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

Facultat d'Informàtica de Barcelona

May 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:

Abstract

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 so that more complex cells become tractable. We evaluate different strategies and try to find the ones with the best solution for a given cell and design rules, either on terms of quality or computational time.

Contents

Chapter 1

Background

1.1 VLSI and EDA

The complexity of Integrated Circuits (ICs) has been growing year after year even since they were first introduced. According to Moore's Law [1], the number of transistors on chips 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 process of creating ICs by combining thousands 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 been increasing dramatically 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 some sort of 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. Normally, Register-Transfer Level (RTL) design abstraction is used to describe a circuit. It models synchronous digital circuits and focuses on the flow of digital signals between hardware registers. Hardware Description Languages

(HDLs) such as Verilog and VHDL are can be used to create such high-level representation of circuits. EDA tools allow then for an automated synthesis process to take place, which allows to design and implement technology-dependent circuits from a higher level of abstraction. This automated synthesis goes through a lot of steps to transform a RTL design into a geometrical layout, creating a physical design for the given RTL specification. Of course, given such and input multiple valid physical designs can be considered valid, so there is room for a lot of decisions and optimizations to be done in order to generate a good physical design of a circuit.

Given a 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 and we can consider that the output of one step is the input for the next one.

Logic Synthesis

Given an circuit abstraction, such as a RTL circuit, logic synthesis returns an implementation of the circuit in terms of logic gates. The RTL design is translated to 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 and other methods. After this 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.

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 placed in the chip. It allows for later hierarchical approaches and enables global wiring as a preparation for detailed routing, which will take place in later steps of the design. Trees and slicing structures, as well as dynamic programming for floorplanning optimization[5], are widely used in this area.

Partitioning

Given the netlist of the functions we want to implement it can easily be very large. Partitioning is the process of dividing the chip into smaller blocks so that later partitioning 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, [6]), 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. Many more partitioning approaches can be found in [7].

Placement

This steps consists in assigning cells to positions in the chip according to some cost functions while preserving legality (for example, with no overlapping). The input 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 [8], or quadratic-placement algorithms (such as Hall placement [9], the first analytical placer).
- Iterative algorithms, such that intermediate placements are modified in order to improve some cost function. This would include analytical methods such as force-directed placement.
- Nondeterministic approaches, including metaheuristics such as simulated annealing and genetic algorithms.

All this methods can be combined to obtain a more accurate result. Additionally other methods can be considered, such as 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 such as [10]. More information about placement algorithms can be found in [11].

Routing

The routing process determines the precise paths for nets on the chip layout respecting a set of design rules ensuring that the chip can be

correctly manufactured. It requires a physical placement of the layout, the netslists and the design rules required by the manufacturing process. The main aim then is to complete all required connections on the layout, although other objectives such as reducing total wirelenght of 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 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 circuits designs and help during the whole process of the design of a chip. All of these steps have an important algorithmic load and many 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. During the last years many algorithmic techniques have been explored to address the complex problem of determining how the pins 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. They have greater flexibility and can handle variable widths and spacing; however, grid-based routing is usually much efficient and easier for implementation given it's lower complexity when 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 it would allow the placement of wires with any direction if the model was the opposite. 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. Consider the grid on figure 1.1. We want to connect the node marked with an S and the one marked with a T. The grayed zones represent obstacles where the wire cannot be placed.

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 node to S. Later, 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

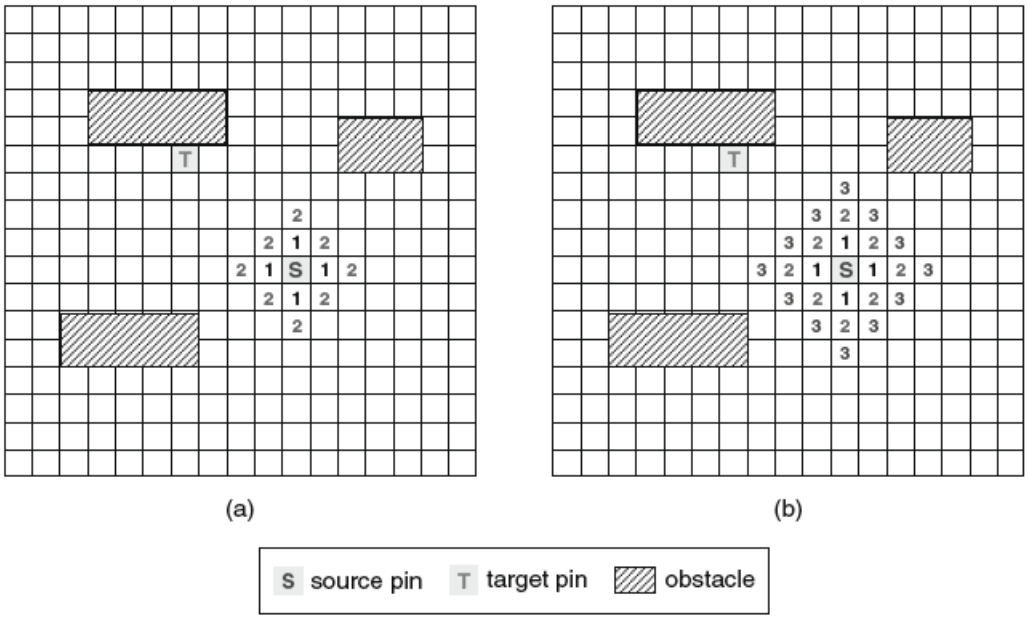

Figure 1.1: Maze Routing: Wave Propagation

1.1 with the waves corresponding to 2 and 3. Once the node T is reached, as seen in figure 1.2(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 1.2(b). Often, the preferred path is the one with the least number of detours. Notice that this algorithm guarantees to find a path between two points if such a path exists and this path is the shortest one.

This algorithm was proposed by Lee in [12] 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 DFS in combination with the BFS, starting point selection or double fan-out.

After this algorithm, many other graph-based search algorithm appeared. The most significant ones probably are Line-search routing [13] and algorithms based in the well known A*-search proposed in [14], which are widely used in modern routers.

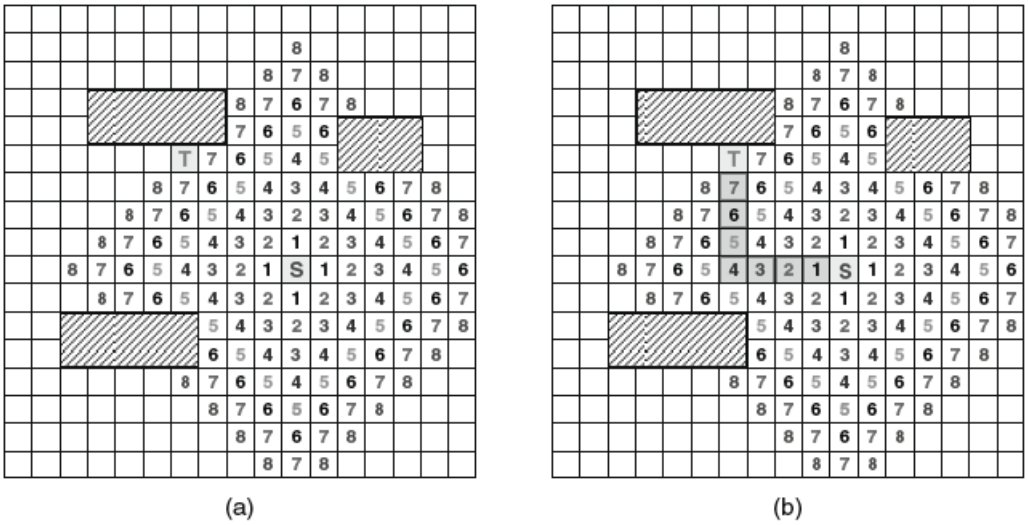

Figure 1.2: Maze Routing: Retracing

1.2.3 Global Routing

As exposed before, global routing can be considered the most coarse grain type of routing. It consists of 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 strategy.

Sequential global routing

In this schema we would select a specific net order and the 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 reroute* heuristic is used to refine the solution. It basically consists of 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 doesn't 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 doesn't 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

Given the output of the global routing stage, they are 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 would be 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 in 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. Given that the routing decisions made at any of the intermediate routing levels might be suboptimal, hybrid approaches using both methods have also been explored. However, given that routing decisions at a given level are irreversible, the quality of the solution is limited. This is a problem we will also face when later in the 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 generates a set of candidate routing trees using algorithms that generate

multiple rectilinear Steiner minimum 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 more fine granularity, at the expense of more computational complexity. However, manufacturing constraints are more easily included in this scheme. Usually, SAT-based formulations are solved 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 approach is the one proposed in [15], a segment-based approach inspired by the satisfiability formulation presented in [16]. However, such approach only managed to route small gates given the high computational complexity of the resulting formulation, as will be shown chapter 2.

As we have seen, many kind of routing strategies and approaches exist. Of course 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 [17] 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 the new requirements, specially when manufacturability considerations are day after day becoming more determining during the physical design process as will be shown in the next section.

The pictures for this section where taken from [18]. For more information on routing or VLSI algorithms in general take a look at that book or at [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. One such a methodology would be full-custom design. It 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 is very labor-intensive to implement. Full-custom design would typically be used in a situation where there are area limitations or special application needs. Some examples would be the design of cells within a standard cell library, memory cells or datapaths for high performance designs. However, aside from full-custom design, other design methodologies exist. We will now focus on the one where this project is involved.

1.3.1 Standard Cell Design

In the standard cell methodology, low-level VLSI layouts are encapsulated in abstract logic representations (such as a NAND gate). This way, logical level becomes independent of physical level design. Using such a 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 an AND or an 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 one shown in figure 1.3. They are normally optimized full-custom layouts so that area and delay are minimized.

Each one of this 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.

Schematic View
: Description of the transistors, their connections to each other and the terminals to the external environment. Multiple possible netlist for a given logical view exist.

Layout View
: Physical representation of the cell. The most important from the manufacturing point of view, as it is the closest to the actual manufacturable design. Again, many possible layouts exist for a given schematic description of a cell.

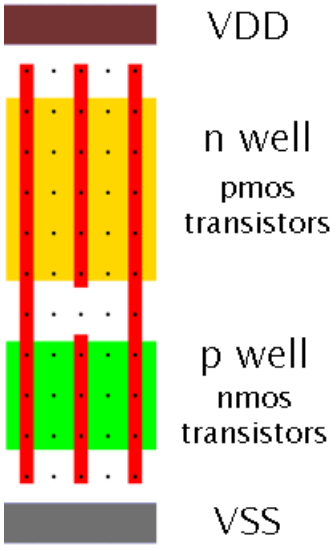

Figure 1.3: Possible structure of a standard cell

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

Figure 1.4: AND gate, logical view

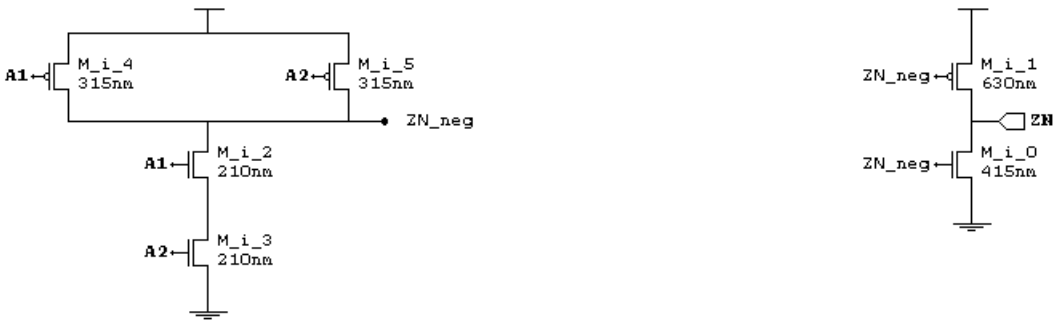

Figure 1.5: AND gate, possible schematic view

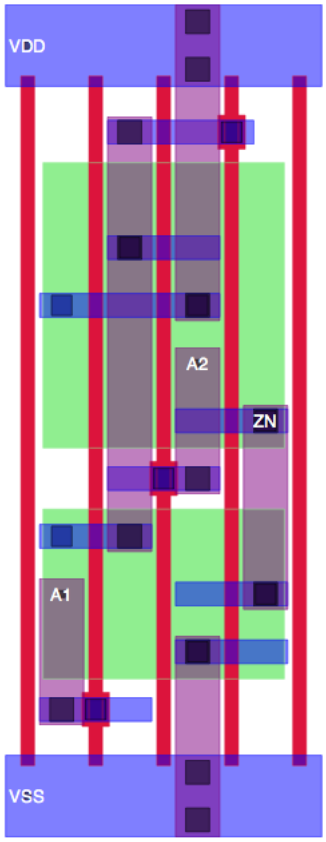

Figure 1.6: AND gate, possible layout for schematic in figure 1.5

1.3.2 Manufacturability-Aware Design

As we have seen, standard cell design provides help by encapsulating primitives so 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 decrease, the manufacturing process has been becoming increasingly complex.

One of the crucial processes involved in chip manufacturing is photolithography. During this process, chemicals sensitive to ultraviolet light are used, which can either harden of soften. Such products are applied onto the surface of the die and are exposed to ultraviolet light trough a mask with a desired pattern. After this step, the softened parts are removed and another mask with the next pattern will be used. Layers grow one above the other until all masks have been applied. Of course, the patterns on the masks are the final

product that the physical design has produced for a given circuit abstraction.

Of course, 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 the light wavelength [20]. Such approach is limited however 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.

To counter all this increasing complexity, litho-friendly layout techniques must be considered. This 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 synthetisation of such manufacturability - aware standard cells. Today, it is a process where complex design rules are imposed. Given so, regularity has been introduces as a means to make design automation tractable. Such regularity can be seen for example in figure 1.3, 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, where the grid unit is a fraction of the wavelength of the light which will later be used during the fabrication process. As it will be later explained, 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 will be a physical design adapted to the desired technology constraints.

1.4 Boolean Satisfiability Problem

As explained before, EDA tools rely on algorithmics to tackle a variety of computational problems which arise during IC design. One of the algorithmi-

cal problems that has attracted many 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 vocabulary 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

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

Which 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 be 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's 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's 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? Of 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's 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 compared to the original one. Solving the SAT problem for a formula in Disjunctive Normal Form (DNF), 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, but no transformation such that an arbitrary formula is transformed into a DNF and avoids an exponential growth has been found yet.

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 HORN-SAT. 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 planification, 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 codifying our problem in 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.

Let's see 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. For every node in the clique, it 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. For every node in the clique, it 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. For every node in the graph, it can't 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.

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

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

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

Now, given an instance of k-CLIQUE, we would encode it in CNF form. We would use it as an input to SAT-solver and the examine it's output. If it returned "unsat", it would mean that no assignation of values to the variables renders the formula true, thus implying that no clique of k nodes exists. However, if it returned a model for the formula, observing the i 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 codify in a boolean formula. This approach is used by CellRouter. To solve the problem, we must essentially describe it in terms of a CNF and use a SAT-solver to obtain a solution. Of course, 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 problem supposedly difficult to solve. If this approach is getting more attention is 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 in the DPLL algorithm, a systematic backtracking looking for a satisfiable assignment of the variables. However, a lot of additions and optimizations have been added, such as conflict analysis, clause learning, backjumping, random restarts and heuristics. These methods have been proven empirically to be essential to solve large instances of SAT. For some more information on the use of SAT in EDA take a look at [25].

Chapter 2

CellRouter

As explained on the abstract, the aim of the 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 be later extensively used will be 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 [15, 16]. 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 parametrized with the resources available

at any technology generation.

- Parametrizable attributes should be allowed for every wire segment.
- Should allow the router to select the best terminal locations.
- Should be independent from the set of design rules.
- In the case of unroutable cells, externally connected pints 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 contributions of the CellRouter tool are as follows.

- An encoding scheme for SAT-based formulas that makes large cells tractable.
- Windowing heuristics that allow to efficiently route large cells.
- A formalism to specify gridded design rules and multiple-patterning constraints.
- A strategy to allow externally-connected pins.
- Heuristics for quality improvement (wirelength and recommended design rules).

Graphs are used to represent a 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).

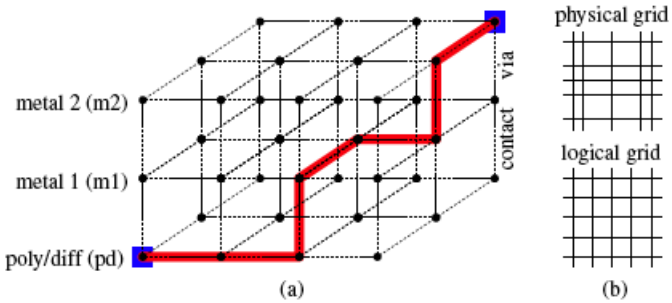

Figure 2.1: (a) Grid model for routing. (b) Physical and logical grid

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.

Figure 2.2 represents an instance of the 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 picture we can see the VDD (red) and VSS (ground) lines crossing the second layer. Will the router be able to find a valid routing for this grid?

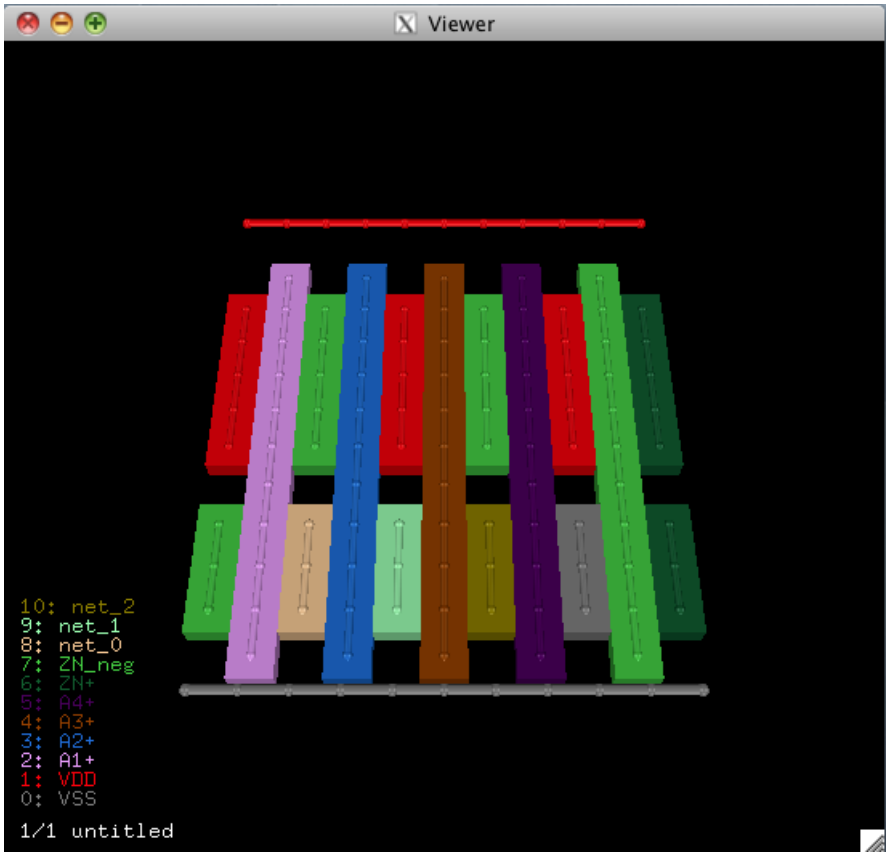

Figure 2.2: Routing grid problem instance

2.3 SAT to Solve the Routing Problem

Once we have some vocabulary 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, in this case as follows.

- $\rho(e)$: A variable that represents when 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 sub-nets 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 constraints for the grid given its routability nature. For example, we must impose that each edge is assigned to at most one net and two adjacent wires are assigned to the same net. Additionally, windowing is considered. That is because, empirically, it is 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. Even if this might imply that no solution is found even if one exists, it transforms some problems from intractable to tractable.

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 Metal 1 can only be laid out in the horizontal direction or that no adjacent vias can be connected to different nets. Extra clauses modeling wire attributes are included among this constrains.

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 a 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.

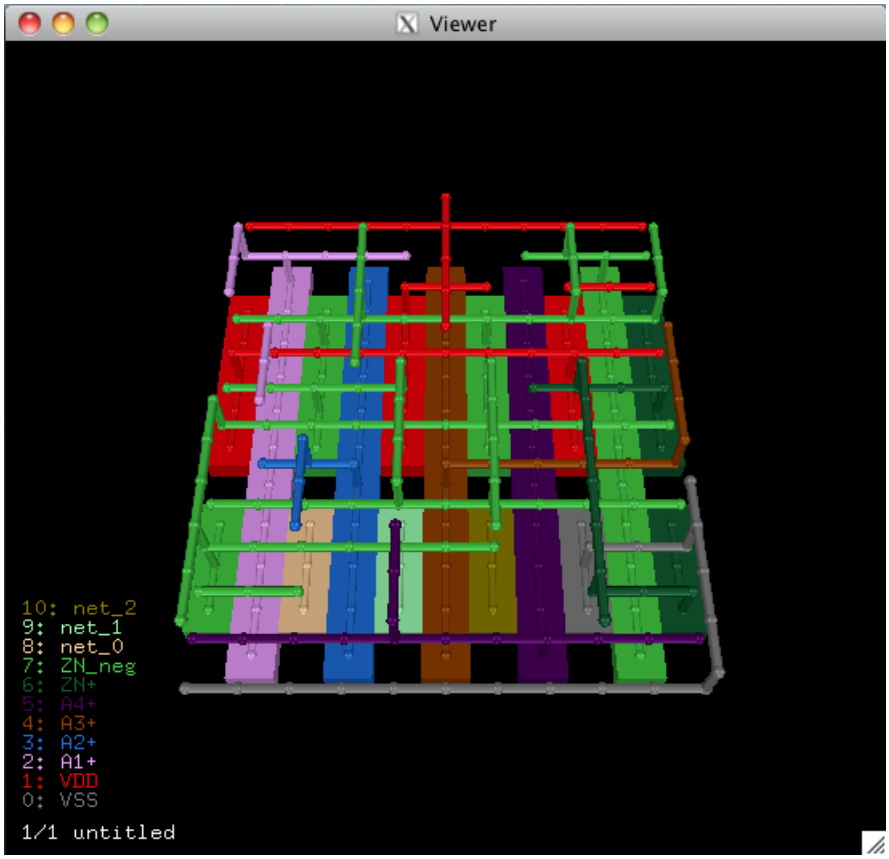

Figure 2.3: First obtained solution for 2.3

As stated before, among all valid solutions, some have better quality than others. For example, cells with smaller wirelength are preferred. We can at first glance notice that probably the solution proposed by the SAT solver in 2.3 has many elements that can be gotten rid of. CellRouter proposes a heuristic method to make this problem tractable using 0-1 linear programming. 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 on the basic solution obtained using the SAT solver until no significant improvement is observed, which in practice takes two rounds of re-routing 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.

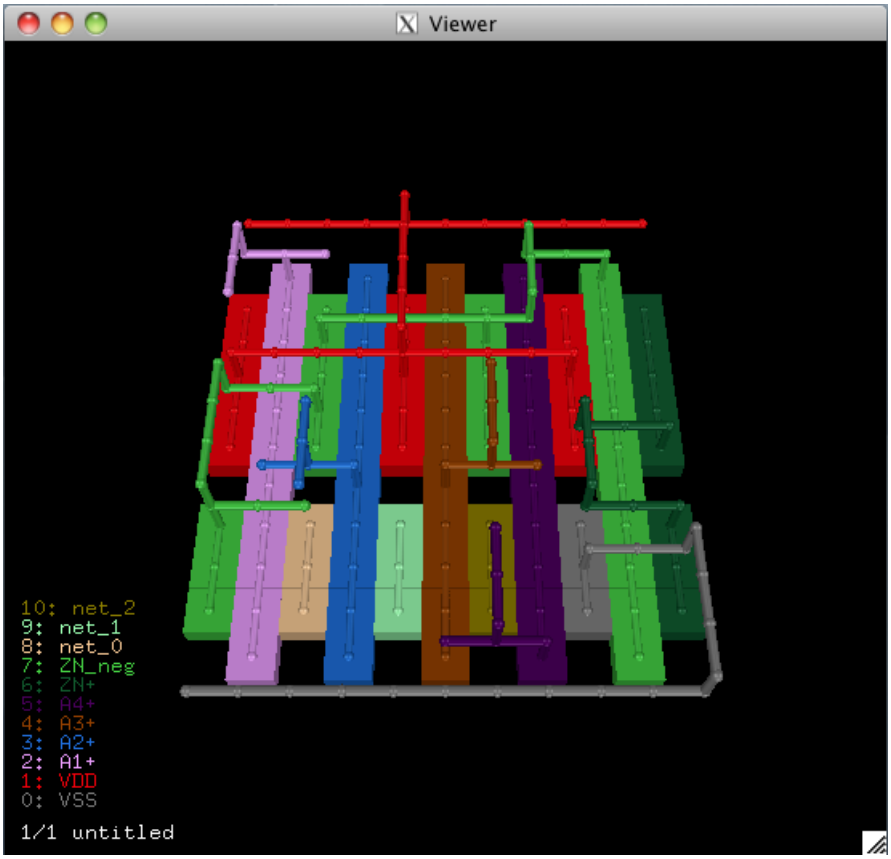

Figure 2.4: Optimized solution for 2.2

For more information on the grid data structure and what the command line interface of CellRouter is please 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 DRC 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 figure 2.1, CellRouters' *sparse* encoding outperforms the *dense* encoding that was used

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

		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
OAI221_X1	33	1626	15.4	1944	98.1	2679	40 hours

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, which is a great result.

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 to 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.

Chapter 3

Development

Consideracions i estudi inicial. Dificultats de l'espai d'exploracio. Flow de les eines. (Gridder-Router-Viewer, on intervenir...)

3.1 First iteration

C++ Cell Divider. Trying to tie C++ and Python with SWIG.

3.2 Second iteration

Eleccio de python i ipython. Python Data Structures. N parts divider, side signals divider, n-m histogram divider.

3.3 Third Iteration

Rutant de n-m deixant signals per banda. Rutant de n-m acostant signals de banda.

3.4 Last Iteration

Concatenaci de celles. Refinament de tot plegat. Metaalgorisme recorrent la cella.

Chapter 4

Results

Results.

4.1 Experiments

Uns quants experiments finals.

4.1.1 Synthesis of a full library

Synthesis of a full library.

4.1.2 Synthesis of concatenated cells

Synthesis of concatenated cells.

4.1.3 More experiments

Whatever.

Chapter 5

Conclusions

Que s'ha aconseguit? Que no? Dificultats previstes, no previstes?

5.1 Personal conclusions

Quin aprenentatge n'he extret?

5.2 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-algorithm. Experiments.	120 h.
June 2013	Experiments. Project Report.	100 h.
		600 h.

Table 5.1: Time study

Considering a salary of about 25 €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	1000€
Ipython, texmaker...	Free
Total tool cost	**2000€**

Table 5.2: Tool cost study

When considering all costs together:

	Cost
Engineering costs	15000€
Tool costs	2000€
Total project cost	**17000€**

Table 5.3: Total costs

5.3 Future work

Now, here comes some future work.

5.3.1 Multiple Rows

5.3.2 Many More Things

Bibliography

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

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

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

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

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

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

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

[10] 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.

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

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

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

[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

CNF Conjunctive Normal Form. 18–21, 25

DAG Directed Acyclic Graph. 5

DNF Disjunctive Normal Form. 19

EDA Electronic Design Automation. 4, 5, 7, 16, 19

FPGA Field Programmable Gate Array. 6

HDL Hardware Description Language. 4

IC Integrated Circuit. 4, 7, 15, 16

ILP Integer Linear Programming. 27

RTL Register-Transfer Level. 4, 5

SAT Boolean Satisfiability Problem. 16–19

VLSI Very Large System Integration. 4, 13

Appendix A

Grid Format

In this annex the format of the grid files will be described. They are kept in a plain text file with the *.grd* extension. 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 Implementation

CellRouter has been implemented using C++. One of the first decisions that was made during the project was to leave CellRouter as it was and focus on interacting with it through the command line instead of modifying the router itself. 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 to in 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 computationally hard.

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 rerouting the 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.