Automatic Transistor-Level Design and Layout Placement of FPGA Logic and Routing from an Architectural Specification

by

Ketan Padalia

Supervisor: Jonathan Rose

April 2001

Automatic Transistor-Level Design and Layout Placement of FPGA Logic and Routing from an Architectural Specification

by

Ketan Padalia

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
BACHELOR OF APPLIED SCIENCE

DIVISION OF ENGINEERING SCIENCE

FACULTY OF APPLIED SCIENCE AND ENGINEERING
UNIVERSITY OF TORONTO

Supervisor: Jonathan Rose

April 2001

ABSTRACT

Automatic Transistor-Level Design and
Layout Placement of FPGA Logic and
Routing from an Architectural Specification

Bachelor of Applied Science and Engineering, April 2001
Ketan Padalia
Division of Engineering Science
Faculty of Applied Science and Engineering
University of Toronto

One of the most intensive tasks involved in the design of FPGAs is chip layout. For commercial FPGAs, the layout is done largely by hand, in a process that takes many months to complete.

Our research creates an infrastructure that begins the process of allowing FPGA architects to create FPGA layouts automatically, requiring only a relatively simple description of the architecture of that FPGA.

In the first phase of our work, we develop tools that allow us to transform architectural descriptions of FPGAs into spice-style, transistor-level netlists that implement the logic and routing for single tiles of those FPGAs.

In the second phase of our work, we develop a tool that creates the layout placement of the FPGA tiles described by those netlists. Finally, we use this tool to demonstrate that the layout placement can be improved by taking advantage of domain-specific knowledge about the structures within our FPGAs.

ACKNOWLEDGMENTS

Above all, I would like to thank my supervisor, Jonathan Rose, for all the guidance and encouragement that he provided throughout this project. His unwavering enthusiasm and generous availability were a tremendous source of motivation for me.

I would also like to thank Vaughn Betz of Altera, whose groundbreaking research at the University of Toronto was essential to the feasibility of this project. In addition, I learned much of what I needed to know for this work through working with him at Right Track CAD and later at Altera Corporation.

I want to acknowledge the generous cooperation of Elias Ahmed and William Chow from Jonathan Rose's research group at the University of Toronto. Elias provided the set of architecture files that I used for my experiments, and William shared a Windows port of VPR's graphics package that I used for my own tool.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

Introduction

1.1 Motivation

Over the past two decades, FPGAs (Field-Programmable Gate Arrays) have become a popular medium for implementing digital circuits. A key reason for this popularity is the ability of a single FPGA chip to implement any circuit simply by being programmed appropriately. Despite the availability of other options, such as ASICs (Application-Specific Integrated Circuits) or Standard Cells, which provide significantly faster and smaller implementations, the programmability of FPGAs has allowed designers to achieve lower non-recurring engineering (NRE) costs and faster time-to-market for their designs [1]. Careful design of FPGAs, however, can limit the speed and area penalties relative to other options, making them a viable option for implementing a broader class of circuits at high volume.

The ability of an FPGA to provide good performance with low area rests on four primary factors: the logic and routing architecture of the FPGA, the transistor-level circuit design that is used to implement it, the software tools that are used to configure it, and its physical layout. When designing an FPGA, these four factors are heavily focused on to achieve the best possible performance, and as a result, they account for most of the time and resources required in the design process.

The complexity involved in each of these factors, however, often requires that decisions regarding one factor be made without any detailed idea of the impact that they would have on the other factors. For example, the software tools for an FPGA might not be able to take advantage of certain complex architectural features that seemed beneficial when the architecture was designed, potentially resulting in wasted area for unused logic on the final chip.

Ideally, every design decision that is made would consider the implications for all the factors influencing an FPGA's performance. In practice, however, this means that all FPGA architectures that are under consideration need to be designed, provided customized software tools, and laid out in order to accurately determine the best design. In a process that might consider hundreds of different architectures, this is obviously not a viable approach.

One solution to this problem is to design CAD tools that automate this process and allow a designer to quickly observe the impact of various decisions on overall FPGA performance. Over the past few years, research done at the University of Toronto has attempted to link three of the four factors presented above – an FPGA's architectural specification, transistor-level design, and software tools – to provide this capability [2]. This research has clearly demonstrated the benefits of being able to design an FPGA in the presence of detailed knowledge about how architectural decisions affect the circuit design and software tools, and hence the FPGA's performance.

These benefits lead to the hope that even greater advantages might be attainable by integrating the last of the four factors influencing FPGA performance with the other three. If there was a CAD tool that took an architectural specification of an FPGA and

automatically provided its physical layout in addition to the circuit design and the software tools, then design decisions could be made in the presence of precise information about the impact that they would have on all aspects of the FPGA's performance and cost.

The implications such tools would have on FPGA design go far beyond making the design process more informed. Using these tools, an FPGA could be designed and manufactured with a reasonable idea of its performance and cost, all without any significant engineering design effort. Depending on the performance penalties associated with designing an FPGA in this way, the very low cost of development could make it an extremely attractive solution. An FPGA manufactured in this way could also serve as an early prototype that could be followed up by improved versions. Alternatively, these tools could give designers a starting point that would save time in the design cycle and thus reduce costs.

Clearly, there are many advantages that could be reaped by the ability to provide physical layout in addition to transistor-level design and software tools, all from a single architectural specification. This work is an attempt to move closer to that goal.

1.2 Scope

This research involved two phases. In the first phase, we developed a tool that automatically creates a spice-style, transistor-level netlist of FPGA logic and routing based on an architectural specification given to it. In addition to the transistor-level netlist, it generates a cell-level netlist that allows small groups of transistors to be abstracted into cells.

In the second phase of our work, we developed a tool that creates layout placements for cell-level netlists of FPGA logic and routing. When creating these placements, we attempt to take advantage of the structure of the FPGAs that we are dealing with to achieve a better result. In this research, we demonstrate that this "domain-specific knowledge" can guide the automatic layout placement of an FPGA to a better solution.

The scope of our research was limited to performing placement for the cell-level netlist, which essentially forms a detailed floorplan of the transistor-level netlist. Determining the exact transistor-level layout within the different cells, and performing the routing between the cells, is left to future work.

1.3 Thesis Organization

Chapter 2 presents background information and details about the previous work that is relevant to this research. Chapter 3 describes the first phase of our work, which involves tileable netlist generation. Chapter 4 discusses the second phase of our work, which explores automatic layout placement of FPGA tile netlists. Chapter 5 demonstrates how FPGA layout can be improved by taking advantage of domain-specific knowledge about the structure of our netlists. Finally, Chapter 6 summarizes our conclusions and gives suggestions for future work.

Chapter 2

Background

The first section of this chapter provides a very brief overview of the FPGAs that we deal with in this work. The second section presents the use of repeated "tiles" of logic and routing in the implementation of FPGAs. The third section provides information about VPR, a result of previous research that was extended for use in the first phase of our work. The fourth section presents the transistor-level structures that we use to implement the FPGA tiles we deal with. The final section is a brief outline of the previous work that is relevant to the second phase of our research.

2.1 Overview of FPGAs

An FPGA is a circuit that can be configured to implement a wide variety of digital logic circuits. The FPGAs we consider in this work are composed of three broad classes of structures – logic blocks, programmable routing, and I/O pads. The logic blocks and routing are found in the "core" of the chip, surrounded by a ring of I/O pads on the perimeter. This arrangement is shown in Figure 2.1.

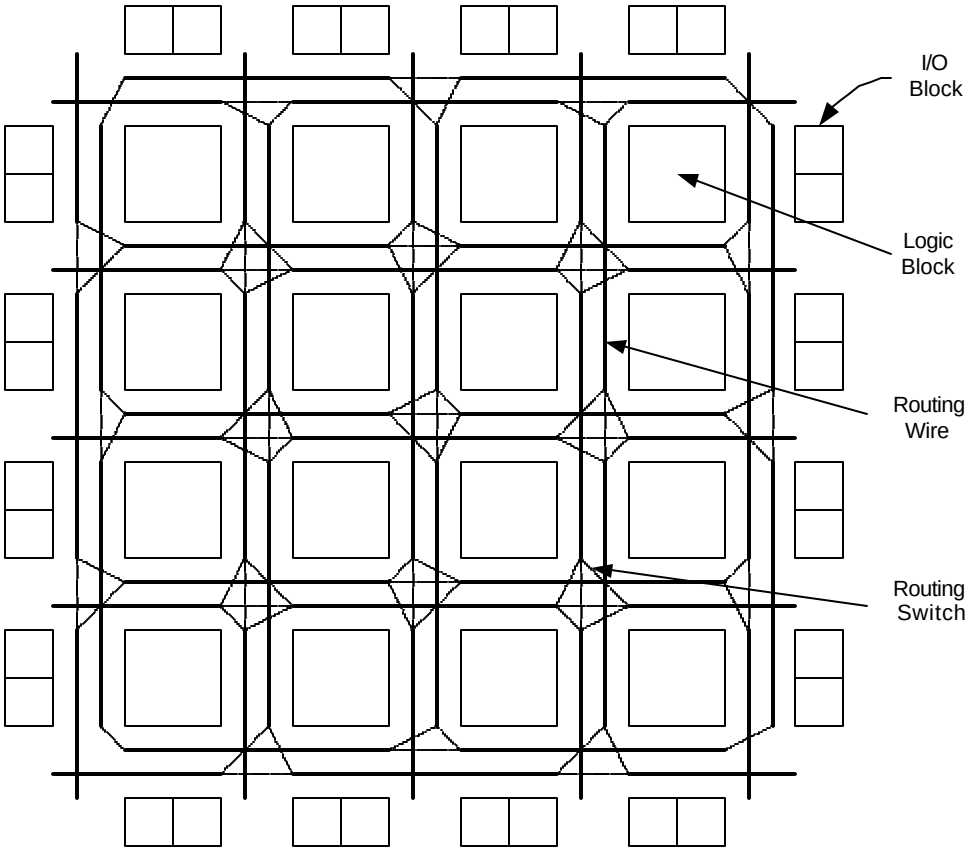


Figure 2.1 High-level view of FPGAs that we work with

Figure 2.1 also shows the wires that run in the channels between the logic blocks, as well as the programmable switches that allow signals to be sent from one wire to another. Switches can be simple pass transistors controlled by an SRAM cell, or can use a buffer to provide greater drive strength. Figure 2.2 shows both of these types of switches being used to connect different routing wires together.

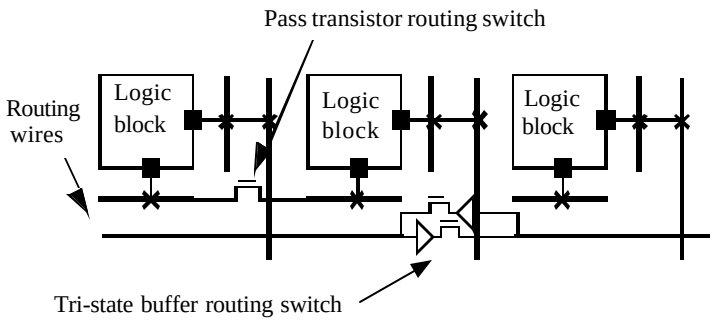


Figure 2.2 Two types of routing switches – adapted from [1]

Ultimately, routing wires and routing switches are used to connect logic blocks together. Figure 2.3 shows the contents of a single FPGA logic block.

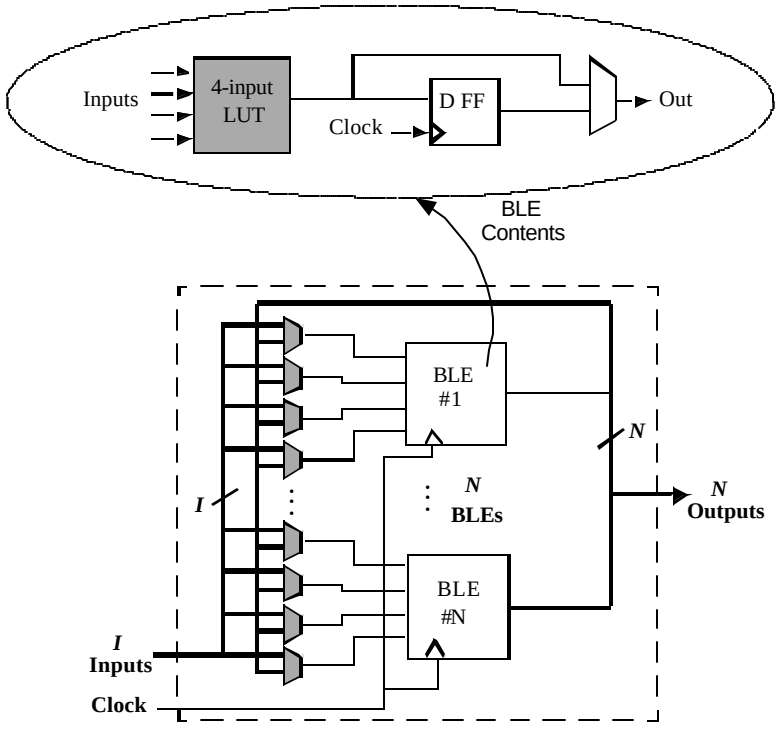


Figure 2.3 Logic block contents – adapted from [1]

As the figure shows, a logic block is made up of several BLEs (Basic Logic Elements). These BLEs have a programmable look-up table and a flip-flop that can be

used to implement small logic functions. These BLEs are connected to other BLEs in the same logic block by the internal feedback paths shown, as well as to BLES in other logic blocks via the I inputs and N outputs of the logic block.

To allow logic blocks to connect to routing wires, an FPGA has input connection blocks and output connection blocks. A set of typical input connection blocks is shown in Figure 2.4.

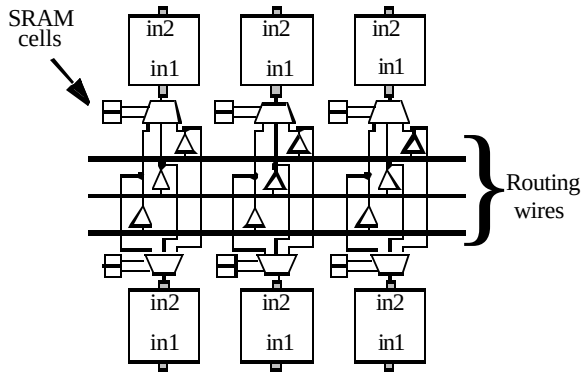


Figure 2.4 Typical input connection blocks – adapted from [1]

The figure shows three input connection blocks, one shared by each pair of vertically aligned logic blocks. Figure 2.5 shows a typical output connection block.

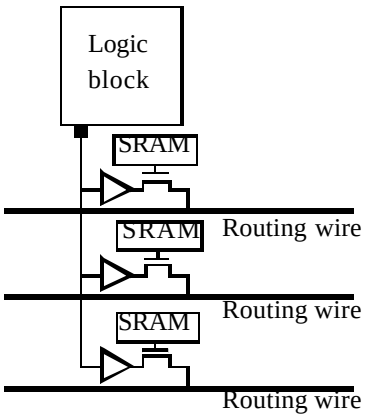


Figure 2.5 Typical output connection block – adapted from [1]

The architecture of an FPGA is determined by many different parameters, each of which affects one or more of the building blocks presented above. More detail about these parameters can be found in [1]. For the purposes of our research, we used values for these parameters that were found to be good in [1].

2.2 FPGA Tiles

An FPGA of the form shown in Figure 2.1 is often implemented by designing only one logic block and the programmable routing around it, forming an FPGA "tile". This single tile can, if designed properly, be duplicated and laid in a regular array to form the core of the FPGA. This process is shown in Figure 2.6, with a single tile being used to generate a 3-by-3 portion of the FPGA core.

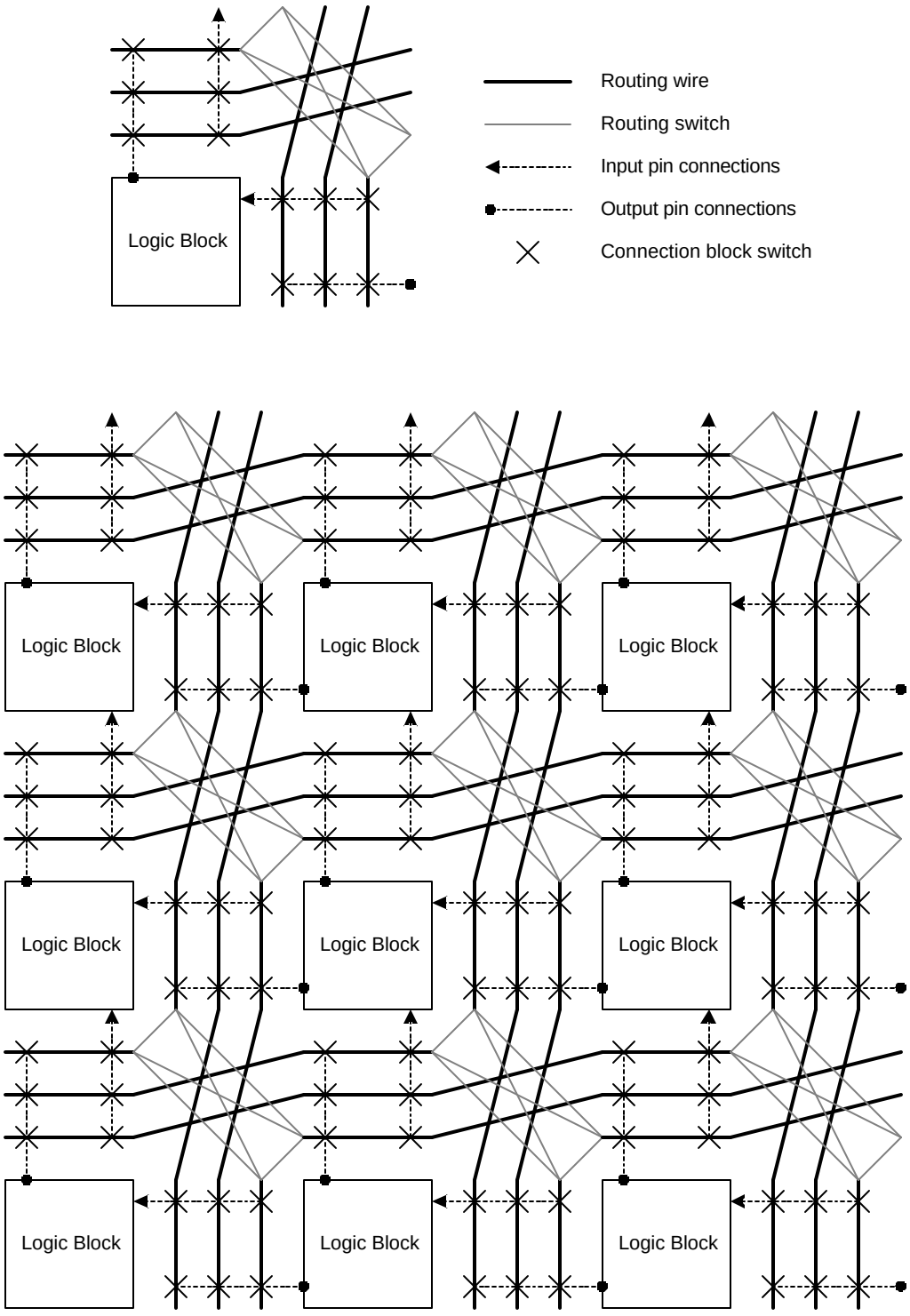


Figure 2.6 FPGA formed by replicating a single tile (shown at top)

2.3 VPR (Versatile Place and Route)

VPR is a flexible CAD tool that was designed at the University of Toronto [3]. As we describe in Chapter 3, we extended VPR to generate netlists for FPGA tiles. Thus, the VPR flow is intimately linked to the first phase of our work.

VPR performs clustering, placement, and routing of circuit netlists for a wide variety of FPGA architectures by using this architectural description. VPR gives FPGA designers the ability to observe the effects of various architectural decisions on an FPGA's software performance and transistor-level design. A simplified view of its CAD flow is shown in Figure 2.7.

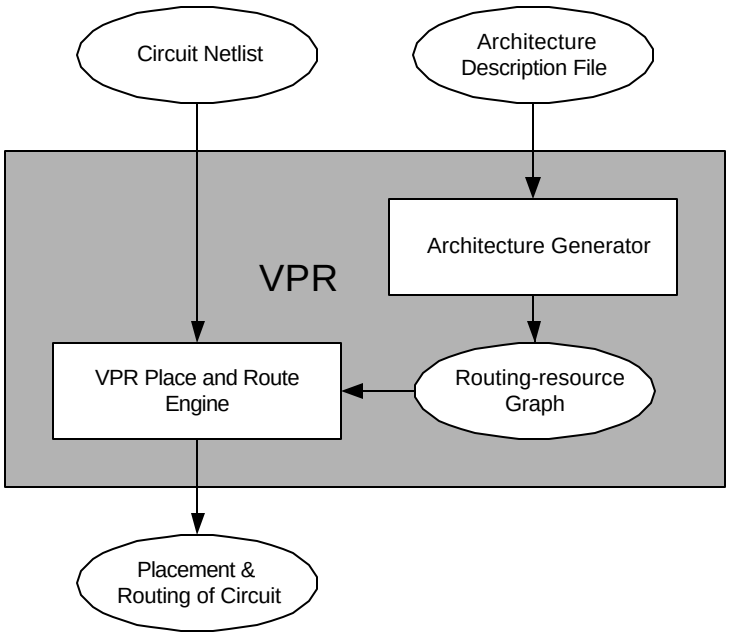


Figure 2.7 Simplified VPR CAD flow

2.3.2 VPR Architecture File

The primary input to VPR is an "architecture file", which contains a description of the architecture for the FPGA being considered. VPR uses this architectural specification to provide place and route capability for circuits. Figure 2.8 shows an excerpt of a typical VPR architecture file.

```
# Cluster of size 4, with 10 logic inputs (I = 10, N = 4)

# Logic block information
subblocks_per_clb 4
subblock_lut_size 4

# Logic block pin information
# Class 0 is LUT inputs, class 1 is the output

inpin class: 0 bottom
...
inpin class: 0 left
outpin class: 1 top
outpin class: 1 right
outpin class: 1 bottom
outpin class: 1 left

# Wire information
segment frequency: 0.5 length: 4 wire_switch: 0 opin_switch: 1 \
Frac_cb: 1 Frac_sb: 1 Rmetal:  300.0 Cmetal:  10.0e-14

# Switch information
switch 1 buffered: yes R: 500.0  Cin: 10.0e-15 Cout: 1.0e-15 \
Tdel: 1.0e-10

...
```

Figure 2.8 Excerpt of a VPR architecture file

The excerpt begins with a description of the number of logic blocks and the size of the lookup table in each logic block. It then describes the location and type of the logic block pins. The wire information is presented next, including the length, connectivity, as well as R and C values used in delay estimation. Finally, the excerpt

shows information for a switch, including the type of the switch and additional electrical information used for sizing and delay estimation. With only a few additional lines, this simple text description would capture a complete FPGA architecture.

VPR also uses this specification to compute an estimate of the area and delays that would characterize the FPGA if it were manufactured. The area estimates are based on assuming certain types and sizes for transistor-level structures in the FPGA, and the delay estimates are based on the resistance and capacitance values obtained for these structures through circuit simulation.

2.4 Transistor-level Structures

This section presents the transistor-level structures we assume throughout our work. For the most part, they are reproductions of the transistor-level structures assumed by VPR (the figures, where indicated, are adapted versions of those that appear in Appendix B of [1]). Cases where we assumed different structures than VPR are noted.

Figure 2.9 shows the schematics assumed for one of the key structures in an FPGA – the SRAM cell. Our schematic differs from the one assumed in VPR because we do not provide the inverted prog_data signal. Careful design of the SRAM cell would be able to overcome the need for that inverted signal in writing values into the cell [5].

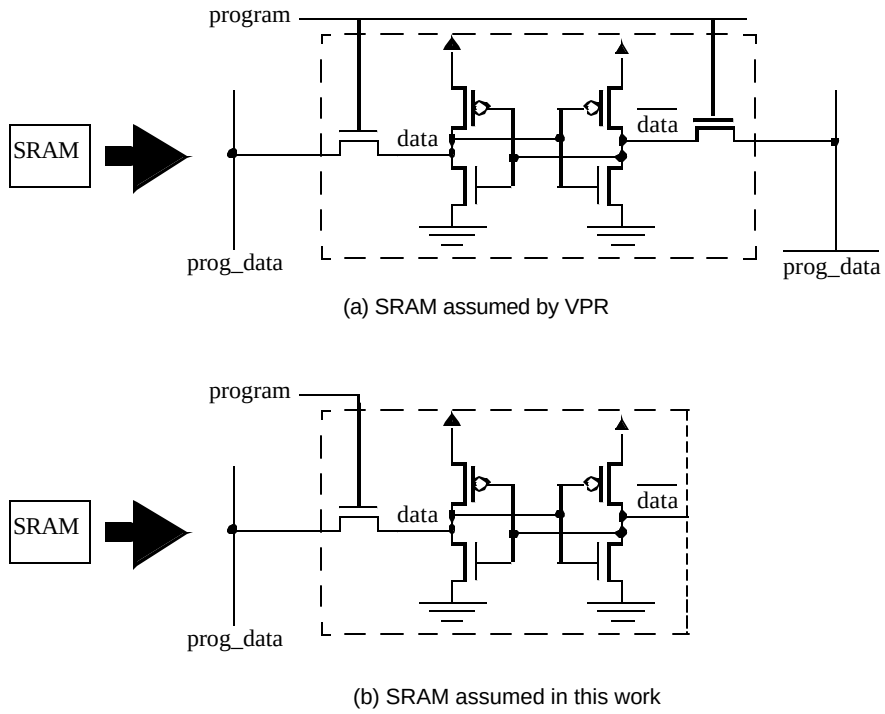


Figure 2.9 SRAM schematics used in (a) VPR and (b) in this work – adapted from [1]

Figure 2.10 shows the schematic assumed for a buffer. The exact sizing of the transistors depends on the drive strength required for the buffer, but follows the common technique of cascading increasingly larger stages together with a stage ratio near 4.

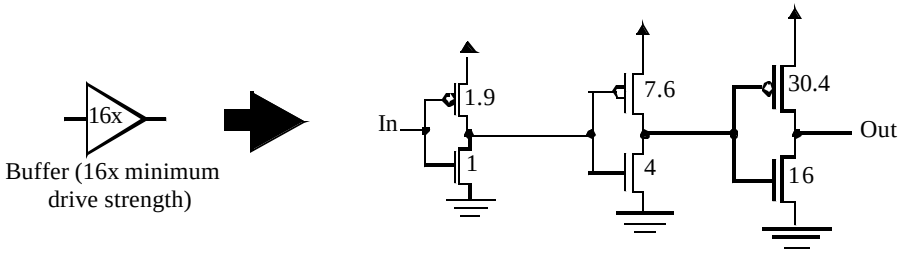


Figure 2.10 Buffer schematic – adapted from [1]

Figure 2.11 shows the schematic for a 4-input multiplexer, controlled by two SRAM cells. Larger or smaller multiplexers are implemented by modifying this structure as necessary.

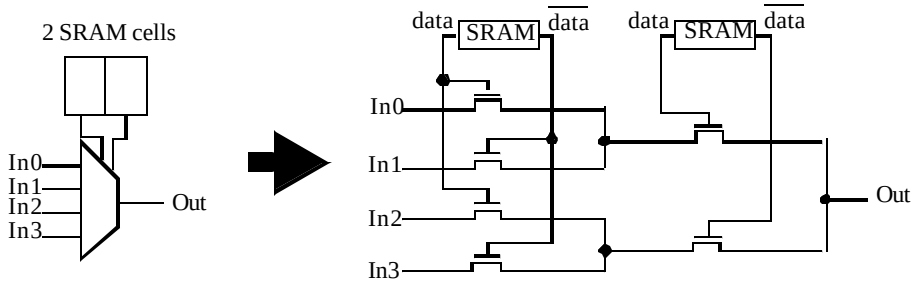


Figure 2.11 Multiplexer schematic – adapted from [1]

Figure 2.12 shows an overall view of the logic block schematic we use in our work.

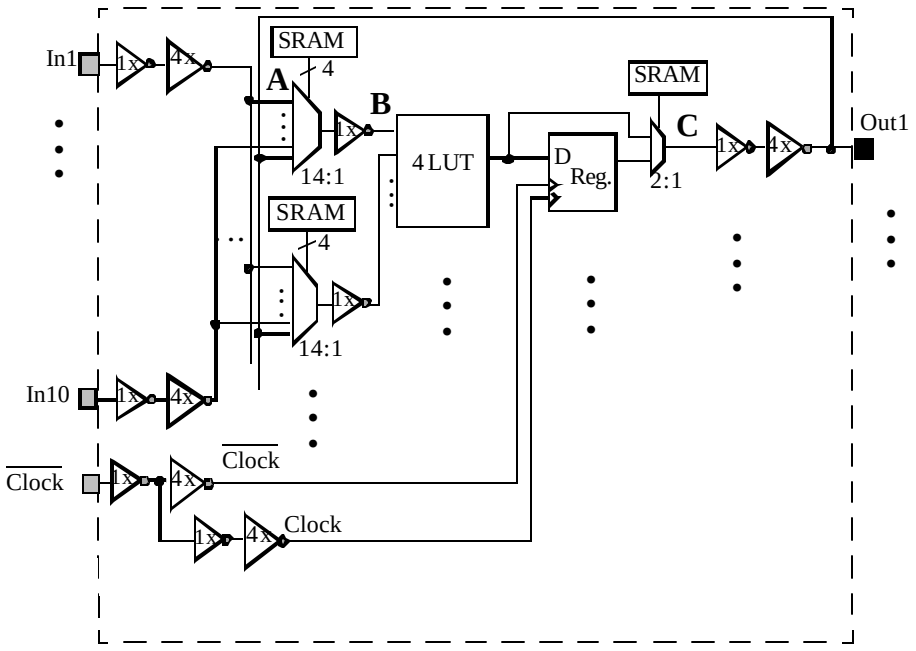


Figure 2.12 Logic block schematic – adapted from [1]

Figure 2.13 shows the schematics used for flip-flops in VPR and in our work. The only difference between the two is the availability of set and reset inputs in VPR that were left out in our work.

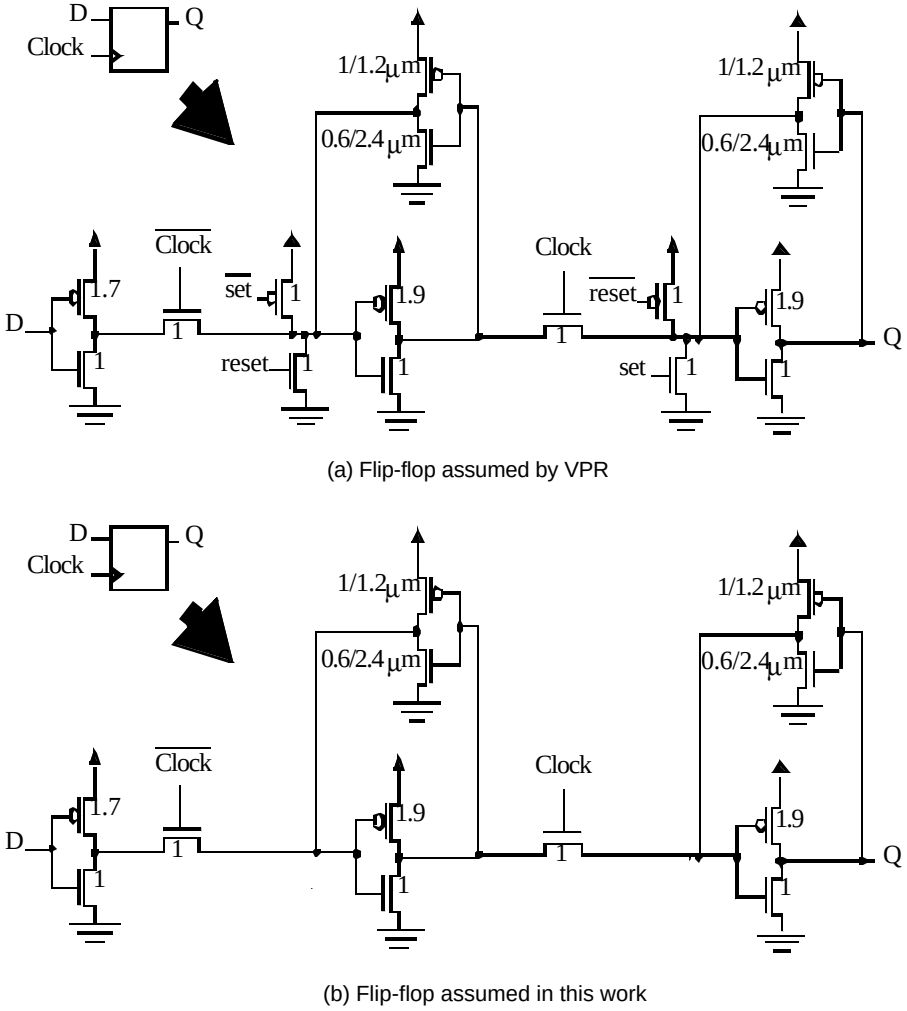


Figure 2.13 Flip-flop schematics used in (a) VPR and (b) in this work – adapted from [1]

Figure 2.14 shows the schematic used for LUTs in our work.

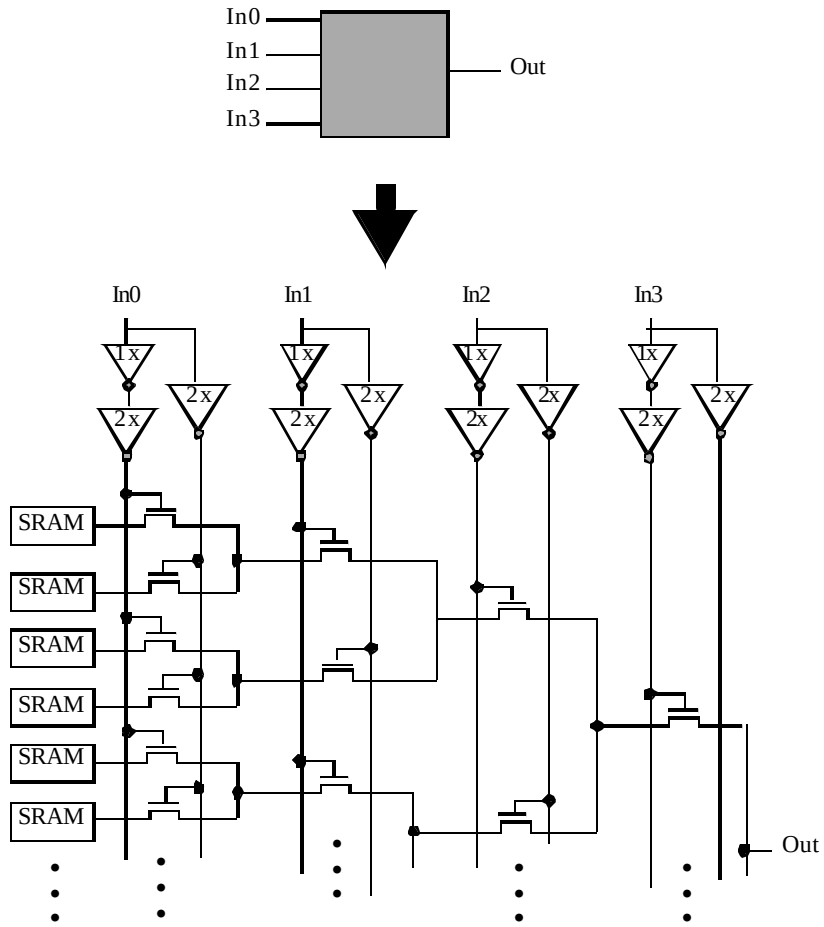


Figure 2.14 Look-up Table (LUT) schematic – adapted from [1]

2.5 Automatic Cell Layout Placement

As far as we are aware, there is no previous work that has attempted to link the layout of an FPGA tile to an architectural specification. However, much work has been done on the problem of placing blocks of varying sizes on a grid. This placement problem requires only that all blocks be placed onto the grid such that there is no overlap between them. While fulfilling this requirement, one of the jobs of a placement algorithm is to try and reduce the length of connections between blocks.

Our layout placement problem reduces to this placement problem because we operate on the cell-level netlist with the actual transistors abstracted away.

The simulated annealing algorithm is one popular approach in solving this problem. The Timberwolf tool [4] used this approach to handling "macrocell placement", with an algorithm that initially allows overlap in the placement of the differently sized blocks. A gradually increasing penalty is applied to this overlap to force the blocks apart. This is followed by a final "clean-up" phase that eliminates any overlap left at the end of the placement process.

The placement algorithm used in VPR [2] also uses simulated annealing, although without the presence of differently sized blocks.

Chapter 3

Tileable Netlist Generation

This chapter describes the first phase of our research, which involves generating netlists representing FPGA tiles.

3.1 Goals and Requirements

Our primary goal is to generate the transistor-level netlist of an FPGA tile using only an architectural description as input.

We also want these netlists to be usable in performing automatic layout for the tiles that they represent. In Chapter 4, we describe an automatic layout procedure that involves grouping transistors into cells that represent the various structures used in the netlist. This grouping makes the layout placement problem simpler by allowing groups of highly related transistors to be treated as one "black box". To make our netlists suitable for such use, we need to generate an additional, "cell-level", netlist that abstracts away some of the details of the transistor-level netlist.

Finally, in order to guide the automatic layout process in the second phase of our research, we need to annotate our netlist with as much domain-specific knowledge as possible about the structures in the tile.

3.2 CAD Flow

We decided to extend an existing tool, VPR (described in Section 2.3), in order to generate the netlists that we require. VPR already provides the ability to describe an FPGA with a simple "architecture file" and was an ideal starting point for meeting our goals. Figure 3.1 shows the CAD flow involved when using VPR_Layout. Notice that it is very similar to the CAD flow for VPR shown in Figure 2.7. The additional step of the tile netlist generator creates two extra outputs – the transistor-level and cell-level netlists.

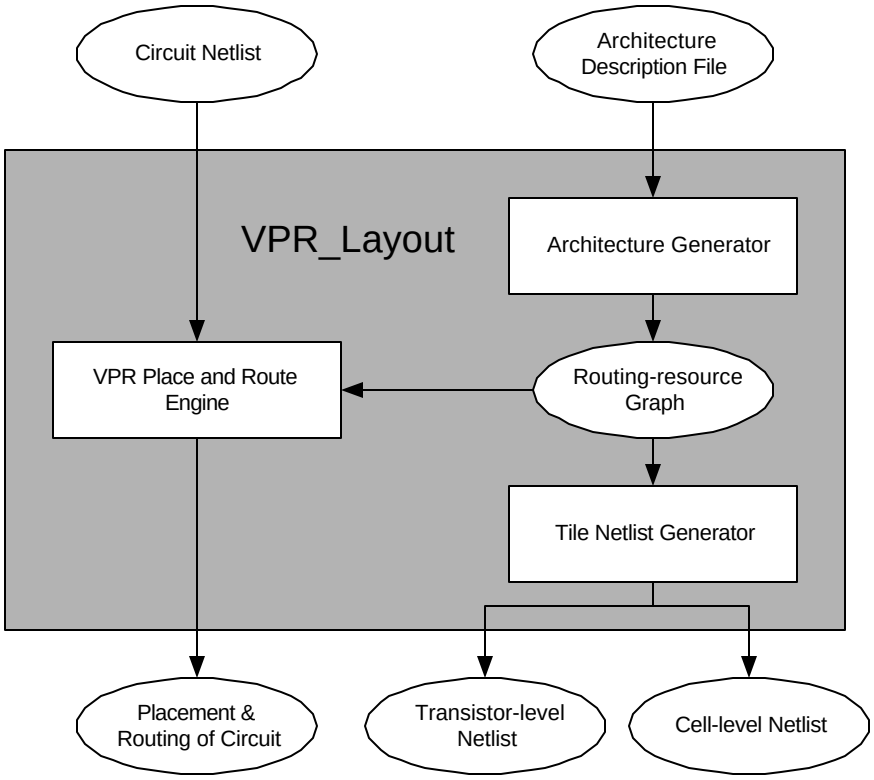


Figure 3.1 VPR_Layout CAD flow

The tile netlist generator in VPR_Layout operates on the routing-resource graph that VPR creates based on the architecture file. This routing-resource graph contains a

detailed representation of all the programmable routing in the FPGA and the connectivity to all the logic blocks as well. For generating the logic block structures, which cannot be found in the routing-resource graph, VPR_Layout assumes the transistor-level schematics that are presented in Section 2.4.

3.3 Transistor-level Netlist Generation

3.3.1 Transistor-level Netlist Boundary

In order to generate tile netlists that can be replicated and laid in an array as described in Section 2.2, we define a clear boundary for the part of the FPGA that the netlist represents. Figure 3.2 shows the boundary we use when making our netlists.

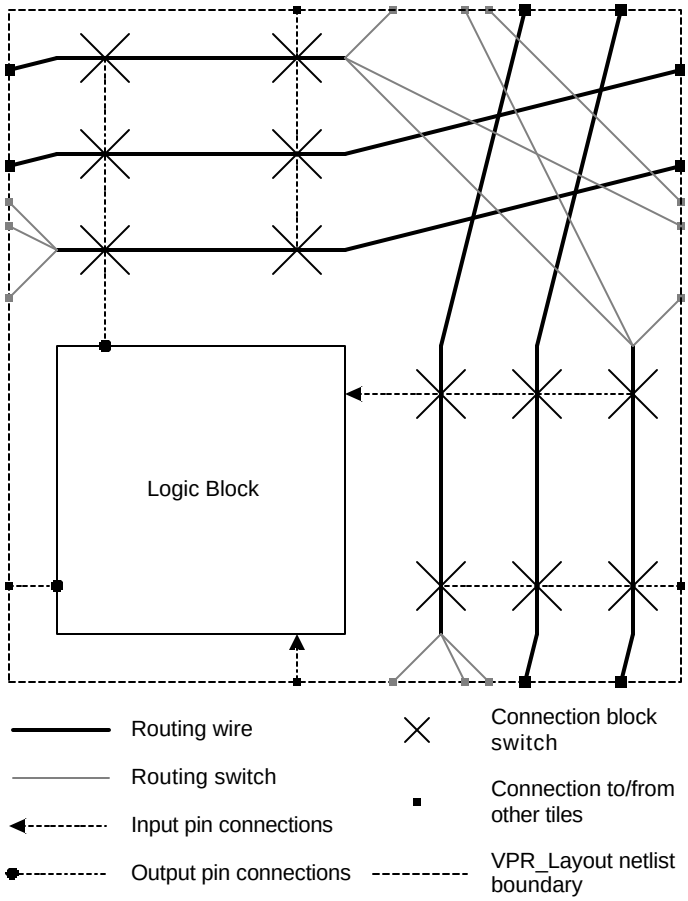


Figure 3.2 Boundary of VPR_Layout netlists

Our netlists include the logic block as well as the routing wires to the right of and above it. This view of the tile is a little less regular than the one presented in Figure 2.6, although replicating either tile results in the same FPGA. We used this slightly more complicated boundary because it simplified implementation of the netlist generator by requiring it to consider only those connections that have at least one end touching a wire or logic block pin that is included in the tile. The switch connection in the top-right corner of Figure 2.6 doesn't touch any wires in the tile and would complicate the implementation slightly.

3.3.2 Transistor-level Netlist Structure

A few sample lines from a typical transistor-level netlist are shown in Figure 3.3.

```
# FPGA tile transistor-level netlist
# Output by VPR_Layout

# PORT Format: <id> <node> <constraint class> <orientation>

# XTOR Format: <id> <drain> <gate> <source> <type> <size>
#              <cell type> <cell id> <subgroup type> <group type>

...
P11 3595 64 L
P12 3595 64 R
...
P24 3636 69 T
P25 3636 69 B
...
M0 35 2 1 P 2 0 0 0 0
M1 35 2 0 N 1 0 0 0 0
M2 36 35 1 P 8 0 1 0 0
M3 36 35 0 N 4 0 1 0 0
...
```

Figure 3.3 Sample section from a typical transistor-level netlist output by VPR_Layout

This sample illustrates the information that our netlists can convey about the tiles that they represent. Figure 3.4 shows the tile portion that would be implemented from the sample netlist of Figure 3.3.

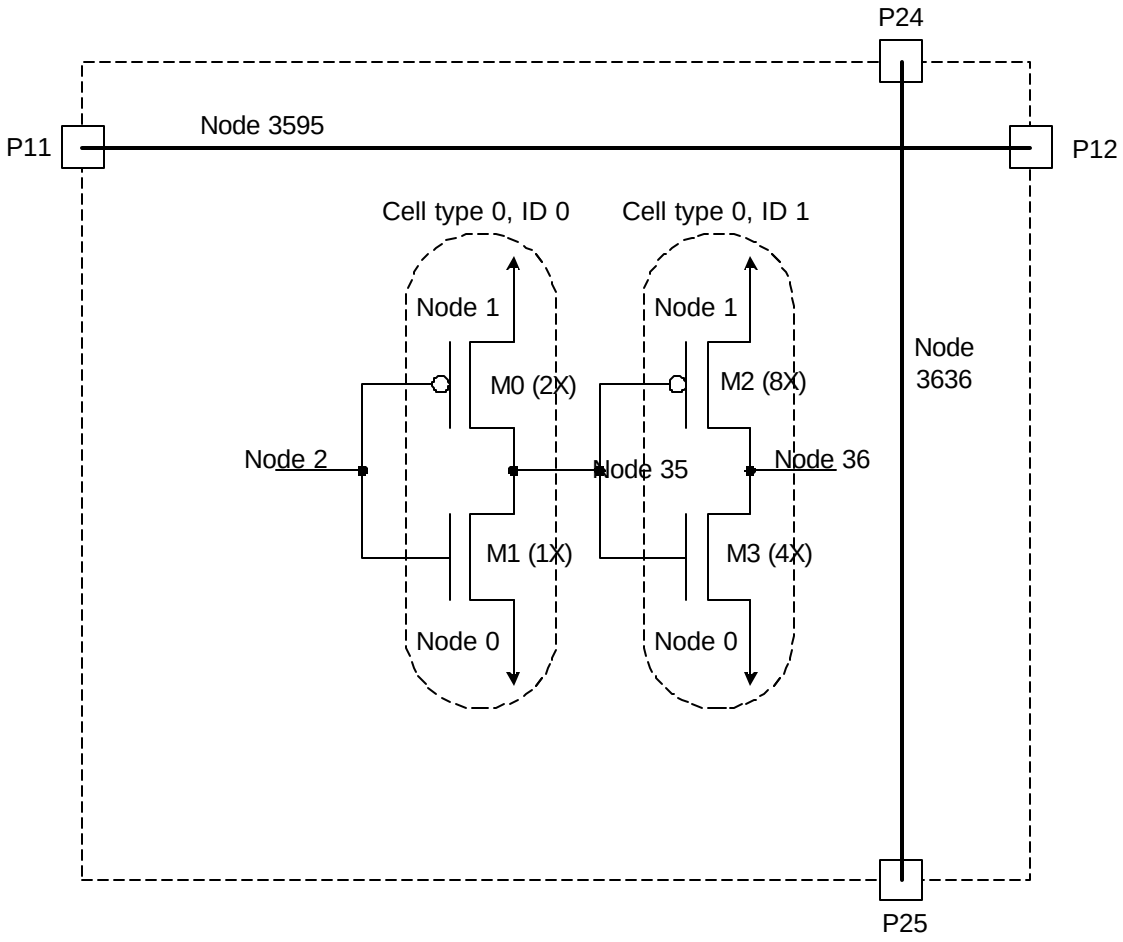


Figure 3.4 Tile portion generated by sample netlist in Figure 3.3

There are two types of blocks in our netlists – transistors and ports. The letter at the beginning of each line determines which type of block the line is describing ('M' for transistors and 'P' for ports). Transistors are used to implement the various structures that are necessary in our tiles, while ports are used to fix the locations that a signal enters or exits a tile.

Every transistor is described by an ID number, followed by three node numbers that represent the drain, gate, and source connections respectively. This is followed by the type of the transistor (N- or P-mos), and then the size of the transistor relative to the

minimum-width transistor of the process. The relative size was used in VPR's area model [1] to create greater process-independence and we continue that here. The definition of a minimum-width transistor is presented in Figure 4.2. Finally, each transistor has information about its role in the FPGA tile attached to it. The cell type and ID give the lowest-level hierarchy, and correspond to the cell-level netlist that is also output by VPR_Layout. The various types of cells currently used in VPR_Layout are shown in Table 3.1.

Cell type
Port
Buffer
SRAM
Multiplexer
Look-up table (LUT)
Flip-flop
Pass transistor routing switch
Buffered routing switch

Table 3.1 Cell types used in VPR_Layout

The subgroup and group fields are used to specify more hierarchy information, and the different values for these fields used in VPR_Layout are shown in Table 3.2.

Group type	**Subgroup types found in group**
Logic block	Logic block input circuitry
	Logic block LUT circuitry
	Logic block output circuitry
Routing switch block	Routing circuitry
Input connection block	Routing circuitry
Output connection block	Routing circuitry

Table 3.2 Group and subgroup types used in VPR_Layout

All of this hierarchy information is intended to help guide the automatic layout tool that will be discussed later in Chapter 4. The current types supported are arbitrarily chosen, and although we present some exploration about the information that is most useful in automatic layout of the tile, much research remains to be done to determine exactly what knowledge is useful to have in the netlist.

Besides transistors, our netlists include port blocks. Every port is described with an ID number, along with a node number indicating what node in the circuit that port is connecting to. The next two fields allow VPR_Layout to inform our automatic layout tool about any placement constraints that ports have.

The first of these is a constraint class number that, when applied to two ports, forces those ports to be placed opposite one another on the perimeter of the tile. The second field is a character specifying the edge of the tile that the port must be placed on ('T' for top, 'B' for bottom, 'L' for left, 'R' for right, and 'N' for no requirement). The result of using these constraints is shown in the sample netlist and the tile that it generates in Figure 3.3 and Figure 3.4. Ports need to be constrained using these values so that they lie on specific edges of the tile and so that they line up with other ports to make a tileable block. This subject is discussed in greater depth in Section 3.3.3

3.3.3 Tileability Constraints for Ports

The ports in our netlists need to have constraints specified so that when they are placed on the perimeter of the tile, the result is a tileable placement. The constraints involved for various connections can be derived from the boundary that is shown in

Figure 3.2. There are four classes of tileability concerns that need to be considered – routing wire tileability, routing switch block tileability, input connection block tileability, and output connection block tileability.

For routing wires, we need to ensure that the wires "twist" so that they each end at a switch block every L adjacent tiles, where L is the length of the wire. This can be accomplished by constraining the ports so that a wire exits the tile at a location exactly opposite to the one where the next wire enters it. Thus, when two tiles are placed adjacent to one another, the first wire in the first tile is connected to the same electrical node as the second wire in the second tile. Figure 3.5 illustrates one example of ports being placed exactly opposite one another on our netlist boundary to achieve this twisting effect.

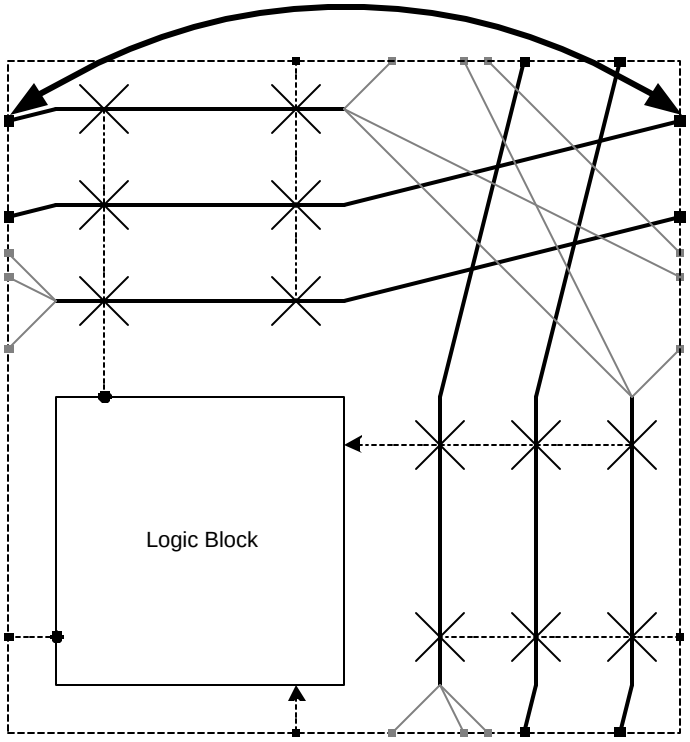


Figure 3.5 Ports lined up to create routing wire tileability

In order to create this twisting pattern successfully, we need to be able to find groups of L wires (where L is the length of *all* the wires in that group – 3 in the example of Figure 3.5). If we have such a group, one wire can start in this tile, twist and continue in the second adjacent tile, and ultimately end at a routing switch block in the "L^{th}" adjacent tile.

If there are many wires of length L (as is commonly the case in large FPGAs), they can be dealt with as long as the wires can be arranged into distinct groups of exactly L wires. Each of these groups then has the same twisting pattern described above.

Finally, if there are wires with different lengths in the tile, they can still be dealt with by the twisting method if they can be arranged into groups with other wires of the

same length. All such groups must have L_i wires when the length of the wires in group i is L_i.

These conditions, when combined, result in the following more compact requirement (mentioned in Section 4.2 of [1]) – the tileability constraint for routing wires can be met by the twisting method described above if, for all wire lengths L present in the tile, the number of wires of length L is an integer multiple of L.

When generating routing switch blocks, the netlist generator in VPR_Layout must create all the connections that are shown in the switch block of Figure 3.2. Because our netlist boundary includes only the routing wires to the right of and above the logic block, some switch block connections connect wires in our tile to wires in adjacent tiles. These connections must exit the tile at one edge (via a port), and enter the tile at the opposite edge. Figure 3.6 shows one example of ports lined up to create these types of connections.

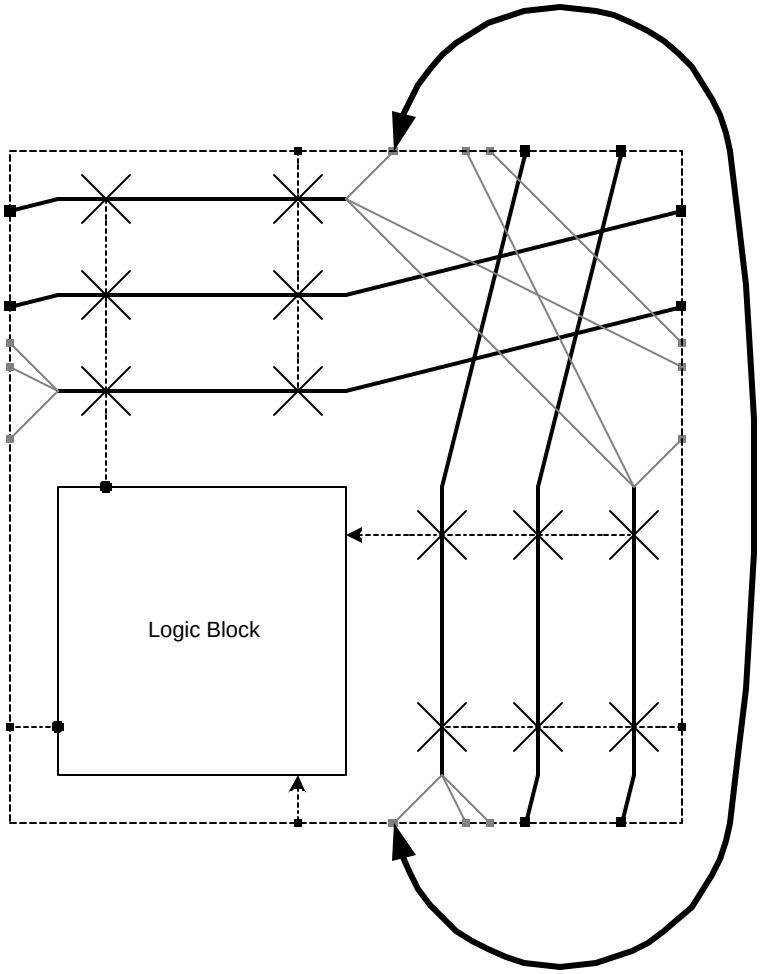


Figure 3.6 Ports lined up to create switch block tileability

The two ports indicated in the figure will create connections from a tile's horizontal wire to a vertical wire in the tile directly above it. This type of connection is used to make the connections that go from a given tile to the tile that is horizontally or vertically adjacent to it. However, there is the additional possibility of a diagonal connection that needs to connect a wire in our tile to another wire in the tile below and to the right of it.

We deal with this type of connection by extending the process used to deal with adjacent connections. As shown in Figure 3.7, the connection exits the tile on the bottom edge, enters at the top edge, exits again on the right edge, and finally enters to finish the connection on the left edge.

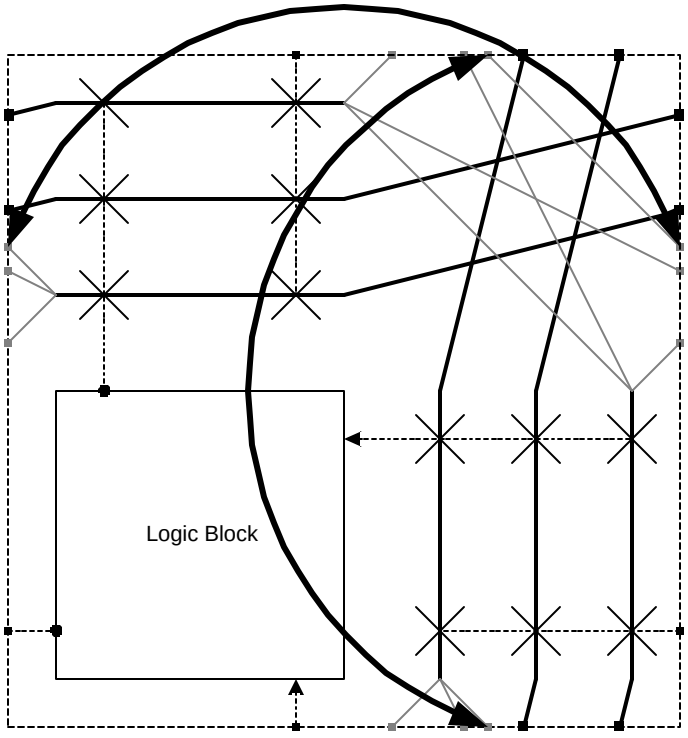


Figure 3.7 Ports lined up to create tileable, diagonal switch block connections

When the tiles are laid in an array, this will connect a wire in a tile to the tile below and to the right of it as required. The arrows in the figure indicate the four ports involved in making these types of connections.

Finally, to create tileable input connection block and output connection block connections, we need to allow for wires below and to the left of the logic block (which

are not included in the same tile as the logic block) to connect to logic block pins. This is equivalent to the wires in our tile connecting to pins of logic blocks in the tile above it and to the right of it. To create these connections, we again line up pairs of ports to allow connections leaving a tile to enter an identical tile placed adjacent to it. Figure 3.8 illustrates connections made in this way.

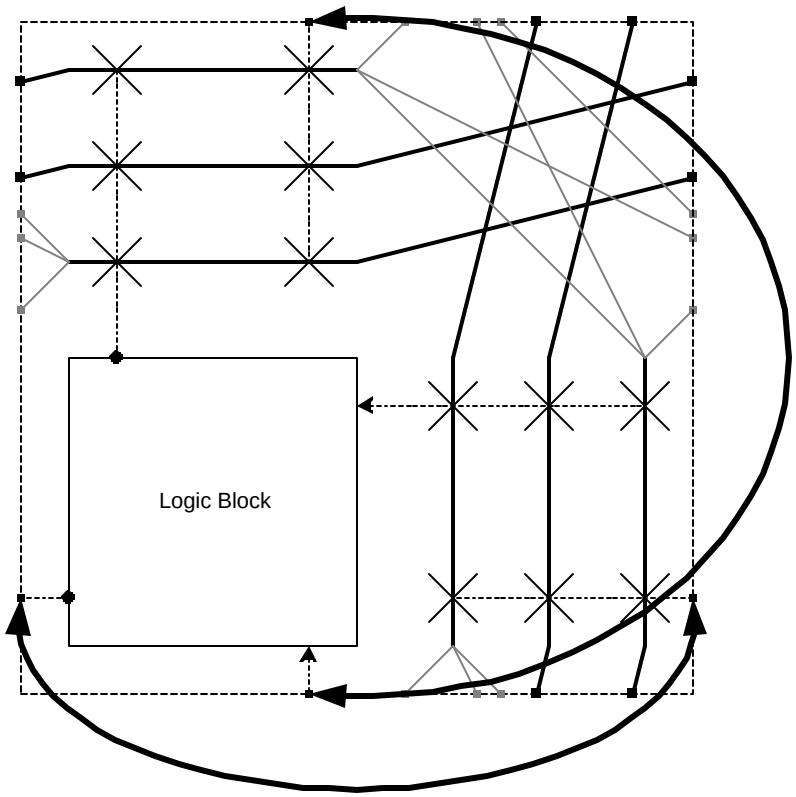


Figure 3.8 Ports lined up to create connection block tileability

The arrows in the figure show an output connection block connection (horizontal) and input connection block connection (vertical) being made to adjacent tiles such that a logic block has access to the wires on all four of its edges once the full array is replicated.

3.3.4 SRAM Programming

SRAM cells are a key component of our FPGAs because the values they are programmed with determine the circuit that the FPGA implements. In our tile netlists, each SRAM cell controls cells with its data value. However, the programming connections of an SRAM cell need to be made keeping in mind that the entire set of cells will eventually need to be treated as a memory array. Because of this requirement, the tile netlist generator automatically assigns programming lines to SRAM cells such that they form word lines and bit lines that can be used to access the entire array.

Our SRAM programming assignments are done so as to generate the same number of word and bit lines (resulting in a square memory array). The assignment of word and bit lines can be done in an arbitrary fashion since the assignment does not affect the FPGA's functionality. However, since our ultimate goal was to use our tile netlists to perform automatic layout while taking advantage of the domain-specific knowledge *explicitly* embedded in the netlist, we needed to make sure that no additional domain-specific knowledge was left *implicitly* in the netlist. If this occurred, our results would be affected by that hidden information.

This is an issue that must be considered when assigning the SRAM programming lines. For example, a 4-input LUT is driven by 16 SRAM cells (refer to Section 2.4 for schematics). Figure 3.9 shows a word line assignment that results in all 16 SRAM cells that are driving the LUT having the same word line.

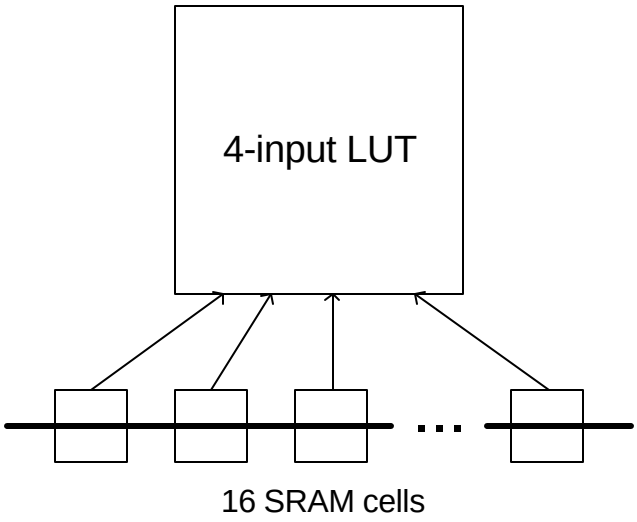


Figure 3.9 A 4-LUT driven by 16 SRAM cells on the same word line

If this were done systematically for all the SRAMs that drove LUTs, there would be implicit domain-specific knowledge in the netlist. Without explicitly stating so, the netlist has been designed such that the 16 SRAM cells driving a single LUT have more connections in common because they have a similar role in the context of the FPGA.

The solution to this problem is to assign all SRAM programming lines randomly, making sure that multiple SRAMs that might have been generated to control a given cell (like the LUT in the example above) do not systematically get placed onto the same programming lines.

3.4 Cell-level Netlist Generation

VPR_Layout generates the cell-level netlist at the same time as the transistor-level netlist. Since each transistor line in the transistor-level netlist specifies the cell that it is part of, this is the easiest way to obtain the abstracted cell-level netlist. Ports are not

specified in the cell-level netlist, because their specification is unchanged at the cell level (each port is a cell of its own). Thus, information about the ports comes only from the transistor-level netlist.

3.4.1 Cell-level Netlist Structure

A few sample lines from a typical cell-level netlist are shown in Figure 3.10.

```
# FPGA Tile cell-level netlist
# Output by VPR_Layout

# CELL Format: <id> <cell type> <subgroup type> <group type>
#              <width> <height> <num pins>
#              (pin class,node,x-offset,y-offset) (...) (...) etc
#               for "num pins" times

C0 0 0 0 3 2 2 (0 2 0 1) (1 8 2 1)
C1 0 0 0 5 4 2 (0 8 0 2) (1 9 4 2)
C2 0 0 0 3 2 2 (0 3 0 1) (1 10 2 1)
...
```

Figure 3.10 Sample section from a typical cell-level netlist output by VPR_Layout

The cell level netlist is made up only of cells, although each has parameters that allow us to differentiate between the cell types used in VPR_Layout.

The first field is an identifier, followed by three fields that give hierarchy info that was described in Section 3.3.2. The integer width and height (respectively) of the rectangular cell are next. The cell area determines the height and width of the cell, which is made as close to square as possible. The area is set based on the total transistor area required for the cell, with an extra 50% of that area added to allow for intra-cell routing. The value of 50% is an arbitrary value that can be explored in any future work that attempts to actually determine cell layouts.

The next field represents the number of pins that the cell has. These pins allow cells to connect to each other to form the FPGA tile. For each pin, a set of information about the pin forms the rest of the line, with a pin class, the node that the pin connects to, and the pin offsets relative to the lower-left corner of the cell all specified for each pin.

A more detailed look at using some of this information will be given in Chapter 4 and Chapter 5. Some of the fields, such as the pin offsets, have not been used in our work but would likely be necessary in any future work that attempted routing between cells or determining the layout inside individual cells.

Chapter 4

Automatic Cell Placement of FPGA Tile Netlists

This chapter describes the second phase of our research, which involves performing the automatic layout placement of the netlists output by VPR_Layout.

4.1 Goal

In this part of our work, our goal is to develop a tool to read in transistor-level and cell-level netlists of the form described in Section 3.3.2 and Section 3.4.1, respectively. Using these netlists, we want to generate good placements for the cell-level netlists.

4.2 CAD Flow

To meet the goal stated above, we developed a new tool called ATL (Automatic Tile Layout). It reads in netlists of the form output by VPR_Layout, and provides the infrastructure for obtaining cell-level placements and for using the domain-specific knowledge associated with the netlists to improve those placements. Figure 4.1 shows the CAD flow for ATL.

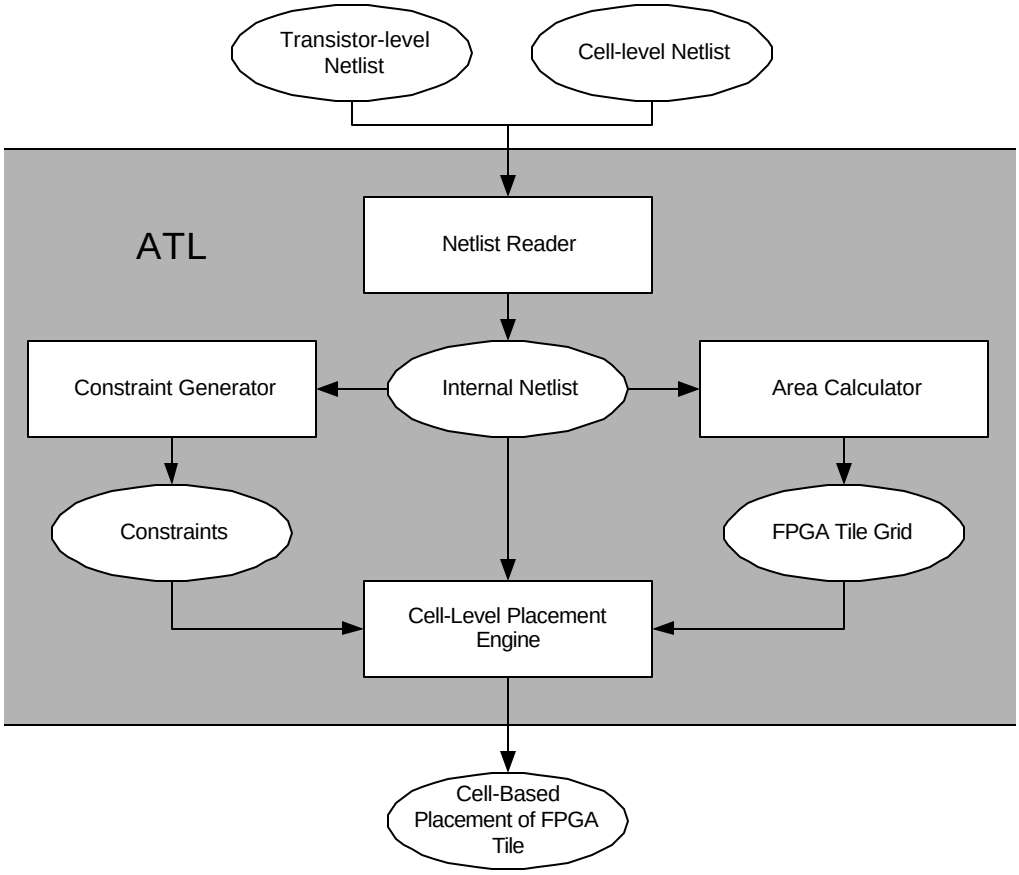


Figure 4.1 ATL CAD flow

4.3 Netlist Reader

The netlist reader parses in the transistor-level and cell-level netlists, and generates an internal netlist. Whereas the input netlists are specified in a spice-style format with node numbers, the internal netlist transforms them into a compact netlist that is more efficient for algorithms to operate on.

4.4 Area Calculator

The area calculator uses the information in the internal netlists to determine the size of the tile used for placement. We have assumed square tiles in our work, though such an assumption is arbitrary and rectangular dimensions can easily be explored in future work.

To determine the area, the area calculator uses the following equations:

$$width(side) = \max(\sqrt{total\_cell\_area} * AREA\_FUDGE\_FACTOR,\ num\_ports(side))$$
$$width = \max(width(left),\ width(right),\ width(top),\ width(bottom))$$
$$height = width$$

The width and height are set to the maximum required by either the total cell area or the ports that lie on the edge of the tile assigned the most ports. The AREA_FUDGE_FACTOR in the equation above is used to allow extra space when the tile area is determined by cell area. This space is needed to allow cells to be able to move around and to allow space for routing. Since the scope of this work did not involve routing, though, an arbitrary value of 1.4 is used for this factor. This is clearly a value that needs to be explored in any future work that attempts to route our placements.

Once the area of the tile has been calculated, the grid that represents the FPGA tile is fixed to that area. The area is specified in units of "minimum-width transistor areas", and each square in the placement grid represents one minimum-width transistor area. The definition of a minimum-width transistor area is shown in Figure 4.2.

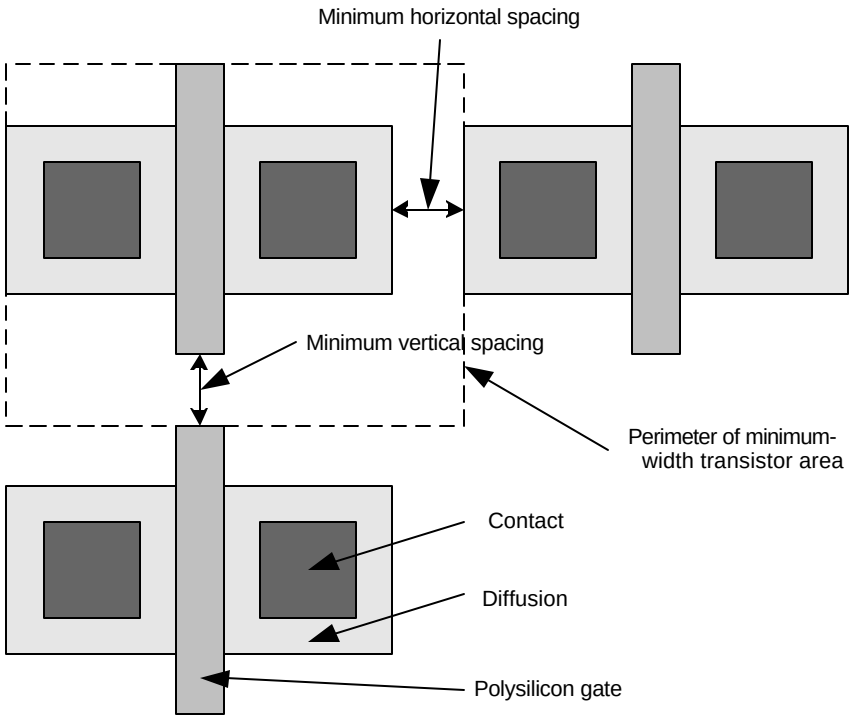


Figure 4.2 Definition of a minimum-width transistor area – adapted from [1]

4.5 Constraint Generator

The constraint generator is the portion of ATL that we used to demonstrate the value of using domain-specific knowledge in generating the placements of an FPGA tile. In particular, it uses the hierarchy information provided in the cell-level netlist to determine ways of improving the placements.

One of the features provided by the constraint generator is the ability to specify a rectangular region in which a cell must be placed. Every cell can be given such a placement constraint.

This constraint generator is one tool that can help leverage the domain-specific knowledge to floorplan groups of cells into certain areas in order to improve the layout.

4.6 Placement Engine

The placement engine in ATL is based on the simulated annealing algorithm described in Chapter 2. The details of our placement approach largely follow the one presented in [1]. Some of the key features are described in the sections below.

4.6.1 Initial Placement

The initial placement that is used by our placement engine is generated in a random fashion. However, because the cells we are placing are all of different sizes, our initial placement places cells in order of their size. With larger cells placed first, the smaller cells can "fit in the gaps" much more easily.

Another challenge is presented by the presence of placement constraints of the type described in Section 4.5. If cells that are unconstrained are placed before cells that are constrained, the limited space in which the constrained cells can be placed are often already taken by cells that do not need to be there. To handle this, we use an initial placement algorithm that makes repeated initial placement attempts, with each attempt first placing the cells that failed in the previous attempt.

4.6.2 Cost Function

One of the most important factors in an annealing-based placement algorithm is the cost function that is used to evaluate changes to the initial placement. Our cost

function is based on the bounding box enclosing the terminals of each net, and is calculated according to the formula below [1]:

$$Cost = \sum_{inet=1}^{\#nets} q(inet)\left[bb_x(inet) + bb_y(inet)\right]$$

The bb_x and bb_y values are determined based on the smallest box that encloses the lower-left corner of all the terminals of a net. The q(inet) factor is used to model the fact that the bounding box usually underestimates the amount of routing needed for nets with more than 3 terminals [6]. The value of q(inet) is determined based on [6]: it is 1 for nets with 3 or fewer terminals, and linearly increases to 2.79 for nets with 50 or more terminals. An example of a bounding box cost calculation for one net is shown in Figure 4.3.

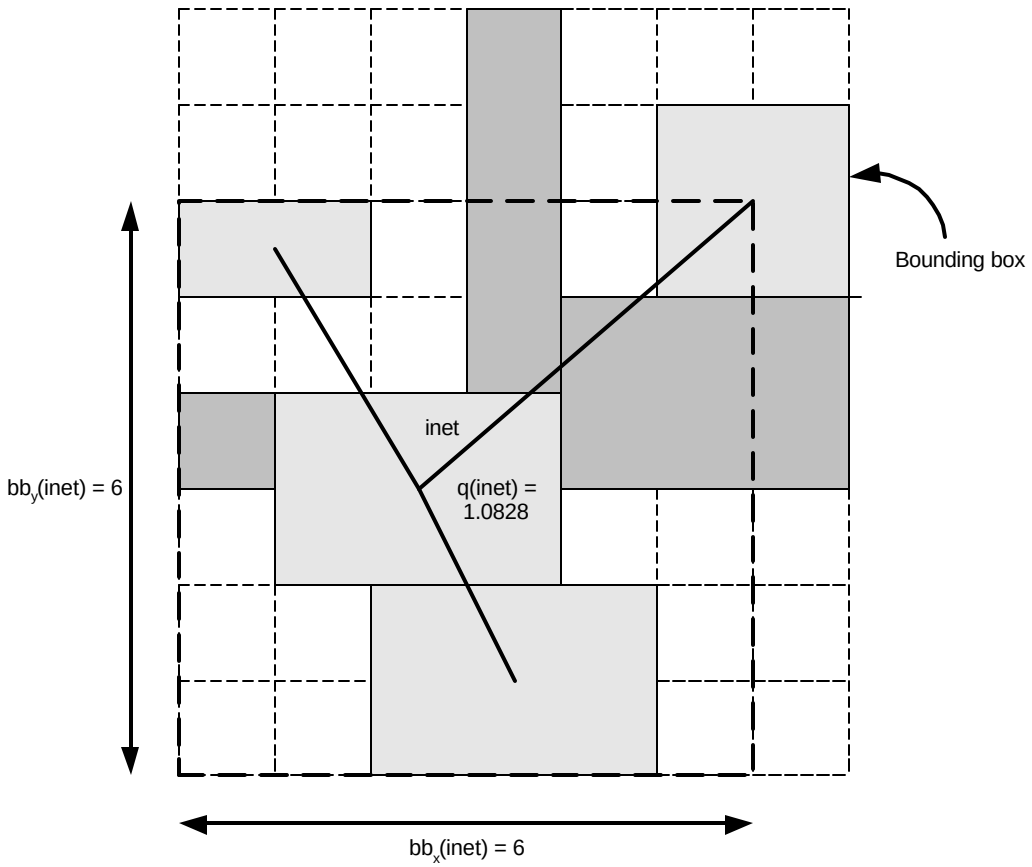


Figure 4.3 Example bounding box cost calculation for one net

4.6.3 Annealing Schedule

We use the same adaptive annealing schedule that is presented in [1], which includes the temperature update scheme, the range limiting scheme, and the exit criterion. A detailed explanation of these methods can be found in Chapter 3 of [1].

The number of moves attempted per temperature is also set to the same value as [1]:

$$moves\_per\_temperature = inner\_num * (num\_cells)^{\frac{4}{3}}$$

The "inner_num" factor is a user-specified option that is directly proportional to the amount of CPU time spent in the placement tool. This option allows the user to explore the time-quality trade-off for placements.

4.6.4 Move Generation

Because of the varied size of the cells in our tiles, the move generator in our placement engine must be careful to swap cells such that the legality of the placement is maintained. This means that there must be no overlap of cells in the final placement. As described in Section 2.5, many placement algorithms attempt to achieve this by applying an increasing overlap penalty, followed by a phase that fixes any remaining overlap.

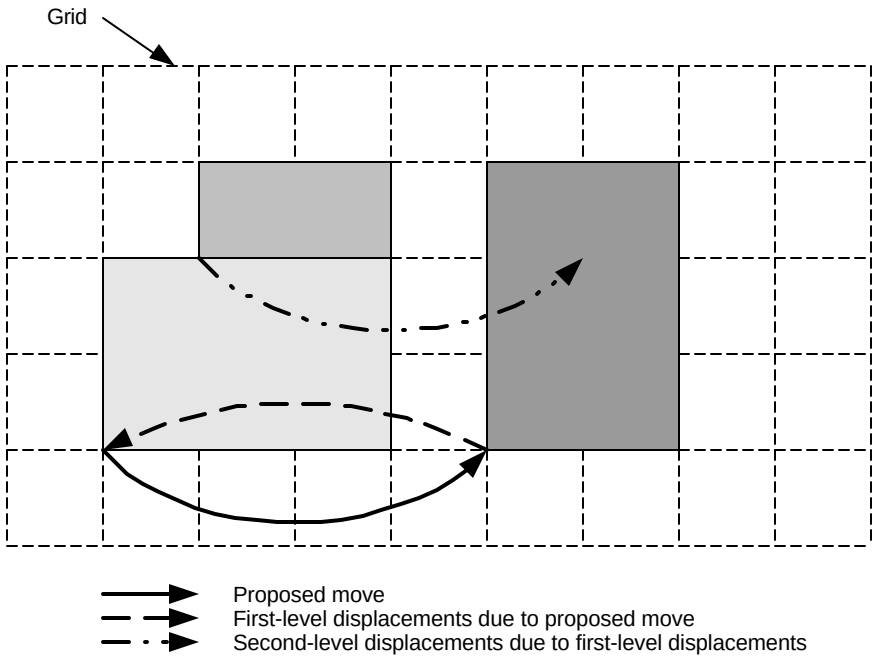
Our placement algorithm disallows overlap at any stage in the placement, ensuring that the algorithm is always working with a legal placement throughout the entire process. Determining whether allowing overlap would work better for our application is beyond the scope of our work, but could be explored in future work.

To prevent overlap, we use a move checking procedure that forbids all moves that would create overlap if accepted. The basic rule is that when a cell is moved, all cells that it displaces (which are swapped to the area that the original cell is leaving) must either fit into the space left by the original cell, or if they displace further cells, those newly displaced cells must fit into the space left by the cells displacing them. Figure 4.4 shows pseudo-code that describes the move checking procedure.

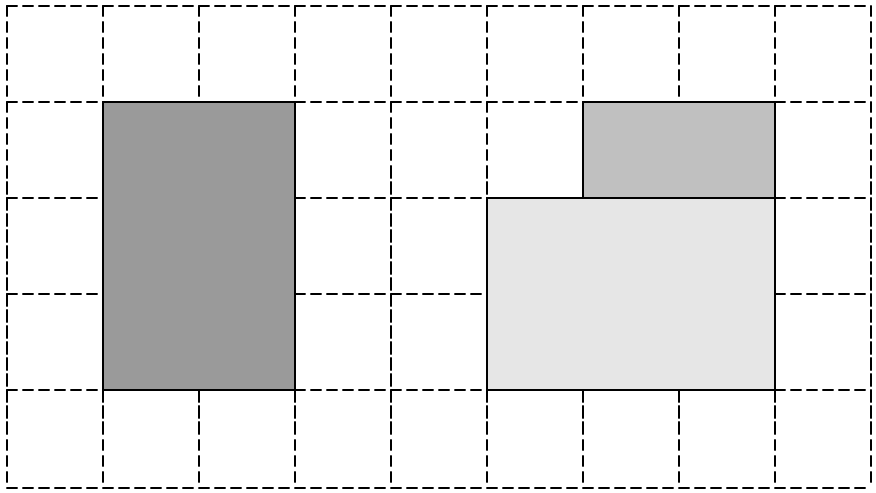
```
move_allowed = 1; /* Assume okay until problem found */
target_cell = select_random_cell();
(x_to, y_to) = select_random_target_location();
for (icell = cells target_cell will be displacing) {
  if (icell fits once target_cell has left) {
    /* Does not make move illegal */
  }
  else {
    for (blocking_cell = cells icell will be displacing) {
      if (blocking_cell fits into space left by icell) {
        /* Does not make move illegal */
      }
      else {
        move_allowed = 0; /* Forbid this move! */
      }
    }
  }
}
```

Figure 4.4 Pseudo-code describing move checking procedure

Figure 4.5 shows an example of a legal move. The move of the lower-left cell results in a "first-level" displacement of the cell on the right. That displacement in turn results in a "second-level" displacement. However, because this second displacement does not result in any further displacements (the top-left cell is moving such that it fits into the space left by the cell on the right), this move is allowed by ATL. The figure also shows the resulting placement after the move has been made.



(a) Proposed move and consequences



(b) Result after move

Figure 4.5 A legal move in ATL – (a) The move and its consequences, and (b) The resulting placement after the move

An example of a move that ATL rejects is shown in Figure 4.6. Now the cell on the top-left is a second-level displacement that does not fit into the space left by the block

displacing it (i.e. the cell on the right). This could result in further displacements and so ATL declares the move illegal. Notice that in this particular case, the move would be legal if ATL allowed it because no other cell occupies the space above the cell on the right and hence no further displacements would actually be required.

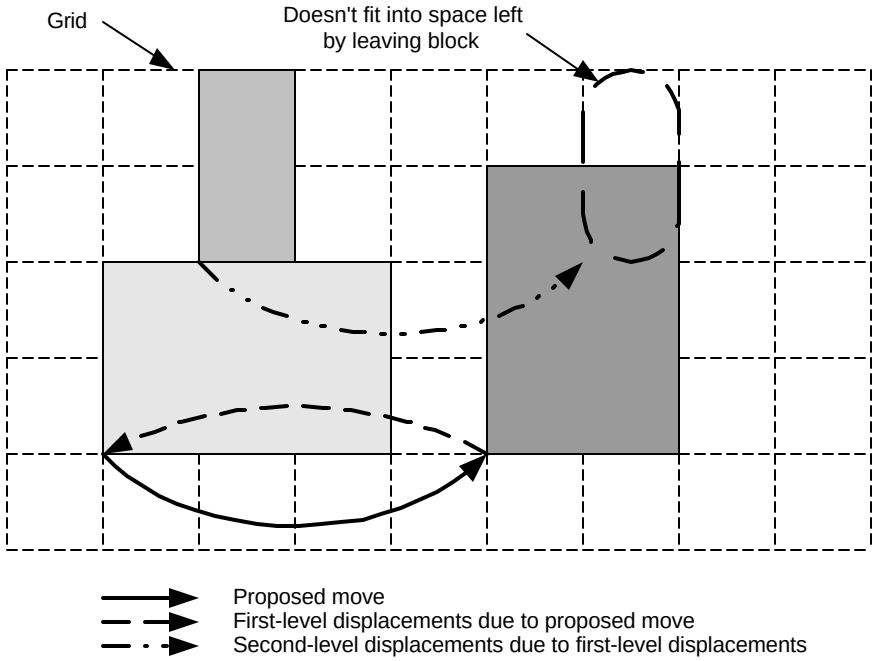


Figure 4.6 An illegal move in ATL – this type of move will be rejected

4.7 Placer Quality

In the absence of a router that can determine whether the cell placements we create are actually routable, it is difficult to determine the quality of our placement tool with respect to our ultimate goal of creating an FPGA tile layout.

We believe that our placement tool is of sufficient quality to allow exploration of using domain-specific knowledge to improve FPGA layouts. However, a complete layout tool is clearly the most pressing need for future work to try and meet.

Chapter 5

Using Domain-Specific Knowledge to Improve Cell Placements

The work described in this chapter uses the infrastructure provided by the tools presented in Chapter 3 and Chapter 4 to evaluate the benefit of using domain-specific knowledge to improve cell-level placement of FPGA tiles.

In this dissertation, we present only one particular application of domain-specific knowledge. Exploring more ways of using this information is an obvious area requiring much further work.

5.1 Experimental Methodology

All of our experiments used a set of ten "benchmark tiles" that were generated using VPR_Layout. These benchmark tiles were formed with architecture files that have been found to result in high-quality FPGAs in [1]. The electrical parameters in the architecture files, used to size buffers and switches in our tiles, were based on TSMC's 0.18µm process technology and were taken from the work done in [9]. To obtain the set of tiles that we used in our experiments, we varied the number of LUTs from 1 to 10 in the architecture files, and ran each architecture file through VPR_Layout to obtain the tile netlists.

In our experiments, we run ATL with all ten tile netlists and calculate the geometric average of the final placement costs for those netlists. We use geometric averages to make sure that all tiles have equal weight in the average; an arithmetic average would give greater weight to the larger tiles that have more cells and hence higher costs. Our experiments compare these geometric averages to evaluate the effects of optimizations.

5.2 SRAM Placement

5.2.1 SRAMs in our Tile Netlists

SRAM cells are one of the most important components of an FPGA because they provide the programmability of its routing and logic components. Section 3.3.4 describes our method of assigning word and bit lines to SRAM cells. This method results in a netlist that has groups of SRAM cells that the placement cost function will prefer to put together (because they share the same word or bit lines) even though they might not have anything else in common.

In an FPGA, though, the choice of word and bit lines is arbitrary, and can be done in any way that reduces layout complexity. For example, groups of SRAM cells that are close together and connected to the same blocks can be put onto common word or bit lines to make the layout simpler.

This is an example of domain-specific knowledge that we have about our netlists. We know that FPGA SRAM cells in our netlist are assigned arbitrarily to word and bit

lines, and that those assignments can be swapped with other SRAM cells without any consequence.

5.2.2 SRAM Regularization

To improve our layouts by using the netlist properties described in Section 5.2.1, we perform "SRAM regularization", a process that is described in this section.

First, we fix the locations of all SRAM cells based on their arbitrary programming line assignments in the netlist. This results in a square array of fixed SRAM cells, with all the cells in a row connected to the same word line, and all the cells in a column connected to the same bit line. This SRAM placement has a very low placement cost for its programming nets, since the SRAM cells are all lined up according to those nets. However, since the SRAMs sharing a given word or bit line do not necessarily have anything in common, this SRAM placement makes it hard to bring cells that are driven by multiple SRAM cells close to the SRAMs that drive them (since those SRAMs are likely to be on separate sets of programming lines). Table 5.1 shows that if we keep these regularized locations for the SRAMs (by locking them down throughout the annealing process), the final placement is much worse (by more than 30%) than the original algorithm that treats SRAMs like all other cells.

Geometric Average Wirelength for 10 tiles	
Normal Placement	Regularized SRAM Placement with SRAM cells locked to initial locations
94 062	124 361

Table 5.1 Comparison of normal placement and regularized SRAM placement with locked SRAM cells

Once we have placed the SRAMs into an array of cells, however, we can allow them to swap with other SRAM cells without affecting the extremely low cost for programming nets. This is because of the domain-specific property described in 5.2.1. If two SRAM cells from different rows swap locations, for example, the netlist indicates that the word lines for both of them will need to be routed out to the new locations instead of in a straight line as before. We know, however, that the SRAM cells can just swap the word line they are using so that those word lines can still run in straight lines. An example of this strategy is shown in Figure 5.1.

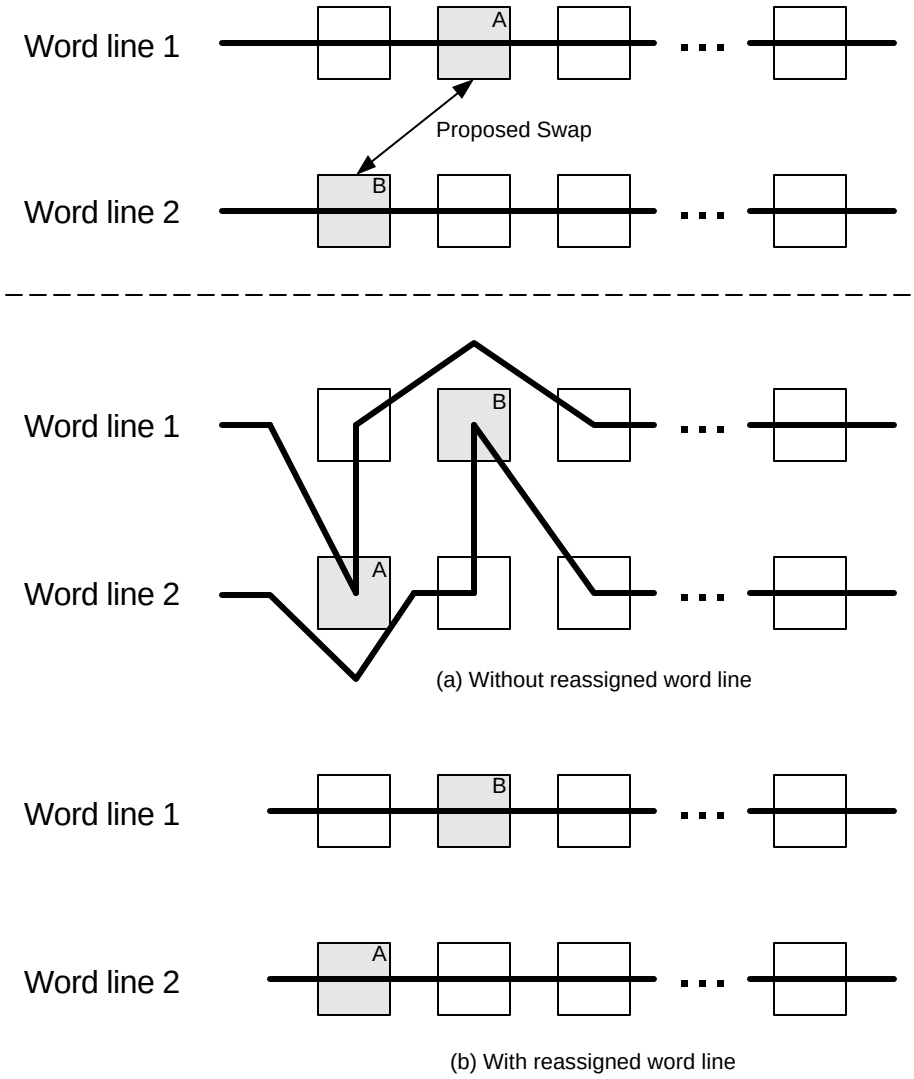


Figure 5.1 Effect of an SRAM cell swap on word lines (a) without reassignment and (b) with reassignment

We tested this procedure by directing the placement tool to ignore the programming line costs as it evaluated moves involving SRAMs (instead of continuously reassigning programming lines in the netlist). We then add back the cost of the original programming nets (i.e. the ones running in straight lines) to the final cost using this method. The result of this process is shown in Table 5.2.

Geometric Average Wirelength for 10 tiles	
Normal Placement	Regularized SRAM Placement with SRAM cells allowed to move
94 062	90 259

Table 5.2 Comparison of normal placement and regularized SRAM placement with SRAM cells allowed to move without penalizing programming connections

Regularizing SRAMs in this fashion provides an improvement in the wirelength of about 4%.

Chapter 6

Conclusions and Future Work

6.1 Summary

The main contribution of this research is to provide an infrastructure that allows simple architectural descriptions to be turned into the detailed transistor-level design of an FPGA tile, and that begins the process of automatically laying out that design. We have also shown that taking advantage of domain-specific knowledge can help create better layout placements, and can potentially reduce the penalties associated with performing layout automatically.

6.2 Future Work

This research has opened up many promising avenues of further work. The largest of these is to finish the task of automatically laying out our tiles by developing a router for connections between cells, and by developing a tool to automatically determine the layout inside the cells themselves. Clock, power, and other special nets should also be handled in an appropriate fashion.

Achieving a complete layout would determine what the performance and area penalties are likely to be for laying out FPGAs automatically versus manually. It would

also allow the layout to be extracted and simulated for more detailed verification of performance and functionality.

Using a completed automatic layout engine, the transistor-level structures we used in our work could be modified or optimized to result in better layouts. Also, further exploration of using domain-specific knowledge to improve those complete layouts would help determine how much of the area and performance penalty can be removed by a smarter layout tool.

Future work could also take the spice-style netlists that we produce with VPR_Layout and simulate them to ensure functionality and to get an idea of how performance of various structures in the tile is affected by simulating the entire tile as a whole.

Appendix A

Graphical Representation of Tiles in ATL

ATL includes a graphical tool that displays the cells and cell-level nets of an FPGA tile on the screen. This tool is an extended version of the one first presented in [2], and was ported to Windows as part of the work done in [7] and [8].

This tool is a particularly useful aid in visualizing the contents of the tile, as well as the interactions between various types of cells. Table A.1 provides a legend for the cell colours by cell type, which is needed to interpret the graphical output of ATL.

Cell type	Colour
Buffer	Dark green
SRAM	Red
Multiplexer	Yellow
Look-up Table	Magenta
Flip-flop	Cyan
Pass transistor routing switch	Dark grey
Buffered routing switch	Light grey
Port	White

Table A.1 Legend of cell colours in ATL graphical representation

The following pages show some screen captures of our graphical tool in various stages of placement and with different optimization options.

REFERENCES

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

[2] V. Betz, "Architecture and CAD for Speed and Area Optimization of FPGAs", *Ph.D. Thesis*, University of Toronto, 1998.

[3] V. Betz and J. Rose, "VPR: A New Packing, Placement and Routing Tool for FPGA Research", *Int. Workshop on Field-Programmable Logic and Applications*, 1997, pp. 213 - 222.

[4] W.P. Swartz Jr., "Automatic Layout of Analog and Digital Mixed Macro/Standard Cell Integrated Circuits", *Ph.D. Thesis*, Yale University, 1993.

[5] F. Najm, VLSI Systems, *Course Notes*, University of Toronto, 2001.

[6] C. Cheng, "RISA: Accurate and Efficient Placement Routability Modeling", ICCAD, 1994, pp. 690-695.

[7] P. Leventis, "Placement Algorithms and Routing Architecture for Long-Line Based FPGAs", *Undergraduate Thesis*, University of Toronto, 1999.

[8] W. Chow, *M.A.Sc. Thesis*, University of Toronto, In Preparation.

[9] E. Ahmed, "The Effect of Logic Block Granularity on Deep-Submicron FPGA Performance and Density", *M.A.Sc. Thesis*, University of Toronto, 2001.