A Divide-and-Conquer Approach for Cell Routing using Litho-friendly Layouts

Author:
Alexandre Vidal Obiols

Supervisor:
Jordi Cortadella Fortuny

Dept. de Llenguatges i Sistemes Informàtics

Enginyeria Informàtica
Facultat d'Informàtica de Barcelona

June 17, 2013

DADES DEL PROJECTE

Títol del projecte:	A Divide-and-Conquer Approach for Cell Routing using Litho-friendly Layouts
Nom de l'estudiant:	Alexandre Vidal Obiols
Titulació:	Enginyeria Informàtica
Crèdits:	37,5
Director:	Jordi Cortadella Fortuny
Departament:	Llenguatges i Sistemes Informàtics

MEMBRES DEL TRIBUNAL *(nom i signatura)*

President:	Jordi Petit Silvestre
Vocal:	Roser Rius Carrasco
Secretari:	Jordi Cortadella Fortuny

QUALIFICACIÓ

Qualificació numèrica:

Qualificació descriptiva:

Data:

Preface

The aim of this project is to develop divide-and-conquer algorithms for the routing of signals in nanoelectric standard cells. We work on an already existing cell synthesis framework based on solving a boolean satisfiability formulation of the routing problem. This approach takes too much computational time when dealing with big and complex cells. Our goal is for such cells to become tractable. Different ways to divide the routing problem of a cell are explored. We evaluate several strategies and try to find the ones with the best solution for a given cell and design rules on terms of computational time. The approach gives good results where a valid routing is found in much less computational time than before, but still work has to be done for specially congested cells.

Chapter 1 provides background on logic synthesis, the routing problem, circuit physical design considerations and the applications of the satisfiability problem. The following chapter is about CellRouter, the cell routing framework this project aims to improve, including a brief description of what it is based on and some experimental results. Chapter 3 is devoted to explain what the initial considerations of the project were with respect to the original tool and the implementation decisions that were made. Chapter 4 includes an overview of the partitioning decisions that have been used as a divide-and-conquer strategy. Chapter 5 includes several experiments conducted using the Nangate Open Cell Library and another cell library created for testing purposes. Finally, conclusions and future work directions are given in Chapter 6.

This work has been funded by a gift from Intel Corporation (Strategic CAD Lab, Hillsboro, USA).

Contents

Chapter 1

Background

1.1 VLSI and EDA

The complexity of Integrated Circuits (ICs) has been growing year after year since they were first introduced. According to Moore's Law [1], the density of transistors on a chip doubles approximately every 2 years. This tendency has been followed for the last 40 years, but of course such a fast-paced evolution of the number of transistors comes with a lot of challenges at many levels, such as technology, design and tools. Very Large System Integration (VLSI) is the technology that allows combining millions of transistors in a single chip such as a microprocessor. This field has been constantly evolving, trying to make faster chips and integrate more transistors generation after generation. As the number of transistors has dramatically increased over the years, the complexity of circuits has also increased enormously; and with it, the challenges associated to the design of such circuits.

The design of VLSI circuits is therefore a very complex process that requires automation. Electronic Design Automation (EDA) is a category of software tools for designing electronic systems such as ICs. This aid has been evolving together with the needs of VLSI design since the mid-70s. Nowadays, given the level of complexity that VLSI design has reached, EDA tools play a very important role in the fabrication of ICs.

Current workflows for the fabrication of chips are very modular. The Register-Transfer Level (RTL) design abstraction, which models synchronous digital circuits and focuses on the flow of digital signals between hardware registers, is used to describe a circuit. Hardware Description Languages (HDLs) such as Verilog and VHDL can be used to create such high-level

representation of circuits. EDA tools are used to design and implement technology-dependent circuits from a higher level of abstraction. Automated synthesis goes through a lot of steps to transform an RTL design into a geometrical layout, creating a physical design for the given RTL specification. Given a high-level specification multiple final circuits can be considered valid, so there is room for a lot of decisions and optimization to be done in order to generate a good physical design of a circuit.

Given an RTL circuit, these are some of the steps it goes through during the physical design phase. Note that all of these processes are automated by using the above mentioned EDA tools. We can consider that the output of one step is the input for the next one.

Logic Synthesis

Given a circuit abstraction, such as an RTL circuit, logic synthesis returns an implementation of the circuit in terms of logic gates. The RTL design is translated into boolean expressions. These formulas can be optimized using exact methods such as the Quine-McCluskey algorithm [2, 3], heuristic methods such as Espresso [4] or kernel and boolean factorization. After these technology-independent steps, the formulas are mapped onto a given cell library resulting on a netlist in that technology library, using data structures such as Directed Acyclic Graphs (DAGs) and dynamic programming algorithms. To show a little example, this is a very simple module written in Verilog.

```
module some_logic(a, b, c, out);
    input a, b, c, out;
    output out;

   assign out = (a & b) | c;

endmodule
```

Figure 1.1 represents the output of the logic synthesis phase receiving this code as an input.

Floorplanning

This is the first step in the physical design flow. Floorplanning consists in identifying structures that should be placed close together, capturing relative positions rather than fixed coordinates. It can be considered a generalization of placement, a first draft of how things will be al-

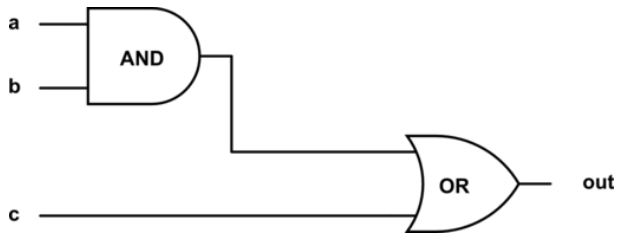

Figure 1.1: Logic gate level representation

located in the chip, allowing transformations of the components such as rotations and modifying their shapes. Simulated annealing, trees and slicing structures, as well as dynamic programming for floorplanning optimization[5], are widely used in this area. A floorplan can be optimized for metrics such as area, wirelength, routability and others. Figure 1.2, taken from [6], shows two different floorplans for a given set of components. The floorplan on the left is optimal in area while the one in the right introduces white spaces.

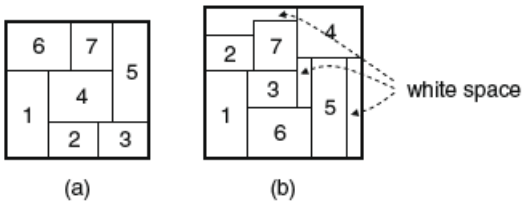

Figure 1.2: (a) Optimal area floorplan (b) Non-optimal area floorplan

Partitioning

The netlist of the functions to implement can be very large. Partitioning is the process of dividing the chip into smaller blocks so that later placement and routing are easier, using a divide-and-conquer strategy to tackle design complexity. It is also a necessary step in the case of synthetisation on Field Programmable Gate Arrays (FPGAs), where a mapping from the netlist to hardware is needed. Many variants of partitioning exist, such as two-way partitioning (one of the first approaches, [7]), multi-way partitioning, which can be seen as an extension of the min-cut for two-way partitioning, and multi-level partitioning, where the result is represented by a tree structure. More partitioning approaches can be found in [8].

Placement

This step consists in assigning cells to positions in the chip according

to some cost functions while preserving legality (for example, with no overlapping). The inputs are the netlists and the goal is to find the best position for each module considering wirelength, routability density, power and other metrics. Many placement styles exist depending on the design methodology it is integrated with (such as building blocks, standard cells or gate arrays). This step is tightly related to the next phase, routing. Some placement paradigms are:

- Constructive algorithms, such that when the position of a cell is fixed, it is not anymore modified. Some examples are cluster growth, min-cut [9], or quadratic-placement algorithms (such as Hall placement [10], the first analytical placer).
- Iterative algorithms, where intermediate placements are modified in order to improve some cost function. This would include analytical methods such as force-directed placement. Figure 1.3, taken from [6], shows several phases of the placement in a force-driven algorithm. The elements approach their final position iteration after iteration.
- Nondeterministic approaches, including metaheuristics like simulated annealing and genetic algorithms.

All these methods can be combined to obtain a more accurate result. Additionally other methods can be considered, for example a flow consisting of a global placement and legalization phase followed then by detailed placement step. There are many interesting research directions in placement such as manufacturability-aware placement, but probably the most interesting for our project would be routability-driven placement[11]. More information about placement algorithms can be found in [12].

Routing

The routing process determines the precise paths for nets on the chip layout, respecting a set of design rules to ensure that the chip can be correctly manufactured. It requires a physical placement of the layout, the netlists and the design rules required by the fabrication process. The main aim is to complete all required connections on the layout, although other objectives such as reducing total wirelenght or meeting timing requirements have become of essential relevance in modern chip design. The routing phase represents a very complex combinatorial problem. Usually, a two-step approach consisting of a global routing

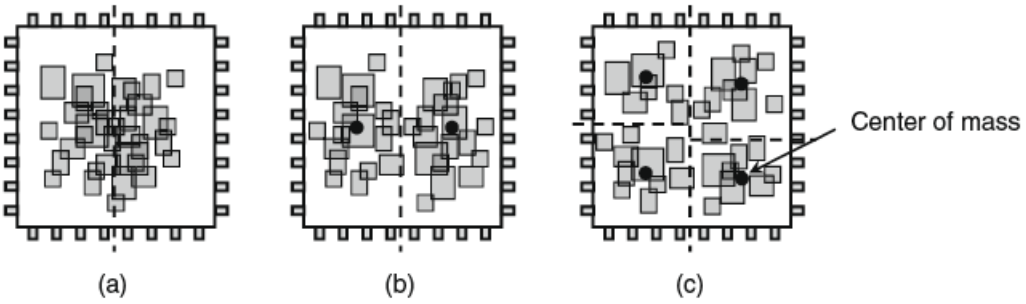

Figure 1.3: Placement of a chip

followed by a detailed routing is used. The first considers the connection between different regions of the chip, while the second focuses on obtaining a definite geometric layout for the wire connections.

These are only some examples of steps where algorithms have become indispensable for the design of ICs. As we can see, EDA tools have become a basic component of digital circuit design. All of these steps have an important algorithmic load and much effort has been invested in such crucial synthesis tools.

1.2 Cell Routing

Routing is one of the multiple steps that take place in the physical design process. For the last years many algorithmic techniques have been explored to address the complex problem of determining how the components of circuits should be interconnected. As the number of transistors per chip grows, the increasing complexity of the design becomes a challenge for the routing stage. It is typically a very complex combinatorial problem that, as mentioned before, is usually solved using a two-stage approach: global routing and detailed routing. In this section we will overview both, as well as algorithms fitted for general routing.

1.2.1 General Considerations

The main aim of the routing problem is to find a valid interconnection of terminals that honors a set of design rules. Typically, most routing algorithms are based on graph-search techniques guided by parameters such as

congestion and timing information, trying to find a balance of the net distribution among routing regions. For example, a chip might be partitioned into an array of tiles. Then the global router would find tile-to-tile paths for all nets on such a graph and use this information to guide the detailed router.

For the detailed routing step, two kinds of models exist: the grid-based and the gridless-based models. In the first, a grid is superimposed on the routing region and the detailed router finds routing paths in the grid. Gridless-based models, on the other hand, can use different wire widths and spacing. Even if they have more flexibility, grid-based routing is usually more efficient and easier to implement given its lower complexity compared to the other model.

When routing, two kinds of constraints appear: performance constraints and design-rule constraints. The objective of the performance constraints is to make connections meet the performance specifications provided by the chip designers. Design rules, on the other hand, are a set of additional constraints imposed by a given technology node that will have to be honored if we want the chip to be correctly manufactured. They impose restrictions on, for example, the minimum width of the wires or the wire-to-wire spacing. Another example of design rules is related to the layer models, which can be either *reserved* or *unreserved*. In the first case each layer is allowed only one routing direction, whereas the placement of wires with any direction is permitted in the other one. Most of the routers, however, use the reserved model because it has lower complexity and is much easier for implementation; as we will see, manufacturability has a great impact on many of the decisions taken during the physical design flow.

1.2.2 General-purpose Routing

As mentioned before, graph-based algorithms have been extensively used for both global and detailed routing. In this section we will introduce the *maze routing algorithm*, probably one of the most basic graph-search based algorithms.

Maze routing is based on a Breadth-First Search (BFS). Consider the grid on Figure 1.4. We want to connect the node marked with an S to the one marked with a T. The grayed zones represent obstacles where the wire cannot be placed.

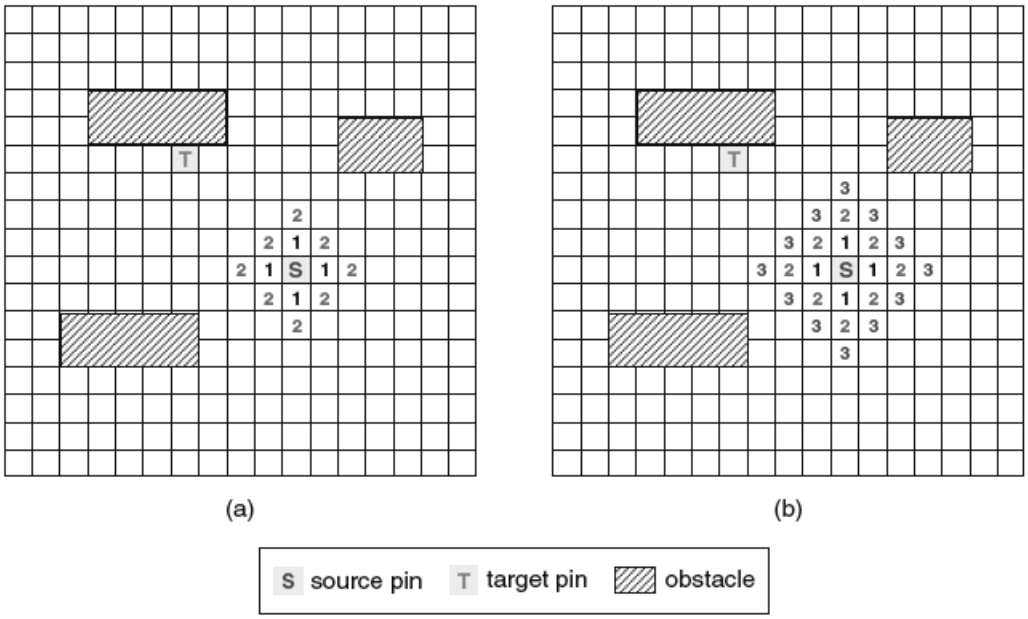

Figure 1.4: Maze Routing - Wave Propagation

The maze router is composed of two basic steps: *wave propagation* and *retracing*. During the first step, starting from the source S, all adjacent nodes get labeled with a 1, which is the distance from such nodes to S. In the next step, all cells adjacent to the nodes labeled 1 get labeled with a 2. This continues until the node T is reached. We can see the wave propagation phase in Figure 1.4 with the waves corresponding to 2 and 3. Once T is reached, as seen in Figure 1.5(a), a shortest path from T to S can be retraced by following any path such that the labels of the nodes decreases as shown in Figure 1.5(b). Often, the preferred path is the one with the least number of detours. Notice that this algorithm guarantees to find the shortest path between two points if such a path exists. Both figures illustrating the example have been taken from [6].

This algorithm was proposed by Lee in [13] and is also widely known as *Lee's Algorithm*. In practice, it is slow, memory consuming and difficult to apply to large-scale dense designs. Many methods have been proposed to reduce its running time and memory requirements. For example, alternative coding schemes for the nodes have managed to reduce the number of needed bits to merely two. Other variants include using depth-first search in combination with the BFS, starting point selection or double fan-out.

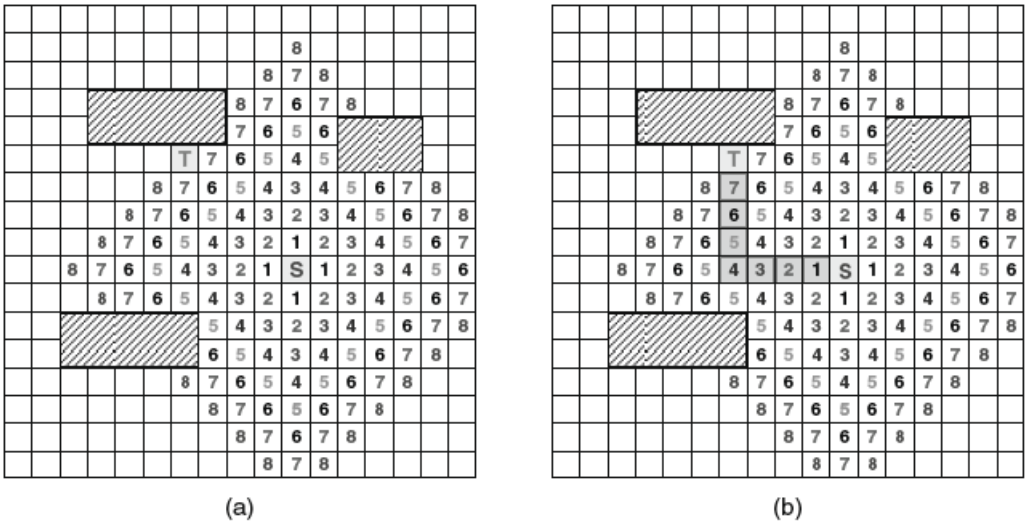

Figure 1.5: Maze Routing - Retracing

After this algorithm, many other graph-based search algorithm appeared. The most significant ones probably are Line-search routing [14] and algorithms based on the well known A*-search proposed in [15], which are widely used in modern routers.

1.2.3 Global Routing

Global routing can be considered the most coarse grain type of routing. It consists in defining routing regions, generating tentative routings for nets and associating them to routing regions, but without specifying the actual layout of the wires. There exist two kind of global routing algorithms which differ on the basic routing strategy: sequential and concurrent algorithms. Whereas the first ones try to route signals one by one, concurrent algorithms try to find a valid solution for all signals at once. We will see a small introduction to each approach.

Sequential global routing

In this schema we select a specific net order and then route nets sequentially according to that order. The quality of the solution greatly depends on the ordering given that an already routed net might block the routing of subsequent nets. Finding the optimal net ordering has been proven NP-hard. Often, a *rip-up and re-route* heuristic is used to refine the solution. It basically consists in ripping-up some already connected nets and then re-route the ripped-up connections. It usually

performs iteratively until all nets are routed, a time limit is exceeded or no gain is obtained. As we will see, CellRouter uses a similar heuristic to obtain better solutions once an initial legal routing has been found. The main drawback of this method is that, because of the dependence on net-ordering, if no feasible solution is obtained, it is not clear whether it does not exist or the chosen ordering was not good enough.

Concurrent global routing

Concurrent global routing tries to establish all connections at the same time. Therefore, whether or not a solution is found does not depend on any net ordering. One of the most popular approaches is to model the layout as a graph and then use *0-1 integer linear programming*. However, given it is NP-complete, another approach would be to solve the continuous *linear programming* relaxation and the transform the fractional solution to integer solutions through a rounding scheme such as randomized rounding. In practice, such techniques are embedded into larger global routing frameworks which use a hierarchical, divide-and-conquer strategy. For routing multi-pin nets instead of two-pin nets, those multi-pin nets are usually decomposed using *minimum rectilinear Steiner trees*.

1.2.4 Detailed Routing

The output of the global routing stage is used by the detailed router to determine the exact geometry of the nets in the chip. A popular type of detailed routing related to this project is full-chip routing. To cope with the scalability problem, routing frameworks use hierarchical and multilevel frameworks for large-scale designs. They use a divide-and-conquer approach by transforming large routing instances into smaller subproblems and later proceeding with a top-down, bottom-up or hybrid manner. In the first approach, the algorithm recursively divides the routing regions into smaller regions, routes the current level and refines the result in the next level. On the other hand, a bottom-up approach consists on initially partitioning the region into multiple small cells and, at each step, routing each region individually and merging it with its neighbors to form a larger supercell until the whole initial region is routed. The routing decisions made at any of the intermediate routing levels might be suboptimal, so hybrid approaches using both methods have also been explored. However, since routing decisions at a given level are irreversible, the quality of the solution is limited. This is a problem we will also face later in this project.

The same classification that we exposed for global routing algorithms can be applied to detailed routing algorithms. Sequential algorithms may not guarantee a solution even if it exists. For this reason, the most recent approaches tend to use concurrent routing. When doing detailed routing from a concurrent approach, two kind of algorithms can be considered, depending on the objects used to take routing decisions. *Tree-based* algorithms first generate a set of candidate routing trees using algorithms that generate multiple *minimum rectilinear Steiner trees*. Next, the problem is formulated as a multicommodity flow problem and solved as a 0-1 integer linear programming problem. On the other hand, *segment-based* algorithms take decisions at the level of individual metal segments. This version has a finer granularity, at the expense of more computational complexity. However, manufacturing constraints are more easily represented in this scheme. Usually, SAT-based formulations are used to solve the routing problem under the segment-based approach.

1.2.5 Modern Routing

It is interesting to consider the previous work directly related to the tool this project is based on. The closest is proposed in [16], a segment-based approach inspired by the satisfiability formulation presented in [17]. However, [16] only managed to route small gates given the high computational complexity of the resulting formulation, as will be shown in Chapter 2.

As we have seen, many kind of routing strategies and approaches exist. It is not by sticking to one of the methodologies that routing can be easily solved. Given the complexity of the problem, many kinds of routers exist that combine several of the ideas briefly exposed in this short introduction to routing. For example, BoxRouter 2.0 [18] is an academic global router that uses A*-search considering the congestion history of edges, wire rip-up and rerouting and finally a progressive integer linear programming approach to do layer assignment. The routing problem is constantly evolving and new algorithms and techniques will for sure continue to arise in order to meet with its new requirements. Fabrication considerations are day after day becoming more determining during the physical design process.

The figures for this section where taken from *Electronic Design Automation: Synthesis, Verification, and Test*[6]. For more information on routing or VLSI algorithms in general, refer to at that book or to *Handbook of Algorithms for Physical Design Automation*[19].

1.3 Design Considerations

When building digital circuits there exist multiple design styles, and the one used is chosen depending on the needs of the target design. Full-custom design is based on specifying the layout of each individual transistor and the interconnections between them. It potentially maximizes the performance and minimizes the area, but it is very laborious to implement. Full-custom design is typically used in a situation where there are area limitations or special application needs. Some examples are the design of cells within a standard cell library, memory cells or datapaths for high performance designs. However, other design methodologies exist.

1.3.1 Standard Cell Design

In the standard cell methodology, low-level VLSI layouts are encapsulated in abstract logic representations (for example, as a NAND gate). This way, the logical level becomes independent from the physical level design. Using this design methodology, high-level design time becomes shorter as designs can be reused. Standard cell design relies on so-called standard cell libraries which contain primitive cells (such as AND or inverter) required for cell design. Additionally, more complex optimized cells can also be included. The cells in the library have a fixed height and variable width. This way they can be easily placed in rows easing the synthesis process. All cells on the row will be constructed according to a certain structure, for example the one shown in Figure 1.6. They are normally optimized full-custom layouts so that area and delay are minimized.

Each one of these cells is described by the following views. Additionally, metrics such as timing, power and noise for each cell are provided.

Logical View

Cell's boolean logic formula, captured in a truth-table or boolean algebra equation, in the case of combinational logic, and a state transition table, in the case of sequential logic. Figure 1.7 is an example of a state transition table for an AND gate taken from the Nangate Cell Library documentation.

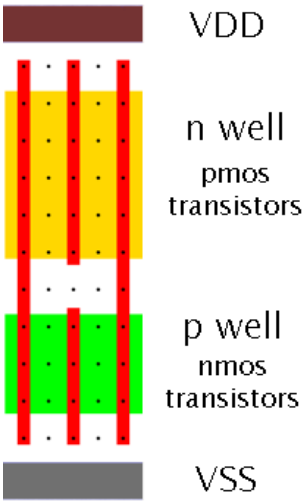

Figure 1.6: Possible physical structure

State Table		
A1	A2	ZN
L	-	L
H	H	H
-	L	L

Figure 1.7: AND gate, logical view

Schematic View

Description of the transistors, their connections to each other and the terminals to the external environment. Multiple possible schematic representations for a given logical view exist. Figure 1.8 is a schematic representation of an AND gate, also taken from the Nangate Cell Library documentation.

Layout View

Physical representation of the cell. The most important from the manufacturing point of view, as it is the closest to the actual final design. Again, many possible layouts exist for a given schematic description of a cell. Figure 1.9 represents the layout of an AND gate.

Given the project works on the routing of standard cells, a simple method of representation of such cells has been chosen to illustrate important concepts in this report. It does not intend to be formal or exact, but useful as an

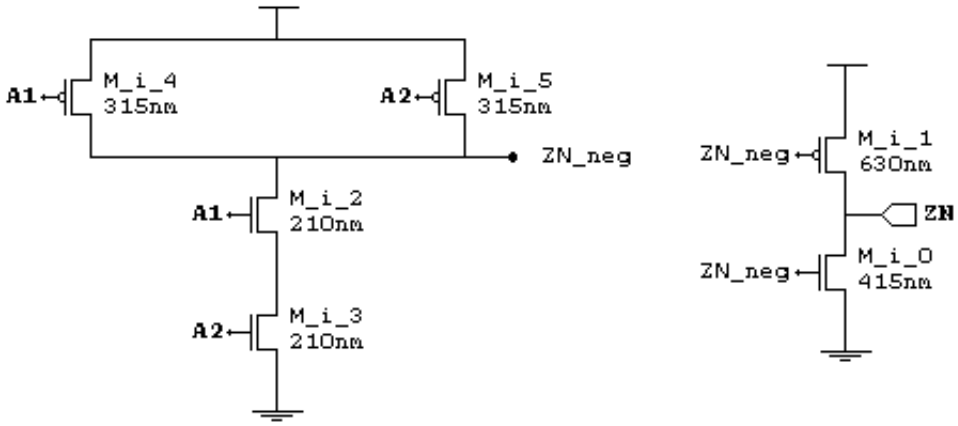

Figure 1.8: AND gate, schematic view

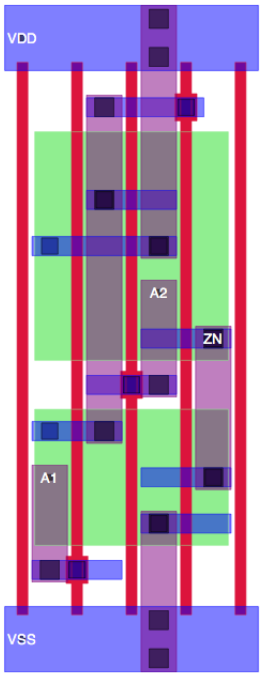

Figure 1.9: AND gate, layout for schematic in Figure 1.8

abstraction to show situations that happen in real routing problem instances. Figure 1.10 shows an example of how one of these grids would look like. In such a 2D representation of a cell, all components sharing the same color are the ones that must be connected. When a terminal has no color, it means it is not important for what the figure intends to show so it is ignored. The

voltage and ground signals are always on the top and the bottom of the cell. In this simple representation we will consider that wire segments can run along the columns and the indicated rows in the figure.

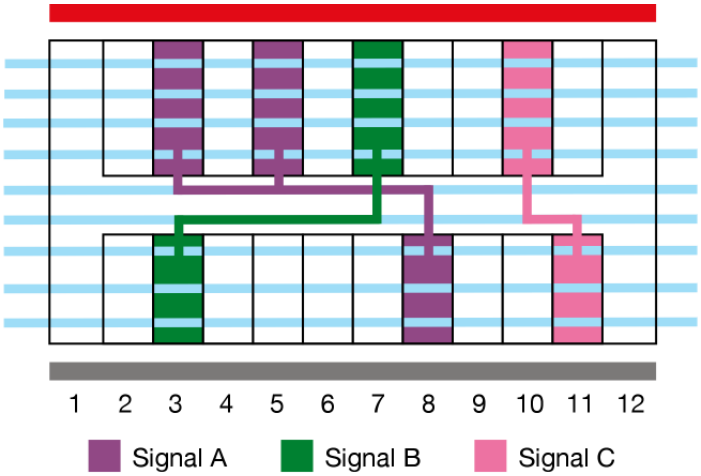

Figure 1.10: An example of a cell representation

1.3.2 Manufacturability-Aware Design

As we have seen, standard cell design provides help by encapsulating primitives so that design becomes more modular and regular. However, in the last years additional problems have been adding up to the challenges of IC design. As the size of transistors decreases, the manufacturing process has become increasingly complex.

One of the crucial processes involved in chip manufacturing is photolithography. During this process, chemicals which either harden or soften when exposed to ultraviolet light are used. Such products are applied onto the surface of the die and are exposed to light trough a mask with the desired pattern. After this step, the softened parts are removed and another mask with the next pattern is used. Layers grow one above the other until all masks have been applied. The patterns on the masks are the final product that the physical design produces for a given circuit abstraction.

This whole lithographic process is crucial to the fabrication of ICs. Given the constant reduction in the size of the transistors as fabrication technology changes, the lithographic gap between such small component sizes and the light wavelength is having an increasingly important impact on the patterns

used during the manufacturing process. Currently, Resolution Enhancement Techniques are applied to obtain transistor sizes much smaller than the light wavelength [20]. Such approach is limited to a certain amount of geometrical configurations, contributing to an enormous increase of layout design rules at each technology generation. The design costs increases for two reasons. On one hand, the enormous effort required to verify such layouts. On the other hand, masks must be pre-distorted in order to compensate the distortions later introduced during the photolithography phase.

Litho-friendly layout techniques must be considered to deal with all this increasing complexity. These techniques exploit the use of one-dimensional features and gridded locations for layout elements. This complicates the design of standard cell libraries, that must now cope with such restrictions trying to provide the best possible area, performance and power consumption. However, using such techniques, manufacturability-aware physical layouts are produced, which are friendly to the fabrication process.

The aim of this project is to help in the synthesis of such manufacturability-aware standard cells. Today, it is a process where complex design rules are imposed. Given so, regularity has been introduced as a means to make design automation tractable. Such regularity can be seen for example in Figure 1.6, where a gridded layout is used. A clear example of such methodology can be found in [21], which proposes a model where all layout elements are located on a grid of evenly spaced points, with the grid unit begin a fraction of the wavelength of the light which will later be used during the fabrication process. As it will be explained later, the tools used in this project use a similar grid approach and try to be as technology-independent as possible, so that any desired design rule can be specified and the output of the tools is a physical design adapted to the desired technology constraints.

1.4 Boolean Satisfiability Problem

As explained before, EDA tools rely on algorithms to tackle a variety of computational problems which arise during IC design. One of the algorithmic problems that has attracted much attention during the last years has been the Boolean Satisfiability Problem (SAT). As we will see, much progress has been done and it can be used to face real industrial problems. For example, the routing tool used in this project is based on SAT.

1.4.1 SAT problem

First of all we will formalize the SAT problem and give some nomenclature that will be used in the later chapters. Suppose we have a set of *variables*,

$$P = \{p, q, r, \ldots\}$$

We will define a *formula* as follows.

1. A variable is a formula.
2. If F is a formula, $\neg F$ is a formula.
3. If F and G are formulas, $(F \wedge G)$ is a formula.
4. If F and G are formulas, $(F \vee G)$ is a formula.

We will now define an *interpretation* I over P as a map

$$I\colon P \mapsto \{0, 1\}$$

It can be thought of as defining a value, either 0 or 1 (which can be read as *false* or *true*), to every variable in P.

We will also define the function

$$eval_I\colon F \mapsto \{0, 1\}$$

as follows.

1. $eval_I(p) = I(p)$
2. $eval_I(\neg F) = 1 - eval_I(F)$
3. $eval_I(F \wedge G) = min(eval_I(F), eval_I(G))$
4. $eval_I(F \vee G) = max(eval_I(F), eval_I(G))$

We will also consider that I satisfies F ($I \models F$) if $eval_I(F) = 1$.

For example, over the variable set P that has been defined before, these would be well-constructed formulas.

- $F_1 = p$
- $F_2 = r$

- $F_3 = p \wedge q$
- $F_4 = p \wedge \neg p$
- $F_5 = (p \vee q) \wedge r$

Let us define an interpretation I_1 for P.

$$I_1(p) = 1,\ I_1(q) = 1,\ I_1(r) = 0$$

Under that interpretation, if we evaluate all five formulas we obtain

- $eval_{I_1}(F_1) = eval_{I_1}(p) = 1$
- $eval_{I_1}(F_2) = eval_{I_1}(r) = 0$
- $eval_{I_1}(F_3) = eval_{I_1}(p \wedge q) = 1$
- $eval_{I_1}(F_4) = eval_{I_1}(p \wedge \neg p) = 0$
- $eval_{I_1}(F_5) = eval_{I_1}((p \vee q) \wedge r) = 1$

Now, let us define a *model* to be an interpretation for which a given formula evaluates to 1. For example, I_1 is a model of F_1, whereas it is not a model of F_2. We will say that a formula is *satisfiable* if it has at least a model, and we will say it is *unsatisfiable* if there is no I such that $eval_I(F) = 1$. In the case above, F_3 is clearly satisfiable, for I_1 is a model for it. On the other hand, F_4 is clearly unsatisfiable since, for any interpretation I, $p \wedge \neg p$ evaluates to 0.

The SAT problem now is straightforward to define. Given a formula, is there any interpretation that satisfies it? Or, in other words, is there any variable assignment such that the formula evaluates to 1? From the examples above, we can very easily see that all formulas except F_4 are satisfiable.

We will consider that any formula used as an input to SAT is in Conjunctive Normal Form (CNF). First, let us define a *literal* as a variable or the negation of a variable $(l_1, \neg l_1, l_3, \ldots)$. Second, let's define a *clause* as a disjunction of literals $(l_1 \vee l_2, \neg l_1 \vee l_3, \ldots)$. Now, we will say a formula is a CNF if it is a conjunction of zero or more clauses, such as

$$(l_1 \vee l_2) \wedge (\neg l_1 \vee l_3) \wedge (\neg l_2 \vee \neg l_3)$$

Note that for a given F there always exists a CNF G such that $G \equiv F$. CNFs are used because the Tseitin Transformation [22] allows to obtain a CNF from any arbitrary formula such that it is satisfied only by the interpretations that satisfied the original formula, with only a linear growth in size with respect to the original one. Solving the SAT problem for a formula in Disjunctive Normal Form (DNF), which is defined as the CNF but changing disjunctions for conjunctions and viceversa, would be achievable in linear time by scanning the clauses until a satisfiable clause appeared. However, no transformation such that an arbitrary formula is converted into a DNF and avoids an exponential growth has been found.

It is important to note that SAT is NP-Complete, in fact the first one to be known [23]. Some restricted versions are known to be solvable in polynomial time, such as 2SAT and HORNSAT. However, even if it is NP-Complete, many practical instances can be solved in affordable time. Efficient and scalable algorithms for SAT developed in the last years have contributed to the use of SAT-solving engines as an essential tool in EDA.

1.4.2 Using SAT

The SAT problem is of central importance in many areas of computer science and industry. How can the SAT problem help in problems apparently as unrelated as industrial planning, scheduling of football leagues or the routing of standard cells? It is done by reducing a problem to SAT.

Consider a black-box SAT-solver, such that it receives a CNF F as an input and it returns "YES" if satisfiable, with a model that satisfies it, or "NO" otherwise. Reducing a problem to SAT consists on encoding our problem into a formula that we can give as an input to a SAT-solver in such a way that we can, in return, construct a solution to our problem from the answer the SAT-solver has provided.

The following example illustrates a simple reduction. We will use SAT to solve the k-CLIQUE problem. Given a graph of size N and an integer k, k-CLIQUE returns "YES" if there is a totally connected subgraph of size k, "NO" otherwise. We will use the following variables.

$$p_{i,j} = \text{"The } i\text{-th node in the graph is the } j\text{-th node in the clique"}$$

Now we will explain how to construct a CNF such that, if it is satisfiable we can get a k-clique from the graph, and there is no k-clique otherwise. We will have four groups of clauses.

1. Every node in the clique must be at least one of the nodes of the graph. We can encode this clause as

$$p_{1,j} \vee p_{2,j} \vee \ldots \vee p_{N,j}$$

$$\forall j, 1 \leq j \leq k$$

2. Every node in the clique must be at most one of the nodes of the graph. We can encode this clause as

$$\neg p_{i,j} \vee \neg p_{i',j}$$

$$\forall i \forall i', i \neq i'$$

$$\forall j, 1 \leq j \leq k$$

3. Every node in the graph cannot occupy two nodes in the clique.

$$\neg p_{i,j} \vee \neg p_{i,j'}$$

$$\forall j \forall j', j \neq j'$$

4. For every two positions in the clique, if there is no edge connecting their nodes, they cannot both be in the clique.

$$\neg p_{i,j} \vee \neg p_{i',j'}$$

$$\forall i \forall i', i \neq i' \text{ and no edge between nodes } i \text{ and } i'$$

$$\forall j \forall j', j \neq j'$$

Given an instance of k-CLIQUE, we can encode it in CNF form. We use it as an input to a SAT-solver and examine its output. If it returns "unsat", it means that no assignment of values to the variables renders the formula true, thus implying that no clique of k nodes exists. However, if it returns a model for the formula, by observing the ith index of the variables assigned to one we would be able to know which vertices are on the k-clique and which are not.

As we have seen, SAT can be used as a black-box tool to solve any problem that we can encode into a boolean formula. This approach is used by CellRouter. To solve the routing problem, we describe it in terms of a CNF and use a SAT-solver to obtain a solution. There are many ways of creating such a formula and the success of this approach depends to a great extent on how variables are picked and restrictions are codified. However, as stated before, SAT is a supposedly difficult to solve problem. This approach is getting more attention because a lot of work is being done on the field, not only on SAT solving but in constraint programming (with examples such as Satisfiability Modulo Theories [24]). Most modern SAT solvers are based on the DPLL algorithm, a systematic backtracking looking for a satisfiable assignment of the variables. However, a lot of additions and optimization have been added, such as conflict analysis, clause learning, backjumping, random restarts and heuristics. These methods have been proven empirically to be essential in solving large instances of SAT. For some more information on the use of SAT in EDA please refer to [25].

1.5 Conclusions

This chapter has provided an overview of the key concepts this project is based upon. First the VLSI design cycle has been introduced, with special emphasis on the use of EDA tools and algorithms. Then the routing problem has been presented, discussing about different algorithmic approaches that have been used to tackle the problem. Several design considerations that affect the fabrication process have been exposed, as well as a presentation to what a standard cell is and why it is important to VLSI design. Finally, notions on what the boolean satisfaibility problem is and how it can be used to solve different problems have been given. Based on what has been shown, the next chapter will present the original framework this project is aiming to improve.

Chapter 2

CellRouter

As explained in the abstract, the aim of this project is to develop divide-and-conquer strategies on a previously existing framework to route standard cells. This router, called CellRouter, uses a technology-independent and parametrizable approach which can be adapted to different fabrics and rules. It uses a boolean formulation of the problem to find a legal detailed routing of a cell represented by a gridded layout. However, as cells become larger, approaches such as the one this project explores become mandatory to keep SAT formulas tractable. In this section, basic insight on how the CellRouter tool works and necessary vocabulary that will later be extensively used is given.

2.1 The Routing Problem

As explained in Section 1.1, routing and manufacturability-aware design have attracted lots of attention in the last years. This routing tool addresses both issues by considering geometrical regularity for the routing process. As mentioned before, it is not the first time that a boolean formulation of the problem is presented [16, 17]. However, the complexity of the problem restricted the applicability to small cells. Additionally, algorithms based on using regular layout fabrics had already been proposed in [26], but they were specially customized for that fabric and a specific set of design rules.

The router we are working on proposes an algorithmic approach for a generic problem of cell routing, which has the following characteristics.

- Should be independent from the layout templates and the interconnect resources, so that it can be configured with the resources available at

any technology generation.

- Attributes should be allowed for every wire segment.
- Should allow the router to select the best pin locations.
- Should be independent from the set of design rules.
- In the case of unroutable cells, externally connected pins should be allowed.
- A set of recommended design rules to improve yield should be specifiable.
- Wirelength should be a parameter for optimization.

To do so, the CellRouter tool uses an encoding scheme for SAT-based formulas that makes large cells tractable by applying windowing heuristics, as explained in Section 2.3. A formalism to specify gridded design rules and multiple-patterning constraints is provided. CellRouter also uses heuristics for quality improvement (wirelength and recommended design rules) and allows the connection of external pins in case of unroutability.

Graphs are used to represent the gridded routing problem. Every net has a set of terminals that must be connected. Each terminal is represented by a set of vertices. Edges represent wire segments that can be used to connect pairs of vertices. The routing problem is defined as follows.

Find a set of edges that define routes connecting the terminals of each net. The routes must be disjoint (cannot have common vertices) and satisfy a set of design rules.

It is important to realize that the number of possible solutions is finite. It can be reduced to a SAT formula in which a variable is associated to every edge representing the presence or absence of a given signal in that position. To find such a solution with the maximum quality, CellRouter uses two steps.

1. Finding a legal solution that honors the design rules.
2. Improving the solution by iteratively re-routing nets and using quality terms in the cost function.

2.2 Routing Problem Representation

The routing region is represented by a 3D undirected grid graph $G(V, E)$ as depicted in Figure 2.1(a).

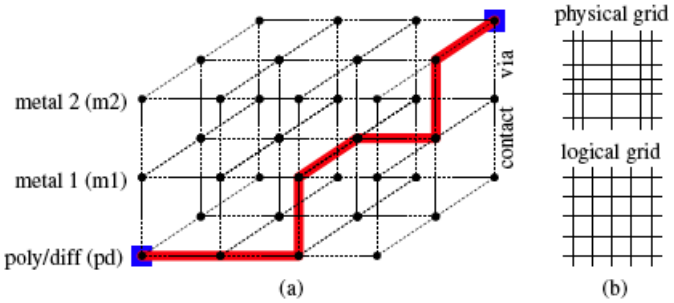

Figure 2.1: (a) Grid model for routing. (b) Physical and logical grid

The vertices have associated integer coordinates in $\{1, ..., W\} \times \{1, ..., L\} \times \{1, ..., H\}$, where W, L and H represent width, length and height. The edges of the graph connect grid points. Notice that the represented physical grid may not have a uniform distribution such as the one shown in the grid as can be seen in Figure 2.1(b).

Every vertex v is denoted by its coordinates $v = (x(v), y(v), z(v))$. In our context, $z(v)$ represents the layer of the layout, thus $z(v) \in \{pd, m1, m2\}$ as shown in Figure 2.1(a). Every edge will be denoted by its endpoints, as in $e(v, u)$. We will also define a *net* $n \subset V$ as a set of grid points, called terminals, that must be connected. A *subnet* will be a pair of terminals of the same net.

A Viewer program is provided in order to see how a grid looks like. Given the description of a grid it shows a 3D representation of it using OpenGL, allowing interaction with the model including zoom and rotation. Figure 2.2 represents an instance of a real routing problem. Each color represents a different net or signal. We can see in the lowest layer some terminals that need to be connected. On the top and bottom of the cell we can see the VDD (voltage, red) and VSS (ground, grey) lines crossing the second layer. On the bottom left side of the screen, a complete list of the signals that appear on the cell is provided.

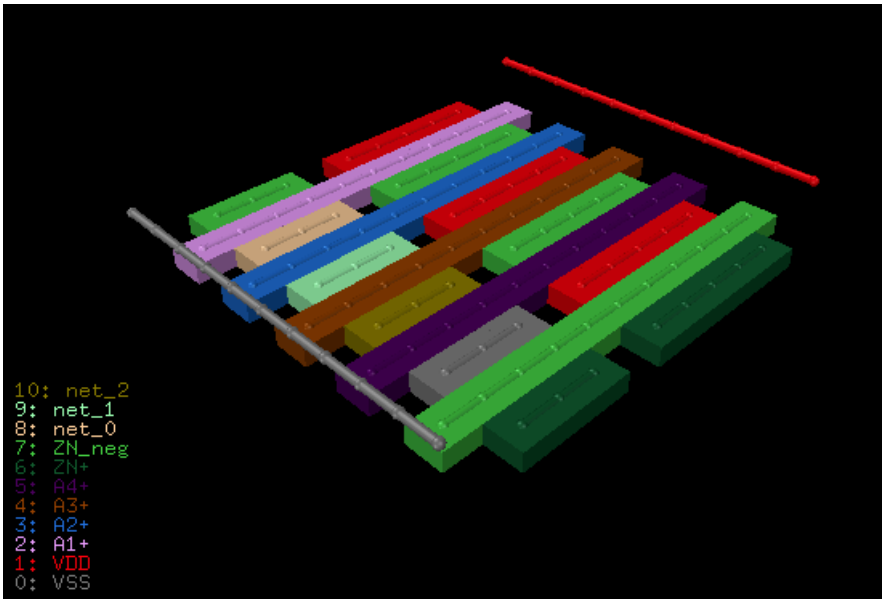

Figure 2.2: Routing grid problem instance

2.3 SAT to Solve the Routing Problem

Now that we have some terminology we can make a broad overview at how CellRouter uses SAT to solve the routing problem. To do so, CellRouter codifies the routing problem to a CNF formula that can be given as an input to a SAT solver. First, some variables to model the problem are needed.

- $\rho(e)$: A variable that represents when edge e is occupied by a wire.
- $\rho(e, n)$: A set of variables that represent the associated net in case e is occupied by a wire.
- $\rho(e, n, s)$: A set of variables that represent the subnets associated to every wire.

Given these variables, the Boolean formula F that represents the problem is as follows.

$$F \equiv C \wedge R \wedge DR$$

Here we will have a brief description of the elements in each of the components of F.

C, Consistency constraints
These clauses ensure the consistency of the formula. For example, make sure that if an edge is associated with a net, such net is occupied by a wire, or that if an edge is associated to some subnet of a net, it is also associated to that net.

R, Routability constraints
This clause set represents the routing constraints for the grid. For example, we must impose that each edge is assigned to at most one net and that two adjacent wires are assigned to the same net.

DR, Design-rules contraints
Finally, these clauses represent constraints imposed by the user-defined set of design rules. Such design rules might impose, for example, that no adjacent vias can be connected to different nets. Extra clauses modeling wire attributes are included among this constrains.

An important idea which is encoded in the form of routability constraints is windowing. Empirically, it has been shown that the route of a two-terminal subnet rarely spans beyond the bounding box determined by the two terminals. Thus, clauses that enforce the variables outside the region to be falsified can also be added. This might imply that no solution is found even if one exists, but it greatly improves the tractability of the problem. The halo parameter indicates how far outside the bounding box a subnet is allowed to expand. Several halo values will be used in the experiments presented in Chapter 5

CellRouter allows for any discrete set of attributes to be binary-encoded and incorporated to the formula. For example, wires might have two different widths (thin and thick) or could be assigned to different masks to comply with some patterning lithography rules. Additional variables with the form $\rho(e, x)$ which represent the presence of attribute x in edge e are then added, as do the necessary clauses to deal with such attributes.

The formula F thus generated is given as an input to the *picosat* SAT solver which will return a satisfying model, if such exists. Given this model, a solution for the original routing problem can be obtained: this is the main goal of the router. In Figure 2.3 we can see a solution to the routing problem in 2.2.

As stated before, among all valid solutions, some have better quality than others. For example, cells with smaller wirelength are preferred. The solution

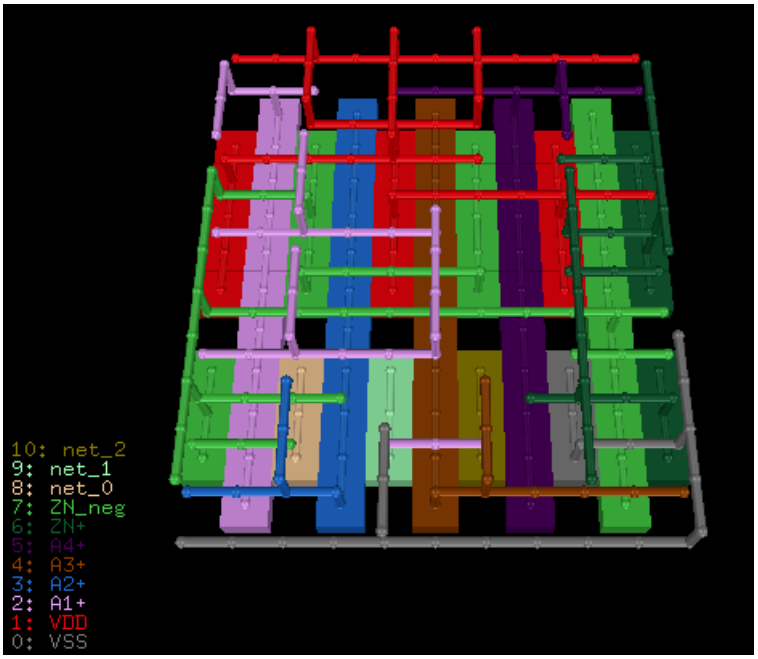

Figure 2.3: First obtained solution for 2.3

proposed by the SAT solver in 2.3 has many redundant wires. CellRouter proposes a heuristic method to make this problem tractable using a mixed integer linear programming engine, *gurobi*. However, this model becomes intractable when dealing with large cells. Large neighborhood search is then used to reduce complexity of the problem in combination with Integer Linear Programming (ILP). The algorithm consists on ripping-up and re-routing nets starting from the basic solution obtained using the SAT solver until no significant improvement is observed. In practice, this takes two rounds of re-routing for each net. This strategy admits variants such as ripping and re-routing more than one net simultaneously; additionally, other aspects such as the ordering of the nets could be considered to search for even better local minima. Figure 2.4 shows an optimized version of the first obtained solution.

For more information on the grid data structure and what the command line interface of CellRouter is, refer to appendixes A and B.

2.4 Results

The CellRouter tool was used to synthesize the Nangate 45nm Open Cell Library, which contains 127 cells. All layouts were checked for design rule

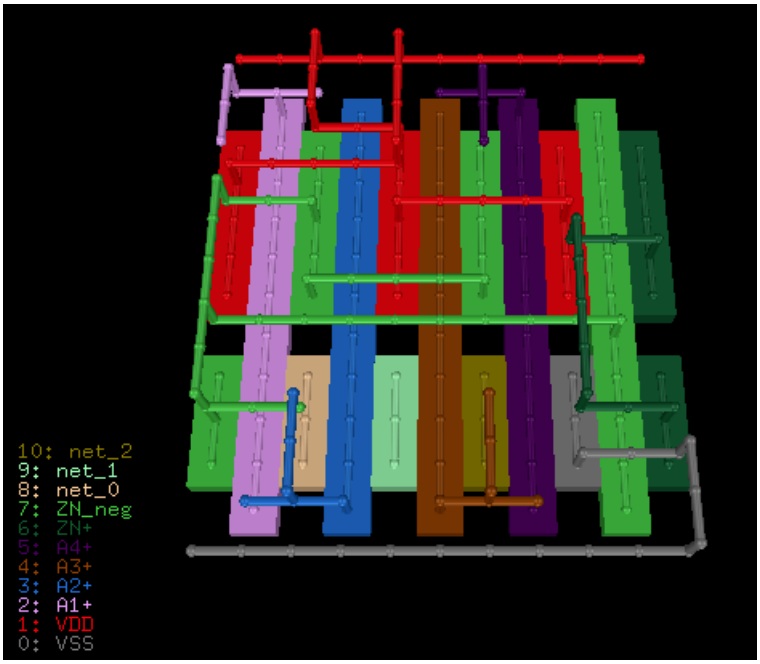

Figure 2.4: Optimized solution for 2.2

correctness and can be found in

http://layout.potipoti.org

The encoding scheme of CellRouter is compared to another SAT formulation of the routing problem presented in [27]. As we can see in Table 2.1, CellRouters' *sparse* encoding outperforms the *dense* encoding that was used in the previous work. Additionally, it is interesting to see how the windowing heuristic has a big impact on the CPU time for routing cells, managing to divide the computation time at the expenses of losing some solutions. All library cells were routed in 1 hour and 5 minutes.

However, we must take into account that CellRouter has been applied to cells of a limited size. What happens when it has to deal with bigger, more complex cells? Given that the tool is based on a SAT-solver, and SAT is a hard problem, as soon as the complexity of the problem scales, it becomes intractable. This project aims to find a way for such hard cells to be routed and, to do so, it uses a technique that is not so new in the field of routing algorithms as we have seen in Chapter 1: The divide-and-conquer approach.

		Sparse (w=2)		Sparse (w=5)		Dense [27]	
Cell	Area	Size	CPU	Size	CPU	Size	CPU
OAI221_X1	5	102	0.1	141	0.1	96	0.1
HA_X1	9	198	0.1	249	0.1	239	29.4
FA_X1	15	521	1.2	624	8.7	486	2959.0
DFFS_X1	21	731	2.0	893	8.1	903	205.4
SDFF_X1	25	657	2.3	1073	7.2	1380	1424.0
SDFFRS_X2	33	1626	15.4	1944	98.1	2679	40 hours

Table 2.1: Results for SAT solving (Size in 10^3 literals, CPU in secs.)

2.5 Conclusions

This chapter has explained the basics of how CellRouter works. First it has more accurately defined what the routing problem for a cell is and how it represents internally the routing grids. Figures of an example of the routing flow have been provided. A basic description of the structure of the SAT formula has also been presented. Finally, origianl times for the CellRouter tool have been provided. Next chapter will focus on how the project uses CellRouter in the process to route bigger and more complex cells, whereas Chapter 4 includes the description of the divide-and-conquer techniques used to route such cells.

Chapter 3

Design

In this chapter, the design of the project will be described. Following some preliminary considerations, multiple iterations were developed in order to look for the best possible solution to the proposed problem. We will overview the flow between the files, CellRouter and the contributions of this project.

As we have seen in the previous chapter, CellRouter is a tool based on SAT that finds valid routings for grids. However, given the nature of SAT, when the problem grows in complexity or size it becomes intractable. The main aim of this project is to help CellRouter dealing with such cases when normal brute-force SAT-solving is not enough. In order to do so, the chosen approach is one that has already shown up in the routing world: the divide-and-conquer strategy.

3.1 Preliminary Study

When interacting with an already existing tool, knowing its ecosystem becomes of great relevance, as the project will need a very close interaction, if not modifying the tools themselves. Figure 3.1 shows the basic workflow in the routing of a standard cell.

The input to the flow is a *.pla* file which contains a linear placement of the nets that must be routed. It is given to the *Gridder*, which is a C++ binary that, with the placement and a *.tpl* template file, outputs a *.grd*, a grid containing the placed nets. The template file contains information related to the geometrical structure of the problem, for example the length of both p and n transistors on the cell. Figure 3.2 shows five possible templates that a

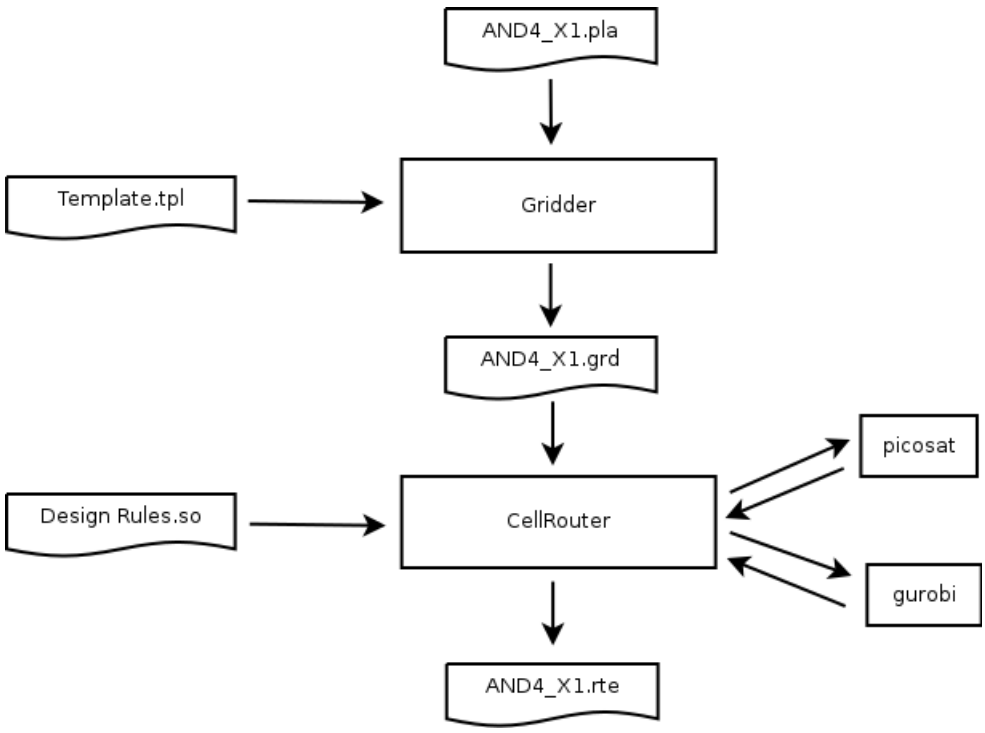

Figure 3.1: CellRouter data flow

given netlist could be adapted to. In fact, the decision of which template to use is crucial, since a routing problem solvable for a template is not necessary solvable for another one. Figure 3.3 shows the result of gridding an AND gate using three different templates. The same placement is respected but the final grid may even vary on vertical height.

The resulting *.grd* file represents the grid that will be routed by the CellRouter. More details about it can be found in appendix A. CellRouter has been implemented in C++ and uses external software: *picosat*, the SAT solver used to find the initial valid routing, and *gurobi*, the mixed-integer linear programming tool used for optimization. Both processes are called internally from CellRouter when needed. In addition to the *.grd*, a file specifying the design rules to be followed by the router is required. It consists of a *.so* library that is dynamically loaded, allowing different rule sets to be used for different routing instances. Given such file and a *.grd*, CellRouter proceeds to find a valid assignment for the SAT formula and to optimize it using gurobi. More information on the interface of CellRouter can be found in appendix B.

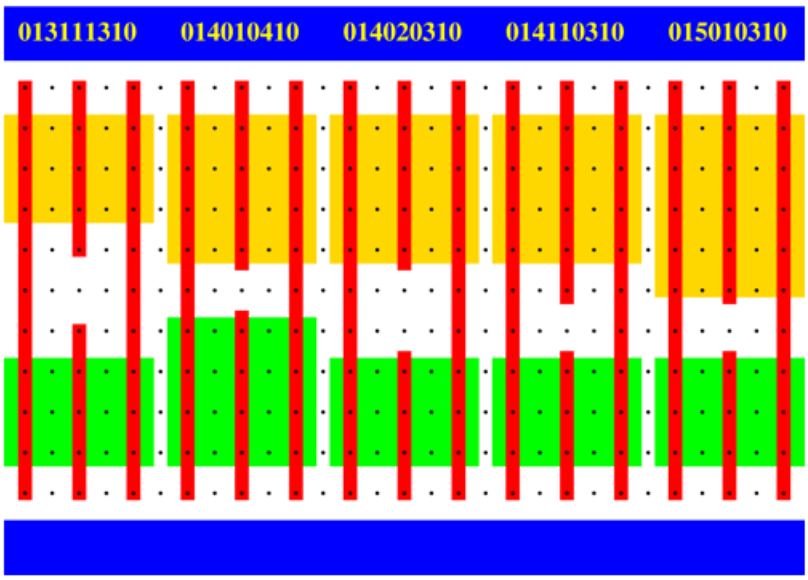

Figure 3.2: Five possible templates

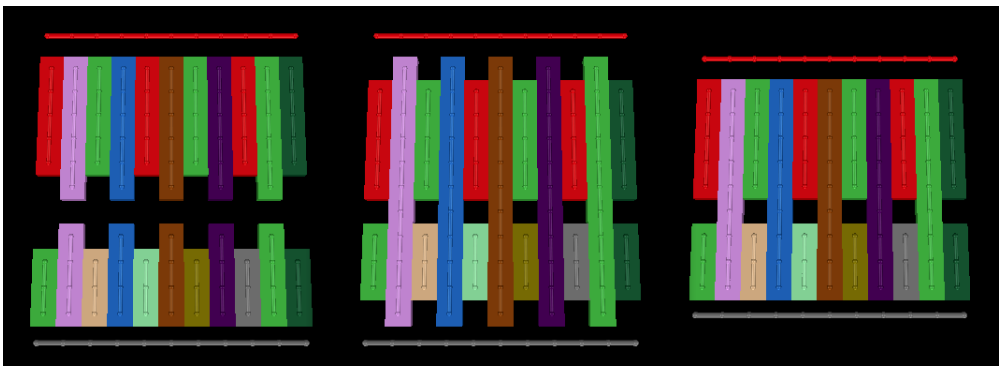

Figure 3.3: Three different griddings for the same placement

It is important to note that the problem of finding a routing for a given grid is a combinatorial optimization problem. The goal is to find the best possible grid in the set of those which are valid routings. The output of running CellRouter does not depend exclusively on the *.grd* input and the specified design rules. A lot of parameters can be modified for CellRouter, either when looking for a solution or during optimization (halo size, number of escapes, heuristic rounds...). As we will see later, the use of divide-and-conquer techniques will make the search space greater. This is a problem since the best configuration to route a given cell does not necessarily generate a good solution for another one. This inherent difficulty must be taken

into account when designing the tool and the experiments.

3.2 Design Decisions

Given that the original tool was developed using C++, the first implementation that was made was a C++ binary called CellDivider. Its aim was to begin interacting with the problem, getting to know the data structures and overall flow. CellDivider received the same input that was before given to CellRouter and directly interacted with the infrastructure that called picosat and gurobi. It interacted directly with many of the classes that were used by the original tool. The grid data structure already existed in C++, so CellDivider focused on interacting with the problem through it. It did the most basic partition one could think of; given a cell as an input such as the one on Figure 3.4, the program generated a new cell that was exactly the left half of the original one (Figure 3.5, left side). The router, called from inside CellDivider, solved the left-hand part (Figure 3.5, right side). When solving a half of the problem, the program did not consider any information of the remainder of the cell. Finally, CellDivider copied the solution onto the original cell, as shown in Figure 3.6, and the partially routed problem was sent to the router to obtain a valid solution for the whole cell.

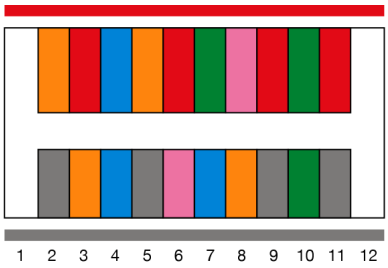

Figure 3.4: Input for CellDivider

When the router receives a grid as an input all the variables are fixed, including those which come from an earlier partial solution. This first version showed how a poorly chosen distribution on a partial routing could lead the total cell to become unsatisfiable. It was tested against a subset of cells obtaining very different results, from cells that were unsatisfiable to cells that were routed in half the original time. From this moment on, the focus was set on obtaining partial solutions that were conscious of the rest of the cell,

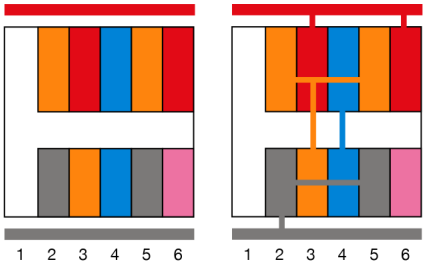

Figure 3.5: Partial problem and solution

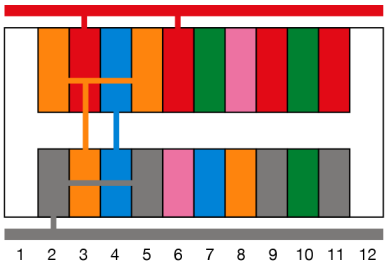

Figure 3.6: Final routing problem

trying to avoid unsatisfiable global situations.

Following this first version, it was decided to use python to continue with the project and to leave CellRouter as it was, interacting with it through the command line and external files instead of modifying the router itself. The idea was to work at a grid level, separating CellDivider from the other parts of the project. Python seemed a good implementation language given that the work CellDivider itself does is not computationally intensive and that much prototyping and testing had to be done. Ipython was used to develop the project; it is a framework that allows using Python in a more interactive fashion, with improved debugging and an enriched web-based editor.

Given that CellDivider was going to be developed using python and work at a grid level, with the grid being a C++ data structure, SWIG was used to interface the class and use it from python. However, many problems arose when trying to create the interface for such a complex class and, finally, the grid data structure was replicated in python. Complete separation from the C++ project was achieved, so the development of CellDivider became independent of the internals of CellRouter. This involved some extra work as the original data structures were modified halfway through the project, which would have been easier if python accessed the C++ classes directly.

The final flow using CellDivider is shown in Figure 3.7. CellDivider reads the cell.grd file, containing the grid to be routed, and a configuration file, which includes the routes of the cells, binaries and the design rules file. Several smaller grid routing problems are created. For each one, CellDivider creates a temp.grd file and writes in it the grid that needs to be routed. Then a CellRouter process is created, which uses temp.grd as an input and routes it independently. When the router ends, the routed grid is saved in a temp.rte file and CellRouter returns the output to CellDivider, including information such as if the cell was routable or not, computation time and other metrics. Finally, CellDivider creates a cell.rte file where the final routed cell is saved if a routing has been found, or announces the opposite otherwise.

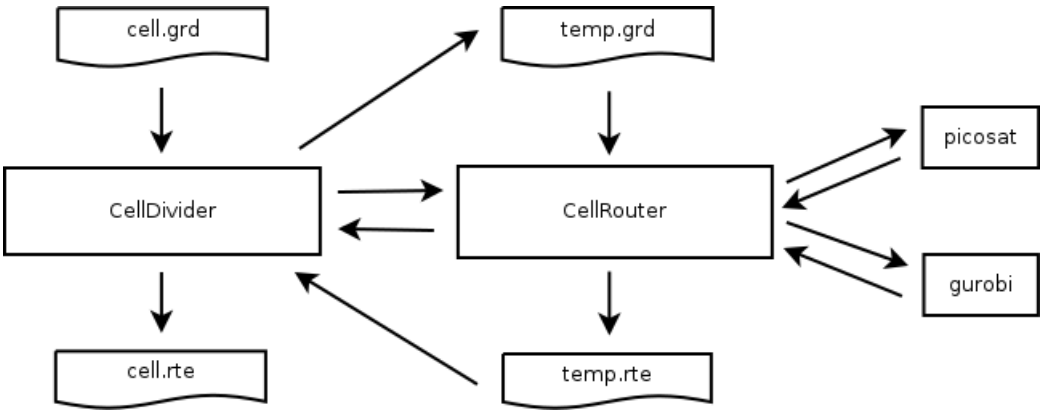

Figure 3.7: CellDivider data flow

3.3 Conclusions

In this chapter we have exposed what the design decisions have been for the development of the divide-and-conquer strategy, including the reasons why python was chosen as a development tool and how the contributions of this project adapt to the already existing framework. Many versions of CellDivider have existed due to the experimental nature of the problem. All of them share the basic functions, which include reading the configuration file, routing a set of cells specified on a given file and calling the Gridder, Cellrouter or Viewer, extracting information from their results. Aside from this common features, every version of CellDivider can be described in terms of their partition algorithm, how to prepare a partial routing, and their meta-

algorithm, how to use the partial problems to obtain a global solution. The main task during the project has been coming up with several algorithms for both cases, as will be explained in next chapter.

Chapter 4

Algorithmic Strategies

In the last chapter we described how the CellDivider tool interacted with the routing framework and the already existing data structures. However, probably the most interesting part is what CellDivider does in order to achieve routing big and complex cells using the divide-and-conquer scheme.

As exposed in Section 1.2, two approaches for divide-and-conquer can be considered: bottom-up and top-down. The last one would imply to first create some general routing connections and then work at a lower level, whereas the first one implies first routing little zones and then expand the routing to bigger ones, such as an entire cell. In this project, the bottom-up methodology seems the most natural approach. A top-down scheme would imply first routing the connections of parts far away in the cell, which would potentially occupy spaces needed for the detailed local routing. Given the characteristics of the tool, a bottom-up approach seems a more easily implementable option. The idea would be to solve parts of a given cell so that finally the whole routing problem becomes simpler than it originally was.

The remainder of the chapter is structured in two sections. Section 4.1 exposes several methods that can be used to route a part of a cell. Section 4.2 explains how to use such partial routings in order to find a valid solution for the whole cell.

4.1 Partition algorithms

A partition algorithm decides how to partially route some part of a cell. As explained before, the most basic approach would be to absolutely ignore what lies on the other sides of the limits when doing a partial routing. This

is dangerous because, as stated before, a bad partial solution could lead the global problem to be unsatisfiable. For example, consider routing the cell in Figure 4.1 by first routing the left half. Figure 4.2 shows what the partition would look like and a possible solution the that partial problem. Note that, on column 4, all horizontal positions are occupied. This is entirely valid on the partial solution as the green node does not need to be connected to any component on the right side. But when the partial solution is incorporated to the total cell and we try to route it, we are unable to find a valid routing because the green net cannot be connected as shown in Figure 4.3.

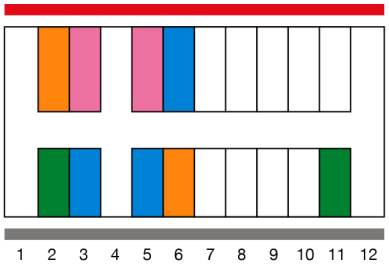

Figure 4.1: Input for CellDivider

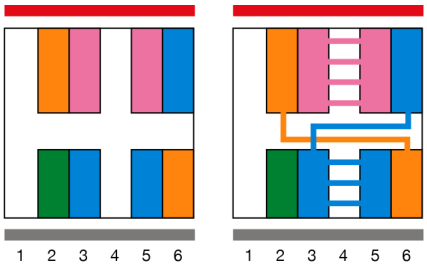

Figure 4.2: Partial problem and solution

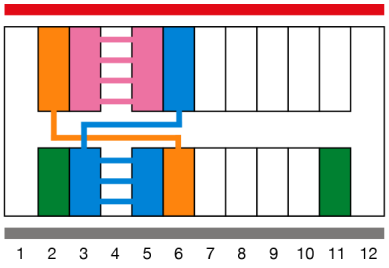

Figure 4.3: Final routing problem

This example perfectly illustrates the situation of a partial routing turning the whole cell unroutable. It must be kept in mind that when CellRouter returns a satisfying model it will probably not be optimal, so high-congested zones may appear. It is also important to notice that the design rules may be very strict, so finding a situation where a partial solution renders the whole problem unsatisfiable has proved to be quite usual. To cope with this problem, the following partitioning approaches try to keep some information on what lies on the parts of the cell that are not being routed in the partial problem.

4.1.1 Realistic Partitioning

Consider a more general form of routing a part of the cell, where we want to solve the routing problem from column n to column m. The realistic partitioning algorithm considers the central part of the cell, from columns n to m, and certain regions at their right and left. The main idea is that any signal which has a terminal on the central zone and another one on any of the side zones should be granted an exit to the terminal outside the central zone. Additionally, if a signal has a terminal in the right and left zone but none in the central zone, it should be imposed that a path crossing the central zone exists.

The left zone ranges from column ini to column n, whereas the right zone ranges from column m to end. ini is the leftmost column where there is a terminal that must be connected to another terminal on either the central or right zone. end is defined the other way around. A temporal cell is created, where the width is $end - ini$. The zone from n to m receives an exact copy of the original cell from n to m. The zone from ini to n and from m to end, however, will only contain the closer terminals to n and m such that they are required to be connected with the central zone or the opposite side. Finally, the positions where the connection pins are considered valid are calculated and the cell can be routed. After the cell is routed, all wires that are not part of a subnet when the solution is copied on the original cell are removed. The function looks as follows.

function REALISTIC PARTITIONING(G grid, n and m column indexes)

Find signals in the center, left and right
$ini \leftarrow 0$
$end \leftarrow G$'s width
$partial\_signals \leftarrow \{\}$
$partial\_terminals \leftarrow \{\}$

for all signal in G **do**
  if signal in left and in either center or right, or viceversa **then**
    Save closer signal appearance to n and m
    Update *ini* and *end* accordingly
  end if
  if signal will be in the partial cell **then**
    Add signal into *partial_signals*
    Add wether it needs an external pin or not into *partial_terminals*
  end if
end for

partial_pins $\leftarrow \{\}$
for all *pin* in G **do**
  if *pin*'s column index betweem *ini* and *end* **then**
    Add *pin* to *partial_pins* substracting *ini* to column index
  end if
end for

$R \leftarrow$ new grid with width $end - ini$
Add *partial_signals*, *partial_terminals* and $partial_pins$ to R

for all *element* (vertex or edge) in R **do**
  original $\leftarrow$ mapping of *element* in G
  if *original* column between n and m **then**
    element $\leftarrow$ signal of *original* in G
  else if *element* on lowest level **then**
    element $\leftarrow$ *locked_signal*
  else
    element $\leftarrow$ *free_signal*
  end if
end for

for all *terminal* needed outside n-m range **do**
  Add *terminal* to R
end for

Save R to a file
Call CellRouter over R
if call is unsat **then**
  return *unsat*
end if
Load solved R from file

for all *element* (vertex or edge) in R **do**
original ← mapping of *element* in G
if *original* column between n and m **then**
original ← signal of *element* in R
end if
end for
Clean G grid
return *routing_time*

end function

For example, we want to route the zone defined by n and m on the cell in Figure 4.4. We need to ensure that both signals on column 6 have an exit to the other parts of the cell where they are required, which are columns 3 and 9. Additionally, on these two columns there is a terminal of the same signal that should cross the central zone. We need to set *ini* to column 3 and *end* to column 9. Now, as explained earlier, the zone between n and m will have all the terminals included and the rest of the partial cell will only have the required terminals, as shown in Figure 4.5.

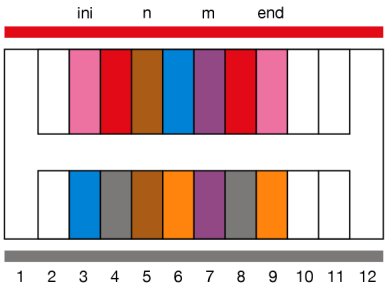

Figure 4.4: Input for CellDivider

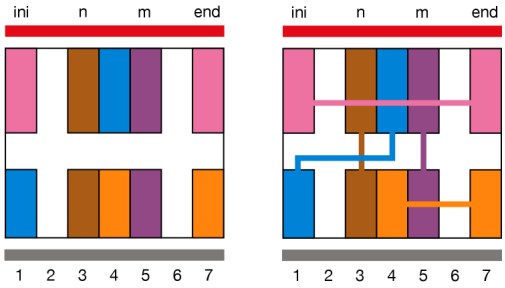

Figure 4.5: Partial problem and solution

After the partial routing is completed, only the signals routed in the range

from column n to column m are copied back to the original cell. All the wires that are not in a path between two terminals of the central zone are also removed. This would be the case of a wire that simply crossed the central region but did not have any terminal in it. Since we only wanted to impose that such a path existed, the routing tool will occupy again those positions if needed. Figure 4.6 shows how the final cell looks like after the partial solution has been added. Notice that all wires that did not begin and end in the central zone have disappeared, but the path they occupied is available.

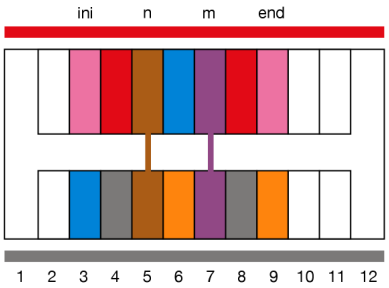

Figure 4.6: Final routing problem

Now let's take the example that was shown before, where a cell was rendered unsatisfiable. If the partial routing is done using realistic partitioning, Figure 4.7 shows how the partial routing problem and solution looks like. Given that there is a signal in column 2 whose closer terminal is in column 11 on the original cell, the partial cell will have to include up until that column. However, since it is the only signal shared between the n-m zone and the rest of the cell, it will be the only terminal from column 6 to column 10 in the partial cell. Finally, Figure 4.8 shows the final routing problem. Notice that now a path for the signal that before rendered the cell unsatisfiable exists. This does not guarantee that now the total cell will have a valid routing, but chances are higher than before.

Notice that a number of variations exist for this partitioning algorithm. For instance, when copying the partial solution to the original cell, all wires from n to m can be copied, not only the ones beginning and ending in that range. Another option would be copying all the wires of the partial solution, regardless of what zone they are in. Another option would be, when preparing the partial cell, including all terminals outside the central zone and not only those strictly required to be routed. It is specially useful when multiple partial routings are performed on a single cell, given that zones that might look empty for a partial solution could have already been occupied by an

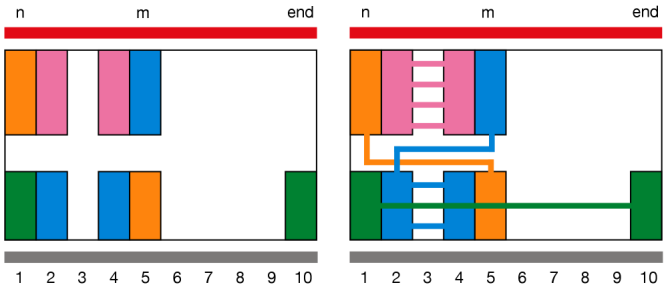

Figure 4.7: Partial problem and solution

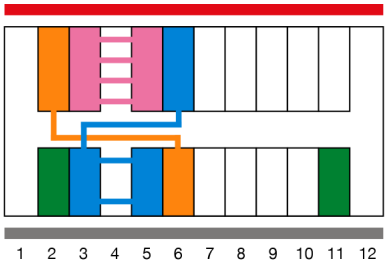

Figure 4.8: Final routing problem

earlier routing.

4.1.2 Boundary-Conscious Partitioning

This partitioning approach is similar to the one described in the previous subsection. Once again we intend to route a cell from column n to a column m. However, the idea is to approximate all terminals outside the routing zone that should be connected with signals that are between the n-th and m-th columns or that should simply cross that zone.

Consider the case of the right part of the zone we want to route. The algorithm will find, for every signal that is both in the right part and outside it, the terminal that is closer to the boundary between the zones. The partial cell can be created when the number of signals to approximate in both sides is known. As in the realistic partitioning approach, the zone between n and m is entirely copied. As for the other parts, respecting the closeness of every terminal to the boundary, fake terminals are added to represent that they must be connected with a terminal in that zone. The pseudocode, which is very similar to the one in previous subsection, is as follows.

function BOUNDARY-CONSCIOUS PARTITIONING(G grid, n and m column indexes)

Find signals in the center, left and right
$partial\_signals \leftarrow \{\}$
$partial\_terminals \leftarrow \{\}$
for all signal in G **do**
  if signal in left and in either center or right, or viceversa **then**
    Save closer signal appearance to n and m
    Decide if placing the terminal on p or n zone if needed
  end if
  if signal will be in the partial cell **then**
    Add signal into $partial\_signals$
    Add wether it needs an external pin or not into $partial\_terminals$
  end if
end for
Calculate how many terminals will be added at left and right
$partial_width \leftarrow m - n + termianls_left + terminals_rigth$

$partial\_pins \leftarrow \{\}$
for all pin in G **do**
  if pin's column index betweem ini and end **then**
    Add pin to $partial\_pins$ substracting ini to column index
  end if
end for

$R \leftarrow$ new grid with width $partial_width$
Add $partial\_signals$, $partial\_terminals$ and $partial_pins$ to R

for all $element$ (vertex or edge) in R **do**
  $original \leftarrow$ mapping of $element$ in G
  if $original$ column between n and m **then**
    $element \leftarrow$ signal of $original$ in G
  else if $element$ on lowest level **then**
    $element \leftarrow locked\_signal$
  else
    $element \leftarrow free\_signal$
  end if
end for

for all $terminal$ needed outside n-m range **do**
  Add $terminal$ to R
end for

```
    Save R to a file
    Call CellRouter over R
    if call is unsat then
        return unsat
    end if
    Load solved R from file

    for all element (vertex or edge) in R do
        original ← mapping of element in G
        if original column between n and m then
            original ← signal of element in R
        end if
    end for
    Clean G grid
    return routing_time
```

end function

The following example illustrates how boundary-conscious partitioning works. Figure 4.9 shows the cell we want to route. We decide to partially route the cell between n and m. The signal on the top part of column 6 must be connected to the one on the top part of column 3 and both parts of column 10. In the partial problem, we approximate those terminals to the boundary of the zone we want to route. On the left part, the signals on columns 2 and 3 will approach column 4. On the right part we will proceed respecting which signal is closer to the m-th column. First comes the signal in column 8, which is already close to the boundary. Following come the terminals on column 10. However, two of them exist at the same distance. In this case, we consider only one of them, the one in the zone which has a lower number of approximated terminals - which is the top zone in this case. Finally, the terminal in column 11 also moves. The partial problem and a possible solution are shown in Figure 4.10.

Figure 4.11 shows how the final routing problem looks like once the partial solution is copied. Notice that, since this method alters the geometry of the original problem even more than when using realistic partitioning, all the wires routed outside the n-m range need to be discarded. Only wires that both end and begin in between the n-th and m-th column are kept.

Boundary-conscious partitioning is useful when we are partially routing big cells that have signals which are potentially far away. We can come back

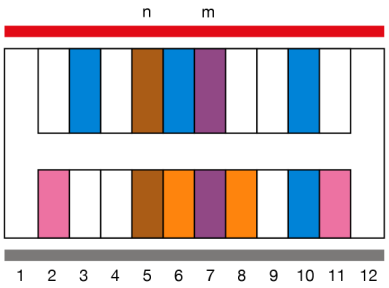

Figure 4.9: Input for CellDivider

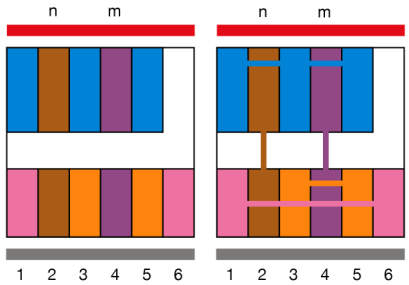

Figure 4.10: Partial Problem and Solution

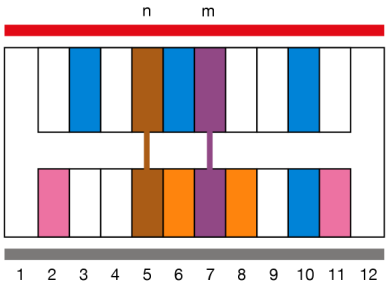

Figure 4.11: Final routing problem

to our example in Figure 4.1. As we saw in Figure 4.7, a lot of empty columns were added in the partial solution. Using boundary-conscious partitioning, all that empty zone can be spared as shown in Figure 4.12. The result when copying the information back to the originall cell is the same as the one shown in Figure 4.8. Avoiding all that white cell space simplifies the boolean formula, thus reducing the amount of memory and computation time that SAT will need to obtain a partial solution.

As in the case of realistic partitioning, many variations exist. For example, we could decide to add terminals in both the high and the low parts

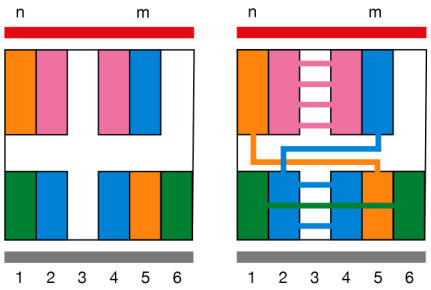

Figure 4.12: Partial problem and solution

of the cell, not only on one of them, when they appear on both areas in the original cell. This partitioning algorithm admits a *pad* parameter which specifies the number of columns that will be left empty among the approximated terminals. It is used to reduce congestion in the cell at the expenses of making it bigger. The examples of this section have considered a value of 0, but Figure 4.13 shows how the partial problem on Figure 4.12 would look like with a different pad value.

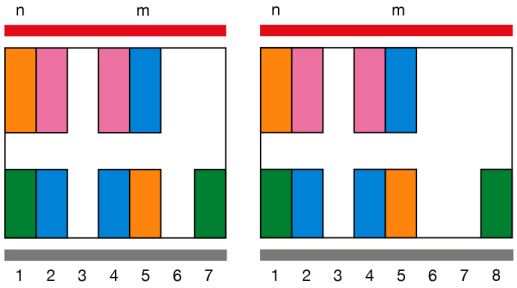

Figure 4.13: Partial problem with $pad = 1$ and $pad = 2$

Independently of what the partial routing algorithm is, after the results of a partial routing are copied back to the cell, an additional optimization can be performed. It consists on eliminating any wire that is not part of a path between two terminals. These wire segments are normally added because of the design rules or as connections to pins. They potentially occupy a space that is needed for a global routing to be found, and maybe another configuration which also respects the design rules and the pin connectivity can be reached in another routing stage. It is important to note, however, that such action must only be performed if the wires that are eliminated will be added again in a posterior routing of the same area, so that the final routing is correct.

This section has introduced the two most important approaches to cell partitioning. Other methods, composed of any of these algorithms and other variations, have been used. One of them, which was used in scan routing, is described in the next section.

4.2 Meta-algorithms for Cell Routing

Given a complete cell we want to route, meta-algorithms define how to use partition algorithms, either the ones described before, variations or completely different ones, in order to find a solution which is valid for the whole cell. Most CellDivider versions use a bottom-up approach consisting on routing parts of the cell and finally trying to route the entire cell with the information provided by the partial solutions. However, other approaches were a final global routing is not needed have also been explored.

It is important to remark that the number of possible meta-algorithms is enormous. Not only can they differ in the partitioning algorithms they use, but also in where and when they use them. The described meta-algorithms should be regarded as a methodology for cell routing using partial solutions. In every partial solution to the problem, parameters such as the halo have to be chosen. Any wrong decision could lead to an unsatisfiable solution and, given the case, how the meta-algorithm reacts is also of great relevance.

4.2.1 2-Cell Routing

This meta-algorithm was the first one used during the development of the project and is presented to illustrate the number of variables that can affect the execution of CellDivider. It consists on dividing the cell by the half and solving one or both parts to find a global solution. Now, which sides should be partially routed, the left one, the right one or both? Using realistic partitioning, boundary-conscious partitioning or a variation of any of them? What halo should be used when routing the partial problems? And when looking for a global routing? Even if this algorithm is very simple, this approach shows how big the experiment exploration can become. The number of parameters and decision combinations will only grow as the meta-algorithm's complexity increases.

However, 2-Cell routing is very limited. For example, when cells grow big, it can be interesting to divide the cell in more than two parts. This led to *N-Cell Routing*, a variation of this algorithm in which the cell was divided in an arbitrary number of parts. For example, a cell such as the one in Figure 4.14 could be divided in three parts. A valid strategy would be to route the two side parts of the cell as shown in Figure 4.15, which shows those parts already routed using boundary-conscious partitioning. Finally, we obtain the final problem and only routing it is left as seen on 4.16. Routing only the central part of the cell would also have been a valid approach. During development it was seen that partially routing the whole cell before doing the total routing would lead to unsatisfiability in many of the cases. However this method proves to be useful when the cell has a known physical structure such as concatenated cells.

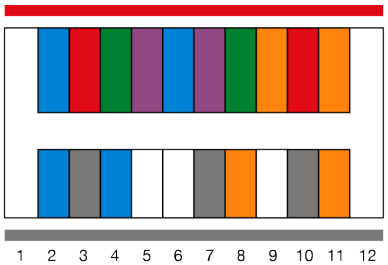

Figure 4.14: Input for CellDivider

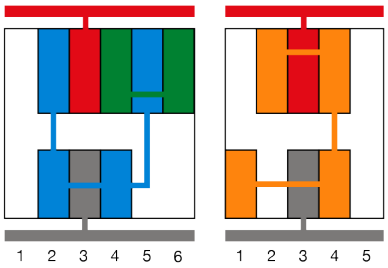

Figure 4.15: Partial problems and solutions

After some experimentation, blindly partitioning the cell in equal chunks still seemed quite a basic approach. Additionally, this meta-algorithm needs for a total route of the entire cell as a last step. The meta-algorithms that will be described in the following subsections try to take such observations into account. Many variations of them are also possible but have not been explored.

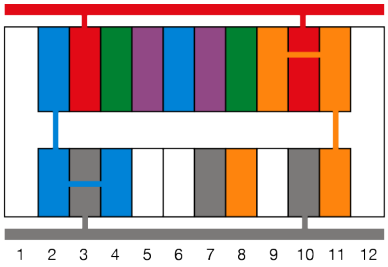

Figure 4.16: Final routing problem

4.2.2 Congestion-Driven Routing

When deciding which part of the cell to route before, first solving the most congested part seemed a good approach. Given that we want to avoid a final unsatisfiable result, the idea is dealing with the hard parts in the first instance so that later only easy to route parts are left.

The idea to measure congestion is simple: The number of subnets that must cross each column. This can be calculated easily by scanning the cell from side to side and considering the first and last appearance of each signal. Each one of them will add congestion to every column between the first and the last appearance. A wire of said signal will have to cross each one of these columns in order to connect all the terminals of the net. Figure 4.17 shows a cell with the congestion vale of each column above it.

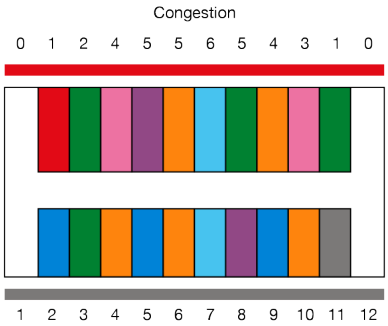

Figure 4.17: Congestion of the columns in a cell

Congestion-driven routing can be applied in a wide range of ways. Here follow some of the versions that were developed.

Window Routing

This method uses a window of a fixed length w, which is either a number given with the input or some measure relative to the width (for example, a half or a third of its width). It proceeds by partially routing the zone of the cell with width w such that the sum of the congestion of the columns in the zone is maximal. It usually falls somewhere in the middle of the cell, which usually is the most congested part. Figure 4.18 shows what part of the cell would be partially routed in the one presented above. The implementation was abandoned to try other strategies and did not allow for multiple overlapping partial routings at the time. The pseudocode is as follows.

function WINDOW ROUTING(G grid, w window size)

$time \leftarrow 0$
$histogram \leftarrow histogram(G)$
$zone \leftarrow congested_zone(histogram, w)$

$partial\_route \leftarrow route\_range(G, range)$
if $partial\_route.state = SAT$ **then**
$time \leftarrow time + partial\_route.time$
else
return $unsat$
end if

$total\_route \leftarrow route(G)$
if $total\_route.state = SAT$ **then**
return $time + total\_route.time$
else
return $unsat$
end if
end function

Zone Routing

This method used the congestion metric in a different way. It incorporated a function that, given the congestion value of each column, returned a set of column ranges that should be partially routed. The one used considered those zones were congestion was maximum and only on unit under the maximum, but different strategies can be applied. Those zones have a variable length which can vary for different zones in the same execution. From the cells used during experimentation, the partially routed area amounted from a 14% to a 95% of the cell. For example, if applied to the cell introduced in this subsection on

Figure 4.17, the zone that would be routed is between the 5th and 8th columns as shown in Figure 4.19, but on bigger cells it would produce several partial routings. This would be a version in pseudocode.

```
function ZONE ROUTING(G grid)

    time ← 0
    histogram ← histogram(G)
    zones ← congested_zones(histogram)

    for all zone in zones do
        partial_route ← route_range(G, zone.begin, zone.end)
        if partial_route.state = SAT then
            time ← time + partial_route.time
        else
            return unsat
        end if
    end for
    total_route ← route(G)
    if total_route.state = SAT then
        return time + total_route.time
    else
        return unsat
    end if
end function
```

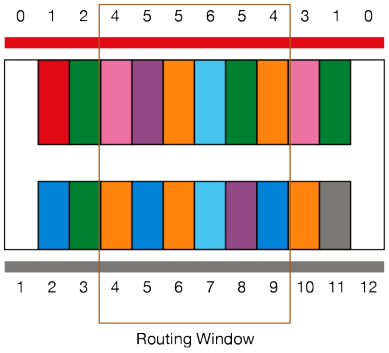

Figure 4.18: Window routing example

No more work was done in congestion-driven routing when development of the last meta-algorithm began. All of these methods require of a final routing step where the whole cell is included, which was something that we

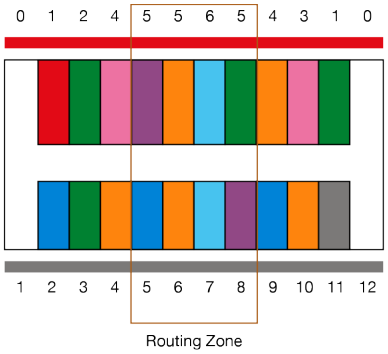

Figure 4.19: Zone routing example

wanted to avoid as will be seen in next sub-section.

4.2.3 Scan Routing

The aim of scan routing is to route the cell without the need of a final step where the whole cell is codified into a SAT formula. In order to do so, the cell is divided in a given number of contiguous chunks which are then routed from left to right. After routing the ith part of the cell, all subnets which should both begin and end in the ith chunk or before are already routed. This way, after routing the final division of the cell, the whole cell has been routed. In order to do so, considering we want to route from column n to column m, the partitioning algorithm used is as follows.

For the part at the left of this zone we use the approach of the realistic partition algorithm with the variant where all the terminals and wires of the already routed zone are included on the partial problem. In scan routing, we consider that the left part of a partial routing has already been routed and incorporated to the final solution. If this is not regarded when routing the next part, it is possible that the new partial solution would need to assign certain signals to positions were a different wire has already been defined. The boundary-conscious partitioning approach can not be used in this part, given that it does not respect the geometry of the cell and does easily lead to unsatisfiable solutions. When obtaining a partial result, all wires at the left of n will be considered as fixed and all wires at the right of m will be discarded, given that we only wanted to impose that a path from the zone we already routed and some signals at the right of column m existed. Additionally, some cleaning of not necessary wires in the left part is performed before a partial routing, knowing that the router will put them back again if

they are really necessary.

On the right part of the zone we want to route we will use the approximation routing approach. This way we can potentially save a lot of blank space when looking for the partial solution. It is not so important to keep the original geometry, given that the part in the right will not be fixed after the partial routing.

The partial solutions used in scan routing, thus, take advantage of both of the approaches introduced in Section 4.1. After all the parts of the cell have been routed from left to right, a solution for the routing of the whole cell is obtained. Signals that need to have a connection pin only require it on the first partial routing where they appear, given that from then on all the other terminals of the net will be connected to the subnets of the first part, which are connected to the pin. Partial routings are optimized in order to raise the chances of getting a satisfying assignment and, at the same time, yield better final solutions. However, as happens with the other methods, scan routing not finding a solution does not imply it does not exist. The pseudocode looks as follows.

function SCAN ROUTING(G grid, n number of parts)

```
    time ← 0
    ranges ← boundaries of the n parts of G
    for all range in ranges ordered from left to right do
        partial_route ← route_range(G, range)
        if partial_route.state = SAT then
            time ← time + partial_route.time
        else
            return unsat
        end if
    end for
end function
```

The following example will illustrate how the algorithm works. Consider the cell in Figure 4.20.

We decide to use the scan algorithm on that cell in three parts. The cell is divided into the following ranges: $(1, 8)$, $(9, 15)$ and $(16, 22)$. Figure 4.21 shows the routing of the first part. In this case, the closest terminal of all signals which are to be connected to a net in the $(1, 8)$ range are approximated to the boundary. Figure 4.22 shows the state of the whole cell after

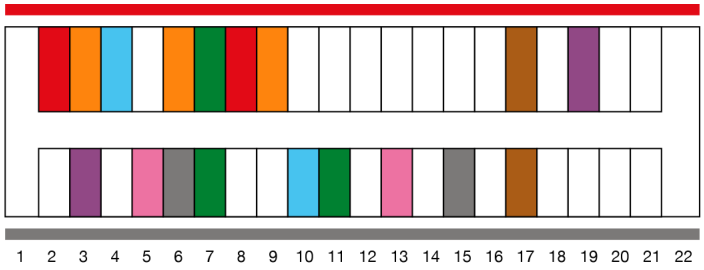

Figure 4.20: Input for CellDivider

the partial routing has been added.

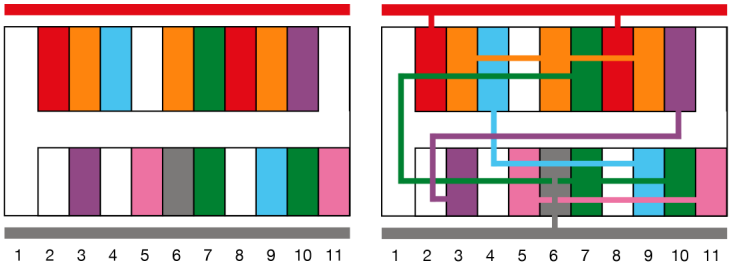

Figure 4.21: Partial routing, first part

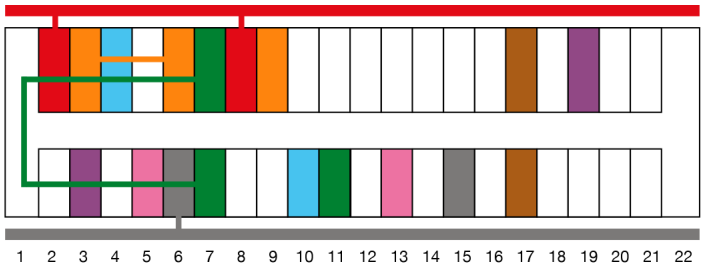

Figure 4.22: Cell after first partial routing

Now it is time to route the second range as seen in Figure 4.23. This time only one signal will be come closer on the right side. However, a big part of the already routed cell will be included in the partial routing, as some of the terminals in $(9, 15)$ have to be routed with terminals in the already routed part. Figure 4.24 shows the state of the cell after this second routing.

The last step is to route the third part of the cell. After the last partial solution is computed, no more routing is be needed. Figure 4.25 shows

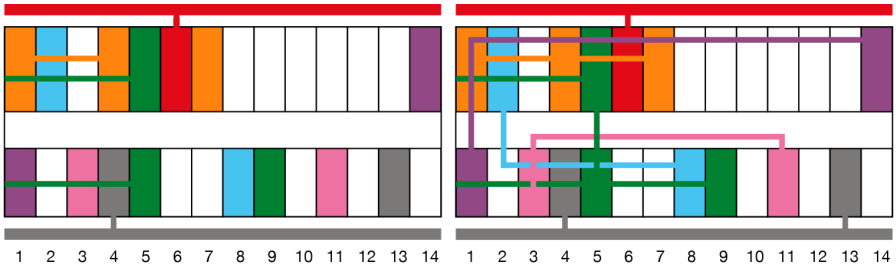

Figure 4.23: Partial routing, second part

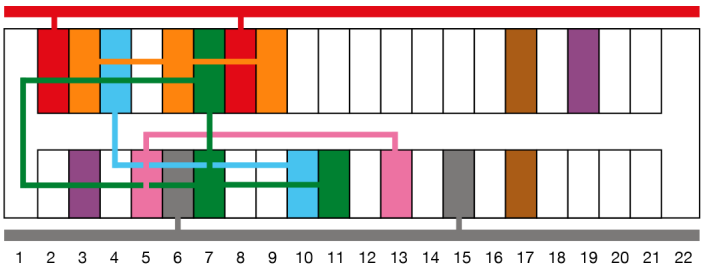

Figure 4.24: Cell after second partial routing

the solution for the last partial routing and how it is all integrated in the cell.

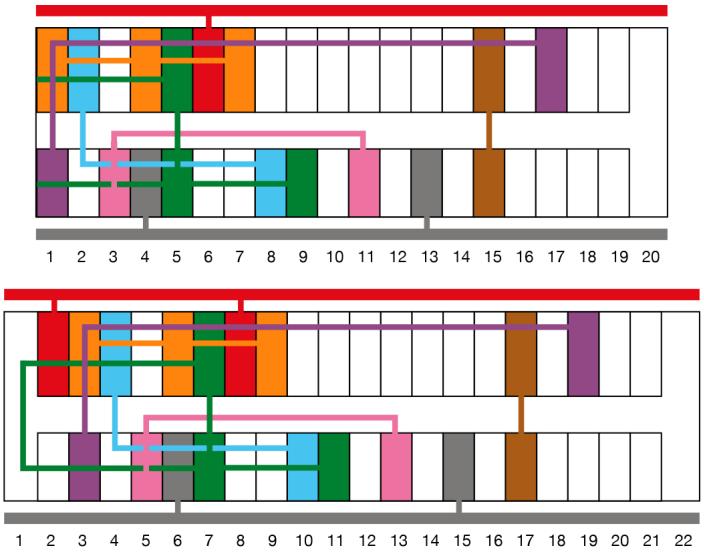

Figure 4.25: Third partial routing and final solution

However, the scan algorithm needs an extra tweaking on the partitioning algorithm. Notice how the terminals on column 6 have been routed twice.

This is because when doing the second partial routing, as seen in Figure 4.23, the connection of those terminals disappeared beyond the left boundary. The following addition is included to solve this problem. When the left border is known, the algorithm checks if in that column more than one wire for the same signal exists. If it is the case and they are not connected, the shortest path between those positions is added to the partial problem. Given that this implies expanding the cell by the left side, all positions on the left that are not part of a shortest path are blocked. Figure 4.26 shows how the second partial routing problem is. The black zone represents positions that can not be modified but have been added in order to let the router know that both terminals on the original column 6 are already connected. Notice that avoiding duplicate connections is important, since it could mean the difference between a satisfiable and an unsatisfiable cell.

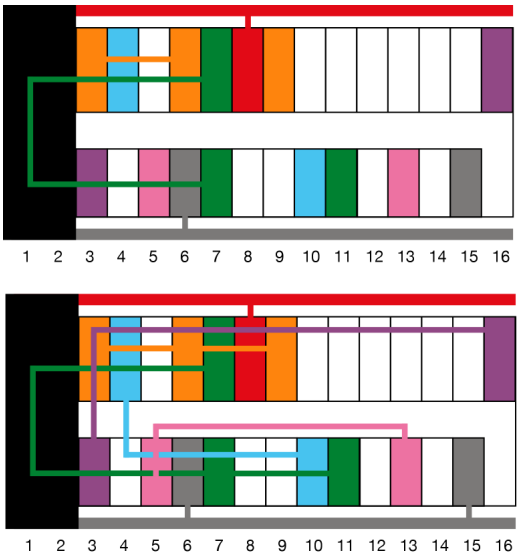

Figure 4.26: Second partial routing with blocks and solution

As we wanted, the scan approach has an important benefit over the others: it does not need a total routing over the cell. This is very interesting because, when cells get big and the number of signals increases, the generation of the formula can become expensive. Besides, never having the whole formula at once allows to route cells using less maximum RAM memory. However, this method also turns out to produce many unsatisfiable cells, specially when they are very congested.

4.3 Conclusions

This chapter has presented the algorithms that have been used to both route a part of a cell and to use these partial routing to obtain a global valid solution. Many more options and variants could have been explored but these seemed the ones that could lead to a better final solution. The following section focuses on the results of several experiments done using most of the algorithms that have been introduced in this section on several kinds of cells.

Chapter 5

Results

This chapter will show some tests that have been done using two different cell sets. The first section is devoted to the use of the cells provided by the Nangate Open Cell Library, whereas the second section uses the *CatLib Cell Library*. In both of them, tables regarding different CellDivider versions are presented and some conclusions are drawn. Given the number of possible experiments that could have been conducted, the ones presented have been selected to show the most important results of the work that has been done. A summary of the project's general conclusions will be presented in the next chapter.

Much testing was done during the development of the project. Each time a new version or variant was produced, it was tested with a set of representative cells of the Nangate library, about which we will explain more in Section 5.1. Those experiments were generally done in the development laptop, given that the size of the Nangate instances is manageable using a normal computer. The experiments that will be shown in this chapter have been conducted on the LSI Cluster, using two Dell PowerEdge R410 using an Intel Xeon X5660 processor, with 12 cores and 2,8 Ghz frequency, and equipped with 96GB of RAM memory each. Using the cluster was advantageous over running the tests on the laptop for several reasons. Both nodes are equipped with high-end processors, so the computational time is lower. The main advantage is that, given the number of cores, many routing instances were solved in parallel. Besides, the memory of the system also mattered; there were routing instances that required more than 4GB of memory, which the laptop used for development could not provide.

5.1 Nangate Open Cell Library

The first set is the Nangate Open Cell Library, which is the same that was used to test CellRouter. It is an open-source standard cell library provided for research purposes. It includes several functions, for example buffers, combinational gates and flip-flops, all of which come in different drive strength variants. When referring to the name of a cell, first will come the name of the function followed by the drive strength (for example, an AND4_X1 gate is a 4-input AND with the first level of drive strength).

A subset of the cells in the Nangate library has been selected for the purpose of making these experiments. The aim of this project is to help with big and complex cells. The cells that will be used are those that fall into one of such categories. The big ones cells are AND4_X4, AOI22_X4, BUF_X32, INV_X32, NOR4_X4, OAI22_X4 and OR4_X4, whereas the complex cells, which are the most congested ones, are DFF_X1, DFFR_X1, DFFS_X1, DFFRS_X1, FA_X1, SDFF_X1, SDFFR_X1, SDFFS_X1 and SDFFRS_X1.

Table 5.1 includes the routing time for all those cells using a halo of 6. We consider the case where the cell has not been optimized using heuristics and also when one or two optimization rounds have been done. The more heuristic optimization rounds are conducted, the higher the computational time becomes. The area is computed as the number of columns of the cell.

A first interesting experiment consists on simply dividing the cell in a half, routing the part on the right and then solving the whole cell together. Table 5.2 shows the obtained results. The numbers in the top row indicate the partial routing halo, the number of partial routing optimization rounds, final routing halo and the final routing optimization rounds. For example, the 6 0 12 0 column means that the right half of the cell has been routed with halo 6, has undergone 0 rounds of optimization, and that the final routing of the cell has been done using routing 12 also without any optimization round. The No Opt and 1 Round columns are the original times from Table 5.1. A value of 0 indicates that the cell was found to be unroutable.

It is important to notice the difference on behavior of the two kinds of cells, the big and the complex ones. Most of the cells are found to be routable in the first group, even though there are exceptions such as some cases in the AOI22_X4, OAI22_X4 and OR4_X4 gates. However, in most of these cells there is no significant reduction in the amount of CPU time requiered to solve the cell, either in just routing it or obtaining a result with one round

Cell	Area	No Optimization	1 Round	2 Rounds
AND4_X4	12	3,83	9,45	30,84
AOI22_X4	16	6,26	40,20	90,06
BUF_X32	24	33,03	155,20	290,97
DFF_X1	17	80,11	87,41	109,51
DFFR_X1	19	228,69	237,75	296,88
DFFS_X1	20	38,66	49,98	67,08
DFFRS_X1	24	150,58	175,56	221,56
FA_X1	15	30,11	35,28	45,11
INV_X32	32	13,37	58,56	107,15
NOR4_X4	16	5,64	23,21	54,40
OAI22_X4	16	6,36	13,64	49,27
OR4_X4	12	4,28	7,00	29,15
SDFF_X1	24	60,56	80,98	109,37
SDFFR_X1	27	40,94	68,33	99,13
SDFFS_X1	25	441,60	473,08	590,88
SDFFRS_X1	30	310,45	365,86	459,83

Table 5.1: Time (s) used to solve Nangate cells

Cell	No Opt	6 0 6 0	6 0 15 0	1 Round	6 1 6 1	6 1 15 1
AND4_X4	3,83	6,15	6,36	9,45	26,48	22,25
AOI22_X4	6,26	7,64	7,94	40,20	0	28,59
BUF_X32	33,03	25,62	25,62	155,20	119,2	176,8
DFF_X1	80,11	0	0	87,41	0	0
DFFR_X1	228,69	0	0	237,75	0	0
DFFS_X1	38,66	0	0	49,98	43,92	184,81
DFFRS_X1	150,58	0	0	175,56	0	0
FA_X1	30,11	12,68	27,33	35,28	0	0
INV_X32	13,37	12,32	17,77	58,56	42,7	60,58
NOR4_X4	5,64	6,36	9,08	23,21	22,94	28,34
OAI22_X4	6,36	0	9,63	13,64	0	30,21
OR4_X4	4,28	0	0	7,00	15,51	20,69
SDFF_X1	60,56	0	0	80,98	0	0
SDFFR_X1	40,94	38,7	97,4	68,33	0	0
SDFFS_X1	441,60	81,29	981,93	473,08	221,01	1242,59
SDFFRS_X1	310,45	0	0	365,86	0	0

Table 5.2: Time (s) used to solve Nangate cells using the 2-Cell routing

of optimization. Notice how the versions with more allowed halo on the final routing find more valid solutions than the others. Similarly, when optimization is done to the partial routing, the total cell can switch from unroutable from routable, as in the case of OR4_X4.

The case for the congested cells is different. Most of them happen to be unroutable using the 2-Cell approach. This is because the partial solution makes some decisions that do not allow a total routing to be found. However, when routed, some impressive time reductions are also achieved. In the case of FA_X1, it is routed close to three times faster. An even more impressive result is the one for SDFFS_X1, which took 441,6 seconds origianlly and can be routed with merely 81,29 seconds. Still most of the times the output happens to be unsatisfiable. Even when adding more halo to the final routing or optimizing to ease the process some routable results are lost, as happens with SDFFR_X1 and FA_X1.

It is unclear whether or not adding more optimization rounds leads to more satisfiable cells in the case of congested cells. When a partial solution is copied to the original cell, the parts in the boundary are the less congested ones. More optimization rounds are not meaningful to this areas. However, the sole fact that an optimized result may vary the position of the wires explains why some move from routable to unroutable as the number of cleaning rounds increases.

In general, we can see that using the 2-Cell routing schema leads to very good results in some cases but, in many others, fails to find a solution or does not reduce the computational time.

Table 5.3 shows the routing time in seconds using window congestion-driven routing. The No Opt column represents the original routing times without optimization. The numbers after "Window" represent the halo that was used for the partial routing and for the final total routing.

The result is that most of the cells happen to be unsatisfiable. This can be partly solved by using a greater halo when routing the whole cell in the last step, but then the computation times rise too much. However, studying this case was interesting due to the potential in the use of congestion metrics to decide which parts to route. As explained before, the congestion-driven versions were left partway through development to center the effort in the scan algorithm, whose results follow now.

Cell	No Opt	Window, 6 6	Window, 6 12
AND4_X4	3,83	13,09	15,22
AOI22_X4	6,26	0	54,45
BUF_X32	33,03	0	0
DFF_X1	80,11	0	0
DFFR_X1	228,69	0	0
DFFS_X1	38,66	69,11	74,4
DFFRS_X1	150,58	173,93	1094,21
FA_X1	30,11	47,35	73,76
INV_X32	13,37	0	0
NOR4_X4	5,64	0	38,68
OAI22_X4	6,36	0	39,21
OR4_X4	4,28	0	0
SDFF_X1	60,56	0	48,68
SDFFR_X1	40,94	0	0
SDFFS_X1	441,60	0	0
SDFFRS_X1	310,45	0	0

Table 5.3: Time (s) used to solve Nangate cells using window congestion-driven routing

Table 5.4 shows the results of using the scan meta-algorithm, partitioning in two parts. The first number above each column indicates the halo that was used when doing each routing and the second one indicates the number of optimization rounds each part underwent. The "No Opt" column represents the original routing time and the "1 Round" column represents the original time for routing and doing 1 round of optimization.

As happenes with the case presented before, a large number of cells turns out to be unsatisfiable for the last routing step. We can observe that most of the big cells are routable but no gain in performance is obtained except in the case of AOI22_X4 and BUF_X32, where time has dropped to a half in the first case and to two thirds in the second case.

It is interesting to focus again on the more complex cells. Once again, in most of the configurations we happen to find unsatisfiable final problems. However, when a satisfiable solution is found, it is faster than when using normal routing on the cells. This is the case of DFF_X1, FA_X1, SDFF_X1 and SDFFR_X1. Each of these cells has been routed using from three quarters to a mere third of the original routing time. In some of the cases, not

Cell	No Opt	6 0	15 0	1 Round	6 1	15 1
AND4_X4	3,83	0	0	9,45	15,49	21,01
AOI22_X4	6,26	11,59	8,58	40,20	24,18	19,47
BUF_X32	33,03	28,69	25,07	155,20	111,18	166,64
DFF_X1	80,11	0	0	87,41	0	51,34
DFFR_X1	228,69	0	0	237,75	0	0
DFFS_X1	38,66	0	0	49,98	0	0
DFFRS_X1	150,58	0	0	175,56	0	0
FA_X1	30,11	0	0	35,28	20,48	30,73
INV_X32	13,37	10,23	14,85	58,56	50,56	80,32
NOR4_X4	5,64	0	8,37	23,21	0	25,19
OAI22_X4	6,36	0	8,21	13,64	0	25,55
OR4_X4	4,28	0	7,06	7,00	0	0
SDFF_X1	60,56	23,21	0	80,98	0	0
SDFFR_X1	40,94	0	0	68,33	30,86	0
SDFFS_X1	441,60	0	0	473,08	0	0
SDFFRS_X1	310,45	0	0	365,86	0	0

Table 5.4: Time (s) used to solve Nangate cells using scan routing

only has the computational time been reduced compared to the version where optimization is done, but even when compared to simply routing the cell.

Experimentation with this subset of cells, which is representative of the Nangate cells that we want to tackle in this project, has provided interesting results. On one hand we have learned how thin the barrier between a solution leading to unsatisfiable global routings and one leading to an overall valid solutions is. On the other hand, we see that the divide-and-conquer approach has achieved much time reduction in the case of big Nangate cells, whereas it shows to be powerful on the most congested cells when it manages to find a valid solution. Next section explores what happens with cells which are bigger than the ones offered by the Nangate library.

5.2 CatLib Cell Library

The second set of cells consists of the concatenation of Nangate cells which have been grouped under the name of *CatLib*. They have been created on purpose to test several properties of the divide-and-conquer algorithms given

that we wanted to test the tool on cells bigger than those offered by the Nangate library.

In order to do so, *CellCat*, a basic cell concatenation tool, has been developed. It outputs the concatenation of a given number of instances of the cell. When choosing which cells should be on CatLib we wanted to focus on two kind of cells again, depending on whether they were heavily congested or not. An analysis of the whole Nangate library was conducted to determine which cells would be useful to generate bigger cells with the lowest congestion possible. To select these cells, both the mean number of subnets that crossed each column and the total size of the cell were considered. Finally, NOR4_X4 and OAI22_X4 where chosen. In the case of the congested cells, given that the time of solving each cell by the original tool was known, those which proved to be hard were selected: The FA_X1 full adder cell and various flip-flops. When referring to a CatLib cell, the nomenclature NAME_N will indicate the cell that was concatenated in NAME and the times it was included in N. For exmaple, the FA_X1_3 cell is the concatenation of three full adder cells.

Finally, CatLib cells can be divided into three groups: Combinational cells, full adders and flip-flops. The first case simply considers the concatenation of some combinational cells (*COMB* in Figure 5.1) which share up to one signal. These should prove to be the easiest to route given that close to no congestion should exist in the region between cells.

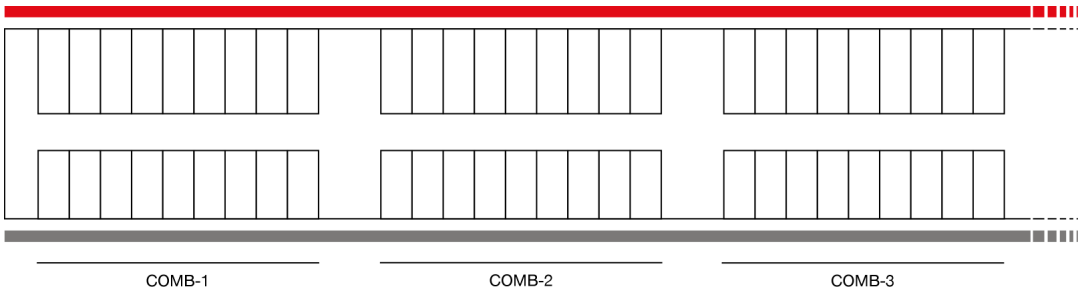

Figure 5.1: Concatenation of combinational cells

In the second case, where 1-bit full adders are concatenated, the carry out signal of a given full adder has to be connected with the carry in of the next full adder, with the exception of the first carry in and the last carry out which are terminals of the final cell. In Figure 5.2 it can be seen how a routing of a n-bit full adder might look like.

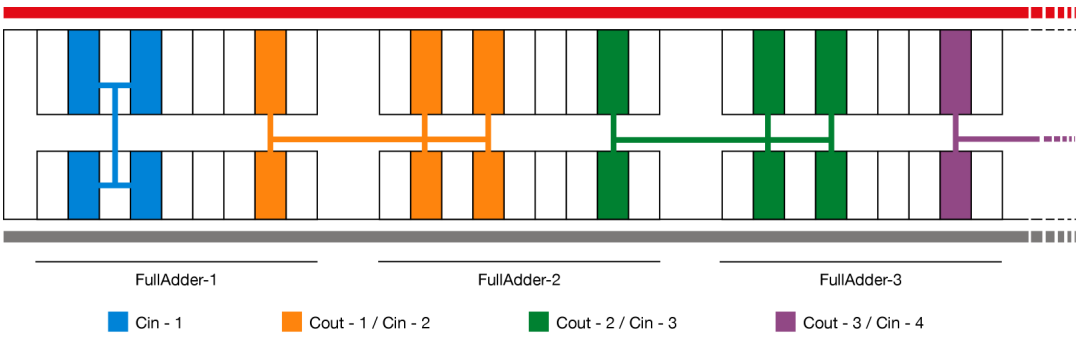

Figure 5.2: Concatenation of full adders cells

The last case is the concatenation of 1-bit flip-flops, generating a n-bit flip-flop cell. This time we do not need to carry signals from one region to the next one, but we need to share signals across the whole cell. The most basic flip-flop only needs to share the clock signal, but up to five signals need to cross the whole cell in the case of scan flip-flop with set and reset, as is the case of the SDFFRS cells. In Figure 5.3, the concatenation with a possible routing of reset flip-flops is shown.

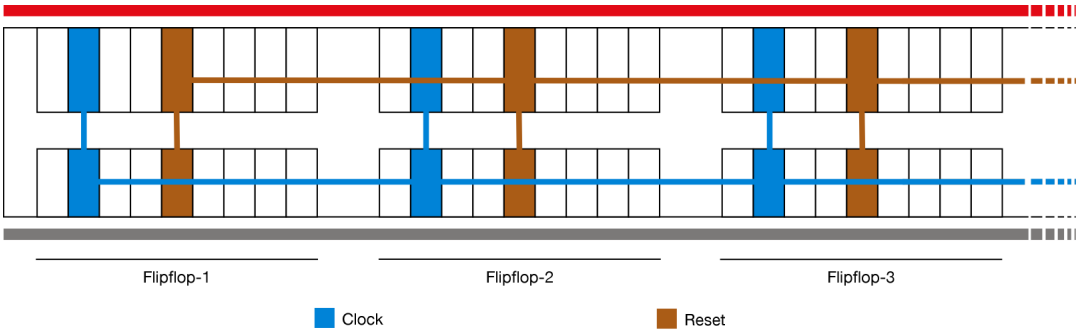

Figure 5.3: Concatenation of reset flip-flops

These cells were routed in the cluster using various halo sizes. Table 5.5 shows the routing time in seconds for the adders and flip-flops using a halo of 2. The numbers on the columns indicate how many times each cell was concatenated. Originally no more than the concatenation of 4 cells was used, given the enormous time it took to solve smaller cases, except for the case of NOR4_X4 and OAI22_X4, where up to 8 cells were concatenated. Table 5.6 shows the routing time in seconds and the maximum consumed memory in GBytes for these two cells.

Cell	1	2	3	4
FA_X1	17,05	101,95	316,96	1440,18
DFF_X2	10,4	74,14	221,16	726,14
DFFS_X1	22,25	289,59	2644,04	14138,55
DFFS_X2	14,38	398,74	2018,35	8542,4

Table 5.5: Time (s) used to solve concatenated adders and flip-flop

Cell	1	2	3	4	5	6	7	8
NOR4_X4	4,8	21,6	71,53	127,23	295,46	482,79	695,27	962,64
	0,19	0,48	1,1	1,94	3,47	5,35	7,66	10,62
OAI22_X4	6,8	35,78	92,41	213,17	381,42	597,23	984,24	1407,86
	0,19	0,57	1,72	3,63	6,38	7,97	14,16	21,35

Table 5.6: Time (s) and memory (GB) used to solve concatenated cells

Additionally, another kind of cell is considered. It consists not on the concatenation, but the fusion of several cells having some signals in common. These gates can share their inputs, outputs or both. Figure 5.4 represents the schematic view of how such a cell could be. This cell, named *HAX*, has been used to test how CellDivider would act in front of the combination of arbitrary different combinational cells. It has 134 columns and 51 different signals. Its purpose is solely to illustrate what happens with a cell that has no regular structure, such as concatenated cells, and which has several components that share signals or must be crossed by other signals.

5.2.1 Combinational Cells

When dealing with the combinational cells, the scan routing algorithm proved to work very well. Each cell was divided into the number of original gates it contained and they were routed one after another. Given that no signal is shared among them, these routings can be done very easily and with close to no extra congestion. Table 5.7 shows the time values that were obtained. The number on each column represents how many cells were concatenated in each case.

As it can be easily seen, they follow a linear relation with the number of cells that get routed. Another interesting result comes in terms of used memory. When routing all the gates no more than 300Mbytes were used, which is negligible compared to how much memory (up to 21 GB) was used by the biggest cells when routing them directly.

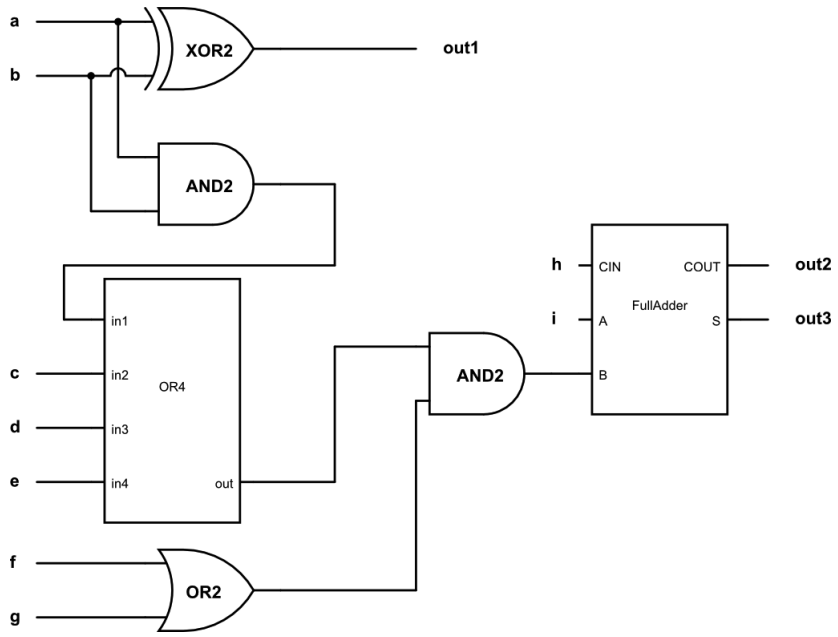

Figure 5.4: HAX gate - Schematic view

Cell	1	2	3	4	5	6	7	8
NOR4_X4	8,27	16,36	24,43	33,51	41,9	50,7	59,42	68,77
OAI22_X4	9,56	18,94	28,45	37,86	47,41	58,19	68,87	77,68

Table 5.7: Concatenated combinational cells - Time (s) used

These are good results, but are applied to a very restricted and simple case. Consider now the combinational cells where each cell shares a signal with the following one. This case is a little more complex because, now, the parts are not so independent from each other. NORN_X4 is a gate composed of the concatenation of NOR4_X4 gates sharing a signal, as described before. Table 5.8 includes the time in seconds for finding a valid solution and additionally performing an optimization round with CellRouter. It also contains the time used for the scan algorithm to find a valid routing for the cell using halo 6. It must be kept in mind that scan algorithm outputs already optimized cells given that it performs optimization in every partial routing. It can be seen how, despite performing worse in the case of the smaller cells, scan routing does a better job when they become bigger. When dividing the cell with 8 NORN gates we obtain an unsat result with halo 6 as shown in the table, but by using a halo of 10 it can be routed in 500 seconds.

Method	1	2	3	4	5	6	7	8
Route	4,74	20,81	65,33	122,89	230,73	427,72	636,21	931,03
Route and optimize	20,84	144,92	184,59	410,05	713,03	1320,29	2056,7	2704,96
Scan Algorithm	24,8	52,39	112,65	149,16	231,24	265,28	278,91	unsat

Table 5.8: Concatenated NORN_X4 gates - Time (s) used

Figure 5.5 shows a graphic displaying the values of the table. The number on the column represents the number of concatenated NORN_X4 cells. It can be seen how, despite performing worse in the case of the smaller cells, scan routing does a better job when they become bigger.

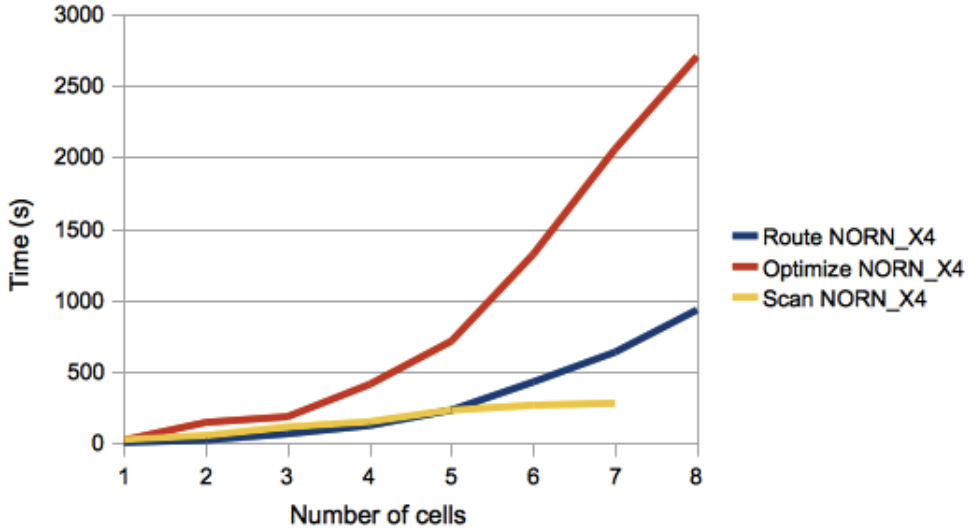

Figure 5.5: NORN_X4 gate - Solving time comparison

5.2.2 Full adders

The case for concatenated full adders is more complicated. Not only is a signal shared with each full adder on both sides, but also the full adder standard cell is a very congested one. When experimenting with the scan algorithm, in many of the cases the result was unsat. This is because some partial routings did a bad choice and the algorithm was not able to find a valid solution.

However, some positive results were achieved. With a halo of 15 and partitioning the cell in the same number of concatenated full adders that it

Method	1	2	3	4	5	6
Route	17,05	101,95	316,96	1440,18	2127,56	4985,79
Route and optimize	24,62	150,27	545,79	1636,00	2430,75	6697,23
Scan Algorithm	39,42	92,33	163,84	257,54	395,02	unsat

Table 5.9: Concatenated FA_X1 gates - Time (s) used

contained, the results are as shown in Table 5.9. The number on the column represents the number of concatenated FA_X1 cells. Each partial solution used two optimization rounds so that the probability of a finding a valid assignment raised.

The trend of all the methods can be seen in Figure 5.6. Note that the execution time of the scan routing version outperforms the direct routing from 2 concatenated full adders. When routing with different halos, most of the times the result happened to be non routable. The number of unsatisfiable final results is very high and several combinations of halo and number of divisions have to be explored to finally route the full adder cells.

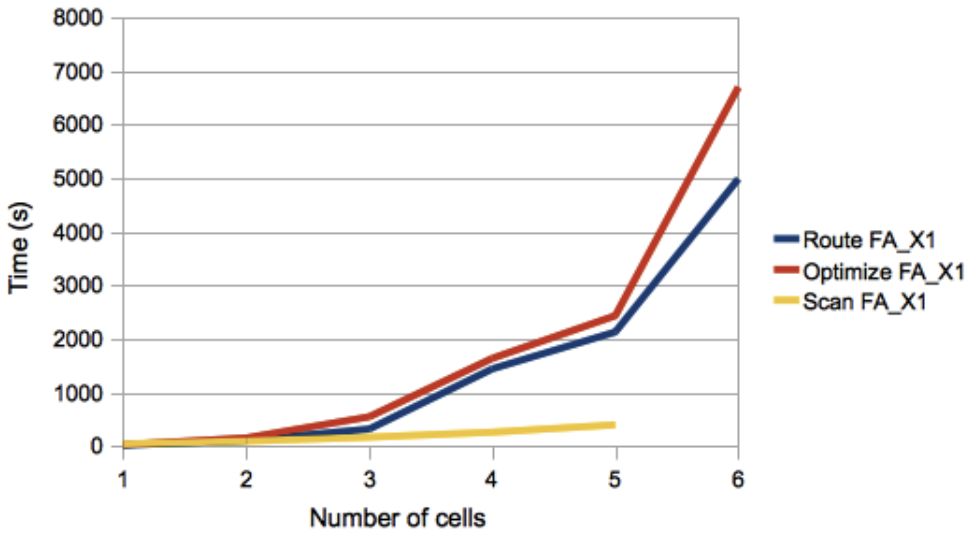

Figure 5.6: FA_X1_N gate - Solving time comparison

Method	1	2	3	4
Route	10,4	74,14	221,16	724,14
Route and optimize	16,11	130,47	483,13	1324,17
Scan Algorithm	28,68	131,36	301,13	476,69

Table 5.10: Concatenated DFF_X2 gates - Time (s) used

5.2.3 Filp-flops

The flip-flop cells are the most difficult to route. Results with flip-flop concatenations have reflected this clearly. These cells share from 1 signal, in the case of the DFF cells, to 5 signals, in the case of the SDFFRS cells. The more signals each concatenated cell has to share with the others, the more complex is for the complete cell to be routed. No cell where its components shared more than 2 elements was routable. In fact, considering only the concatenation of two flip-flop cells, the only ones that were routed are DFF_X2, DFFS_X1, DFFS_X2 and DFFR_X2. Even when trying to route them using different halos and algorithms, hardly any of them gave satisfiable solutions. In the case of DFF_X2, it was routed with the scan algorithm using halo 50, up to DFF_X2_4. The results can be seen on Table 5.10. The number on the column represents the number of concatenated DFF_X2 cells.

As happened with the case of the full adders, a lot of unsatisfiable results appear, specially when the number of divisions done to the cell is big. Aside from the particular case of DFF_X2 and the concatenation of two DFFS cells, no satisfying results for the concatenated flip-flops were found.

5.2.4 HAX Gate

This cell is used as an example of an unstructured big and not very congested cell. Each original part of the cell shares from none to two signals with the parts next to it. Using a halo of 2 it can be routed in 208,29 seconds, having a total wirelength of 1784 wire segments. Applying one round of optimization, the computational time ascends to 511,81 seconds but the wirelength is reduced to 874, less than a half. Table 5.11 shows the results of routing it using the scan algorithm with halos 2 and 6, partitioning the cell in 2, 3 or 4 parts.

When partitioning the cell in 4 parts it is routed in half the time that was used before. However, the wirelength is of 904 wire segments, which is

	2 parts	3 parts	4 parts
Halo 2	301,6	unsat	107,72
Halo 6	341,98	254,96	135,37

Table 5.11: HAX gate - Time (s) used

comparable to the normal routing using one round of optimization, and has been obtained 5 times faster.

We can observe that when partitioning the cell into 3 parts using a low halo we obtain an unsatisfiable solution. The same happens when routing it with more than 5 parts. This is because of the same reason that caused unsatisfiability when dealing with the concatenated cells: congestion on the zone of the cut. The division in 4 parts cuts the cell in pieces which are not as highly congested as the ones on the other division possibilities.

5.3 Conclusions

As a conclusion, we can say that the divide-and-conquer scheme works well for many cells but, the more congested they become, the harder it is to find a valid global routing. However, when it is found, it is up to many times faster than it was using other methods such as directly using the complete boolean formula. In the next chapter, conclusions on the whole project will be given and possible future work lines will be discussed.

Chapter 6

Conclusions

This chapter summarizes the conclusions that have been drawn during the development of this project, most of which have already been mentioned in other parts of the report. Additionally, possible future lines of work are given. It also includes a cost study of the project.

6.1 Final Conclusions

Since the beginning, that partial solutions not being modified in successive routings meant discarding solutions that would have been found otherwise was the major nuisance to the development of the project. We focused on finding methods for the partial solutions to be intelligent enough so that satisfiability was preserved as much as possible. Advances have been done in this sense but the tool still has trouble in the case of congested cells.

In the case of the Nangate cells, not much improvement has been observed. These cells are relatively small and can be solved in tractable time. The ones that were found satisfiable by CellDivider did not usually present a reduction on computational time except on the case of the congested ones, which were difficult to solve but, when routed, it was usually done using a fraction of the original time.

The cells in the CatLib library also reacted differently depending on whether they were big or complex. In the first case, mainly the combinational gates, a great improvement has been found in terms of both computing time and memory usage. In the latter, the case of the full adder and flip-flop cells, the difficulties imposed by congestion have proved to be very hard to

overcome. The HAX gate have presented very interesting results on how a cell without a clear structure can be routed in a fraction of the original time. As seen on the results chapter, when CellDivider was able to route the cells, in general it meant saving a lot of computational time.

The use of divide and conquer techniques that this project presents has helped solving cells that were much harder to solve before. However, when the division is done in a congested place of the cell, chances that the router will not find a valid global solution are high. In the end there is a trade off between valid solutions and computation time, as it happens when using halo to route a cell: it can discard valid solutions but it speeds up many others.

The goal of this project was to make big and complex cells become tractable using a SAT-based already existing framework. As a conclusion, much advance has been done but still a lot of work must be done in order to allow the divide-and-conquer method to discard less satisfiable solutions.

6.2 Future Work

From what has been done on this project several ideas for future work arise. Many of them are based on the idea of finding more cells to be satisfaible.

The first one would be to look for a method such that previous partial routings are not imposed, but only suggested to the router. We would avoid a number of unsatisfiable results arising from poorly chosen partial solutions. This would probably require a more close interaction with the C++ parts of the project, since CellDivider only works at the grid abstraction level.

It would also be interesting to combine the ideas of congestion-driven routing and scan routing so that the zones the scan router divides the cell into are not a given fraction of the cell, but those with the highest congestion. This way, the boundaries between regions would have the number of signals as low as possible. When scan-routing a cell using the zones where the number of signals are minimal as part boundaries, the chances that it will be unsatisfiable decrease and a valid global solution will probably be found.

In the end, any work that reduces the number of unsatisfiable results would be welcomed. As it has been seen in the conclusions, the vast majority of times that cells were routable using CellDivider, it used less time and memory. The preservation of satisfaibility is the basic key in the divide-and-

conquer strategy.

Additionally, this project has dealt with single-height standard cells. However, other geometrical dispositions such as double-height cells and 3D cells also exist. It would be interesting to expand the ideas of this work and find ways to apply the methods proposed in such cells.

6.3 Cost Study

To calculate the cost of the project we will consider two aspects. One will be the cost of the work done. The second is the cost of the equipment that has been used to develop the project. On the work costs:

Month	**Work done**	**Man-hours**
October 2012	Knowing problem. Environment.	30 h.
November 2012		40 h.
December 2012	C++ Porting.	40 h.
January 2013	Python Porting.	40 h.
February 2013	Cluster. Part Routing. Concatenation.	60 h.
March 2013		80 h.
April 2012		90 h.
May 2013	Meta-algorithms. Experiments.	120 h.
June 2013	Experiments. Project Report.	100 h.
		600 h.

Table 6.1: Time study

Considering a salary of about 25 euros per man hour, the total human cost of the project would be of 15000€.

As for the tools used during the project:

Tool	**Cost**
Laptop	1000€
LSI Cluster	331,35€
Software tools	Free
Total tool cost	**1331,35€**

Table 6.2: Tool cost study

We consider the estimated cost of the laptop that was used for development and the cost of using the LSI Cluster for the experiments and running tests during the project. 705 hours of cluster computation were used, which at a price of 0,47 euros per hour adds up to 331,35€.

When considering all costs together, the final cost calculation can be seen in the following table.

	Cost
Engineering costs	15000€
Tool costs	1331,35€
Total project cost	**16331,35€**

Table 6.3: Total costs

Bibliography

[1] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics (magazine)*, 1965.

[2] Willard V. Quine. The problem of simplyfing truth functions. *American Mathematical Monthly*, 59(8):521–531, 1952.

[3] Edward J. McCluskey. Minimization of boolean functions. *The Bell Systems Technical Journal*, 35(5):1417–1444, 1956.

[4] Richard L. Rudell. Multiple-valued logic minimization for pla synthesis. *Memorandum No. UCB/ERL M86/65*, 1986.

[5] Ralph H.J.M. Otten. Efficient floor plan optimization. In *Proceedings of International Conference on Computer Design*, pages 499–503, 1983.

[6] Yao-Wen Chang and Kwang-Ting Cheng, editors. *Electronic Design Automation: Synthesis, Verification, and Test*. Morgan Kaufmann, 2009.

[7] Brian W. Kernighan and Shen Lin. An efficient heuristic procedure for partitoning graphs. *Bell System Technical Journal*, 49(2):291–307, 1979.

[8] Sao-Jie Chen and Chung-Kuan Cheng. Tutorial on vlsi partitioning. *VLSI Design*, 2000.

[9] A. E. Dunlop and Brian W. Kernighan. A procedure for placement of standard-cell vlsi circuit. *IEEE Transactions of Circuits and Systems*, 4(1):92–98, 1985.

[10] K. M. Hall. An r-dimensional quadratic placement algorithm. *Management Science*, 17(3):219–229, 1970.

[11] Meng-Kai Hsu. Routability-driven analytical placement for mixed-size circuit designs. *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 80–84, 2011.

[12] Chris Chu. *Electronic Design Automation: Synthesis, Verification, and Test*, chapter 11, Placement. Morgan Kaufmann, 2009.

[13] C. Y. Lee. An algorithm for path connection and its application. *IRE. Trans. on Electronic Computer*, 10:346–365, 1961.

[14] K. Mikami and K. Tabuchi. A computer program for optimal routing of printed circuit connectors. In *Proc. Int. Federation for Information Processing*, pages 1475–1478, 1986.

[15] P. E. Hart, N. J. Nilsson, , and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Trans. on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[16] B. Taylor and L. Pielaggi. Exact combinatorial optimization methods for physical design of regular logic bricks. In *Proc. ACM/IEEE Design Automation Conference*, pages 344–349, 2007.

[17] W. Hung, X. Song, T. Kam, L. Cheng, and G. Yang. Routability checking for three-dimensional architectures. *IEEE Transactions on VLSI Systems*, 12(12):1371–1374, 2004.

[18] M. Cho, K. Lu, K. Yuan, and D. Z. Pan. Boxrouter 2.0: Architecture and implementation of a hybrid and robust global router. *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 503–508, 2007.

[19] Charles J. Alpert, Dinesh P. Mehta, and Sachin S. Sapatnekar, editors. *Handbook of Algorithms for Physical Design Automation*. CRC Press, 2009.

[20] R. F. Pease and S. Y. Chou. Lithography and other patterning techniques for future electronics. In *Proceedings of the IEEE*, volume 96, pages 248–270, Feb 2008.

[21] W. Maly, Y.-W. Li, and M. Marek-Sadowska. Opc-free and minimally irregular ic design style. In *Proc. ACM/IEEE Design Automation Conference*, pages 954–957, 2007.

[22] G.S. Tseitin. On the complexity of derivation in propositional calculus. *Structures in Constructive Mathematics and Mathematical Logic, Part II, Seminars in Mathematics*, pages 115–125, 1968.

[23] Stephen Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, pages 151–158, 1971.

[24] Roberto Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving sat and sat modulo theories: from an abstract davis-putnam-logemann-loveland procedure to dpll(t). *Journal of the ACM*, 53(6):937–977, 2006.

[25] Joo P. Marques-Silva and Karem A. Sakallah. Boolean satisfiability in electronic design automation. In *Proceedings of the 37th Annual Design Automation Conference*, pages 175–180, 2000.

[26] N. Ryzhenko and S. Burns. Physical synthesis onto layout fabric with regular diffusion and polysilicon geometries. In *Proc. ACM/IEEE Design Automation Conference*, pages 83–88, 2011.

[27] B. Taylor. Automated layout of regular fabric bricks. Master's thesis, Carneige Mellon University, 2005.

Glossary

BFS Breadth-First Search. 6, 7

CNF Conjunctive Normal Form. 17–19, 24

DAG Directed Acyclic Graph. 2

DNF Disjunctive Normal Form. 18

EDA Electronic Design Automation. 1, 2, 5, 15, 18, 20

FPGA Field Programmable Gate Array. 3

HDL Hardware Description Language. 1

IC Integrated Circuit. 1, 5, 14, 15

ILP Integer Linear Programming. 26

RTL Register-Transfer Level. 1, 2

SAT Boolean Satisfiability Problem. 15, 17, 18

VLSI Very Large System Integration. 1, 11, 20

Appendix A

Grid Format

In this annex the format of the grid files is described. They are kept in a plain text file with the *.grd* or the *.rte* extension, depending on whether or not the cell has already been routed. The file consists of a series of headers followed by the information indicated in such header. The sections of the file are as follows.

Title
: Title of the grid.

Sizes
: One line with several sizes of the grid, including

  - Width.
  - Length.
  - Height.
  - Number of signals.
  - Number of properties.
  - Number of attributes.
  - Number of iopins.

Signals
: The list with the name of the signals included in the grid, on for each line.

Terminals
: A boolean indicating if such signal is a terminal or not, on for each line.

Iopins
: One line with the coordinates of all positions where iopins are considered legal.

Attributes
: The list with the name of the attributes included in the grid, on for each line.

Properties
: A list of properties of the grid, each line including its name and value.

Grid
: The actual values of the grid points. Every line represents a vertex in the grid. For a given vertex, the signal present in said vertex and all edges of that vertex is represented with the index of the signal in the signal list. -1 indicates the position is free and -2 indicates the position is locked. In the case of attributes being present on the grid, they will also be expressed for every vertex and edge right after the corresponding signal.

Below comes a reduced example of a .grd file corresponding to an AND4 gate.

```
TITLE AND4_X1
SIZES
13 11 3 11 3 0 78
SIGNALS
VSS
VDD
A1
A2
A3
A4
ZN
ZN_neg
net_0
net_1
net_2
TERMINALS
0
0
1
1
1
1
1
```

```
0
0
0
0
IOPINS
0 0 2   1 0 2   2 0 2   3 0 2   4 0 2   5 0 2   6 0 2   7 0 2
    8 0 2   9 0 2   10 0 2   11 0 2   12 0 2   0 2 2   1 2 2
    2 2 2   3 2 2   [...]
ATTRIBUTES
PROPERTIES
PLACEMENT /some_path/some_name.pla
TEMPLATE /some_path/some_name.tpl
TIME 1979-01-00@12:00:00
GRID
-2 -2 -2 -2
1 -1 1 -1
-1 -1 -1
-2 -2 -2 -2
1 -1 1 -1
-1 -1 -1

[...]

0 -1
-2 -2 -2
0 0 -1
-1 -1
-2 -2 -2
0 0 -1
-1 -1
-2 -2
0 -1
-1
END
```

Appendix B

CellRouter Command Line Interface

In this appendix we will explain what the interface of CellRouter is. CellRouter admits the following command line arguments. All grid files follow the .grd structure exposed in appendix A.

Input
: Path of the input grid file.

Output
: Path of the output grid file.

Result
: Path of the file where execution data such as partial times is stored.

Rules
: Path of the file where the design rules are stored.

Rules set
: Name of the rules set that will be used, located into the file mentioned above.

Halo
: As explained before, given a subnet, all variables not included in a certain routing region defined by the subnet elements get a direct value of false. The halo metric, which is a positive integer, allows to expand such region. Sometimes, when the halo is too little, no solution is found because some subnet becomes unroutable. However, when the halo is big, the problem might become intractable.

Escapes

When no valid routing is found, if the escapes argument is given, the router will allow for some pins to be connected externally. The argument is the number of pins which are allowed to be left unconnected; it should be minimum.

Rounds

Number of rip-up and re-routing iterations the optimization heuristic will make. More rounds usually means a better result at the expense of more computation time.

Packs

Number of signals that the optimization phase will rip-up and reroute at once.