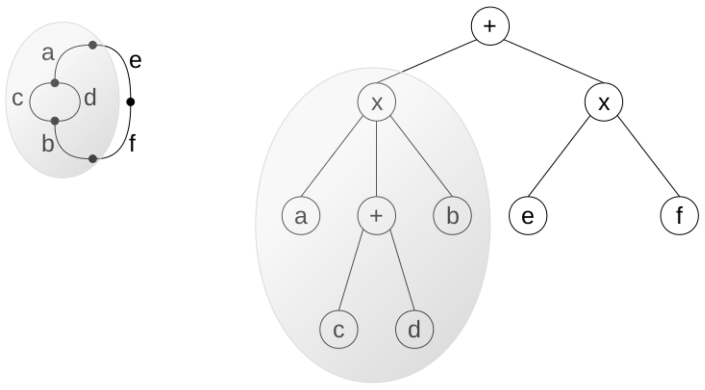


FIG. 7. An example of the mapping described.

For example, in Figure 7 the parity of the node selected is the pair $(1,1)$. Consider now the following function

$$f : V \times \mathbb{Z}_2 \times \mathbb{Z}_2 \longrightarrow \mathbb{Z} \cup \{\infty\}$$

where V is the set of nodes in the tree and given a node and its parity, $f(u, d_H(v_1), d_H(v_2))$ denotes the minimum number of internal odd nodes to represent u with such parity. Then the minimum number of odd degrees that we can achieve is

(7) $$\min_{a,b}\{f(\text{root}, a, b) + a + b\}$$

That function can be calculated using dynamic programming. There are three distinct cases for $f(u, a, b)$:

- u is a leaf: For a leaf, the associated graph is just an edge, so its parity is always $(1,1)$. Therefore $f(u,1,1) = 0$ and $f(u,a,b) = \infty$ for any other pair a, b (where the ∞ is just to put the fact that the configuration is not reachable).
- u is a sum node: In this case the order of the subtrees does not matter, but the parities for them have to be considered. Let $g(i, a', b')$ be the minimum number of odd nodes required to put the first i subtrees of u with parity (a', b'), taking only into account the placed subtrees when computing the parity (or ∞ if not possible). Then $f(u,a,b) = g(r,a,b)$, where r is the number of subtrees of u.
  Besides, g can be calculated in linear time in the number of children of u, because

(8) $$g(i, a', b') = \min_{x,y}\{g(i-1, x, y) + f(u(i), a' - x, b' - y)\}$$

  where $u(i)$ denotes the i-th children of u.
- u is a product node: This case is slightly more complicated because the order does matter. Consider the function h defined as

(9) $$h : 2^{[r]} \times \mathbb{Z}_2 \longrightarrow \mathbb{Z} \cup \{\infty\}$$

  such that $h(S, b')$ is the minimum number of odd nodes to place the set of children S with parity degree (a, b'). Therefore $f(u,a,b) = h([r], b)$ and again, this can be computed using dynamic programming:

$$h(S,b') = \begin{cases} f(i,a,b') & \text{if } S=\{i\} \\ \min\{\min\limits_{x,y} h(S \setminus \{i\},x) + f(u(i),y,b') + x + y : i \in S\} & \text{otherwise} \end{cases} \tag{10}$$

This recurrence contains some abuses of notation, so let us comment it. The idea is to keep removing from the total set one child each time. Fixing the last inserted son, the rest of the set can be placed with two finishing parities (denoted here by x) and the same holds for the selected son i that is being placed (denoted by y its finishing parity). Notice that if x and y are different, then one extra odd node is created, what is reflected under the $x+y$ part with some abuse of notation using that they are in $\mathbb{Z}_2$.

Theorem 10. *The f function is correctly calculated for a node u with r children in time $\Theta(r)$ if u is a sum node and time $\Theta(2^r r)$ if u is a product node.*

Proof. The fact is obvious for sum nodes. For a product node, the recurrence is correct because once is known the set of placed children, the order does not matter anymore as far as it is known the parity degree so far.

There are $\Theta(2^r)$ states and each one runs in $\Theta(r)$, so the total time is $\Theta(2^r r)$. □

With this approach, a problem initially very difficult to solve becomes instantly solvable because for factored cuts the number of children in a node is always very small, as checked empirically.

Chapter 3
Delay Optimization

After developing techniques to optimize the area, the next natural step is to move that idea to the delay optimization. The approach is going to be different to obtain smaller delays because using large gates that appear in the circuit may lead to a smaller area, but those large gates tend to have longer delays. Therefore, new techniques are presented for the delay optimization.

The key idea of the method proposed is called gate sizing. The next section explains how the delay works more in detail, so the purpose here is just to present a brief idea on why modifying the size of the transistors of a gate leads to shorter delays.

In fact, that is not always true. By making a transistor larger, the gate is going to be faster in general, but at the same time the previous gates (the ones that are connected to the inputs) are going to increase its delays, because they have to load a larger capacitance at their outputs.

Thus, the method proposed is divided in three steps:

(1) Initial library. An initial library to work with is needed. For that, let us take an existing library and recalculate the timings with the approximation model presented in the next sections, in order to work at every step with the same delay model.
(2) Run Design Compiler[**2**] (a software of Synopsys to perform technology mapping) to obtain an initial mapped circuit to start the optimization.
(3) Apply the optimization using the TILOS algorithm to size transistors across critical paths.

The following two sections explains the first step and the section after them covers the third step.

1. Delay Estimation

In order to optimize the delay of the circuit, an efficient method to approximate the time is needed. Let us first present what determines the delay to be able to present in the next section a simple yet precise model called Elmore Delay Model[**4**]. With this model the delay for a gate can be determined and using those times, the delay of the entire circuit can be approximated.

The delay of a gate is mainly originated from the time for loading (in case of value one) or discharging (in case of value zero) the capacitance connected to the output, but this delay depends on several aspects. The model proposed considers it as a function depending on the

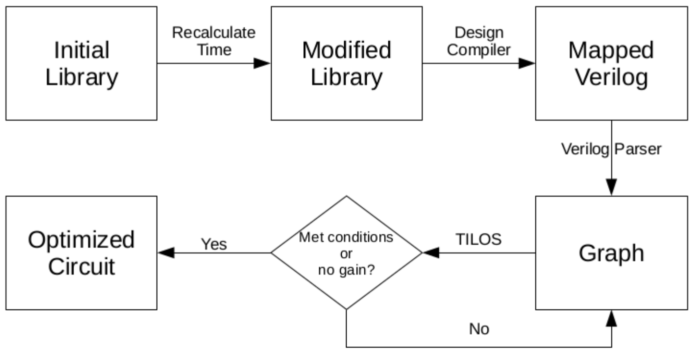


FIG. 1. The flow diagram for the delay optimization method proposed

structure of the gate (its graph), the properties of its transistors (both p and n), the output capacitance, and the slew.

It is very intuitive that the properties of the transistor affect the delay, because a transistor can be modeled mainly as a resistor and a capacitor. Therefore, the larger the capacitance, the longer it takes to discharge it and thus, the longer the delay is. A similar reasoning can be used to the resistor: more resistance implies more time to discharge the capacitor.

Another important factor is the output capacitance. Consider some gate whose output is connected to a single inverter and now the same gate connected to five inverters. In the second case, the capacitance to load is larger and that also makes the delay longer.

The slew parameter is more subtle. Ideally, the signal in every input switches from zero to one and from one to zero instantaneously. But in a real circuit this is completely wrong and the plot of the signal is continuous indeed. This is what the slew parameter tries to represent: the slope of that change. More specifically, the slew is a value that tells the time to pass from some percentage of the change of phase (typically around 30%) to another percentage (typically around 70%).

Notice that under this model the delay caused by the wires is ignored, which are considered ideal, thus assuming that their resistance and capacitance is negligible. The Elmore Delay Model is presented in the following section.

2. Elmore Delay Model

The Elmore Delay model is a simple method to approximate the delay of a circuit. The strong points are that it is efficient and usually accurate. Besides, even when the results are not accurate, reducing the Elmore Delay will usually lead to a reduction the real delay of the circuit.

DEFINITION 11. *A resistor-capacitor network (RC network) is an electric circuit composed of resistors and capacitors.*

The Elmore Delay Model is able to calculate the delay for a RC network. Let us see how it is derived. First of all, consider a simple RC circuit consisting of a single resistor and a single capacitor, as can be seen in Figure 2. Recall Kirchhoff's voltage law and Ohm's law:

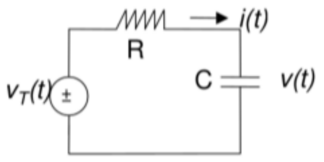


FIG. 2. A simple RC circuit

THEOREM 11 (Kirchhoff's voltage law). *The directed sum of the electrical potential differences (voltage) around any closed network is zero.*

THEOREM 12 (Ohm's law). *The potential difference through a conductor between two points is equal to the product of the resistance and the current, i.e.*

$$V = IR \tag{11}$$

Let $v_T(t)$ be the input waveform and $v(t)$ the state variable, where t is the time. Therefore, due to Kirchhoff's voltage law and to Ohm's law

$$Ri(t) + v(t) = v_T(t) \tag{12}$$

Using that $i(t) = \frac{d(Cv(t))}{dt} = Cv'(t)$ and substituting in the previous equation

$$RCv'(t) + v(t) = v_T(t) \tag{13}$$

This is an easy differential equation. Consider $v_T(t) = v_0u(t)$, where Figure 3 shows the function $u(t)$, corresponding to a step input.

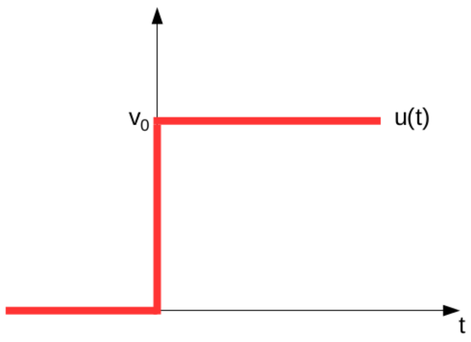


FIG. 3. The function $u(t)$

Therefore, the general solution is $v(t) = Ke^{-t/RC} + v_0u(t)$ and using $v(0) = 0$, the unique solution is

$$v(t) = (1 - e^{-t/RC})v_0u(t) \tag{14}$$

Having that, a couple of properties are assumed to continue with the derivation of the model:

- The step input is of value one, i.e. $v_0 = 1$.
- The function $v(t)$ has monotonic responses to changes in the input waveform.

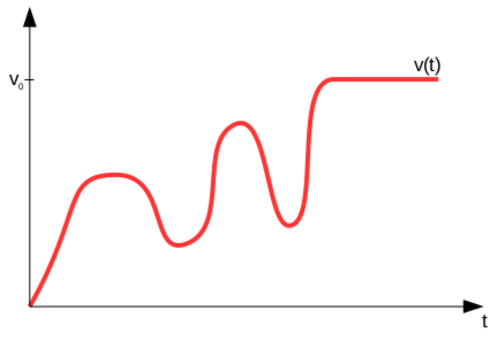


FIG. 4. An example of non-monotonic response to a change in the input waveform.

The goal is computing the time. Let T_{50} the value such that $v(T_{50}) = 0.5$, then computing the time is equivalent to calculate the median

$$\int_0^{T_{50}} v'(t)dt \tag{15}$$

The idea of the model is to compute the mean instead of the median as an approximation, as this leads to a simple result:

$$\int_0^{\infty} tv'(t)dt = \int_0^{\infty} \frac{1}{RC}u(t)e^{-t/RC} = \int_0^{\infty} \frac{1}{RC}e^{-t/RC} = -e^{-t/RC}(RC+t)\Big]_0^{\infty} = RC \tag{16}$$

This is the time approximated for the simple circuit under the Elmore Delay Model. Using this basic scheme, models for more generic circuits can be derived. Typically it is used on circuits that can be represented as a tree, but for this optimization the graphs that are going to be considered are those associated with the gates, that are series-parallel.

The main idea to produce an algorithm for this family of circuits is that a series-parallel circuit can be described recursively: any series-parallel circuit is either a series configuration composed of parallel blocks or a parallel configuration composed of series blocks. Using that, the problem is reduced to calculate the delay for a parallel configuration and for a series configuration independently, assuming that every needed information is known for each subblock.

Series Configuration. Figure 5 shows a general series configuration. Assume that the driving source is at A. Let R_i be the resistance of the i-th series-parallel block and let C_i denote the capacitance between blocks i and $i+1$ and the capacitance in the block i. Let T_i be the delay of block i, taking into account only C_i (i.e. as a parallel subblock of the whole series configuration).

THEOREM 13. *The delay T for the whole series configuration can be calculated as follows:*

$$\hat{T}_n = T_n \tag{17}$$

$$\hat{T}_i = \hat{T}_{i+1} + T_i + R_i \sum_{k=i+1}^{n} C_k \quad \textit{for} \quad i < n \tag{18}$$

$$T = \hat{T}_1 \tag{19}$$

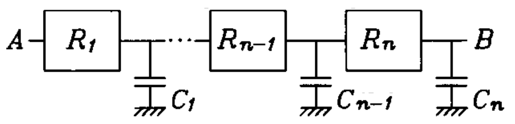


FIG. 5. A general series configuration with n segments

Parallel Configuration. Figure 6 shows a general parallel configuration. Again assume that the delay is associated to the propagation from A to B. Let R_i be the equivalent resistance of the i-th block and T_i the delay of that branch considering that it is the only branch loading C.

This time the delay is harder to calculate than in the series case. Let $\hat{T}_i$ represent the delay from A to B considering only the first i branches, i.e. removing the remaining branches, and let $\hat{R}_i$ be the equivalent resistance of the first i branches in parallel. Then,

$$\hat{R}_1 = R_1 \tag{20}$$

$$\hat{R}_i = \frac{R_i \hat{R}_{i-1}}{R_i + \hat{R}_{i-1}} \quad \text{for} \quad i > 1 \tag{21}$$

THEOREM 14. *The delay T for the whole parallel configuration can be calculated as follows:*

$$\hat{T}_1 = T_1 \tag{22}$$

$$\hat{T}_i = \frac{\hat{R}_{i-1} T_i + R_i \hat{T}_{i-1} - R_i \hat{R}_{i-1} C}{R_i + \hat{R}_{i-1}} \quad \textit{for} \quad i > 1 \tag{23}$$

$$T = \hat{T}_n \tag{24}$$

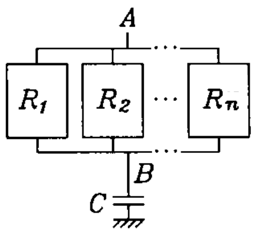


FIG. 6. A general parallel configuration with n branches

The proof of the first theorem is obtained using the Elmore Delay Model for RC-Trees, because a series configuration is just a particular case of a tree. The parallel case is more complicated. Both proofs can be found at [**13**].

With that, the model to compute the delay for any general series-parallel graph is finished, thus obtaining a method to approximate the delay for any gate.

3. TILOS algorithm

Once the initial mapped circuit is obtained with Design Compiler, the method proposed tries to reduce the delay by doing gate sizing, i.e., by adding transistors to certain gates.

For a certain transistor of size x, its associated resistance is proportional to $1/x$, because the larger the transistor, the more distributed is the resistance and hence, the smaller it is. For the capacitance, assuming that the wire capacitance is negligible, it is proportional to the sizes of the transistors whose drain or source is connected to the node. Therefore, for a path with N transistors of sizes $x_1, \ldots, x_N$, the general form of a path delay can be obtained applying the Elmore Delay Model previously defined, obtaining the following expression for some constants a_{ij}, b_i (most of them zero):

$$\sum_{i,j=1}^{N} a_{ij}\frac{x_i}{x_j} + \sum_{i=1}^{N} \frac{b_i}{x_i} \tag{25}$$

Considering (25) as a function of the x_i, it is a convex function. But we are not interested in the delay of a certain path, but in the whole circuit. The important part comes from the point that the delay through a single combinational circuit is just the maximum over every path in the block. Thankfully, convexity is preserved under sums and maxima. Therefore, we want to minimize a convex function subject to some upper bound constraint on another convex function (in this case, limiting the area, which is directly related to the sum of the transistor sizes, by some constant factor). This implies that any local minima also globally minimizes the objective function.

Convex programming is a topic in which many techniques are available, check for instance [**14**]. But the function that is being studied is more that just a convex function, as it belong to class of functions called posynomials.

Definition 12. *A posynomial is a function of the form*

$$f(x_1, x_2, \ldots, x_n) = \sum_{k=1}^{K} c_k x_1^{a_{1k}} \cdots x_n^{a_{nk}} \tag{26}$$

where all the coordinates x_i and coefficients c_k are positive real numbers, and the exponents a_{ik} are real numbers.

Therefore, the techniques of Geometric Programming can be used to efficiently solve that kind of optimization problems.

Even though, is interesting to use the algorithm on big circuits, and then a sacrifice in accuracy is necessary to gain efficiency. The algorithm presented may not find the real optimum, but empirical results prove it to be very good in practice.

```
initialization;
while conditions are not met do
    find a critical path $C$;
    max_gain := $-\infty$;
    foreach transistor with size $x$ in $C$ do
        max_gain := max(max_gain, gain($x \to x + 1$))
    end
    if max_gain $< 0$ then
        break;
    else
        increase the size of the transistor achieving the maximum gain;
    end
end
```

Algorithm 1: TILOS algorithm

The idea behind TILOS is very simple. It is a greedy algorithm which tries to reduce the delay at each step. A pseudocode of the algorithm is presented in Algorithm 1. The idea is to keep doing iterations until the objective conditions are met or until the maximum has been achieved and doing more iterations means obtaining worse results. An iteration is just finding a critical path of the circuit (one that achieves the delay of the circuit) and increasing in one unit the size of the transistors of the gate with the most profit per increase. Figure 7 shows an example of an iteration of TILOS. For more details on the algorithm, check [**15**].

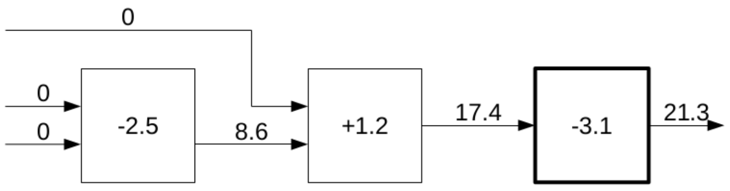


FIG. 7. An example of critical path. The wires have the delay for them and the gates have the increment or decrease of time in case of adding a transistor. In this case, the marked block with a gain of delay of 3.1 would be the one selected by TILOS.

That is the last step in the process of optimizing the delay. With that, the only remaining part is the presentation of the results and some conclusions about both optimization processes (area and delay).

Chapter 4
Results and Conclusions

1. Area Results

This section presents the results that have been obtained applying the techniques for area optimization. To test the proposed method the NanGate Open Cell Library[**16**] has been used as a starting point, extending it with gates obtained from the circuit that is being designed. The test circuits chosen are those of the ISCAS85[**5**], a benchmark that consists of several combinational circuits. To make the results significant, the smaller circuit (c17) is not analyzed as it is too small to check the algorithm.

The results are separated depending on the model used to generate the library, that can be either the simple or the complex model. For technological restrictions, the cuts generated are limited to those having at most 4 series p transistors and at most 4 series n transistors. They are presented in Table 1.

Therefore, the results show that the library-free technology mapping obtains better results when optimizing the area of the final circuit. The average of the improvement in area is 3.37% for the simple model and 3.35% for the complex model, which are good percentages in terms of cost of manufacturing.

The most time-consuming step of the whole optimization is the Boolean matching. This step can last for several hours depending on the circuit and the number of extracted cuts.

Circuit	**Simple Model**	**Complex Model**
c432	6.0%	6.4%
c499	0.4%	1.4%
c1355	0.4%	1.4%
c1908	4.1%	3.9%
c2670	3.4%	3.9%
c3540	8.5%	7.8%
c5315	6.2%	3.5%
c6288	0.0%	0.0%
c7552	1.4%	1.9%

TABLE 1. The area results classified by circuit and model

Circuit	Gain
c17	60.67%
c432	44.50%
c499	28.20%
c1355	31.47%
c1908	31.24%
c3540	40.00%
c5315	38.91%
c6288	26.70%
c7552	61.77%

Table 2. The delay results classified by circuit

2. Delay Results

This section presents the results obtained by applying the techniques for delay optimization. Again, NanGate Open Cell Library is used as a starting point, recalculating its times using the Elmore Delay Model. The benchmarks are again those of the ISCAS85.

As results in Table 2 show, gate sizing obtains better results when optimizing the delay of the circuit by a considerable amount. In fact, the average of this results is a promising 40.38%.

This process is much more efficient than in the area case, running for only several minutes in the worst case of the experiments.

3. Conclusions and future work

The goal of this project was to see whether not using a fixed library for the technology mapping process was exploitable to obtain better results. The results, as expected, show that this is a good way for optimizing the technology mapping process and that future research in the topic can be prolific. There are two main lines to extend this research: one is to move to more realistic models that approximate the area and the delay of a circuit much more precisely (with some trade-off in the efficiency and the effort), the second is to move to more realistic circuits instead of using benchmark circuits. That second approach means working with circuits that are larger and that imply that new algorithms and models are needed to deal with them.

Future work may also be directed to a different kind of delay optimization, using transistor sizing instead of gate sizing. The difference is that in gate sizing the transistors are added to all the gate, i.e. to every input, and in transistor sizing only to the critical transistors. That allows us a larger variety of new possible gates.

References

[1] ABC: A System for Sequential Synthesis and Verification. Berkeley Logic Synthesis and Verification Group.

[2] Design Compiler, from Synopsys. `http://www.synopsys.com/Tools/Implementation/RTLSynthesis/DesignCompiler/Pages/default.aspx`

[3] CUDD: CU Decision Diagram Package. Fabio Somenzi, Colorado University. `http://vlsi.colorado.edu/~fabio/CUDD/`.

[4] W. C. Elmore. The Transient Response of Damped Linear Networks with Particular Regard to Wideband Amplifiers. Journal of Applied Physics, Vol. 19, 1948.

[5] ISCAS85 Combinational Benchmark Circuits. `http://www.cbl.ncsu.edu/benchmarks/`.

[6] R. Rudell. Logic Synthesis for VLSI Designs. PhD thesis, University of California, Berkeley, 1989.

[7] R. Brayton, R. Rudell, A. Sangiovanni-Vincentelli and A. Wang. Multilevel logic synthesis. Notes for lectures at Oxford/Berkeley Summer Engineering Programme, July 1989.

[8] Andreas Kuehlmann and Florian Krohm. Equivalence checking using cuts and heaps. In DAC '97: Proceedings of the 34th annual conference on Design automation, pages 263?268, New York, NY, USA, 1997. ACM Press.

[9] Peichen Pan and Chih-Chang Lin. A new retiming-based technology mapping algorithm for LUT-based FPGAs. In FPGA '98: Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays, pages 35-42, New York, NY, USA, 1998. ACM Press.

[10] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. Commun. ACM, 13(7):422-426, 1970.

[11] Donald Chai and Andreas Kuehlmann. Building a better Boolean matcher and symmetry detector. DATE '06 Proceedings of the conference on Design, automation and test in Europe, pages 1079-1084.

[12] O. Bula, J. Moser, J. Trinko, M. Weismann and F. Woytowich. Gross Delay Defect Evaluation for a CMOS Logic Design System Product. IBM Technical Memorandum, May 1989.

[13] Jean-Paul Caisso, Eduard Cerny and Nicholas C. Rumin. A Recursive Technique for Computing Delays in Series-Parallel MOS Transistor Circuits. IEEE Transactions on Computer-Aided Design, Vol. 10, No. 5, May 1991, pages 589-595.

[14] R. Tyrrell Rockafellar. Convex Analysis. Princeton University Press. 1970.

[15] Alfred E. Dunlop, John P. Fishburn, Dwight D. Hill and Donald D. Shugard. Experiments Using Automatic Physical Design Techniques for Optimizing Circuit Performance. 1990 IEEE.

[16] NanGate Open Cell Library. `http://www.nangate.com/?page_id=22`.