Enhancing and Using an Automatic Design System for Creating FPGAs

by

Aaron Charles Egier

A thesis submitted in conformity with the requirements
for the degree of Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto

Abstract

Enhancing and Using an Automatic Design System for Creating FPGAs

Aaron Charles Egier

Master of Applied Science

Graduate Department of Electrical and Computer Engineering

University of Toronto

2005

The creation of integrated circuits has progressed from custom design and layout to the less time-intensive implementation media of ASICs and FPGAs. FPGAs provide the lowest development cost and fastest development time; however, the design of the FPGA itself is still a time-consuming, expensive, custom layout task that takes at least 50 person-years to complete. This work explores new techniques to automate the design and layout of FPGAs. An existing automatic layout system is improved by changing the grouping of transistors that form the basic building blocks of the system. These improvements result in a 16.8% area savings over previous versions and only a 36% area increase compared to equivalent custom designs. The system was also extended to create the first automatic layout of an FPGA from a generic architecture description. These improvements and additions suggest that the automatic layout system is a viable alternative to custom layout of FPGAs.

Acknowledgements

I would like to thank my supervisor, Professor Jonathan Rose, for his advice and guidance in all aspects of this work and my education. Also, Ian Kuon deserves my profound thanks and gratitude for his achievements and co-operation that led to the completion of this work. This work would not have been possible without the people who worked on it before me. They are Ketan Padalia, Ryan Fung, Mark Bourgeault, Josh Slavkin, and Chris Sun.

I am grateful to Simon So for providing additional cell layouts and Kostas Pagiamtzis for sharing his knowledge and experience regarding the fabrication of chips through CMC. In addition, Professor Rose's students and all the students in LP392 have been extremely helpful by offering new perspectives on this work.

Funding for this project was provided by Altera and the NSERC CRD program. CMC granted us silicon area without which it would not have been possible for us to fabricate our FPGA. I received funding for my studies from the University of Toronto and NSERC.

Thanks to my parents for their support and encouragement in all aspects of my life. Last but not least, to Katherine, I hope I can help you reach your goals as well as you have helped me reach mine.

Table of Contents

List of Tables

List of Figures

List of Acronyms

ASIC	application-specific integrated circuit
BLE	basic logic element
CMC	Canadian Microelectronics Corporation
CMOS	complementary metal-oxide-semiconductor
DRC	design rule check
FPGA	field-programmable gate array
GILES	Good Instant Layout of Erasable Semiconductors
HDL	hardware description language
IC	integrated circuit
I/O	input/output
IP	intellectual property
LUT	look-up table
LVS	layout versus schematic
NMOS	n-channel metal-oxide-semiconductor
PGA	pin grid array

PMOS	p-channel metal-oxide-semiconductor
POWELL	Pushbutton Optimized Widely Erasable Logic Layout
SRAM	static random-access memory
TSMC	Taiwan Semiconductor Manufacturing Company
VLSI	very large-scale integration
VPR	Versatile Place and Route

Chapter 1

Introduction

The implementation of digital circuits has changed dramatically since the invention of the transistor in 1947 [1]. In the early years, circuits were designed with discrete components and connected on a circuit board. As the technology to create transistors improved, multiple transistors were integrated on a single chip to form logic gates. With time, entire circuits fit on a chip. When the size of integrated circuits (ICs) reached the scale of microprocessors, this level of complexity became know as very large-scale integration (VLSI) [2]. It became possible to create application-specific integrated circuits (ASICs) for unique applications. ASIC tools were developed to automate the design cycle and reduce the time to market [3]. Although these tools create ASICs that occupy more area, run at a slower speed, and consume more power compared to manually-designed, custom ICs, most applications are suitable to accept this tradeoff. Custom ICs are only used for the most advanced designs when area, speed, and power are critical.

In recent years, the complexity of IC fabrication processes has continued to increase requiring more optical lithography masks at an increased price. In addition, once the masks have been made, the cost to modify the IC for bug fixes or feature enhancements is similarly increased. This has created a market for a new class of ICs called field-programmable gate arrays (FPGAs) that can be programmed to implement any circuit [4]. The companies that create FPGAs incur the cost of fabrication and sell them to customers who are willing to spend more per chip to reduce development costs, risks, and manufacturing time. FPGAs also reduce costs because they can be reconfigured for bug fixes and feature enhancements. These savings makes FPGAs attractive even though they are slower and use more power than ASICs.

The market for FPGAs is increasing but so is their complexity and the time required to design them. To compete with ASICs, FPGA companies use custom IC design techniques to reduce area and power while increasing speed. The standard ASIC flow cannot meet these requirements. Thus, FPGA companies must spend on the order of at least 50 person-years to create new FPGAs.

Previous work at the University of Toronto has created a software tool that reduces the time to develop FPGAs [5]. The inputs to that system are an architectural description of the FPGA and the mask layouts of custom cells that implement the basic building blocks of the architecture. The output is a placed and routed netlist of the FPGA tile. The tile is repeated to create large FPGAs. Creating a new FPGA with a different architecture is achieved by changing a few lines in the input file and providing a different set of custom cells optimized for the new architecture. Therefore, the time to design an FPGA is no more than the time required to design the custom cells. This is dramatically less time than the full custom approach.

This tool, named Good Instant Layout of Erasable Semiconductors (GILES), contains a placer and router similar to those used in ASIC tools. However, it addresses some of their shortcomings without sacrificing the benefit of reduced time to market. ASIC tools use standard cells to implement basic logic functions. These cells are generic and not optimized for specific circuits [3]. Instead the custom cells used by GILES are optimized for each specific FPGA architecture. This hybrid custom/ASIC approach allows GILES to achieve silicon area smaller than ASIC tools and near that of custom designs. In addition, GILES places these cells more efficiently than ASIC tools by using optimizations specific to FPGAs.

1.1 Motivation and Goals

The first version of GILES obtained promising results. However, many assumptions were made that can be revisited and explored for improvement. One such assumption, which has a large impact on area, is the choice of custom cells. The first goal of this work is to revisit this choice to achieve area results closer to that of commercial FPGAs designed using custom flows. Part of that goal is to obtain more accurate results by improving the models needed to estimate the area of the custom cells.

The second goal of this work is to extend GILES to create a complete, fully-functional FPGA. In the previous version, many issues were ignored regarding the power grid, clock network, and programming circuitry that must be resolved. The overriding theme is to automate the entire process and to provide the flexibility to quickly implement alternative architectures.

1.2 Organization

The remainder of this dissertation is divided into four chapters. Chapter 2 provides a background on FPGA architecture, VLSI design methodologies, and prior work that contrasts with GILES. Chapter 3 describes tradeoffs in choosing the custom cells used by GILES and which choices achieve smaller chip area. Chapter 4 explains our proof-of-concept FPGA and the issues surpassed during the design process. Also, it discusses the infrastructure implemented to automate most of the steps involved. Finally, Chapter 5 concludes and provides suggestions for future work.

Chapter 2

Background

This chapter is divided into four sections. It begins with a review of FPGA architecture focusing on the features used in this work. It follows with a brief background on VLSI design methodologies. Then it examines other approaches to automatic layout of FPGAs. Finally, the chapter concludes with an explanation of the GILES system flow, which is the basis for the present work, and the previous results obtained with it.

2.1 FPGA Architecture

FPGAs are designed to implement any logic circuit. Their inherent reconfigurability derives from their programmable architecture. The main parts of the FPGA are the logic blocks, the input/output (I/O) blocks, and the routing [4]. All are programmable using static random-access memory (SRAM) to allow different connections and functionalities. Their arrangement for an island-style FPGA is shown in Figure 2.1. The logic blocks are distributed in an array with the routing fabric running between them. The I/O blocks surround the logic blocks and routing tracks. The direction of each I/O block and their connections to the routing tracks are configurable.

The logic blocks implement arbitrary logic functions. Connections are made between them with the programmable routing fabric. The inputs and outputs of the logic block

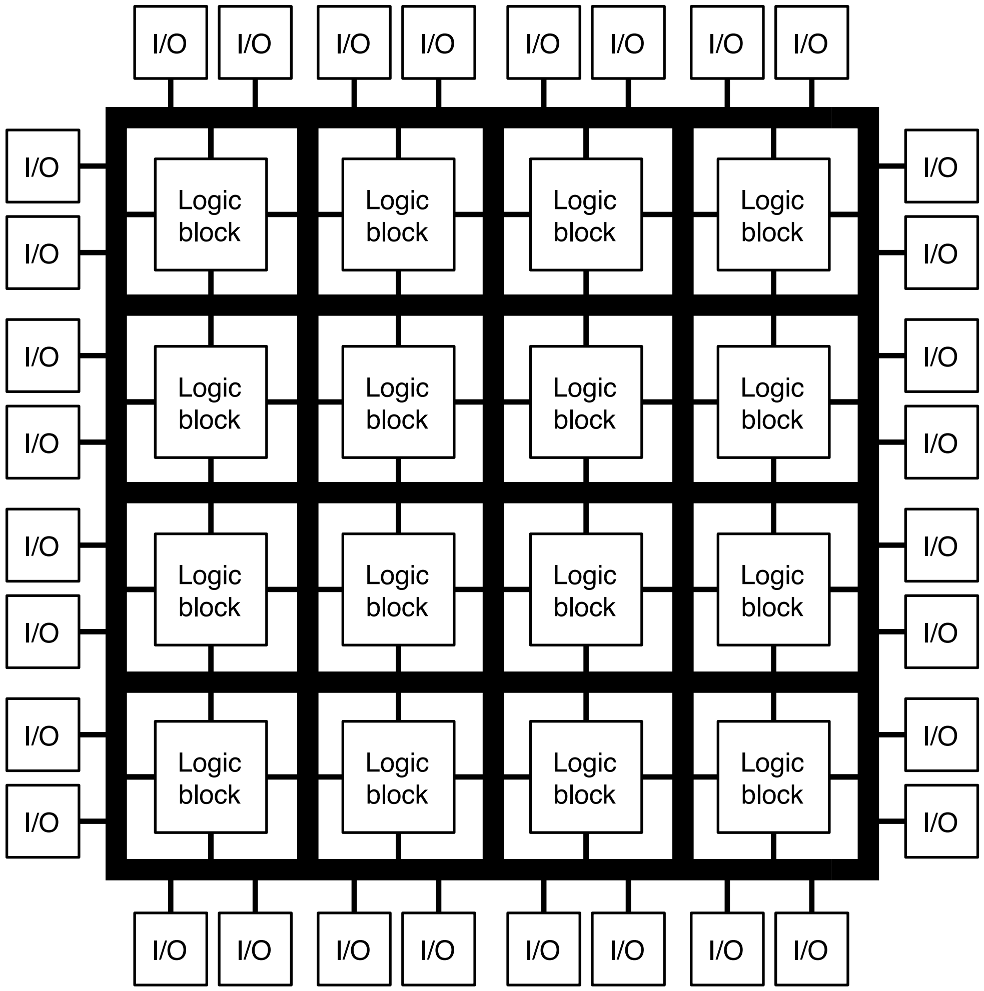


Figure 2.1: An island-style FPGA

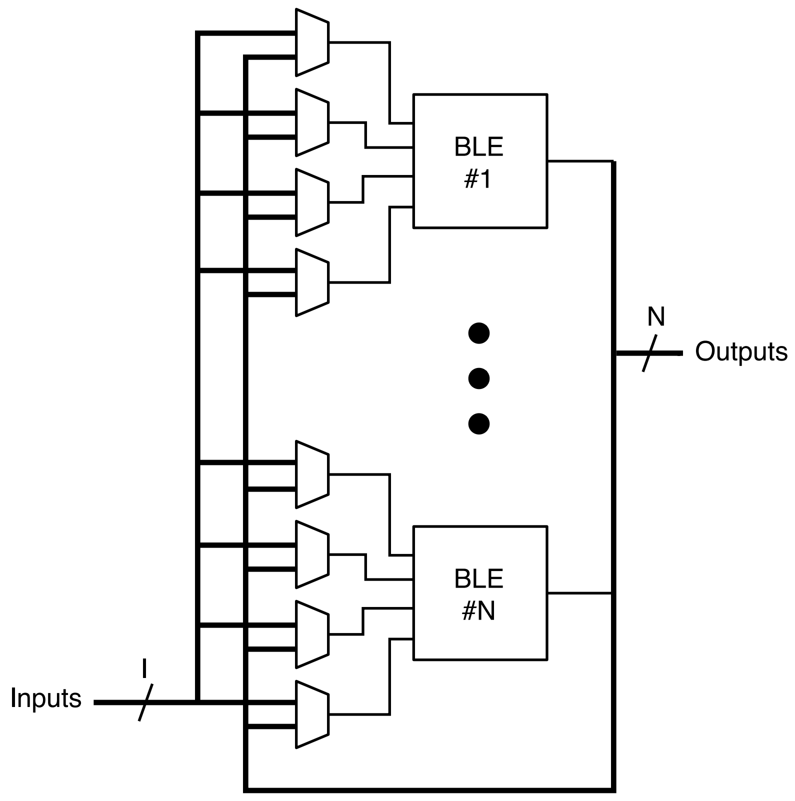


Figure 2.2: A logic cluster

connect to some of the adjacent routing tracks. Logic blocks are often implemented with clusters of one or more basic logic elements (BLEs) as shown in Figure 2.2. This structure is called a logic cluster. The inputs to the BLE are selected using multiplexers from the set of inputs to the logic block and the set of BLE outputs. These multiplexers are called input crossbars and in this work they are fully populated meaning all logic block inputs and BLE outputs are connected to each multiplexer. Configuration SRAM bits select which signals are connected to each BLE.

A BLE is often composed of a look-up table (LUT) and a flip-flop. A K-input LUT implements an arbitrary K-input combinational logic function and the flip-flop allows sequential circuits to be created. The BLE circuit used in this work is shown in Figure 2.3 and employs a four-input LUT and a flip-flop. The output of the BLE is either registered or unregistered based on a configurable multiplexer. In this figure, the LUT has four

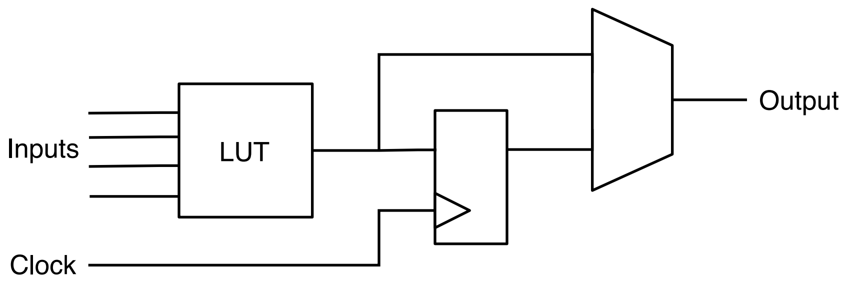


Figure 2.3: Basic logic element (BLE)

inputs so it can realize any logic function of four inputs. It is typically implemented with a 16-input multiplexer. The four inputs of the LUT connect to the select lines of the multiplexer. They select one of sixteen SRAM bits that correspond to the entries of the truth table.

The configurable connections from the routing tracks to the inputs of the logic block are implemented with multiplexers as shown in Figure 2.4. The programmable output connections from the logic block to the routing tracks are made with buffered switches and are also shown in Figure 2.4. These are called the input and output connection blocks, respectively. The number of routing tracks that connect to each logic block input or each output is know as the connection block flexibility and is represented as $F_{c,input}$ or $F_{c,output}$ [6]. In Figure 2.4, $F_{c,input}$ is four and $F_{c,output}$ is two. Similarly, the number of routing tracks that connect to each I/O block for both input and output is represented as $F_{c,pad}$.

The routing wires may stretch the length of one logic block or they may span multiple blocks. The number of logic blocks a routing track spans is know as its length. All routing wires may be of the same length or some fraction of them may be different lengths. Routing wires can be connected together at their ends and where they cross. A set of programmable switches allows these connections. They can be buffered or unbuffered switches as shown in Figure 2.5. The buffered switches can also be unidirectional or bidirectional. The set of switches is called a switch block and the number of possible

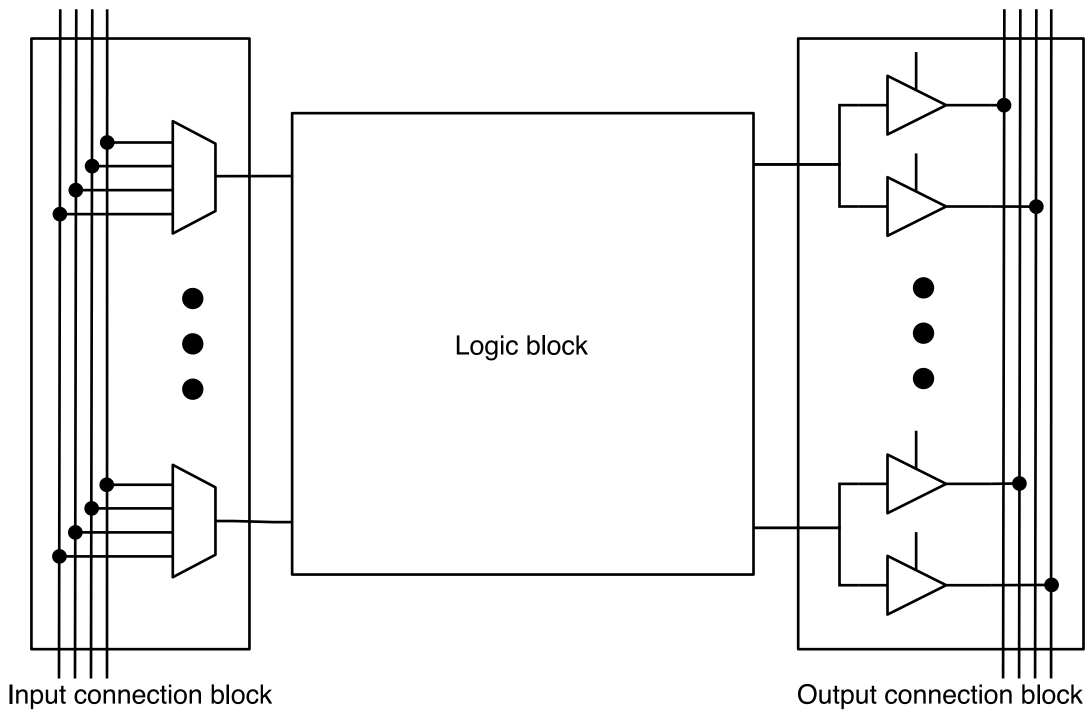


Figure 2.4: Input and output connection blocks

connections each wire can make is called the switch block flexibility, F_s [6]. In this figure, F_s is three for the first routing track in both the horizontal and vertical channels.

It is desirable to use the regular structure of FPGAs to simplify their design and physical layout. An FPGA tile refers to a logic block and one set of adjacent horizontal and vertical routing tracks. This tile is repeated in an array to form the FPGA as shown in Figure 2.6. Since most FPGA architectures require routing tracks that span more than one tile yet not the entire length of the chip, Figure 2.6 also shows how "twisting" the routing tracks achieves other track lengths [7]. In this figure, all routing tracks are length four.

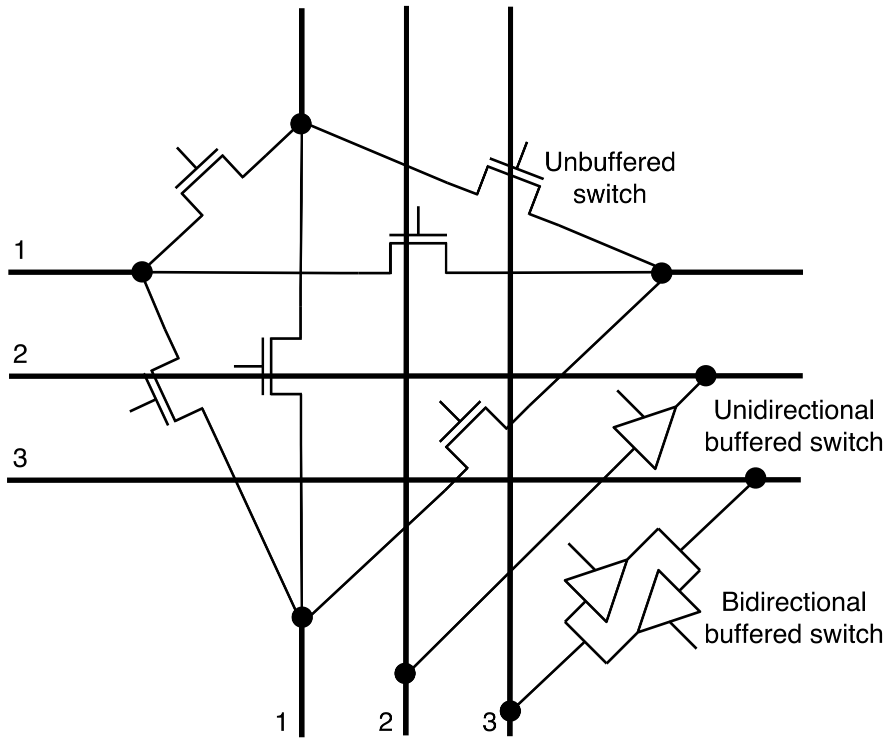


Figure 2.5: Switch block

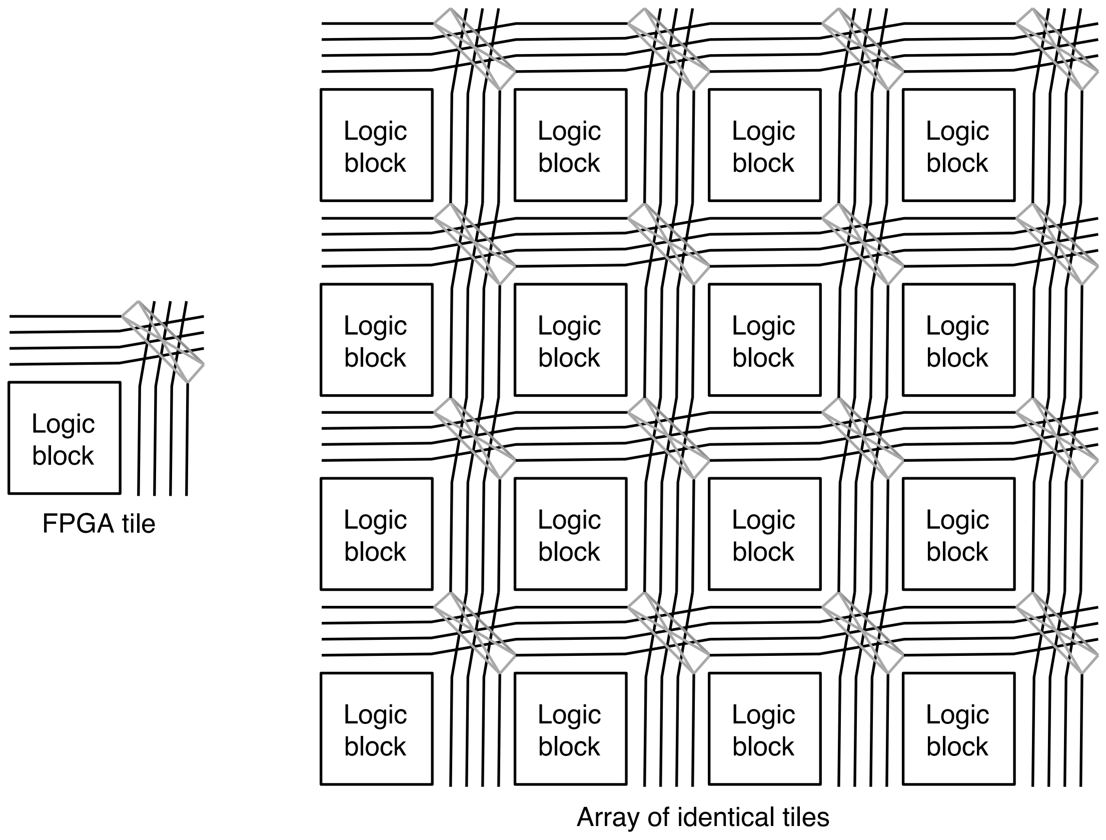


Figure 2.6: FPGA tile and array

2.2 VLSI

Implementing massive circuits on a single chip is called very large-scale integration (VLSI). These chips are fabricated on silicon wafers using chemical and mechanical processes. Many stages of oxidation, diffusion, deposition, ion implantation, metallization, and polishing are performed to create the chip [8]. Photolithography masks are used to selectively perform each task on different areas of the wafer. These masks are derived from layouts containing two dimensional diagrams of each layer. A complementary metal-oxide-semiconductor (CMOS) process is designed to implement both p-channel metal-oxide-semiconductor (PMOS) and n-channel metal-oxide-semiconductor (NMOS) transistors. Ion implantation dopes the silicon with n+ or p+ impurities. Transistors are created where active n+ and p+ diffusion regions intersect with the polysilicon layer that forms the gate of the transistor [1]. For the process used in this work, PMOS transistors must be contained in a n-well region. Contacts connect active n+ and p+ diffusion regions as well as polysilicon to the first metal layer. Several metal layers exist in modern processes and they are connected with vias. An example layout of an inverter is shown in Figure 2.7. To connect wires off chip, large bonding pads are used. After fabrication, the wafers are cut into chips and wires are bonded between the pads and the pins of the chip package.

To ensure that the layout is manufacturable, each fabrication process has a list of design rules that must be met [1]. As part of the design flow, these rules must be verified with design rule check (DRC) software. Figure 2.8 shows rules involving minimum width, fixed width, and minimum spacing of some drawing layers as well as extensions of one layer over another. The exact specifications for each rule are different for each process. Other rules involve minimum area of an object and minimum density across the entire chip. For finer geometry processes, large metal wires can act as unwanted antennas and build up charge during fabrication, potentially causing damage to transistor gates [9]. Antenna rules dictate when a wire is likely to cause this problem. Connecting a diode as

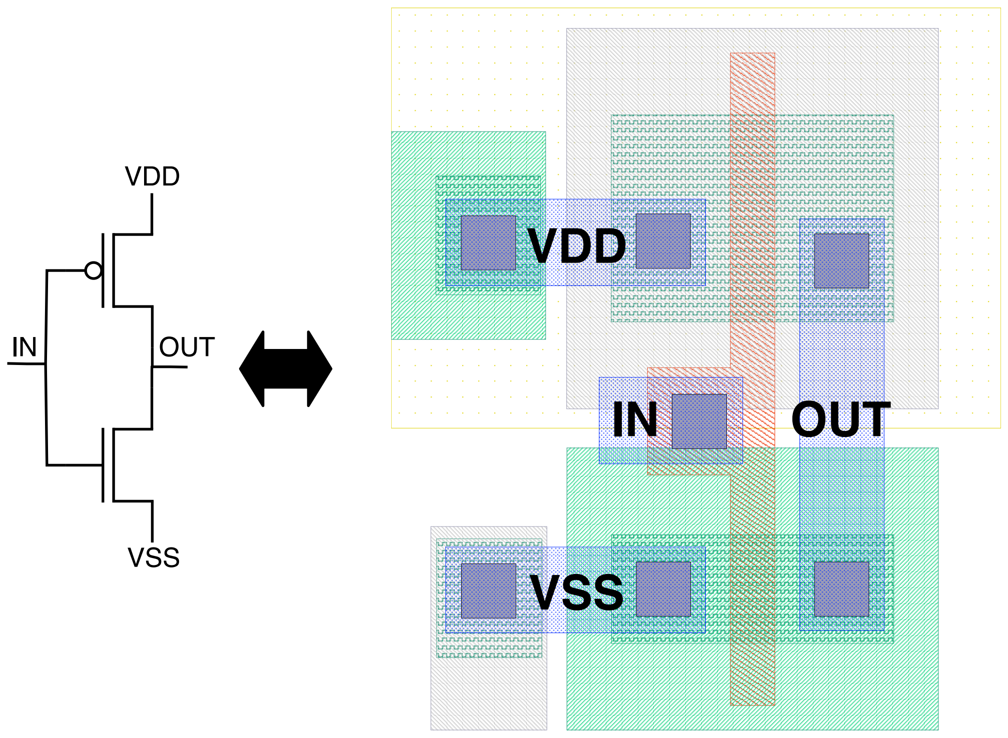


Figure 2.7: Schematic and layout of an inverter

shown in Figure 2.9 prevents a large voltage between the gate and the substrate. During normal operation, the substrate is grounded so the diode is reverse biased and the circuit functionality is unaffected.

Making sure the design is free of DRC and antenna errors improves fabrication yield; however, it does not guarantee the design will work to specifications or at all. To verify the design, a schematic is extracted from the layout [1]. The extracted version represents the circuit that is implemented by the layout. It is compared to the original design schematic using a process called layout versus schematic (LVS). If the netlists match it is likely that the layout will behave the same as the schematic simulation. To provide more confidence, the extracted view can be simulated with the extracted parasitic capacitors and the correct diffusion area resistances.

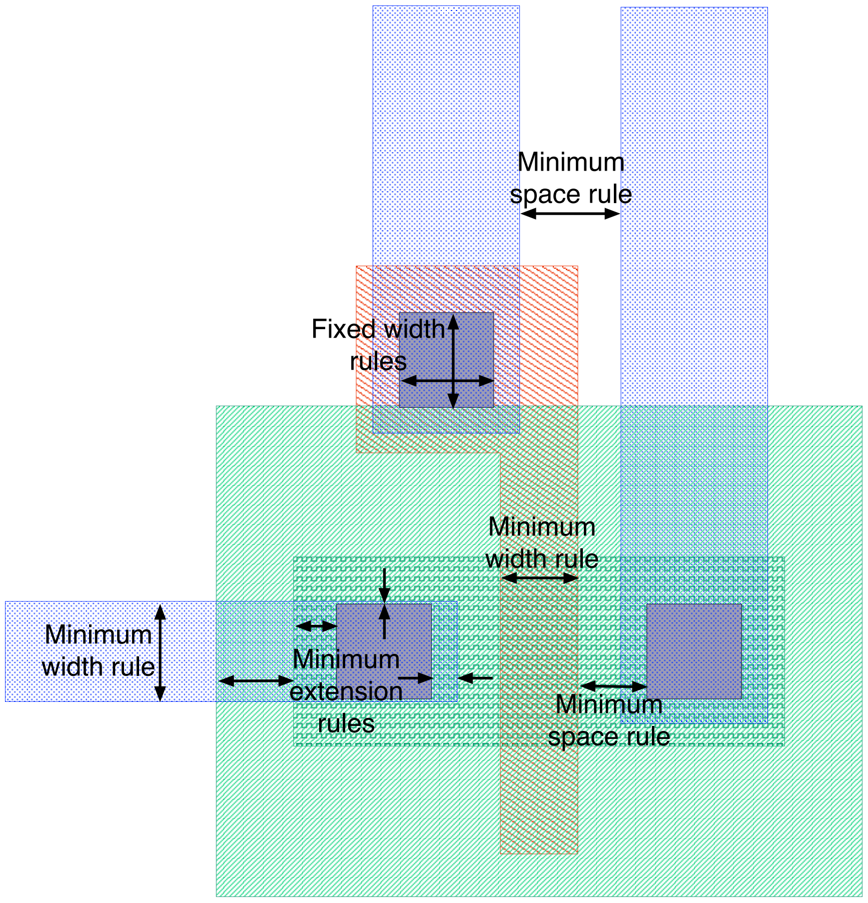


Figure 2.8: Examples of design rules

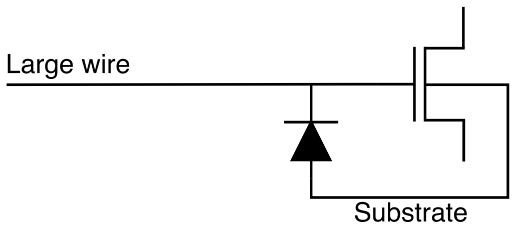


Figure 2.9: Diode prevents large voltage that could damage transistor gate

2.2.1 ASICs

VLSI design methodologies include ASIC and custom flows. Custom IC design involves drawing every geometric shape on every layer needed to create photolithography masks. ASIC tools automate the design process to reduce development time at the expense of decreased speed and increased area and power [3]. A simplified ASIC design flow is shown in Figure 2.10. Design specification is done using a schematic or hardware description language (HDL). This is synthesized to a netlist of components from an ASIC library. These components, called standard cells, contain transistor layouts designed for various functions such as logic gates and flip-flops. The cells, which are required to be of equal height but have variable width, are placed on a grid and the connections between them routed using automated tools.

2.3 Related Prior Work

This section divides related prior work into two categories. Section 2.3.1 examines previous techniques to estimate the layout area of cells. Section 2.3.2 reviews relevant portions of the vast field of automatic layout.

2.3.1 Area Estimation

Layout designers often estimate the layout area of an IC before completing the entire design process. This gives designers early information about the IC's size that can help with floorplanning and with evaluating the silicon area and packaging requirements. Several approaches exist for estimating the area of standard cell designs assuming the area of each standard cell is known [10, 11].

In this work, we compare the tile area of many FPGA architectures. An automatic layout tool determines the exact tile area based on the size of the custom cells in the tile netlist. However, the set of custom cells required for an FPGA tile changes with the

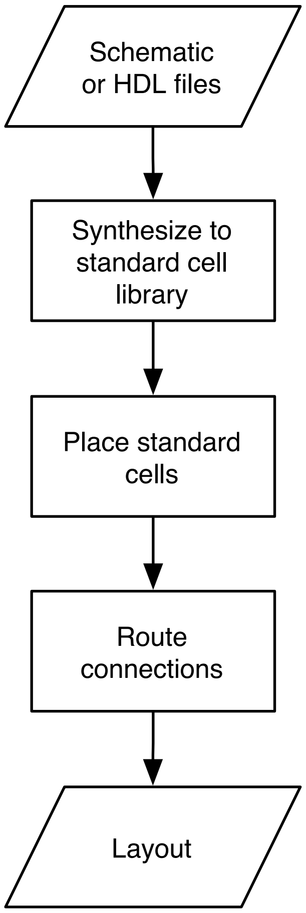


Figure 2.10: ASIC design flow

FPGA architecture. It would be very time consuming to layout every cell used in every experimental architecture to obtain the area of all possible cells. Instead, the area of each cell is estimated based on its transistor implementation.

Few researchers have looked at area estimation of ICs based on the transistor implementation. Wu et al. [12] propose formulas for estimating the size of datapath and control logic based on the transistor implementation of standard cells. They sum the number of transistors in the design and multiply by the pitch between transistors and the height of the standard cells.

Betz et al. [4] estimate the layout area of FPGAs. They measure area in terms of the number of minimum-width transistors that exist in the layout. However, many transistors are larger than minimum width to increase their drive strength. Betz et al. devised an equation to determine the area of each transistor, i, in terms of the number of minimum-width transistors that occupy the same area:

$$area_i(minWidthTransistors) = 0.5 + \frac{driveStrength_i}{2 \times minWidthDriveStrength} \tag{2.1}$$

This equation recognizes that a transistor with 2x drive strength takes up less space than two 1x transistors. Betz et al. sum the result of Equation 2.1 for all the transistors in the FPGA. Then to obtain the area in μm^2, they multiply by the layout area required for a minimum-width transistor. They also assume custom designers achieve 60% of the maximum transistor density so they divide the final area by 0.6.

2.3.2 Automatic Layout

There is an extensive body of work on automatic layout [13, 14] as well as widely used commercial tools. These projects and tools can be classified depending on whether the layout is performed at the transistor-level or the cell-level. In addition, floorplanning, which determines the high-level layout structure, can be performed using automatic tools. After reviewing different techniques, this section examines automatic layout projects that target FPGAs.

Transistor-Level Layout

A research project by Serdar and Sechen [15] explored automatic layout at the transistor-level. Their tool, AKORD, performs automatic layout for small groups of transistors. Their results indicate automatic layout is comparable to manual layout for up to fifty transistors; however, no results are presented for larger designs. Synopsys has a commercial tool called Cadabra for the automatic creation of standard cells from a transistor-level netlist [16]. Again it is designed for small circuit sub-blocks (typically standard cells) and not a whole chip. While these tools could be used to create cells for cell-level automatic layout, they do not scale well enough to create automatic layouts of entire FPGAs or even FPGA tiles on their own.

Cell-Level Layout

An approach that scales to a larger number of transistors is to break the problem into two levels of hierarchy. At the bottom-level, the transistors are grouped into cells with two to twenty transistors and their layout is performed manually or with transistor-level automatic layout tools. The top-level netlist is composed only of these cells. The position of the cells in the top-level layout is determined automatically using a placement tool and connected automatically using a router.

Cell-level layout can be classified into standard cell layout and general cell (or building block) layout [13]. Standard cells have a fixed height and variable width. They have power and ground wires running along the top and bottom of the cells. Abutting standard cells horizontally connects the power and ground nets. General cells have variable width *and* height. The power and ground connections must be connected by a router. Standard cells are part of the ASIC design flow described in Section 2.2.1. Libraries of these cells are available so the layouts are created once and used repeatedly. The present work uses general cells like the work by Ogawa et al. [17] and Onodera et al. [18]. General cells are not used in ASIC designs because of the lack of cell libraries and the increased difficulty in routing power and ground.

Chinnery and Keutzer [3] compare automated ASIC designs that use standard cells to custom designs where the layout of the entire IC is performed manually. While ASIC tools dramatically reduce layout time, they found that ASIC designs are three to eight times slower than custom designs. This performance gap is partially attributed to restrictive standard cell ASIC libraries that often limit the choice of cells and result in non-optimal drive strength. The standard cell libraries also provide poor latches and lack dynamic logic gates used in high-performance custom ICs. Chinnery and Keutzer give suggestions for improving the quality of ASIC designs. They suggest adding application-specific standard cells to the library as well as better logic design, pipelining, and floorplanning. In one chapter of their book, which was contributed by Chang and Dally, an ASIC implementation of a 64-bit microprocessor register fetch stage was found to be 14.5x larger and 3.72x slower than a custom implementation. By adding some manual layout effort to exploit the regularity of each bit-slice and to add application-specific standard cells to the library, the partially automated design was only 1.64x larger and 1.11x slower than the custom implementation.

Floorplanning

Floorplanning determines the placement of large blocks such as intellectual property (IP) cores, embedded memories, and groups of standard cells. The area of each block is fixed but there may be several alternative layouts with different shapes [14]. It is similar to general cell layout but the shape of each block has to be determined in addition to its placement. Floorplanning is performed manually or with automatic tools. Some of these tools attempt to minimize the placement area and eliminate the whitespace between blocks [19] while others optimize the placement within a fixed area [20]. When the area is fixed, the goal is to distribute the whitespace between the cells to reduce routing congestion. Some floorplanning tools also allow the blocks to be mixed with standard cells and the tool will perform standard cell placement at the same time as floorplanning [21].

Automatic Layout of FPGAs

Most previous approaches to the automatic layout of FPGAs have employed the standard cell design flow. Phillips and Hauck applied a standard cell flow to implement a reconfigurable datapath architecture called RaPiD [22]. This architecture is similar to a one-dimensional FPGA with datapath operations for logic blocks. They obtain a standard cell version of RaPiD that is 42% larger and 64% slower than the custom implementation. They improve this result by targeting a specific set of circuits and removing reconfiguration flexibility that is not necessary for those specific circuits. This technique achieves designs that are up to 46% smaller and 36% faster than the original custom implementation. While the results are promising, these chips are not nearly as flexible as a general purpose FPGA. Phillips and Hauck also achieve benefits from adding FPGA-specific standard cells to the ASIC library. They reduced the area of their various standard cell versions by 9% to 18.9% and improved the speed by 7% to 36%.

A similar approach by Kafafi, Bozman, and Wilton uses a standard cell flow to create small customizable FPGA cores for use as part of an IC [23]. They simplify the programmable architecture by removing flip-flips and combinational loops and rely on external logic to perform these functions. The results show that the automated approach creates designs 6.4 times larger than the custom implementations of their FPGA cores. Clearly, it is challenging to create FPGAs using existing ASICs tools that are similar in size to custom FPGAs.

In the present work, we employ an automatic layout tool called GILES [5]. The next section discusses how GILES works compared to previous approaches.

2.4 The GILES Automatic FPGA Design and Layout System

There are several features that make GILES [5] unique compared to other previous work on automatic layout of FPGAs. It is similar to the general cell layout approach so it handles large numbers of transistors; however, it uses a custom placer with optimizations specific to FPGAs so it outperforms standard cell design flows on FPGA-specific layouts. A key feature is that it is the only automatic layout tool that uses a high-level architectural description as input to specify the FPGA structure. The architecture description is easily understandable and configures most of the architectural parameters described in Section 2.1. Its format is the same as the architecture file used in the architecture exploration tool called Versatile Place and Route (VPR) [4].

An excerpt of an architecture description file is shown in Figure 2.11. It specifies one BLE per cluster with a four-input LUT. The logic block has four inputs, one output, and a clock for the flip-flop. The connection block flexibilities are specified as a fraction of the routing tracks instead of an absolute number so they adjust when a different number of routing tracks is used. The file also specifies that half the routing tracks are buffered. All the routing tracks have a length of four.

The previous version of GILES focused on the automatic layout of a single FPGA tile. The tile is repeatable as illustrated in Figure 2.6. The GILES tool flow is illustrated in Figure 2.12. It begins with an architecture description and finishes with the layout of the tile. The first stage is the netlist generator. It translates the architecture description into two netlists: one composed of cells and one of transistors. The next stage places the cell-level netlist. Finally, the connections are routed to obtain the layout.

```
# Logic block parameters
subblocks_per_clb 1         # 1 BLE per logic cluster
subblock_lut_size 4         # 4-input LUTs

# Logic block inputs and outputs
inpin class: 0 bottom
inpin class: 0 left
inpin class: 0 top
inpin class: 0 right
outpin class: 1 bottom      # Logic block output
inpin class: 2 global left  # Clock input

# Connection block flexibilities
Fc_type fractional          # Specified as fractional number of tracks
Fc_input 0.5625             # Flexibility of input connection block
Fc_output 1                 # Flexibility of output connection block
Fc_pad 1                    # Flexibility of I/O pads

# Switch types
switch 0 buffered: no ...
switch 1 buffered: yes ...

# Length 4 routing tracks, half buffered
segment frequency: 0.5 length: 4 wire_switch: 0 ...
segment frequency: 0.5 length: 4 wire_switch: 1 ...
```

Figure 2.11: Excerpt from a VPR architecture description file used as input to GILES

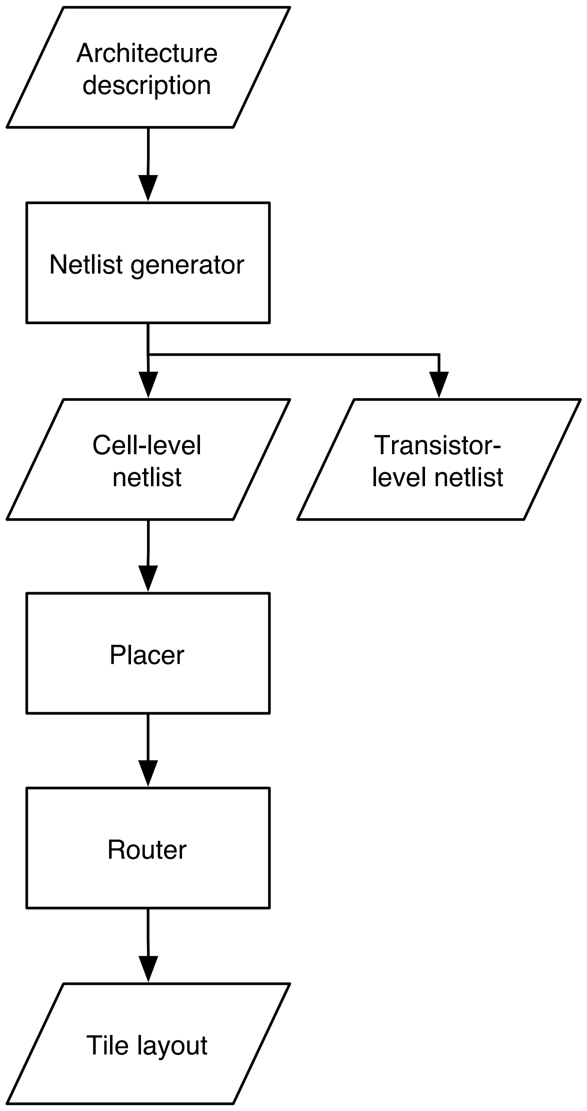


Figure 2.12: GILES flow

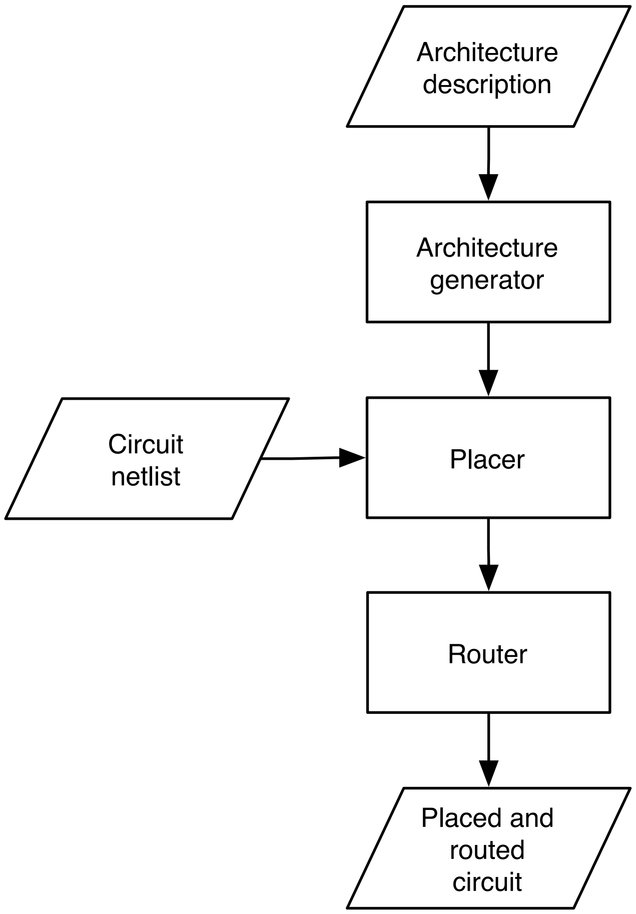


Figure 2.13: VPR flow

2.4.1 Netlist Generator

The netlist generator is based on the architecture exploration tool called VPR [4]. VPR places and routes circuits on an FPGA. The goal of VPR is to easily implement circuits on different architectures. As shown in Figure 2.13 the inputs are a circuit netlist and an architecture description of the FPGA on which to implement that circuit. The output is the placement and routing of an application circuit on the specified FPGA. This should not be confused with the goal of this work, which is the placement and routing of the FPGA layout required to build the FPGA itself.

VPR was not designed to create FPGAs. However, it has an internal representation of the FPGA and its programmable routing in the form of a routing-resource graph. The

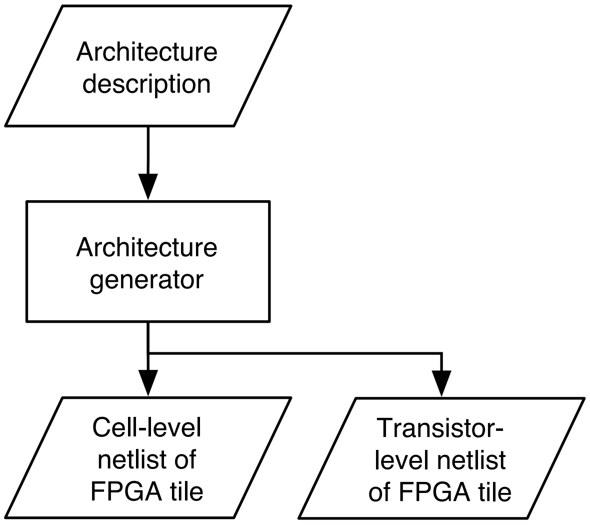

Architecture description
Architecture generator
Cell-level netlist of FPGA tile
Transistor-level netlist of FPGA tile

Figure 2.14: Netlist generator based on VPR

netlist generator of GILES is a modified version of VPR that outputs a transistor-level netlist and a cell-level netlist of the FPGA tile based on the logic block structure and the routing-resource graph as shown in Figure 2.14 [24]. Instead of using standard cells taken from an ASIC library for the cell-level netlist, the netlist generator assumes the creation of custom, general cells optimized for the specific FPGAs architecture. The transistor-level netlist contains the transistor implementation of each cell. Additionally, the netlists describe the connections out of the tile, called ports. Because the tile will be abutted to copies of itself to create the array, ports on each side of the tile will connect to ports on the opposite side. Therefore, each port must match up with another port called its “tiling partner”. The tiling partners are labelled so the placer knows to keep them aligned.

2.4.2 Placer

The placer reads in the cell-level netlist and outputs the placement of the tile. The placer’s goal is to minimize area and wirelength [5]. It begins with a random placement and moves the cells to minimize its cost function. Whether an attempted move is accepted is based on a simulated annealing algorithm [25, 26].

A unique feature of the placer is that it compacts the tile between placement phases as illustrated in Figure 2.15 [5]. The entire process begins with a course grid that is the size of the largest cell. Cells are forced to spread out so they have plenty of space to move around the grid. The placer performs an optimization phase to obtain good relative positions for the cells. Then the first compaction phase removes as much empty space as possible by shrinking the tile. After compaction, a new optimization phase reduces the wirelength and strives to move the blocks off the tile edges. Afterwards another compaction phase tries to reduce the area further. These phases alternate until the tile has not shrunk in several iterations. A final optimization phase is performed without biasing cell moves that help to compact the tile. An example of initial and final placements is shown in Figure 2.16.

Another unique feature of the placer is its ability to make FPGA specific optimizations. The inputs and outputs of the cells are called pins. Many of the connections to these pins can be swapped because the pins are logically equivalent [27]. In other words, the circuit would still perform the same function whether the pin connections are swapped or not. The benefit of swapping the pins is that it reduces wirelength. Examples of swappable pins are the input and select signals of multiplexers. Because SRAM bits connect to the select lines, the input lines can be swapped and the SRAM bits will be programmed to reflect the change. The case is similar for swapping the select lines as well as the input and select signals of LUTs. The multiplexers in this work require both regular and complemented select lines so the SRAM cells output both signals. For both cells, the two signals can be swapped by flipping the value programmed into the SRAM bit. All these pin swaps are illustrated in Figure 2.17. ASIC flows also allow connections to standard cells be swapped but our placer has more options because it allows changes in how the SRAM bits are programmed.

The logical equivalence of SRAM bits is also leveraged to minimize the wirelength

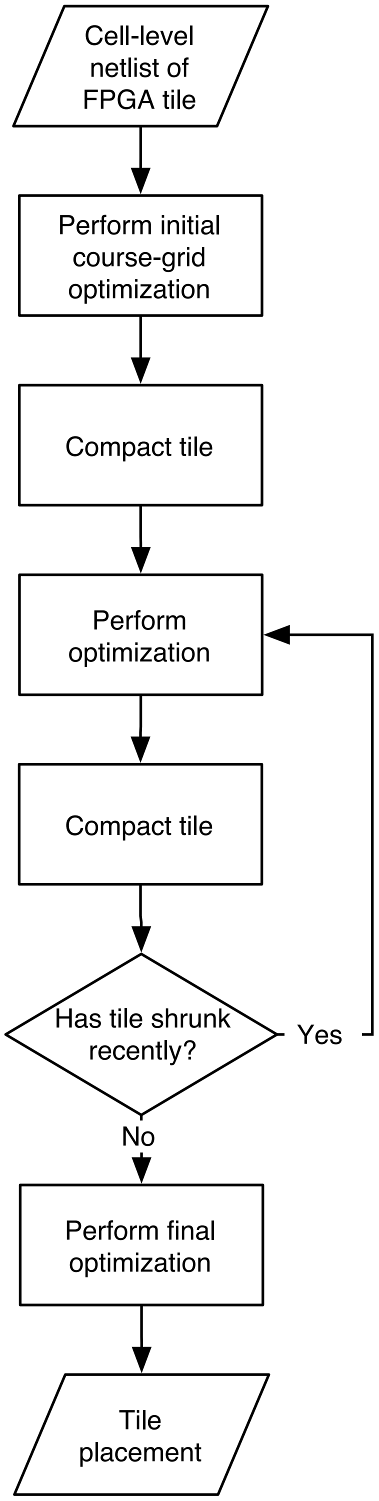

Cell-level netlist of FPGA tile
Perform initial course-grid optimization
Compact tile
Perform optimization
Compact tile
Has tile shrunk recently?
Yes
No
Perform final optimization
Tile placement

Figure 2.15: Placer algorithm

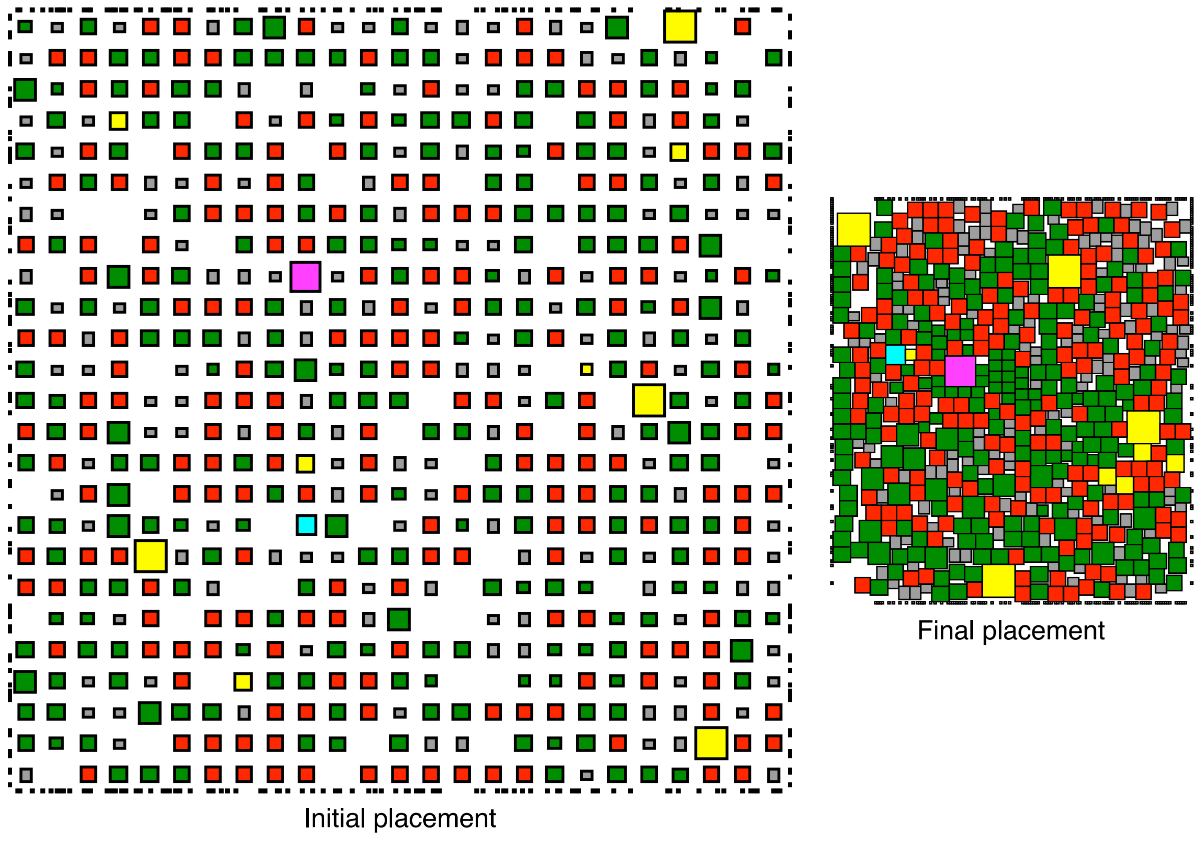


Figure 2.16: Initial and final tile placements

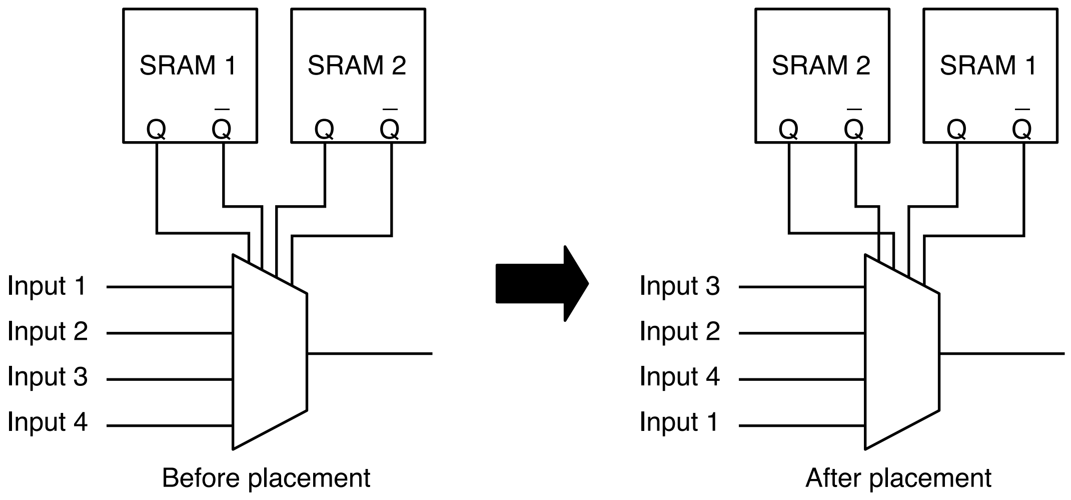


Figure 2.17: Placer swaps logically equivalent pins

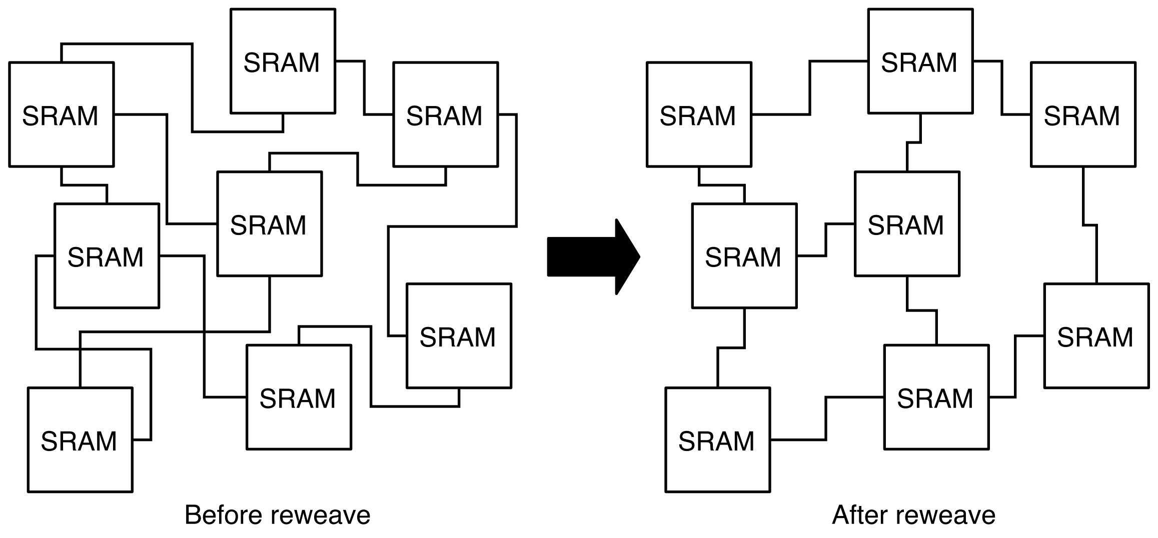


Figure 2.18: Reweaving SRAM word and bit lines

of their programming lines. A word line selects a row of SRAM bits for writing and a series of bit lines drive individual values into each SRAM bit. These lines may get tangled when the placer moves cells. To untangle the wires, the placer rips up and rewires the programming lines after each anneal [27]. This process is called reweaving and is illustrated in Figure 2.18.

2.4.3 Router

The router is the last stage of GILES. It routes the connections between the cells and to the tile ports. The input is the tile placement and the number of metal layers used for routing. The output is a description of the routing. It is an implementation of the maze router algorithm [28]. A unique feature of the router is that when it fails to route a design, it adds space in congested regions and starts again [29].

The router uses a grid on which it draws wires. The grid is sized so that two minimum size vias can be placed in adjacent grid squares and meet minimum spacing design rules. When the router completes, all wires in the design are connected, including power, ground, and clock networks. However, there are no special structures such as a clock tree or power grid. Therefore, they must be implemented separately on metal layers not used for routing. The router is not capable of avoiding existing metal traces so the layers used for routing cannot be used by the power grid, clock tree, or the cells.

2.4.4 Previous Results

Padalia et al. [5] compared GILES to two commercial FPGAs: the Xilinx Virtex-E and the Altera Apex 20K400E. The comparisons were very approximate because GILES is limited to VPR architectures and cannot reproduce the features of these commercial chips exactly. Using a total of eight metal layers, the FPGA tiles created by GILES were 47% and 97% larger than the tiles of the Virtex-E and Apex 20K400E, respectively. This is quite impressive considering the ease in which the tiles are generated. Instead of performing the layout of the entire tile, only the set of cells used by GILES is required.

Chapter 3

Improvements to the GILES Automated FPGA Layout System

The goal of the GILES automated layout project is to significantly reduce the development time of FPGAs while maintaining competitive area, speed, and power compared to custom designs. Competitive area results are critically important since any increase in silicon area increases production costs and detracts from cost benefits resulting from reduced development time. Smaller, compact designs also have side benefits affecting speed and power by reducing the length of connecting wires. The previous results mentioned in Section 2.4.4 compare the area of FPGA tiles made with GILES to commercial FPGAs. The results show GILES creates FPGA tiles that are 47% to 97% larger than commercial FPGAs [5]. The primary goal of this chapter is to improve these results.

To accomplish this goal, this chapter explores the set of cells that make up the netlist of the FPGA tile. These cells are the basic building blocks used by the GILES placer and router. The size of these cells have an impact on the density of the layout and the effectiveness of the placer. The original version of GILES used arbitrary groupings of transistors to form these cells. This chapter revisits these groupings to determine how they affect the area results and then chooses the best groupings to minimize the area.

To obtain accurate area results for comparison, the layout of each unique cell in the tile is required. New cells are required for different groupings and different FPGA architectures. However, the only information needed from the layout is the size and pin positions of each cell. So, to avoid manual layout, the GILES netlist generator estimates the size of each cell based on its transistor implementation and assigns pin positions such that they are spread out over the cell. The accuracy of the area results are dependent on the accurate modelling of the cell sizes. Therefore, before improving the results of GILES, Section 3.1 devises a new area model for estimating cell area with better accuracy. Then Section 3.2 examines the area results of different groupings of transistors into cells. Section 3.3 compares the area of tiles generated with GILES to the layout area of a commercial FPGA.

3.1 A New Area Model for Cells

GILES is capable of automatically generating the layout of any FPGA that can be described using the VPR architecture description. This implies that the netlist of the FPGA tile will be different depending on the architecture. Some cells are needed for some architectures but not for others. Examples of cells are inverters, buffers, SRAMs, multiplexers, flip-flops, and pass transistors. All but the SRAMs and flip-flops have transistors that are sized based on the architecture description file. Instead of manually creating the layout of each cell for numerous architectures, GILES contains an area model for estimating the layout area of a cell based on the number and size of transistors. This section measures the accuracy of the area model used in the previous version of GILES. Then a new area model is derived that achieves better results.

3.1.1 Previous Area Model and Measurement of Accuracy

We measure cell area in grid squares corresponding to the granularity of the placement and routing grid. The grid size is set to 0.66 µm by 0.66 µm, which is the minimum distance allowed between metal vias in 0.18 µm technology. The previous GILES area model [27] uses the following equation to estimate the area in grid squares of each cell:

$$cellArea(gridSquares) = max(2.25 \times \sum_{i \in xtors} driveStrength_i, 3 \times numPins) \qquad (3.1)$$

This equation sums the drive strengths of the transistors in a cell to obtain a rough estimate of the number of minimum-width transistors that occupy the same area. It assumes that a 1x drive strength transistor is equal to one minimum-width transistor. Therefore, a transistor with 2x drive strength is equivalent in area to two minimum size transistors. The number of minimum-width transistors is multiplied by 2.25 for the number of grid squares required to layout a minimum-width transistor in 0.18 µm technology. For some cells, this area is not enough to fit all the pins and routing needed for external connections. In these cases, the area is set to be three times the number of pins in the cell.

The GILES placer obtains the best results when the cells are as close to square as possible [27]. Therefore, the width of each cell is determined by taking the square root of its area. The width is rounded up to the nearest integer since cells cannot occupy a fraction of a grid square. The cell height is determined by dividing the cell area by the rounded width. The cell height is rounded up as well. The final cell area is equal to the product of the rounded width and height.

To determine the accuracy of the previous model, we examined the layouts of sixteen cells of various types with sizes ranging from 20 to 418 grid squares. These cells were laid out using a 0.18 µm technology and Micro Magic MAX. The layout area is compared to the estimated area obtained from the model in Table 3.1. The area model of Equation 3.1 underestimates the area of all the cells except the 15x buffer cell. Overall, the previous model has an average absolute error of 43.7%.

To see how this error impacts the use of GILES and its comparisons, we examined the same two tiles as Padalia et al. [5]. These tiles have similar features to the tiles in the Xilinx Virtex-E and the Altera Apex 20K400E. The differences between our tiles and the commercial FPGA tiles are a result of the limitations of the VPR architecture description language. We ran GILES with the actual cell layouts and the modelled cell layouts for both tiles. As shown in Table 3.2, when using the modelled cell layouts instead of the actual cell layouts, GILES underestimates the tile areas by 33.7% and 7.5%. The error for the tile areas is smaller than for individual cell areas because the router adds space between the cells to ease routing congestion.

3.1.2 New Area Model

To improve these results, we derive a new area model based on the area model by Betz et al. [4]. The area of each transistor, i, is measured in terms of the number of minimum-width transistors that occupy the same area using the following equation:

$$area_i(minWidthTransistors) = 0.5 + \frac{driveStrength_i}{2} \tag{3.2}$$

This equation is the same as Equation 2.1 except the $minWidthDriveStrength$ term in the denominator is missing. GILES specifies all drive strengths relative to the drive strength of a minimum-width transistor so this term is not needed.

Cell areas are estimated by summing the result of Equation 3.2 for each transistor in the cell:

$$cellArea(minWidthTransistors) = \sum_{i \in xtors} area_i(minWidthTransistors) \tag{3.3}$$

In this equation, the cell area is still represented in terms of the number of minimum width transistors that occupy the same area. To convert the area to the number of grid squares used by the cell, we multiply by the number of grid squares required to lay out

Table 3.1: Accuracy of old area model

Cell	Layout area (grid squares)	Estimated area (grid squares)	Error
1x inverter	25	12	-52.0%
2x inverter	30	16	-46.7%
4x inverter	36	30	-16.7%
4x buffer	56	36	-35.7%
5x buffer	90	64	-28.9%
15x buffer	121	144	+19.0%
SRAM	49	20	-59.2%
2-input multiplexer	25	20	-20.0%
12-input multiplexer	156	72	-53.8%
24-input multiplexer	306	110	-64.1%
32-input multiplexer	342	144	-57.9%
36-input multiplexer	418	169	-59.6%
LUT	196	81	-58.7%
Flip-flop	90	42	-53.3%
3x pass transistor	20	12	-40.0%
8x pass transistor	30	20	-33.3%
Average absolute error			43.7%

Table 3.2: Accuracy of tile area with old area model

Approximate architecture	Tile area (grid squares)		Error
	Actual cell layouts	Modelled cell layouts	
Xilinx Virtex-E	52,268	34,640	-33.7%
Altera Apex 20K400E	124,161	114,873	-7.5%

a minimum-width transistor:

$$cellArea(gridSquares) = 3.3 \times cellArea(minWidthTransistors) \tag{3.4}$$

The number of grid squares required to lay out a minimum-width transistor has been changed from 2.25 in the previous area model to 3.3 to include the space needed *between* transistors.

Using Equation 3.4, we determined that the areas of SRAMs and multiplexers are underestimated, which can be attributed to their relatively complex layout. In comparison, inverters, buffers, flip-flops, and pass transistors are more simple structures and/or take greater advantage of diffusion region sharing. Accordingly, a complexity factor is added to Equation 3.4 to increase the cell area when it is more difficult to create a compact layout:

$$cellArea(gridSquares) = complexity \times 3.3 \times cellArea(minWidthTransistors) \tag{3.5}$$

The complexity factor is calibrated to 1.455 for SRAMs and multiplexers and to 1.0 for the remaining cells to closely estimate the area of the test cells. At the end of this section, the value of this factor is verified against a new set of cells. An important difference between the new and old area models is that the previous area model of Equation 3.1 depends on the number of pins in the cell. The new area model of Equation 3.5 does not have this dependency because the areas of our actual cell layouts are not pin-limited.

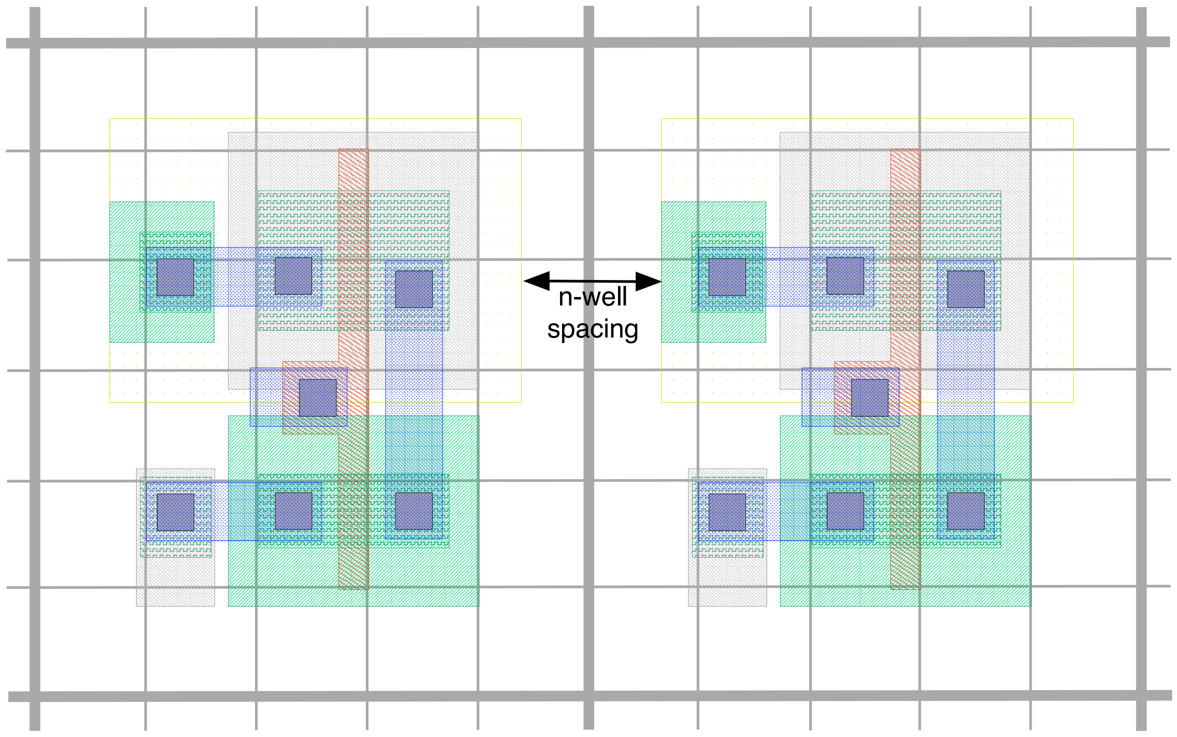


Figure 3.1: Required n-well spacing between inverter cells

The width and height of each cell are calculated as before to obtain cells that are approximately square. However, the width and height are rounded to the nearest integer instead of rounding up since two grid squares are now added to both the width and the height to create a one grid square border around each cell. This models the n-well, n+, and p+ regions that extend beyond the transistors as well as the design rules for the space required between these regions. Figure 3.1 shows the space needed between two inverter cells. The space is designed so that any side of one cell can be abutted against any side of another cell without violating design rules. The final area estimates include this border as do the actual areas obtained from the layouts. This is one reason why the previous area model underestimated the cell areas.

To measure the accuracy of the new area model, the actual layout area of each cell is compared to the area estimated by Equation 3.5. The percent error for each cell and the average absolute error for all cells is shown in Table 3.3. The average absolute error has been reduced to 5.8% from 43.7% with the previous area model. The worst area estimate of any cell is for the largest buffer with an error of +28.9%. Its area is overestimated due to extensive use of diffusion region sharing in the layout.

To see how the new area model impacts the area of the tile, we ran GILES with the new area model on the approximate Virtex-E and Apex 20K400E from Table 3.2. The results using the new area model are shown in Table 3.4. Now the tile areas are 3.2% and 8.0% larger than when using the actual cell layouts. These results are more accurate than when using the previous area model and they also show that tile areas are likely to be slightly pessimistic when using the new area model.

The cells in Table 3.3 were used to formulate the new area model so it is not surprising that it performs well for these cells. To verify the area model accuracy in general, we examine a new set of cells. Chapter 4 discusses using GILES to create an FPGA. To accomplish this, new cells were laid out in Cadence's Virtuoso Layout Editor [30] using a 0.18 µm technology library that has small differences compared to the 0.18 µm library used with Micro Magic MAX. In addition, new types of cells were required. The layout area of each cell is compared with the estimated area using the area model of Equation 3.5 in Table 3.5. The average absolute error of these fifteen new cells is 16.4%.

The area model was not tuned to these cells yet it still performs reasonably well. Many of the cell areas are overestimated but this is to be expected since when these cells were laid out, we had more experience and were able to create more compact layouts. The worst estimate is the 4x buffer and pass transistor grouping with an error of +52.4%. The layout of this cell makes extensive use of diffusion region sharing to compact the layout. It may be possible to use a complexity factor of less than one to estimate the area of cells with high levels of optimization; however, it is difficult to predict which cells will use these optimizations.

Table 3.3: Accuracy of new area model

Cell	Layout area (grid squares)	Estimated area (grid squares)	Error
1x inverter	25	25	0.0%
2x inverter	30	30	0.0%
4x inverter	36	42	+16.7%
4x buffer	56	56	0.0%
5x buffer	90	90	0.0%
15x buffer	121	156	+28.9%
SRAM	49	49	0.0%
2-input multiplexer	25	25	0.0%
12-input multiplexer	156	156	0.0%
24-input multiplexer	306	289	-5.6%
32-input multiplexer	342	380	+11.1%
36-input multiplexer	418	420	+0.5%
LUT	196	196	0.0%
Flip-flop	90	81	-10.0%
3x pass transistor	20	20	0.0%
8x pass transistor	30	36	+20.0%
Average absolute error			5.8%

Table 3.4: Accuracy of tile area with new area model

Approximate architecture	Tile area (grid squares)		Error
	Actual cell layouts	Modelled cell layouts	
Xilinx Virtex-E	52,268	53,957	+3.2%
Altera Apex 20K400E	124,161	134,050	+8.0%

Table 3.5: Verification of new area model

Cell	Layout area (grid squares)	Estimated area (grid squares)	Error
1x inverter (with level restorer)	35	30	-14.3%
2x inverter	30	30	0.0%
4x inverter	36	42	+16.7%
4x buffer	42	56	+33.3%
SRAM (4x4 grouping)	480	462	-3.75%
2-input multiplexer	20	25	+25.0%
11-input multiplexer	110	144	+30.9%
12-input multiplexer	120	156	+30.0%
20-input multiplexer	252	240	-4.8%
LUT	198	196	-1.0%
Flip-flop	108	90	-16.7%
Flip-flop with enable	126	110	-12.7%
4x buffer and pass transistor grouping	42	64	+52.4%
Level restorer	30	30	0.0%
AND gate	40	42	+5.0%
Average absolute error			16.4%

3.2 Grouping Transistors into Cells

Now that GILES has a more accurate area model, we focus on the primary goal of this chapter: to reduce the area of FPGA tiles created with our automatic layout system. One of the key steps in the GILES system is the choice of which groups of transistors should form the cells for the placement and compaction stage. In the original GILES [5], these are set somewhat arbitrarily to be inverters, buffers, SRAMs, multiplexers, LUTs, flip-flops, and pass transistors. There are several tradeoffs involved with these choices that could markedly affect the quality of results. These tradeoffs are discussed here.

In general, using smaller cells (in the extreme, each cell would be a single transistor) gives the placement and compaction stage more freedom to move individual cells, providing opportunity to produce placements with less wirelength and less "whitespace" (space that does not contain cells). However, the layout of each cell has an empty border around it, as shown in Figure 3.1, to ensure that design rules are not violated when cells are placed next to each other. If transistors are grouped into larger cells, the amount of border space needed is reduced.

For example, on the left of Figure 3.2, there are two cells that contain 12 grid squares of transistors each (shown in black) but occupy a total of 30 grid squares each due to the border (shown in grey). Placed side-by-side as neighbours, they occupy 60 grid squares, but their transistors only occupy 24 grid squares so 60% of the area is wasted. If the same transistors are grouped into a new cell, shown on the left, the transistors occupy the same 24 grid squares but no longer need the space between them. The layout designer can intelligently remove this space because he or she knows the implementation of the cells. The border is still needed around the new cell so the total cell area is 48 grid squares, a 20% savings from the total of 60 grid squares for the two separate cells. Smaller cells have a greater perimeter to area ratio so they have the most to gain from grouping.

To determine an estimate of the largest possible gain from eliminating borders, we used the new area model to estimate the area of the cells with and without the border.

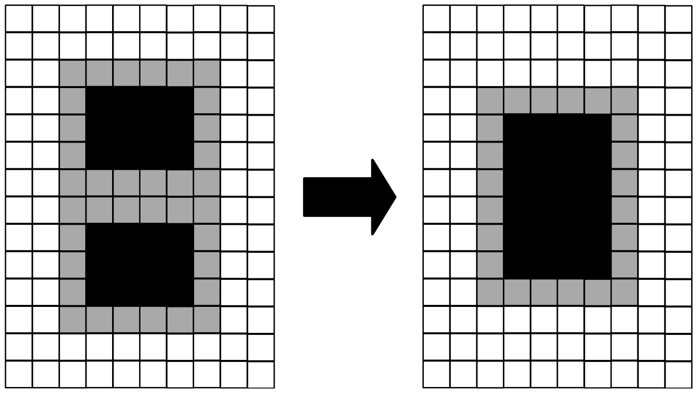


Figure 3.2: Grouping cells saves border area

We ran GILES using cells with the border and again without the border. Then we calculated the change in tile area between the two runs. We performed this procedure for ten FPGA architectures and obtained the geometric average of the area savings for each architecture.

The parameters of the ten FPGA architectures are summarized in Table 3.6. These are the same architectures used by Padalia et al. [5]. The main difference between these ten architectures is that each one has a different cluster size ranging from one to ten. The number of routing tracks and the connection block flexibilities, F_c, for input and output are set to optimize routability using minimal area for each architecture. All the architectures use four-input LUTs and a switch block flexibility, F_s, of 3. In each architecture all the routing tracks are length four wires and half of the routing switches are buffered. The GILES inter-cell router uses all but three of the available metal layers for routing. The three reserved layers are used for intra-cell connections and global distribution networks for power and clocking.

We ran GILES with and without the space around the cells for these ten FPGA architectures. The tile area for each case is shown in Table 3.7. For each architecture, we calculated the ratio of the tile area without cell borders to the tile area with cell borders. The geometric average of the ratios over the ten architectures is 0.659, which means that

Table 3.6: Parameters of ten experimental architectures

# of LUTs per cluster	# of tracks (W)	$F_{c,input}$	$F_{c,output}$	Metal layers
1	32	0.56W	1.00W	7
2	56	0.44W	0.50W	8
3	80	0.30W	0.33W	8
4	96	0.23W	0.25W	8
5	120	0.19W	0.20W	8
6	144	0.15W	0.17W	8
7	160	0.13W	0.14W	8
8	176	0.11W	0.13W	8
9	192	0.10W	0.11W	8
10	200	0.10W	0.10W	8

removing the border reduces the tile area of the ten architectures by an average of 34.1%. This gives us an approximate upper bound on the gain we can achieve by merging cells and eliminating the empty space between them.

Another advantage of merging cells is that the connections between them no longer have to be routed. This eases the congestion faced by the inter-cell router. The human designer will connect the cells internally in the new cell. The designer may also be able to compact the design even further by using techniques such as diffusion region sharing.

One disadvantage of combining cells is that the separate cells may still be required in the tile. For example, a buffered switch is made up of a buffer and a pass transistor switch. The buffered switch could be implemented using a single cell or two cells that make up its components. In either case, the amount of manual layout effort is similar. However, if a single cell is used for the buffered switch and the buffers and pass transistors are used elsewhere in the FPGA then all three cells will have to be laid out. This increases

Table 3.7: Effect of cell border on tile area

Architecture	Tile area (grid squares)		Ratio
	With cell borders	Without cell borders	
1	26350	15812	0.600
2	50007	27692	0.554
3	71273	43870	0.616
4	96084	56862	0.592
5	126374	90846	0.719
6	156832	97175	0.620
7	180994	126324	0.698
8	203320	148500	0.730
9	225094	166690	0.741
10	252324	190569	0.755
Geometric average			0.659

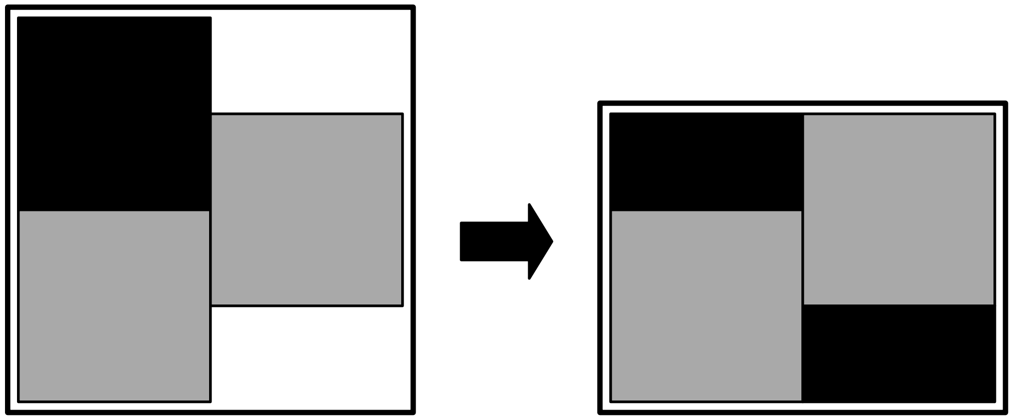


Figure 3.3: Smaller cells result in less whitespace

the number of cells and the total amount of manual layout effort.

A second disadvantage of making larger cells is that there is a point where the cells are made so large that the placer/compacter has limited freedom to place the cells. A simple example of this is shown in Figure 3.3. The placement on the left cannot be compacted any further; however, if the black cell is divided into two smaller cells then the whitespace is filled in and the placement area is reduced.

The potential for area savings is greatest when combining cells that are used more frequently. Therefore, we analyzed the distribution of cell types for the architectures in Table 3.6. The number of cells of each type in a single tile is summarized in Table 3.8. The frequency of each cell type increases linearly with the total number of cells in the tile so the percent of each cell type out of the total number of cells is approximately constant across the ten architectures. The average of this percent for each cell type across the ten architectures is shown in Figure 3.4. The number of flip-flops and LUTs each account for only 0.3% of the total number of cells in each tile so merging these cells will not save much area compared to merging SRAMs, which account for 36% of the cells. Buffers and pass transistors are also common since they form the routing fabric that occupies a larger percentage of the total area than the logic elements.

Because SRAMs, buffers, and pass transistors are by far the most common cells,

Table 3.8: Distribution of cell types

Architecture	1	2	3	4	5	6	7	8	9	10
Buffers	156	291	403	472	591	686	757	822	910	965
Flip-flops	1	2	3	4	5	6	7	8	9	10
Inverters	30	52	74	96	118	140	162	184	206	228
LUTs	1	2	3	4	5	6	7	8	9	10
Multiplexers	9	16	23	30	37	44	51	58	65	72
Pass transistors	120	210	301	360	450	540	601	660	726	750
SRAMs	169	298	440	542	695	832	940	1046	1159	1230
Total	486	871	1247	1508	1901	2254	2525	2786	3084	3265

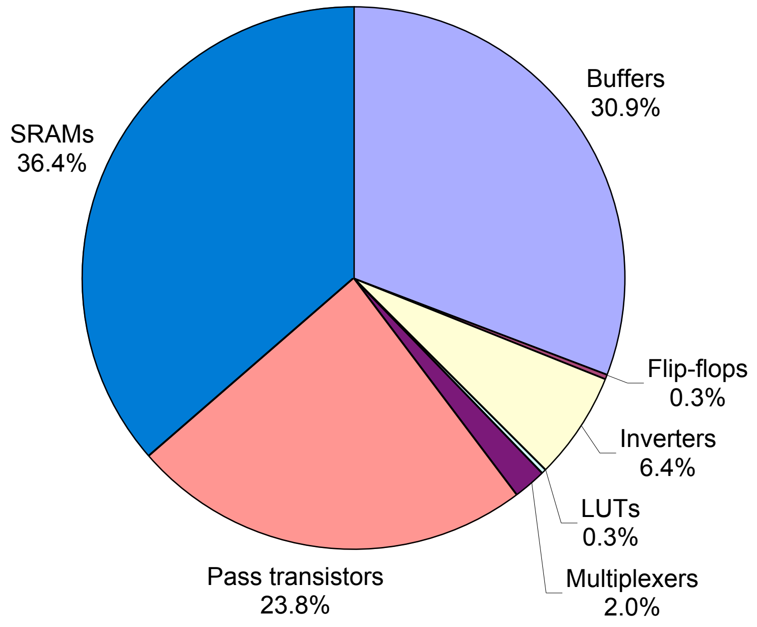


Figure 3.4: Average frequency of cell usage

we focus on grouping the transistors of these cells into larger cells. We categorize the different ways we group these cells into three categories: functional groupings, SRAM groupings, and combined groupings.

3.2.1 Functional Groupings

The goal at this stage is to search for the best way to group transistors into cells to minimize the tile area. From the discussion in the previous section, we expect that creating larger cells will reduce tile area by reducing the wasted border around each cell. Secondly, smaller cells have a larger proportion of wasted space to useful space. Finally, the more frequent a cell appears in the tile, the more area is saved when merging it with another cell.

Based on the data in Figure 3.4, the SRAMs, buffers, and pass transistors are the most common cells in our ten FPGA architectures. The pass transistor cells are also the smallest of any cell because they contain only one transistor each. These tiny cells are found only in the routing switches of our architectures. Each routing switch is controlled by an SRAM bit and in our architectures half of the switches are buffered so half of the pass transistors are preceded by a buffer. Because routing switches contain all three of the most common cells, they are a natural starting point to create larger cells by combining their components. We explore different ways of grouping these components into cells. Collectively, we call these new groupings functional groupings because they form larger functional units in the FPGA fabric.

The unbuffered switches consist of a pass transistor and an SRAM bit that controls its state. Buffered switches have a buffer that precedes the pass transistor. The first grouping we try combines the pass transistor and SRAM cells into a new cell for both buffered and unbuffered switches as shown in Figure 3.5. The second and third groupings consider just the buffered switches. The second grouping combines the buffer and pass transistor and the third grouping combines all three cells. These two groupings are illustrated in

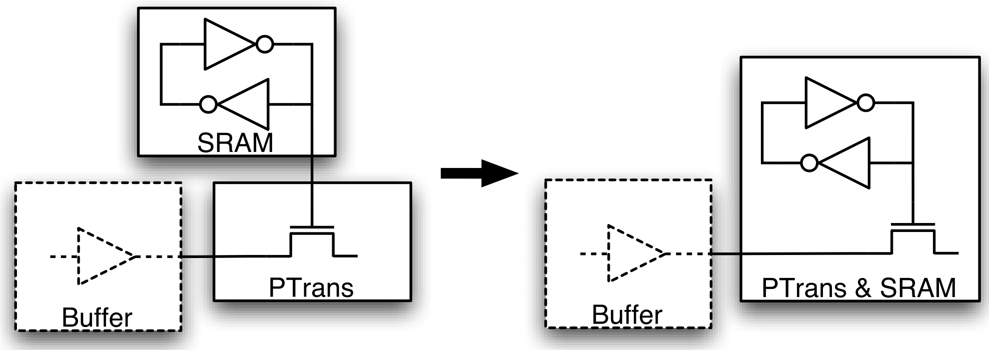


Figure 3.5: Functional grouping number 1 (PTrans & SRAM)

Table 3.9: List of functional groupings

Grouping number	Unbuffered switches	Buffered switches
1	PTrans & SRAM	PTrans & SRAM
2	No grouping	Buffer & PTrans
3	No grouping	Buffer & PTrans & SRAM
4	PTrans & SRAM	Buffer & PTrans
5	PTrans & SRAM	Buffer & PTrans & SRAM

Figure 3.6. For the fourth and fifth groupings, we use different combinations for the buffered and unbuffered switches. We combine the pass transistor and SRAM for just unbuffered switches and either the buffer and pass transistor or the buffer, pass transistor, and SRAM for buffered switches. These five groupings are summarized in Table 3.9.

Infrastructure Changes

We modified the GILES netlist generator to create a different cell-level netlist for each of the five functional groupings. For each new cell type, the area model of Section 3.1.2 is used to determine the cell's area and dimensions. The complexity factor in Equation 3.5 is set to the weighted average of the complexity factors for the cells being combined using

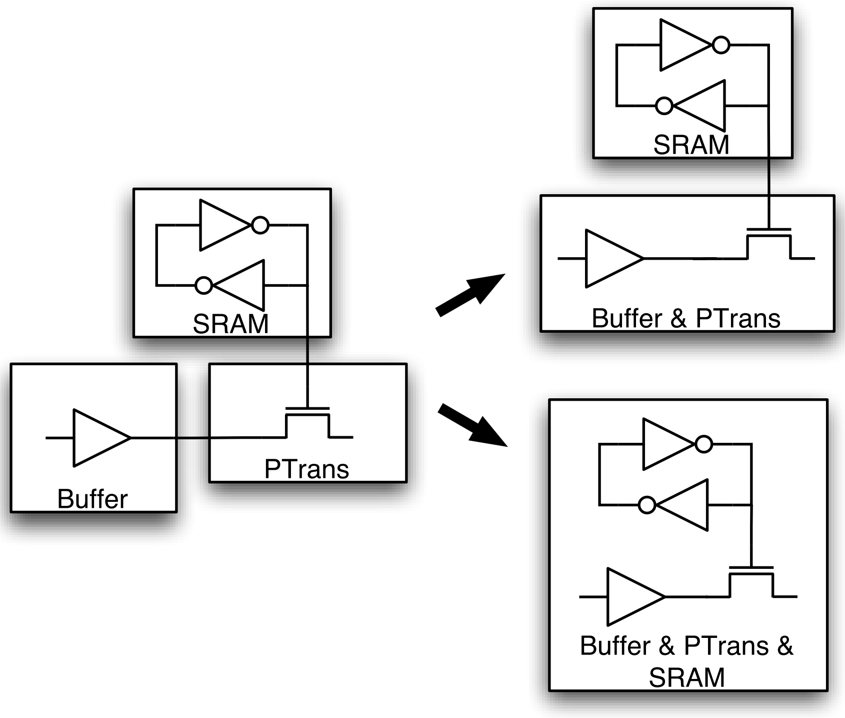


Figure 3.6: Functional groupings 2 and 3 (Buffer & PTrans, Buffer & PTrans & SRAM)

the following equation:

$$complexity = \frac{\sum_{i \in cells}(cellArea_i \times complexity_i)}{\sum_{i \in cells} cellArea_i} \tag{3.6}$$

where $cells$ is the set of cells being combined and $cellArea_i$ is the area of cell i obtained from Equation 3.3. Using this complexity factor to calculate the cell area achieves the same result as summing the areas of the cells being combined.

Experiment and Results

We ran GILES without any functional grouping for each of the ten architectures of Table 3.6 and recorded the tile area after placement and routing for each architecture. We repeated this procedure for each of the five functional groupings. For each architecture and grouping, we calculated the ratio of the tile area with grouping to that without. Then for each grouping, we obtained the geometric average of the area ratios over the ten architectures. This gives us the average tile area of each grouping relative to the area without grouping.

The average reduction in tile area for each grouping is represented as a percentage in Figure 3.7. The experimental results used to generate this graph and all subsequent graphs are listed in Appendix A. Figure 3.7 shows that functional groupings one and two achieve the same average area savings of 4.9%. However, the best functional grouping is the third grouping that combines the buffer, pass transistor, and SRAM of the buffered switch. This achieves an average 9.8% reduction in area. Groupings four and five are the same as groupings two and three, respectively, but the unbuffered switch is grouped as well. We expected that more groupings would result in more area saving; however, in this case the opposite is true. Grouping number four only achieves a 4.5% reduction in tile area compared to 4.9% for grouping two. Similarly, using grouping five results in less area savings than grouping three. This indicates that the placer is having more difficulty compacting the tile with the larger cells of groupings four and five.

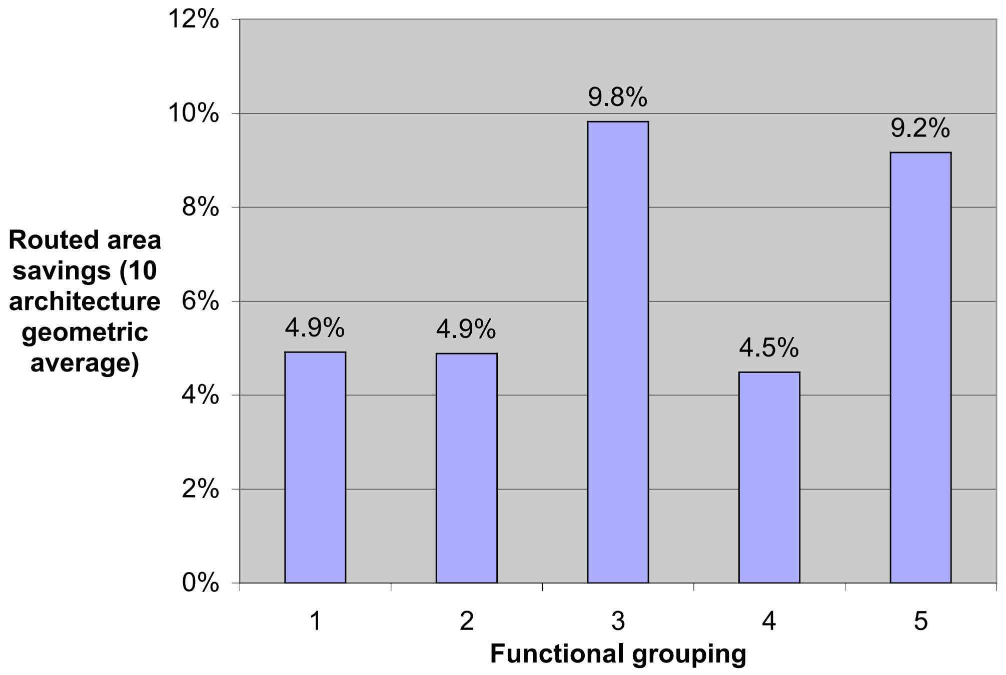


Figure 3.7: Routed area comparison for functional groupings

We confirm this point by measuring the whitespace remaining in each tile after placement. Using the same procedure as we did for the tile area, we calculated the average whitespace for each grouping relative to the whitespace without grouping. Figure 3.8 plots the average increase in whitespace for each grouping. Groupings four and five exhibit a much larger increase in whitespace compared to groupings two and three indicating the placer is having more difficulty compacting the tile. Figure 3.9 illustrates this by comparing tile placements without grouping and with grouping number five for the one four-input LUT per cluster architecture. There is more whitespace in the placement with grouping because the placer has less freedom to move the larger blocks and cannot compact the tile further. This extra whitespace reduces the area saved by increased grouping.

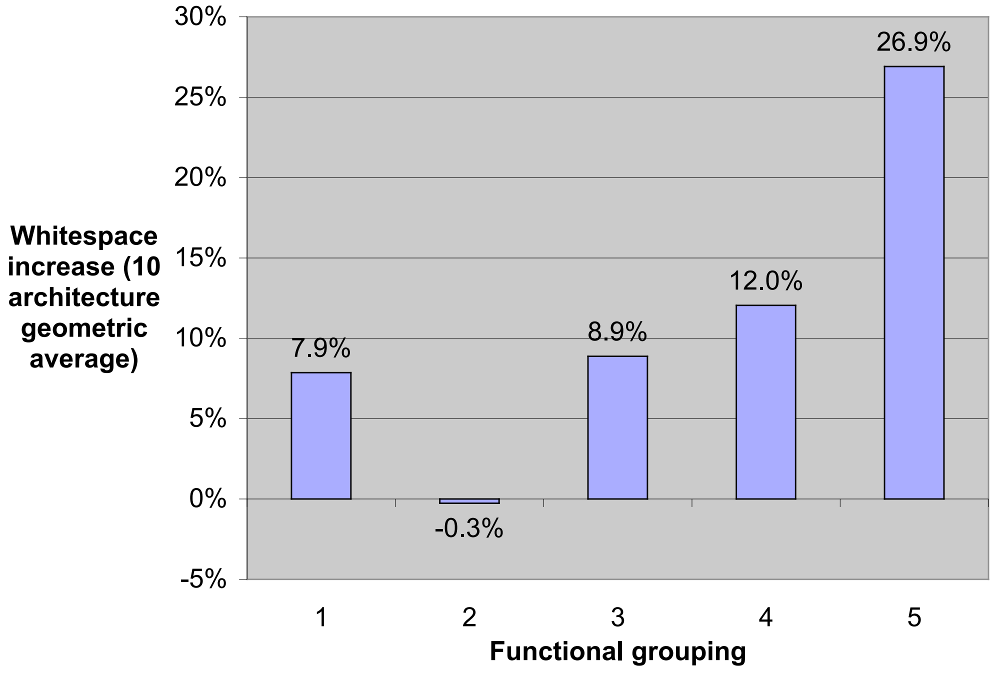


Figure 3.8: Whitespace comparison for functional groupings

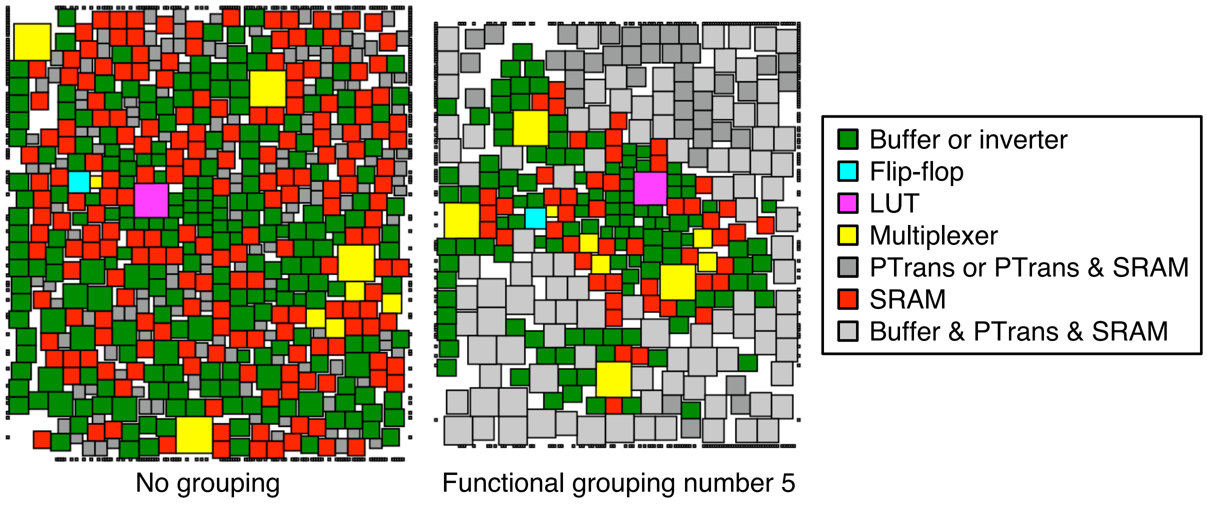


Figure 3.9: 1x4-LUT tile placements with and without functional groupings

3.2.2 Groupings of Configuration SRAMs

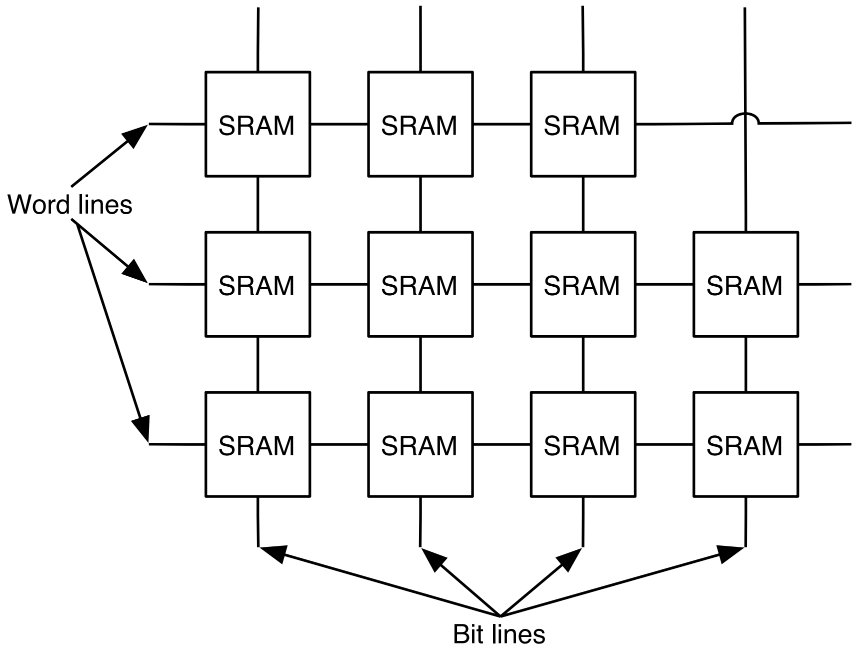
As shown in Figure 3.4, SRAM cells are the most frequently used cell type across the ten architectures. Each SRAM cell contains a single SRAM bit. It has two external connections for programming: one for enabling the write into the cell and the other for specifying the bit to be stored in the cell. The SRAM cells are arranged in a two dimensional array and the programming lines are connected horizontally and vertically into word and bit lines as shown in Figure 3.10a. Since SRAM cells share programming signals, we explore grouping varying numbers of these cells together as shown in Figure 3.10b.

Tradeoffs

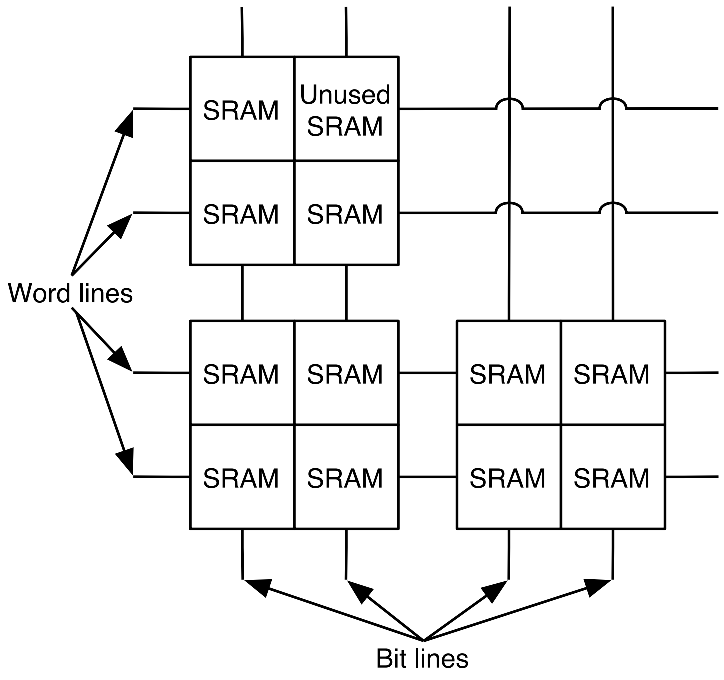
Similar to functional groupings, SRAM groupings reduce the wasted border around each cell but the placer has trouble eliminating the whitespace when the cells become too large. Also, by grouping SRAM cells, the length of the wires that connect to them is affected. We categorize the wires connected to SRAM cells into those used for loading the SRAM and those used to output the stored values to the programmable elements. We call these the SRAM programming wires and SRAM output wires, respectively.

In some cases as in Figure 3.10, there are additional programming lines when grouping is used. However, this is offset by fewer programming connections between cells. For example, the 2x2 SRAM group contains four SRAM bits. Each SRAM bit needs to be connected to both a word and a bit line. Instead of exposing eight connections to the cell, the SRAM group is internally connected in a two by two array. Therefore, only two word and two bit lines are connected externally, making the inter-cell router's job easier. This has a large impact on reducing the programming wirelength since the SRAM cells are typically not placed in a straight line as they are shown in Figure 3.10.

The length of the SRAM output wires always increases with grouping. The more SRAM bits are grouped together, the more centralized they become as shown in Figure 3.11. The output wires still must connect to programmable elements in other cells



(a) Array of SRAM bits



(b) Array of SRAM bits grouped into 2x2 SRAM cells

Figure 3.10: Grouping of configuration SRAMs

yet it becomes harder to place these cells close to the SRAM bits that control them. Not only are there more cells that need to be close to the grouped SRAM but the SRAM bit might be in the middle of the SRAM cell, not its edge. Therefore, the output wirelength increases making the router's task more difficult and causing the router to increase the tile area to ease congestion.

Infrastructure Changes

To experiment with groupings of SRAM cells, the GILES netlist generator and placer were modified to support SRAM cells with multiple bits. The netlist generator was modified to create SRAM cells that contain two dimensional arrays of SRAM bits such as the 2x2 array shown in Figure 3.10b. The dimensions of the SRAM arrays are specified as input to the program. All the SRAM cells in the tile have the same dimensions so that only one SRAM cell is needed for the tile layout. Therefore, when the total number of SRAM bits in the tile is not a multiple of the array size, extra bits are added to the netlist that are not used to configure any programmable element. These extra bits occupy unnecessary area so they reduce the effectiveness of larger groupings.

The previous version of the netlist generator adds SRAM cells to the netlist one by one as they are needed to configure programmable elements. Padalia [24] chose to make the netlist generator randomize the programming connections of the SRAM cells so that when the placer optimizes for wirelength, it is not influenced by the order that the SRAM cells were added to the netlist. Instead, the placer positions the SRAM cells near the programmable elements they control and rewires the programming connections to reduce their wirelength as shown in Figure 2.18.

Similarly, when grouping SRAM bits into cells, rather than grouping as the bits are added to the netlist, our version of the netlist generator collects the bits and groups them randomly after all the bits are known. The random grouping of bits prevents the placer from being influenced by the order the bits are added to the netlist. We let the placer

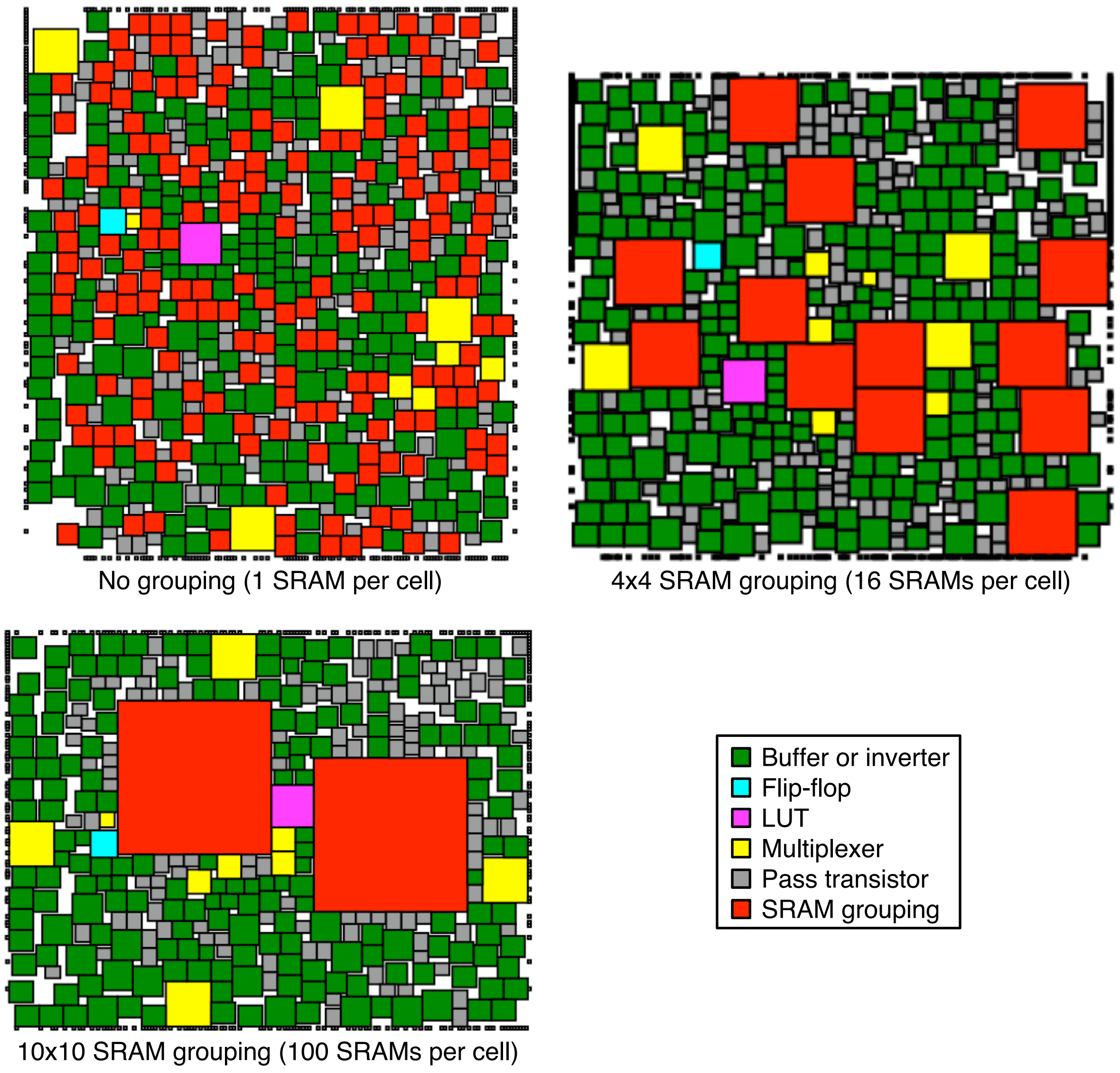

No grouping (1 SRAM per cell)
4x4 SRAM grouping (16 SRAMs per cell)
10x10 SRAM grouping (100 SRAMs per cell)
🟩 Buffer or inverter
🟦 Flip-flop
🟪 LUT
🟨 Multiplexer
⬜ Pass transistor
🟥 SRAM grouping

Figure 3.11: 1x4-LUT tile placements with groupings of 1x1, 4x4, and 10x10 SRAMs

rearrange the SRAM bits to optimize the groupings and reduce the wirelength between each bit and the programmable element it controls. After placement, the SRAM bits are grouped based on the locations of the programmable elements they control rather than a random or netlist sequential grouping. This procedure is illustrated in Figure 3.12.

To create the initial random grouping, the netlist generator stores connectivity information about each SRAM bit as it emits the netlist. Once it has a list of all the SRAM bits, it adds extra bits to round the total number up to a multiple of the array size of the SRAM cell. Then it randomly chooses bits for each SRAM group and connects the programming lines between them. The programming lines are not connected randomly since the SRAM bits in each cell are already chosen randomly.

The netlist is then passed to the GILES placer. The placer was modified to handle reweaving of programming lines when SRAM cells are grouped as well as swapping of SRAM bits to improve on the initial random grouping. The details of these changes are now discussed.

As shown in Figure 2.18, the placer “reweaves” the SRAM word and bit lines so that the logical equivalence of the SRAM bits is used to reduce the wirelength of the programming lines [27]. We enhanced this reweaving process to handle cells containing multiple SRAM bits since each grouped cell has multiple word and bit lines as shown in Figure 3.10b. The placer reweaves the vertical bit lines first. It orders the SRAM cells based on their horizontal position. Cells closest to the left of the tile are chosen to connect to the first bit line. To accommodate grouped SRAM cells, additional bit lines are connected to the same set of cells until all the bit lines for those cells are connected. Then the next set of cells closest to the the left of the tile are chosen and they are connected to a different set of bit lines. After all the cells have been connected to bit lines, the process is repeated with the horizontal word lines starting with the cells closest to the bottom of the tile. Care is taken to avoid connecting a word line to two SRAM bits with the same bit line as shown in Figure 3.13. Connecting two SRAM bits in this

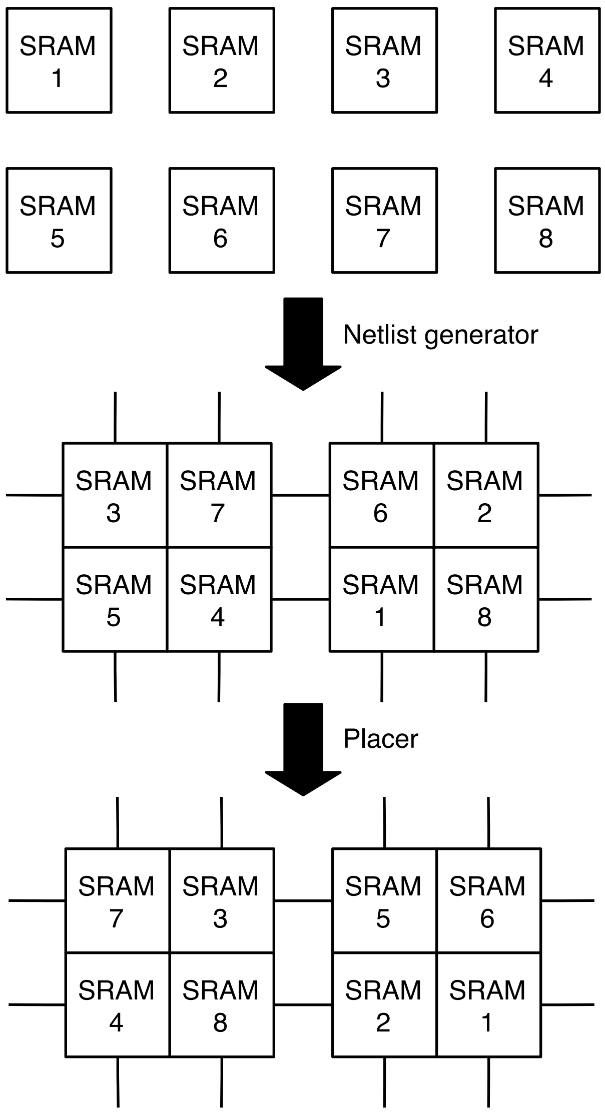


Figure 3.12: Netlist generator creates random groupings then placer optimizes them

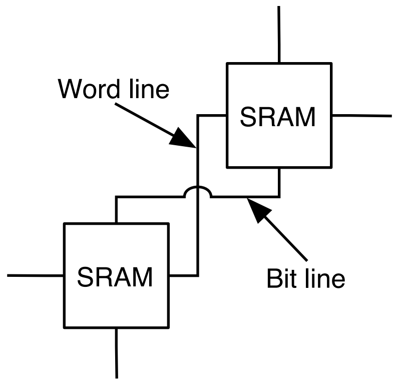


Figure 3.13: SRAM bits cannot be programmed with different values

way would prevent programming the two SRAM bits to different values.

The other change to the placer adds the ability to swap SRAM bits to improve on the initial random grouping. To reduce the SRAM output wirelength, the placer is able to swap the location of SRAM bits within a cell or between cells. The previous version of the placer is capable of swapping connections of functionally equivalent pins such as the output and inverted output of an SRAM bit as shown in Figure 2.17 [27]. For grouped SRAM cells, this feature is also used to swap the output connections of one SRAM bit with those of another in the same cell. The only change to the placer involves specifying that the output pins of each SRAM bit in a cell are functionally equivalent. To expand this feature to include swapping of SRAM bits in *different* cells, we enlarged the placer's scope of possible bit swaps to include bits in different SRAM cells. Examples of intra-cell and inter-cell bit swap moves are shown in Figure 3.14. This allows the placer to connect programmable elements to SRAM groups that are nearby without moving an entire group. It also prevents the SRAM groups from being pulled in different directions to minimize the wirelength between their initial random connections.

To determine the importance of inter-cell SRAM bit swapping, we ran GILES using the 2x2 SRAM grouping with and without this feature for the ten FPGA architectures of Table 3.6. We calculated the ratio of some tile statistics with and without the swapping

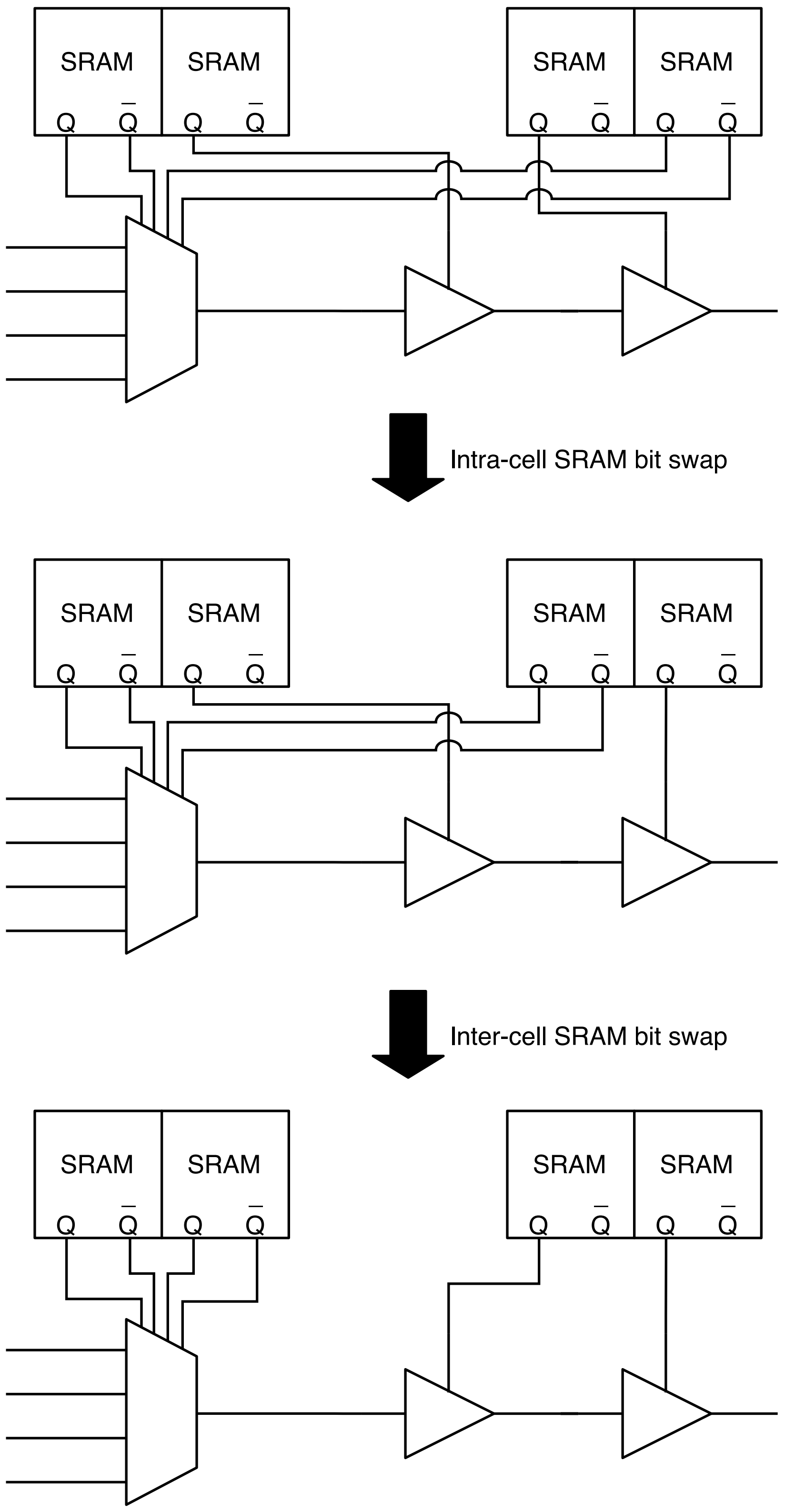


Figure 3.14: SRAM bit swaps reduce wirelength

Table 3.10: Benefits of inter-cell SRAM bit swapping for the 2x2 SRAM grouping

	Increase compared to no grouping (10 architecture geometric average)	
Tile statistic	Without inter-cell SRAM bit swapping	With inter-cell SRAM bit swapping
Placed area	-7.4%	-7.8%
Routed area	-1.0%	-8.6%
Total wirelength	+17.5%	+3.1%
SRAM programming wirelength	-21.5%	-21.4%
SRAM output wirelength	+162.4%	+38.1%

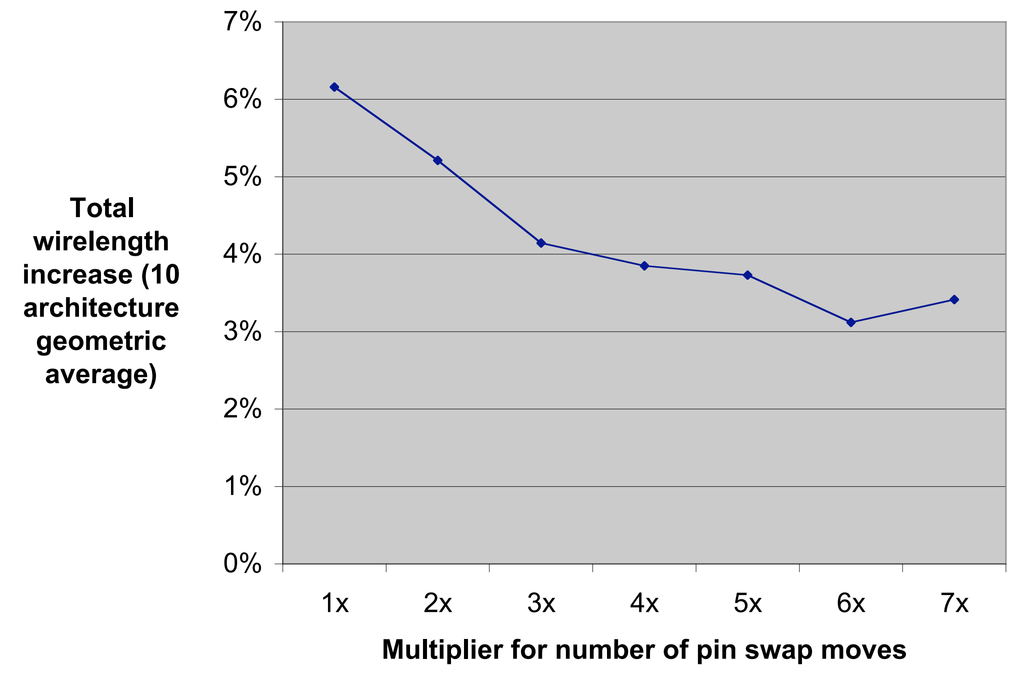
feature compared to the case without grouping. Table 3.10 shows the geometric average of these results over the ten architectures. Regardless of whether inter-cell SRAM bit swapping is used, the improvements in placed area with the 2x2 SRAM grouping remain the same. However, the routed area decreases by only 1.0% without the new feature but decreases by 8.6% with the new feature. The additional decrease in routed area is because the router experiences less congestion due to less total wirelength when the new feature is added. To understand the difference in wirelength, we examine the SRAM programming wirelength and SRAM output wirelength. Whether the inter-cell SRAM bit swapping feature is used or not, the SRAM programming wirelength is reduced similarly. This is expected because the programming wires are not affected by which SRAM bits are grouped together. The SRAM output wirelength increases substantially when grouping SRAM bits without the inter-cell bit swap move because the placer has more difficulty positioning the SRAM bits close to the programmable elements they control. However, when the placer is able to swap SRAM bits between different cells, it reduces the output wirelength by grouping SRAM bits that control nearby programmable elements.

Even with inter-cell SRAM bit swapping, the SRAM output wirelength is 38.1% larger than without grouping. The placer has less flexibility to reduce the output wirelength when grouping SRAM bits but some of this increase may still be attributed to sub-optimal SRAM groupings. The placer has several types of moves it can attempt. The swapping of functionally equivalent pins, of which SRAM bit swapping is a subset, is only attempted for 4% of all placer moves. Fung found that 4% obtained the best results with the original pin swapping move [27]. However, we have added the inter-cell SRAM bit swap move that was proven to be critically important. Therefore, we tried increasing the number of pin swap moves performed by the placer.

We ran GILES repeatedly using the 4x4 SRAM grouping with up to seven times the default number of pin swap moves. We recorded the placed area, the routed area, the total wirelength, and the SRAM output wirelength for each of the ten architectures and compared them to the case without grouping. As shown in Figure 3.15, increasing the number of pin swap moves does reduce the SRAM output wirelength and the total wirelength. However, Figure 3.16 shows that the routed area stays relatively flat except for small fluctuations that are attributed to the randomness of the tools. The routed area is limited by the placed area so improvements to the routed area are only possible when the tile is congested. In this case the tile is not congested so for now we leave the number of pin swap moves at the default value to reduce runtime.

Experiment and Results

Now that GILES is able to handle SRAM groupings, we examine different grouping sizes to determine the ideal number of SRAM bits per cell that minimizes the tile area across the ten FPGA architectures. We explore grouping sizes of 2x2 to 10x10 SRAM bits. Since the GILES placer performs better with square cells, we keep the array size of each cell to be roughly square to make it easier for the layout designer to make a square cell. We ran GILES for each SRAM grouping and each of the ten architectures and obtained ratios



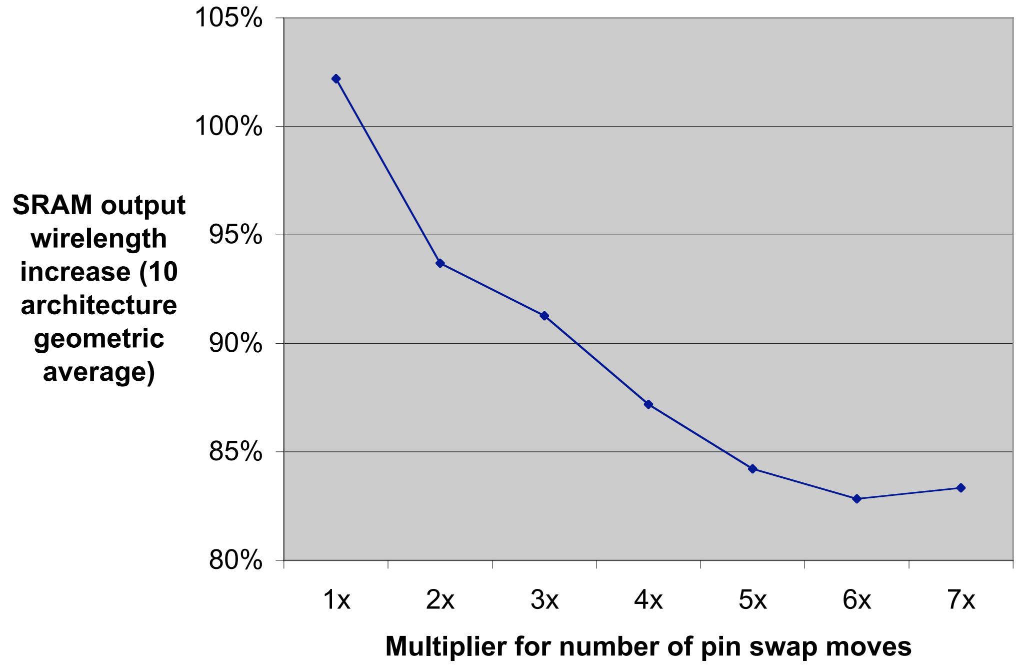


Figure 3.15: Number of pin swap moves versus wirelength for 4x4 SRAM grouping

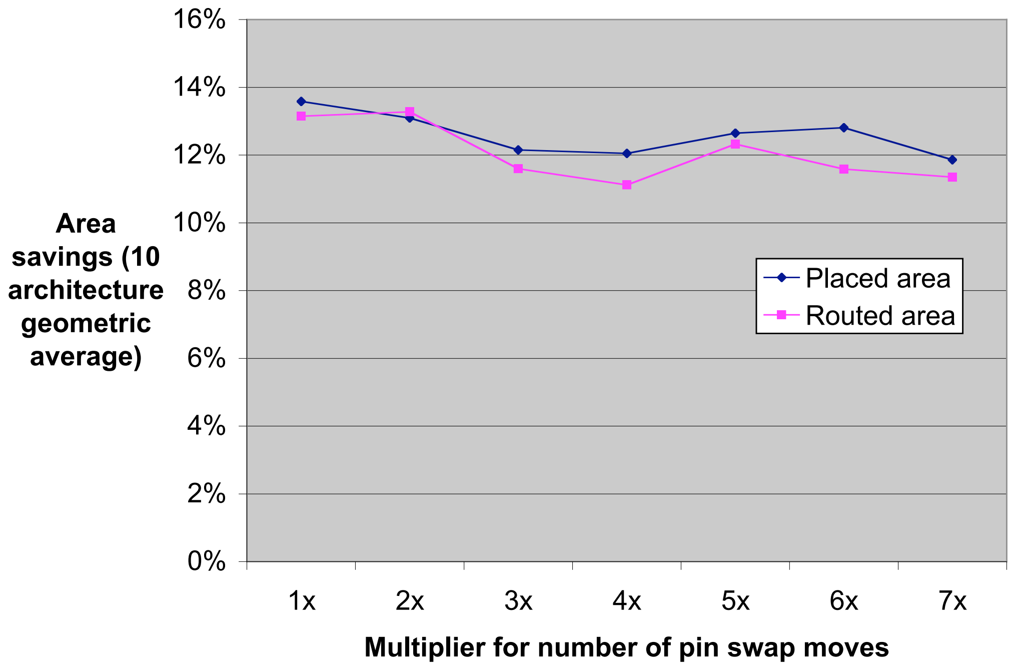


Figure 3.16: Number of pin swap moves versus area for 4x4 SRAM grouping

for the placed area and the routed area when using the SRAM groupings compared to no grouping. The geometric average of the ratios for the ten architectures was obtained for each grouping and is represented as a percent increase relative to the case without grouping in Figure 3.17.

SRAM grouping successfully decreases the routed area as the grouping size increases up to the 4x4 grouping. The area savings remain near 13% until it increases for the larger groupings starting at 7x7. Unlike the functional groupings, the reason for this increase is not due to an increase in whitespace after placement. The whitespace stays relatively constant for all groupings. As shown in Figure 3.17, the placement area follows the same trend as the routed area but it does not increase for larger groupings. Therefore, the router is increasing the area due to heavy congestion.

To confirm this conclusion, we examine the increase in wirelength for each SRAM grouping compared to the case without grouping and average the results across all ten

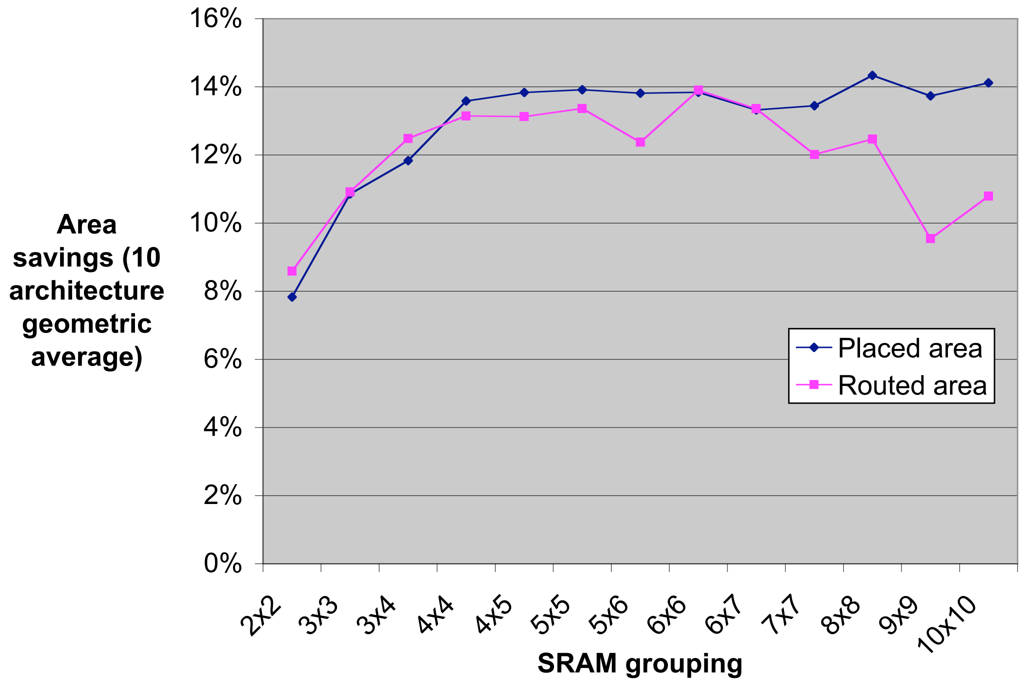
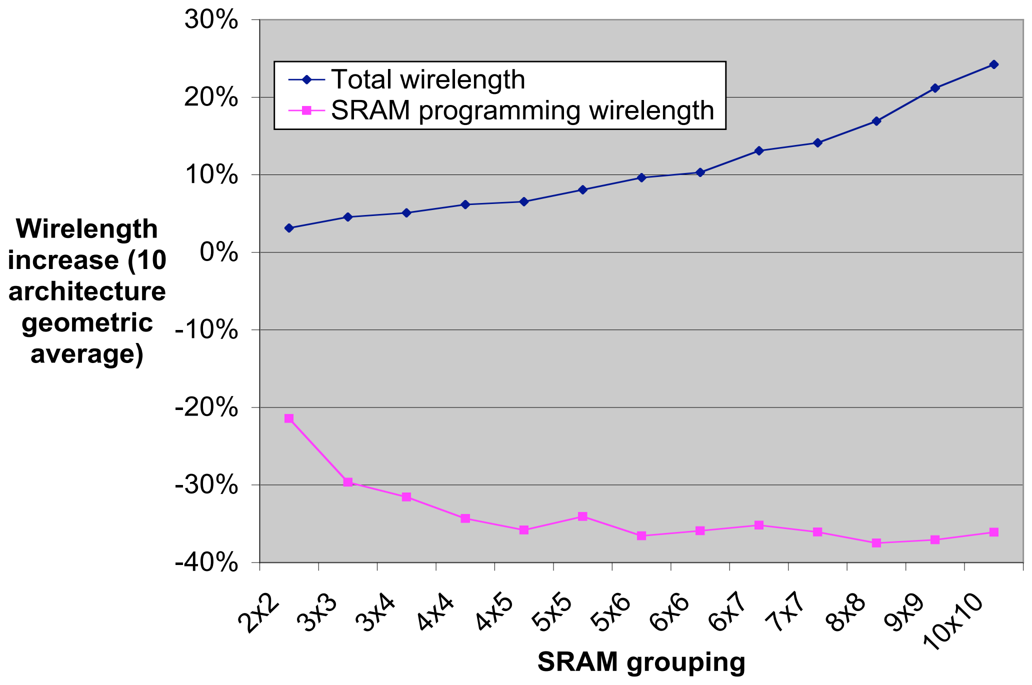


Figure 3.17: Area comparison for SRAM groupings

architectures in Figure 3.18. In the first chart, the total wirelength increases with larger SRAM groupings up to 24.2%. Again, we examine the two classifications of SRAM wires: programming wires and output wires. With larger groupings, the programming wirelength decreases up to 36.1% and the output wirelength increases up to 280%. The total wirelength increases because the wirelength of the output lines increases much more rapidly than the wirelength of the programming lines is reduced. Therefore, for larger groupings the total wirelength increases and in addition, the router has less area to start with so it increases the area needed to ease congestion.

Since the SRAM output wirelength is causing congestion for larger groupings, we considered increasing the number of pin swap moves performed by the placer to reduce it. However, groupings larger than 4x4 have similar placed area so even if the congestion is eased in the larger groupings they will not achieve significantly better results than the 4x4 case. Therefore, it is best to choose a smaller grouping with less congestion so



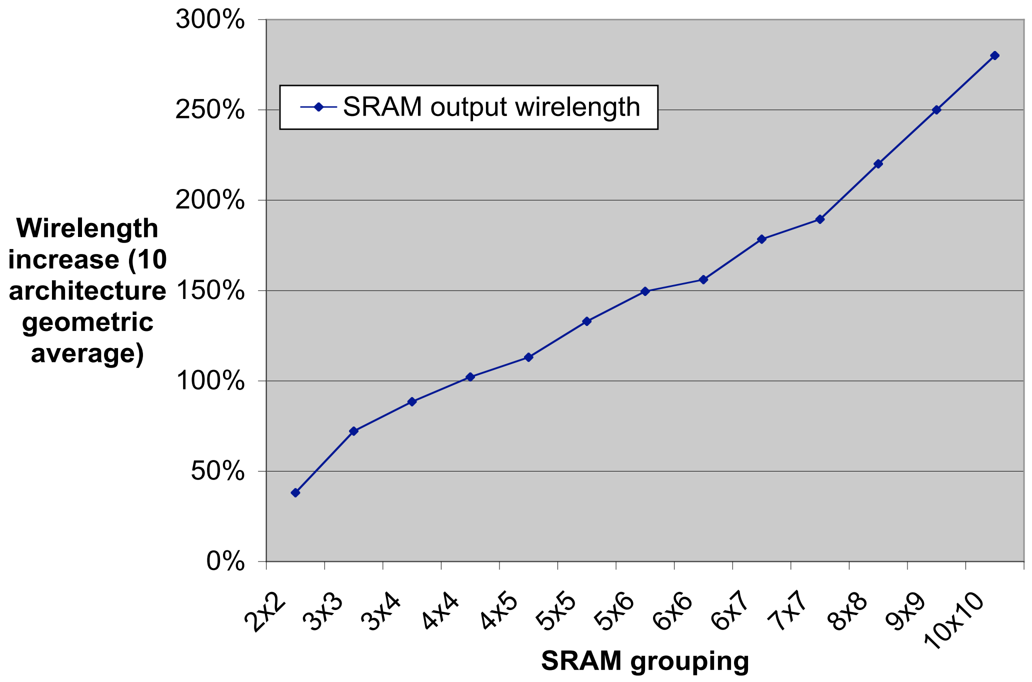


Figure 3.18: Wirelength comparison for SRAM groupings

increasing the number of pin swap moves and hence the runtime is avoided.

In summary, these results indicate that the best SRAM grouping occurs with the 4x4 cell because it achieves an average routed area savings of 13.1% with minimal manual layout effort, runtime, and less chance the increase in wirelength will cause problems for the router. This conclusion may also be applicable to custom FPGA designers when choosing how to lay out the configuration SRAM bits.

3.2.3 Combined Groupings

To achieve more area savings, we considered combining functional groupings with SRAM groupings. However, four out of the five functional groupings include a single SRAM bit that causes problems when reweaving the word and bit lines of grouped SRAM cells. Therefore, we only combine functional grouping number two (Buffer & PTrans) with SRAM grouping. The tradeoffs of combined groupings are the same as with the previous grouping types and no further infrastructure changes are needed to support combined groupings.

Experiment and Results

We ran GILES with functional grouping number two and SRAM groupings with sizes varying from 1x1 to 8x8. Using the 1x1 SRAM grouping is the same as functional grouping two without SRAM grouping. As with SRAM groupings, we recorded the placed area and routed area of the tile for each grouping and each of our ten FPGA architectures. The ratio of the area with grouping to the area without grouping is obtained and averaged across the ten architectures.

Figure 3.19 shows the average savings for the placed area and the routed area for each combined grouping. Both areas decrease with larger SRAM groupings, but do not vary significantly for groupings larger than 4x4. The larger groupings provide fewer gains but require more manual layout effort and risk causing routing congestion. Therefore,

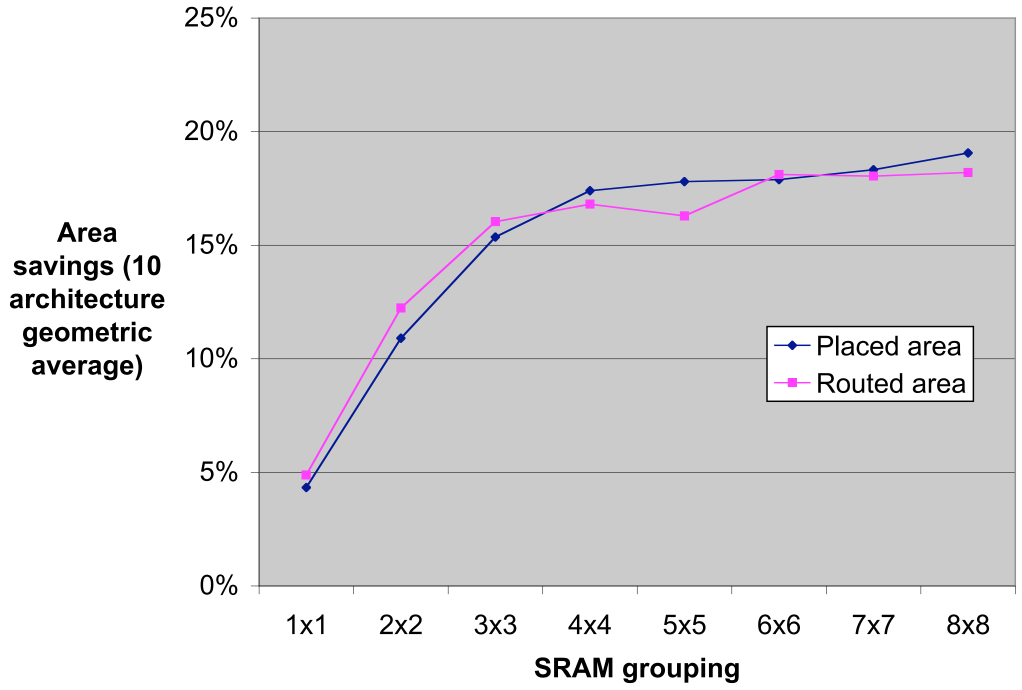


Figure 3.19: Functional grouping number 2 combined with SRAM grouping

the 4x4 SRAM grouping achieves the best balance between area reduction and increases in wirelength and manual layout effort. The combined grouping of 4x4 SRAM with functional grouping two outperforms all separate functional and SRAM groupings with a routed area savings of 16.8%.

3.3 Tile Area Compared to a Commercial FPGA

Kuon [31] compared the area of a Xilinx Virtex-E tile to the area of an approximated Virtex-E tile generated by GILES. His comparison is more accurate than the one by Padalia et al. [5], which found that the tile generated by GILES is 47% larger than the actual Virtex-E tile. Kuon's more accurate tile is 198% larger than the Virtex-E tile. He attributes most of this increase to routing congestion that caused the router to increase the area. To reduce the area of the generated tile, Kuon experimented with the transistor groupings explored in this work. SRAM groupings and combined groupings increase congestion so they were found to further increase the tile area. However, using the best functional grouping, which grouped the buffer, pass transistor, and SRAM in the buffered switch, reduced the area difference to 187%. To ease congestion, Kuon added an extra layer for inter-cell routing that was previously used for intra-cell routing to reduce the tile area of the GILES Virtex-E to be 36% larger than the actual Virtex-E tile. If two extra metal layers are used with GILES in addition to the six metal layers used in the real Virtex-E, then the tile generated by GILES is only 13% larger. Kuon also compared the real Virtex-E tile to a comparable standard cell implementation and found the standard cell version to be 102% larger than the custom implementation. This shows that our automatic layout system obtains smaller designs than commercial standard cell tools and comes remarkably close to the area of custom layouts.

This chapter studied the choice of cells used by the automatic layout tool. The best functional grouping helped Kuon reduce the area of an approximated Virtex-E tile. The best grouping found in this work was a combined grouping that used the buffer and pass transistor functional grouping and the 4x4 SRAM grouping. The approximated Virtex-E tile would likely benefit from this combined grouping now that the congestion has been reduced by adding inter-cell routing layers. If routing congestion is still a problem then the number of the pin swap moves could be increased to reduce the extra wirelength added by the SRAM grouping. The next chapter uses the improved transistor groupings when designing the first complete FPGA to be generated using this automatic layout tool.

Chapter 4

Automatic Layout of a Complete FPGA

Prior to the present work, the GILES automatic layout system for FPGAs had only been used to layout a single FPGA tile. That tile generation has been improved upon in this work. It has also been compared to commercial FPGA layouts [5, 31]. The next step is to prove that GILES can be used to fabricate an FPGA. The proof-of-concept FPGA we created is called Pushbutton Optimized Widely Erasable Logic Layout (POWELL), an admittedly somewhat tortured acronym. It is the first FPGA designed automatically, beginning with an architecture description and proceeding to layout. It was implemented in a six metal layer, 0.18 μm fabrication process from the Taiwan Semiconductor Manufacturing Company (TSMC) [32]. Access to this technology was provided by the Canadian Microelectronics Corporation (CMC) [33].

This chapter describes all the steps involved in creating this chip. It begins with the architecture of the FPGA and discusses all the issues through to the final layout that is programmed and verified. Along the way it solves some circuit design issues that were encountered. It also discusses our work to integrate GILES with Cadence's Virtuoso custom design platform [34] so our design is compatible with an industrial design flow.

This work was performed jointly with Ian Kuon [31]. Work attributed to Kuon will be identified as it is discussed.

One challenge of fabricating a chip with GILES is that the generated tile layout has never been tested for functionality. This issue is addressed in this chapter and related problems are solved that were neglected in the past. For example, the focus of the layout system was to create a single tile. The system does not consider the periphery of the FPGA for connecting I/O pads. Also it connects the power and clock signals internal to the tile, but does not consider how these signals are distributed to the array of tiles. Finally, there are many problems with the way GILES handles the programming infrastructure that must be solved for a real chip to function.

4.1 Architecture

The architecture of an FPGA is based on the design requirements. POWELL did not have to be large because it is a proof-of-concept, not a commercial product. Also, silicon area is expensive and CMC sets limits on its grants of silicon area so the design had to be small. To make the results applicable to commercial products, we kept in mind that the automatic design system must be scalable to large FPGAs and POWELL must include realistic features of commercial FPGAs. For example, if there is only one BLE per cluster then there will be no intra-cluster routing to test except for feedback into the single BLE. Also, if length four routing tracks are used with a four by four array of tiles, then we cannot test signals that span multiple routing tracks. Therefore, a small architecture that is representative of features found in large FPGAs was chosen. Kuon determined appropriate architectural parameters and verified that routable circuits are possible on the FPGA [31].

Based on Kuon's research, POWELL uses four-input LUTs and contains three BLEs per cluster with each cluster having eight inputs. As with all GILES architectures there is one logic block per tile, which is composed of one logic cluster. The tiles are arranged in an eight by eight array so there are a total of 64 logic blocks and 192 LUTs. The array is surrounded by two I/O pads per tile for a total of 64. The routing architecture consists of 20 tracks per channel. All routing tracks are length four and only use bidirectional buffered switches. The architecture avoids unbuffered switches because of the pass transistor circuit issues discussed in Section 4.2.1. The values of $F_{c,input}$, $F_{c,ouput}$, and $F_{c,pad}$ are 12, $\frac{20}{3}$, and 12 respectively. The complete architecture description is given in Appendix B.

4.1.1 Periphery Tiles

The previous research [5] did not consider how to implement the periphery of the FPGA. This, of course, is an important part of laying out an entire chip. As shown in Figure 4.1, periphery tiles are needed on all four sides of the array as well as three corners. The top-right corner does not require a periphery tile because no additional logic is needed for that location. The logic required for the other periphery locations is discussed in this section. For more details on how the periphery tiles are generated including their implementation and how they connect to the main tiles, see Kuon's work [31].

Figure 2.6 illustrates the full FPGA array, created by replicating a single tile. The routing tracks are always on the top and right of the tile. Therefore, logic blocks on the bottom and left of the array do not have routing tracks on all sides. Because the inputs to logic blocks come from all sides, these routing tracks must be added. This problem is solved by creating periphery tiles to implement the missing routing tracks as shown in Figure 4.1. Two different tiles are needed for the bottom and left of the array and one for the bottom-left corner.

Another problem involves how the I/O pads will connect to the array. Some logic is needed to select which routing track will connect to each pad and the direction of that signal. This is implemented in the top, bottom, left, and right periphery tiles. In

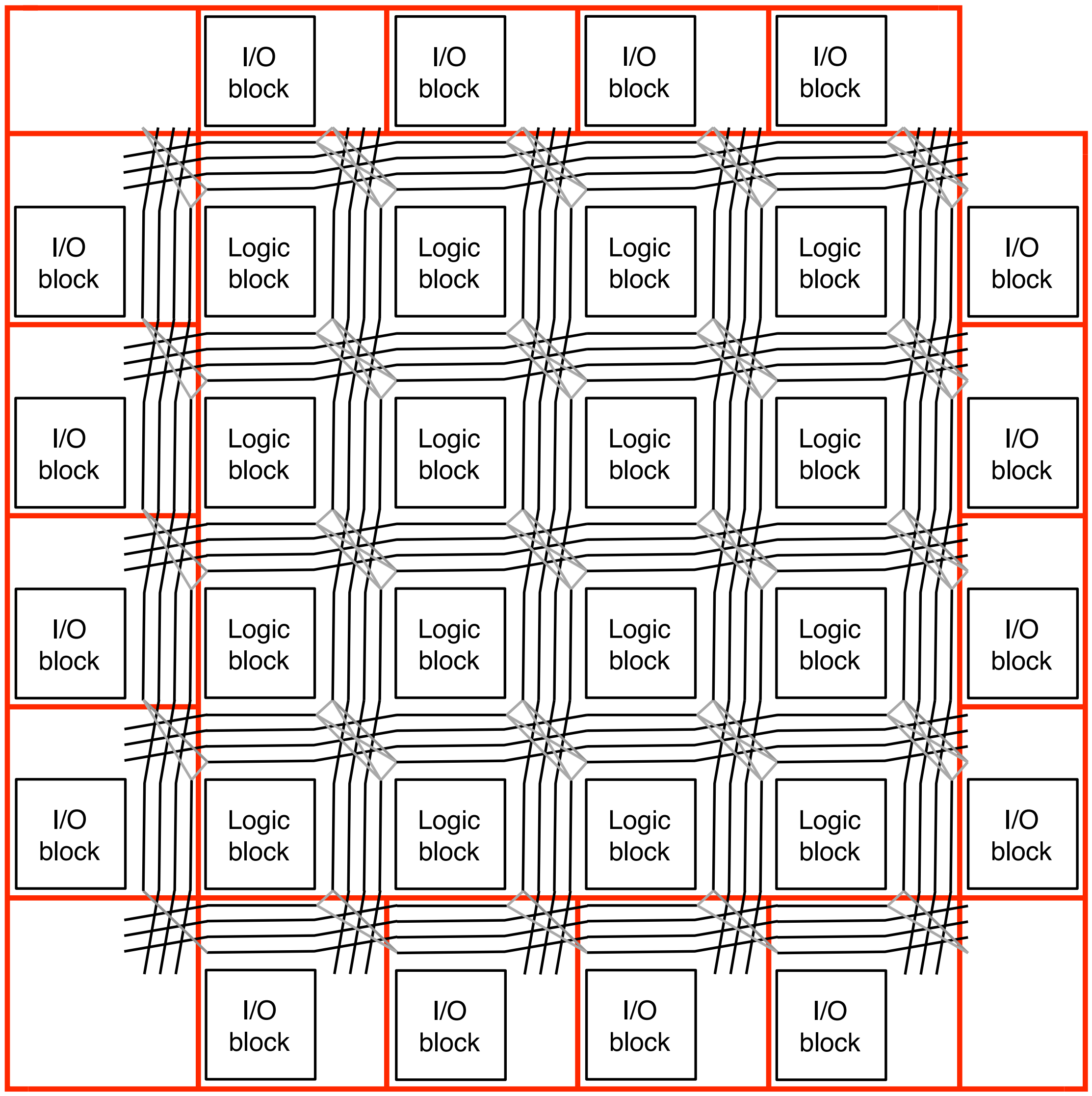


Figure 4.1: Array and periphery tiles

addition, some circuitry is needed for programming the FPGA that is implemented in the bottom and left sides of the array including the three corner tiles. This is discussed in Section 4.2.2.

4.2 Circuit Design

The circuits used to implement POWELL are based on those used in the previous version of GILES [24]. They are similar to those by Betz et al. [4] with the exception of the configuration SRAM cell, which does not require an inverted input for programming, and the flip-flop, which does not have asynchronous set and reset functionality. The flip-flop used in POWELL also has an extra inverter on the end so it is non-inverting. Other exceptions to the circuits described in the previous work are noted in this section.

4.2.1 Level Restorer

The circuits produced by the GILES system use NMOS pass transistors to implement multiplexers and routing switches. NMOS transistors pass a low logic level through them without a problem, but they cause a voltage drop equal to the threshold voltage when passing a high logic signal [1]. The next inverter in the signal path will receive the degraded voltage causing the PMOS to be slightly on and possibly leak power. Betz et al. use a boosted gate voltage for routing switches to allow the NMOS transistor to pass the full voltage level [4]. They do not address the issue for multiplexers. The previous version of GILES neglects this problem for both multiplexers and routing switches.

The complications that arise from using two voltage levels on one chip forced the consideration of options other than the gate boosting approach. Transmission gates use complementary NMOS and PMOS pass transistors to properly transmit both logic levels [1]. However, the additional PMOS transistors require significant extra area considering they require n-wells and have to be added to every routing switch and every

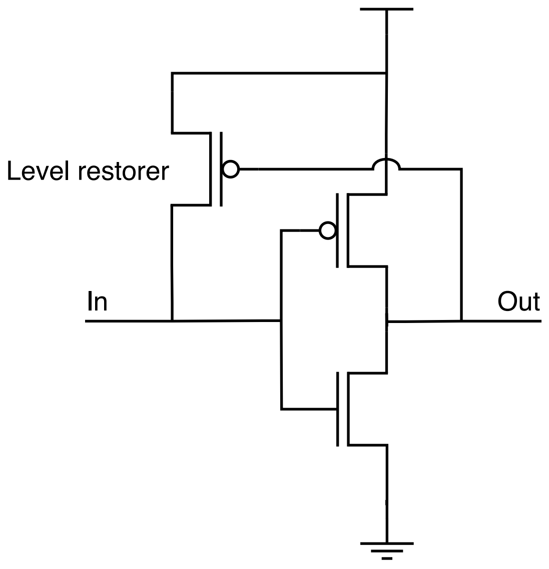


Figure 4.2: PMOS level restorer pulls high logic level to full voltage

transistor in the multiplexers.

Instead, a more area-efficient option was chosen. For any inverter cell following a routing switch or multiplexer, a PMOS level restorer was added as shown in Figure 4.2. When a high logic level arrives at the inverter, the low output signal turns on the PMOS level restorer to pull up the input to the full voltage level [1]. When the input signal is low, the output is high and the PMOS level restorer is disabled.

A problem with this approach is that the feedback loop could prevent the input to the inverter from being driven low. The PMOS level restorer tries to keep the voltage high so it must be weaker than the input driver. Compounding the problem is the fact that each routing switch typically drives multiple inverters since each routing track has many receivers and spans four tiles. Therefore, level restorers in each inverter could combine their strengths to prevent the routing switch from driving its output low. To avoid this situation, only one level restorer is added per routing track. A new level restorer cell that contains an inverter and PMOS level restorer is connected to each routing track. The cell's input connects to the routing track and the output is left unconnected. Simulations were performed to size the PMOS level restorer such that it does not overpower the drivers but also quickly pulls up the signal on the heavily loaded routing track. Kuon modified the netlist generator to add the level restorer only to routing tracks that start in the tile. Tracks that span the tile or end in the tile will connect to a track with a level restorer when the tiles are replicated to form the array.

Unlike the routing switches, the multiplexers (which are used for the input connection blocks, the logic cluster crossbars, and the output selectors of the BLEs) typically drive a single inverter. Therefore, a PMOS level restorer is added to the inverter cell instead of using the dedicated level restorer cell so the transistors can be sized differently. The level restorers following multiplexers cannot be as strong as the ones on the routing tracks because the signal strength is weakened after passing through several stages of pass transistors in the multiplexers. Instead of increasing the strength of each multiplexer's input drivers to overcome the level restorer, a weaker level restorer is used. The stronger level restorer cell is used for the routing tracks so the routing performance is improved with the faster pull-up rate.

The 1x drive strength inverter cell is present at the output of every multiplexer so a PMOS level restorer was added to this cell. However, the cell is also used in places where the PMOS level restorer is not needed and could prevent proper functionality. In these cases, a 2x drive strength inverter is used instead since the extra drive strength is not an issue and the area impact is minimal. Kuon performed simulations that found other cases where the level restorers were too strong for the drivers. This resulted in resizing the PMOS level restorer, adding a level restorer to the output of each LUT, and buffering the output of each SRAM bit that connects to a LUT so the SRAM bits are isolated from the level restorer on each LUT output.

4.2.2 Programming Infrastructure

The programming infrastructure is perhaps the most important part of the FPGA. If some portion of the logic or routing fabric does not work then the programming can route around the problem. However, if the programming does not work then the chip is useless. Kuon [31] performed the design and verification of the programming circuitry. The circuit design issues that arose from his work are summarized here.

The SRAM cells allow the FPGA to be programmed. Therefore, it is critical that they perform correctly. This involves writing values into the SRAM and making sure those values are not likely to change. The GILES system employs a single-sided write to SRAM bits instead of the more robust two-sided write so as to reduce the number of connections [24]. Kuon sized the transistors of the SRAM cell and verified that the cell works correctly.

The SRAM bits are arranged in a rectangular array. Horizontal connections, or word lines, select which row will be enabled for writing. Vertical connections, or bit lines, drive values to be written into the SRAM. These lines extend across the length of each tile. When the tiles are arranged in an array, the programming lines extend across the entire length of the chip. Kuon decided to buffer each line as it enters the tile to reduce the propagation time of the programming signals.

The programming lines are driven by shift registers in the periphery tiles. There is also an on-chip programmer that is implemented separate from the tiles and drives the shift registers. This programmer was implemented with a standard ASIC flow. For more details see Kuon's research [31].

4.2.3 Power-up Protection

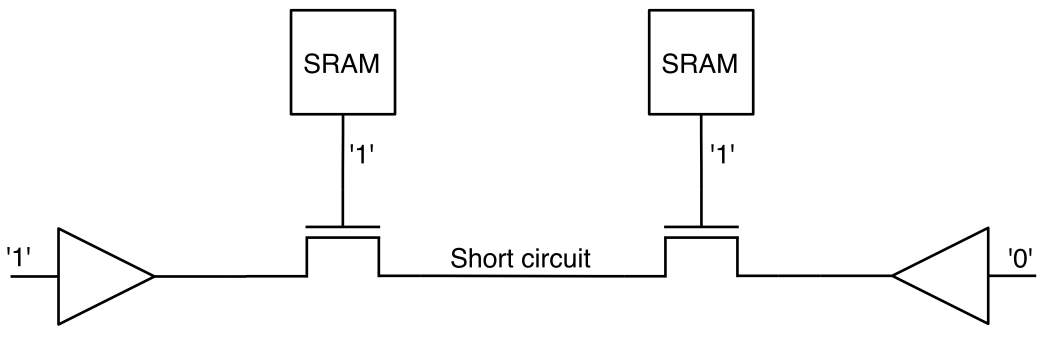
Another issue with the configuration SRAM in FPGAs is that when the power is turned on, each bit is in an unknown state. A situation could arise where two SRAM bits enable two routing switches to drive opposite logic values as shown in Figure 4.3a. This causes a short between power and ground and could damage the chip. To prevent this from occurring, we use the same approach as Chow et al. [7]. We added AND gates between every SRAM bit and pass transistor switch as shown in Figure 4.3b. The power-up protection net connects to one input of the AND gate so that when it is driven low, the switch is disabled regardless of the state of the SRAM bit. The power-up protection net is held low during power-up and driven high only after programming is complete and the SRAM bits are in a known state.

4.3 Metal Layer Allocation

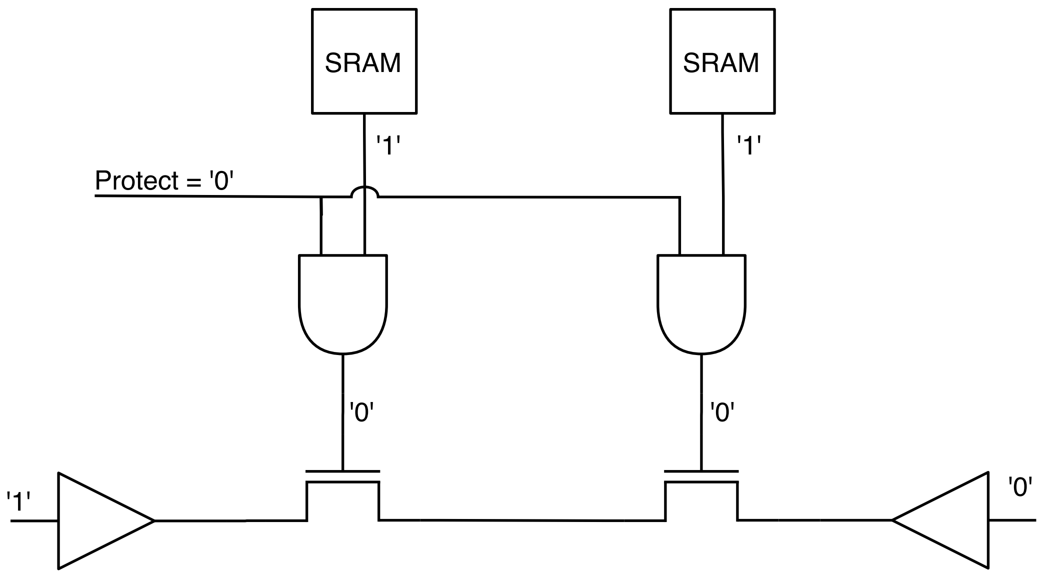
With only six metal layers in our fabrication process, we need to be careful how they are used. It has been shown that GILES is highly sensitive to the number of metal layers used for routing [31]. Four layers is often enough but if only three are used then the router increases the tile area significantly to complete the routing. The GILES router is not capable of routing wires around obstacles on each metal layer. Therefore, routing layers cannot be used for tasks other than routing. If four layers are used for routing then only two layers can be used for the cell internals, the power grid, and the clock and power-up protection networks. Also, the router does not obey the design rules that specify the minimum area of metal wires on each layer.

To solve these problems, Cadence's Virtuoso Chip Assembly Router [35] is used instead of the GILES router. Unlike the GILES router, this commercial router adheres to all design rules and routes around previously drawn metal on all layers. This allows us to draw the cell internals, the power grid, and the clock and power-up protection networks on any or all metal layers and the router will avoid these obstacles while using all metal layers.

However, we still must allocate metal layers for the cell internals, the power grid,



(a) Without power-up protection



(b) With power-up protection

Figure 4.3: Power-up protection prevents short circuit

Table 4.1: Metal layer allocation

Metal layer	Preferred direction	Purpose
1	Horizontal	Cells
2	Vertical	Cells (and clock between tiles)
3	Horizontal	Clock and power-up protection
4	Vertical	Tile ports
5	Horizontal	Power grid
6	Vertical	Power grid

and the clock and power-up protection networks so these wires do not overlap each other. Also, the Cadence router achieves better results when it routes wires in the same direction on a given metal layer. The purpose and preferred routing direction of each metal layer are listed in Table 4.1. The two bottom metal layers are used for the cell internals. The clock and power-up protection networks are drawn on metal three with the clock network switching to metal two between the tiles so it does not overlap the power-up protection network. The connections between tiles are called ports and are drawn on metal four. GILES is capable of using multiple layers for the tile ports but there is plenty of space in this tile of all the ports on one layer and not enough layers to use two. The two top metal layers are used for the power grid. We still try to minimize the use of layers two through five so that the router has as much space as possible for inter-cell routing.

The wires listed in Table 4.1 are not drawn by the router so they are not required to follow the router's preferred direction; however, doing so helps the router so the preferred direction is followed for the power grid and the clock and power-up protection networks. The clock and power-up protection networks do not follow the preferred direction between the tiles since the router only routes connections inside the tile.

4.4 Cell Layouts

The layout of each cell required for the POWELL architecture was drawn using Cadence's Virtuoso Layout Editor [30] and the TSMC 0.18 µm technology library [32], which was provided by CMC [33]. The cell layouts were based on designs provided by So [36]. Table 4.2 reports the layout area of each cell in grid squares, which have dimensions of 0.66 µm by 0.66 µm. Some cell areas differ slightly from the cell areas used to generate the area model in Section 3.1 because these cells were designed with different design rules. Also these cells only use one metal layer instead of two with the exception of the SRAM cell and the flip-flop with enable. In addition, the 1x inverter and flip-flop used here have different transistor implementations than before but the area model accurately predicts the area of these cells when using the correct transistor implementations.

We use the best combined transistor grouping that was determined in Section 3.2.3. That grouping used the buffer and pass transistor functional grouping and the 4x4 SRAM grouping. Two of the fifteen cells are only used in the periphery tiles. They are the 20-input multiplexer, which is used to connect the I/O pads, and the flip-flop with enable, which is used in the programming shift registers. All cells were verified with Diva DRC and LVS [37] and with simulations using the Virtuoso Analog Design Environment [38] in conjunction with Kuon.

A border was left around each cell so that any two cells can be placed next to each other and meet design rules as shown in Figure 3.1. This space is included in the sizes listed in Table 4.2. To determine how much space to leave, we find the minimum space required between the edge of the cell and any other drawing object. Then half of the minimum space is added to the cell edge. Since half of the minimum space is added to every cell, any two cells can be placed next to each other and will have the required full space between them. Usually the space is greater than the minimum because the cell also has to be aligned to the placement grid.

The cells have pins that specify where to connect the inter-cell routing. The GILES

Table 4.2: Cells used in POWELL and their sizes in 0.66 µm by 0.66 µm grid squares

Cell	Width	Height	Area
1x inverter (with level restorer)	7	5	35
2x inverter	5	6	30
4x inverter	6	6	36
4x buffer	7	6	42
SRAM (4x4 grouping)	20	24	480
2-input multiplexer	4	5	20
11-input multiplexer	11	10	110
12-input multiplexer	12	10	120
20-input multiplexer	14	18	252
LUT	11	18	198
Flip-flop	12	9	108
Flip-flop with enable	14	9	126
4x buffer and pass transistor grouping	7	6	42
Level restorer	6	5	30
AND gate	8	5	40

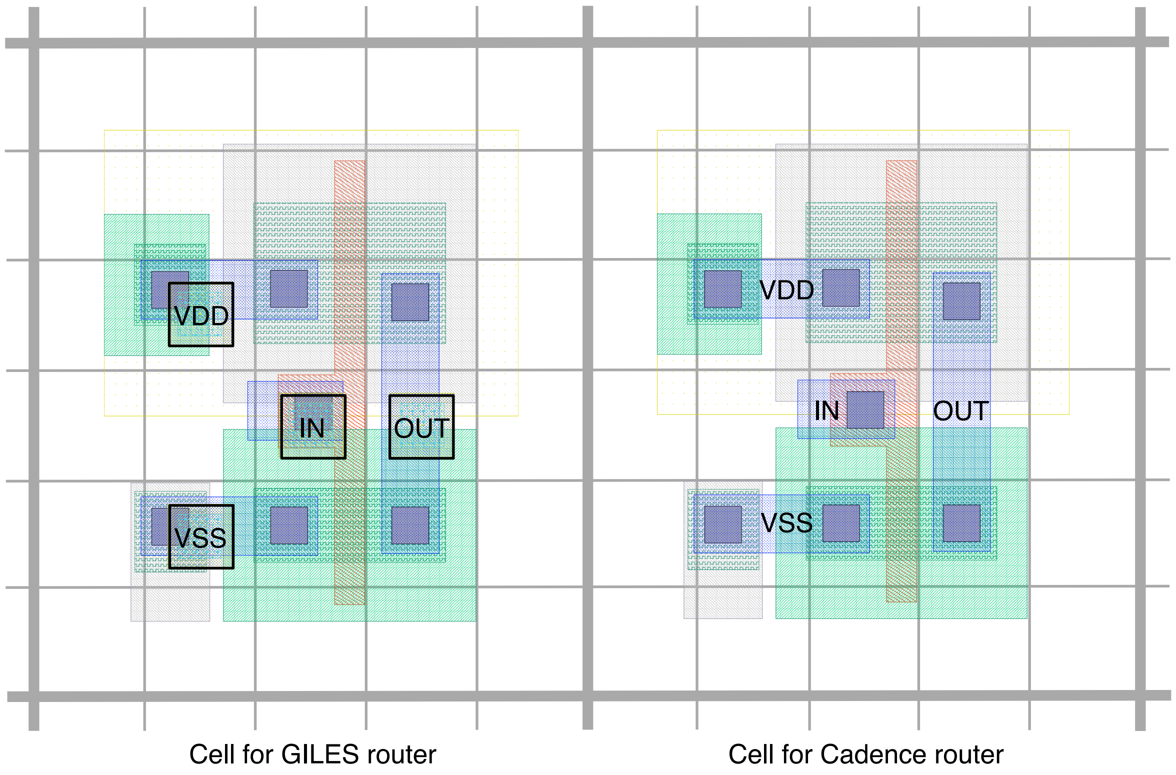


Figure 4.4: Cadence router does not require cell pins to be aligned to routing grid

router requires these cell pins to be aligned to the placement grid and attached to the first routing layer with vias as shown in Figure 4.4. Cadence's Virtuoso Chip Assembly Router does not have these requirements. It connects to pins on any metal layer and they do not need to be aligned to the grid. In some cells, it was difficult to connect all the power (VDD) and ground (VSS) connections to one pin for each net. Rather than use higher metal layers to connect the nets, multiple VDD and VSS pins are exposed so the router connects them automatically in the best way it sees fit.

4.5 Creating Tile Placements with GILES

Some modifications to GILES were required to obtain tile placements that use the new cell layouts. The cell-level netlist created by the netlist generator contains the dimensions

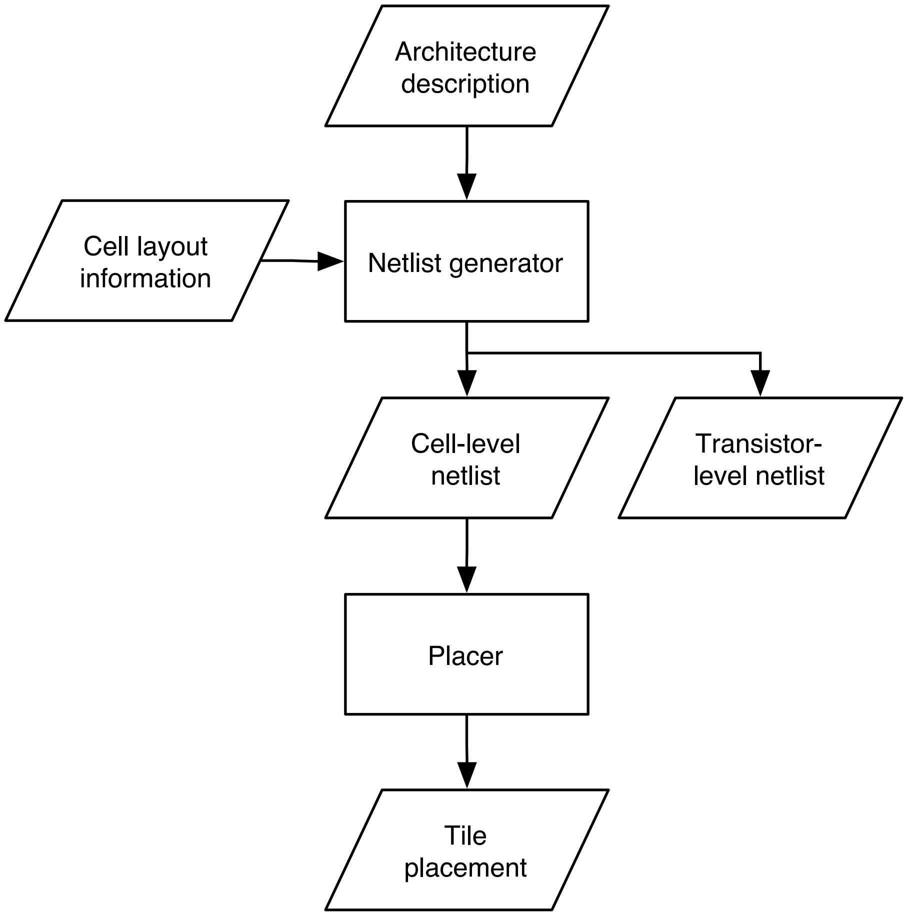


Figure 4.5: Revised GILES flow

and pin positions of each cell. However, the dimensions are estimated by the area model and the pin positions are arbitrarily spread out across the cell. The placer needs the real dimensions and pin positions from the cell layouts to create valid placements and to properly minimize wirelength. Therefore, the cell layout information must be given to the netlist generator so it passes the correct cell information to the placer. The original GILES flow from Figure 2.12 was modified so the netlist generator has a new input that specifies the cell layout information. The new flow is shown in Figure 4.5. The cell information input specifies the dimensions and pin positions of the cells. The netlist generator uses this information when creating the netlist. The cell information file used for POWELL is in Appendix C.

Another change to the flow is that we no longer run the GILES router since we have decided to employ Cadence's Virtuoso Chip Assembly Router [35]. The output of the GILES flow is now the tile placement that needs to be imported into the Virtuoso custom design platform. Using this flow, the size of the main tile of POWELL is 168 by 202 grid squares or 110.88 µm by 133.32 µm. See Kuon's research [31] for a description of how the periphery tile placements are generated.

4.6 Integrating with the Virtuoso Custom Design Platform

Each tile is imported into Cadence's Virtuoso custom design platform [34] after placement. The Virtuoso Chip Assembly Router [35] is used for routing. The Virtuoso Layout Editor [30] is used for adding the power grid, clock H-tree, power-up protection network, I/O pads, and the programmer. We automate all the time consuming tasks using the Cadence scripting language called SKILL [39]. The GILES placer was modified to emit SKILL files for each tile it generates. Each SKILL file contains six functions that are executed in Virtuoso sequentially. These functions import the tile placement, draw the power grid, clock H-tree, and power-up protection network, and tile the array. Each SKILL file has all the information needed to generate a routed array of tiles and each function takes several arguments to set configurable parameters. In addition, there is a separate SKILL file that helps route the connections between the array and the I/O pads and programmer. Each of these functions are discussed in the following sections.

4.6.1 Tile Placements

The first stage of integrating the design with Virtuoso is to import the placement of each unique tile. Each tile has a SKILL function called GilesCreateTile() that was generated by the GILES placer. Calling one of these functions in Virtuoso draws the tile in a new layout. The function creates instances of each cell in the tile and positions it on the placement grid. The function also describes the connections between the pins of every cell, which will later be wired together by the router.

The SKILL function also creates connections to port locations on the sides of each tile. It treats a port like a cell with one pin. The cell library must contain port cells that are merely a patch of metal and a single pin. For this design, metal layer four is used for tile ports but this is easily changed by modifying the port cells. There are separate port cells for each tile edge because the metal in each port cell must extend in the correct direction to reach the adjacent tile. The port cells extend far enough outside the tile so we can leave a gap between tiles for drawing the clock and power-up protection networks.

The placement of the main tile for POWELL is shown in Figure 4.6. The layout of each cell is shown and the spacing between them. The largest cells are the 4x4 SRAM cells. Here it can be observed how grouping cells results in less wasted space between cells. Around the border are the tile ports. They extend in each direction towards the tile they will connect too. A thin line runs around the tile and through the ports. This is the routing border. The router must keep all wires inside this box so they do not overlap other tiles or the clock and power-up protection networks that are drawn between the tiles.

4.6.2 Power Grid

The power grid is drawn before routing the tile connections so that the router will automatically connect the power grid to all the cells in the tile. The power gird used for the main tile is shown in Figure 4.7. It consists of alternating power (VDD) and ground (VSS) vertical stripes on metal layer six and a single pair of VDD and VSS horizontal stripes on metal layer five in each tile. The horizontal stripes allow current to be shunted between vertical stripes quickly in the case of a large power spike on one vertical stripe. Using only a single pair of horizontal stripes per tile allows the router to use most of

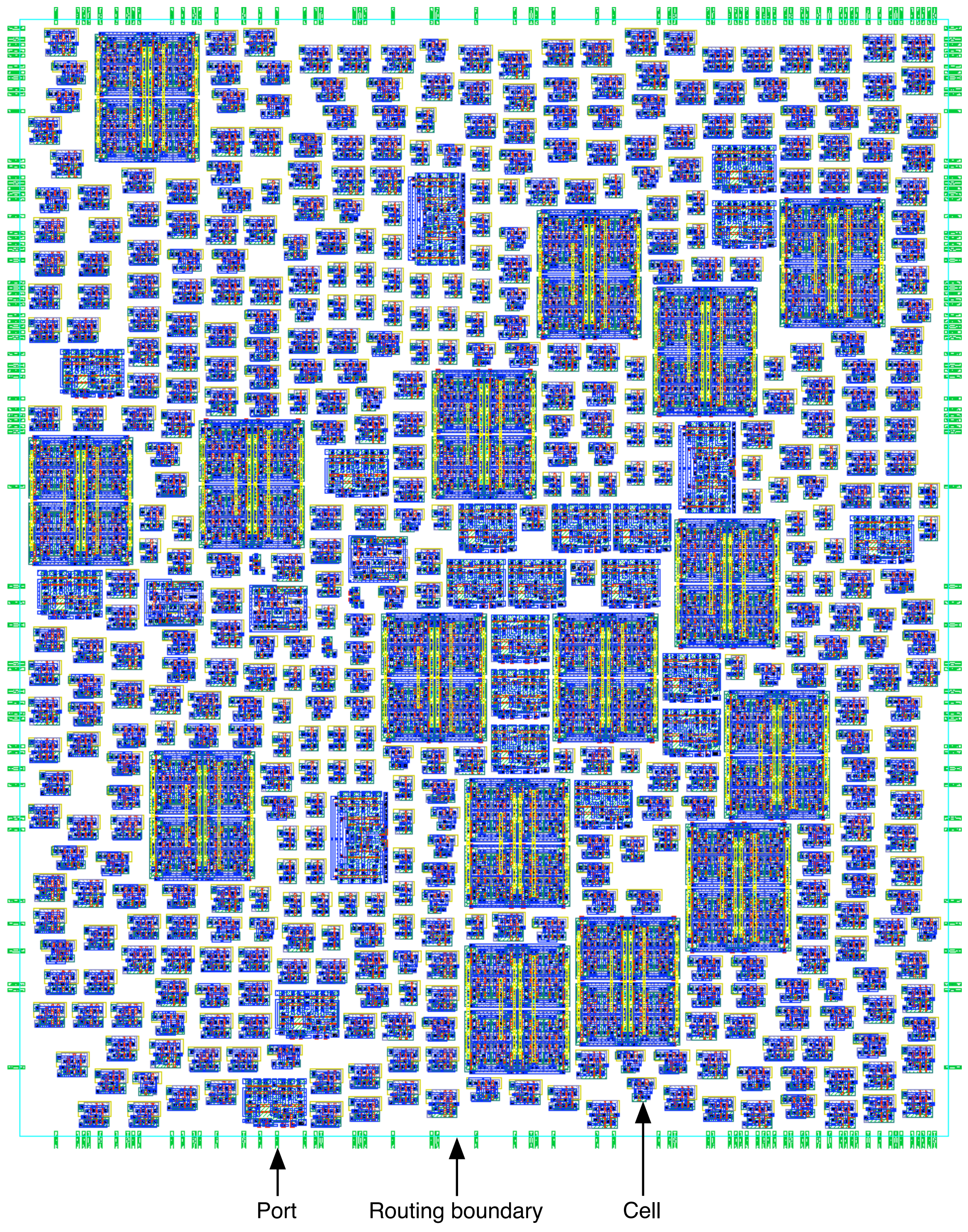


Figure 4.6: Placement of main tile

metal layer five for routing. The vertical and horizontal stripes are connected where they cross with vias. The power grids for the periphery tiles are the same except there are fewer vertical stripes to accommodate the smaller tile dimensions.

An issue arose when using the Virtuoso Chip Assembly Router. Very few connections were made between the power grid on metal five and six and the cells on metal one and two. The router preferred to make connections between the cells on the lower metal layers and brought up only a few connections to the power grid. The router has a power routing feature but our custom cells are not compatible with this feature so these power nets were routed like any other net. The typical behaviour of the router was to make only five connections per tile to the power grid. Kuon determined the maximum current draw of the main tile is 63.6 mA if all cells switch at the same time. Assuming only a quarter of the cells switch simultaneously, the current draw is 15.9 mA. With only five connections to the power grid, each connection must supply 3.18 mA. However, the maximum current each connection can supply is only 0.28 mA for our fabrication process. Therefore, at least 57 connections are needed to supply 15.9 mA.

Our solution to the power distribution problem is to plan a power grid that forces the router to make more connections to it. This was achieved by dividing the tile into regions using the following procedure. The original power grid of Figure 4.7 is divided into four columns corresponding to the four pairs of vertical stripes. Each stripe is then divided into separate patches of metal as shown in Figure 4.8. Each patch of metal is assigned to a unique VDD or VSS net. A single region contains one VDD and one VSS net. The cells located in that region are modified to connect to the VDD and VSS nets for the region. Then the router connects the VDD and VSS nets in each region and brings at least one connection up to the power grid per region. After routing, the nets of the power grid are connected together to create the full grid structure of Figure 4.7.

In Figure 4.8 as in the main tile of POWELL, there are four pairs of vertical stripes that are divided into twenty sections each. This gives a total of eighty power regions and

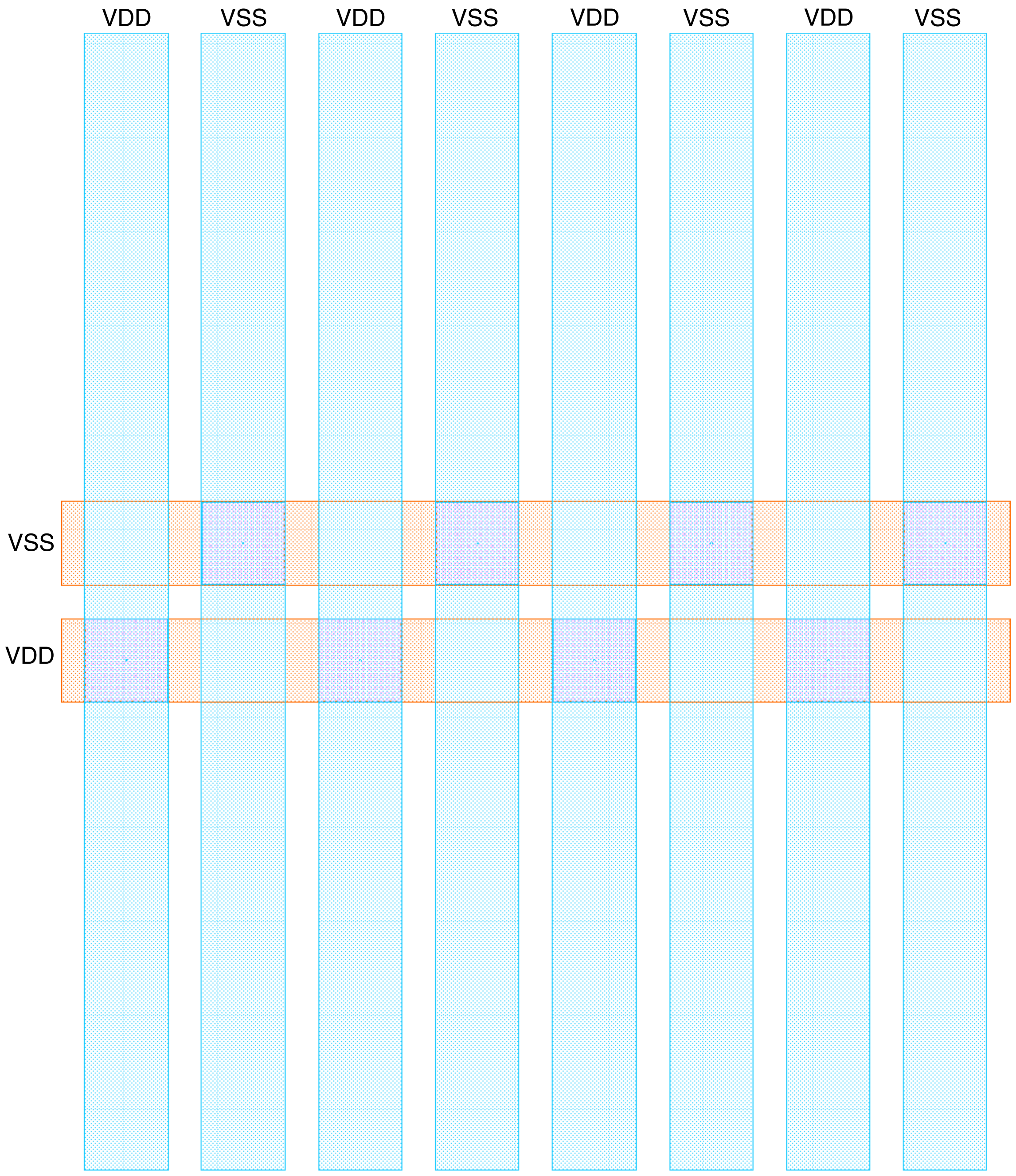


Figure 4.7: Power grid for main tile

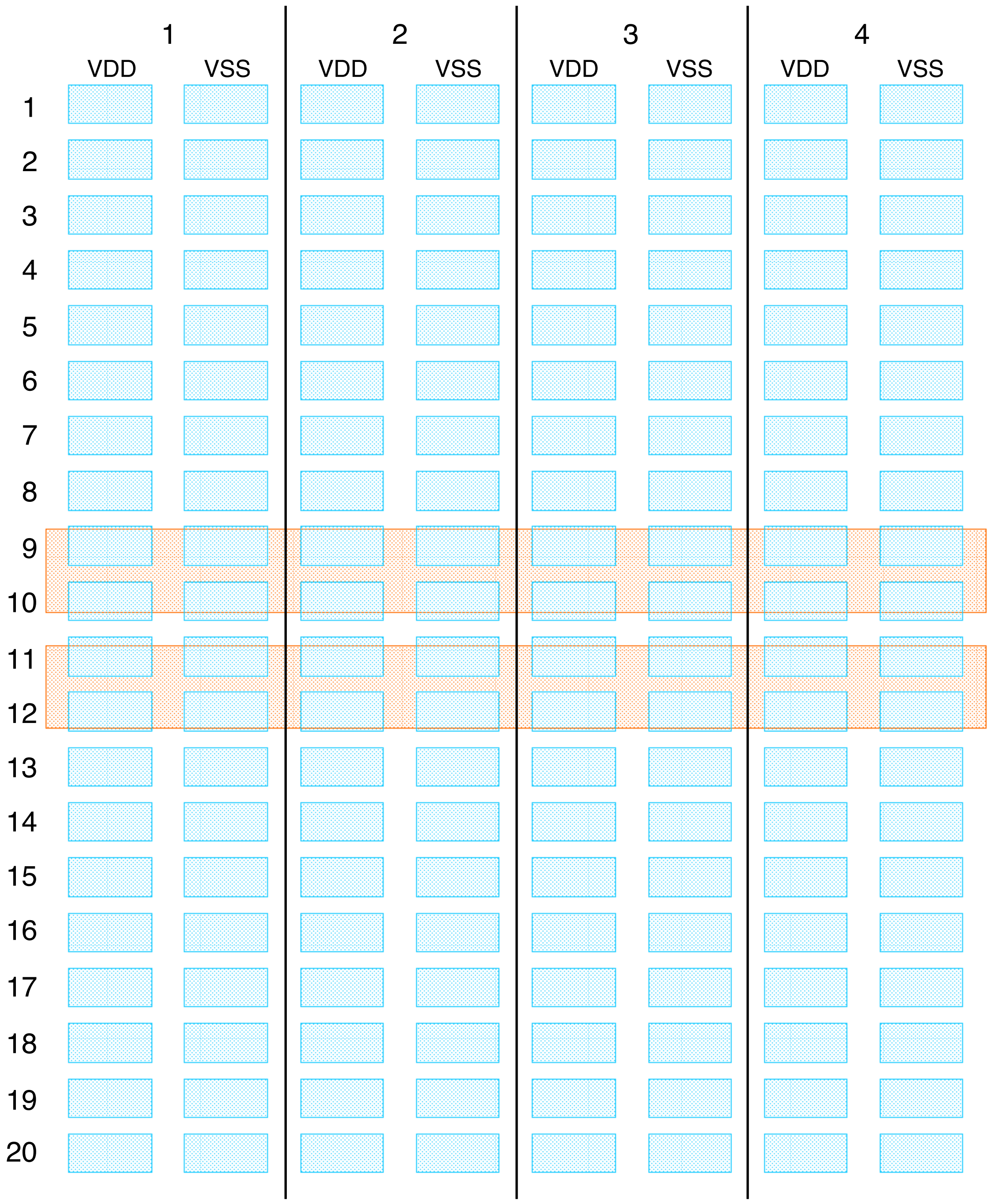


Figure 4.8: Power grid regions

guarantees a minimum of eighty connections to the power grid. Having too many regions complicates the router's task but too few regions will slow down the FPGA when the current is restricted. For the purpose of this chip, we were not concerned about speed but in the future, we would like to verify the number of regions required to achieve the best performance. The power grid regions are generated automatically by parameterized SKILL scripts so it is possible to easily create power grids with more or less regions. For example, the periphery tiles are divided into fewer regions because of their smaller dimensions. The area of each region is similar to those in the main tile.

4.6.3 Clock Tree

Before routing the connections in the tile, we consider how the clock network connects into the tile. To minimize the skew of the clock between tiles, an H-tree is used as shown in Figure 4.9. Metal layer three is used for the connections into the tile and metal layer two is used for the connections between the tiles. Metal layer two is used to avoid crossing the power-up protection network that is described in Section 4.6.4. The periphery tiles are shown but they do not connect to the clock tree because some do not need a clock and the others use a separate programming clock. The programming circuitry was designed by Kuon [31]. It consists of shift registers so the programming clock is routed automatically through the tiles in the opposite direction of the shifting to avoid hold time violations. The programming clock frequency is set to be slow enough to avoid setup time violations.

The clock tree of Figure 4.9 is driven by a large clock driver input pad. Because POWELL only contains 192 flip-flops in the main eight by eight tile array, Kuon's simulations showed that this clock driver is sufficient for driving all the flip-flops without additional buffering. However, if larger FPGAs are created, clock buffers can be inserted between the tiles.

The clock tree connects to the centre of each tile. A clock pin is drawn on metal three

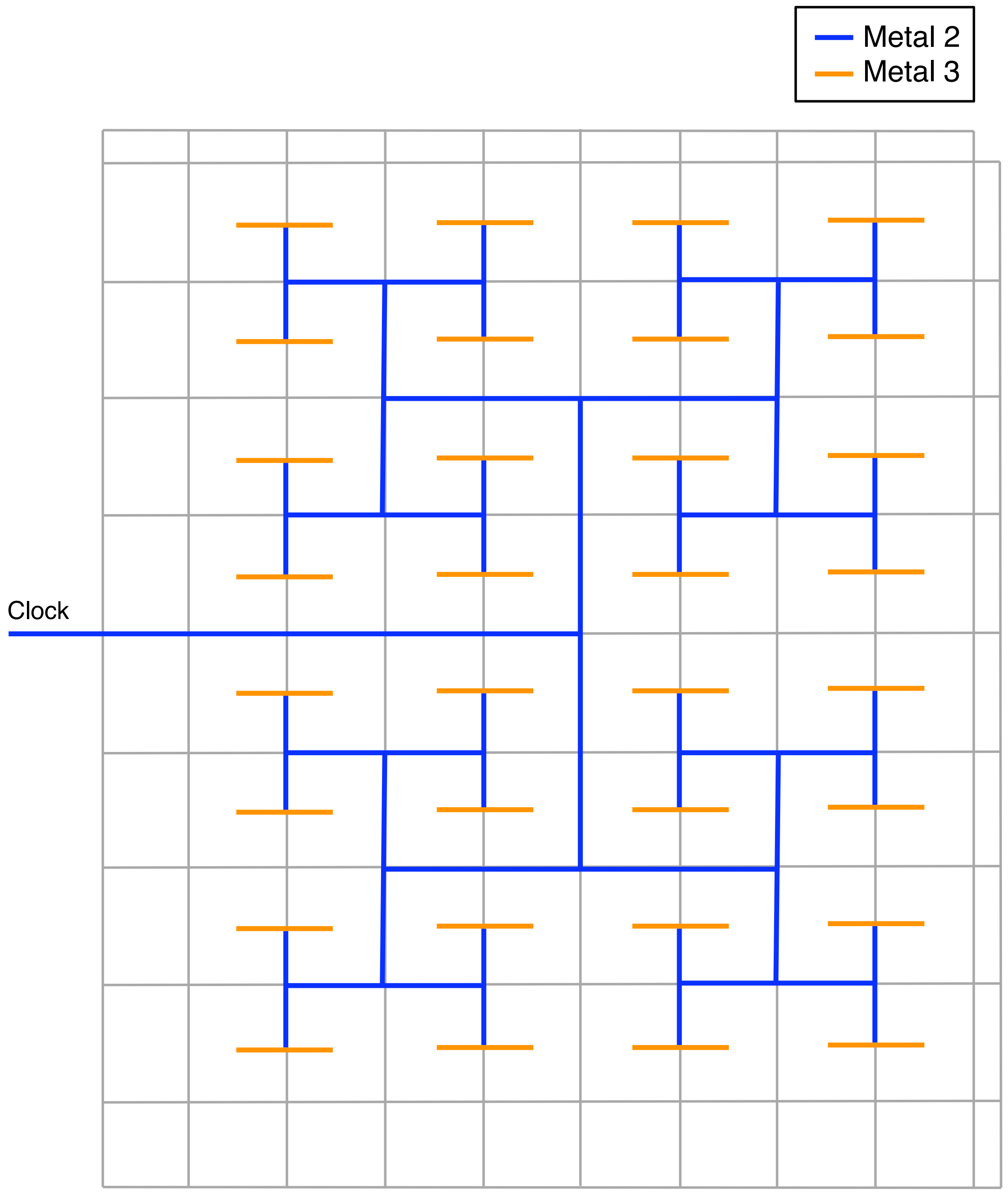


Figure 4.9: Clock H-tree

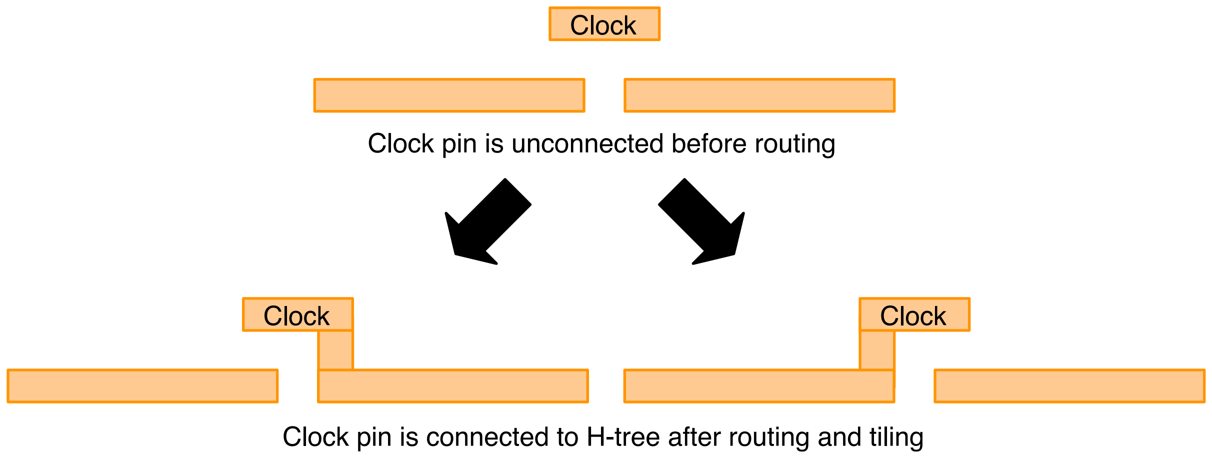

Clock
Clock pin is unconnected before routing
Clock
Clock
Clock pin is connected to H-tree after routing and tiling

Figure 4.10: Procedure for connecting clock pin to H-tree

and the router connects the flip-flops within the tile to this pin. The connection from the edge of the tile to the centre of the tile must be drawn before routing so it does not cross any other connections. However, half the tiles require the connection to come from the left of the tile and the other half from the right. To keep all tiles identical so that only one main tile needs to be routed, wires are drawn to both sides of the tile as shown in Figure 4.10. The clock pin and the wires to the edge of the tile are drawn before routing. After the design is routed and the array is tiled, the clock pin in each tile is connected to the H-tree through one of the two wires depending on where the tile is in the array.

4.6.4 Power-up Protection

Another concern that must be addressed before routing occurs is the design of the power-up protection network. As described in Section 4.2.3, the power-up protection signal connects to AND gates that prevent switches in the routing tracks from shorting power to ground before programming is complete. There are 111 AND gates in each main tile so this net has a much larger fan-out than the clock network. We use a large clock driver for its input pad but the speed of this net is not critical so additional buffering is not needed. Also, the skew across tiles is not a concern so an H-tree is not required.

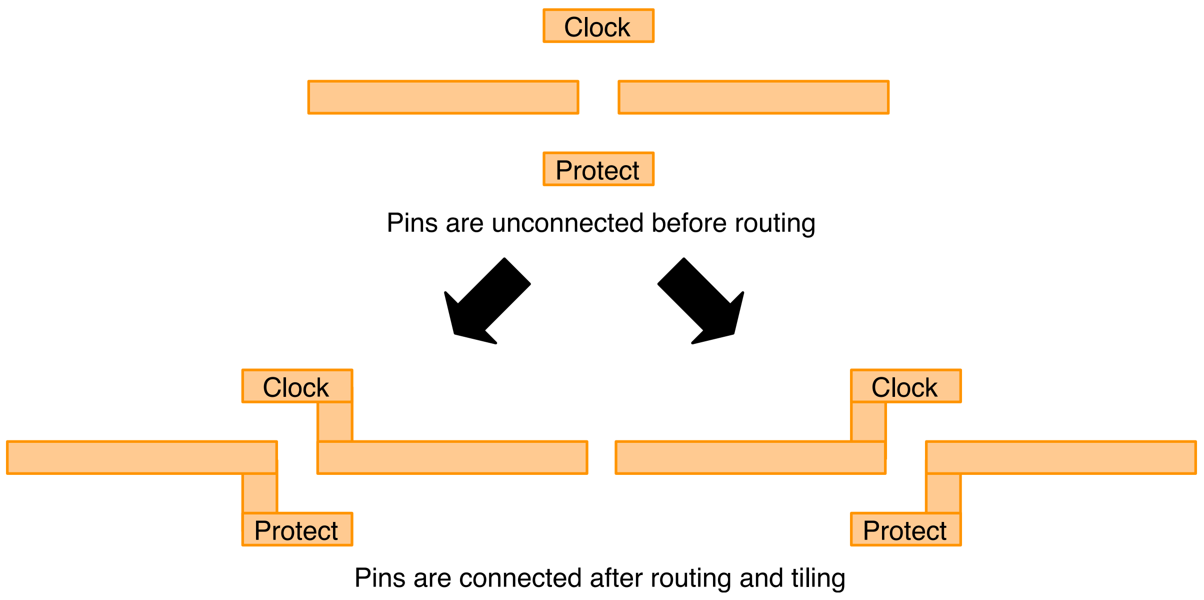

Clock
Protect
Pins are unconnected before routing
Clock
Protect
Clock
Protect
Pins are connected after routing and tiling

Figure 4.11: Power-up protection pin uses unconnected clock wire

To bring the power-up protection net out of the tile, a pin is created opposite the clock pin in the centre of the tile. The router connects this pin to all the AND gates in the tile. After routing and tiling of the array, the power-up protection pin is connected to the wire not used by the clock network as shown in Figure 4.11. Since skew is tolerable on this signal, the tiles are connected as illustrated in Figure 4.12. The clock network between the tiles is drawn on metal two so it crosses under the power-up protection network.

4.6.5 Routing

Routing is performed on each unique tile using Cadence's Virtuoso Chip Assembly Router [35]. The router uses all layers of metal and alternates the preferred wire direction on every other layer as listed in Table 4.1. It obeys the route boundary that was drawn during the placement import phase of Section 4.6.1. For some of the eight tiles, the router encountered congestion and did not find a valid route for all nets. However, these errors were minor and were easily fixed by hand. In total there were 26 DRC errors fixed manually including those in periphery tiles. After routing, the power grid

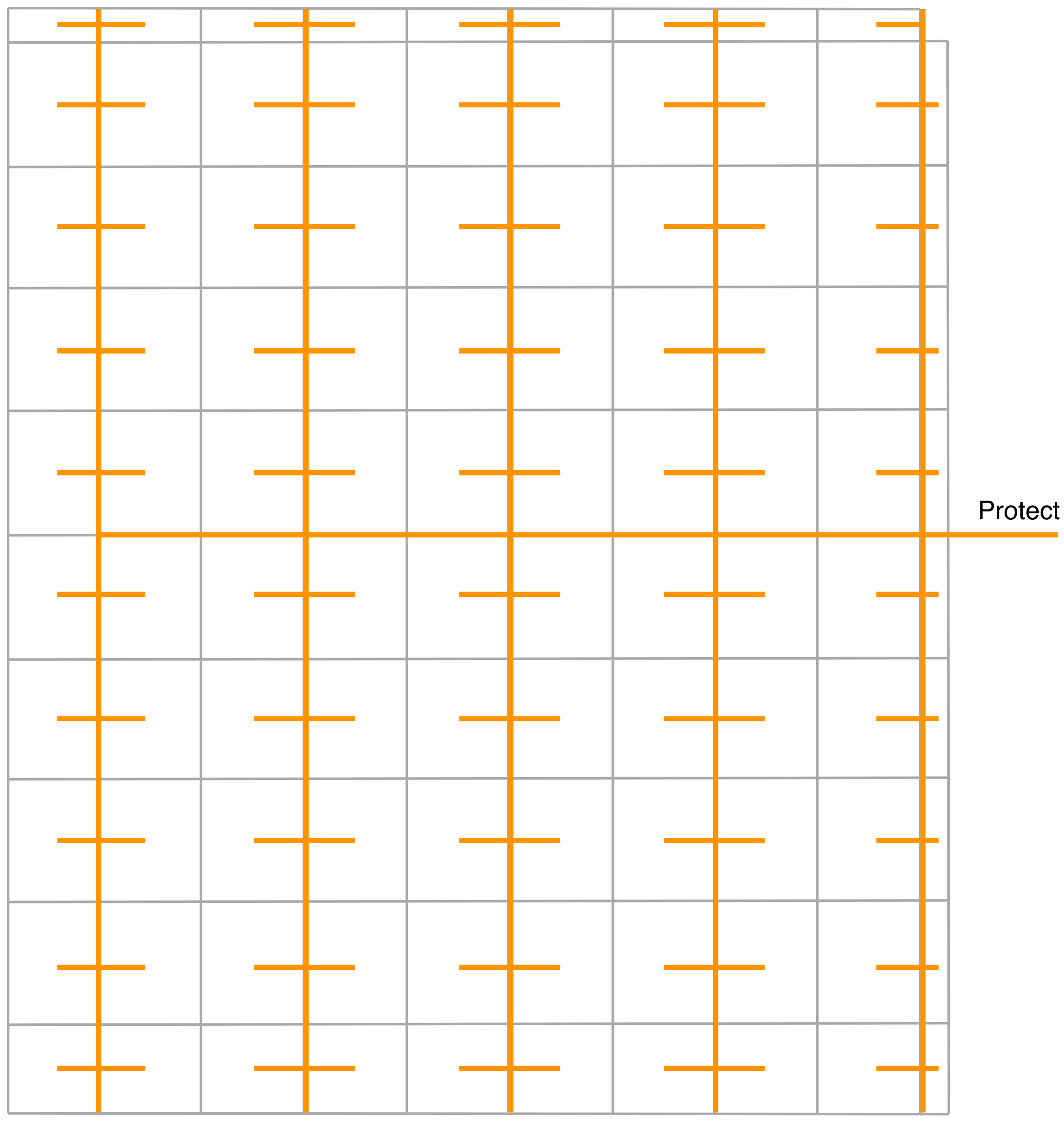


Figure 4.12: Connections of power-up protection network between tiles

regions are connected by running a SKILL script. The fully routed main tile is shown in Figure 4.13.

4.6.6 Tiling the Array

In an empty layout cell view, a parameterized SKILL script creates the array of tiles. For POWELL, the main tile array is eight by eight. These are surrounded by periphery tiles. As shown in Figure 4.14, the tiles are abutted to each other and the ports bridge the gap between them. The gap is left for the clock and power-up protection networks, which are now drawn by another SKILL function. At the same time, the clock and power-up protection pins at the centre of each tile are connected to the wires leading to the opposite sides of the tile as shown in Figure 4.11. This connects the clock and power-up protection nets inside each tile to the global networks.

4.6.7 I/O Pads and Programmer

The final details required to complete the FPGA include the I/O pads and the programmer. The programmer was designed using standard cells and typical ASIC tools by Kuon [31]. Kuon also created the I/O ring that consists of 84 pins including power, clock, power-up protection, and general purpose I/O. Both the I/O pads and the programmer were added to the design manually because it was easy to do so.

To enable automatic routing between the tile array, the programmer, and the I/O pads, a SKILL script labels the ports on the periphery tiles to match the manually labelled signals on the programmer and I/O pads. This greatly simplified the task of identifying the signals for the 64 I/Os since each I/O consists of input, output, and output enable signals and the GILES placer is free to move them around within each periphery tile. The SKILL script also creates power and ground rings around the array of tiles and connects the power grid to the rings.

The connections between the tile array, the programmer, and the I/O pads are routed

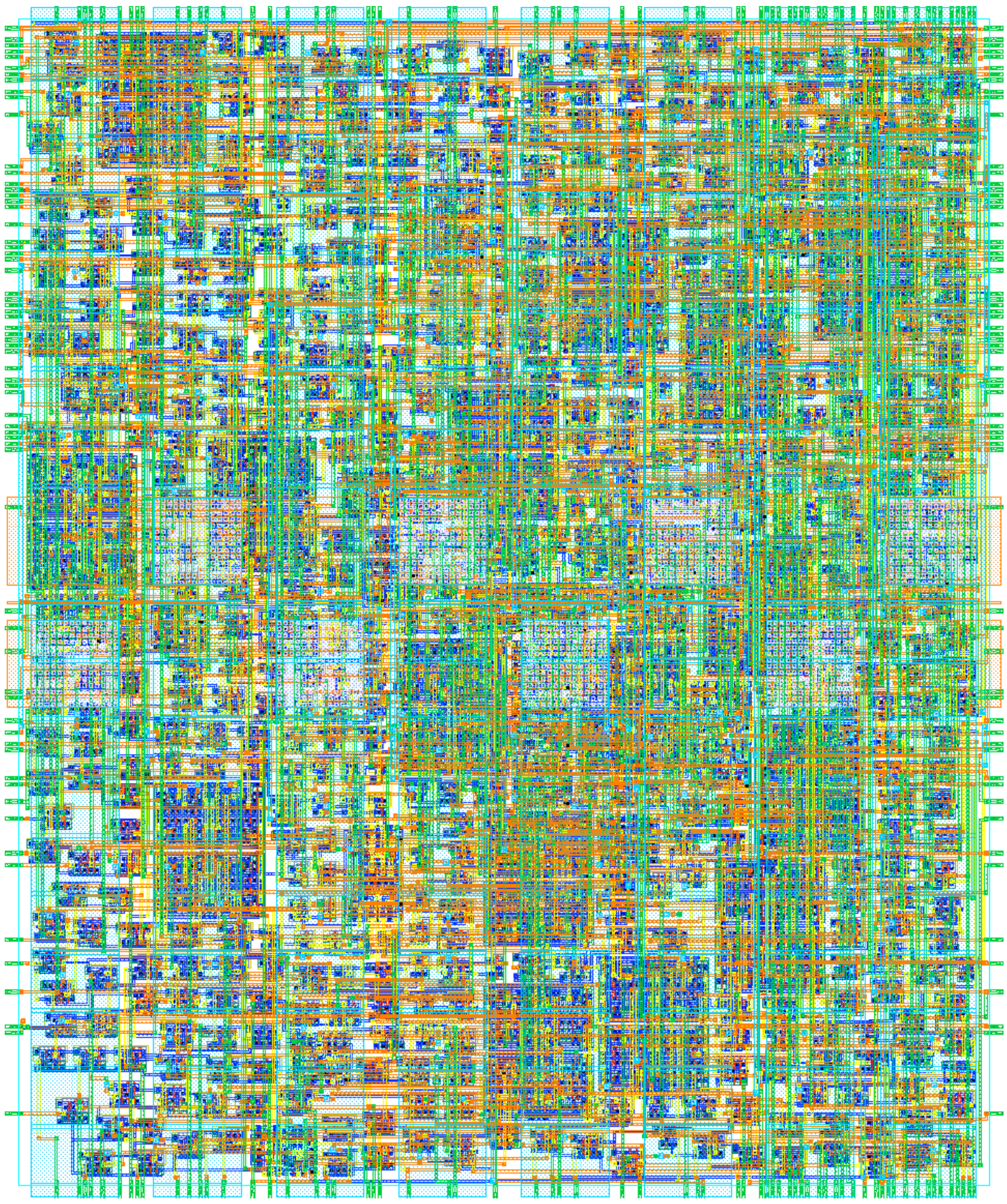


Figure 4.13: Fully routed main tile

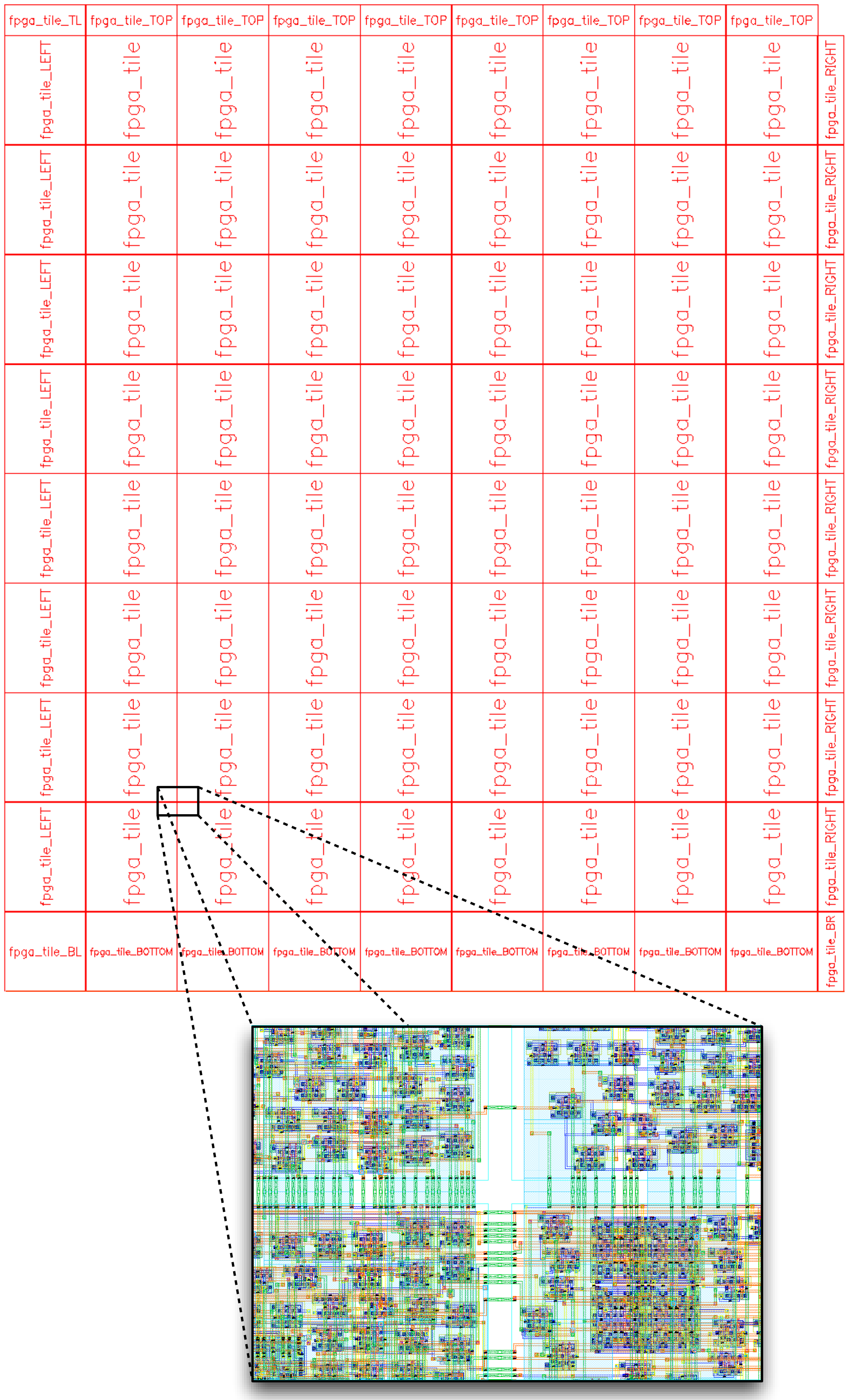


Figure 4.14: Array of tiles

automatically using the Virtuoso Chip Assembly Router except for four critical nets: power, ground, clock, and power-up protection. All four nets are critically important for the functionality of the chip. The clock net was routed carefully to the tile array and the programmer to reduce the clock skew. There are four power and four ground I/O pads that are connected to the power and ground rings in four different places. These connections are drawn 28 μm wide to carry large amounts of current.

4.7 Verification

The design was verified for both design rules and functionality in conjunction with Kuon [31]. CMC does not provide the layouts of the standard cells that were used in the programmer or the layouts of the I/O pads so the design was submitted to CMC to check for design rule violations using Calibre DRC [40]. Besides the DRC errors caused by the router described in Section 4.6.5, the only other errors were polysilicon density and antenna errors. To fix the polysilicon density error, polysilicon was added using Diva DRC's [37] fill feature.

There were 218 antenna errors caused by long wires that could build charge during fabrication and damage transistor gates. These errors were fixed by adding a small diode between the problematic wire and the substrate to regulate the charge on the wire as shown in Figure 2.9. Because the FPGA is composed of identical tiles, fixing one error fixed identical errors in copies of the same tile. In one case, adding a single diode to the main tile fixed 168 errors. In total, only eleven diodes were added and it was not difficult to find space for these between the existing cells.

Verification of functionality was done by Kuon [31] using Calibre LVS [40] and extensive simulations. To simulate the FPGA, Kuon enhanced VPR to generate programming bitstreams for test circuits. He simulated the programming of each bitstream and each resulting test circuit. Initially, simulation was performed with a Verilog netlist of POW-

ELL. Later, the transistor implementation was simulated with Synopsys NanoSim [41]. Finally, NanoSim was used to simulate the extracted schematic to verify the layout.

The final layout for POWELL is shown in Figure 4.15. The top two metal layers have been removed for clarity. The tile array and the I/O pads are clearly visible. The programmer is in the bottom-left corner of the chip. The large red rectangles on either side of the tile array are areas of polysilicon to meet the density requirements.

POWELL was submitted to CMC on March 24, 2004 with a bonding diagram for an 84 pin grid array (PGA) package. Fabrication and packaging has been completed. We received five packaged chips on November 22, 2004. Testing is currently underway using the TH1000 test fixture and other equipment available at the University of Toronto.

4.8 Design Time

The goal of our automatic design system is to reduce the time required to create FPGAs. Creating a commercial FPGA using custom design techniques requires at least 50 person-years. Using our automated approach, we created an FPGA in approximately eight person-months. The breakdown of the time required for our FPGA is listed in Table 4.3.

It took two graduate students four months to complete the design not including the time required to develop the tools. Admittedly, the design of commercial FPGAs is significantly more complex; however, this still represents a huge time savings compared to custom design. With additions to the automatic design system, more complex FPGAs could be created while maintaining significant reductions in design time.

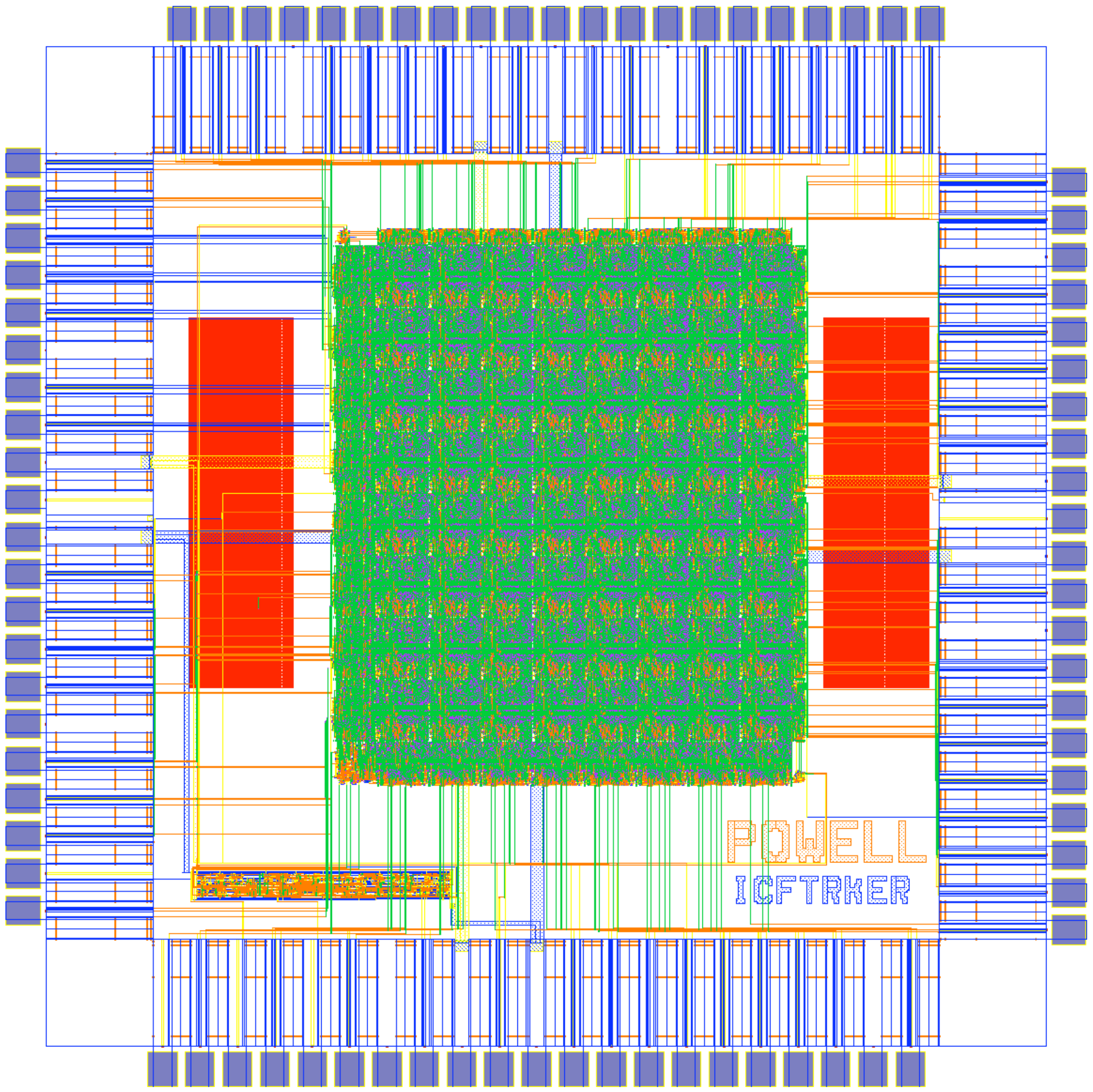


Figure 4.15: POWELL layout

Table 4.3: Breakdown of time required to design POWELL

Design task	Time required (person-weeks)
Architecture exploration	2
Circuit design	8
Cell layout	6
Tile layout	0.5
Programming infrastructure	2
Power grid	1
Clock tree	1
Fixing design rule violations	0.5
I/O pads	1
Verification	12
Total	34

Chapter 5

Conclusions

The two goals of this work were to improve the area results of an automated layout system for FPGAs and to fabricate the first automated FPGA from architectural specification to layout. To achieve the first goal, the accuracy of the area model was improved. Using this improved area model, the set of cells used by the placer was evaluated and better cells were sought. Experiments showed that grouping larger numbers of transistors into cells was beneficial to the final routed area. The best grouping used a new 4x4 SRAM cell and a buffer and pass transistor cell. This balanced the tradeoff of reduced area without increasing wirelength and manual layout effort.

Using this work on transistor grouping, Kuon [31] found that GILES produces an FPGA tile that is just 36% larger than a commercial Xilinx Virtex-E. This is much better than a standard cell version of the Virtex-E, which is 102% larger than the custom layout by Xilinx. With the addition of extra metal layers, the automatically-produced tile is only 13% larger.

The second goal of this work was to manufacture an FPGA created with these automatic tools. The GILES tool [5] was extensively modified and extended to reach this goal. Cadence SKILL scripts are now generated to automate most of the tasks to take the tile placement and obtain a complete FPGA layout ready for fabrication. This includes unique approaches to the design of the power grid and clock network. The FPGA, called POWELL, was designed in only eight person-months not including the time required to develop the automatic design system. POWELL was fabricated through CMC using a TSMC 0.18 μm process.

5.1 Contributions

The contributions of this research are the following:

1. A model for estimating the layout area of small groups of transistors.
2. Improved area results of an automated FPGA design tool by grouping transistors into larger cells.
3. Extensions to the automated design tool to enable generating complete FPGAs.
4. The first automatically generated FPGA to be created from an architectural specification.
5. Important steps in proving the viability of an automated approach to FPGA design, which is still done manually in industry.

5.2 Future Work

This work would benefit from a continued effort to improve the area results. Some possible avenues for improvement involve creating a smarter netlist generator that would map a transistor netlist to a cell library and automatically optimize the specific architecture to the most efficient groupings. Another approach is to make the placer able to place cells that do not have space around them. It would need knowledge of the cell internals and the design rules of the process to avoid any violations. Alternatively, a layout compactor could be run on the tile after placement to reduce the space left between cells.

With improved area results, it may be possible to surpass the area efficiency of custom designers. However, to replace custom designers, automatically generated FPGAs must also be competitive in terms of speed and power. In the future, the speed and power of automatic designs need to be compared to custom designs. Then the automatic layout system can be improved to obtain better results in these categories.

GILES also needs updating to handle modern FPGA architectures. This involves updating the architecture description language and the architecture generator to support new interconnect structures, logic block designs, and heterogeneous structures such as hard multipliers and memories. The automatic layout tools will also have to be tested with smaller fabrication processes.

Appendix A

Experimental Results for Transistor Groupings

Table A.1: Tile area before routing for functional groupings

Architecture	Tile area before routing (grid squares)					
	No grouping	Functional grouping				
		1	2	3	4	5
1	26832	24585	25418	24220	24167	23925
2	49896	45990	47952	45288	46632	44880
3	70932	68352	69048	64944	68340	64904
4	96398	91168	91956	89060	89082	85544
5	124942	120574	125874	114437	116960	113321
6	150100	145668	141804	137940	142835	148176
7	170982	165946	161579	157488	181008	168520
8	195960	184260	187000	189288	181746	189161
9	217413	206912	203432	200994	206480	198856
10	239259	231632	226320	219252	238620	214700

Table A.2: Tile area after routing for functional groupings

Architecture	Tile area after routing (grid squares)					
	No grouping	Functional grouping				
		1	2	3	4	5
1	26832	24585	26718	24220	25431	23925
2	49896	45990	47952	45288	47066	44880
3	70932	69133	69048	64944	69647	64904
4	98587	91168	91956	89060	90890	85544
5	126360	123370	126252	115434	119712	114000
6	154014	147600	141804	137940	142835	148176
7	186192	173382	167678	158304	184851	173494
8	201312	199230	193536	194084	192126	199728
9	228206	218183	218094	201894	219897	200175
10	254828	245310	233532	225400	247995	217490

Table A.3: Tile whitespace before routing for functional groupings

Architecture	Tile whitespace before routing (grid squares)					
	No grouping	Functional grouping				
		1	2	3	4	5
1	3920	3337	3710	3772	2999	4017
2	5949	6096	6420	6003	6423	6918
3	7172	7995	7351	7801	7813	8931
4	10838	10612	10896	11996	10074	10532
5	12478	15640	19590	13358	13241	14807
6	15410	17908	14458	16768	17109	28624
7	17268	21769	16609	19400	39098	33492
8	20758	19530	20928	30300	19040	33539
9	22026	23045	18131	23526	24851	25060
10	28614	35437	22525	28082	38650	27355

Table A.4: Tile area before routing for 2x2 SRAM grouping

Architecture	Tile area before routing (grid squares)		
	No grouping	2x2 SRAM grouping	
		Without inter-cell SRAM bit swapping	With inter-cell SRAM bit swapping
1	26832	24436	24178
2	49896	46209	45980
3	70932	66045	64498
4	96398	88755	89012
5	124942	116955	116795
6	150100	138600	144375
7	170982	158950	159032
8	195960	179350	181900
9	217413	211896	198907
10	239259	214700	213834

Table A.5: Tile area after routing for 2x2 SRAM grouping

Architecture	Tile area after routing (grid squares)		
	No grouping	2x2 SRAM grouping	
		Without inter-cell SRAM bit swapping	With inter-cell SRAM bit swapping
1	26832	24436	24178
2	49896	46209	45980
3	70932	66045	64498
4	98587	88755	91104
5	126360	121446	116795
6	154014	152358	145899
7	186192	187220	174000
8	201312	207935	187920
9	228206	266760	205205
10	254828	282220	216618

Table A.6: Tile wirelength before routing for 2x2 SRAM grouping

Architecture	Tile wirelength before routing (grid squares)		
	No grouping	2x2 SRAM grouping	
		Without inter-cell SRAM bit swapping	With inter-cell SRAM bit swapping
1	46605	50046	46989
2	95662	109386	98383
3	151285	175348	155364
4	202051	233250	213390
5	281464	325858	294573
6	353384	421068	367707
7	423721	499892	440330
8	477275	584354	496504
9	549871	680144	551736
10	600678	745113	615258

Table A.7: Programming wirelength before routing for 2x2 SRAM grouping

Architecture	Programming wirelength before routing (grid squares)		
	No grouping	2x2 SRAM grouping	
		Without inter-cell SRAM bit swapping	With inter-cell SRAM bit swapping
1	9173	7566	7638
2	17346	14644	14607
3	26516	21221	21562
4	35394	28144	28458
5	48282	37930	37004
6	59761	45091	46531
7	70140	52513	53614
8	79443	62196	61588
9	91726	71560	69225
10	100623	74782	73814

Table A.8: SRAM output wirelength before routing for 2x2 SRAM grouping

Architecture	SRAM output wirelength before routing (grid squares)		
	No grouping	2x2 SRAM grouping	
		Without inter-cell SRAM bit swapping	With inter-cell SRAM bit swapping
1	6288	11740	8764
2	11005	23972	15585
3	17134	44555	23783
4	22320	53873	33317
5	31586	78141	42132
6	36987	100227	51888
7	46876	128459	63014
8	46841	150945	65120
9	55157	172554	71969
10	58670	191742	79524

Table A.9: Tile area before routing for 4x4 SRAM grouping

Architecture	Tile area before routing (grid squares)						
	Multiplier for number of pin swap moves						
	1x	2x	3x	4x	5x	6x	7x
1	22464	22833	23188	22801	23146	22680	23595
2	43281	43554	44070	43979	43472	43560	43884
3	62275	62464	62178	62618	62976	62712	61997
4	82940	84854	84084	84208	84099	84348	83142
5	108924	109188	112896	110745	110889	109512	110400
6	130285	129596	129792	129591	131930	129210	130592
7	148740	149040	149144	153543	150220	149480	163625
8	168597	170280	171768	179780	169644	174096	169122
9	189125	190920	197400	187812	188958	189996	190080
10	205660	203472	206974	210672	206150	207792	209196

Table A.10: Tile area after routing for 4x4 SRAM grouping

Architecture	Tile area after routing (grid squares)						
	Multiplier for number of pin swap moves						
	1x	2x	3x	4x	5x	6x	7x
1	22464	22833	23188	22801	23146	22984	23595
2	43281	43554	44070	43979	43472	43560	43884
3	62275	62464	63684	62618	62976	62712	61997
4	82940	84854	84084	84208	84680	84348	83142
5	108924	110853	112896	111758	112896	109512	112404
6	131008	135050	129792	132487	134862	130662	131688
7	154570	153340	149144	163116	162432	160094	166530
8	180744	181152	181020	195300	174603	189318	189696
9	205572	205552	222300	211754	206610	217350	217945
10	235708	212160	240552	231352	222750	232243	218880

Table A.11: Tile wirelength before routing for 4x4 SRAM grouping

Architecture	Tile wirelength before routing (grid squares)						
	Multiplier for number of pin swap moves						
	1x	2x	3x	4x	5x	6x	7x
1	48604	48193	48982	47501	48247	47208	47879
2	102232	101405	99573	101276	98918	102078	99230
3	163300	159200	161195	157380	156611	155999	155622
4	218981	218488	212646	212847	209297	212740	209592
5	301808	299409	297939	295468	299375	291324	291519
6	380016	379120	359150	367213	375200	357937	369487
7	439076	432635	425574	430101	429571	430583	425491
8	499729	505960	498125	497900	485013	496944	503040
9	576914	570019	571395	564743	565421	564389	571342
10	638094	625010	627127	624710	630826	611249	620468

Table A.12: Programming wirelength before routing for 4x4 SRAM grouping

Architecture	Programming wirelength before routing (grid squares)						
	Multiplier for number of pin swap moves						
	1x	2x	3x	4x	5x	6x	7x
1	6285	6504	6899	7078	6876	6561	7096
2	12268	12745	12522	12846	12263	12691	12174
3	19113	18555	18736	18587	18498	18815	18690
4	23289	25291	23643	24757	23411	24162	24772
5	33366	32844	34334	32320	32883	33199	33109
6	37481	40545	39760	39063	39023	37666	39152
7	44605	45754	44597	45637	45945	44610	45334
8	49595	50537	52297	53364	50610	52049	51286
9	56210	56500	58877	56425	56069	57656	58732
10	62046	59902	61791	62384	61240	62088	60738

Table A.13: SRAM output wirelength before routing for 4x4 SRAM grouping

Architecture	SRAM output wirelength before routing (grid squares)						
	Multiplier for number of pin swap moves						
	1x	2x	3x	4x	5x	6x	7x
1	12216	12035	12201	11803	11789	11313	11613
2	22531	22706	21815	22488	22158	21097	20651
3	37608	33947	36689	33167	31203	31368	31994
4	44760	44320	42060	43795	41786	41840	40737
5	63460	60104	60673	56662	56436	57692	57464
6	77157	77593	67164	66457	68627	67888	70161
7	86717	79177	79130	78169	80729	78953	72926
8	94760	94635	90997	89252	86181	87546	97718
9	109530	101624	100236	103433	96055	101497	97964
10	122889	111831	119176	110769	112189	106561	107137

Table A.14: Tile area before routing for SRAM groupings

Architecture	Tile area before routing (grid squares)						
	No grouping	SRAM grouping					
		2x2	3x3	3x4	4x4	4x5	5x5
1	26832	24178	23100	24048	22464	22704	22378
2	49896	45980	44096	43758	43281	42728	43054
3	70932	64498	62331	63250	62275	61360	61696
4	96398	89012	84970	85039	82940	83811	83122
5	124942	116795	111555	109890	108924	108836	109200
6	150100	144375	131350	132588	130285	128975	130221
7	170982	159032	149768	149040	148740	146960	147864
8	195960	181900	189272	171808	168597	170602	168245
9	217413	198907	203343	191406	189125	186263	188550
10	239259	213834	208182	210132	205660	206400	202950

Architecture	Tile area before routing (grid squares)						
	SRAM grouping						
	5x6	6x6	6x7	7x7	8x8	9x9	10x10
1	22078	22378	24420	22575	22419	23625	22968
2	45888	43560	44908	43537	45567	42080	41478
3	62997	62478	61020	60784	60973	62602	60696
4	82524	82446	81355	86142	83230	85228	81984
5	108661	108225	107200	108924	107868	108896	112161
6	128412	130755	129210	131369	127117	129430	129360
7	146216	149733	149362	147132	145112	144824	149946
8	167217	167475	168378	171741	165170	167085	166752
9	186036	187425	187766	186192	187515	188131	189354
10	202419	201960	201132	203775	199368	200688	200640

Table A.15: Tile area after routing for SRAM groupings

Architecture	Tile area after routing (grid squares)						
	No grouping	SRAM grouping					
		2x2	3x3	3x4	4x4	4x5	5x5
1	26832	24178	23100	24048	22464	22704	23598
2	49896	45980	44096	43758	43281	42728	43054
3	70932	64498	62832	65535	62275	61360	61696
4	98587	91104	85554	85039	82940	83811	83122
5	126360	116795	112224	111555	108924	108836	109200
6	154014	145899	135750	132588	131008	128975	131688
7	186192	174000	159200	154088	154570	153900	157963
8	201312	187920	192375	175140	180744	190806	175775
9	228206	205205	214020	195364	205572	206150	206330
10	254828	216618	228656	228830	235708	229439	220430

Architecture	Tile area after routing (grid squares)						
	SRAM grouping						
	5x6	6x6	6x7	7x7	8x8	9x9	10x10
1	22078	22378	24420	22575	22419	23625	22968
2	45888	43560	44908	43537	45567	42080	41478
3	63246	62980	61517	60784	60973	62602	60696
4	82823	82446	81355	86142	83230	85813	81984
5	108661	110536	107870	109935	107868	108896	112161
6	132750	130755	129930	139040	132925	136710	136965
7	159879	154026	155220	155324	152036	166950	163800
8	181882	175824	175848	184730	183570	182988	198900
9	218446	201978	204223	209728	209292	229297	222950
10	217968	222222	220891	233220	241664	261630	239720

Table A.16: Tile wirelength before routing for SRAM groupings

Architecture	Tile wirelength before routing (grid squares)						
	No grouping	SRAM grouping					
		2x2	3x3	3x4	4x4	4x5	5x5
1	46605	46989	48045	48511	48604	49615	50893
2	95662	98383	100905	100109	102232	100629	106932
3	151285	155364	153968	162841	163300	164395	168077
4	202051	213390	213334	216673	218981	219751	218101
5	281464	294573	295478	298208	301808	304942	298018
6	353384	367707	373125	370252	380016	373248	386210
7	423721	440330	436722	427063	439076	442266	449214
8	477275	496504	506274	499823	499729	512007	515519
9	549871	551736	580451	575030	576914	571261	581300
10	600678	615258	627015	641292	638094	642332	637138

Architecture	Tile wirelength before routing (grid squares)						
	SRAM grouping						
	5x6	6x6	6x7	7x7	8x8	9x9	10x10
1	51167	52618	55227	55410	56133	61674	60619
2	107991	106976	109894	112140	118849	120141	124889
3	167597	168031	172588	175012	178764	187874	189440
4	222779	223223	229826	234765	239508	247081	254283
5	304699	306085	322923	319302	328186	340241	354990
6	380022	395405	398077	401747	414305	424807	431974
7	455111	460797	459882	457848	471950	495204	510501
8	531762	528606	534619	545619	554396	562500	577300
9	595347	596711	615619	616939	625304	645092	671704
10	659252	650675	661315	671776	679974	692226	715522

Table A.17: Programming wirelength before routing for SRAM groupings

Architecture	Programming wirelength before routing (grid squares)						
	No grouping	SRAM grouping					
		2x2	3x3	3x4	4x4	4x5	5x5
1	9173	7638	6761	7243	6285	6469	7610
2	17346	14607	13531	13148	12268	11823	12298
3	26516	21562	18105	19462	19113	19243	19270
4	35394	28458	26267	25227	23289	23500	24739
5	48282	37004	33178	33127	33366	31465	30135
6	59761	46531	42247	39392	37481	37055	37941
7	70140	53614	46815	44131	44605	41302	45030
8	79443	61588	55554	50251	49595	49864	46979
9	91726	69225	63129	58057	56210	55033	52976
10	100623	73814	65839	63633	62046	57512	60168

Architecture	Programming wirelength before routing (grid squares)						
	SRAM grouping						
	5x6	6x6	6x7	7x7	8x8	9x9	10x10
1	6534	6933	8333	6969	7266	8834	7667
2	12861	12333	12446	13658	13457	12241	13304
3	18051	19483	18028	17610	18106	17497	18924
4	22444	23520	24230	23040	22762	24042	22378
5	30257	28899	31187	30824	28860	27507	30749
6	35532	36661	35565	36750	34126	37115	34860
7	43693	44712	41581	40432	39469	38096	42234
8	48272	47268	46320	48140	45285	45474	44942
9	52388	52175	54809	50595	53761	50669	51530
10	57852	56621	54654	58802	52972	53006	55535

Table A.18: SRAM output wirelength before routing for SRAM groupings

Architecture	SRAM output wirelength before routing (grid squares)						
	No grouping	SRAM grouping					
		2x2	3x3	3x4	4x4	4x5	5x5
1	6288	8764	11122	11072	12216	12993	14478
2	11005	15585	20109	21168	22531	24541	26071
3	17134	23783	29065	34782	37608	37526	42396
4	22320	33317	40233	42257	44760	48696	51651
5	31586	42132	49375	60009	63460	68046	70863
6	36987	51888	65325	67530	77157	75693	88375
7	46876	63014	74399	79261	86717	92815	102453
8	46841	65120	83567	92068	94760	105464	111823
9	55157	71969	96459	104821	109530	113735	125510
10	58670	79524	99224	116465	122889	126897	138998

Architecture	SRAM output wirelength before routing (grid squares)						
	SRAM grouping						
	5x6	6x6	6x7	7x7	8x8	9x9	10x10
1	15460	16112	17301	18822	20383	22041	24396
2	29007	28554	31109	32731	37432	41379	46301
3	44177	46413	48551	53101	57109	62522	67988
4	53876	56239	64029	66465	74681	80628	85454
5	73187	78580	84722	88309	95761	107790	117632
6	90574	95240	102768	109282	120162	128388	139574
7	108903	111008	118106	119624	135074	151654	159149
8	124185	123897	137628	138603	156888	168265	180970
9	139504	138302	159337	157759	168095	185761	206825
10	153839	155558	163164	164975	185899	200902	215452

Table A.19: Tile area before routing for combined groupings

Architecture	Tile area before routing (grid squares)				
	No grouping	Combined grouping			
		1x1	2x2	3x3	4x4
1	26832	25418	23400	22200	21648
2	49896	47952	43956	42636	40994
3	70932	69048	64507	61468	60876
4	96398	91956	84091	80330	78957
5	124942	125874	110124	105525	102610
6	150100	141804	135036	129084	124656
7	170982	161579	149625	142880	142048
8	195960	187000	176336	164811	160776
9	217413	203432	194682	183897	178928
10	239259	226320	221188	204700	198475

Architecture	Tile area before routing (grid squares)			
	Combined grouping			
	5x5	6x6	7x7	8x8
1	21594	21315	21195	21195
2	40800	42612	42680	41322
3	58928	59220	58824	59286
4	78279	79523	77283	76708
5	101440	102510	103734	101332
6	124500	123004	120373	119238
7	139876	139795	139515	135014
8	158840	157500	160272	158782
9	179280	178500	174985	178048
10	206064	196692	196686	196650

Table A.20: Tile area after routing for combined groupings

Architecture	Tile area after routing (grid squares)				
	No grouping	Combined grouping			
		1x1	2x2	3x3	4x4
1	26832	26718	23400	22200	21648
2	49896	47952	43956	42636	40994
3	70932	69048	64507	61468	60876
4	98587	91956	84091	80330	78957
5	126360	126252	110466	105525	106829
6	154014	141804	135772	129804	130410
7	186192	167678	155856	146688	145452
8	201312	193536	180128	170520	172610
9	228206	218094	196880	194712	196690
10	254828	233532	232674	221760	214896

Architecture	Tile area after routing (grid squares)			
	Combined grouping			
	5x5	6x6	7x7	8x8
1	22750	21315	21195	21195
2	40800	42612	42680	41322
3	58928	59220	58824	59286
4	78570	79523	77283	76708
5	108570	102510	103734	101970
6	124500	123004	124573	127253
7	144020	144300	141372	147420
8	171936	167616	176732	168084
9	199230	189216	190393	189189
10	234188	218085	214830	219897

Table A.21: Tile wirelength before routing for combined groupings

Architecture	Tile wirelength (grid squares)				
	No grouping	Combined grouping			
		1x1	2x2	3x3	4x4
1	46605	42944	43471	45158	45525
2	95662	92296	93695	96853	97428
3	151285	144429	148057	153921	156185
4	202051	193651	204774	205612	206542
5	281464	270736	279489	281359	286227
6	353384	337163	349106	361473	363189
7	423721	400603	410145	412234	418920
8	477275	461410	473124	470140	492189
9	549871	534098	543562	544321	559892
10	600678	573367	612036	606945	624168

Architecture	Tile wirelength (grid squares)			
	Combined grouping			
	5x5	6x6	7x7	8x8
1	49081	48898	51883	53586
2	95870	104011	107393	108640
3	159873	165221	166587	172571
4	208621	216274	221149	228523
5	294330	299096	304423	311302
6	375315	373496	384609	395404
7	428442	442322	443655	454334
8	488258	497881	515649	528717
9	572498	574545	586230	599001
10	644447	633306	652475	662758

Table A.22: Programming wirelength before routing for combined groupings

Architecture	Programming wirelength (grid squares)				
	No grouping	Combined grouping			
		1x1	2x2	3x3	4x4
1	9173	8507	7438	6643	6763
2	17346	17202	14212	12899	12184
3	26516	26486	20806	18626	17789
4	35394	35653	27616	25255	24079
5	48282	48277	36263	32465	32228
6	59761	57921	44956	41140	37517
7	70140	66893	50701	44864	43521
8	79443	79242	60086	51821	49363
9	91726	87590	68879	59493	57296
10	100623	97213	75231	63864	58507

Architecture	Programming wirelength (grid squares)			
	Combined grouping			
	5x5	6x6	7x7	8x8
1	7176	6488	6250	6801
2	11878	12472	13335	12700
3	17863	18630	16225	18028
4	23008	22635	23591	21805
5	29308	28664	29856	27559
6	36077	36475	35420	37351
7	41262	41123	38643	38952
8	46269	45805	45525	46096
9	51222	49740	49090	48734
10	61440	54552	55919	55883

Table A.23: SRAM output wirelength before routing for combined groupings

Architecture	SRAM output wirelength (grid squares)				
	No grouping	Combined grouping			
		1x1	2x2	3x3	4x4
1	6288	6217	8635	10103	12190
2	11005	12407	16234	20643	24183
3	17134	18686	25182	34485	36978
4	22320	24387	33558	39846	47875
5	31586	33309	47295	54240	70109
6	36987	42382	53874	70270	81715
7	46876	49604	67475	75172	88420
8	46841	53814	69747	86541	101835
9	55157	64270	77825	95853	120314
10	58670	64124	89784	108152	131343

Architecture	SRAM output wirelength (grid squares)			
	Combined grouping			
	5x5	6x6	7x7	8x8
1	14998	16683	18400	21309
2	25621	31002	33162	37344
3	40768	45754	53699	58757
4	54739	61361	66118	73798
5	74361	78452	88689	99663
6	93414	98013	113084	123522
7	101484	114612	127546	142663
8	117643	130118	143808	159957
9	140064	151334	162994	170895
10	154304	158804	174356	193178

Appendix B

POWELL Architecture Description

```
# Uniform channels. Each pin appears on only one side.
io_rat 2         #2 pads per row or column
chan_width_io 1
chan_width_x uniform 1
chan_width_y uniform 1

#Cluster of size 3, with 8 logic inputs
inpin class: 0 bottom
inpin class: 0 left
inpin class: 0 top
inpin class: 0 right
inpin class: 0 bottom
inpin class: 0 left
inpin class: 0 top
inpin class: 0 right
outpin class: 1 bottom
outpin class: 1 left
outpin class: 1 top
inpin class: 2 global right       #Clock; shouldn't matter
#Class 0 is LUT inputs, class 1 is the output, class 2 is the clock.

subblocks_per_clb 3
subblock_lut_size 4

#parameters needed only for detailed routing.
switch_block_type subset
Fc_type fractional
Fc_output 0.333333333333333
Fc_input 0.6
```

```
Fc_pad 0.6

segment frequency: 1.0 length: 4 wire_switch: 0 opin_switch: 0 \
Frac_cb: 1 Frac_sb: 1 Rmetal:  29.079 Cmetal:  3.546e-14

# The routing architecture is fully buffered!
switch 0 buffered: yes R: 2032.037  Cin: 1.6200e-15 Cout: 1.6849e-15 \
Tdel: 3.5610e-11

R_minW_nmos 4565
R_minW_pmos 8674   # 1.9x R of an nmos

# Timing info below.
C_ipin_cblock 1.62e-15
T_ipin_cblock 3.7700e-10
T_ipad 242e-12    #Clk_to_Q + 2:1 mux
T_opad 4.7e-11
T_sblk_opin_to_sblk_ipin 2.7000e-10
T_clb_ipin_to_sblk_ipin 2.7000e-10
T_sblk_opin_to_clb_opin 0

T_subblock T_comb: 3.73e-10  T_seq_in: 3.48e-10   T_seq_out: 2.42e-10
T_subblock T_comb: 3.73e-10  T_seq_in: 3.48e-10   T_seq_out: 2.42e-10
T_subblock T_comb: 3.73e-10  T_seq_in: 3.48e-10   T_seq_out: 2.42e-10
```

Appendix C

POWELL Cell Layout Information

```
# CELL Format: cell_type "Name" width height num_pins
#              (pin_class x_offset y_offset) (...) etc for num_pins times

0 "1x_Inverter" 7 5 4 (5 2 4) (0 5 1) (1 2 2) (6 4 1)
0 "2x_Inverter" 5 6 4 (5 1 3) (0 2 2) (1 3 2) (6 1 1)
0 "4x_Inverter" 6 6 4 (5 3 4) (0 2 2) (1 3 2) (6 3 0)
0 "4x_Non_inv_Buffer" 7 6 4 (5 3 4) (0 1 2) (1 4 2) (6 3 0)

1 "SRAM" 20 24 42 (5 7 12) (2 9 23) (2 9 12) (2 9 11) (2 9 0) (3 1 12)  \
                  (3 9 12) (3 10 12) (3 18 12) (4 2 20) (4 4 20)        \
                  (4 8 20) (4 5 20) (4 11 20) (4 14 20) (4 17 20)       \
                  (4 15 20) (4 2 15) (4 4 15) (4 8 15) (4 5 15)         \
                  (4 11 15) (4 14 15) (4 17 15) (4 15 15) (4 2 8)       \
                  (4 4 8) (4 8 8) (4 5 8) (4 11 8) (4 14 8) (4 17 8)    \
                  (4 15 8) (4 2 3) (4 4 3) (4 8 3) (4 5 3) (4 11 3)     \
                  (4 14 3) (4 17 3) (4 15 3) (6 12 12)

2 "2_input_MUX" 4 5 6 (0 1 4) (0 1 0) (0 0 3) (0 0 1) (1 2 2) (6 2 0)
2 "11_input_MUX" 11 10 22 (0 0 2) (0 0 6) (0 1 9) (0 2 5) (0 2 9)      \
                          (0 6 5) (0 6 9) (0 9 6) (0 9 9) (0 7 5)      \
                          (0 7 9) (0 1 0) (0 4 0) (0 6 0) (0 8 0)      \
                          (0 7 3) (0 3 4) (0 4 6) (0 4 8) (1 3 1)      \
                          (6 0 1) (6 10 1)
2 "12_input_MUX" 12 10 23 (0 3 5) (0 3 9) (0 1 6) (0 1 9) (0 7 5)      \
                          (0 7 9) (0 4 5) (0 4 9) (0 8 5) (0 8 9)      \
                          (0 10 6) (0 10 9) (0 2 0) (0 5 0) (0 7 0)    \
                          (0 9 0) (0 3 3) (0 8 3) (0 5 6) (0 5 8)      \
                          (1 4 1) (6 0 1) (6 10 1)
2 "20_input_MUX" 14 18 34 (0 12 3) (0 12 0) (0 10 3) (0 10 0) (0 6 4)  \
                          (0 6 0) (0 8 3) (0 8 0) (0 3 3) (0 4 0)       \
                          (0 2 4) (0 2 0) (0 6 13) (0 6 17) (0 8 14)    \
                          (0 8 17) (0 3 14) (0 4 17) (0 2 13) (0 2 17)  \
                          (0 12 8) (0 12 12) (0 10 7) (0 10 11) (0 7 9) \
                          (0 4 8) (0 2 8) (0 3 8) (0 1 8) (0 0 8)       \
                          (1 12 9) (6 4 8) (6 10 14) (6 13 1)

3 "LUT" 11 18 28 (0 0 8) (0 1 8) (0 3 8) (0 2 8) (0 5 8) (0 7 9)        \
                 (0 10 11) (0 10 6) (0 6 4) (0 6 0) (0 8 3) (0 8 0)     \
                 (0 3 3) (0 4 0) (0 2 4) (0 2 0) (0 6 13) (0 6 17)      \
                 (0 8 14) (0 8 17) (0 3 14) (0 4 17) (0 2 13) (0 2 17)  \
                 (1 10 9) (6 4 8) (6 10 14) (6 10 3)

4 "Flipflop" 12 9 7 (5 5 4) (0 1 2) (0 9 6) (0 5 2) (1 1 6) (6 0 0)     \
                    (6 9 5)
4 "FlipFlopenable" 14 9 8 (5 7 4) (0 1 1) (0 11 6) (0 7 2) (1 3 3)      \
                          (6 0 2) (6 11 5) (0 2 6)

6 "4x_Buffer_Switch_Size_4" 7 6 5 (5 3 4) (0 1 2) (0 6 2) (1 5 1) (6 2 1)

9 "And_gate" 8 5 5 (5 3 4) (0 2 2) (0 5 2) (1 6 2) (6 3 0)

13 "Pmos_pullup" 6 5 3 (5 2 4) (0 4 1) (6 3 1)
```

References

[1] J. M. Rabaey, *Digital Integrated Circuits: A Design Perspective.* Prentice Hall, 1996.

[2] M. M. Mano, *Digital Design.* Prentice Hall, 2nd ed., 1991.

[3] D. Chinnery and K. Keutzer, *Closing the Gap Between ASIC & Custom: Tools and Techniques for High-Performance ASIC Design.* Kluwer Academic Publishers, 2002.

[4] V. Betz, J. Rose, and A. Marquardt, *Architecture and CAD for Deep-Submicron FPGAs.* Kluwer Academic Publishers, 1999.

[5] K. Padalia, R. Fung, M. Bourgeault, A. Egier, and J. Rose, "Automatic transistor and physical design of FPGA tiles from an architectural specification," in *ACM/SIGDA Int'l Symp. Field-Programmable Gate Arrays*, pp. 164–172, 2003.

[6] J. Rose and S. Brown, "Flexibility of interconnection structures for field-programmable gate arrays," *IEEE J. Solid-State Circuits*, vol. 26, pp. 277–282, Mar. 1991.

[7] P. Chow, S. O. Seo, J. Rose, K. Chung, G. Paez-Monzon, and I. Rahardja, "The design of a SRAM-based field-programmable gate array—Part II: Circuit design and layout," *IEEE Trans. VLSI Systems*, vol. 7, pp. 321–330, Sept. 1999.

[8] A. S. Sedra and K. C. Smith, *Microelectronic Circuits.* Oxford University Press, 4th ed., 1998.

[9] J. Ferguson and A. J. Moore, “Solutions for maximizing die yield at 0.13 µm,” *Solid State Technology*, vol. 45, July 2002.

[10] F. J. Kurdahi and C. Ramachandran, “Evaluating layout area tradeoffs for high level applications,” *IEEE Trans. VLSI Systems*, vol. 1, pp. 46–55, Mar. 1993.

[11] H. Mecha, M. Fernandez, F. Tirado, J. Septien, D. Mozos, and K. Olcoz, “A method for area estimation of data-path in high level synthesis,” *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 15, pp. 258–265, Feb. 1996.

[12] A. C.-H. Wu, V. Chaiyakul, and D. D. Gajski, “Layout-area models for high-level synthesis,” in *IEEE Int'l Conf. Computer-Aided Design*, pp. 34–37, 1991.

[13] S. H. Gerez, *Algorithms for VLSI Design Automation.* John Wiley & Sons, 1999.

[14] N. Sherwani, *Algorithms for VLSI Physical Design Automation.* Kluwer Academic Publishers, 1993.

[15] T. Serdar and C. Sechen, “AKORD: Transistor level and mixed transistor/gate level placement tool for digital data paths,” in *IEEE Int'l Conf. Computer-Aided Design*, pp. 91–97, 1999.

[16] Synopsys, “Cadabra.” `http://www.synopsys.com/products/ntimrg/cadabra_ds.html`, Oct. 2004.

[17] Y. Ogawa, M. Pedram, and E. S. Kuh, “Timing-driven placement for general cell layout,” in *IEEE Int'l Symp. Circuits and Systems*, pp. 872–876, 1990.

[18] H. Onodera, Y. Taniguchi, and K. Tamaru, “Branch-and-bound placement for building block layout,” in *ACM/IEEE Design Automation Conf.*, pp. 433–439, 1991.

[19] M. Pedram, M. Marek-Sadowska, and E. S. Kuh, “Floorplanning with pin assignments,” in *IEEE Int'l Conf. Computer-Aided Design*, pp. 98–101, 1990.

[20] S. N. Adya and I. L. Markov, “Fixed-outline floorplanning through better local search,” in *Int'l Conf. Computer Design*, pp. 328–334, 2001.

[21] C.-C. Chang, J. Cong, and X. Yuan, “Multi-level placement for large-scale mixed-size IC designs,” in *Asia South Pacific Design Automation Conf.*, pp. 325–330, 2003.

[22] S. Phillips and S. Hauck, “Automatic layout of domain-specific reconfigurable subsystems for system-on-a-chip,” in *ACM/SIGDA Int'l Symp. Field-Programmable Gate Arrays*, pp. 165–173, 2002.

[23] N. Kafafi, K. Bozman, and S. J. E. Wilton, “Architectures and algorithms for synthesizable embedded programmable logic cores,” in *ACM/SIGDA Int'l Symp. Field-Programmable Gate Arrays*, pp. 3–11, 2003.

[24] K. Padalia, “Automated transistor-level design and layout placement of FPGA logic and routing from an architectural specification.” Bachelor's thesis, University of Toronto, 2001.

[25] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, “Optimization by simulated annealing,” *Science*, vol. 220, pp. 671–680, May 1983.

[26] C. Sechen and A. Sangiovanni-Vincentelli, “The TimberWolf placement and routing package,” *IEEE J. Solid-State Circuits*, vol. 20, pp. 510–522, Apr. 1985.

[27] R. Fung, “Optimization of transistor-level floorplans for field-programmable gate arrays.” Bachelor's thesis, University of Toronto, 2002.

[28] C. Y. Lee, “An algorithm for path connections and its applications,” *IRE Trans. Electronic Computers*, vol. EC-10, pp. 346–365, Sept. 1961.

[29] M. Bourgeault, J. Slavkin, and C. Sun, “Automatic transistor-level design and layout of FPGAs.” Design project report, University of Toronto, 2002.

[30] Cadence Design Systems, "Virtuoso Layout Editor." http://www.cadence.com/products/custom_ic/veditor/index.aspx, Oct. 2004.

[31] I. C. Kuon, "Automated FPGA design, verification and layout," Master's thesis, University of Toronto, 2004.

[32] Taiwan Semiconductor Manufacturing Company, "TSMC 0.18 and 0.15-micron technology platform." http://www.tsmc.com/download/english/a05_literature/0.15-0.18-micron_Brochure.pdf, Oct. 2004.

[33] Canadian Microelectronics Corporation. http://www.cmc.ca/, Oct. 2004.

[34] Cadence Design Systems, "Virtuoso custom design platform." http://www.cadence.com/products/custom_ic/index.aspx, Oct. 2004.

[35] Cadence Design Systems, "Virtuoso Chip Assembly Router." http://www.cadence.com/products/custom_ic/chip_assembly/index.aspx, Oct. 2004.

[36] S. So, "Automatic layout of FPGA tiles using one-layer metal cells." Bachelor's thesis, University of Toronto, 2003.

[37] Cadence Design Systems, "Diva physical verification." http://www.cadence.com/products/dfm/diva/index.aspx, Oct. 2004.

[38] Cadence Design Systems, "Virtuoso Analog Design Environment." http://www.cadence.com/products/custom_ic/analog_design/index.aspx, Oct. 2004.

[39] Cadence Design Systems. http://www.cadence.com/, Oct. 2004.

[40] Mentor Graphics, "Calibre DRC/LVS: Physical verification and manufacturability." http://www.mentor.com/calibre/datasheets/calibre/html/index.html, Oct. 2004.

[41] Synopsys, "NanoSim." `http://www.synopsys.com/products/mixedsignal/nanosim/nanosim.html`, Oct. 2004.