NTNU

Norwegian University of
Science and Technology

Towards Predictable Placement of Standard Cells for Regularly Structured Designs

Auritro Paldas

Embedded Computing Systems
Submission date: June 2018
Supervisor: Snorre Aunet, IES
Co-supervisor: Ronan Barzic, Nordic Semiconductor

Norwegian University of Science and Technology
Department of Electronic Systems

Master Assignment

Candidate name:
Auritro Paldas

Assignment title:
Towards Predictable Placement of Standard Cells for Regularly Structured Designs

Assignment text:
A lot of components in modern digital designs have very regular structures. Some examples are Programmable Ring Oscillators, Time to Digital Converters and CPU register files. The proper functioning of these components heavily depend on the way they are implemented in the design with respect to the placement of standard cells. This is due to the fact that many of these components are delay sensitive and the placement of cells in the layout affects the delay. Standard place and route tools, however, do not always ensure that the placement of standard cells is regularised, which can lead to sub-optimal results from these designs. The work on this thesis is aimed towards ensuring a regular placement of standard cells for such components, by developing a framework in a high-level language, from which the placement information needed by the place and route tools can be obtained. This information, when used by the tool, should result in a more predictable placement of standard cells, and should thus result in more optimal behaviour of such components.

Co-supervisor:
Ronan Barzic, Nordic Semiconductor ASA

Supervisor:
Snorre Aunet, Professor, Department of Electronic Systems, NTNU

Abstract

Electronic Design Automation (EDA) tools have revolutionised the way digital integrated circuits are being designed. Modern EDA tools are capable of implementing circuits with billions of transistors. An important step in the design flow of digital ICs is the placement of standard cells, where millions of cells are automatically placed on the layout by place and route tools, while adhering to timing, power and congestion constraints. However, a completely automatic flow can have certain drawbacks, especially when it comes to the placing of certain regularly structured designs. These designs are extremely sensitive to delays, and hence, call for a systematic and predictable way of being placed in the layout.

This work creates a software framework which enables a designer to effortlessly describe digital circuits and its placement information in a high level language. The program generates a Verilog netlist, a relative placement script and image files corresponding to the design described by the designer. It supports hierarchical designs. It also allows the designer to select the desired library data to be used to implement the netlist. The netlist and the placement script can be read in directly by a place and route tool to create a design exactly as specified by the designer. This can improve the performance of such regularly structured designs. The program was run on three test designs, and all the outputs were generated as expected. Comparisons of the layouts were made in Synopsys® IC Compiler™ with and without using the generated placement data. Structured designs, as expected, were obtained, when the generated placement script was used.

Preface

This work was done as a part of my master's thesis at the Norwegian University of Science and Technology, Trondheim, from January to June of 2018. This was the second and final year of my Erasmus Mundus Master's programme in Embedded Computing Systems. This work was done in collaboration, and at the premises of Nordic Semiconductor ASA, Trondheim.

Auritro Paldas

June, 2018

Trondheim

Acknowledgements

Snorre Aunet, my thesis advisor.

Ronan Barzic, my supervisor at Nordic Semiconductor.

My colleagues at Nordic Semiconductor : Knut Austbø, Jan Egil Øye, Johnny Pihl, Torstein Nesje, Sondre Nesset

Wolfgang Kunz, EMECS Coordinator

NTNU Trondheim and TU Kaiserslautern

Nordic Semiconductor ASA

The European Commission

My friends at NTNU: Aleksandar, Alex, Andrea, Aysel, Bruno, Camillo, Deepak, Diego, Dmitry, Domitilla, Emil, Ezio, Frida, Guilherme, Jasmina, Lorenzo, Luca, Ludovico, Matteo, Mattia, Maurice, Pauline, Rikke, Sebastian, Sergey, Stephan, Vera

My friends at TU Kaiserslautern: Abror, Amina, Anes, Anna, Asmaa, Bruno, Canhui, Carlos, Chinmaya, Claudia, Felipe, Fidel, Filip, Galina, Goce, Hallie, Hlib, Hugo, Ivica, John, Kavya, Maria, Mayara, Munther, Nejdi, Olga, Pranav, Qingying, Rashed, Rodrigo, Salah, Sebastian, Selma, Sonja, Sophie, Srishti, Vishnu, Zhani

My family

My parents.

Contents

List of Figures

Listings

Chapter 1

Introduction

1.1 Motivation

The complexity of digital designs have perpetually been on the rise ever since the invention of the transistor. Designs have grown from a few hundred to tens of thousands to more than a billion transistors in chips of today [7]. This has made the custom design of circuits in the form of schematics impossible. Furthermore, the time to market requirements of modern designs have called for a more systematic and automated way to design digital circuits [3]. This has led to the birth and the tremendous growth of the Electronic Design Automation (EDA) industry over the past four decades.

This growth has been a boon to electronic design engineers. EDA has influenced all aspects of the design flow, and has automated various tedious tasks. But the biggest impact of EDA is seen in the realm of physical design, which involves the steps leading to the conversion of the structural description of a circuit to a physical description, ready to be sent to a foundry for manufacture. This involves a multitude of steps, one of which is the placement of standard cells. It is a very complex task, with billions of standard cells needed to be placed on the available die area, while trying to meet the timing requirements, and ensuring that the routing of the cells can go through smoothly. A place and route tool is utilised in this step, which, with the right constraints, is able to perform this complex task with little human intervention.

However, for certain special cases, the results obtained by the placement tool might not be satisfactory. One such case is in the placement of certain regularly structured designs, for example, programmable ring oscillators and time to digital converters. The performance of such designs is heavily influenced by the delay between their logic cells, and the delay is

influenced by their placement. This is especially true in lower technology nodes as the delay there is strongly dependent on the parasitics, which in turn depends on the spacing between the cells. A place and route tool typically tries to optimise the global parameters, without any consideration towards the structure of individual logic blocks. This results in a sub-optimal performance of such regularly structured designs. Special directives are thus needed to be given to the tool to ensure that the desired placement of such special designs is achieved.

1.2 Objectives

The objective of this thesis is to develop a software framework which would enable a digital designer to, with relative ease, ensure a predictable placement of the cells in his/her design. This involves the ability to easily describe the design in a high level language, along with defining the relative placement information of its cells. The program should then be able to generate a Verilog netlist of the defined design, along with a script describing the relative placement of the cells, which can be read in by a place and route tool. The framework should support both leaf-level and hierarchical designs. Further, an image of how the placed cells would look like is to be obtained. This would help the designer in quickly evaluating the placement of cells in his/her design. Finally, it is also desired to have the ability to easily switch between different library data before generating the Verilog netlist.

1.3 Structure of the Thesis

This thesis is structured in the following way.

Chapter 2 discusses about the theory needed to understand the work done in this thesis. At first, it looks into the design flow of digital circuits. Then, it discusses about standard cell placement in a little more detail, and looks into a few placement algorithms.

Chapter 3 talks about the need for predictable placement. It looks into a few regularly structured designs and talks about the drawbacks of completely automated placement. Finally, it describes the relative placement methodology followed in Synopsys® IC Compiler™.

Chapter 4 describes in some detail the software framework that was developed as a part of this thesis.

Chapter 5 presents the results obtained by running the program on three sample designs. The generated netlists, placement scripts and images are presented. Layout data from IC Compiler™ is also shown in comparison with data obtained without using the placement

scripts.

Chapter 6 summarises and concludes the work done in this thesis. It also gives a few suggestions for possible future work that could be done related to this thesis.

Chapter 2

Background

In this chapter, we are going to look into the theory that is required to understand the work done in this thesis. We would start off by describing the Application Specific Integrated Circuit (ASIC) flow, and the role of standard cell placement in it. We would then discuss placement in more detail. Next, we will look into the basics of static timing analysis and discuss why it is relevant to the placement of standard cells. Then, we will look at the placement algorithms used by commercial placement tools. Finally, we will look at a few regularly structured designs and discuss why it is important for them to have a predictable placement.

2.1 The ASIC Flow

Complex digital circuits of today are typically designed by the semi-custom design flow. In this strategy, blocks that have strict area and timing requirements are custom designed, while the rest of the design is subjected to a standard sequence of steps, commonly referred to as the ASIC flow. In standard cell based designs, elementary logic functions are custom designed and are called "standard cells". These standard cells are then used to construct the whole design, and are free to be placed anywhere in the design area. An overview of the steps of the ASIC flow can be seen in Figure 2.1. We will now briefly discuss the various steps of the flow [5] [3] [6] [12].

Specification and Partitioning

The first step in the design process is to capture the specifications of the design. These include metrics such as the functionality, algorithms, clock frequencies, voltages, power requirements, communication protocols, type of packaging, etc. Often, especially in the case

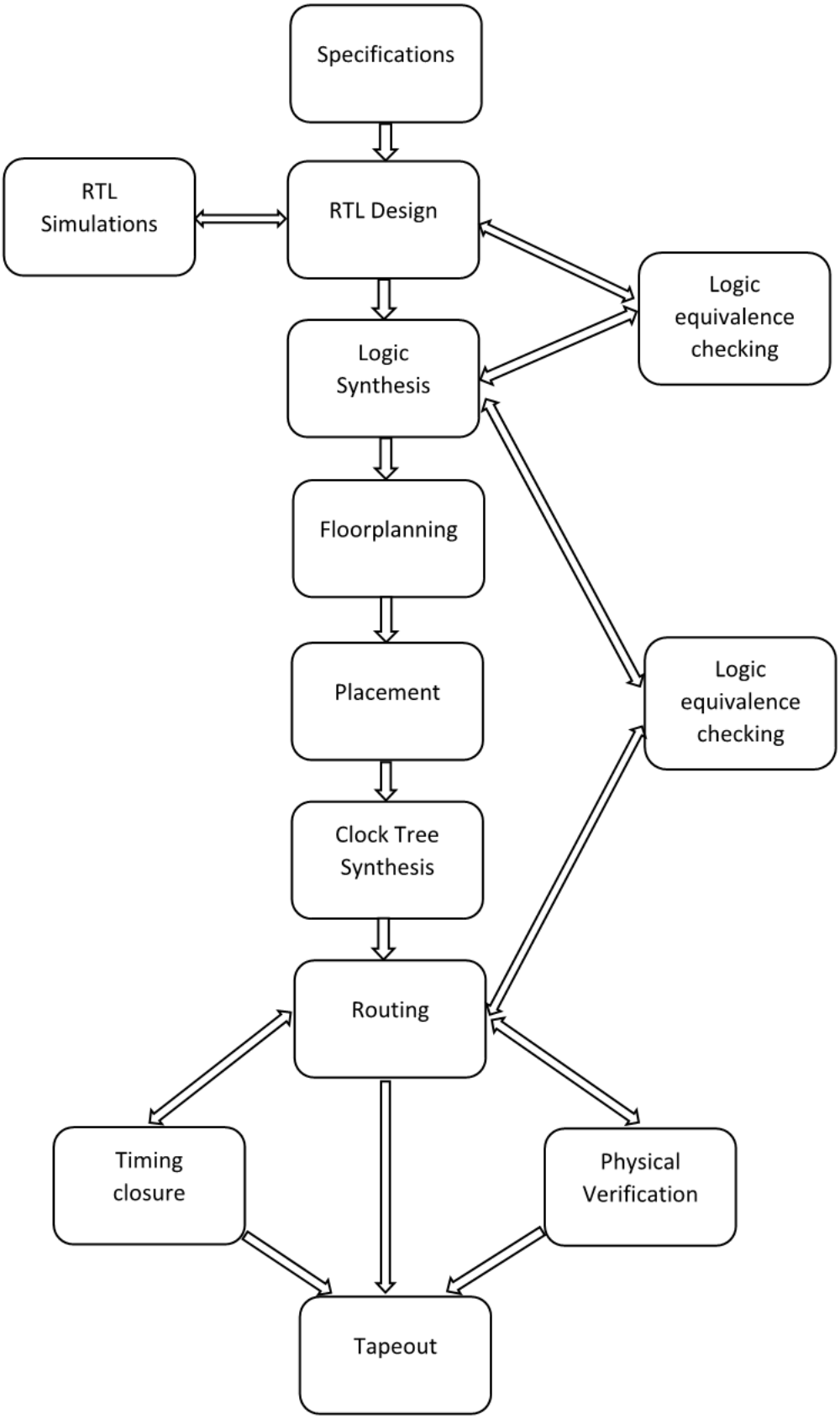

Figure 2.1: The ASIC Flow

of embedded system design, a partitioning is done on the algorithm to divide its implementation into hardware and software. The hardware part is taken up to be implemented as a dedicated ASIC, whereas the software part is then run on a microprocessor. Algorithms such as simulated annealing help us to formulate the partitioning such that metrics like area, power and performance are optimised [1] [4] [13].

RTL Design

After the specifications have been derived and the parts to be implemented in hardware have been identified, the next step is to implement these specifications. In earlier days, this was done by drawing the circuit schematics, but, as mentioned in the introduction to this chapter, this is not feasible anymore. Modern designs are entered by using Hardware Description Languages (HDLs). Commonly used HDLs are Verilog, SystemVerilog and VHDL. Using these languages, the design can be specified as a behavioural model, much like a high level language, such as C. This makes it easy to describe complex designs without worrying about the structural components of the circuit [3].

RTL Simulation

After the RTL design is complete, the next step is to simulate the design to verify its functional correctness. Simulations are typically done in an RTL design environment, where testbenches are written and the design is subjected to a variety of stimulus to check for correctness. Formal methods such as assertion based verification are used in modern designs to ensure a more exhaustive verification. Simulations performed at the RTL level are much faster than simulations done at the gate level, and hence, it is important to check for design correctness at this stage [17] [2].

Logic Synthesis

Once the functionality of the design has been satisfyingly verified, the next step is to synthesise the design. Synthesis is the process of converting the behavioural description of the RTL to a structural description. This is performed by the help of a synthesis tool. The tool takes the RTL description, the timing and power constraints, and the technology library as inputs. The technology library contains a description of standard cells that are available to be implemented. It performs logic minimisation and technology mapping to generate a structural description of the design, called the netlist, trying to meet the timing and the power require-

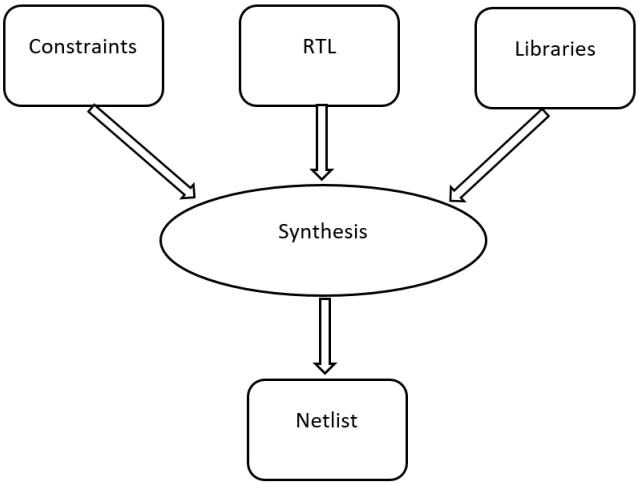

Figure 2.2: The Logic Synthesis Flow

ments. The synthesis flow is shown in Figure 2.2. At this stage, the design is typically also made testable by inserting design for test (DFT) logic [19].

Logic Equivalence Checking

On obtaining the synthesised netlist, we would like to ensure that its functionality is the same as that of the RTL. This is done by using a logic equivalence checker. Logic equivalence checking is a part of the formal verification methodology. As opposed to verification through simulation, formal verification does not use input vectors to verify the behaviour of a system, but, rather, mathematically proves or disproves properties of a design. It uses the concepts of satisfiability, binary decision diagrams, temporal logic, among others, to do so. It is advantageous over simulation as it exhaustively covers all possible cases to be looked for, and often takes much shorter times to complete than simulations. Logic equivalence checking uses formal methods to prove or disprove the equivalence of two designs. Thus, it is used in the ASIC design flow to help us in ensuring that the functionality of the design has not been affected by the various optimizations performed by the tools. In this case, it is used to check whether the RTL and the synthesised netlist are functionally equivalent or not. This results in a much faster verification than running simulations on the netlist again. Equivalence checking is used at various stages of the design flow to ensure design correctness, as shown in Figure 2.3 [20].

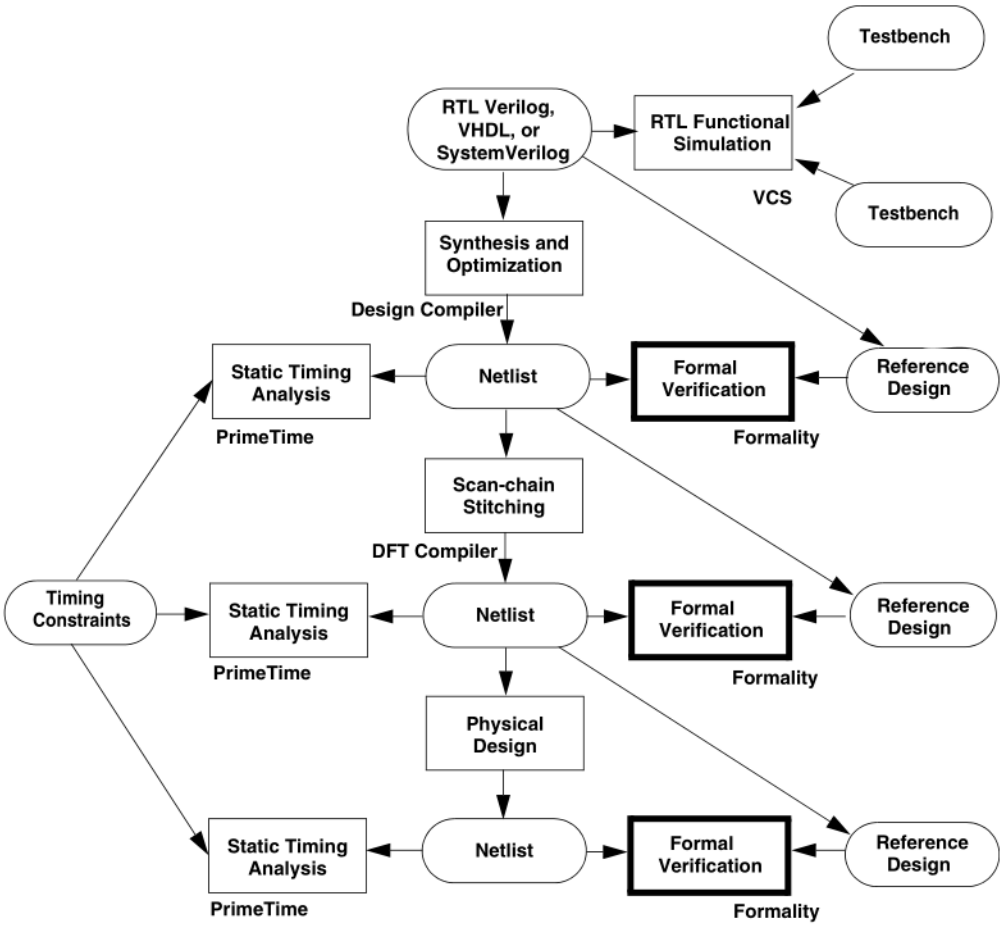

Figure 2.3: Logic equivalence checking in the Synopsys® flow. Courtesy [20]

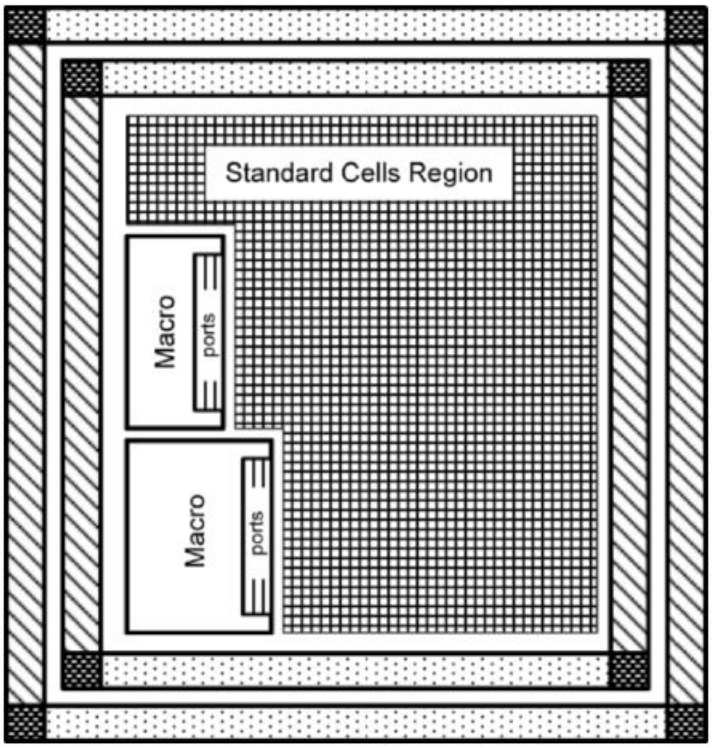

Figure 2.4: A Typical Floorplan. Courtesy [6]

Floorplanning

After having got the structural netlist of the design, we move on to the next sequence of steps, collectively called physical design. The first step of physical design is floorplanning. In this step, the dimensions of the chip are estimated and defined. The macro blocks are also custom placed in this step. The power architecture of the chip is also built here such that a stable voltage is available to all standard cells to be later placed in the layout. The I/O pads are also placed in this step. A good floorplan ensures that there are no routing congestions in the later steps of the flow, and that the design timing is easily met [23] [12]. A pictorial representation of a typical floorplan is shown in Figure 2.4.

Placement

Once the floorplan is completed, the next step is the placement of standard cells in the layout. Standard cells are the various logic gates available in the library. Most of them have the same height (or a multiple of that) which makes it easy for a placement tool to place them on the standard cell rows. The standard cell rows are alternate rails of power and ground which are defined in the layout during the floorplanning stage. The power and ground pins

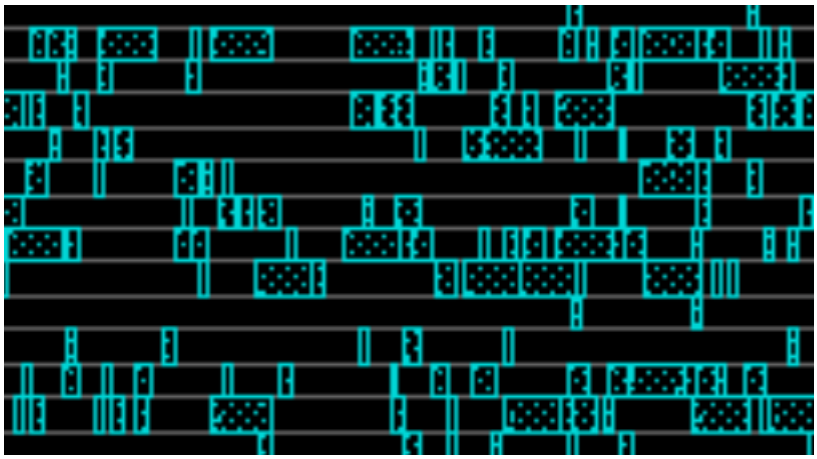

Figure 2.5: Placed Standard Cells

of the standard cells are positioned in such a manner that they get shorted to the power and ground rails when placed in the standard cell rows. This ensures that no further power routing is required for the standard cells. The placement is done in such a way that the routing congestion is minimised, and also that the timing constraints are met. We will discuss placement in a little detail in Section 2.2. An illustration of placed standard cells is shown in Figure 2.5.

Clock Tree Synthesis

After the standard cells have all been placed, the next step is the distribution of the clock signal to all the cells and macros. This is known as clock tree synthesis. One of the aims of this step is to distribute the clock in such a manner that the clock skew is minimised between end points of the leaf cells of the clock tree. This helps in meeting the timing requirements of the design. Various structures and algorithms have been proposed in literature to achieve this. The clock tree is built using special buffers and inverters called clock buffers/inverters which have equal rise and fall delays. This ensures that the clock signal is not distorted and that the pulse width of the clock is maintained. Clock gating cells are often added on the clock network to prevent the clock signal from reaching logic that is not in use. This helps conserve dynamic power of the design.

Routing

The final step of the physical design process is to route the signals to all standard cells and macros in the design. The routing step is broken down into two parts, global routing and detailed routing. During global routing, the design is broken down into fragments and the

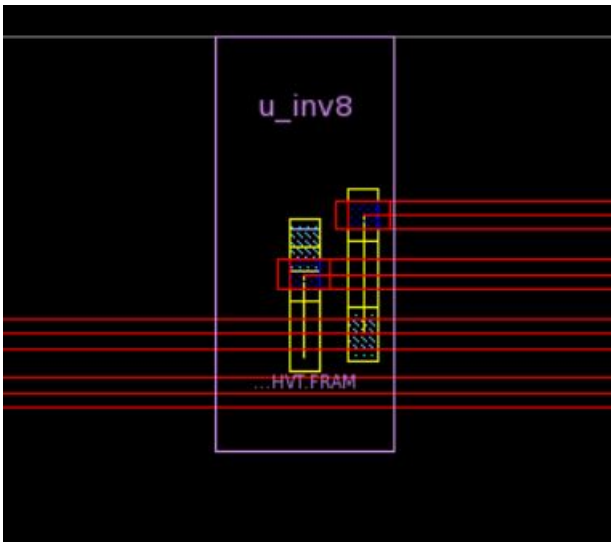

Figure 2.6: Routing in 2 layers, Yellow (vertical) and Red (horizontal)

feasibility of routing along the edges of the fragments is established. During detailed routing, the actual nets are laid down along routing tracks to connect to the pins of the various cells in the design. These steps are done while trying to minimise the route lengths and meet the timing constraints. Various routing algorithms are mentioned in literature. The routing is performed over multiple metal layers, which lie on top of one another. By convention, the routing is done alternating vertically and horizontally on adjacent layers. This makes it easier for the routing tool to route the design. An example is shown in Figure 2.6.

Timing Closure

Before taping out, a timing analysis is to be performed on the fully routed design. This is to ensure that all the timing constraints are indeed met. At first, a parasitic extraction is done on the design which gives us the parasitic data in a specific format. This data, along with the final netlist and the timing constraints is used by a timing analysis tool to exhaustively check for timing violations in the design. This is called static timing analysis. This is preferred over simulations as it is much faster and as it can check for all possible timing paths, not limited by input vectors. Timing violations like setup, hold, recovery, removal, etc are checked. If corrections are needed, incremental changes are made in the design and such parts are re-implemented. This is done in an iterative process until all the timing violations are fixed.

Physical Verification

Finally, before sending the design data to the foundry, certain physical checks need to be performed on the design. These checks correspond to certain design check rules provided by the foundry. Examples of such rules are minimum spacing between nets, minimum thickness of nets, minimum and maximum density of metal in a given area, etc. These checks ensure that the yield of the chips are high. Similar to timing closure, corrections are made in the design and such parts are re-implemented iteratively until all such checks are satisfied.

2.2 Placement and Placement Algorithms

Placement of standard cells is one of the most critical steps of the physical design process as it has a very high impact on design convergence [6] [12]. The objectives of the placement step is to successfully place all the standard cells in as little area as possible, while ensuring that the design remains routable and that the timing constraints can be met. Additional requirements like minimising the power dissipated and the cross-talk between nets is often applied [12]. The placement process is usually broken down into two stages, called global placement and detailed placement. In global placement, the interconnects between the various cells are minimised, whereas in detailed placement, the timing and the routability of the design is tried to be met [6]. We shall now look at these two steps.

2.2.1 Global Placement

The global placing process starts off by placing all the standard cells in the placable area of the die. Then partitions are made to divide the die into multiple segments. Then, the problem reduces to moving the cells from one segment to another such that a cost function is reduced. The cost function can be interconnect lengths, or ensuring homogeneous placement of cells (to ensure that routing along one direction is not unduly stressed.) A variety of min-cut algorithms are utilised to solve this problem. Algorithms such as quadrature, bisection and clustering are commonly used [15] [6]. Global placement defined the initial placement of the cells and this is further refined by the detailed placement. [24].

2.2.2 Detailed Placement

Detailed placement algorithms refine the global placement by looking into the timing, power and routability constraints. For ensuring routability, the algorithms try to keep the cells apart

from each other. For meeting timing constraints, the cells are moved such that the wire delays in a timing path are minimised. Cells are also swapped with ones having a higher driving strength if required. This improves the input transition time on the following cell, thereby reducing its delay. For reducing the power dissipated, the algorithms try to arrange the cells such that the capacitive loads seen at the cell outputs are minimised. Simulated annealing is a common algorithm used for detailed placement [6] [12].

It can also be noted that the global placement algorithms are constructive in nature, and the detailed placement algorithms are iterative in nature.

Genetic algorithms, which model the problem based on the principles of selection, crossover and mutation, are another common class of algorithms used to solve the placement problem [11] [8].

Chapter 3

The Need for Predictable Placement

3.1 Drawbacks of fully automated placement

While in most cases, EDA tools for placement are a boon to design engineers, they might produce sub-optimal results certain times. One such situation arises during the placement of certain regularly structured designs. These are designs where the performance of the circuit is heavily dependant on the physical structure of the circuit. Some common examples of regularly structured designs are programmable ring oscillators, time to digital converters, CPU register files, multipliers based on partial product addition, etc.

The reason why these designs need to be regularly structured is that they are extremely delay sensitive. A sub-optimal placement of these designs would thus result in poor behaviour. For example, in a programmable ring oscillator, if the delay between the inverters is not matched, the oscillation frequency will cease to be a linear function with respect to the number of stages selected.

The problem of dependence of delay on the placement is further exacerbated in lower technology nodes. As standard cell delays become smaller and smaller with reducing device dimensions, the parasitic delays start to become a significant factor in the total delay in the timing path. In earlier designs, the parasitic delay used to be an insignificant contributor to the total delay, and hence, the placement of the cells did not matter much in the delay calculations. But in nanometer designs of today, the interconnect between cells plays a crucial part in determining its performance. The dependence of delays on certain designs can be seen in [22], [14], [16], and [9].

The placement algorithms of EDA tools do not necessarily address this problem. We see in Section 2.2 that the global placement algorithm does indeed try to minimise the inter-

connect lengths. But this is done in a global sense and would not necessarily lead to the optimised design of a particular module. Moreover, the detailed placement algorithms try to optimise the congestion, timing and power, which might be in conflict to the desired regular structure of a module. In particular, the algorithms targeting timing closure would target the critical paths of the whole design, de-prioritising the non-critical or non-constrained paths. This might lead to an extremely staggered placement of the cells in the desired module. These considerations call for a need to be able to predictably place selected modules in the tool flow of place and route.

3.2 Relative Placement in Synopsys® IC Compiler™

To address the problems mentioned in the previous section, the place and route tool Synopsys® IC Compiler™ provides a way for the designer to describe the relative placement of desired modules. This can then be integrated in the regular place and route flow as shown in Figure 3.1.

This is done by providing a framework where the cells can be marked to a location in a placement matrix, with a certain row and column number. This is called a relative placement group. An example of such a group is shown in Figure 3.2.

A relative placement group can be created by the following command

```
create_rp_group rp1 -design designA -columns 6 -rows 6
```

Listing 3.1: Creating a relative placement group

This creates a 6x6 group called rp1 as shown in Figure 3.2.

Next, we need to add cells to the group. This can be done by using a command as shown below

```
add_to_rp_group design1::rp1 -leaf U23 -column 0 -num_columns 2 -row 0
    -num_rows 1
```

Listing 3.2: Adding cells to a relative placement group

This would add the leaf cell called U23 to the position in the group represented by column 0 and row 0. Similarly, other cells can be added to the group [18].

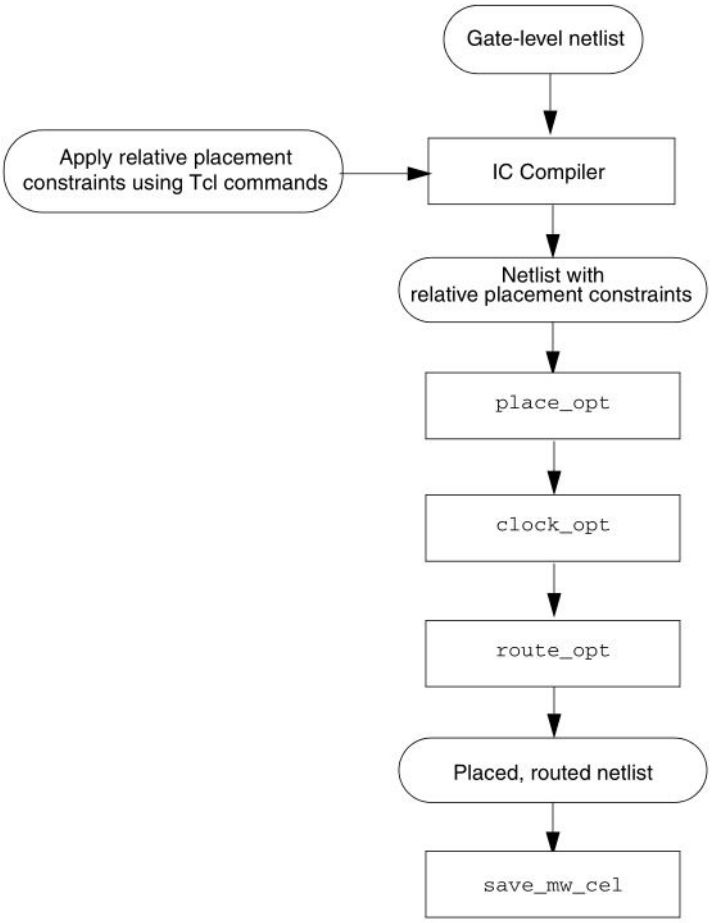

Figure 3.1: Relative placement flow. Courtesy [18]

.

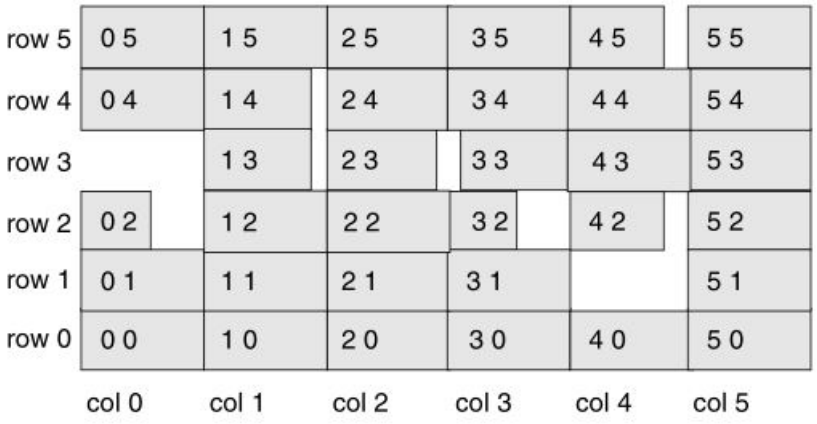

Figure 3.2: Relative placement group. Courtesy [18]

.

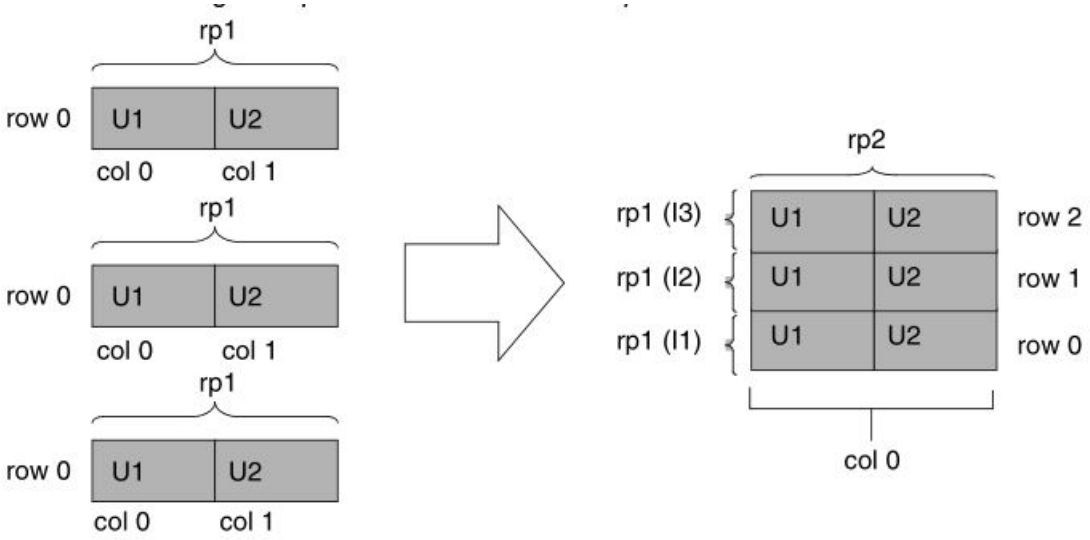

Figure 3.3: Hierarchical relative placement. Courtesy [18]

.

3.2.1 Hierarchical Relative Placement

It is possible to define relative placement for hierarchical designs. This is done by creating a relative placement group for the leaf level design, and instantiating it in the hierarchical design. This is shown in Figure 3.3.

This can be specified by the following code

```
create_rp_group rp1 -design pair_design -columns 2 -rows 1
add_to_rp_group pair_design::rp1 -leaf U1 -column 0 -row 0
add_to_rp_group pair_design::rp1 -leaf U2 -column 1 -row 0
create_rp_group rp2 -design mid_design -columns 1 -rows 3
add_to_rp_group mid_design::rp2 \
-hierarchy pair_design::rp1 -instance I1 -column 0 -row 0
add_to_rp_group mid_design::rp2 \
-hierarchy pair_design::rp1 -instance I2 -column 0 -row 1
add_to_rp_group mid_design::rp2 \
-hierarchy pair_design::rp1 -instance I3 -column 0 -row 2
```

Listing 3.3: Hierarchical relative placement

Here, we see that the leaf level group rp1 is first defined and then instantiated in the higher level group called mid_design [18].

Chapter 4

Methodology

Having looked at the concepts of placement of standard cells, and the importance of relative placement in the previous chapters, we shall, in this chapter, discuss the work done in this thesis.

4.1 Overview

Let us begin by reiterating the objectives of the work to be done in this thesis, as was mentioned in the introductory chapter. It is desired to develop a software based framework to describe logical circuits, and to derive design data from it. The description of the circuit has to be simple, like that in a high level programming language. The desired data include the Verilog netlist of the circuit, a script containing the relative placement data which can be used by place and route tools, and an image file which shows how the placement looks like. The framework should support hierarchical designs. Such a framework would help digital designers to easily describe digital circuits and how they intend the cells in it to be placed. They would then be immediately provided with a netlist of their desired circuit along with the placement script describing the relative placement. They would also have a visual representation of their design in the image file, and would thus know exactly how their design would look like in the layout.

In the following sections, we will go through the steps that were followed in the creation of the framework. Python is used as the programming language in this work.

4.2 Describing the Design

To be able to describe the complete design, the logical connections between the cells, and their relative placement need to be defined. But at first, we need to define what a cell is. For that, an abstract class called *Cell* is created. All modules, logic cells and standard cells are defined with respect to this. Similarly, a class called *Pin* is also defined. These can be seen in Appendix A.

An example of a definition of a standard cell using this class is shown below

```
class AND2(Cell):
    size_x = 1
    size_y = 1.4

    def __init__(self,instance):
        super(AND2, self).__init__(instance)
        self.A = Input("A")
        self.B = Input("B")
        self.Z = Output("Z")
```

Listing 4.1: Standard Cell Definition

The relative dimensions of the cell, taken from the library data, are entered in the variables *size_x* and *size_y*. The pins of the cell are then defined to be input or output.

4.2.1 Logic Definition of the Design

We next have to define the logic of the circuit. We start off in the same way as a standard cell, defining the size and the input-output pin definitions. A snippet of such a definition for an adder is shown below.

```
class Adder(Cell):
    size_x = 5
    size_y = 2
    def __init__(self,instance):
        super(Adder, self).__init__(instance)
        self.A  = Input("A")
        self.B  = Input("B")
```

```
self.CI = Input("CI")
self.S  = Output("S")
self.CO = Output("CO")
```

Listing 4.2: Adder: Size and I/O

It is to be noted that the size of such a module (Adder in the example above) is only needed to be defined if it is used as a leaf cell in a hierarchical design.

We next have to define the cells used in the design, and assign them to standard cells. This is done as shown below.

```
self.u_and0 = AND2("u_and0")
self.u_and1 = AND2("u_and1")
self.u_and2 = AND2("u_and2")
self.u_or0  = OR3("u_or0")
self.u_xor0  = XOR2("u_xor0")
self.u_xor1  = XOR2("u_xor1")
```

Listing 4.3: Adder: Cell instantiations

Here *AND2*, *OR3*, etc are standard cells already defined in the framework.

Finally, we need to make the logical connections between the various pins of the design. We use the *connect* method of class *Cell* as described in Appendix A. A snippet is given below.

```
self.connect(self.u_and0.Z, self.u_or0.A)
self.connect(self.u_and1.Z, self.u_or0.B)
self.connect(self.u_and2.Z, self.u_or0.C)
self.connect(self.u_or0.Z,  self.CO)
```

Listing 4.4: Adder: Cell connections

In the first line of the code above, the *Z* pin of *u_and0* is connected to the *A* pin of *u_or0*. A dictionary keeping track of connections is updated at each step. This is explained in Section 4.3.

4.2.2 Relative Placement Definition

After having defined the design from a logic standpoint, we next need to extend the definition to include the relative placement information. We need to create a coordinate system to

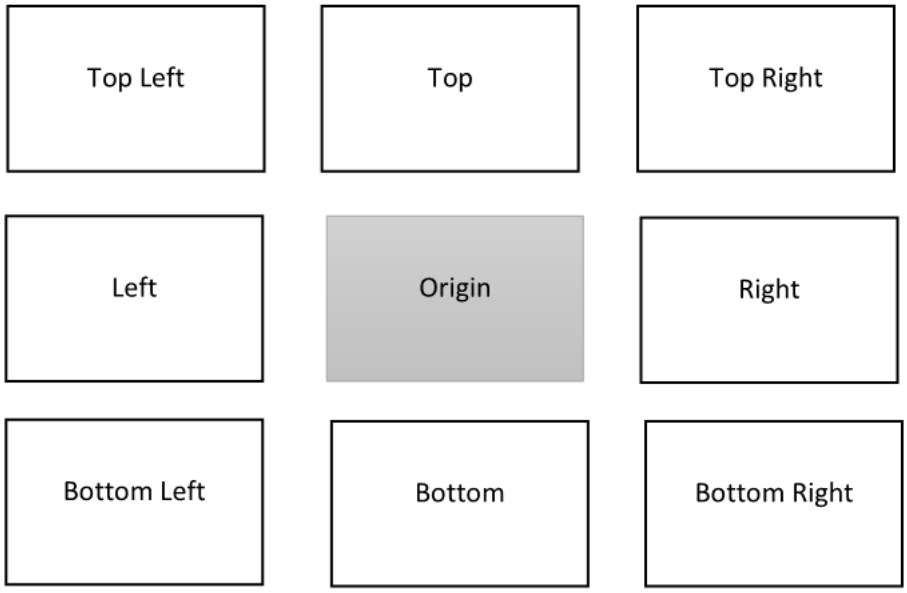

Figure 4.1: Origin and Relative Placement

describe this as is mentioned in Section 3.2. We first define an origin and assign it to a cell, as seen below:

```
self.rp_origin(self.u_and0)
```

Listing 4.5: Relative Placement: Origin

This assigns to the cell *u_and0* the coordinate of (0,0). It is, however, to be noted that this is not necessarily the origin of the coordinate system, and merely an anchor point such that other cells can be placed relative to it. The framework allows for the other cells to be placed in eight positions, as shown in Figure 4.1.

The code below is an example showing how this is achieved.

```
self.rp_rightof(self.u_and0, self.u_or0)
self.rp_rightof(self.u_or0, self.u_xor0)
self.rp_topof(self.u_and0, self.u_and1)
self.rp_topof(self.u_or0, self.u_xor1)
self.rp_topof(self.u_xor0, self.u_and2)
```

Listing 4.6: Relative Placement: Top and Right

We have used two functions here, *rightof* and *topof*, from the available eight. The first line places the cell *u_or0* to the right of *u_and0*. The placement of the first cell must already be defined. This can either be the origin, or a cell placed relative to an already placed cell. If the placement of the first cell is not defined, the code will throw an exception and exit, as described in Section 4.4.

These considerations complete the part of the framework needed to describe the design. We now move on to describe how the various data are generated.

4.3 Generating the Netlist

(Note: The work in this section was performed by my mentor and is not my contribution. I was involved in making some syntactical corrections. The discussion is thus brief.)

Having described the design, the next step is to generate the Verilog netlist of the design. The methods *connect*, *update_connection* and *to_verilog*, as can be seen in Appendix A, are used in this context. The *connect* method creates a list of ordered pairs of every pin to pin connection in the design. The *update_connection* method takes this list and appropriately assigns wires and net names wherever required. Finally, the *to_verilog* method uses this data, along with the data about the pins, inputs and outputs, to generate the Verilog netlist of the design. The methods are described in Appendix A.

4.4 Generating the Placement Script

The next step is to generate the placement script which can be read by a place and route tool to adhere to the relative placement rules. The place and route tool considered is Synopsys® IC Compiler™. The placement script (a tool command language <.tcl> file) to be generated should thus be in the format specified in Section 3.2. We shall now see in brief how this is done.

The relative placement data is defined for the design as described in Section 4.2.2. A dictionary called *pl_data* is updated each time an entry is made. This can be seen in the example below for the origin.

```
def rp_origin(self,scell):
    self.pl_data[scell] = [0,0]
```

Listing 4.7: Updating dictionary for the origin

A cell is placed relative to another cell by one of the eight ways as seen in Figure 4.1. The dictionary is updated accordingly as can be seen in the example below:

```
def rp_rightof(self,scell1,scell2):
    try:
        self.pl_data[scell2] = [self.pl_data[scell1][0], \
        self.pl_data[scell1][1] + 1]
    except KeyError:
        sys.exit("ERROR : Cell placed relative to unplaced cell")
```

Listing 4.8: Updating dictionary: right of cell

Here, *scell2* is to be placed to the right of *scell1* and the coordinates in the dictionary are updated accordingly.

A cell cannot be placed relative to an unplaced cell. If an user tries to do so, an exception is generated warning the user as can be seen in the code above.

With these data we are ready to write the placement script. But first, we need to make sure that there are no overlapping cells, i.e., no two cells have the same coordinates. An easy way to check this is to uniquify the list of coordinates and compare with the original list. If the sizes of the two lists are not the same, then there are overlapping coordinates. This is implemented as shown below.

```
coords = [v for v in self.pl_data.values()]
uniq_coords = [v for c,v in enumerate(coords) \
if v not in coords[:c]]
if len(uniq_coords) != len(coords):
    sys.exit("ERROR : Overlapping Cells")
```

Listing 4.9: Checking for overlapping cells

In the way that the placement information is entered according to section 4.2.2, the first cell is assigned the coordinates of (0,0). Cells may then be placed either to the left of right of it. This results in cells being assigned negative coordinates, something that the place and route tool does not support. Thus, the coordinates of all cells must be offset in such a manner that there exists no negative coordinates, while still preserving the relative placement information. This is taken care of in the code.

Finally, with the corrected coordinates, the placement script can be written out in the format specified in Section 3.2. This is shown in the method *write_tcl* in Appendix A.

4.5 Placement Scripts for Hierarchical Designs

The method *write_tcl* as described in Section 4.4 would only work for non-hierarchical designs. As many common logic structures are indeed hierarchical in nature, we must have a provision for generating placement scripts for hierarchical designs as well. As mentioned in Section 3.2, there exists a convention for doing that. However, the format of the script is somewhat different from that of the non-hierarchical designs. In particular, there are different sets of commands for specifying leaf and non-leaf cells. We must, therefore, modify the above method to incorporate these changes.

At first, we need to change the way the method is called, as it has to support both non-hierarchical, and hierarchical designs with an arbitrary level of hierarchy. Thus, the *write_tcl* method is modified to accept an arbitrary number of variables. This is shown below

```
def write_tcl(self,*args):
```

Listing 4.10: Hierarchical Designs: write_tcl

An example of a function call using the above method is given below

```
cell_hier.write_tcl(leaf,next,last_but_top)
```

Listing 4.11: Hierarchical Designs: function call

Here, *cell_hier* (i.e. *Self* in Python parlance), is the top cell, *leaf* is the leaf level hierarchy, *next* is the next level, etc, until the *last_but_top* level.

While working with this modified method, we first need to check if the design is hierarchical or not. This can be done by checking for *args*. If the size of *args* is zero, the design is non-hierarchical and we can proceed as before.

```
if len(args) == 0: #Non-hierarchical design
```

Listing 4.12: Checking for hierarchical design

If the design is indeed hierarchical, the placement data for the *leaf*, i.e. *args[0]* is written out at first, in the same format as before. Next, the remaining *args* are iterated over and the

hierarchical placement data is written out. Finally, the top level module, i.e. *Self* is considered, and the hierarchical placement data is written out. These can be seen in the modified def *write_tcl(self,*args):* method in Appendix A.

4.6 Generating Images

We next go on to describe the process by which the relative placement information is expressed in a visual form by outputting an image. The image has to be an exact representation of how the design would look in the layout once the relative placement scripts are incorporated into the place and route tool.

In this work, images are generated in the SVG format. SVG stands for Scalable Vector Graphics, and it defines vector graphics in an XML format [21]. This allows for the SVG images to be easily scripted. The Python library called *svgwrite* [10] is used to easily generate SVG files from within Python.

To proceed with the image generation, at first, the coordinate data must be offset for negative values, as mentioned before. Next, the widths of each row and column must be figured out. This number depends on the cell with the maximum width and height in each row and column, as shown in Figure 4.2. Column 1 and Row 3 are larger here due to the larger cells.

The origins in IC Compiler™ and SVG are defined differently, as shown in Figure 4.3. Thus, changes are made to the way the coordinates are defined before writing out the SVG data.

With these considerations kept in mind, the design data is iterated over and the SVG data for the cells are written out using information about their dimensions, the data about heights and widths of rows and columns and the coordinates of the cells. These can be seen in the method *write_svg* in Appendix A.

4.7 Images for Hierarchical Designs

The above method is extended to support SVG generation for hierarchical designs. This is done in a similar fashion to that of the generation of the relative placement script for hierarchical designs. The method is defined with a variable number of arguments as shown below

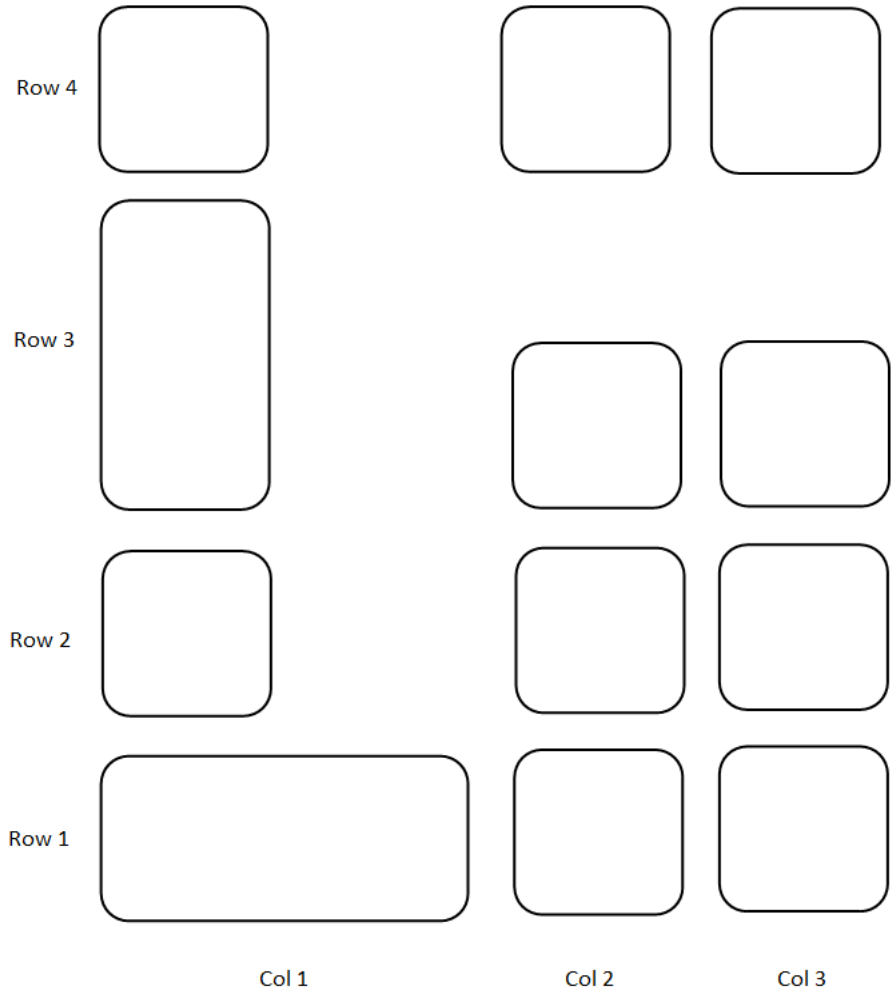

Figure 4.2: Widths and Heights of Rows and Columns

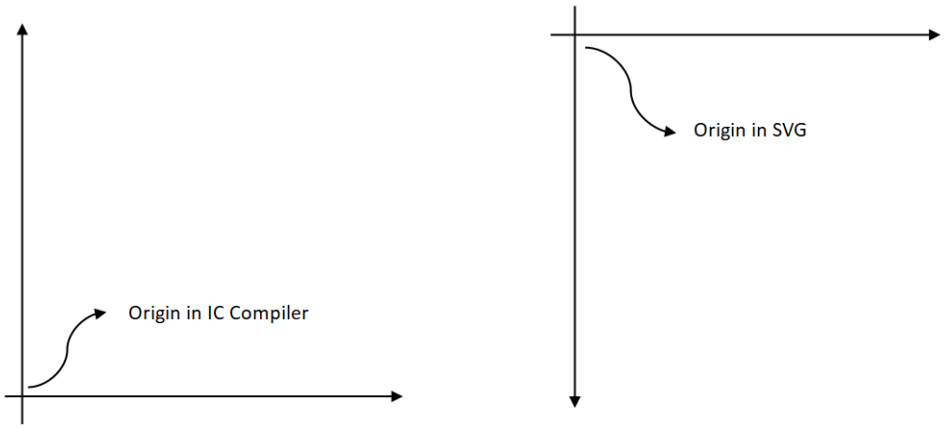

Figure 4.3: Origins in IC Compiler™ and SVG

```
def write_svg(self,*args):
```

Listing 4.13: SVG generation for hierarchical designs

where the non-top level cells are passed through *args*.

The code is then iterated over *Self* and all the *args* and images are generated for each.

4.8 Selecting Libraries on the Fly

The final aspect of the framework is to be able to change the library of the standard cells used easily with the help of a parameter. This can be useful to a designer as he/she can quickly generate different netlists and placement scripts for many different libraries. The designer is also relieved from worrying about specifying the exact names of the library cells while defining the design and can use generic names like *AND2*, *OR2*, etc. An example of a design as entered by a designer is shown below

```
self.u_and0 = AND2("u_and0")
self.u_and1 = AND2("u_and1")
self.u_and2 = AND2("u_and2")
self.u_or0  = OR3("u_or0")
self.u_xor0  = XOR2("u_xor0")
self.u_xor1  = XOR2("u_xor1")
```

Listing 4.14: Design entry in simplified form

where the cells are mapped to the generic standard cells.

To be able to generate the Verilog netlist with the actual library data, the mapping information between the generic cells and the library cells needs to be captured. This is done by two methods, *change_module* and *change_pin*. The change module method parses the Verilog netlist and changes the cells are per the definitions. A snippet of the code is shown below

```
def change_module(self,lib_name,module_name):

    if lib_name == "foundry_x_hvt_1V0SS-40C":
        if module_name == "AND2":
```

```
            return "AN2D1BWP7THVT"
        if module_name == "OR3":
            return "OR3D1BWP7THVT"
```

Listing 4.15: Change module

Here, if the selected library is *foundry_x_hvt_1V0SS-40C*, then the cells, whenever encountered, are changed accordingly.

As different cells of different libraries may have different pin names, it is also important to make sure that the pin names are changed appropriately. This is done by using the change pin module. A snippet is shown below

```
def change_pin(self,lib_name,pin_name,cell_name):

    if lib_name == "foundry_x_hvt_1V0SS-40C":
        if cell_name == "AND2":
            if pin_name == "A":
                return "A1"
            if pin_name == "B":
                return "A2"
```

Listing 4.16: Change pin

Here, after the library is matched, the cell needs to be matched. Then, each pin is searched for and the replacements are done. The complete methods are listed in Appendix A.

Once these methods have been defined, the write to verilog function is modified and called as shown below to generate the netlists with the appropriate library cells. The library name is passed into the function as an argument.

```
txt = cell.to_verilog("foundry_x_hvt_1V0SS-40C")
```

Listing 4.17: Verilog generation with modified library

This completes the creation of the required framework.

Chapter 5

Results

We would take a look at the results obtained by using the software framework on three simple designs. The designs used are a full adder, a hierarchical 8-bit adder made using the full adder, and a ring oscillator. We would first present the data used to define the design. We would then present the generated Verilog netlist, relative placement script and images. Finally, we would have a look at the layout in Synopsys® IC Compiler™ when the netlists and the placement scripts are read in. We would also compare the layouts to the layouts generated without using the placement data.

5.1 Design Inputs

Adder

```
class Adder(Cell):
    size_x = 5
    size_y = 2
    def __init__(self,instance):
        super(Adder,self).__init__(instance)
        self.A  = Input("A")
        self.B  = Input("B")
        self.CI = Input("CI")
        self.S  = Output("S")
        self.CO = Output("CO")

        self.u_and0 = AND2("u_and0")
        self.u_and1 = AND2("u_and1")
```

```
self.u_and2 = AND2("u_and2")
self.u_or0  = OR3("u_or0")
self.u_xor0  = XOR2("u_xor0")
self.u_xor1  = XOR2("u_xor1")

# First AND gate
self.connect(self.A,self.u_and0.A)
self.connect(self.B,self.u_and0.B)
# 2nd AND gate
self.connect(self.A,self.u_and1.A)
self.connect(self.CI,self.u_and1.B)
# 3rd AND gate
self.connect(self.B,self.u_and2.A)
self.connect(self.CI,self.u_and2.B)

# OR  gate
self.connect(self.u_and0.Z,self.u_or0.A)
self.connect(self.u_and1.Z,self.u_or0.B)
self.connect(self.u_and2.Z,self.u_or0.C)
self.connect(self.u_or0.Z, self.CO)

# XOR gate #0
self.connect(self.A,self.u_xor0.A)
self.connect(self.B,self.u_xor0.B)

# XOR gate #1
self.connect(self.u_xor0.Z,self.u_xor1.A)
self.connect(self.CI,self.u_xor1.B)
self.connect(self.u_xor1.Z,self.S)

# Relative placement data
self.rp_origin(self.u_and0)
self.rp_rightof(self.u_and0,self.u_or0)
self.rp_rightof(self.u_or0,self.u_xor0)
self.rp_topof(self.u_and0,self.u_and1)
self.rp_topof(self.u_or0,self.u_xor1)
self.rp_topof(self.u_xor0,self.u_and2)
```

Listing 5.1: Adder

8-bit Adder

```
class Adder_8bit(Cell):
    def __init__(self,instance):
        super(Adder_8bit,self).__init__(instance)

        self.A0  = Input("A0")
        self.B0  = Input("B0")
        self.A1  = Input("A1")
        self.B1  = Input("B1")
        self.A2  = Input("A2")
        self.B2  = Input("B2")
        self.A3  = Input("A3")
        self.B3  = Input("B3")
        self.A4  = Input("A4")
        self.B4  = Input("B4")
        self.A5  = Input("A5")
        self.B5  = Input("B5")
        self.A6  = Input("A6")
        self.B6  = Input("B6")
        self.A7  = Input("A7")
        self.B7  = Input("B7")
        self.CI  = Input("CI")
        self.S0  = Output("S0")
        self.S1  = Output("S1")
        self.S2  = Output("S2")
        self.S3  = Output("S3")
        self.S4  = Output("S4")
        self.S5  = Output("S5")
        self.S6  = Output("S6")
        self.S7  = Output("S7")
        self.CO  = Output("CO")

        self.u_adder0 = Adder("u_adder0")
        self.u_adder1 = Adder("u_adder1")
        self.u_adder2 = Adder("u_adder2")
        self.u_adder3  = Adder("u_adder3")
        self.u_adder4  = Adder("u_adder4")
```

```
        self.u_adder5  = Adder("u_adder5")
        self.u_adder6  = Adder("u_adder6")
        self.u_adder7  = Adder("u_adder7")

        # First Adder
        self.connect(self.A0,self.u_adder0.A)
        self.connect(self.B0,self.u_adder0.B)
        self.connect(self.S0,self.u_adder0.S)
        self.connect(self.CI,self.u_adder0.CI)

        # Second  Adder
        self.connect(self.A1,self.u_adder1.A)
        self.connect(self.B1,self.u_adder1.B)
        self.connect(self.S1,self.u_adder1.S)
        self.connect(self.u_adder1.CI,self.u_adder0.CO)

        # Third Adder
        self.connect(self.A2,self.u_adder2.A)
        self.connect(self.B2,self.u_adder2.B)
        self.connect(self.S2,self.u_adder2.S)
        self.connect(self.u_adder2.CI,self.u_adder1.CO)

    # Forth Adder
        self.connect(self.A3,self.u_adder3.A)
        self.connect(self.B3,self.u_adder3.B)
        self.connect(self.S3,self.u_adder3.S)
        self.connect(self.u_adder3.CI,self.u_adder2.CO)

        # Fifth Adder
        self.connect(self.A4,self.u_adder4.A)
        self.connect(self.B4,self.u_adder4.B)
        self.connect(self.S4,self.u_adder4.S)
        self.connect(self.u_adder4.CI,self.u_adder3.CO)

        # Sixth Adder
        self.connect(self.A5,self.u_adder5.A)
        self.connect(self.B5,self.u_adder5.B)
        self.connect(self.S5,self.u_adder5.S)
        self.connect(self.u_adder5.CI,self.u_adder4.CO)
```

```
        # Seventh Adder
        self.connect(self.A6, self.u_adder6.A)
        self.connect(self.B6, self.u_adder6.B)
        self.connect(self.S6, self.u_adder6.S)
        self.connect(self.u_adder6.CI, self.u_adder5.CO)

        # Eighth Adder
        self.connect(self.A7, self.u_adder7.A)
        self.connect(self.B7, self.u_adder7.B)
        self.connect(self.S7, self.u_adder7.S)
        self.connect(self.u_adder7.CI, self.u_adder6.CO)
        self.connect(self.CO, self.u_adder7.CO)

        # Relative placement data
        self.rp_origin(self.u_adder0)
        self.rp_rightof(self.u_adder0, self.u_adder1)
        self.rp_rightof(self.u_adder1, self.u_adder2)
        self.rp_rightof(self.u_adder2, self.u_adder3)
        self.rp_topof(self.u_adder0, self.u_adder4)
        self.rp_topof(self.u_adder1, self.u_adder5)
        self.rp_topof(self.u_adder2, self.u_adder6)
        self.rp_topof(self.u_adder3, self.u_adder7)
```

Listing 5.2: 8-bit Adder

Ring Oscillator

```
class Ring_Oscillator_simple(Cell):
    size_x = 12.6
    size_y = 1.4
    def __init__(self,instance):
        super(Ring_Oscillator_simple,self).__init__(instance)

        self.O = Output("O")

        self.u_inv0 = INV("u_inv0")
        self.u_inv1 = INV("u_inv1")
        self.u_inv2 = INV("u_inv2")
```

```
        self.u_inv3 = INV("u_inv3")
        self.u_inv4 = INV("u_inv4")
        self.u_inv5 = INV("u_inv5")
        self.u_inv6 = INV("u_inv6")
        self.u_inv7 = INV("u_inv7")
        self.u_inv8 = INV("u_inv8")

        self.connect(self.O, self.u_inv0.A)
        self.connect(self.u_inv0.Z, self.u_inv1.A)
        self.connect(self.u_inv1.Z, self.u_inv2.A)
        self.connect(self.u_inv2.Z, self.u_inv3.A)
        self.connect(self.u_inv3.Z, self.u_inv4.A)
        self.connect(self.u_inv4.Z, self.u_inv5.A)
        self.connect(self.u_inv5.Z, self.u_inv6.A)
        self.connect(self.u_inv6.Z, self.u_inv7.A)
        self.connect(self.u_inv7.Z, self.u_inv8.A)
        self.connect(self.u_inv8.Z, self.u_inv0.A)
        self.connect(self.u_inv8.Z, self.O)

        # Relative placement data
        self.rp_origin(self.u_inv0)
        self.rp_rightof(self.u_inv0, self.u_inv1)
        self.rp_rightof(self.u_inv1, self.u_inv2)
        self.rp_rightof(self.u_inv2, self.u_inv3)
        self.rp_rightof(self.u_inv3, self.u_inv4)
        self.rp_rightof(self.u_inv4, self.u_inv5)
        self.rp_rightof(self.u_inv5, self.u_inv6)
        self.rp_rightof(self.u_inv6, self.u_inv7)
        self.rp_rightof(self.u_inv7, self.u_inv8)
```

Listing 5.3: Ring Oscillator

5.2 Generated Verilog Netlists

Adder

```
module Adder (/*AUTOARG*/

input A,
```

```
input B,
input CI,
output S,
output CO
);
wire net0;
wire CO;
wire net2;
wire net3;
wire CI;
wire net1;
wire A;
wire S;
wire B;


AN2D1BWP7THVT u_and0 (
.A1(A),
.A2(B),
.Z(net0)
);


AN2D1BWP7THVT u_and1 (
.A1(A),
.A2(CI),
.Z(net1)
);


AN2D1BWP7THVT u_and2 (
.A1(B),
.A2(CI),
.Z(net2)
);


OR3D1BWP7THVT u_or0 (
.A1(net0),
```

```
.A2(net1),
.A3(net2),
.Z(CO)
);


XOR2D1BWP7THVT u_xor0 (
.A1(A),
.A2(B),
.Z(net3)
);


XOR2D1BWP7THVT u_xor1 (
.A1(net3),
.A2(CI),
.Z(S)
);


endmodule // Adder
```

Listing 5.4: Netlist: Adder

8-bit adder

```
module Adder_8bit (/*AUTOARG*/

input A0,
input B0,
input A1,
input B1,
input A2,
input B2,
input A3,
input B3,
input A4,
input B4,
```

```
input A5,
input B5,
input A6,
input B6,
input A7,
input B7,
input CI,
output S0,
output S1,
output S2,
output S3,
output S4,
output S5,
output S6,
output S7,
output CO
);
wire S5;
wire S4;
wire net3;
wire B5;
wire S0;
wire B0;
wire net5;
wire A7;
wire A3;
wire S2;
wire A0;
wire B6;
wire B1;
wire A5;
wire S3;
wire B4;
wire A1;
wire S1;
wire net0;
wire net2;
wire A6;
wire CI;
```

```
wire net1;
wire net4;
wire S6;
wire A4;
wire B7;
wire S7;
wire net6;
wire B3;
wire CO;
wire B2;
wire A2;


Adder u_adder0 (
.A(A0),
.B(B0),
.CI(CI),
.S(S0),
.CO(net0)
);


Adder u_adder1 (
.A(A1),
.B(B1),
.CI(net0),
.S(S1),
.CO(net1)
);


Adder u_adder2 (
.A(A2),
.B(B2),
.CI(net1),
.S(S2),
.CO(net2)
);
```

```
Adder u_adder3 (
.A(A3),
.B(B3),
.CI(net2),
.S(S3),
.CO(net3)
);


Adder u_adder4 (
.A(A4),
.B(B4),
.CI(net3),
.S(S4),
.CO(net4)
);


Adder u_adder5 (
.A(A5),
.B(B5),
.CI(net4),
.S(S5),
.CO(net5)
);


Adder u_adder6 (
.A(A6),
.B(B6),
.CI(net5),
.S(S6),
.CO(net6)
);


Adder u_adder7 (
.A(A7),
```

```
.B(B7),
.CI(net6),
.S(S7),
.CO(CO)
);


endmodule // Adder_8bit
```

Listing 5.5: Netlist: 8-bit adder

Ring Oscillator

```
module Ring_Oscillator_simple (/*AUTOARG*/


output O
);
wire net5;
wire O;
wire net6;
wire net3;
wire net1;
wire net4;
wire net0;
wire net7;
wire net2;


INVD1BWP7THVT u_inv0 (
.I(O),
.ZN(net0)
);


INVD1BWP7THVT u_inv1 (
.I(net0),
.ZN(net1)
```

```
);


INVD1BWP7THVT u_inv2 (
.I(net1),
.ZN(net2)
);


INVD1BWP7THVT u_inv3 (
.I(net2),
.ZN(net3)
);


INVD1BWP7THVT u_inv4 (
.I(net3),
.ZN(net4)
);


INVD1BWP7THVT u_inv5 (
.I(net4),
.ZN(net5)
);


INVD1BWP7THVT u_inv6 (
.I(net5),
.ZN(net6)
);


INVD1BWP7THVT u_inv7 (
.I(net6),
.ZN(net7)
);
```

```
INVD1BWP7THVT u_inv8 (
.I(net7),
.ZN(O)
);


endmodule // Ring_Oscillator_simple
```

Listing 5.6: Netlist: Ring Oscillator

It can be seen that the Verilog netlists are generated exactly as expected. It can also be seen that the cell names have been replaced to the names corresponding to the chosen library.

5.3 Generated Relative Placement Scripts

Adder

```
create_rp_group rp_Adder -design Adder -columns 3 -rows 2
add_to_rp_group Adder::rp_Adder -leaf u_and0 -column 0 -row 0
add_to_rp_group Adder::rp_Adder -leaf u_or0 -column 1 -row 0
add_to_rp_group Adder::rp_Adder -leaf u_xor0 -column 2 -row 0
add_to_rp_group Adder::rp_Adder -leaf u_and1 -column 0 -row 1
add_to_rp_group Adder::rp_Adder -leaf u_xor1 -column 1 -row 1
add_to_rp_group Adder::rp_Adder -leaf u_and2 -column 2 -row 1
```

Listing 5.7: Relative placement script: Adder

8-bit Adder

```
create_rp_group rp_Adder -design Adder -columns 3 -rows 2
add_to_rp_group Adder::rp_Adder -leaf u_and0 -column 0 -row 0
add_to_rp_group Adder::rp_Adder -leaf u_or0 -column 1 -row 0
add_to_rp_group Adder::rp_Adder -leaf u_xor0 -column 2 -row 0
add_to_rp_group Adder::rp_Adder -leaf u_and1 -column 0 -row 1
add_to_rp_group Adder::rp_Adder -leaf u_xor1 -column 1 -row 1
add_to_rp_group Adder::rp_Adder -leaf u_and2 -column 2 -row 1


create_rp_group rp_Adder_8bit -design Adder_8bit -columns 4 -rows 2
```

```
add_to_rp_group Adder_8bit::rp_Adder_8bit -hierarchy Adder::rp_Adder -instance
    u_adder0 -column 0 -row 0
add_to_rp_group Adder_8bit::rp_Adder_8bit -hierarchy Adder::rp_Adder -instance
    u_adder1 -column 1 -row 0
add_to_rp_group Adder_8bit::rp_Adder_8bit -hierarchy Adder::rp_Adder -instance
    u_adder2 -column 2 -row 0
add_to_rp_group Adder_8bit::rp_Adder_8bit -hierarchy Adder::rp_Adder -instance
    u_adder3 -column 3 -row 0
add_to_rp_group Adder_8bit::rp_Adder_8bit -hierarchy Adder::rp_Adder -instance
    u_adder4 -column 0 -row 1
add_to_rp_group Adder_8bit::rp_Adder_8bit -hierarchy Adder::rp_Adder -instance
    u_adder5 -column 1 -row 1
add_to_rp_group Adder_8bit::rp_Adder_8bit -hierarchy Adder::rp_Adder -instance
    u_adder6 -column 2 -row 1
add_to_rp_group Adder_8bit::rp_Adder_8bit -hierarchy Adder::rp_Adder -instance
    u_adder7 -column 3 -row 1
```

Listing 5.8: Relative placement script: 8-bit adder

Ring Oscillator

```
create_rp_group rp_Ring_Oscillator_simple -design Ring_Oscillator_simple -columns 9
    -rows 1
add_to_rp_group Ring_Oscillator_simple::rp_Ring_Oscillator_simple -leaf u_inv0 -column
     0 -row 0
add_to_rp_group Ring_Oscillator_simple::rp_Ring_Oscillator_simple -leaf u_inv1 -column
     1 -row 0
add_to_rp_group Ring_Oscillator_simple::rp_Ring_Oscillator_simple -leaf u_inv2 -column
     2 -row 0
add_to_rp_group Ring_Oscillator_simple::rp_Ring_Oscillator_simple -leaf u_inv3 -column
     3 -row 0
add_to_rp_group Ring_Oscillator_simple::rp_Ring_Oscillator_simple -leaf u_inv4 -column
     4 -row 0
add_to_rp_group Ring_Oscillator_simple::rp_Ring_Oscillator_simple -leaf u_inv5 -column
     5 -row 0
add_to_rp_group Ring_Oscillator_simple::rp_Ring_Oscillator_simple -leaf u_inv6 -column
     6 -row 0
add_to_rp_group Ring_Oscillator_simple::rp_Ring_Oscillator_simple -leaf u_inv7 -column
     7 -row 0
```

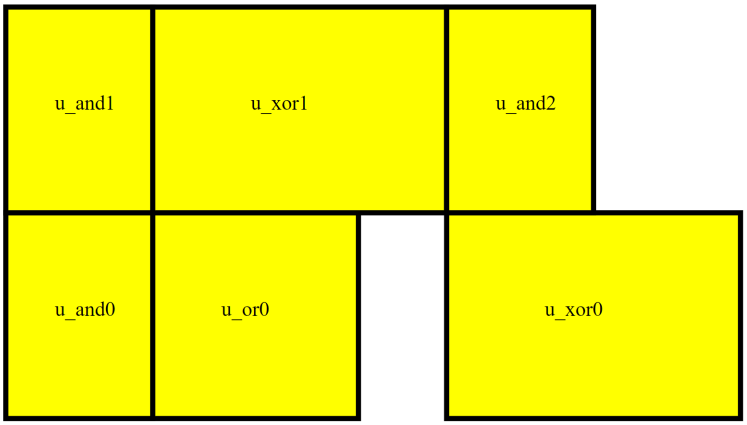

Figure 5.1: Generated Image: Adder

Figure 5.2: Generated Image: 8-bit Adder

```
add_to_rp_group Ring_Oscillator_simple::rp_Ring_Oscillator_simple -leaf u_inv8 -column
    8 -row 0
```

Listing 5.9: Relative placement script: Ring Oscillator

The relative placement scripts are generated as expected for both hierarchical and non-hierarchical designs.

5.4 Generated Images

Adder

The image is shown in Figure 5.1.

8-bit adder

The image is shown in Figure 5.2.

The image of an Adder (Figure 5.1) is also generated along with that of an 8-bit adder.

Ring Oscillator

The image is shown in Figure 5.3.

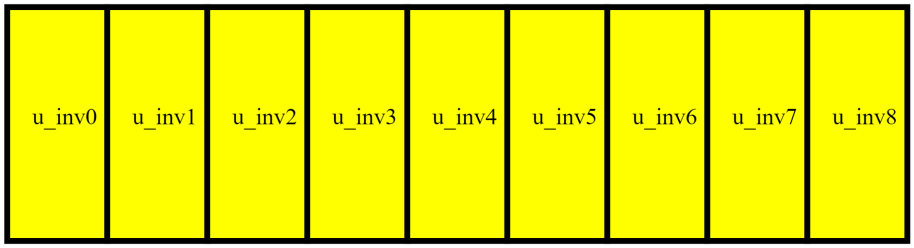

Figure 5.3: Generated Image: Ring Oscillator

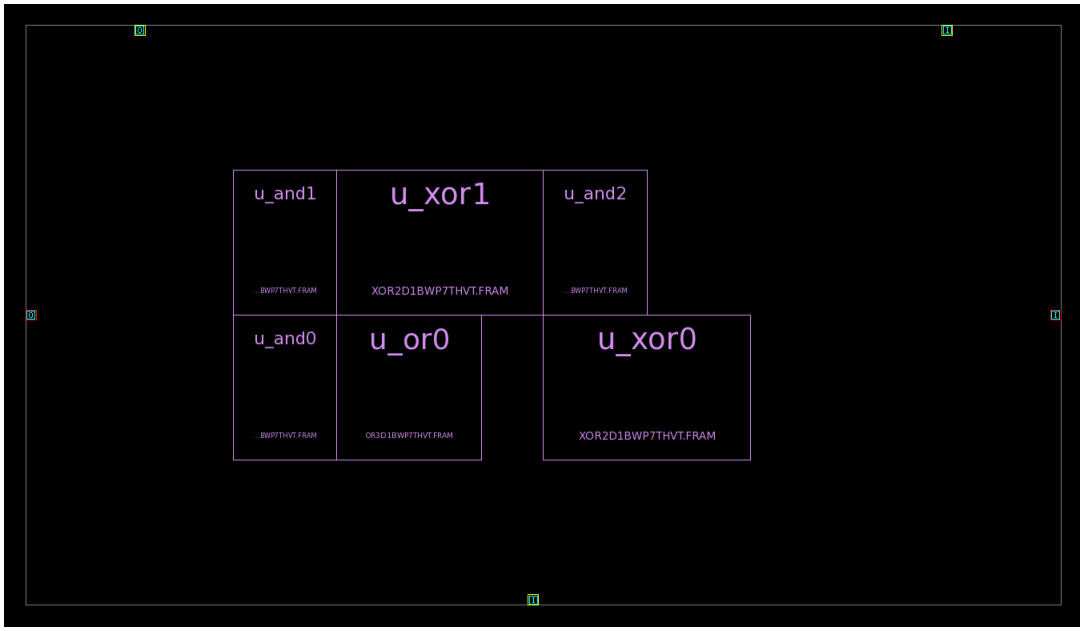

Figure 5.4: Adder, with placement script

5.5 Layout Results in Synopsys® IC Compiler™

The layouts in IC Compiler™ for all the designs, with and without using the relative placement script, and for both routed and non-routed cases, are shown from Figure 5.4 to Figure 5.15.

From these figures, it can be seen that the place and route tool is adhering to the relative placement scripts generated by the flow. It can also be seen that the images generated in Section 5.4 exactly match with the layout data. Finally, it can be observed that much compact designs with shorter and predictable wire lengths are obtained on providing the tool with the relative placement script as opposed to a free placement.

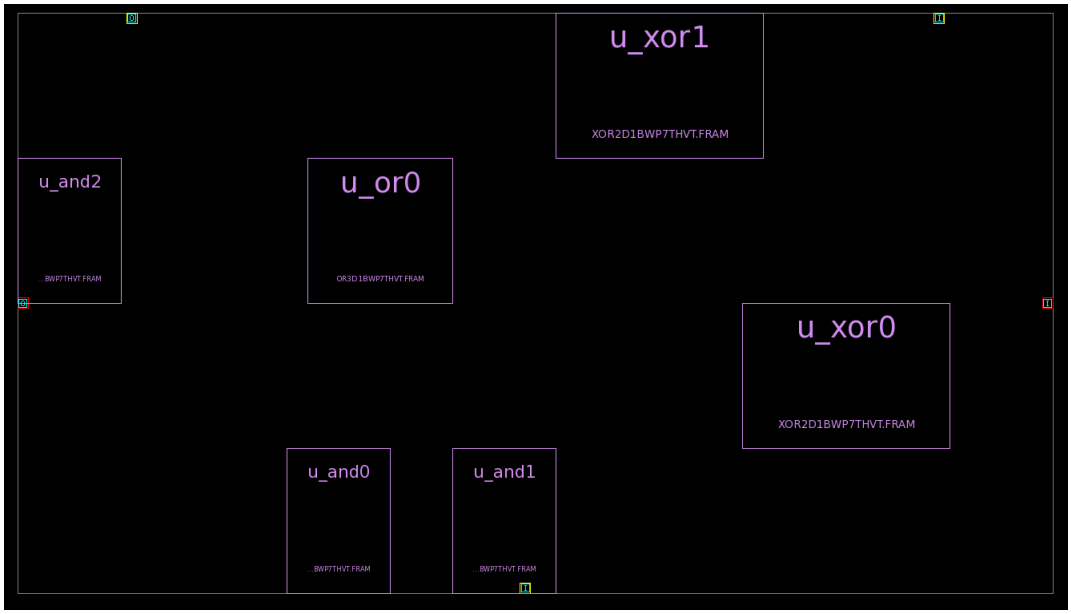

Figure 5.5: Adder, without placement script

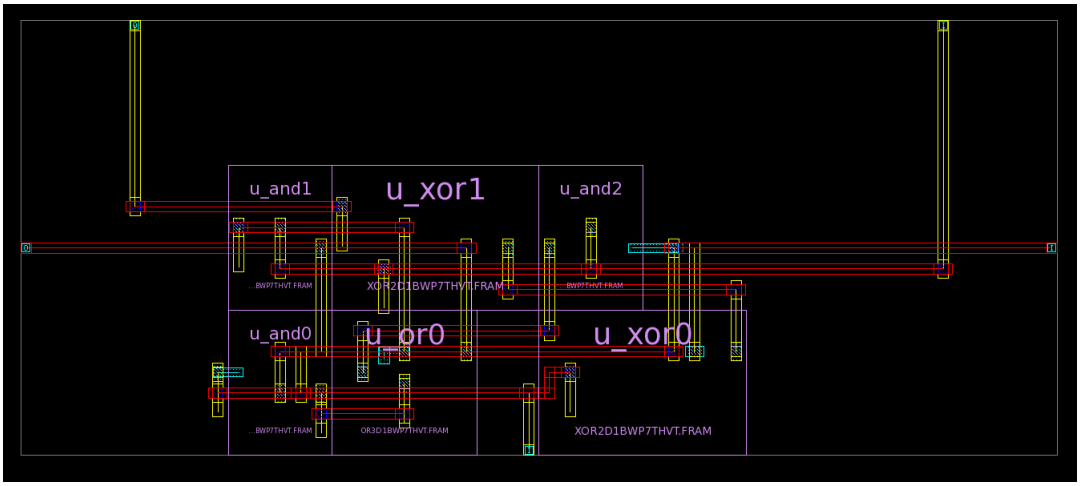

Figure 5.6: Adder, routed, with placement script

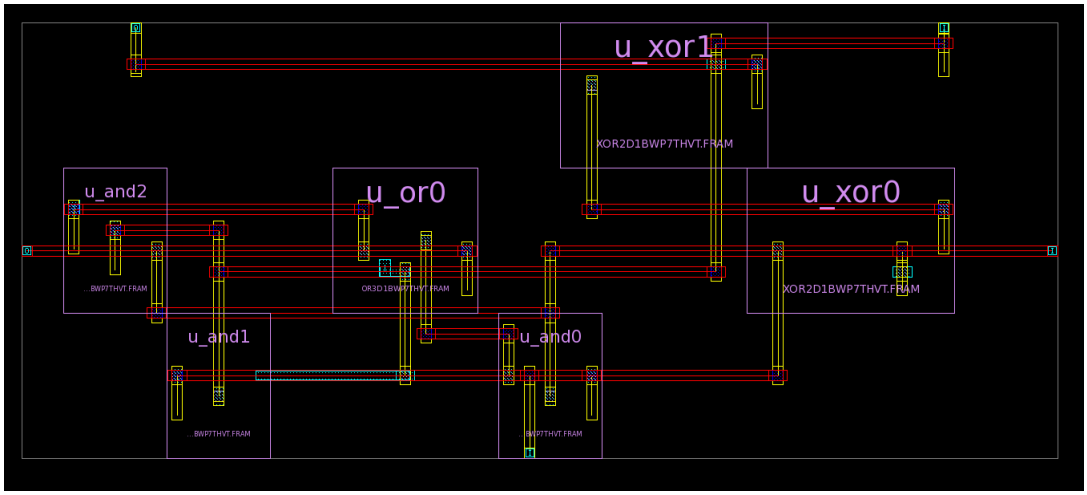

Figure 5.7: Adder, routed, without placement script

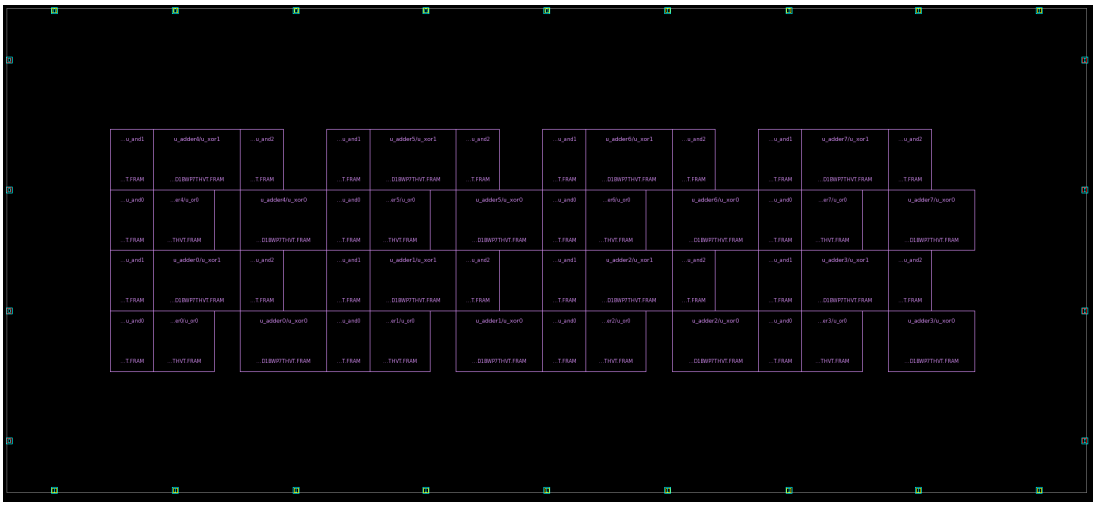

Figure 5.8: 8-bit adder, with placement script

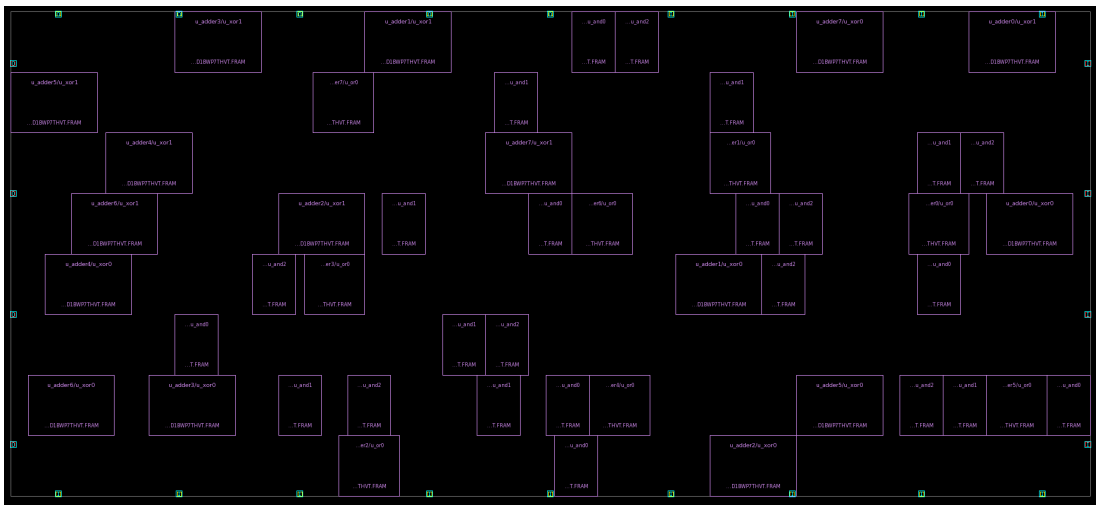

Figure 5.9: 8-bit adder, without placement script

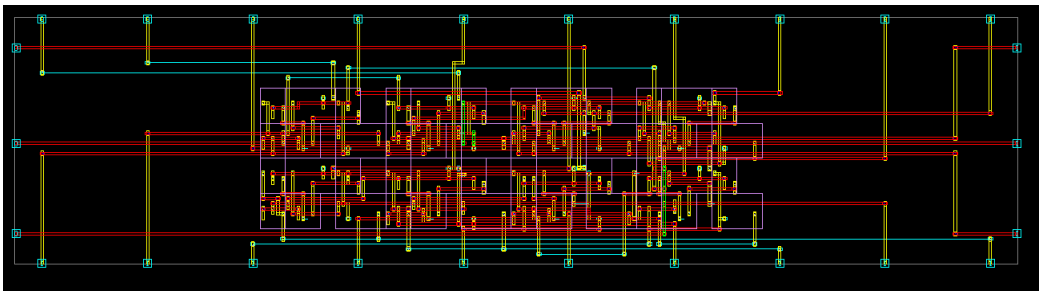

Figure 5.10: 8-bit adder, routed, with placement script

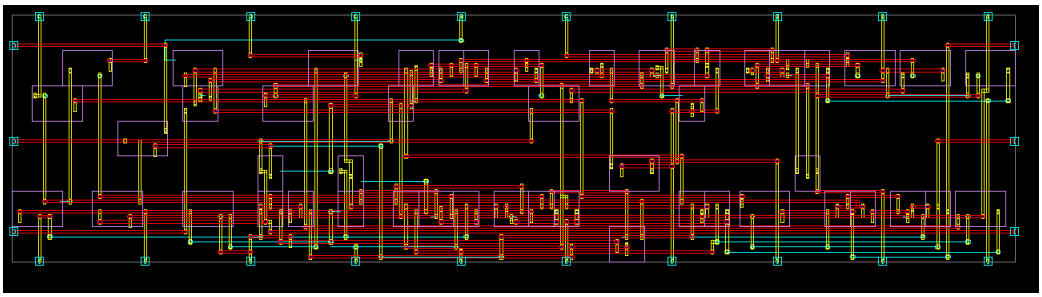

Figure 5.11: 8-bit adder, routed, without placement script

Figure 5.12: Ring Oscillator, with placement script

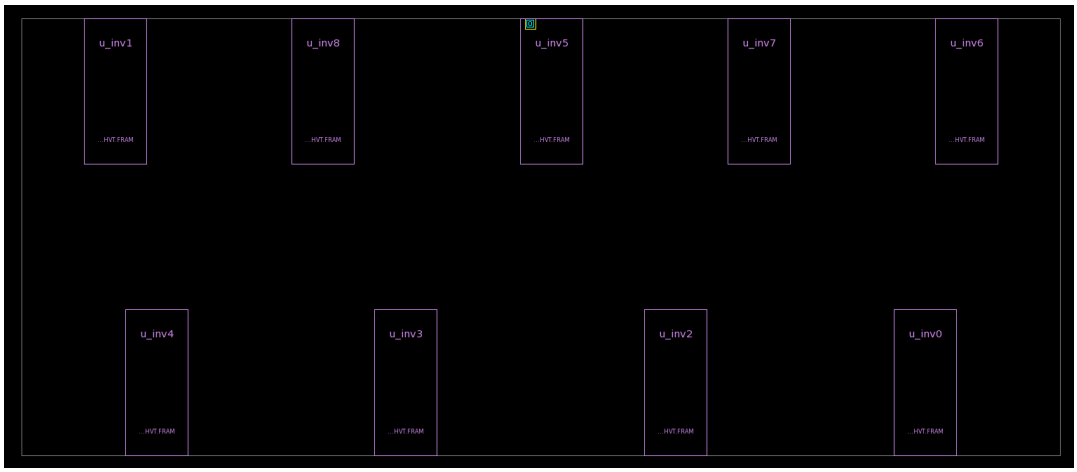

Figure 5.13: Ring Oscillator, without placement script

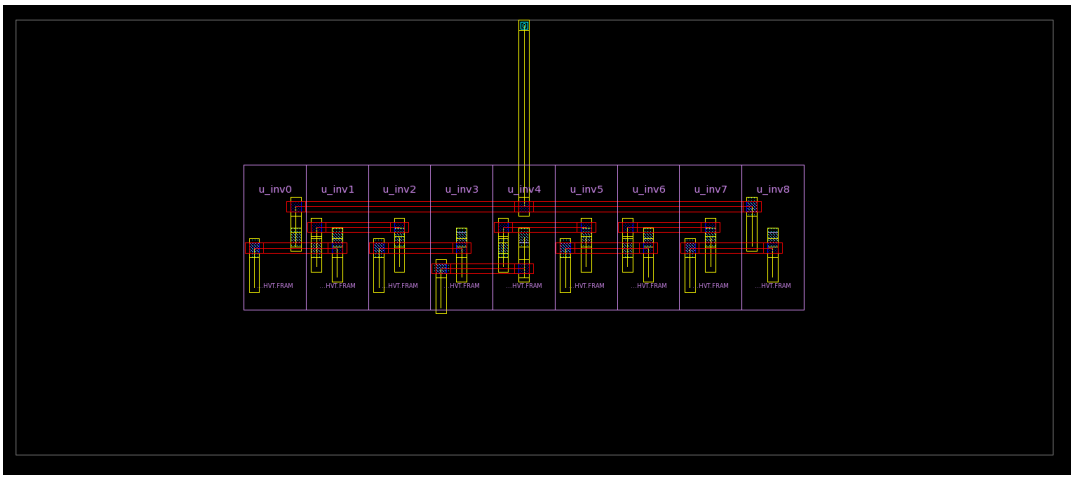

Figure 5.14: Ring Oscillator, routed, with placement script

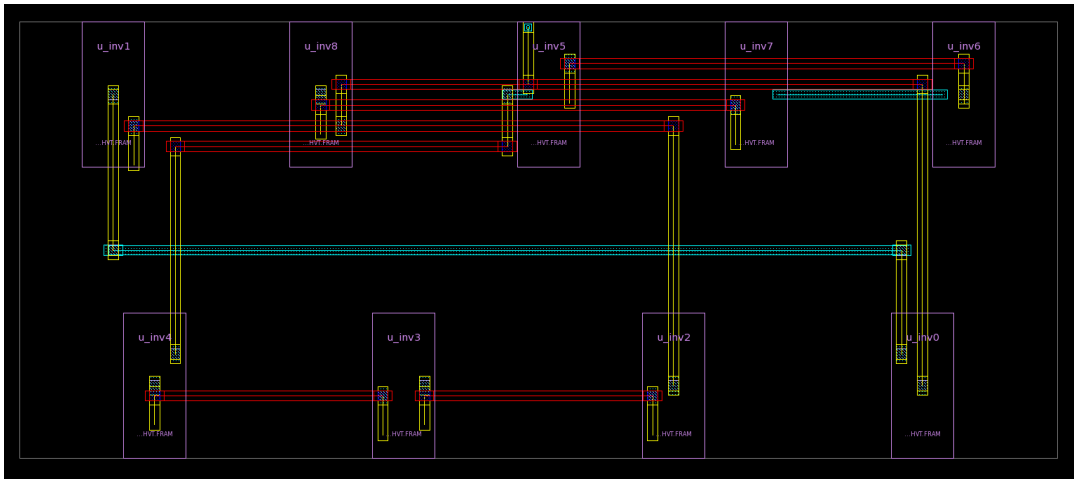

Figure 5.15: Ring Oscillator, routed, without placement script

Chapter 6

Conclusions

In this thesis, a software framework to seamlessly describe a digital circuit along with its placement data was developed. Verilog netlists, relative placement scripts and images corresponding to the placement were designed to be generated through the framework. The program was tested on three designs, and the results were obtained as expected. This framework can be used by designers to predictably design regularly structured circuits such that their performance is not compromised. All the objectives of the problem statement were met.

6.1 Future Work

As a followup to this thesis, the placement of regularly structured designs can be analysed within a much larger design. The framework developed can be used and the performance improvements from an unconstrained placement can be computed.

Bibliography

[1] ARATO, P., JUHASZ, S., MANN, Z. A., ORBAN, A., AND PAPP, D. Hardware-software partitioning in embedded system design. In *IEEE International Symposium on Intelligent Signal Processing, 2003* (Sept 2003), pp. 197–202.

[2] ASIC-WORLD. *Design and Tool Flow*. `http://www.asic-world.com/verilog/design_flow1.html`.

[3] BHATNAGAR, H. *Advanced ASIC Chip Synthesis: Using Synopsys Design Compiler Physical Compiler and Prime Time*, 2nd ed. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[4] ELES, P., PENG, Z., KUCHCINSKI, K., AND DOBOLI, A. System level hardware/software partitioning based on simulated annealing and tabu search. *Design Automation for Embedded Systems 2*, 1 (Jan 1997), 5–32.

[5] ELEYAN, N. N., LIN, K., KAMAL, M., MOHAMMAD, B., AND BASSETT, P. Semi-custom design flow: Leveraging place and route tools in custom circuit design. In *2009 IEEE International Conference on IC Design and Technology* (May 2009), pp. 143–147.

[6] GOLSHAN, K. *Physical Design Essentials: An ASIC Design Implementation Perspective*. Springer-Verlag, Berlin, Heidelberg, 2007.

[7] HENNESSY, J. L., AND PATTERSON, D. A. *Computer Architecture, Fifth Edition: A Quantitative Approach*, 5th ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2011.

[8] KLING, R.-M., AND BANERJEE, P. Esp: A new standard cell placement package using simulated evolution. In *Proceedings of the 24th ACM/IEEE Design Automation Conference* (New York, NY, USA, 1987), DAC '87, ACM, pp. 60–66.

[9] NAJM, M. Y. Z. F. N. Extreme delay sensitivity and the worst-case switching activity in vlsi circuitsy. In *32nd Design Automation Conference* (1995), pp. 623–627.

[10] PYPI. *svgwrite*. `https://pypi.org/project/svgwrite/`.

[11] SHAHOOKAR, K., AND MAZUMDER, P. A genetic approach to standard cell placement using meta-genetic parameter optimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 9*, 5 (May 1990), 500–511.

[12] SMITH, M. J. S. *Application-specific Integrated Circuits*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[13] STITT, G., LYSECKY, R., AND VAHID, F. Dynamic hardware/software partitioning: a first approach. In *Proceedings 2003. Design Automation Conference (IEEE Cat. No.03CH37451)* (June 2003), pp. 250–255.

[14] STRAAYER, M. Z., AND PERROTT, M. H. A multi-path gated ring oscillator tdc with first-order noise shaping. *IEEE Journal of Solid-State Circuits 44*, 4 (April 2009), 1089–1098.

[15] SUARIS, P. R., AND KEDEM, G. An algorithm for quadrisection and its application to standard cell placement. *IEEE Transactions on Circuits and Systems 35*, 3 (Mar 1988), 294–303.

[16] SUN, L., AND KWASNIEWSKI, T. A. A 1.25-ghz 0.35- mu;m monolithic cmos pll based on a multiphase ring oscillator. *IEEE Journal of Solid-State Circuits 36*, 6 (Jun 2001), 910–916.

[17] SYNOPSYS. *High Performance Simulation*. `https://www.synopsys.com/verification/simulation.html`.

[18] SYNOPSYS. *IC Compiler™ Implementation User Guide*, version m-2016.12, december 2016 ed. Synopsys, Inc., Mountain View, CA, USA, 2016.

[19] SYNOPSYS. *Design Compiler® User Guide*, version n-2017.09, september 2017 ed. Synopsys, Inc., Mountain View, CA, USA, 2017.

[20] SYNOPSYS. *Formality ® User Guide*, version n-2017.09, september 2017 ed. Synopsys, Inc., Mountain View, CA, USA, 2017.

[21] W3C. *W3C SVG Working Group*. `https://www.w3.org/Graphics/SVG/`.

[22] WEIGANDT, T. C., KIM, B., AND GRAY, P. R. Analysis of timing jitter in cmos ring oscillators. In *Proceedings of IEEE International Symposium on Circuits and Systems - ISCAS '94* (May 1994), vol. 4, pp. 27–30 vol.4.

[23] WESTE, N., AND HARRIS, D. *CMOS VLSI Design: A Circuits and Systems Perspective*, 4th ed. Addison-Wesley Publishing Company, USA, 2010.

[24] YILDIZ, M. C., AND MADDEN, P. H. Global objectives for standard cell placement. In *Proceedings of the 11th Great Lakes Symposium on VLSI* (New York, NY, USA, 2001), GLSVLSI '01, ACM, pp. 68–72.

Appendix A

Parts of the Code

The class *Cell*, which contains most of the logic in this work, is presented below.

```
class Cell(_Cell):
    size_x = None
    size_y = None

    def __init__(self,instance_name):
        self.instance_name = instance_name
        self.l_pos_x = None # logical position
        self.l_pos_y = None
        self.connections = []
        self.wires = dict() # a dictionnary, indexed by pins, to track net names
        self.row_n = 0
        self.col_n = 0
        self.pl_data = dict() # a dictionary to keep track of the placement data
        self.new_pl_data = dict() # a dictionary to keep track of the changed
            placement data
        self.svg_col_data = dict() # a dictionary to keep track of the widths of
            coloumns
        self.svg_row_data = dict() # a dictionary to keep track of the heights of rows


    def connect(self,pin1,pin2):
        self.connections.append((pin1,pin2))
        pass

    def get_pins(self,ofType=Pin):
```

```
        return [v  for k,v in vars(self).items() if isinstance(v,ofType)]

    def get_pins_of_instance(self,instance,ofType=Pin):
        return instance.get_pins(ofType)



    def get_instances(self,ofType=_Cell):
        return [v  for k,v in vars(self).items() if isinstance(v,ofType)]

    def update_connection(self):
        """Update wires dictionnary to assign wire name to every pin"""
        # all inputs/outputs pins are connected to internal cells
        # through wires that have the same name as the pin
        ios = self.get_pins()
        for io in ios:
            self.wires[io] = io.name

        prefix = "net{}"
        net_idx = 0
        for c1,c2 in self.connections:
            c1_connected = (c1 in self.wires)
            c2_connected = (c2 in self.wires)
            both_already_connected = c1_connected  and c2_connected
            one_already_connected = c1_connected ^ c2_connected
            if both_already_connected:
                if self.wires[c1] != self.wires[c2]:
                    print("-E- Error - Found connected pins, already connected")
                    print("-E- to different wires")
                    print("-E- 1) {}::{}".format(c1.name,self.wires[c1]))
                    print("-E- 2) {}::{}".format(c2.name,self.wires[c2]))
                    exit(1)
            elif one_already_connected:
                if c1_connected:
                    self.wires[c2] = self.wires[c1]
                else:
                    self.wires[c1] = self.wires[c2]
            else:
                self.wires[c1] = prefix.format(net_idx)
                self.wires[c2] = prefix.format(net_idx)
```

```
            net_idx = net_idx + 1


def to_verilog(self,lib_type):
    self.update_connection()
    instances = self.get_instances()

    d = dict()
    txt = ""
    d['body'] = ""
    d['wires'] = ""
    d['module_name'] = type(self).__name__

    for inst in instances:
        d1 = dict()
        print("-D- Instances {} : of type {}".format(inst.instance_name,
                                                     type(inst).__name__))
        d1['module'] = self.change_module(lib_type,type(inst).__name__)
        d1['inst']   = inst.instance_name
        d1['connectivity']   = ""

        pins = self.get_pins_of_instance(inst)
        connections = []
        for pin in pins:
            wire = self.wires[pin]
            new_pin_name = self.change_pin(lib_type,pin.name,type(inst).__name__)
            connections.append( "\n.{}({})".format(new_pin_name,wire))
        d1['connectivity'] = ','.join(connections)
        d['body'] += vt.tpl_inst.format(**d1)


    inputs = ["input {},".format(x.name ) for x in self.get_pins(ofType=Input)]
    outputs = ["output {},".format(x.name ) for x in self.get_pins(ofType=Output)]
    #print(outputs)
    last_out = outputs[-1]
    #print(last_out)
    last_out = re.sub(r',',"",last_out)
    #print(last_out)
    outputs[-1] = last_out
```

```
        #print(outputs)
        all_wires = ["wire {};".format(x) for x in set(self.wires.values())]
        d['inputs'] = '\n'.join(inputs)
        d['outputs'] = '\n'.join(outputs)
        d['wires'] = '\n'.join(all_wires)
        txt = vt.tpl_module.format(**d)
        return txt
##############################################updated_code#############################

    def input(self,name):
        return setattr(self,name,Input(name))

    def output(self,name):
        return setattr(self,name,Output(name))

    def inst(self,name,IPType):
        setattr(self,name,IPType)

    def get_pin(self,pin_name):
        return getattr(self,pin_name)


    def get_pin_of_instance(self,pin_name,instance):
        obj = getattr(self,instance)
        return getattr(obj,pin_name)



    def get_instances_path(self,separator='.',only_leaf_cells=  False):


        result =[]
        result.append(self.instance_name)
        instances=self.get_instances()
        for inst in instances:
            res = inst.get_instances_path(separator)
            res2 =[]
            for x in res:
                res2.append(self.instance_name +separator + x)
```

```
            if len(inst.get_instances()) ==0:
                only_leaves.append(inst.instance_name)


            result  += res2

        if only_leaf_cells:
            return only_leaves
        else:
            return result



def rp_origin(self,scell):
    self.pl_data[scell] = [0,0]



def rp_rightof(self,scell1,scell2):
    try:
        self.pl_data[scell2] = [self.pl_data[scell1][0], self.pl_data[scell1][1] +
             1]
    except KeyError:
        print("ERROR : Cell placed relative to unplaced cell")
        sys.exit("ERROR : Cell placed relative to unplaced cell")

def rp_topof(self,scell1,scell2):
    try:
        self.pl_data[scell2] = [self.pl_data[scell1][0] + 1, self.pl_data[scell1
            ][1]]
    except KeyError:
        print("ERROR : Cell placed relative to unplaced cell")
        sys.exit("ERROR : Cell placed relative to unplaced cell")

def rp_leftof(self,scell1,scell2):
    try:
        self.pl_data[scell2] = [self.pl_data[scell1][0], self.pl_data[scell1][1] -
             1]
    except KeyError:
```

```
            print("ERROR : Cell placed relative to unplaced cell")
            sys.exit("ERROR : Cell placed relative to unplaced cell")

    def rp_bottomof(self,scell1,scell2):
        try:
            self.pl_data[scell2] = [self.pl_data[scell1][0] - 1, self.pl_data[scell1
                ][1]]
        except KeyError:
            print("ERROR : Cell placed relative to unplaced cell")
            sys.exit("ERROR : Cell placed relative to unplaced cell")

    def rp_toprightof(self,scell1,scell2):
        try:
            self.pl_data[scell2] = [self.pl_data[scell1][0] + 1, self.pl_data[scell1
                ][1] + 1]
        except KeyError:
            print("ERROR : Cell placed relative to unplaced cell")
            sys.exit("ERROR : Cell placed relative to unplaced cell")

    def rp_topleftof(self,scell1,scell2):
        try:
            self.pl_data[scell2] = [self.pl_data[scell1][0] + 1, self.pl_data[scell1
                ][1] - 1]
        except KeyError:
            print("ERROR : Cell placed relative to unplaced cell")
            sys.exit("ERROR : Cell placed relative to unplaced cell")

    def rp_bottomrightof(self,scell1,scell2):
        try:
            self.pl_data[scell2] = [self.pl_data[scell1][0] - 1, self.pl_data[scell1
                ][1] + 1]
        except KeyError:
            print("ERROR : Cell placed relative to unplaced cell")
            sys.exit("ERROR : Cell placed relative to unplaced cell")

    def rp_bottomleftof(self,scell1,scell2):
        try:
            self.pl_data[scell2] = [self.pl_data[scell1][0] - 1, self.pl_data[scell1
                ][1] - 1]
```

```
    except KeyError:
        print("ERROR : Cell placed relative to unplaced cell")
        sys.exit("ERROR : Cell placed relative to unplaced cell")


def write_tcl(self,*args):
    if len(args) == 0:
        #check for overlaps
        coords = [v for v in self.pl_data.values()]
        uniq_coords = [v for c,v in enumerate(coords) if v not in coords[:c]]
        #print(uniq_coords)
        if len(uniq_coords) != len(coords):
            print("ERROR : Overlapping Cells")
            sys.exit("ERROR : Overlapping Cells")
        row_list = [i[0] for i in self.pl_data.values()]
        col_list = [i[1] for i in self.pl_data.values()]
        row_max = max(row_list)
        col_max = max(col_list)
        row_min = min(row_list)
        col_min = min(col_list)

        f = open(self.instance_name+"_rp.tcl", "w+")
        f.write("create_rp_group rp_%s -design %s -columns %d -rows %d\n" % (self.
            instance_name, self.instance_name, col_max - col_min + 1, row_max -
            row_min + 1))
        for k, v in self.pl_data.items():
            f.write("add_to_rp_group %s::rp_%s -leaf %s -column %d -row %d\n" % (
                self.instance_name, self.instance_name, k.instance_name, v[1] +
                abs(col_min), v[0] + abs(row_min)))

    else:

        coords = [v for v in args[0].pl_data.values()]
        uniq_coords = [v for c,v in enumerate(coords) if v not in coords[:c]]
        if len(uniq_coords) != len(coords):
            print("ERROR : Overlapping Cells")
            sys.exit("ERROR : Overlapping Cells")
        row_list = [i[0] for i in args[0].pl_data.values()]
        col_list = [i[1] for i in args[0].pl_data.values()]
```

```
row_max = max(row_list)
col_max = max(col_list)
row_min = min(row_list)
col_min = min(col_list)

f = open(self.instance_name+"_rp.tcl", "w+")
f.write("create_rp_group␣rp_%s␣-design␣%s␣-columns␣%d␣-rows␣%d\n" % (args
    [0].instance_name, args[0].instance_name, col_max - col_min + 1,
    row_max - row_min + 1))
for k, v in args[0].pl_data.items():
    f.write("add_to_rp_group␣%s::rp_%s␣-leaf␣%s␣-column␣%d␣-row␣%d\n" % (
        args[0].instance_name, args[0].instance_name, k.instance_name, v
        [1] + abs(col_min), v[0] + abs(row_min)))

for el in range(1,len(args)):

    coords = [v for v in args[el].pl_data.values()]
    uniq_coords = [v for c,v in enumerate(coords) if v not in coords[:c]]
    if len(uniq_coords) != len(coords):
        print("ERROR␣:␣Overlapping␣Cells")
        sys.exit("ERROR␣:␣Overlapping␣Cells")

    row_list = [i[0] for i in args[el].pl_data.values()]
    col_list = [i[1] for i in args[el].pl_data.values()]
    row_max = max(row_list)
    col_max = max(col_list)
    row_min = min(row_list)
    col_min = min(col_list)

    f.write("\n\n")

    f.write("create_rp_group␣rp_%s␣-design␣%s␣-columns␣%d␣-rows␣%d\n" % (
        args[el].instance_name, args[el].instance_name, col_max - col_min
        + 1, row_max - row_min + 1))
    for k, v in args[el].pl_data.items():
        f.write("add_to_rp_group␣%s::rp_%s␣-hierarchy␣%s::rp_%s␣-instance␣
            %s␣-column␣%d␣-row␣%d\n" % (args[el].instance_name, args[el].
            instance_name, args[el-1].instance_name, args[el-1].
            instance_name, k.instance_name, v[1] + abs(col_min), v[0] +
```

```
                    abs(row_min)))

        last_el = args[-1]
        coords = [v for v in self.pl_data.values()]
        uniq_coords = [v for c,v in enumerate(coords) if v not in coords[:c]]
        if len(uniq_coords) != len(coords):
            print("ERROR : Overlapping Cells")
            sys.exit("ERROR : Overlapping Cells")

        row_list = [i[0] for i in self.pl_data.values()]
        col_list = [i[1] for i in self.pl_data.values()]
        row_max = max(row_list)
        col_max = max(col_list)
        row_min = min(row_list)
        col_min = min(col_list)

        f.write("\n\n")

        f.write("create_rp_group rp_%s -design %s -columns %d -rows %d\n" % (self.
            instance_name, self.instance_name, col_max - col_min + 1, row_max -
            row_min + 1))
        for k, v in self.pl_data.items():
            f.write("add_to_rp_group %s::rp_%s -hierarchy %s::rp_%s -instance %s -
                column %d -row %d\n" % (self.instance_name, self.instance_name,
                last_el.instance_name, last_el.instance_name, k.instance_name, v
                [1] + abs(col_min), v[0] + abs(row_min)))



def change_module(self,lib_name,module_name):

    if lib_name == "tcbn55lpbwp7thvt_1V0SS-40C":
        if module_name == "AND2":
            return "AN2D1BWP7THVT"
        if module_name == "OR3":
            return "OR3D1BWP7THVT"
        if module_name == "XOR2":
            return "XOR2D1BWP7THVT"
        if module_name == "NAND2":
```

```
                return "ND2D1BWP7THVT"
            if module_name == "OR2":
                return "OR2D1BWP7THVT"
            if module_name == "INV":
                return "INVD1BWP7THVT"
            if module_name == "MUX2":
                return "MUX2D1BWP7THVT"

            else:
                return module_name

        if lib_name == "TSMC_LIB_001":
            if module_name == "AND2":
                return "TSMC_AND2"
            if module_name == "OR3":
                return "TSMC_OR3"
            if module_name == "XOR2":
                return "TSMC_XOR2"
            if module_name == "NAND2":
                return "TSMC_NAND2"
            if module_name == "OR2":
                return "TSMC_OR2"
            if module_name == "INV":
                return "TSMC_INV"
            if module_name == "MUX2":
                return "TSMC_MUX2"

            else:
                return module_name


        else:
            print("ERROR : The library name you have entered is not defined")
            sys.exit("ERROR : The library name you have entered is not defined")



    def change_pin(self,lib_name,pin_name,cell_name):
```

```
if lib_name == "tcbn55lpbwp7thvt_1V0SS-40C":
    if cell_name == "AND2":
        if pin_name == "A":
            return "A1"
        if pin_name == "B":
            return "A2"
        if pin_name == "Z":
            return "Z"

    if cell_name == "NAND2":
        if pin_name == "A":
            return "A1"
        if pin_name == "B":
            return "A2"
        if pin_name == "Z":
            return "ZN"


    if cell_name == "OR2":
        if pin_name == "A":
            return "A1"
        if pin_name == "B":
            return "A2"
        if pin_name == "Z":
            return "Z"

    if cell_name == "OR3":
        if pin_name == "A":
            return "A1"
        if pin_name == "B":
            return "A2"
        if pin_name == "C":
            return "A3"
        if pin_name == "Z":
            return "Z"


    if cell_name == "INV":
```

```
            if pin_name == "A":
                return "I"
            if pin_name == "Z":
                return "ZN"


        if cell_name == "XOR2":
            if pin_name == "A":
                return "A1"
            if pin_name == "B":
                return "A2"
            if pin_name == "Z":
                return "Z"


        if cell_name == "MUX2":
            if pin_name == "A":
                return "I0"
            if pin_name == "B":
                return "I1"
            if pin_name == "S":
                return "S"
            if pin_name == "Z":
                return "Z"

        else:
            return pin_name


    if lib_name == "TSMC_LIB_001":
        if cell_name == "AND2":
            if pin_name == "A":
                return "TSMC_A"
            if pin_name == "B":
                return "TSMC_B"
            if pin_name == "Z":
                return "TSMC_Z"
```

```
if cell_name == "NAND2":
    if pin_name == "A":
        return "TSMC_A"
    if pin_name == "B":
        return "TSMC_B"
    if pin_name == "Z":
        return "TSMC_Z"

if cell_name == "OR2":
    if pin_name == "A":
        return "TSMC_A"
    if pin_name == "B":
        return "TSMC_B"
    if pin_name == "Z":
        return "TSMC_Z"


if cell_name == "OR3":
    if pin_name == "A":
        return "TSMC_A"
    if pin_name == "B":
        return "TSMC_B"
    if pin_name == "C":
        return "TSMC_C"
    if pin_name == "Z":
        return "TSMC_Z"


if cell_name == "INV":
    if pin_name == "A":
        return "TSMC_A"
    if pin_name == "Z":
        return "TSMC_Z"


if cell_name == "XOR2":
    if pin_name == "A":
        return "TSMC_A"
    if pin_name == "B":
```

```
                    return "TSMC_B"
                if pin_name == "Z":
                    return "TSMC_Z"


            if cell_name == "MUX2":
                if pin_name == "A":
                    return "TSMC_A"
                if pin_name == "B":
                    return "TSMC_B"
                if pin_name == "S":
                    return "TSMC_S"
                if pin_name == "Z":
                    return "TSMC_Z"




            else:
                return pin_name

        else:
            print("ERROR : The library name you have entered is not defined")
            sys.exit("ERROR : The library name you have entered is not defined")


###SVG###

    def write_svg(self,*args):

        row_list = [i[0] for i in self.pl_data.values()]
        col_list = [i[1] for i in self.pl_data.values()]
        row_max = max(row_list)
        col_max = max(col_list)
        row_min = min(row_list)
        col_min = min(col_list)

        cols = col_max - col_min + 1
        rows = row_max - row_min + 1
```

```
for k, v in self.pl_data.items():
    self.new_pl_data[k] = [v[0] + abs(row_min), v[1] + abs(col_min)]

for i in range(cols):
    self.svg_col_data[i] = 0

for i in range(rows):
    self.svg_row_data[i] = 0

for k, v in self.new_pl_data.items():
    #print("%s :: %d :: %d" % (k.instance_name, v[0], v[1]))
    #pass
    if k.size_x > self.svg_col_data[v[1]]:
        self.svg_col_data[v[1]] = k.size_x

    if k.size_y > self.svg_row_data[v[0]]:
        self.svg_row_data[v[0]] = k.size_y



row_sum = 0
col_sum = 0
for i in range(rows):
    row_sum = row_sum + self.svg_row_data[i]
for i in range(cols):
    col_sum = col_sum + self.svg_col_data[i]

dwg = svgwrite.Drawing(filename=self.instance_name+".svg")
dwg.viewbox(minx=0-k.size_y/20,miny=0-k.size_y/20,width=col_sum+k.size_y/10,
    height=row_sum+k.size_y/10)
for k, v in self.new_pl_data.items():

    x_coord = 0
    y_coord = 0
    for i in range(v[1]):
        x_coord = x_coord + self.svg_col_data[i]

    for i in range(v[0]):
        y_coord = y_coord + self.svg_row_data[i]
```

```
        y_coord = row_sum - y_coord - k.size_y


        dwg.add(dwg.rect(insert=(x_coord*1,y_coord*1),size=(k.size_x*1,k.size_y*1)
            ,fill='yellow',stroke='black',stroke_width=k.size_y/40))
        dwg.add(dwg.text(k.instance_name,insert=(x_coord*1+((k.size_x*1)/4),
            y_coord*1+((k.size_y*1)/2)),font_size=k.size_y*1/10))
        #print("%s::%d::%d::%d::%d" % (k.instance_name,x_coord*100,y_coord*100,k.
            size_x*100,k.size_y*100))

    dwg.save()

    for a in args:

        row_list = [i[0] for i in a.pl_data.values()]
        col_list = [i[1] for i in a.pl_data.values()]
        row_max = max(row_list)
        col_max = max(col_list)
        row_min = min(row_list)
        col_min = min(col_list)

        cols = col_max - col_min + 1
        rows = row_max - row_min + 1

        for k, v in a.pl_data.items():
            a.new_pl_data[k] = [v[0] + abs(row_min), v[1] + abs(col_min)]

        for i in range(cols):
            a.svg_col_data[i] = 0

        for i in range(rows):
            a.svg_row_data[i] = 0

        for k, v in a.new_pl_data.items():
            #print("%s :: %d :: %d" % (k.instance_name, v[0], v[1]))
            #pass
            if k.size_x > a.svg_col_data[v[1]]:
                a.svg_col_data[v[1]] = k.size_x
```

```
        if k.size_y > a.svg_row_data[v[0]]:
            a.svg_row_data[v[0]] = k.size_y

row_sum = 0
col_sum = 0
for i in range(rows):
    row_sum = row_sum + a.svg_row_data[i]
for i in range(cols):
    col_sum = col_sum + a.svg_col_data[i]

dwg = svgwrite.Drawing(filename=a.instance_name+".svg")
dwg.viewbox(minx=0-k.size_y/20,miny=0-k.size_y/20,width=col_sum+k.size_y
    /10,height=row_sum+k.size_y/10)
for k, v in a.new_pl_data.items():

    x_coord = 0
    y_coord = 0
    for i in range(v[1]):
        x_coord = x_coord + a.svg_col_data[i]

    for i in range(v[0]):
        y_coord = y_coord + a.svg_row_data[i]

    y_coord = row_sum - y_coord - k.size_y

    dwg.add(dwg.rect(insert=(x_coord*1,y_coord*1),size=(k.size_x*1,k.
        size_y*1),fill='yellow',stroke='black',stroke_width=k.size_y/40))
    dwg.add(dwg.text(k.instance_name,insert=(x_coord*1+((k.size_x*1)/4),
        y_coord*1+((k.size_y*1)/2)),font_size=k.size_y*1/10))
    #print("%s::%d::%d::%d::%d" % (k.instance_name,x_coord*100,y_coord
        *100,k.size_x*100,k.size_y*100))

dwg.save()
```

Listing A.1: Parts of the Code