Master's Degree in Computer Engineering
Embedded Systems Track

Politecnico di Torino

EURECOM
Télécom Paris

Implementation of Algorithms for Synthesis of Digital Circuits

Supervisors

Prof. Andrea Calimera

Prof. Ludovic Apvrille

Candidate

Alessandro Tempia Calvino

Industrial Supervisor

Dr. Patrick Vuillod, Synopsys

AY 2019–2020

Abstract

This thesis describes some techniques to be used in EDA synthesis tools to increase the quality of results of digital circuits. We present circuit transformations able to improve the delay by pushing critical signals forward in the logic. The algorithms are applied to netlists of mapped or generic non-sized gates, without electrical optimization. The elaborations we present are graph-based and can be used in conjunction with generic algebraic methods. The first two algorithms rely on the associative and distributive property which are used to restructure a cluster of few gates to reduce the logic levels for critical paths. The last algorithm presents an extension of the Global Flow algorithm. Global Flow is a rewiring algorithm that minimizes a circuit by changing its set of connections to an equivalent one. The new algorithm introduces the possibility of using global flow analysis on generic and mapped gates. Moreover, we present a heuristic to optimize for delay rather than for area. The algorithms have been implemented as part of the EDA tool Fusion Compiler by Synopsys. We describe the results of some experiments showing the potential improvement over the QoR.

Acknowledgements

Firstly, I sincerely express my gratitude to my academic supervisors, prof. Andrea Calimera and prof. Ludovic Apvrille, for their guidance and supervision during my whole internship project.
I sincerely thank my industrial supervisor, Patrick Vuillod for his direction and help throughout the project. A special mention goes to Luca Amaru, who always helped me with his time and precious advice.
I would also like to express my gratitude to all members of the Synopsys' office in Montbonnot-Saint-Martins, in particular to the GDV team.
I thanks my mum Luciana, my grandparents Maria and Bruno, and my family for their amazing support during this academic path. A special thanks go to my friends and colleagues in EURECOM for the astonishing year spent together. In particular, I thanks "Les Grillons", our family in Antibes. I thank all the friends in my home town, in particular, Federico V., the "Casa del Nonno" members, and all the musicians with whom I shared my passion for playing the guitar, in particular Mario.
Moreover, I thanks my friends and colleagues in Torino for a great time spent together. Finally, I would like to express all my gratitude to the academic institutions I am enrolled in, Politecnico di Torino, Télécom-Paris and EURECOM, for the knowledge I acquired.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Electronic Design Automation (EDA) is a category of tools used to accomplish the design of integrated circuits. The exponential increase of requirements and design complexity made the development of automated processes necessary to achieve a better quality of results and productivity. EDA tools focus on the design, simulation, analysis and verification, and the manufacturing preparation phases of an integrated circuit production. EDA tools allow to lower the human interaction and to manage large-scale designs in a reasonable time. EDA must cope with multiple challenges: billions of transistors on a chip (VLSI), new emerging technologies, and reduced time to market. Quality of results (QoR) is a performance indicator of integrated circuits that is used as a quality assessment. Originally, it was mainly based on the speed and area of a chip. Nowadays, with the needs of an evolving industry, it is measured on several additional parameters, like power consumption, routability, and testability.

The electronic digital design phase of the microelectronic production flow[1] is mainly divided into three tasks:

- **Modeling**: hardware description (HDL)
- **Synthesis and Optimization**: create and optimize the logic structure

- **Validation**: check for correctness

Models can be classified into different abstraction levels:

- **Architectural-level**: the operations are represented by hardware resources (adder, multiplier, register, etc.)
- **Logic-level**: the logic functions are represented by gates (nand, nor, xor, etc.)
- **Geometrical-level**: the circuit is implemented as a composition of geometrical objects (layout)

Models can be described at different views:

- **Behavioral or Functional**: description as an abstract function
- **Structural**: description as an interconnection of logic elements
- **Physical**: description as a connection of physical objects with size, position, and layout

The abstract levels and views can be summed up by the "Y"-chart of VLSI (Gajski-Kuhn, 1983) in figure 1.1.

In this thesis, we will focus on **logic synthesis** which is the process of converting a behavioral hardware model into an optimized structural representation[1]. Synthesis creates the structure, through the interconnection of logic elements, it applies cycles of optimization and it refines the model to a lower level of abstraction. This process is clear if applied on the "Y"-chart in figure 1.1. The structure is created when, starting from a behavioral view, the model is converted to a structural one at the same level of abstraction. The optimization is an improvement of the structure at the same level of abstraction. Finally, the refinement is the process that lowers the level of abstraction.
Logic synthesis without optimization is meaningless. The main objective is to find a good structure that optimizes performances, area, energy consumption, and

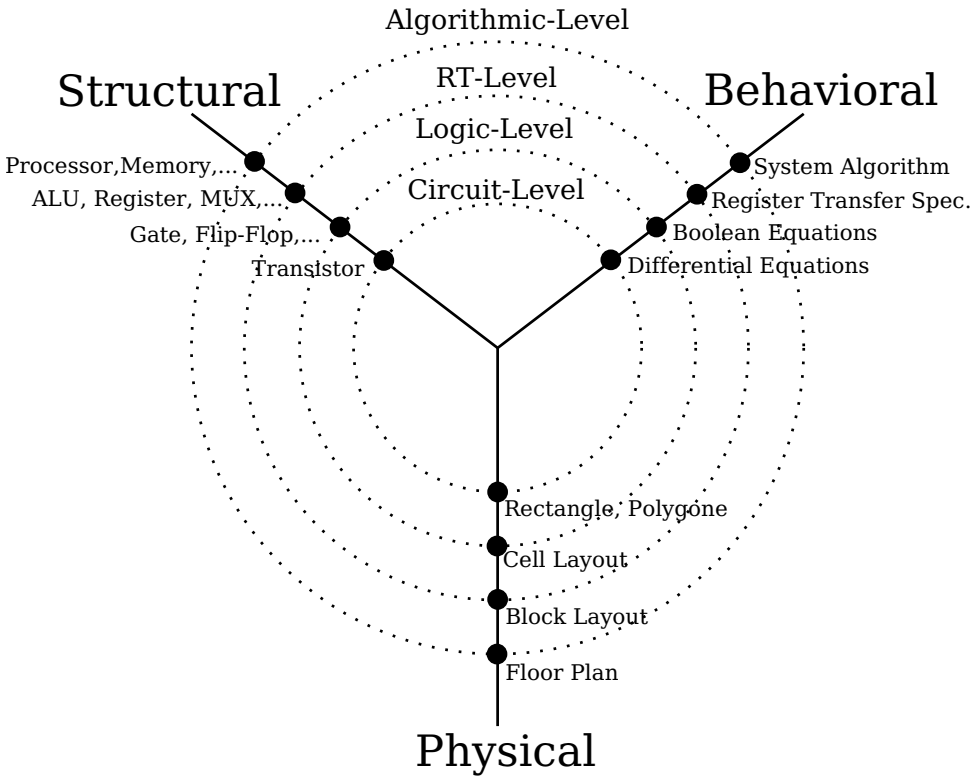

Figure 1.1. "Y"-chart of VLSI (Gajski-Kuhn, 1983)

testability. Optimization depends on trade-offs. For instance, the combination of performance enhancement and area minimization creates different alternatives of the same design as shown in the graph in figure 1.2. Faster structures tend to be bigger due to a resource parallelization while smaller structures tend to be slower due to a resource sharing. Constraints must be defined to guide logic synthesis towards an implementation that satisfies, for how it is possible, the different needs. In figure 1.2, the points in red represents sub-optimal solutions in the design space with respect to other alternatives[1]. The optimal points in black are called **Pareto points**.

A synthesis tool makes use of technology libraries to bind the structural design

[1]The solutions in black do not need to be the global minimums but they are better alternatives that can be found with the same elaborations using different constraints or a better optimization.

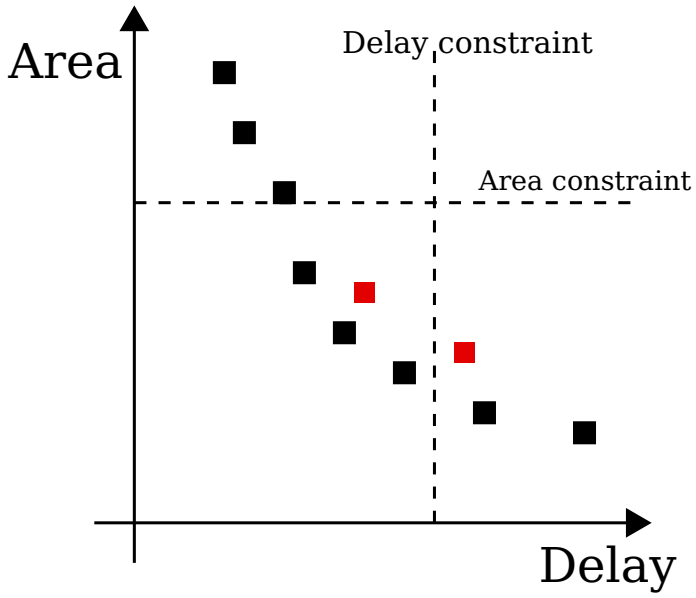

Figure 1.2. Optimization space solutions for a Delay-Area trade-off

into a netlist of gates, and to focus on technology-dependent optimizations. The binding process is called technology mapping.

Logic synthesis is a pretty mature topic of research, without considering the new emerging technology applications, with several algorithms and approaches developed mainly in the last 40 years. The approaches are principally divided in two-level and multi-level. The two-level algorithms' goal is to find a minimal cover[2] of a boolean equation, usually written in disjunctive normal form (DNF), to reduce the size of its representation. Those methods were particularly important to reduce the size of programmable logic arrays (PLA). In multi-level, a circuit is represented as a direct acyclic graph with logic equations at the nodes. This configuration allows several additional degrees of freedom and optimization possibilities. The main methods are divided in algebraic (collapsing, decomposition, substitution, extraction, simplification transformations), boolean (using *don't care*

[2]A cover is a list of implicants sufficient to define a function. A cover is minimal when contains a minimum number of irredundant implicants.

conditions), and graph-based. It is good to cite also the use of *SAT* solvers to prove logic equivalence and simplify the logic. Moreover, different types of representation of the circuit could be used: BDD (Binary Decision Diagram), AIG (*AND*-inverter graph), MIG (Majority-inverter graph), etc. Into this mature stage of synthesis methods, the development of new techniques is necessary to keep on improving the quality of the digital circuits.

The typical design flow steps are the following:

1. **Behavioral model description**: HDL (Verilog, VHDL)
2. **Floorplanning**: schematic geometrical constraints for the placement of the design
3. **Synthesis**: structure creation and technology-independent optimization
4. **Technology Mapping**: bind of the netlist to physical cells from standard cell libraries
5. **Synthesis**: technology-dependent optimization
6. **Placement and Routing**: physical cells and interconnection optimization
7. **Parasitic extraction, post-layout performance verification, and sign-off**
8. **GDSII**: format containing the geometrical description of the layout for fabrication

In this thesis, we will present optimization algorithms applicable in steps 2 and 4. Our research studies various graph-based transformations that improve the QoR. We will focus especially on timing optimization, to enhance circuits' speed, on netlists of mapped or generic (*NAND, NOR, XOR,* etc.) gates but with no electrical optimization, no sizing, and no buffering. This project wants to add some transformations that could not be found with classical algebraic methods in order

to further improve the optimization efficiency. To better present the idea, let's focus on figure 1.3 where one o more critical signals s are inputs to a cone of generic logic[3] with a certain property (in blue).

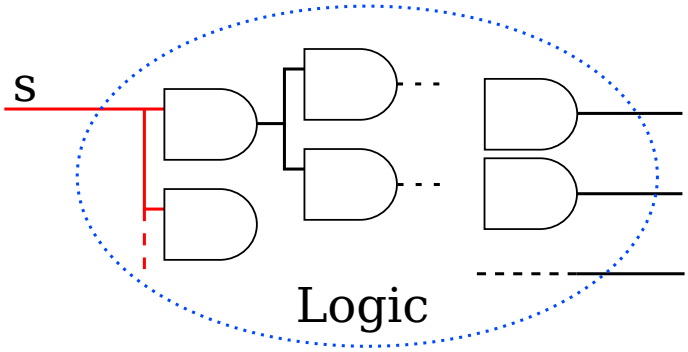

Figure 1.3. Generic logic with critical input signal s

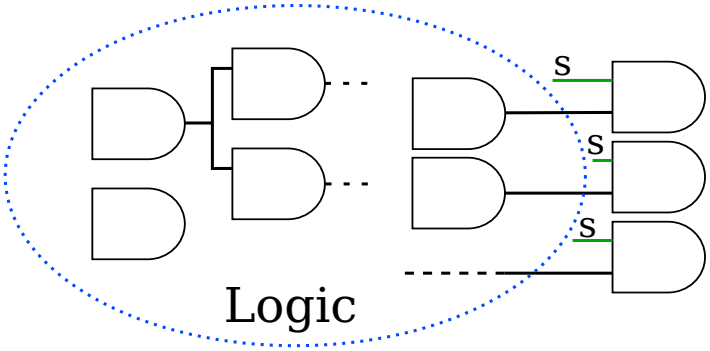

Figure 1.4. Generic logic with critical signal s moved forward at the outputs

This property of the cone can be used to move signals s to the outputs of the cone as shown in figure 1.4. Since, in the original configuration, signals s are criticals, they are slowing down the speed of the whole cone of logic. Moving these signals forward in the logic, after the cone, would decrease the logic levels which signals s have to traverse improving the overall delay. The structure of the cone could be possibly changed to allow the movement of s. The algorithms presented in this thesis born from this idea and from the different possible properties of the cone. The

[3] In figure 1.3 the *AND* gates represent generic gates or generic logic functions.

algorithm Pull-Level uses the associative property to move signals s. Pull-Factor, instead, uses the distributive property. In the end, Global Flow[4] uses controllability conditions over the cone of logic.

The goal of the thesis is to develop the algorithms that use this idea and these properties to show a beneficial improvement over the QoR once integrated inside a synthesis flow. In particular, we will also demonstrate that these logic transformations are not covered by existing algebraic methods. This project contains many challenges regarding the development of efficient algorithms, the identification of properties in a mapped netlist and the integration inside an industrial framework.

The algorithms are implemented as a part of the tool Fusion Compiler by Synopsys. Part of the framework has been adapted to support these algorithms. Moreover, optimization engines have been developed to run the algorithms over critical parts of the circuit inside the synthesis flow. The main tasks of the engines are the extraction of critical sections, the optimization of the structure, the costing of the improvement, the re-mapping (if the netlist is mapped), and the re-integration inside the circuit.

The Pull moves, based respectively on associative and distributive property, helps to improve the total negative slack (TNS) of final optimized circuits of about 13% in average keeping a constant area and a negligible impact over the runtime.
Due to integration challenges, Global Flow's integration inside the synthesis flow has not been finished. So, the QoR stats are not shown in this thesis as for the other algorithms. But from an independent analysis of the algorithm applied to already optimized netlists, Global Flow shows a direct potential improvement of the worst negative slack (WNS) for deep designs which could not be flattened.

1.1 Fusion Compiler

Fusion Compiler is an EDA tool which is built on a single data model that allows an RTL-to-GDSII implementation.

"The Fusion Compiler single data model contains both logical and physical information to enable sharing of library, data, constraints, and design intent throughout the implementation flow. The advanced data model is architected to support ultra-large designs with the smallest possible memory footprint. The key innovation of Fusion Compiler is synthesis and implementation tools access to each others technology, including sharing of optimization engines between the two domains. Fusion Compiler integrates all synthesis, place-and-route and signoff engines on a single data model and eliminates data transfer delivering fastest design closure with highest throughput."[2]

1.2 Synopsys

The work has been developed during an internship of six months in Synopsys. Synopsys is an electronic design automation company based in Mountain View, California that focuses mainly on silicon design and verification, silicon IP, and software security and quality. I worked in a research and development team that works on the software Fusion Compiler. I was based in Montbonnot-Saint-Martin, France, from the start of July to the end of December.

Chapter 2

Pull Moves

2.1 Introduction

Given a netlist of mapped or generic gates, but with no electrical optimization, no sizing, no buffering, we want to optimize its structure for delay. In this chapter, we will present two types of moves that use the associative and the distributive property. With the term "move", we refer to a logic transformation that affects a small number of gates. For a delay estimation, we use logical effort based delay[3] which is a good balance between generality, simplicity, and precision. Logical effort estimates the delay of a gate using a delay technology model τ defined as an inverter driving another single identical inverter. The absolute delay is then defined by $d_{abs} = d\,\tau$. The delay d of a gate s is described by $d_s = g_s h_s + p_s$. g is the logical effort of s, which is the ratio of the input capacitance of gate s and the one of an inverter delivering the same output current. h is the electric effort of s, which is the ratio between the input capacitance of the load and the capacitance of gate s. p is the parasitic delay of s. Table 2.1 contains a reference of logical effort g with the ratio $\upsilon = 2$. υ is the ratio of an inverter's pullup and pulldown transistor width[1]. Table 2.1, instead, contains an estimation of the parasitic delay for simple layouts.

[1]Note that a n-input *NAND* can be implemented to have a $n\log(n)$ delay w.r.t. 2 inputs. In practice, we do not see large input *NAND/NOR*s in libraries but this may change in the future.

	Number of Inputs					
Gate type	**1**	**2**	**3**	**4**	**5**	**n**
Inverter	1					
NAND		4/3	5/3	6/3	7/3	(n+2)/3
NOR		5/3	7/3	9/3	11/3	(2n+1)/3
Multiplexer		2	2	2	2	2
XOR (parity)		4	12	32		

Table 2.1. Logical effort of gates based per number of inputs with $\upsilon = 2$

Gate Type	**Parasitic Delay**
Inverter	p_{inv}
n-input NAND	np_{inv}
n-input NOR	np_{inv}
n-input multiplexer	$2np_{inv}$
XOR, XNOR	$4p_{inv}$

Table 2.2. Parasitic gates based per number of inputs with simple layouts

Timing optimization is the most important aspect of logic synthesis. In the years, many different techniques have been developed. In synthesis, circuits are modeled as direct acyclic graphs (DAG's). Every node represents a logic function or a signal, and every edge represents a wire connection. In this representation, we can distinguish source nodes, which are the inputs, and sink nodes, which are the outputs of the network. These nodes are not associated with a logic function but with a signal. The internal nodes, instead, are represented by a boolean function or a logic gate. In this thesis, we will use the term node or gate to refer to an internal node of a network.

2.2 Associative Property in Networks

In this first part, we will focus our attention on a specific cone of a network for which the associative property is valid. In an associative logic function, the order of the operations can be changed without modifying the logic functionality. For instance, in the function $a \oplus (b \oplus c)$, any permutation of the variables, like $b \oplus (a \oplus c)$, does not change the result of the function. In boolean functions, associative property is valid for the operators *AND*, *OR* and *XOR*. In a network, each node can be described by one or more boolean functions. Since we are working on multi-level networks, we need to find when this property is true among a bunch of nodes in order to reorder them to obtain a gain in delay. This operation is not trivial as it might seem. Let's take the following small network as an example:

$$g_1 = \bar{a} \vee \bar{b}$$

$$g_2 = \bar{g_1}$$

$$g_3 = g_2 \wedge \bar{c}$$

Doing some nodes simplification and collapsing, we can rewrite the network as:

$$g_1 = a \wedge b$$

$$g_2 = g_1 \wedge \bar{c}$$

This network has an associative property among signals a, b, and $\bar{c}$, but, in the first configuration, it was not easy to spot it as in the second one. In fact, in a multi-level network, nodes can be represented by complex functions and they may exhibit this property only when nodes are collapsed together to a unique logic function.

The associative property may be used to optimize the delay in a cone of logic. Let's take as example figures 2.1 and 2.2. For each node, the notation (a, r) represents the arrival time a and the required time r. Let's suppose that the delay of a *AND2* gate, is 2. The load delay of each gate is neglected. The arrival time of a

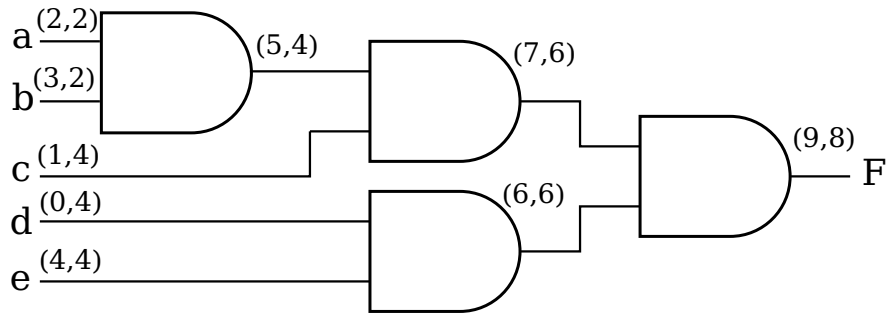

Figure 2.1. Not optimized network with associativity

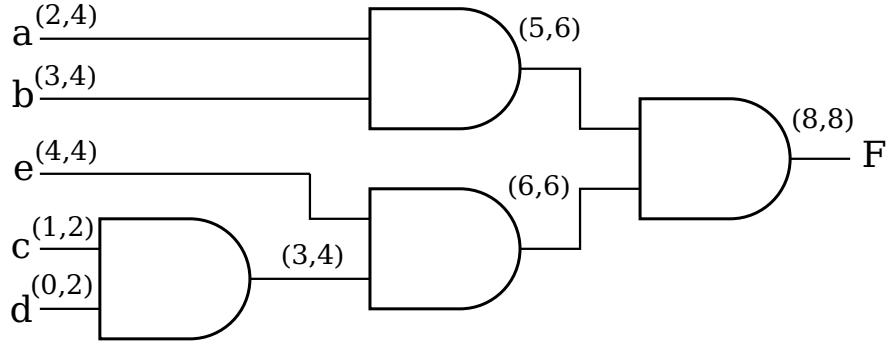

Figure 2.2. Optimized network with associativity

node, or signal, is the time elapsed for data to arrive at the node. For a gate, it is calculated as the worst arrival time among its inputs plus the delay of the gate. The required time is the time within which data is required to arrive at some internal node of the design. This value is originally specified by some timing constraints. For a gate, it is calculated as the minimum required among its outputs minus the gate delay. When, for a node, the arrival is greater than the required, the node is considered critical. The sequence of critical nodes defines critical paths. In figure 2.1 we can see a not optimized network for which the required time at F is not met. In figure 2.2, the same network is re-arranged using the associative property for delay optimization. Now the required time at F is met.

In common netlists, the ordering of gates with associative property is often limited by multiple outputs. Just imagine if, in figure 2.1, the gate connected to d and e

has another output G. Clearly, the optimization in figure 2.2 would not be possible as output G could not be represented in the network. The gate connected to d and e cannot be removed. Aware of that, we will separate the analysis in two different cases: for single-output and multi-output networks.

2.2.1 Single Output Networks

When a network is single-output, an optimal arrangement of the net can be found using Huffman decomposition. Huffman decomposition creates an optimal ordering of a tree of logic. Inputs are inserted in a priority queue of nodes ordered by arrival time. Nodes with lower arrival time have higher priority. Each step, the two nodes with the highest priority are picked and used to create together a new node. For this node, the arrival time is calculated and, then, it is inserted inside the queue. The algorithm ends when the queue contains only one node which is the root of a time-optimal decomposed tree. This approach is simple and finds an exact solution. For instance, figure 2.2 represents an application of a Huffman decomposition from the network in figure 2.1. At the start, the priority queue contains nodes $\{d, c, a, b, e\}$. After the first step, node dc is created. Its arrival time of $max(d_c, d_d) + d_{and2} = 3$ is calculated and it is inserted inside the queue $\{a, b, dc, e\}$. Proceeding with this method, the network will be delay optimal.

2.2.2 Multiple Output Networks

When the network has multiple outputs, exact solutions are expensive and often not solvable in a reasonable amount of time. Different types of elaboration may be performed following some heuristics. Usually, algorithms are run on a small cone of logic extracted by a critical driver. Here, we present an elaboration called Pull-Level.

2.3 Pull-Level Move

The pull-level move is a timing driven logic restructuring trick whose goal is to improve the delay across a cluster of nodes with associative property pushing some critical signals forwards to the outputs of the cluster. Pull-Level is a graph-based elaboration that is applied on a small sub-networks of depth 4 extracted by a critical driver. Nodes are divided, based on the depth level, in the sets $\mathcal{A}, \mathcal{B}, \mathcal{C}$, and $\mathcal{D}$. Some nodes at level $\mathcal{A}$ define a critical path through node $\mathcal{D}$. The idea is to move $\mathcal{A}$ nodes closer to $\mathcal{D}$ to reduce the number of logic levels that critical signals have to traverse. When the associative property is blocked by multiple outputs, nodes are duplicated in order to be able to pull critical signals up to the critical output. An example of an application is shown below. For an understandable notation, nodes' names are assigned based on the set they belong to.

Inputs : $a,\ b,\ g_1,\ g_2,\ e,\ f$

Outputs : $B,\ D$

$A = a \wedge b$

$B = A \wedge g_1 \wedge g_2$

$C = B \wedge e$

$D = C \wedge f$

In this case A is critical and defines a critical path through D. D and B have multiple outputs. The move will push the A's fanin up to D. Here the network is shown after the move is applied:

Inputs : $a\ b\ g_1\ g_2\ e\ f$

Outputs : $B_1\ D$

$A = a \wedge b$

$$B_1 = A \wedge g_1 \wedge g_2$$

$$B_2 = g_1 \wedge g_2$$

$$C = B2 \wedge e$$

$$D = C \wedge f \wedge a \wedge b$$

In this way critical A is moved 3 level of logic forward and the slack is improved. The general rule, to have gain in timing, is that the associative property must be valid at least for $\mathcal{B}$ and $\mathcal{C}$ nodes. If $\mathcal{A}$ and $\mathcal{D}$ are not associative, the pull of one level of logic forward is anyway guaranteed. Differently from the Huffman decomposition, this algorithm does not assure an improvement since some side paths may be slowed down after the transformation. Each manipolation must be tested before the commit.

2.3.1 The algorithm

The slack of a signal s can be significatly improved if s is pushed at least two levels forward in the logic.

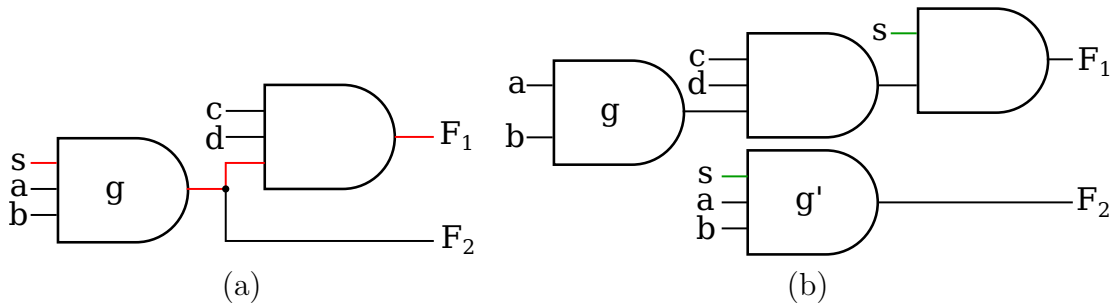

Figure 2.3. Network before (a) and after pulling signal s (b)

In figure 2.3(a), signal s is critical and defines a critical path to F_1. Since the associative property is valid, s can be pulled closer to F_1 in order to improve the *AND* delay levels from two to one. But the first gate g, where s is connected, is

multi-output. To maintain the logic functionality of the network, g must be duplicated to separate the critical path from non-critical. Then, in the critical path, s can be disconnected from g and pulled up to the output F_1 creating a new *AND2* gate, figure 2.3(b). This is the base idea in Pull-Level to improve delay.

The move is applied on a node driver C contained in a cone of logic of size 4. From $C \in \mathcal{C}$, the other set of nodes $\mathcal{A}, \mathcal{B}, \mathcal{D}$ are extracted. Among the boolean operators, three different types of associativity can be found: *AND, OR*, and *XOR*. To start, let's take into consideration networks where only these types of gates are used.

We said before that a signal s is generally improved if moved by 2 levels forward in the logic. So we define that for the sets $\mathcal{B}$ and $\mathcal{C}$, the nodes must have the same associative property. For sets $\mathcal{A}$ and $\mathcal{D}$ we define three other sets:

- **Same-critical**: contains critical gates of the same associative type of C
- **Miscellaneous-critical**: contains critical gates of a different type of C
- **Non-critical**: contains non-critical gates

The Pull-Level move set is defined as $\mathcal{P}^L = \{\mathcal{A}, \mathcal{B}, \mathcal{C}, \mathcal{D}\}$ where $\mathcal{A} = \{\mathcal{A}_s, \mathcal{A}_m, \mathcal{A}_n\}$, $\mathcal{B} = B$, $\mathcal{C} = C$, and $\mathcal{D} = \{\mathcal{D}_s, \mathcal{D}_m, \mathcal{D}_n\}$. These sets are obtained based on the following rules:

- C node is a critical driver of type t: *AND, OR*, or *XOR*.
- B node is a single critical fanin of C of type t. Also, B has a single critical output through C. If these two conditions are not respected, some paths would be slowed down once the pull move is applied. For instance, just imagine if, in figure 2.3(a), another input of the gate connected to F_1 is critical. From the transformation in figure 2.3(b), that path would be slowed down.
- A nodes are fanin nodes of B. They are divided in same, miscellaneus and non-critical sets. Critical nodes of type t are inserted in $\mathcal{A}_s$, critical nodes of

different type are inserted in $\mathcal{A}_m$, and non-critical nodes in $\mathcal{A}_n$. $\mathcal{A}$ nodes must have one single critical output through B

- D nodes are fanout nodes of C. They are also divided in same, miscellaneus and non-critical sets accordingly.

At the start, the algorithm executes a matching phase, starting from a node C, where the $\mathcal{P}_L$ sets are retrieved. This phase is run for all the critical internal nodes in the network, in topological order, until one of them satisfies the properties. If the right conditions are detected, the move is applied.
Nodes are considered critical when their slack is negative and within a defined margin, small enough, from the worst one in the network. The worst negative slack defines a path that is limiting the speed of the network.
When multiple outputs are present, some nodes must be kept in the network to preserve its functionality. In this case, a flag *don't touch* will be assigned.
The following rewiring options are detected:

- If $\mathcal{D}_n = \{\emptyset\}$, all $\mathcal{D}$ nodes are critical, the move will pull $\mathcal{A}_s$ and $\mathcal{A}_m$ up to $\mathcal{D}$
- If $\mathcal{D}_n \neq \{\emptyset\}$, C is *don't touch*
- If C is *don't touch* or B has multiple outputs, B is *don't touch*
- $\forall a \in \mathcal{A}_s$, if B is *don't touch* or a has multiple outputs, a is *don't touch*

The transformation of the network consists in the following operations:

1. Select the set $\mathcal{M} = \{\mathcal{A}_m \cup fanin(\mathcal{A}_s)\}$[2]. It represents all the signals to be pulled up: $\mathcal{A}_m$ and all the fanin of $\mathcal{A}_s$ nodes. Instead of $\mathcal{A}_s$, we pull directly $fanin(\mathcal{A}_s)$ because they are input to a gate with associativity t
2. $\forall d \in \mathcal{D}_m$, create a new node CD_d of type t with inputs C and $\mathcal{M}$. Then cennect CD_d to d replacing C connection. Since d is not of type t, so we must pull the signal before d creating a new gate CD_d

[2]With the function $fanin(x)$ we consider a set containing the input nodes of node x

3. $\forall d \in \mathcal{D}_s$, substitute d with a node of type t with inputs $\{fanin(d) \cup \mathcal{M}\}$. The critical signals are inserted in the fanin of d

4. Update B:

   (a) If B is not *don't touch* and $\mathcal{A}_n = \{\emptyset\}$, B is removed because it will remain without inputs after the move

   (b) If B is *don't touch* and $\mathcal{A}_n \neq \{\emptyset\}$, another node B' is created with $\mathcal{A}_n$ inputs. In order to pull, the tree of logic is duplicated per B's outputs

   (c) If B is not *don't touch* and $\mathcal{A}_n \neq \{\emptyset\}$, B is replaced by a node of type t with $\mathcal{A}_n$ inputs. The pulled inputs are removed from B

   (d) If B is *don't touch* and $\mathcal{A}_n = \{\emptyset\}$, nothing to do

5. Update C:

   (a) If C is *don't touch* and $\mathcal{A}_n = \{\emptyset\}$, another node C' is created with $\{fanin(C) \setminus B\}$ inputs. Since B', after the pull will remain without inputs, its input to C' is not created. For all the CD_d nodes and all the $\mathcal{D}_s$ nodes, substitute C input with C'. The optimized tree has been created

   (b) If C is *don't touch* and $\mathcal{A}_n \neq \{\emptyset\}$, C is duplicated creating a new node C'. Then B fanin of C' is substituted with B'. For all the CD_d nodes and all the $\mathcal{D}_s$ nodes, substitute C input with C'. The optimized tree has been created

   (c) If C is not *don't touch* and $\mathcal{A}_n = \{\emptyset\}$, B fanin of C is removed

   (d) If C is not *don't touch*, B is *don't touch* and $\mathcal{A}_n \neq \{\emptyset\}$, B fanin of C is substituted with B'

6. $\forall a \in \mathcal{A}_s$, if a is not *don't touch*, a can be removed

After the move, the network is costed to verify delay improvement.

A mapped network may contain complex or generic gates as *AOI21* or multiplexers.

Moreover, the mapper tends to use *NAND* and *NOR* gates instead of the less efficient *AND*s and *OR*s. So, to apply the pull-level, the gate type is not sufficient to get the associative property type. The operation may be quite complex since it may need to collapse all the logic into a single logic function which can be re-elaborated using associative property. But, with this operation, it would be likely to lose the logic structure of the network, already improved by other logic transformations, and to be limited by multiple outputs. So, the idea is to decompose[3] the circuit accordingly to the associative property types:

1. Pull-level is applied to the original sub-network. It is likely to match a *XOR* associativity (*XOR*s are implemented, and mapped, as *XOR* gates)
2. The sub-network is decomposed using only *AND* gates and inverters. The associative property of *AND*s is matched
3. The sub-network is decomposed only using *OR* gates and inverters. The associative property of *OR*s is matched

Since the network has a small size, more than one critical associative property cannot be present. So, as soon as one move is successful, the algorithm returns the optimized circuit.

2.3.2 Examples

Let's focus now on some visual examples to understand how the network is transformed by the move.
Picture 2.4, shows a network with *AND* associativity and a critical signal a. The circuit can be described by the following equations:

Inputs : $a,\ b,\ g_1,\ g_2,\ e$

Outputs : $F_1,\ F_2$

[3]Rewrite complex logic functions using elementary gates

$A_1 = a$

$A_2 = b$

$B = A_1 \wedge A_2$

$F_1 = C = B \wedge g_1 \wedge g_2$

$F_2 = D = C \wedge e$

The retreived sets are:

$\mathcal{A} = \{\mathcal{A}_s = \{\emptyset\}, \mathcal{A}_m = \{a\}, \mathcal{A}_n = \{b\}\}$

$\mathcal{B} = \{B\}$

$\mathcal{C} = \{C\}$

$\mathcal{D} = \{\mathcal{D}_s = \{D(F_2)\}, \mathcal{D}_m = \{\emptyset\}, \mathcal{D}_n = \{F_1\}\}$

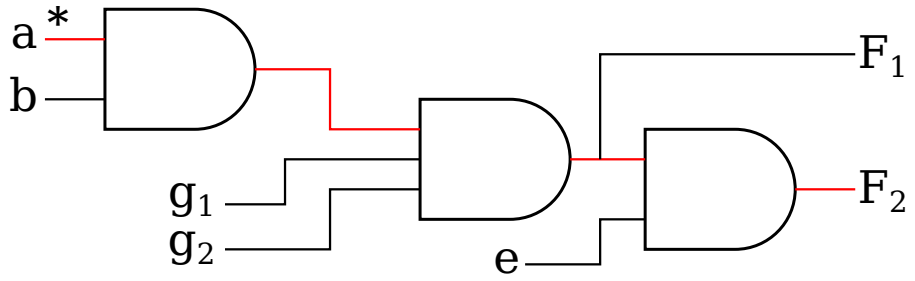

Figure 2.4. Ex1: Not optimized network with associativity

First of all, we notice that a defines a critical path through F_2. F_1, instead, is non-critical. The goal is to improve the delay only for F_2. The only signal that will be pulled up to F_2 is $\mathcal{M} = \{a\}$. From picture 2.5, we can notice how signal a has be added as an input to the $\mathcal{D}_s$. Morover, since both nodes B and C are *don't touch*, they have been duplicated to preserve the tree with root F_1. Node C has been duplicated creating node C'. Node B' has not been created because it would remain with a single input (it would be only a buffer). At the end of the elaboration, we can notice how the delay of a is improved from three gate levels to

only one.

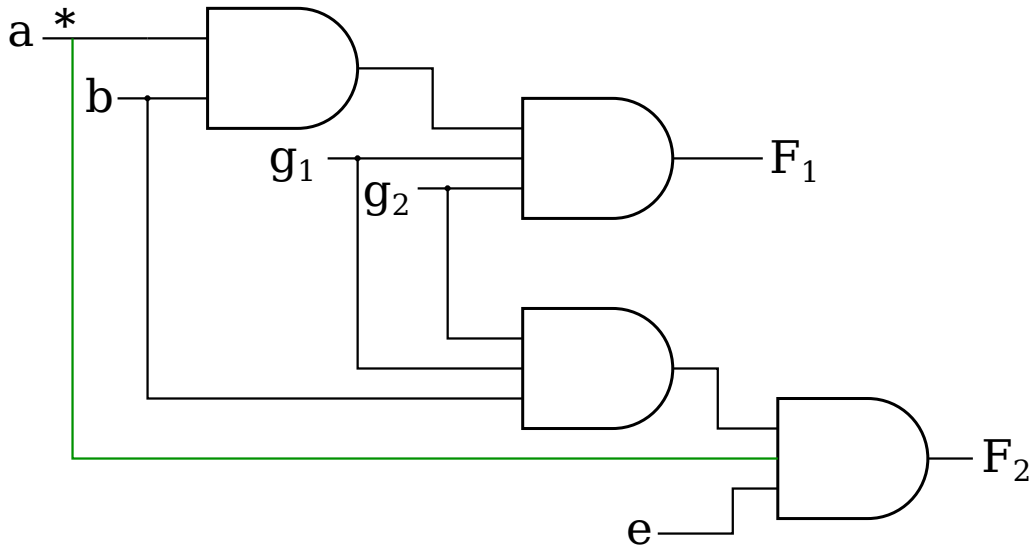

Figure 2.5. Ex1: Optimized network with associativity

A network with *XOR* associativity and two critical gates is shown in picture 2.6. It can be described by the following equations:

Inputs : $a,\ b,\ c,\ d,\ e,\ g$

Outputs : $F_1,\ F_2$

$A_1 = \bar{a} \vee \bar{b}$

$A_2 = \bar{c} \wedge \bar{d}$

$F_1 = B = A_1 \bar{\oplus} A_2$

$C = B \oplus e$

$F_2 = D = C \bar{\oplus} g$

The retreived sets are:

$\mathcal{A} = \{\mathcal{A}_s = \{\emptyset\}, \mathcal{A}_m = \{A_1, A_2\}, \mathcal{A}_n = \{\emptyset\}\}$

$\mathcal{B} = \{B\}$

$\mathcal{C} = \{C\}$

$\mathcal{D} = \{\mathcal{D}_s = \{D(F_2)\}, \mathcal{D}_m = \{\emptyset\}, \mathcal{D}_n = \{\emptyset\}\}$

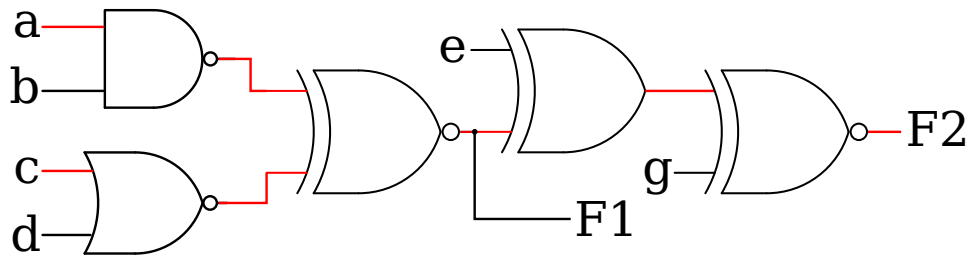

Figure 2.6. Ex1: Optimized network with associativity

In this example all the two gates A_1 and A_2 are critical up to F_2, so $\mathcal{M} = \{A_1, A_2\}$. Moreover, B is *don't touch* while C is not *don't touch*. Since $\mathcal{A}_n = \{\emptyset\}$, node B is disconneted from C to drive indipendently F_1. Node C, then, remains with only one input and so it is removed. At the end we will have a big *XOR* node D that will drive the critical output, in picture 2.7. A Huffman decomposition may be applied on D to decompose the *XOR* into multiple stages and optimize even further the delay.

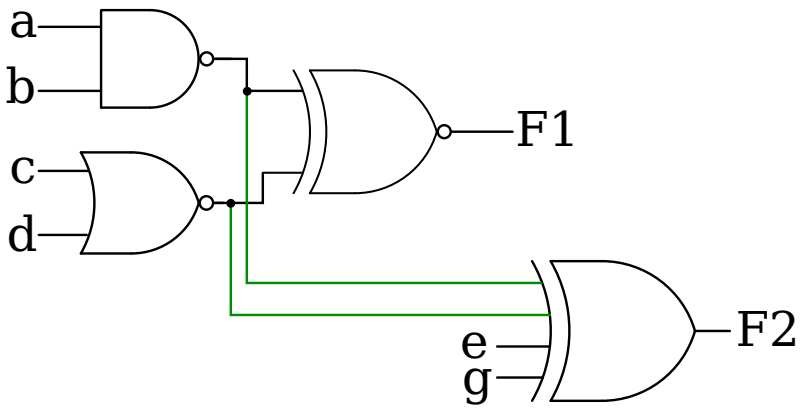

Figure 2.7. Ex1: Optimized network with associativity

2.4 Distributive Property in Networks

After having discussed some possible approaches for the associative property, we now try to analyze the distributive property and how it can be used to delay optimize a network. We define the distributivity of operator ψ over operator ϕ when $x\,\psi\,(y\,\phi\,z)$ can be rewritten as $(x\,\psi\,y)\,\phi\,(x\,\psi\,z)$ and $(y\,\phi\,z)\,\psi\,x$ can be rewritten as $(y\,\psi\,x)\,\phi\,(z\,\psi\,x)$. In boolean algebra, operator ψ can be only an *AND*, while operator ϕ can be an *OR* or a *XOR*. Thus, we have two possible scenarios: *AND-OR* distributivity and *AND-XOR* distributivity.
Let's start with an example to demonstrate how the distributive property may improve the delay of a sub-network.

Inputs : $a\ b\ c\ d\ e\ g$

Outputs : D

$A_1 = a \wedge b$

$A_2 = c \wedge d$

$B = A_1 \vee A_2$

$C = B \wedge e$

$D = C \vee g$

This *AND-OR* network has a possible distributive move between C and B. If we distribute C over B and we collapse some nodes, we obtain the following circuit:

Inputs : $a\ b\ c\ d\ e\ g$

Outputs : D

$A_1 = a \wedge b \wedge e$

$A_2 = c \wedge d \wedge e$

$D = A_1 \vee A_2 \vee g$

If an A nodes' input is critical, its slack would be improved. Let's try to demonstrate it using logical effort. First we optimize the network mapping it to a *NAND-INV* graph. In fact, $(a \wedge b) \vee (c \wedge d) = \overline{\overline{(a \wedge b)} \wedge \overline{(c \wedge d)}}$: an *AND-OR* structure can be mapped as a *NAND-NAND* which is more efficient. So we can rewrite the initial network as:

Inputs : $a\ b\ c\ d\ e\ g$

Outputs : D

$A_1 = \overline{a \wedge b}$

$A_2 = \overline{c \wedge d}$

$B = \overline{A_1 \wedge A_2}$

$C = \overline{B \wedge e}$

$D = \overline{C \wedge \bar{g}}$

So, for signal a we can calculate its delay to D. For simplicity, we will assume that the input capacitance of a is equal to the load of D and all the *NAND2* gates are identical. Accordingly to table 2.1, the logical effort of a *NAND2* gate is $4/3$. From table 2.1, instead, the parassitic delay is $p = 2 \times p_{inv} = 2$. Using the logical effort for multistage networks[3]: $G = g_0 g_1 g_2 g_3 = (4/3)^4$, $H = C_{out}/C_{in} = 1$, and the branching effort $B = 1$ since there is no branching. The delay D_p at of the path $a \to D$ is $D_p = N(GBH)^{1/N} + P = 4 \times 4/3 + 4 \times 2 = 13.\bar{3}$. If we use the same method on the distributed optimized network, the delay D_p of path $p = a \to D$ is $D_p = 2 \times 5/3 + 2 \times 3 = 9.\bar{3}$. For signal a, the delay is significatly improved.
In order to see also the other side of the medal, let's do the same analysis for signal e. In the not optimized network, its delay is $D_p = 2 \times 4/3 + 2 \times 2 = 6.\bar{6}$. For the optimized one, instead, $D_p = 2 \times 5/3 + 2 \times 3 = 9.\bar{3}$. For signal e, as for g, this transformation worsened the delay. Thus, to improve the delay of the network, we must be carefull about which signals are critical.

2.5 Pull-Factor Move

The Pull-Factor move is a timing driven logic restructuring trick whose goal is to improve the delay across a cluster of NAND or XOR gates by pushing some signals forward towards the outputs of the cluster. There are two types of pull-factor moves, *AND-OR* and *AND-XOR*. As Pull-Level, it is applied to sub-networks of depth 4 with distributive property between B and C. The idea is to pull C node back to A level, for critical nodes, to reduce the levels of logic of critical paths.
Let's take the following network showed in picture 2.8. In this case a and c are critical and define a critical path through F_1.

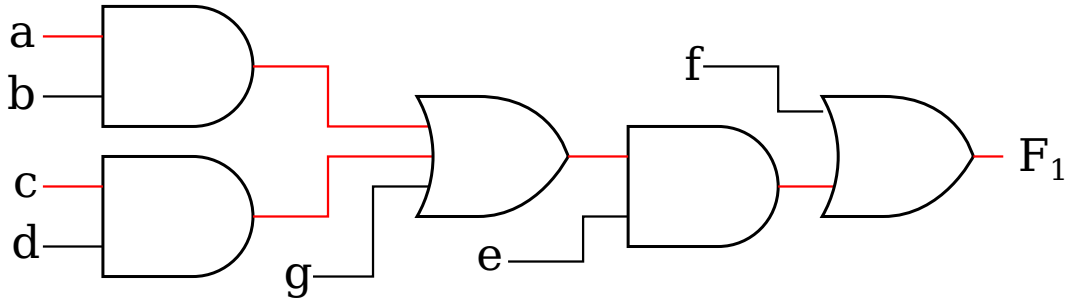

Figure 2.8. *AND-OR* network

Inputs : $a\ b\ c\ d\ g\ e\ f$

Outputs : D

$A_1 = a \wedge b$

$A_2 = c \wedge d$

$B = A_1 \vee A_2 \vee g$

$C = B \wedge e$

$D = C \vee f$

In this network, we will proceed with the method we have seen in the previous section. The network, in picture 2.9, is obtained using the distibrutive property

of the *AND* over the *OR*. Collapsing the *OR* gates, we obtain the circuit in figure 2.10.

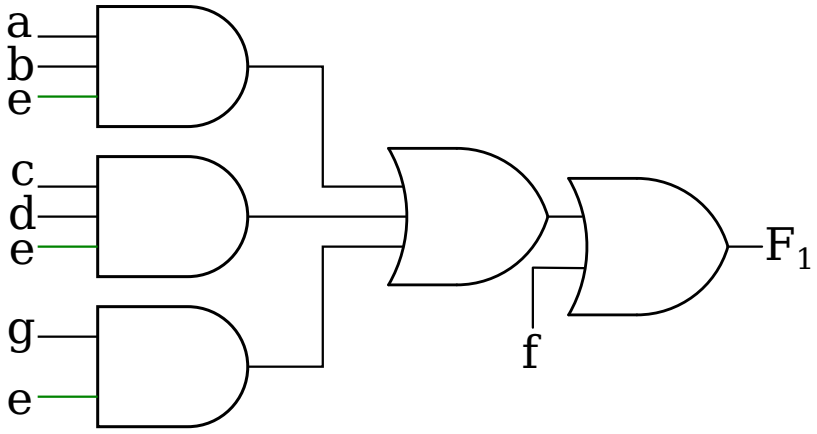

Figure 2.9. *AND-OR* network optimized

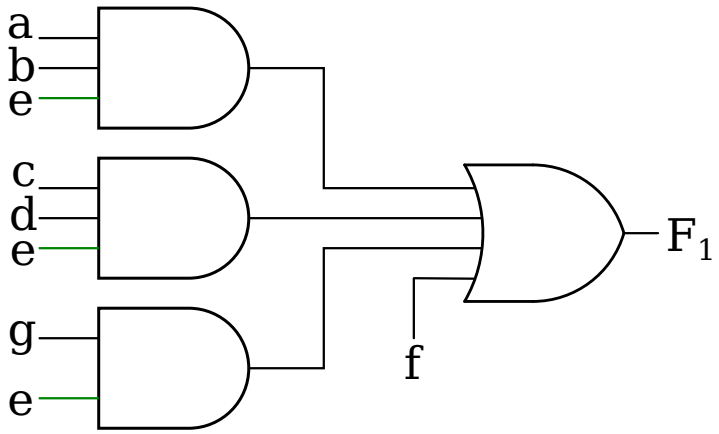

Figure 2.10. *AND-OR* network optimized and collapsed

Inputs : a b c d g e f

Outputs : B_1 D

$A_1 = a \wedge b \wedge e$

$A_2 = c \wedge d \wedge e$

$$C = g \wedge e$$

$$D = A_1 \vee A_2 \vee C \vee f$$

The delay of signals a and c is improved. Of course, the delay of e and g is worsened but, since these signal were not critical, the slack at F_1 is improved.

If Pull-Factor is applied on a mapped network, it is very unlikely to find *AND* and *OR* gates. Their implementation is particularly unefficient with respect to *NAND* and *NOR* gates. Moreover, we saw in the previous section how easily the *AND-OR* sequence may be rewritten as *NAND-NAND*. Thus, we decided to work on a NAND, INV, XOR decomposed network which will give us enough generalization to apply the algorithm and to detect the distributive proprieties. The new matching sequences will be *NAND-NAND* and *NAND-XOR*.

Let's introduce another concept linked to a *NAND-XOR* structure. Distributive property, is valid over *XORs* but not over *XNORs*. Also, inputs are *NANDs* which cannot be distributed. But *XOR* gates have a parity property: $a \oplus \bar{b} = \bar{a} \oplus b = a \bar{\oplus} b$, and $\bar{a} \oplus \bar{b} = a \oplus b$. Also $a \oplus b = a \oplus b \oplus 0$, and $a \bar{\oplus} b = a \oplus b \oplus 1$.
So let's see two cases of how the inverters can be simplified to show a distributive property.
Example 1:

$$A_1 = \overline{a \wedge b}$$

$$A_2 = \overline{c \wedge d}$$

$$B = A_1 \oplus A_2$$

$$C = \overline{B \wedge e}$$

Using the parity property of *XORs*, we can push and simplify the inverters of the *NAND* gate in A_1 and A_2. Moreover we can rewrite C as *AND+INV*.

$$A_1 = a \wedge b$$

$$A_2 = c \wedge d$$

$$B = A_1 \oplus A_2$$

$$C = B \wedge e$$

$$C' = \overline{C}$$

Now, the distributive property is clear and we can distribute e over A.

$$A_1 = a \wedge b \wedge e$$

$$A_2 = c \wedge d \wedge e$$

$$B = A_1 \oplus A_2$$

$$B' = \overline{B}$$

After the move, we *NAND* decompose the network and we collapse the inverter B' over the *XOR* gate.

$$A_1 = \overline{a \wedge b \wedge e}$$

$$A_2 = \overline{c \wedge d \wedge e}$$

$$B = A_1 \bar{\oplus} A_2$$

Let's take, instead, a network where the inverters' parity around the *XOR* gate is odd.
Example 2(a):

$$A = \overline{a \wedge b}$$

$$B = A \oplus c$$

$$C = \overline{B \wedge d}$$

Let's move the inverter over the *XOR*.

$$A = a \wedge b$$

$$B = A \bar{\oplus} c$$

$$C = B \wedge d$$

$$C' = \overline{C}$$

Since an inverter is still present over the *XOR* gate, we add one input at 1 which substitute the inverter.

$$A = a \wedge b$$

$$B = A \oplus c \oplus 1$$

$$C = B \wedge d$$

$$C' = \overline{C}$$

Now, the distributive property is valid. Applying the move and simplifying the logic:

$$A_1 = \overline{a \wedge b \wedge d}$$

$$A_2 = \overline{c \wedge d}$$

$$B = A_1 \oplus A_2 \oplus \bar{d}$$

The inverter C' has been simplified over the *XOR* after inserting back the inverters.

Another alternative could be to move the inversion over c.
Example 2(b):

$$A = a \wedge b$$

$$B = A \bar{\oplus} c$$

$$C = B \wedge d$$

$$C' = \overline{C}$$

$$A = a \wedge b$$

$$B = A \oplus \bar{c}$$

$$C = B \wedge d$$

$$C' = \overline{C}$$

$$A_1 = \overline{a \wedge b \wedge d}$$

$$A_2 = \overline{\bar{c} \wedge d}$$

$$B = A_1 \bar{\oplus} A_2$$

In example 2(b) we obtain less literals, thus a smaller network, than in example 2(a). The choice between the two is based on the nodes criticality and it will be deepen in the next section.

2.5.1 The algorithm

In the previous section, we saw how we can use the distributive property to enhance the delay in a network. Here we will define a strategy and a general approach to restructure the logic. As for Pull-Level, the move is applied to a sub-network of depth 4. Accordingly the same set of nodes are extracted: $\mathcal{P}^F = \{\mathcal{A}, \mathcal{B}, \mathcal{C}, \mathcal{D}\}$ where $\mathcal{A} = \{\mathcal{A}_s, \mathcal{A}_m, \mathcal{A}_n\}$, $\mathcal{B} = B$, $\mathcal{C} = C$, and $\mathcal{D} = \{\mathcal{D}_s, \mathcal{D}_m, \mathcal{D}_n\}$. The matching methodology, anyway, is different. We remind that the move works on a *NAND-XOR-INV* decomposed network. All the inverters, input to *XORs*, are pushed through using parity property. Here we define the matching rules:

- C node is a critical gate of type *NAND*
- B node is a single critical fanin of C, with only one critical output to C, of type t: *NAND*, *XOR* or an inverter. If it is an inverter, its fanin must be a *XOR*

gate. That *XOR* will be the B node, the inverter will be named B_{inv}, and the type t will be *XOR*. If B has multiple critical outputs or C has more than one critical input, the algorithm could not be applied since some paths would be slow down from the pull. This concept has been previously demonstrated using logical effort.

- A nodes are fanin nodes of B. They are divided in same, miscellaneous and non-critical sets. Critical nodes of type *NAND* are inserted in $\mathcal{A}_s$, critical nodes of different type are inserted in $\mathcal{A}_m$, and non-critical nodes in $\mathcal{A}_n$. A nodes must have one single critical output through B.
- D nodes are fanout nodes of C. They are also divided into same, miscellaneous and non-critical sets. Critical nodes of type t are inserted in $\mathcal{A}_s$, critical nodes of different type are inserted in $\mathcal{A}_m$, and non-critical nodes in $\mathcal{A}_n$

As for Pull-Level, when multiple outputs are present, trees of nodes are duplicated in the network to preserve its functionality. The flag *don't touch* will be assigned to these nodes. If, during the B matching phase, an inverter is detected (*NAND-XOR* case), the flag *invx* will be set. We need to be aware that also the inverter might have multiple outputs. More conditions with respect to Pull-Level have to be considered.
The following rewiring options are detected:

- If $\mathcal{D}_n \neq \{\emptyset\}$, C is *don't touch*
- If $\exists invx$, if C is *don't touch* or B_{inv} has multiple outputs, B_{inv} is *don't touch*.
- If C is *don't touch* or B_{inv} is *don't touch* or B has multiple outputs, B is *don't touch*
- $\forall a \in \mathcal{A}_s$, if B is *don't touch* or a has multiple outputs, a is *don't touch*

From now on, once the pull type and the $\mathcal{P}^F$ sets are retreived, for simplicity, the algorithm explaination will be sepatated for *NAND-NAND* and NAND-XOR.

Transformation for *NAND-NAND* pull factor type:

1. Define set $\mathcal{N} = \{fanin(C) \setminus B\}$

2. Update A:

   (a) $\forall a \in \mathcal{A}_s$ if a is *don't touch*, create a *NAND* node a' with inputs $\{fanin(a) \cup \mathcal{N}\}$. C side inputs are distributed to $\mathcal{A}_s$. Then, add node a' to set $\mathcal{M}$

   (b) $\forall a \in \mathcal{A}_s$ if a is not *don't touch*, replace node a by a *NAND* with inputs $\{fanin(a) \cup \mathcal{N}\}$. C side inputs are distributed to $\mathcal{A}_s$. Then, add node a' to set $\mathcal{M}$

   (c) $\forall a \in \mathcal{A}_m$, create an inverter i_a at the output of a. Then, create a *NAND* node a' with inputs $\{\mathcal{N} \cup i_a\}$. C side inputs are distributed to $\mathcal{A}_m$. a' is added to set $\mathcal{M}$

3. Update B and C if $\mathcal{A}_n \neq \{\emptyset\}$:

   (a) Create a new *NAND* node B' with inputs $\mathcal{A}_n$. If B is not *don't touch*, B is replaced by B'

   (b) If C is *don't touch*, duplicate C creating a new node C'. Substitute B fanin of C' with B'. C' is a root of the non-optimized tree

   (c) If C is not *don't touch* but B is *don't touch*, substitute B fanin of C with B'. C is a root of the non-optimized tree

   (d) If $\exists\, C'$, then C' is added to set $\mathcal{M}$ else C is added to set $\mathcal{M}$

4. Update D:

   (a) $\forall d \in \mathcal{D}_s$, create a new *NAND* node d' with inputs $\{\mathcal{M} \setminus \{C, C'\}\}$ and replace d with d'

   (b) $\forall d \in \mathcal{D}_m$, a new *NAND* node d' with inputs $\mathcal{M}$ is created. Then d' is inverted creating a node d'_{inv}. Finally, d'_{inv} is connected to d replacing the connection with C

5. If $\mathcal{A}_n = \{\emptyset\}$, if C is not *don't touch*, C is deleted

6. If $\mathcal{A}_n = \{\emptyset\}$, if B is not *don't touch*, B is deleted

When we have to apply a *NAND-XOR* logic transformation, we have to deal with more complex operations. We saw some properties of *XOR* in the previous section and we have to generalize and easily implement them. In particular, we focused on how a *NAND-XOR* can be transformed into a *AND-XOR* and how we can manage the inverters to be able to apply the distributive property. An inverter must be moved if the number of A same critical plus a possible inverter after the *XOR* is odd. This is simply using the parity property to see if the inverters can be simplified. If the value is odd, or the inverter is moved to A non-critical signals, or an input at constant 1 is added to the *XOR*. Moreover, an inverter might be added also at D level. In fact, the second possible section, *XOR-NAND-XOR*, can be rewritten as *XOR-AND-XOR-INV*. This inverter can be simplified if the number of *NANDs* and inverters inserted at its input after the transformation is odd.

Transformation for *NAND-XOR* pull factor type:

1. If $\exists invx$ and B_{inv} is *don't touch*, B_{inv} is removed from the network. It is done because the inverter will be moved to another operator

2. If $\mathcal{A}_n = \{\emptyset\}$, the flag t_a is set

3. If $|\mathcal{A}_s| + invx$ is *odd*, an inverter must be distributed. Save this inside a flag $cnst_1$.

4. If $|\mathcal{A}_s| + |\mathcal{A}_m| + (t_a \wedge cnst_1) + \neg t_a + 1$ is *odd*, an inverter must be inserted to $\mathcal{D}_s$ outputs. Set the flag inv_d

5. Define set $\mathcal{N} = \{fanin(C) \setminus B\}$

6. Update A:

   (a) $\forall a \in \mathcal{A}_s$ if a is *don't touch*, create a *NAND* node a' with inputs $\{fanin(a) \cup \mathcal{N}\}$. C side inputs are distributed to $\mathcal{A}_s$. Then, node a' is added to set $\mathcal{M}$

(b) $\forall a \in \mathcal{A}_s$ if a is not *don't touch*, replace node a by a *NAND* with inputs $\{fanin(a) \cup \mathcal{N}\}$. C side inputs are distributed to $\mathcal{A}_s$. Then, node a' is added to set $\mathcal{M}$

(c) $\forall a \in \mathcal{A}_m$, create a *NAND* node a' with inputs $\{\mathcal{N} \cup a\}$. C side inputs are distributed to $\mathcal{A}_m$. Then, node a' is added to set $\mathcal{M}$

(d) If $cnst_1$ and $\mathcal{A}_n = \{\emptyset\}$ (t_a), create a new *NAND* node with inputs $\mathcal{N}$ and add it to set $\mathcal{M}$. Here a new input, at 1, is connected to the *XOR* and the side signals of C are distributed over it (previous section, *example 2(a)*).

7. Update B, C if $\mathcal{A}_n \neq \{\emptyset\}$:

(a) If $cnst_1$, create a new *XNOR* node B' with inputs $\mathcal{A}_n$. If B is not *don't touch*, B is replaced by B'

(b) If $\neg cnst_1$, create a new *XOR* node B' with inputs $\mathcal{A}_n$. If B is not *don't touch*, B is replaced by B'

(c) If C is *don't touch*, duplicate C creating a new node C'. If $invx$, substitute B_{inv} fanin of C' with B', else, substitute B fanin of C' with B'. C' is a root of a non-optimized tree

(d) If C is not *don't touch* but B is *don't touch*, if B_{inv} is *don't touch* substitute B_{inv} fanin of C with B', else, substitute B fanin of C with B'. C' is a root of a non-optimized tree

(e) If $\exists\, C'$, then C' is added to set $\mathcal{M}$ else C is added to set $\mathcal{M}$

8. Update D:

(a) If inv_d, $\forall d \in \mathcal{D}_s$, create a new *XNOR* node d' with inputs $\{\mathcal{M} \setminus \{C, C'\}\}$ and replace d with d'

(b) If $\neg inv_d$, $\forall d \in \mathcal{D}_s$, create a new *XOR* node d' with inputs $\{\mathcal{M} \setminus \{C, C'\}\}$ and replace d with d'

(c) If inv_d, $\forall d \in \mathcal{D}_m$, a new *XNOR* node d' with inputs $\mathcal{M}$ is created. Then d' is connected to d replacing the connection with C

(d) If $\neg inv_d$, $\forall d \in \mathcal{D}_m$, a new *XOR* node d' with inputs $\mathcal{M}$ is created. Then d' is connected to d replacing the connection with C

9. If $\mathcal{A}_n = \{\emptyset\}$, if C is not *don't touch*, C is deleted

10. If $\mathcal{A}_n = \{\emptyset\}$, if B is not *don't touch*, B is deleted

An important improvement of this move is in the *NAND-XOR* case. A network with logic blocked between *XOR*s is difficult to rearrange in a better way using normal algebraic methods. This move can remove stuck logic from *XOR*s so that further optimizations can be activated.

2.5.2 Examples

Let's try to analyze some examples of Pull-Factor applications. In the first figure 2.11, a *NAND-NAND* structure with two critical signals a and c is shown.

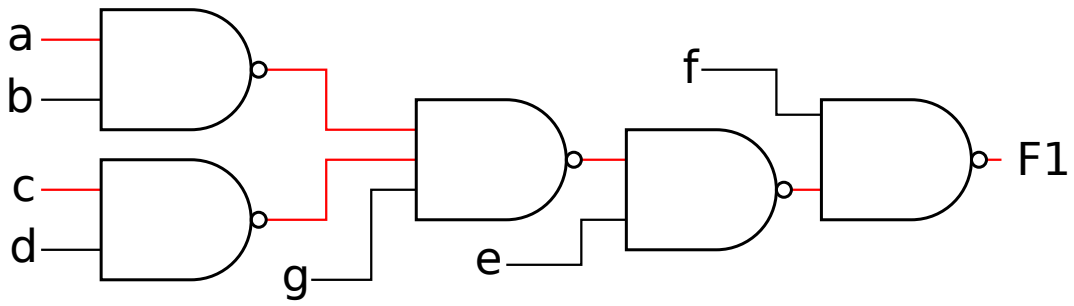

Figure 2.11. *NAND-NAND* network

The circuit in the picture can be rewritten in an *AND-OR* structure as follows:

$$A_1 = a \wedge b$$

$$A_2 = c \wedge d$$

$$B = A_1 \vee A_2 \vee \bar{g}$$

$$C = B \wedge e$$

$$D = C \vee \bar{f}$$

From this structure, is clear that C can be distributed over B trying to gain logic levels. We can so retreive the sets: The retreived sets are:

$$\mathcal{A} = \{\mathcal{A}_s = \{A_1, A_2\}, \mathcal{A}_m = \{\emptyset\}, \mathcal{A}_n = \{g\}\}$$

$$\mathcal{B} = \{B\}$$

$$\mathcal{C} = \{C\}$$

$$\mathcal{D} = \{\mathcal{D}_s = \{D\}, \mathcal{D}_m = \{\emptyset\}, \mathcal{D}_n = \{\emptyset\}\}$$

The nodes are not *don't touch*. The set of signals to be distributed back is $\mathcal{N} = \{e\}$. A_1 and A_2 are simply updated with an additional input e. For $\mathcal{A}_n = \{g\}$, the previous B gate, since it would remain with only one input, it is simplified as an inverter. Then, D node can be update with the additional inputs A_1 and A_2. In picture 2.12, we can see the final optimized structure. The logic leves of the critical paths has been reduced.

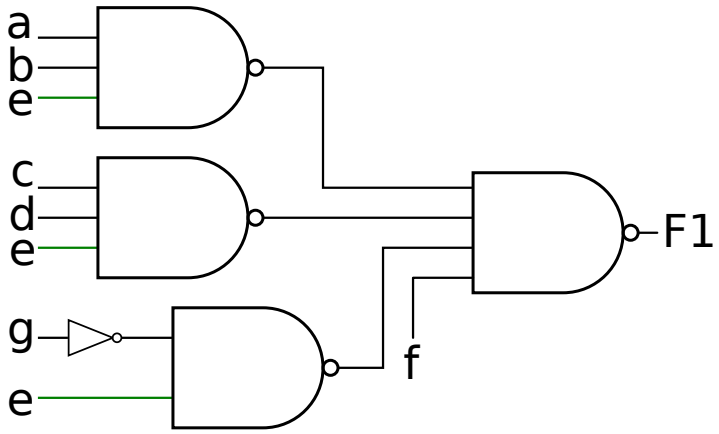

Figure 2.12. *NAND-NAND* network optimized

The second example, in picture 2.13, instead, explains the *NAND-XOR* distributivity. Signals a and b are critical. The circuit can be represented by the following equations:

$$A = \overline{a \wedge b}$$

$$B = A \oplus g_1 \oplus g_2$$

$$C = \overline{B \wedge e}$$

$$D = C \oplus f$$

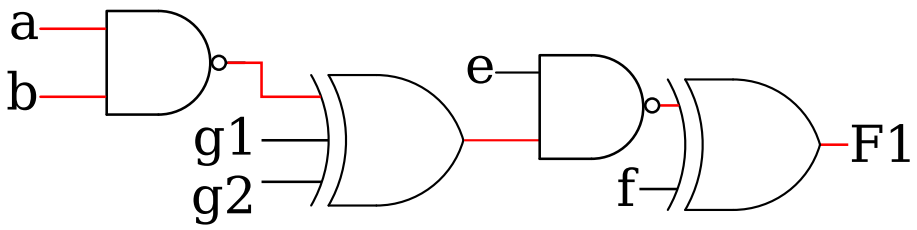

Figure 2.13. *NAND-XOR* network

So we can extract the sets:

$$\mathcal{A} = \{\mathcal{A}_s = \{A\}, \mathcal{A}_m = \{\emptyset\}, \mathcal{A}_n = \{g_1, g_2\}\}$$

$$\mathcal{B} = \{B\}$$

$$\mathcal{C} = \{C\}$$

$$\mathcal{D} = \{\mathcal{D}_s = \{D\}, \mathcal{D}_m = \{\emptyset\}, \mathcal{D}_n = \{\emptyset\}\}$$

All the nodes are not *don't touch* and not t_a. To start, we check if we need to distribute some inverters over the *XOR* gates:

- $|\mathcal{A}_s| + invx = 1$ is *odd*. An inverter (flag $cnst_1$) must be distributed over B
- $|\mathcal{A}_s| + |\mathcal{A}_m| + (t_a \wedge cnst_1) + \neg t_a + 1 = 3$ is *odd*. An inverter must be added to D

Luckily, since $\mathcal{A}_n \neq \{\emptyset\}$, the inverter $cnst_1$ can be distributed to the B side non-critical inputs, using the parity property of the *XOR*s, without adding a constant input at 1. The inverter, over D, transforms the *XOR* to a *XNOR*. The updated equations will be the following:

$$A = a \wedge b$$

$$B = A \oplus (g_1 \overline{\oplus} g_2)$$

$$C = B \wedge e$$

$$D = C \overline{\oplus} f$$

In this configuration, signal e can be distributed over B to A and $(g_1 \overline{\oplus} g_2)$. Connecting and distributing e we obtain the following structure:

$$A = a \wedge b \wedge e$$

$$B = g_1 \overline{\oplus} g_2$$

$$C = B \wedge e$$

$$D = A \oplus C \overline{\oplus} f$$

Then, we can restore the inverters over the *AND*s to re-establish the initial configuration. The final optimized circuit is shown in picture 2.14. In this example, it is clear how critical signals are pushed forward in the logic using distributive property, improving the delay.

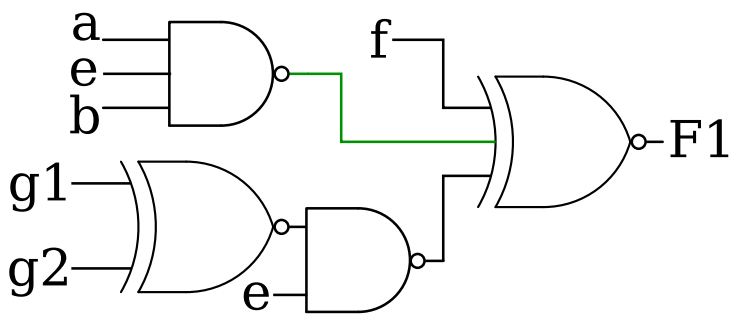

Figure 2.14. *NAND-XOR* network optimized

2.6 Implementation

The pull moves apply a simple logic transformation that can be used in small networks. A combinational part of a design contains usually thousands and thousands of gates. These moves are inserted inside an optimization engine whose objective is to extract small critical sections of a design hierarchy, apply some logic transformations, and insert them back in the design. These moves work on an already mapped design.
The main execution sequence is the following:

1. Compute the slack on the design
2. Select critical gates for each hierarchy based on the slack
3. For all the critical gates:
   (a) Extract a small cone of logic
   (b) Apply the moves to the cone
   (c) Apply some further simple elaboration to prepare the network to the mapper
   (d) Re-map the cone
   (e) Compute the cost per move based on a delay/area trade-off
   (f) Apply the best move
4. Go back to point 1

Since the two pull moves work on a different specific property, and since it is impossible to implement both of them on the same network of size 4, they are executed in parallel. All the moves are competing to be applied.
The Pull moves can be inserted in three parts during the optimization flow:

- On a mapped but not placed design. The engine works on the worst critical paths

- On a mapped and placed design. The engine works on the worst critical paths
- On a mapped and placed design. The engine works on the paths with negative slack to improve the total negative slack

In the next chapter, we will discuss the results of these implementations together with other existing algorithms.

When we presented the pull moves, we did not explicitly explain how a node or signal is defined as critical. We said only that a signal is considered critical if its slack is negative and within a threshold from the worst negative slack. In the implementation, the threshold is described by a constant factor f, which is obtained experimentally. It is used to multiply the *NAND2* standard delay of the design to obtain a delay value. So, a signal is considered critical if its slack is negative and lower than $wns + f \times d_{nand2}$. Previously, we saw how some signals may result slowed down by the pull transformations. It is important to set a good threshold value to mark as critical also those signals whenever they would be surely slowed down.

Two slightly different implementation versions have been developed. The Pull moves are effective on a very specific circuit topology. If an associative or distributive property is not found, the move cannot be applied. Moreover, as we said, the delay improvement is not assured by the logic transformation. So we do not expect the pull moves to be applied as often as generic algebraic methods. For this reason, we might try to be more pedantic and to early bail out the pull moves, before re-mapping, whenever an improvement is not detected, using a logical effort based delay model. In the new alternative, the inner loop, for pull moves, works as follows:

1. Extract a small cone of logic
2. Apply the moves to the cone
3. If a move is not applied, mark it as unsuccessful

4. If the move is applied, cost it. If the worst negative slack is not improved, mark it as unsuccessful and return
5. If successful, apply some further simple elaboration to prepare the cone to the mapper
6. Re-map the cone
7. Compute the cost for a delay/area tradeoff
8. Apply the best move

This modification can be seen as an improved version for runtime but, actually, that filter could modify the resulting QoR. A good structure of the cone, even if the network is not significantly improved after a pull move, can still be found during the pre-mapping optimization and the re-mapping. But, furthermore, it is likely that, if a change is not significantly successful, it will not carry over the change to the following stages of optimization as it may be easily rewritten by another logic transformation.

Chapter 3

Pull Results

3.1 Introduction

In this chapter, we will discuss the results of the pull moves inserted inside the EDA tool Fusion Compiler. The used benchmarking system is the official internal one in Synopsys. The published information are generalized and simplified due to confidentiality reasons. We will analyze the average QoR in three points during the whole flow, in order:

- **PC**: before initial physical optimization
- **DC**: after initial physical optimization
- **IC**: after final physical optimization

The pull moves are executed in the first part of the flow, before the initial physical optimization. From the *PC* results, we expect more the influence of the pull moves as the data is extracted a few steps after. But it will be interesting to see if the changes will add a positive influence over the following steps of the optimization flow.
It is particularly important to understand the contribution of each move. For this reason, we are going to present two different reports. The first one will show

and compare the success rate of the pull moves. Since moves compete to be implemented, it is necessary to analyze how many times they are effectively implemented on a design. This selection parameter will help to understand the potential improvement that can be generated over the flow. The second report will present the QoR results from the Synopsys benchmarking system.
During the tests, we compared two different types of execution:

- **ex1**: vanilla implementation
- **ex2**: optimized version for runtime, the move is costed also before mapping

As a reminder, we will apply the pull moves in three stages in the flow with different optimization strategies:

- **PC1**: On a mapped non-placed design for *WNS*
- **PC2**: On a mapped placed design for *WNS*
- **PC3**: On a mapped placed design for *TNS*

3.2 Success rate

Study the success rate of a move is important to understand how much impact a logic transformation may have on a final design and if it is worth it to use it. The data has been collected from several combinational designs[1] at *PC1* stage. The results in *PC2* and *PC3* are comparable to *PC1*. Different metrics are taken into account:

- **NumApp**: the overall number of applications (executions) of the move over the designs

[1] **ex1** results are extracted from an average over 42 designs while **ex2** on 63. Anyway, the results can be compared since we are interested only at the ratios that are stable among the test cases. That is why **ex2** has a greater number of applications than **ex1**. The 42 designs are included in the 63.

- **NumSucc**: the overall number of successful executions over the designs. A move is considered successful if the delay improves
- **NumFail**: the overall number of not successful executions over the designs
- **FailRate**: percentage of failed executions: $NumFail/NumApp$
- **Impl**: the overall number of times the move is implemented over the designs
- **Elapsed**: overall elapsed time executing the move

In addition to Pull-Level and Pull-Factor, another existing move, which uses algebraic methods, has been inserted in the results for a comparison. Note that other moves are applied in the engine.

The results for *ex1* are shown in the table 3.1. The fail rate is comparable to a general algebraic method while the number of implementations is only the 30%. As we said before, we expect the pull moves to be applied fewer times than an algebraic transformation. So why the failure rate is low? It is mainly due to the pre-mapper optimization and the mapper which are always executed. We will refer to this behavior with the name of **false success**. In fact, even if a move is not successful, the logic can be slightly improved by the mapper. This behavior can be noticed by the low number of implementations with respect to the success rate. Moreover, if *false successful* moves are implemented, it is likely that the transformation they propose will not be significant over the flow.

Move	NumApp	NumSucc	NumFail	FailRate	Impl	Elapsed
PullFactor	68824	29337	39487	0.574	384	1039.25
PullLevel	68826	30791	38035	0.553	342	721.27
Algebraic	68823	31980	36843	0.535	996	1289.69

Table 3.1. Success rate **ex1**

Table 3.2, instead, contains the success rate for *ex2*. Here the failure rate is very

high as the number of failures. Generally, the filter we have inserted is excluding about the 85% of the moves that before were considered successful. The *false success* happens often. If we compare also the number of implementations over the number of success, we notice that, from a 1.1% for *ex1*, it grows up to 2.1% for *ex2*. As a reference, for the algebraic method, it is 3.1%. Furthermore, the elapsed time is considerably decreased. The final number of implementations is about the 11% of the algebraic method. It is a good result considered the specific field of application of the moves. Moreover, this 11% should be guaranteed as a good transformation.

Move	**NumApp**	**NumSucc**	**NumFail**	**FailRate**	**Impl**	**Elapsed**
PullFactor	146635	19284	127351	0.868	417	234.08
PullLevel	146642	12152	134490	0.917	345	117.71
Algebraic	146628	77323	69305	0.473	3367	1289.61

Table 3.2. Success rate **ex2**

Using a filter, may *ex2* exclude some good elaborations? To answer this question, we have to compare the QoR results.

3.3 QoR

The QoR has been extracted from a run over 52 industrial designs. We will show, for each data, the mean value among the designs. For this purpose we select the following data, extracted from the relevant stage *PC, DC*, and *IC*:

- **WNS**: worst negative slack
- **TNS**: total negative slack
- **Area**: total area, combinational plus non-combinational
- **Tot Power**: total power, dynamic plus leakage
- **Runtime**: compile execution time

- **Buf Area**: area occupied by buffers

There is much more information that could be included, like wire length, congestion or leakage power. But, to give an intuitive idea of the possible improvement that can be obtained with the pull moves, we extract only the most relevant information for our purposes. Moreover, all the other fields, not included in the following results, do not contain statistically relevant information.

Another important aspect is the statistical significance which consider the standard deviation of a result. In fact, some results can have a good average improvement but an unstable distribution of values among the different designs. To represent the data consistency, i.e. the statistical significance, the values will be colored in green or red. A dark tonality represents a strong statistical relevance while, a light tonality, represents a weak one. However, not colored values are still valuable because they may describe a trend. Moreover, DC and IC results depend on the optimization trajectory in the downstream flow. They can look different from one build to another, especially during the development period when the overall code is not frozen. To validate an algorithm, we usually rely on the best results found and, for production, we tune the downstream flow so that we can leverage it and consistently show benefit.

In table 3.3, the delay results are shown. If we focus our attention on *PC TNS*, we can notice that the pull moves are actually improving the general delay of the network. But this value tends to change often among the different builts. In fact, usually, the *PC* delay is even better. The *PC WNS*, instead, is not really changed because some endpoints in the design may not be significantly touched by these moves. We expected, also, *ex1* to be slightly better than *ex2* since it is providing more alternatives. To understand if these transformations are significant, we analyze the *DC* results. Here we see that the results are inverted and this trend continues to *IC*. This can suggest that the *false successful* moves are not correlated with significant results in the flow. Moreover, the network structure of *ex2*, without *false successful* moves, seems to have a higher probability of improving the results

further in the flow. The pull moves suggest, also, an optimization effect on the designs. It is possible that the design may be more sensitive to further transformations over the flow. It is particularly true for pull factor which frees blocked logic between *XOR* gates.

Flow	DC WNS	DC TNS	PC WNS	PC TNS	IC WNS	IC TNS
ex1	-0.17%	-3.10%	-0.19%	-2.08%	-0.57%	-10.47%
ex2	-1.12%	-6.42%	+0.13%	-0.50%	-0.71%	-13.72%

Table 3.3. Delay QoR

In table 3.4, other relevant results are shown. Even if, the *PC Area* is well conditioned by the pull moves, they do not have much impact on the *DC Area*. One important result is the big enhancement in run time of *ex2* over *ex1*. The total power is also moderately improved. The buffer area is instead worsened. It is unclear if this is a direct effect of the node duplication technique used by the pull moves or an indirect effect caused by other moves in the flow. Also, we cannot have a *PC* buffer information as the circuits are not yet buffered.

Flow	DC Area	PC Area	DC Tot Power	Runtime	DC Buf Area
ex1	+0.09%	-0.01%	-0.29%	+3.36%	+1.77%
ex2	+0.09%	-0.04%	-0.17%	+0.52%	+1.44%

Table 3.4. Area, power and runtime QoR

From the results, we notice that actually, *ex2* is generally better than *ex1* and that the pull moves increase the general quality of results.

Chapter 4

Global Flow

4.1 Introduction

In the previous chapters, we presented two methods that work on a few gates and that are applied several times on a design. In this chapter, we will introduce a move that restructures a whole combinational hierarchy, composed of thousands of gates. Global Flow is a rewiring algorithm described in the paper "Global Flow Optimization in Automatic Logic Design" [4]. The idea is to change the gates' set of connections to an equivalent one which optimizes the circuit. It uses techniques of data flow analysis to summarize a circuit and to identify a class of circuits that are equivalent. Global Flow reduces the problem of finding a connection set to a cut in an associated summarized graph. In our work, we developed an extension and generalization of the original algorithm. We want to extend the base idea to be able to implement it on a mapped netlist for a delay improvement. The differences from the original version will be explained in details over the next sections.

4.2 Global Flow Analysis

In this section, we will take some time to introduce the basic concepts to comprehend Global Flow. Global Flow's original goal was to minimize a circuit by

changing signal connections to another set of gates and by removing redundancies. This is realize by obtaining global controlling information of a circuit and its wires. In the following paragraphs, we will introduce some definitions needed to define the algorithm. Most of the definitions are in accordance with [4].

Combinational circuits are modelled as direct acyclic graphs (DAG's). Each node is represented by a gate and each edge is a wire connection. Primary Inputs and Primary Outputs are modelled also as special nodes: PI node for the former and PO node for the latter.

"If a node i is a direct input of a node j, we will use the notation $i \rightarrow j$. If, instead, exist a path from node i to node j then ***j* is reachable from *i***, written as $i \xrightarrow{*} j$." The set of input nodes of a node i will be referred as *fanins(i)* while the set of output nodes as *fanouts(i)*.

"Two combinational circuits $\mathscr{C}$ and $\mathscr{C}'$ are equivalent if the corresponding outputs of the two circuits represent the same function."

If x is a node in $\mathscr{C}$ and S is a set of nodes in $\mathscr{C}$ so that every path from x's output to the primary outputs contains a node in S, we say that ***x* is blocked by *S***.

The summary information is described by a composition of the **controlling sets** of a node $x \in \mathscr{C}$ defined as

$$C_{ij}^{\mathscr{C}}(x) \equiv \{s : \text{ if } x = i \text{ then } s = j\}, \quad \text{for } i,j \in \{0,1\} \text{ and } x \xrightarrow{*} s. \tag{4.1}$$

In few words, the set $C_{ij}^{\mathscr{C}}(x)$ contains all the nodes s reachable from x for which a value i on node x implies a value j on node s.

The following lemmas, from [4], will define when a connection to a gate can be

added or removed without changing the global functionality of a circuit. For simplicity of explanation, the nodes are supposed to be NAND's. They will be next generalized.

"*Lemma 1:* if $i \in C_{01}(s)$ then s may be connected as an input to node i without changing the function."

Proof: in a NAND gate any input at 0 can force a 1 at the output. So, if s can control the output of i to 1, it means that it can control at least one of the inputs to 0. So, connecting another input s, that when it is 0 the gate is already forced at 1, and when it is a 1 it doesn't influence the output, doesn't change the gate functionality. This is generally true, for simple gates[1], when a node $x \in$ *fanins(i)* is included in $C_{nn}(s)$ and the node i is included both in $C_{nm}(s)$ and $C_{nm}(x)$. So s can be rewired to i without changing the global functionality of the circuit.

"*Lemma 2:* if s is connected to a node i and i *is blocked by fanouts(s)*, then s can be disconnected from i."

Proof: if s is equal to 1, it is obvious that the circuit's functionality remains the same. In fact a connection removal, in a NAND gate, is equivalent to a replace of the connection with a constant 1 since it is a non-controlling value. If instead s is at 0, the difference in functionality will be extinguished, among all the paths, at *fanouts(s)*, before reaching the primary outputs.

These lemmas introduce a rewiring and a redundancy removal strategy. But this approach, as described, is not very scalable since it can be applied only on simple gates. The goal is to extend these lemmas on generic (*AOI21*, multiplexer, etc.) and mapped gates, described by a *Sum of Products*[2], in order to apply Global Flow

[1]With the term *simple gates*, we refer to general n-input NAND, NOR, AND and OR gates

[2]A *SoP* is a function expressed by a sum of terms (products of literals). For example: $F =$

on mapped circuits.

We can define two types of implications. A **weak** implication occurs when a node x cannot be controlled, to a value v, by a single value assignment to a fanin $i \in$ *fanins(x)*, but by only a combination of assignements to multiple fanins. In this case v is a **weak**, or a non-controlling, value for x. This condition happens, for instance, in an *OR2* gate with inputs a, b, for $v = 0$ where both a and b must be at 0. A **strong** implication happens when a single value assignment to $i \in$ *fanins(x)*, can control x to value v. Value v is a **strong**, or a controlling, value for x. For instance, an *OR2* gate, with inputs a, b, for $a = 1 \vee b = 1$ assumes a strong value 1. Global Flow [4] restricts the rewiring to only **strong** controlled gates, but, actually, the idea can be extended also to the **weak** controlling type. The following corollaries will extend the previous lemmas.

Corollary 3: if $i \in C_{nm}(s)$, s can be connected as an input of i, modifying the *SOP* of the node i as the following:

- Inserting an additional literal in *OR* with the *SOP* when $m = 1$, with positive phase if $n = 1$, or a negative otherwise.

- Inserting additional literals in *AND* with all the cubes of the *SOP* when $m = 0$, with positive phase if $n = 0$, or a negative otherwise.

Proof: The corollary is derived from *lemma 1* and easily verifiable. We will present only two examples in order to avoid doubts. If applied to a AO21 gate, described by the *SOP* $i = (a \wedge b) \vee c$, where $i \in C_{01}(s)$, s can be connected to the *SOP* as $i = (a \wedge b) \vee c \vee \neg s$ without changing its functionality. If, instead, $i \in C_{00}(s)$, s can be rewired as $i = (a \wedge b \wedge s) \vee (c \wedge s)$.

Corollary 4: If s is connected to a node i, *i is blocked by* a set of nodes $B \in$ *fanouts(s)* and $B \in \{C_{n0}(s) \cup C_{n1}(s)\}$, then the connection between s and i can be

$(a \wedge b) \vee (c \wedge d)$

substitute by a constant signal with value $\neg n$.

Proof: If s assumes the value $\neg n$ the functionality of the circuit doesn't change. If instead s has the value n the difference will be extinguished at nodes in B since they are controlled by s at n.

Corollary 3 and 4 are the base for applying the rewiring algorithm. But before describing it entirely, a new definition must be introduced.
For a node s and a value n, we define the **frontier of** $\boldsymbol{s}$ $\mathscr{F}^{\mathscr{C}}(s,n)$ as the set of nodes i such that:

- $i \in \{C_{n0}(s) \cup C_{n1}(s)\}$
- $\exists$ *path* $\mathscr{P}_i : i \to j_1 \to j_2 \to \cdots \to PO$, $j_l \notin \{C_{n0}(s) \cup C_{n1}(s)\}$

We can also define $\mathscr{F}^{\mathscr{C}}(s,n)$ as the set of nodes $i \in \{C_{n0}(s) \cup C_{n1}(s)\}$ which are *not blocked by* the same set $\{C_{n0}(s) \cup C_{n1}(s)\}$.

Let's define also the immediate fanout set $\mathscr{I}^{\mathscr{C}}(s,n)$ as the set of node i:

- $i \in \{C_{n0}(s) \cup C_{n1}(s)\}$
- $i \in fanout(s)$

$\mathscr{I}^{\mathscr{C}}(s,n)$ contains all the controlled nodes which are at the fanout of s. These nodes *are blocked by* $\mathscr{F}^{\mathscr{C}}(s,n)$, unless for the ones which are contained in both sets. For instance, from the corollaries 3 and 4, we could connect s to the set $\{\mathscr{F}^{\mathscr{C}}(s,n) \setminus \mathscr{I}^{\mathscr{C}}(s,n)\}$ and substitute s with constant $\neg n$ at $\{\mathscr{I}^{\mathscr{C}}(s,n) \setminus \mathscr{F}^{\mathscr{C}}(s,n)\}$ without changing the functionality of circuit $\mathscr{C}$.

4.3 Applications

In this section, we are going to understand how we can use this technique to improve a circuit. We have defined some conditions that allow a signal to be rewired to

another set of gates. In particular, to alter the connections of s, we must define a set of node $\mathcal{N}_s$ so that the following conditions hold:

- $\mathcal{N}_s \in \{C_{n0}(s) \cup C_{n1}(s)\}$
- Set $\{\mathcal{I}^{\mathscr{C}}(s,n) \setminus \mathcal{N}_s\}$ *is blocked by* $\mathcal{N}_s$

The set of nodes $\mathcal{N}_s$ can be selected as a new connection set for signal s which can be substituted for $\mathcal{I}^{\mathscr{C}}(s,n)$. It can be used to implement different types of optimizations. For example, a minimal set of nodes decreases the number of connections or, another set, more forward in the logic, can be selected to improve the delay. The problem of finding the best set of nodes that satisfies an optimization goal is *NP*-hard[3]. But, from the conditions we have defined, [4] presents a method to reduce this problem using an efficient approximation by means of the *min-cut* algorithm.

The information included inside the sets $C_{n0}(s)$, $C_{n1}(s)$, $\mathscr{F}^{\mathscr{C}}(s,n)$, and $\mathcal{I}^{\mathscr{C}}(s,n)$, linked to the circuit topology, can be used to build a DAG which summarize the rewiring possibilities. A cut of this graph extracts a valid set of gates $\mathcal{N}_s$ to whom rewire.

4.3.1 Global Flow Graph Example

Before explaining the Global flow graph created in our implementation, let's analyze with an example the process proposed in [4] trying to understand how it works and some possible improvements.
Let's use the *NAND* decompose network in figure 4.1 as an example. We consider the fanout cone of logic of signal s. Signal s is tested with the strong implication

[3]This follows since the satisfiability or tautology problem can be directly reduced to this problem. It is unlikely that the problem is *NP* since it requires to solve an equivalence problem between two sets of connections

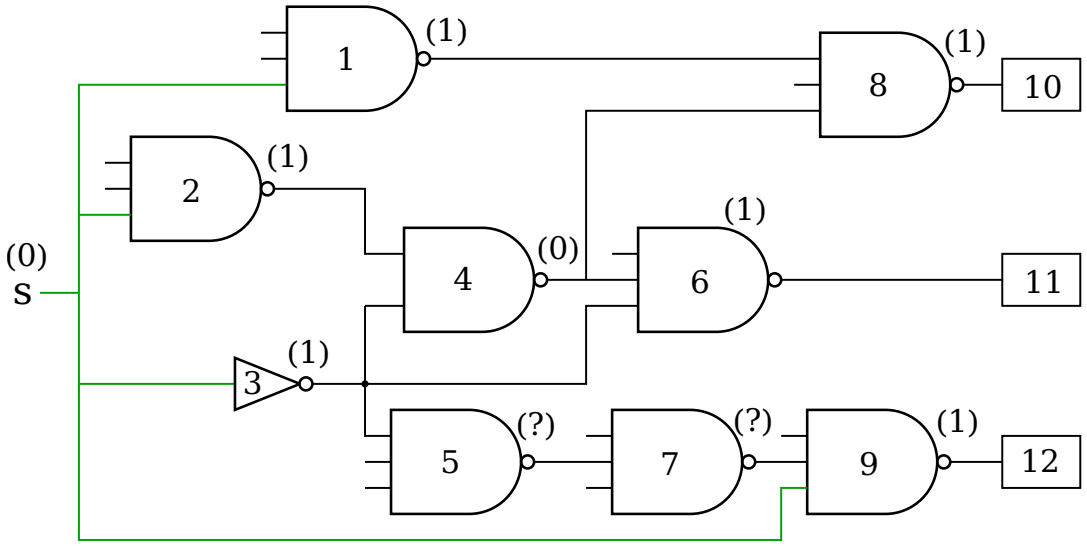

Figure 4.1. *NAND* decomposed circuit for rewiring example

value for *NAND* gates 0. This value is propagated through the network using controlling conditions. As a reminder, if an input to a *NAND* gate is 0, the gate is forced at 1, if all the inputs are at 1, the gate is forced to 0. In picture 4.1, we can notice that, for every gate, an implication value is calculated. Value can be {0,1, ?}. "*?*" means that the controlling value cannot be implied by the input assignement. From the value propagation we can extract the set $C_{01}(s)$ (strong implication) that will be used to build the Global Flow graph. We can also extract the set $\mathscr{F}^{\mathscr{C}}(s,0)$ with nodes $i \in C_{01}(s)$. The two sets contains the following nodes:

- $C_{01}(s) = \{1,2,3,6,8,9\}$
- $\mathscr{F}^{\mathscr{C}}(s,0) = \{8,6,9\}$

From this two sets, we can create a node inside the Global Flow graph as following:

- For each $i \in C_{01}(s)$ a node is created
- A *source* node is created. It represents signal s as the source of the connections
- A *sink* node is created. It represent the network beyond the frontier for s. Only frontier nodes are connected to the sink

Then the edges are assigned using the following strategy:

- $\forall n \in \mathscr{F}^{\mathscr{C}}(s,0)$, create an edge from n to the sink
- Create $SET = \{\mathscr{F}^{\mathscr{C}}(s,0)\}$. Proceeding in a reverse breadth first order, extract a node j from SET and find a node $k \in fanin(j)$ so that $k \in C_{00}(s)$:
  - If k exists, $\forall i \in fanin(k)$, add an edge (i,j), add i to SET. Then remove j from SET
  - If a node k does not exists, add an edge from source to j

Let's remember that the graph is built so that every cut is a valid reconnection set. Then, a weight is assigned to each node. In fact, we can use the min-cut algorithm to extract a minimal cut from the graph. For example, let's use a strategy that will help us to minimize the number of connections of s. In this case, is obvious that we have to select a cut with a minimum number of nodes. So, we assign a unit weight for each node. For the source and the sink nodes, an infinite value is set because they are not a valid cut of the graph (source is signal s, sink represents the network beyond the frontier). The generated graph is shown in picture 4.2. All the possible cuts of the graph define a valid set of blocking nodes to rewire. The cut in red {8,6,9} selects a possible min-cut of the graph. The other possible min-cut is {2,3,9}.

Following lemma 1, s may be directly connected to nodes 8 and 6 (s was already connected to 9) without changing the functionality of the circuit. Also, according to lemma 2, since nodes $\{1, 2,3\}$ are blocked by the cut {8,6,9}, the connection with s is redundant and it can be removed. This will simplify node 3 and remove it from the network. The final results is shown in figure 4.3. We can notice how the number of connections has been reduced from 4 to 3 still maintaining the same functionality at the primary outputs of the circuit.
By applying this method over the whole circuit, it will contribute to the decrease in the number of connections and the overall area. But, actually, we can notice some limitation of this approach:

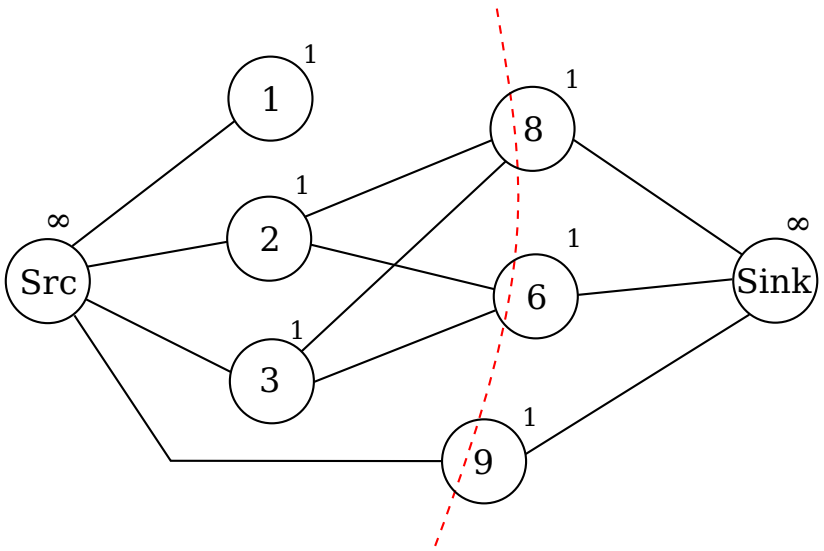

Figure 4.2. Extracted Global Flow graph for *s* example

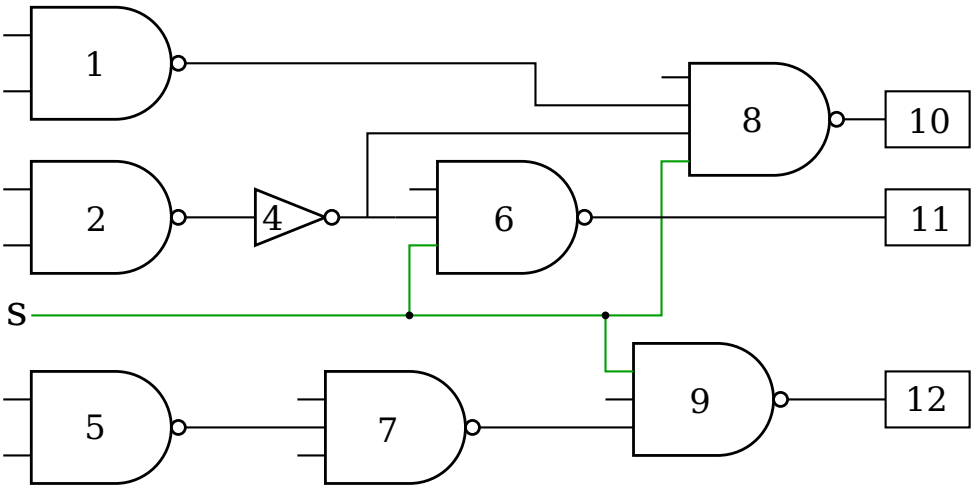

Figure 4.3. Final rewiring in circuit for *s* example

- In this technique, especially for area, if we look for rewiring conditions only at the fanout cone of signals, we could miss some opportunities. So would be better to generate a dual global flow graph [6] both for the fanin (based on observability conditions) and fanout cone

- It cannot be used for mapped gates without an extension (corollary 3, 4)

- The global flow graph excludes rewiring to *weak* controlled nodes. This leads to a loss in optimality when generating the graph. Moreover, some edges may be redundant leading to bigger cutsets than necessary. In the following section, we will see, how this can limit some optimizations
- A more flexible approach may allow us to use global flow for other optimizations as delay or wire length

4.3.2 A new Global Flow Graph

Here we try to generalize the concept of global flow trying to solve some of the issues presented in the previous chapter. Without losing generality, our version is mostly based on delay improvement. Thus, the dual global flow approach has not been implemented as unnecessary for our goal.
From now on, we will proceed with a different global flow graph. It will include both **strong** and **weak** implications as in [5]. But the weight assignment and the edges creation strategy will be different. [4] underlines how some edges in the global flow graph may be removed leading to a better cutset. [5] proposes an approach based on implication graphs to remove these redundant edges obtaining good results. But this approach is quite costly and not trivial to generalize. Our implementation follows the circuit topology.

Signal s is tested with value n and controllability values are propagated through the network. The propagation is done also to the fanin cone of s using observability conditions. Controllability checks if an assignment to the inputs gives information at the outputs, observability checks, instead, if an assignment at the output gives information at the inputs. A back-propagation is done because some observable nodes in the fanin cone, connected somehow to the fanout cone, may reveal some *weak* controllabilities. After this process, the sets $C_{n0}(s)$, $C_{n1}(s)$, and $\mathscr{F}^{\mathscr{C}}(s,n)$ are extracted. Before explaining the method used for the graph generation, let's define another set $C'_n(s)$ which contains all the gates of $\{C_{n0}(s) \cup C_{n1}(s)\}$ which are not buffers, inverters and primary outputs. $C'_n(s)$ will be the set used for the graph

generation. Buffers, inverters, and primary outputs are excluded from the graph since they would only increase its size without adding useful information. In fact, it would make basically no difference to rewire to their predecessors instead. Also, we will use the function $gate(x)$ which returns the gate in the network linked to the flow graph node x.
The nodes and edges are created with the following strategy:

- A *sink* nodes is created
- Starting from s, call a recursive function that goes through the network in topological order and generates nodes and edges:
  - For a gate g, a node i in the global flow graph is created if $g \in C'_n(s)$
  - An edge (i, j) is created if $gate(i) \rightarrow gate(j)$ or $gate(i) \xrightarrow{*} gate(j)$ and exists a path $\mathscr{P} : gate(i) \rightarrow b_1 \rightarrow b_2 \rightarrow \cdots \rightarrow gate(j)$ where b_k is a buffer or an inverter
  - An edge $(i, sink)$ is created if $gate(i)$ is *not blocked by* $C'_n(s)$. To explain it in another way, it exists a path $\mathscr{P}' : gate(i) \rightarrow gate(j)$ or $\mathscr{P}'' : gate(i) \rightarrow b_1 \rightarrow b_2 \rightarrow \cdots \rightarrow gate(j)$ where $gate(j) \notin C'_n(s)$ and b_k is a buffer or an inverter
- Possible edges $(source, sink)$ are not included inside the graph. Non-controlled fanouts of s will remain unchanged after the rewiring

4.3.3 Rewire with a new method

The new rewiring presented with corollaries 3 and 4 will be now used on the new presented global flow graph. From now on, we will work on general networks in order to show how the new connections can be added. Let's introduce this method with an example.
In figure 4.4, a small network containing generic types of gates is presented. Signal s is tested at value 0. Then the value is propagated through the network assigning

to each gate a controllability value. Thus we can extract the various sets that will be used to build the Global Flow graph:

- $C_{00}(s) = \{4\}$
- $C_{01}(s) = \{1, 2, 3\}$
- $C'_0(s) = \{1,2,4\}$
- $\mathscr{F}^{\mathscr{C}}(s, 0) = \{4\}$

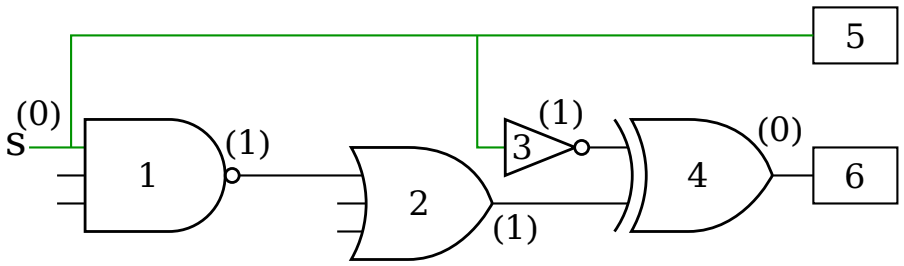

Figure 4.4. Initial rewiring in circuit for s example

Then the global flow graph is generated, in figure 4.5. It is based directly on the controlling sets and the circuit topology. Signal s is connected to its immediate fanout excluding 5 because it is a primary output. For simplicity, in this example, a unit weight assignment strategy is used as in the original method. We can notice that both *strongly* (1,2) and *weakly* (4) controlled nodes are both included in the graph while the inverter 3 has been replaced by an edge. The min-cut extract {4} as the best reconnection set.

The rewiring is based on corollary 3. Since gate $4 \in C_{00}(s)$, we can rewire s to 4 by adding a literal in *AND* with all the cubes of the *SOP* with positive phase. This is equivalent to add by factorization an *AND* level of logic with s at 4's fanout. The additional connection is shown in figure 4.6. Let's try to verify that the new circuit is equivalent to the previous one. When s was at 0, the output of gate 4 was controlled at 0. 6 was also at 0. This characteristic is not changed in the new

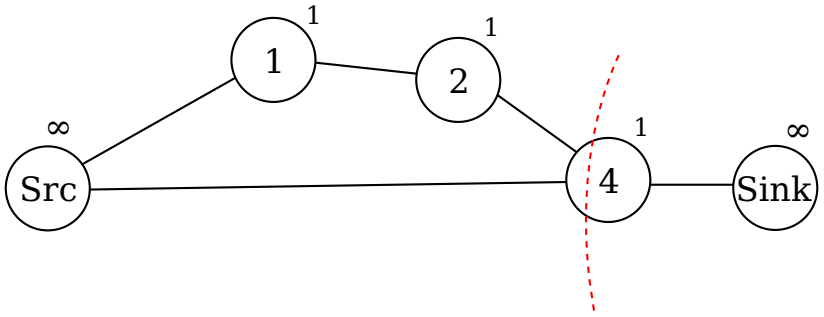

Figure 4.5. New generated global flow graph and cut-set

circuit. When s is 1, the new *AND* gate is positive sensitive[4] to 4's output. So, even in this case, the circuit has the same functionality.

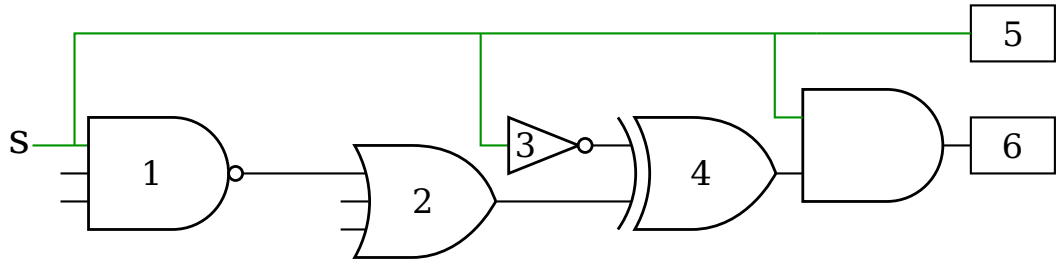

Figure 4.6. Adding new connections to node 4 using corollary 3

The controlled immediate fanouts of s, $\mathcal{I}^{\mathscr{C}}(s,0) = \{1,3\}$, is blocked by node 4. So we can apply corollary 4. The connection between s and 1 and s and 3 is subtitute by a constant 1, the opposite of the tested value. Figure 4.7 represents this elaboration. If s takes value 1, the functionality remains the same. If s takes value 0, the difference will be estinguished at the new *AND* node which is forced to 0.
The new optimized circuit is simplified with constant propagation, as shown in figure 4.8.

A circuit may be optimized also if the cutset remains the same. In fact, if a gate

[4]The output is directly dependent by the value of 4

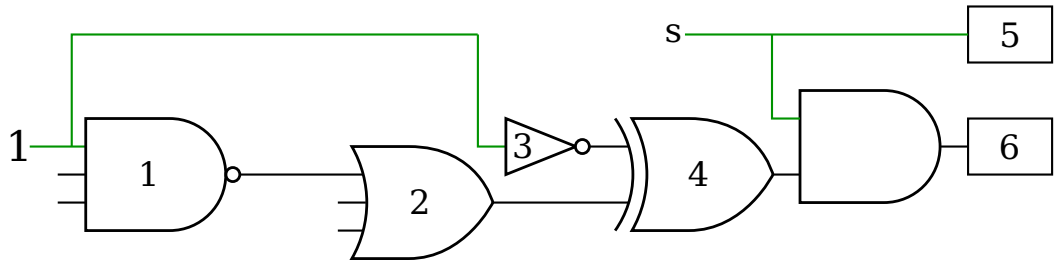

Figure 4.7. Removing redundant connections using corollary 4

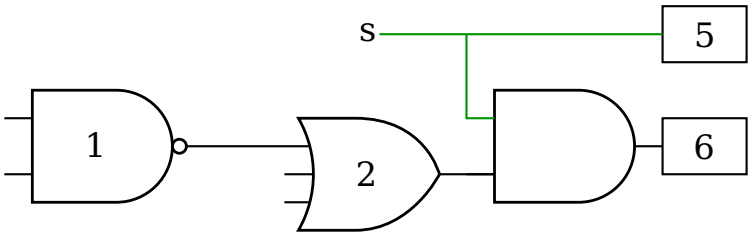

Figure 4.8. Final simplified circuit

$g \in \mathcal{I}^{\mathscr{C}}(s, n)$, it can be simplified using controllability conditions. Let's see it with an example. Figure 4.9(a) shows a sub-circuit where signal s is tested with value 1. *Logic*, in blue, represents a generic part of the circuit with two outputs, inputs of the sub-network under analysis. The tested value 1 of s, before being propagated forward, it is propagated back through the network in *Logic* with observability implications. Then for all the known nodes, a forward controllability implication is executed (as we saw in the previous example). During this process, let's suppose that also the other fanin of 1 is known to take value 1. From the picture is clear that the *XOR* gate can be controlled at 0. We proceed to generate the trivial global flow graph associated with the sub-circuit, in figure 4.9(b). The cutset is obviously gate 1.

In figure 4.10, we apply the corollary number 3. Signal s is strongly rewired with an *AND* gate and a negative phase, since $n = 1$. Now s is relly controlling the output 2 without using external conditions. This will allow us to simplify the *XOR* gate.

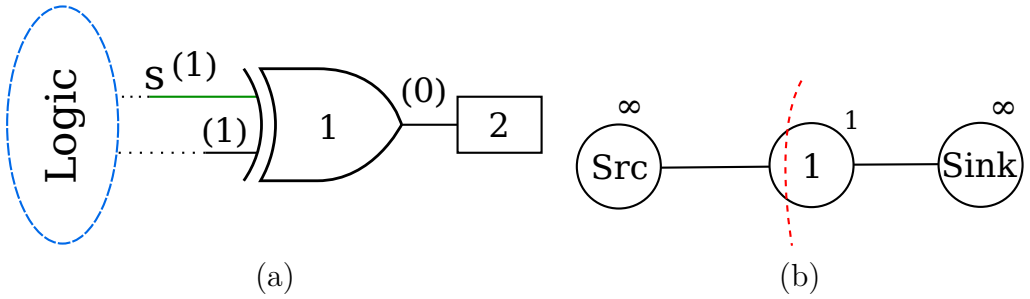

Figure 4.9. Initial circuit to optimize for signal s and its GF graph

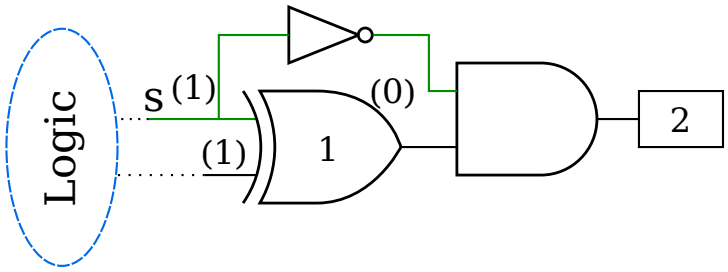

Figure 4.10. Signal s is strongly connected

In figure 4.11, corollary 4 is applied. Signal s is substituted with a constant 0 at gate 1. Since s now has a direct control to the output, the circuit maintains the same functionality.

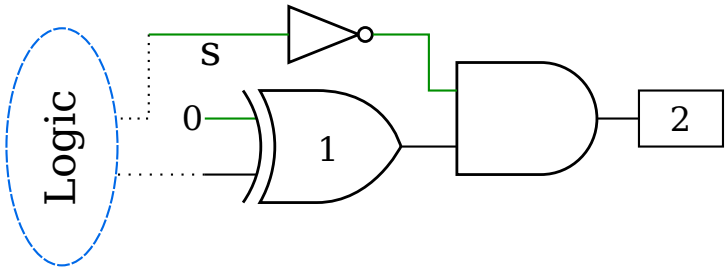

Figure 4.11. Signal s is substituted with a constant 0 at the *XOR* gate

At the end the *XOR* gate is simplified using constant propagation, in figure 4.12.

From this example we noticed how, with controllability informations, we can optimize a circuit even maintaining the same cutset. This example brings a good improvement both for delay and area since a *XOR* is substituted with an *AND*.

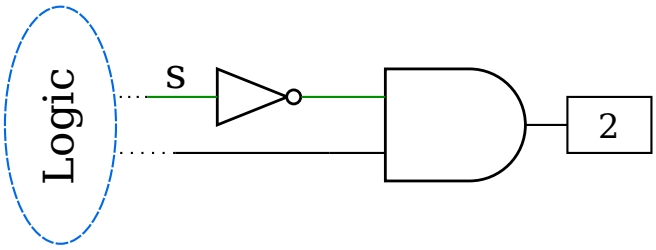

Figure 4.12. Final simplified circuit

4.4 Weights assignment

Finding a better set of connections depends on a good heuristic in assigning nodes' weights. In fact, the min-cut algorithm extracts a cut of the graphs which minimizes the sum of the nodes' weights. A new set of connections can be created to improve the different characteristics of the circuit. If the number of connections is lowered, the area decreases. If a critical signal is connected to a less critical set of gates, the delay improves. If a signal is connected to a closer set of gates, in physical placement and routing, the wire length improves. Other interesting types of applications can be found.
The goal of this research project is to investigate methods to improve the delay in a circuit. So, from now on, we will focus on how to adapt global flow for delay target.

From the previous sections, we can derive some conditions that affect the delay in a circuit:

- A signal s is critical. If s is connected to a less critical set of gates, with higher required time, the delay may improve

- Accordingly to corollary 3, connecting a signal to a gate is different based on the gate type and the controllability condition. In some cases, only a literal is added. In others, an additional gate level and an inverter must be added. We noticed how easy was to rewire in the first example, in figure 4.3, with respect to the second one, in figure 4.8. In general, rewire to *strong* controlled gate is less expensive, in terms of area and delay, than rewire to a *weak* controlled gate, exept for a node in the immediate fanout.

- We said that pushing signals forward helps the delay. We saw that also in chapter 2 with the pull moves. An idea may be to describe how many levels forward in the logic the signal may be rewired and to measure it in *NAND2* delay units. We saw in logical effort how the unit delay was based on an inverter driving another identical inverter. Here we do the same thing using a *NAND2* gate. We can define a number, calculated from the required time of every gate, that will estimate how many *NAND2* gate levels the signal would be pushed forward in a rewiring involving that gate. This measure can be used to assign the weights. Also, we can use this measure for another purpose. If a signal is pushed forward enough, it will not be critical anymore. Pushing the signal even forward, will not improve the delay. Thus, we can use this measure to define a threshold where, if a signal is pushed forward enough, we decrease the number of connections instead of the delay. This reasoning will help us for two things:

  - It helps limiting the area increase of the rewiring (delay/area tradeoff)
  - Adding a connection to a gate (or path) will increase the criticality of side inputs (or side paths). So, limiting the number of connections will decrease this side effect.

- We saw in example 3, figure 4.9, how immediate *weak* controlled fanouts can be simplified and substituted with a better gate. This new gate can improve both delay and area. So this type of rewiring must be well considered.

Grouping all these conditions can give us a general way to assign the weights.

Algorithm 1 is used to assign delay weights. $\mathcal{G}$ is the global flow graph while $\mathcal{W}_n$ is the weight of a node n. In the algorithm, some parameters are used:

- $\mathcal{L}$ is a parameter that defines after how many *NAND2* delay levels the rewiring quality is independent from the delay. It is used also to linearly scale the delay weight based on the delay level
- $\mathcal{R}$ is a constant used to scale the delay weight. The maximum delay weight assignment is $2\mathcal{R}\mathcal{L}$
- $\mathcal{A}$ is used as an additional weight that estimates the criticality of a connection. It has also been tested with a dependency on the number of literals in the *SOP* but it showed a worst quality of the rewiring.

4.5 Algorithm

The developed algorithm is executed for each critical input and gate s inside of the network, excluding buffers and inverters, in topological order. The rewiring conditions are extracted and s is rewired to a new set. These operations can be synthesized by these following steps:

1. Test s at value $v \in \{0,1\}$
2. Propagate the value through the network
3. Check for conflicts
4. Extract a global flow graph based on the controlling conditions
5. Assign weights to the nodes
6. Extract the min-cut
7. Rewire s to the min-cut
   (a) Connect s to the new set

Algorithm 1 Weights computation

$\mathcal{D} = compute\_depth\_graph(\mathcal{G})$ ▷ compute nodes' depth levels in the graph
$d_{nand2} = get\_delay\_nand2()$
$min\_req = \infty$
for all nodes n in $\mathcal{G}$ **do** ▷ Finds the minimum required time
  if n is not *sink* **then**
    $req = \min(req\_time\_rise(n), req\_time\_fall(n))$
    $min\_req = \min(req, min\_req)$
  end if
end for
$\mathcal{W}_{sink} = \infty$ ▷ Sink has infinite weigth
for all nodes n in $\mathcal{G}$ **do** ▷ Calculate weights
  if n is not *sink* **then**
    $req = \min(req\_time\_rise(n), req\_time\_fall(n))$
    $\delta = (req - min\_req)/d_{nand2}$ ▷ Improvement in *NAND2* levels
    if $\delta > \mathcal{L}$ **then**
      $\mathcal{W}_n^{req} = 0$ ▷ Delay has not contribute
    else
      $\delta' = 2^{\delta}$ ▷ Exponential dependancy weight
      if $\delta' > 2\mathcal{L}$ **then**
        $\mathcal{W}_n^{req} = \mathcal{R}$ ▷ Minimal delay weigth assignement
      else
        $\mathcal{W}_n^{req} = \mathcal{R}(2\mathcal{L} - \delta')$ ▷ Scaled delay weigth assignement
      end if
    end if
    if $\mathcal{D}_n \neq 1$ and n is *weakly* controlled **then**
      $\mathcal{W}_n^{crit} = \mathcal{A}$ ▷ Critical rewiring
    else
      $\mathcal{W}_n^{crit} = 0$ ▷ Rewiring is not critical or in the immediate fanout
    end if
    $\mathcal{W}_n = 1 + \mathcal{W}_n^{req} + \mathcal{W}_n^{crit}$
  end if
end for

(b) Simplify redundant connections

8. Cost the move

9. Commit the change if successful

Now we will comment on all these operations in order to better understand how this process works.

The first step consists in testing a value v on s. We assume that s is a constant at value v in order to extract controllability conditions. It is mandatory to do it both for value 0 and 1 since these values find different rewiring conditions.

In step 2, we need to propagate this assumption over the network to see how the circuit reacts. It is based on a recursive function that implies values at the fanin and at the fanout. But for signal s, we must separate the propagation step in back-propagation and in a forward-propagation. The propagations are separated because, during backward propagation, some external conditions, which may lead to conflicts, might be found. A conflict happens when the tested signal s cannot take value v. In this case, signal s is substituted by a constant signal at value $\neg v$. One simple case is when a part of the network is a tautology. Let's see it in a simple example:

$$g_1 = a \wedge b$$

$$g_2 = \bar{a} \wedge \bar{b}$$

$$g_3 = g_1 \vee g_2$$

$$F = g_3 \vee c$$

This network is a tautology. A global flow cycle is first successfully executed on signal a tested at 0. The transformation leads the following network:

$$g_1 = b$$

$$g_2 = \bar{b}$$

$$g_3 = g_1 \vee g_2$$

$$F = g_3 \vee c \vee \bar{a}$$

Now node g_3 is tested at value 0. Doing a back-propagation, we notice that b is suppose to take both value 0 and 1. This is absurd, so for g_3 is impossible to take value 0. The final optimized network, where g_3 has been substituted with a constant 1, is:

$$g_3 = 1$$

$$F = g_3 \vee c \vee \bar{a} = 1$$

At step 4, the global flow graph is generated as explained in the previous sections. If the number of nodes in the graph is not higher than two, only *source* and *sink*, the graph is discarted. If the graph is valid, the weight are assigned using algorithm 1 and the min-cutset of nodes is extracted.

Before executing the move at step 7, the slack of signal s is saved to check improvement later on. Then the new connections are created for all the gates in the cutset:

- If the gate is an immediate fanout of s, excluding buffers and inverters, and it is *weakly controlled*, s is reconnected to its output accordingly, creating a *OR* or *AND* gate accordingly to corollary 3.
- If the gate is not an immediate fanout of s, it is connected using corollary 3. If it is possible, s is connected as an additional input.

Then, the redundant connections are simplified using corollary 4. For all the immediate fanouts of s, which are not in the cutset or are in the cutset but *weakly controlled*, s is substituted with a constant value $\neg v$.

In the end, if the slack of s is improved, the move is committed, else, it is discarded.

4.6 Implementation

Global Flow is an elaboration that can be used for many purposes. Just changing the weight strategy, many different optimizations can be achieved. In our implementation, we focused only on delay optimization. Global Flow gives the best results on big networks since more rewiring conditions can be found. So, our implementation will work directly on entire combinational hierarchies.

Global Flow is executed on all the critical hierarchies of the whole design. We found out that the performances can be improved if the algorithm is executed two times with different parameters. We explained, in section 4.4, how the weights are assigned and which parameters can be used. For instance, incrementing $\mathcal{L}$, the weight depends more on the delay while, decrementing it, it depends more on the area. The first run is the most important. It must find good connection assignments based on the delay. But, doing tests, we noticed that, if we do another run to reduce the number of connections, the delay can be further improved. The second run will have a higher $\mathcal{A}$ parameter. Even if the goal is to find less area expensive connections in terms of area, the algorithm is anyway based on delay. So the second run reduces the connections only if the delay can be improved. Of course, the second run is executed fewer times than the first one since our global flow rewires only on the fanout cone which results smaller after the first run.

After the algorithm is run, the modified gates of the circuit are re-mapped. Then the move is costed and applied.

Chapter 5

Global Flow Results

5.1 Introduction

In this chapter, we will analyze the performance of Global Flow on different designs. We claimed that our algorithm may be applied on a general type of gate for a delay improvement. In the next section, we are going to present some results which support this thesis. Unfortunately, due to a lack of time and technical challenges, the algorithm has not been tested inside a Fusion Compiler flow. So, the results of percentage gain at the end of physical optimization are not presented. However, we are going to show some results on some netlists extracted from the synthesis flow, after some initial optimizations, which could demonstrate an improvement in the QoR.
At the end of this chapter, we will present some ideas to improve even further the algorithm.

5.2 QoR

In this section, we will demonstrate how the quality of results of modern designs can be improved by Global Flow. We will focus our attention on different combinational designs characterizing them by the following parameters:

- **PI**: number of primary inputs
- **PO**: number of primary outputs
- **Nodes**: number gates in the design
- **Terms**: number of products between literals in the design
- **Lits**: number of variables in the design
- **WNS**: worst negative slack of the design

We will look mostly for a *WNS* improvement but we will keep an eye also on the number of literals and terms. The results will show the circuit's parameters before and after optimization. To all the designs, a zero required time has been set for all the primary outputs in order to improve the circuit as much as possible. Thus, the slack will be always negative. Closer the slack is to zero, faster is the circuit.

As a first example, let's try to run Global Flow on a simple not optimize design. All the nodes are represented by a single product of two variables. For instance, $\bar{a}b$ is a possible node. The results, shown in table 5.1, determine how the slack can be enhanced in a design by changing the connection set. In fact, the slack improved by the 34%. Also, we can notice how the number of literals and terms is considerably reduced due mainly to redundancy removal. But, on the other hand, we have also to consider that gates with more than two inputs have been created which may affect the decrease in the number of literals.

Stage	**PI**	**PO**	**Nodes**	**Terms**	**Lits**	**WNS**
Original Design	23	2	2773	2775	5546	-12.19
Global Flow	23	2	1171	1329	3068	-8.03

Table 5.1. Global Flow for delay

As we claimed that the algorithm can be used on a generic context, in the following example, in table 5.2, we executed it on a design with generic gates, extracted

from a mapped netlist. The network may contains gates as *AOI22*, *AO21*, *XOR*, etc. Here the design is bigger with respect to the previous example. The *WNS* is notably improved only with rewiring, without almost changing the number of literals.

Stage	**PI**	**PO**	**Nodes**	**Terms**	**Lits**	**WNS**
Mapped Design	21232	21669	321413	571912	1152308	-27.43
Global Flow	21232	21669	321414	571912	1152315	-22.83

Table 5.2. Global Flow on mapped gates

As the last example, in table 5.3, we want to verify that Global Flow can be successful also on already optimized designs. So, for this test, we will proceed with two steps. First of all, we will apply delay-based optimizations to the original design. Then, we will apply Global Flow on the optimized netlist. From the first pass of optimization, we can notice a considerable delay improvement. Since a lot of operations have been parallelized, the number of nodes and literals is almost doubled. From this optimized network, we run Global Flow. The worst critical slack of the circuit has been further improved of about 7%.

Stage	**PI**	**PO**	**Nodes**	**Terms**	**Lits**	**WNS**
Original Design	1024	1231	46876	47403	93711	-79.53
Initial Opto	1024	1231	87374	133132	170258	-54.56
Global Flow	1024	1231	87286	132920	170363	-50.55

Table 5.3. Global Flow on an already optimized design

From these results, in particular from the last one, we saw how Global Flow can apport some new logic elaboration which improves the delay. Of course, better tests must be performed in order to show how much impact it can have on a full optimization flow. We don't expect it to produce noticeable improvements in flattened optimized netlists. In shallow logic, it may not be able to find enough cuts

to rewire. We believe, though, that Global Flow could improve the QoR on deeper models.

5.3 Further improvements

During our analysis of Global Flow, we announced some limitations of the previous graphs and we focused our work to generalize the context of application. But Global Flow loses its optimality during the creation of the graph. If some edges are redundant, the extracted cutset may be bigger than needed. In this section, we will introduce some improvements that may improve the quality of this algorithm.

In the graph creation section, we said how our graph is mainly based on the topology of the circuit. All the connected implied gates in the netlist will be connected also in the graph. This may lead to redundant connections and bigger cutsets. The best approach would be to connect nodes only when they have a direct impact on the controllability of the gate. Let's take, for instance a *AO21* gate with function $g = ab \vee c$. Let's suppose that the gate g is controlled at 1 thanks to two of its fanins: $a = 1$ and $c = 1$. In the global flow graph, both a and c nodes will be connected to the g node. But we can notice that, in this assignment, the connection between a and g is redundant as only the c assignment is sufficient to imply the value. So, some tests must be executed to see how Global Flow performances can be further improved.

Another similar case may lead to a bigger cutset. Let's suppose that more than one fanin can control by itself a gate. For instance, let's take an *OR* gate $g = a \vee b \vee c \vee d$. Let's suppose that both a and b are at one and control g to one. Only the edges (a, g) and (b, g) are created. We can notice that only one of these two edges is strictly necessary in order to guarantee the controllability and correctness of the graph. Thus, a connection (a, g) or (b, g) may be removed to reduce a cutset. The problem of selecting which edges are kept and which are discarded is mainly discussed in [5]. Some strategies and tests must be conducted to understand how

much it may affect the quality of the min-cut, in particular for delay.

For a last point of reflection, the graph is created with only the controlled gates. We know that we can simplify the connections when the nodes are *blocked* by another set of controlled nodes. This check is realized only for the nodes in the graph. In an alternative version, also non-controlling gates may be checked. In fact, if also other non-controlled gates are blocked by the min-cut, their connection can be simplified with a constant without changing the functionality of the graph.

Chapter 6

Conclusion

In this thesis, we presented some approaches to logic optimization. In particular, the objective was to investigate methods to improve the delay on mapped or generic netlists without electric optimization, which are not sized and not buffered.

In the first part, we focused on restructuring tricks that act on clusters of few gates using the associative and distributive property. We built an approach based on pushing critical signals forward in the logic and duplicating critical nodes under multiple output trees. We inserted these algorithms into an engine responsible for extracting sub-networks to be optimized and applying the changes whenever the delay improves. From the results, we showed that these moves, in specific cases, may find better transformations with respect to algebraic or other techniques. Moreover, the pull moves add a valuable contribution to the QoR. In particular Pull Factor, with its distributivity of *and*s over *xor*s, can propose an elaboration that algebraic techniques are often not able to find.

In the second part, we introduced a new approach to global flow analysis. This approach changes, for each gate, its set of connections with an optimized equivalent one. Earlier implementations were applied to circuits containing simple gates to minimize the area. We extended the method to support mapped and generic

gates. Moreover, the different graph model we use is specific for a delay improvement. So, we developed a heuristic to extract a new set of connections based on the gates' criticality. We presented, also, some ideas for further research to create a better graph model.

From the tests we run, global flow is effective both on generic gates and on already optimized circuits. Compared to algebraic or other techniques, the improvement that can be obtained varies from design to design. In particular, we do not expect it to produce noticeable improvements on optimized flattened netlists. However, supported by the tests we presented, we believe that on big and deep designs, which could not be flattened, it is likely that global flow could help to improve the QoR.

Bibliography

[1] G. De Micheli, "Synthesis and Optimization of Digital Circuits", McGraw-Hill, 1994

[2] Synopsys, Fusion Compiler Datasheet, 2018, https://www.synopsys.com/content/dam/synopsys/implementation&signoff/datasheets/fusion-compiler-ds.pdf, Accessed: 2019-09-30

[3] I. Sutherland, B. Sproull, D. Harris, "Logical Effort: Designing Fast CMOS Circuits", Morgan Kaufmann, 1999

[4] C.L Berman, L.H Trevillyan, "Global flow optimization in automatic logic design", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 10, No. 5, May 1991, pp. 557-564, DOI 10.1109/43.79493

[5] Shih-Chieh Chang, Zhong-Zhen Wu, He-Zhe Yu, "Wire reconnections based on implication flow graph", EEE/ACM International Conference on Computer-Aided Design, Digest of Technical Papers, Feb 2000, pp. 533-536, DOI 10.1109/ICCAD.2000.896527

[6] R. Damiano, L. Berman, "Dual global flow", IEEE International Conference on Computer Design: VLSI in Computers and Processors, Nov 1991, pp. 49-53, DOI 10.1109/ICCD.1991.139842