The Edward S. Rogers Sr. Dept of Electrical and Computer Engineering
University of Toronto

Final Report

Title: **Automatic Transistor-Level Design and Layout of FPGAs**

Project I.D. # 1092001

Prepared by:	Mark Bourgeault (bourgea@ecf.toronto.edu) Joshua Slavkin (slavkin@ecf.toronto.edu) Chris Sun (suny@ecf.toronto.edu)
Supervisor:	Prof. J.S. Rose
Section #:	5
Section Coordinator:	R. Gillett
Date:	Friday, April 12, 2002

Executive Summary

Field-Programmable Gate Arrays (FPGAs) are becoming more prevalent in digital systems and are used to implement a wide range of applications – from telecommunications switching systems to wireless interfaces. This project investigates the feasibility of a tool to automate the process of producing the transistor-level layout of an FPGA. This will reduce the design cycle time while maintaining a comparable quality to layouts fully optimized by hand. We will extend an existing tool, Automated Transistor Layout (ATL), by improving its placement algorithm, designing a router, and developing a Graphical User Interface to visualize the proposed enhancements. This project takes ATL from cell placements without routing between them and defines the inter-cell routing and the intra-cell layout. Our work coincides with another project is further improving the cell placement algorithms.

Team Members' Contributions

Table 1 contains a listing of the project milestones that were defined at the beginning of our project. The table provides a short description of each task, identifies the primary group member responsible for completing the module, and our initial estimates for the duration of each milestone.

Task	Duration	Task Lead
Background Research	September 4 – September 30	ALL
Technical Proposal	September 20 – September 30	ALL
Routing Infrastructure (Inter-cell)	October 1 – December 31	Mark
Graphical Infrastructure	October 1 – December 31	Chris
Support For Hand Generated Placement (Intra-cell)	October 1 – December 31	Josh
Router Algorithm (Inter-cell)	January 1 – March 31	Mark
Placer & Router Feedback Loop	January 1 – January 31	Josh
Graphical Routing Editor	January 1 – March 31	Chris
Auto-Generated Placement (Intra-cell)	February 1 – March 31	Josh

Table 1: Original Milestones for Design Project

Over the course of the project's lifecycle, both the milestones and their durations were updated based on inaccurate estimates in the amount of effort and for unexpected difficulties. Table 2 lists the updated milestone schedule and contains information as to the final status for each of the milestones at the conclusion of the project. We have completed all of our original milestones, except for a module that automatically generates the intra-cell layout. The primary reason for abandoning this milestone was that preliminary estimates for the performance of an automated intra-cell layout engine would be significantly inferior to a hand-generated solution. Additionally, we have

underestimated the total time taken to fulfill the non-technical requirements for the design project (i.e. presentations & reports).

Task	Duration	Task Lead	Status
Background Research	September 4 – September 30	ALL	☑
Technical Proposal	September 20 – September 30	ALL	☑
Inter-cell Routing Infrastructure (software)	October 1 – December 31	Mark	☑
Graphical Infrastructure (software)	October 1 – December 31	Chris	☑
Hand Development of Intra-cell Layouts (hardware schematics)	October 1 – December 31	Josh	☑
Interim Reports	January 1 – January 10	ALL	☑
Placer & Router Feedback Loop (software)	January 1 – January 15	Mark	☑
Inter-cell Router Algorithm Improvements (software)	January 16 – March 31	Mark	☑
Graphical Routing Editor (software)	January 1 – March 31	Chris	☑
CAD Development of Intra-cell Layout (hardware schematics & software)	February 1 – March 31	Josh	☑
Integration & Test	March 1 – April 4	ALL	☑
Poster Presentation	March 14 – March 21	ALL	☑
Oral Presentation	March 21 – April 4	ALL	☑
Final Tuning of Inter-cell Router (experimentation)	April 1 – April 12	Mark	☑
Final Report	April 1 – April 12	ALL	☑

Table 2: Final Milestones for Design Project

As shown in the table, most of the technical components were software-oriented. Chris and Mark exclusively worked on developing software for our CAD tool, while Josh designed several hardware schematics and wrote a software module to transfer the schematic representation into the CAD tool's internal data structures.

The initial partitioning of the final report write-up was based on the technical components that each individual worked on over the course of the year. The remaining sections were allocated evenly between the three of us, based on the comments that we received in our interim reports. Table 3 lists the individual responsible for each section of the final report write-up:

Report Section	Author
Report Outline	Mark Bourgeault
Executive Summary	Josh Slavkin
Team Members' Contributions	Mark Bourgeault
Introduction	Josh Slavkin
Background Work	Chris Sun
Intra-cell Layout	Josh Slavkin
Inter-cell Routing	Mark Bourgeault
Graphical User Interface	Chris Sun
Conclusions & Future Work	Mark Bourgeault
Final Proofing	Josh Slavkin

Table 3: Division of Final Report Write-up Responsibilities

Table of Contents

List of Figures

List of Equations

List of Tables

1 Introduction

Field-Programmable Gate Arrays (FPGAs) are becoming more prevalent in digital systems and have a wide range of applications, ranging from telecommunications switching systems to wireless interfaces. Currently, the transistor-level design and layout of an FPGA is a manual process, with the assistance of Computer Sided Design (CAD) tools, and takes many person-years of effort. This project enhances an existing tool that automates the process of producing the transistor-level design of an FPGA in order to reduce the design cycle time for an FPGA layout to days instead of several months.

One key question our team attempts to answer is how well an automated solution will compare to standard industry results produced by a team of designers. Specifically, we compare the area needed to implement the design in silicon. If our tool produces reasonable results, then it could be used to assist FPGA architects in refining the underlying architecture of FPGAs to produce future devices more rapidly.

Therefore, the main goal of our project is to reduce the length of design time for the development of FPGA layouts while maintaining the quality of results of achieved by current FPGA architects.

1.1 FPGA Layout & Design Considerations

An FPGA is an integrated circuit that allows routing paths to be reconfigured after fabrication. Like any integrated circuit, for an FPGA to be created the complete

transistor-level structure and interconnections must be defined; this is the design phase of creating an FPGA. The main consideration at this stage is the functional correctness of the circuit or sub-circuits. Additional considerations include estimating the transistor sizes to provide some speed and timing optimizations. Once the design is complete, the circuit must be physically laid out so a mask can be created to fabricate the actual silicon implementation of the FPGA. Figure 1 shows the high-level view of the FPGA creation process.

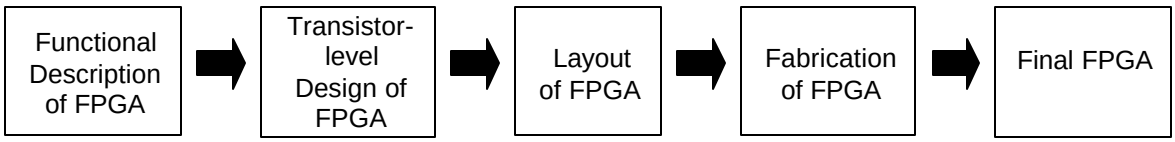


Figure 1: FPGA creation process overview

Determining the exact locations of the silicon representation of each transistor and piece of metal interconnect is the layout phase of creating an FPGA. The main consideration at this stage of development is producing a layout that functionally matches the design. Additionally, the layout created should produce good yields from fabrication. Design rules specify the geometric properties the layout should have to maximize the yield for an integrated circuit from a given fabrication process. Thus, conforming to the design rules specified for a fabrication process is essential to produce sufficient quantities of the FPGA to make the expense of fabrication worthwhile.

Matching the functionality of the design and meeting the design rules are not the only considerations at this stage. Additional considerations include minimizing area and maximizing timing performance. Because these two considerations can be at odds with

each other, a balance between the two must be reached. However, as long as minimum timing requirements are met, area considerations usually take precedence..

Our project deals primarily with the layout portion of the FPGA creation process. The main considerations while attempting to first minimize the area required to layout an FPGA and then optimize the timing performance.

1.2 Motivation

FPGAs are increasingly important components in the design of digital systems. But before any digital system can be implemented upon an FPGA, the layout and design of the FPGA must be defined at the transistor-level. The process of FPGA design and layout is a task that currently requires several person-years worth of work. Existing tools can assist in this process; VPR_LAYOUT converts architecture description files into cell-level and transistor-level netlists for FPGA tile and the previous generation of ATL addresses the initial placement of the cells. However, the process is still largely manual. This generation of ATL further automates the layout and design process. The three main benefits our tool attempts to provide to FPGA designers are:

- Reduction in design cycle time
- Architecture exploration experiments
- Preliminary quality assessments of architectures

1.3 Approach

Our group defined three main tasks to perform in the attempt to achieve our main goal. They are:

- The creation of an inter-cell routing algorithm that selects the width, layer, and position of each metal component that is required to implement a portion of an FPGA on an integrated circuit.
- The definition of a compact layout for each type of cell that appears on an FPGA. A "cell" is a group of transistors that performs a specific digital logic function.
- The development of a graphical user interface that can visualize the efforts of the two previous tasks and accept user input to modify the final layout selected by the routing engine.

Our design project is an extension of ATL [2], a tool that performs the Automated Transistor Layout for a representation of an FPGA segment, hereafter referred to as an FPGA "tile". This software application contains the foundation for achieving the goal of automating the development of FPGA layouts. The previous version of ATL takes the required connectivity of the FPGA tile and then positions the cells such that the anticipated number of wires required to electrically connect the various terminals of the FPGA's transistors is minimized. As already mentioned, our design project will augment ATL with an "intra-cell layout mechanism" and an "inter-cell routing module". These modules will transform ATL into a tool that is closer to creating a viable electrical representation of the FPGA tile so that the output of ATL can be used to produce a

functionally correct integrated circuit that represents the architectural description of the tile. Figure 2contains a pictorial summary of the process tasks that must occur prior to ATL's execution. These steps include the definition of the architecture and creation of the netlist for the FPGA tile and the layouts for each cell. Within the figure, the rectangles represent a process or task to be performed and trapezoids represent the resulting data. Processes that are not greyed out are those we implemented. The computer/person icon reflects the level of computerized tools associated with the process.

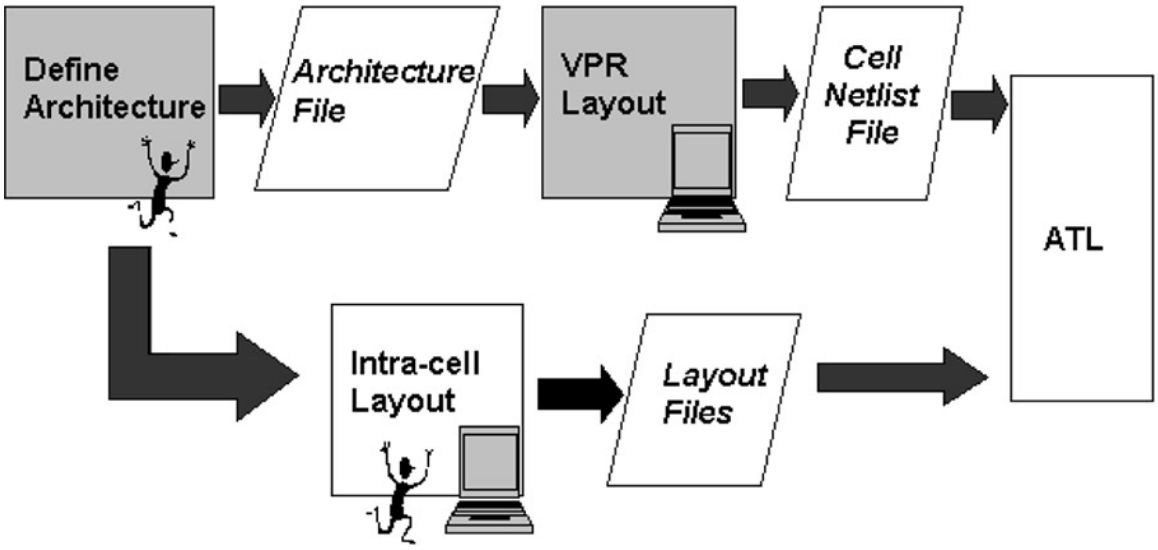


Figure 2: Pre-ATL process flow

Figure 3 is a block diagram of the data flow within ATL. In this figure, rectangles represent functional modules with ATL; those not greyed out are modules our group implemented. The trapezoids represent the input and output data files.

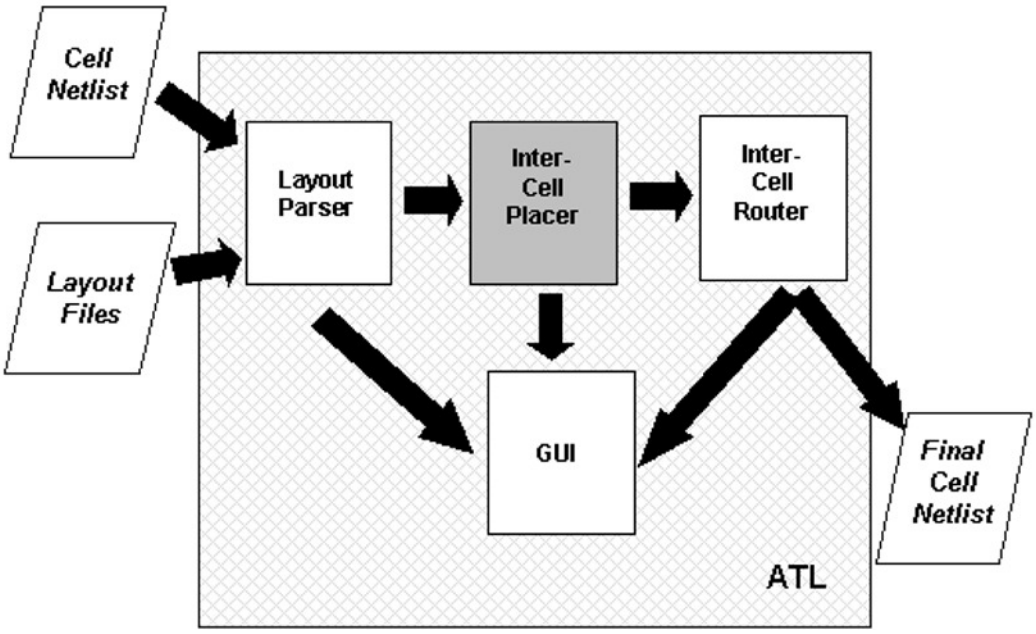


Figure 3: Internal ATL control flow

1.4 Report Organization

The next section of this report provides some background information on the basics of FPGA architecture, the basic Computer Aided Design flow for VLSI designs and an overview of the previous work necessary for this project to be successful.

Sections 3 through 5 provide details about the overall methodology used within the ATL flow. Section 3 outlines the rationale and methodology used in the intra-cell layout process that takes place outside of ATL. Section 4 presents the details of the inter-cell routing to connect the cells. Section 5 provides details on the Graphical User Interface that was created to assist in the use of ATL

Section 6 reports on the conclusions drawn from this work and presents some potential future work that may extend this project. The final section summarizes the work done in this project and draws conclusion this work can provide.

Appendix A contains the schematic and layout library of the cells used in the FPGA architectures considered.

1.5 Acknowledgements

The authors are indebted to Jonathan Rose, our project supervisor, whose guidance and unlimited enthusiasm for this work has been most helpful and inspiring. We would also like to thank several members in the FPGA community – Vaughn Betz, David Lewis, Jordan Swartz, and David Galloway – for the many valuable suggestions and insightful discussions they have provided over the scope of this project. Special thanks are due to Vaughn Betz who supplied considerable direction in the design of the routing algorithm.

Ketan Padalia, the pioneer of the ATL application, deserves special credit for his helpfulness and the time he spent in the explanation of several subtle issues involving the ATL code base. Finally, we appreciate the work of our colleague, Ryan Fung, who has been working concurrently on refining the inter-cell placement algorithm and for engaging in continual debates to improve the quality of this project.

2 Background Work

2.1 FPGA structure

The structure of an FPGA can be viewed from 2 perspectives, the user-design view and the Integrated Circuit (IC) design view. The IC design view looks at the chip at the transistor level while the user-design view is a higher-level, abstraction of the chip. In both cases, common terminology is used. For this document, some terminology is now presented.

Term	**Definition/Use**
Point	Transistor port, cell port or wire connection. Can be abstracted from the cell level to the transistor level.
Netlist Connection	Two points within a design which are electrically connected for the design to operate
Net	Set of netlist connections with a shared point (source)
Netlist	A list of nets that define the connectivity of a design
Interconnect	Metal connecting 2 points

In the user-design view of an FPGA, the overall structure is primarily divided into Logic Blocks and Routing. A logic block consists of a number of Look-Up Tables (LUT), usually 8-10. Each LUT implements a 3-4 input logic function. The overall structure is like a map, where the Logic Array Blocks (LABs) are like cities, the LUTs

are like buildings within the cities and the routing is like the highways connecting them. When the correct connections between the LUTs are made, the chip can implement virtually any function. Setting certain switches program the connections, such that the connections between the LUTs exist. Unlike determining the directions on a map, the path between two points cannot share the same interconnect.

Cores, or specialized function blocks, can be inserted into an FPGA to provide additional functionality. Some of these functions may include, Digital Signal Processing (DSP), memory, processor cores, Phase-Locked Loops (PLLs) or Clock-Data Recovery (CDR) circuits. Figure 4 depicts the high level view of an FPGA.

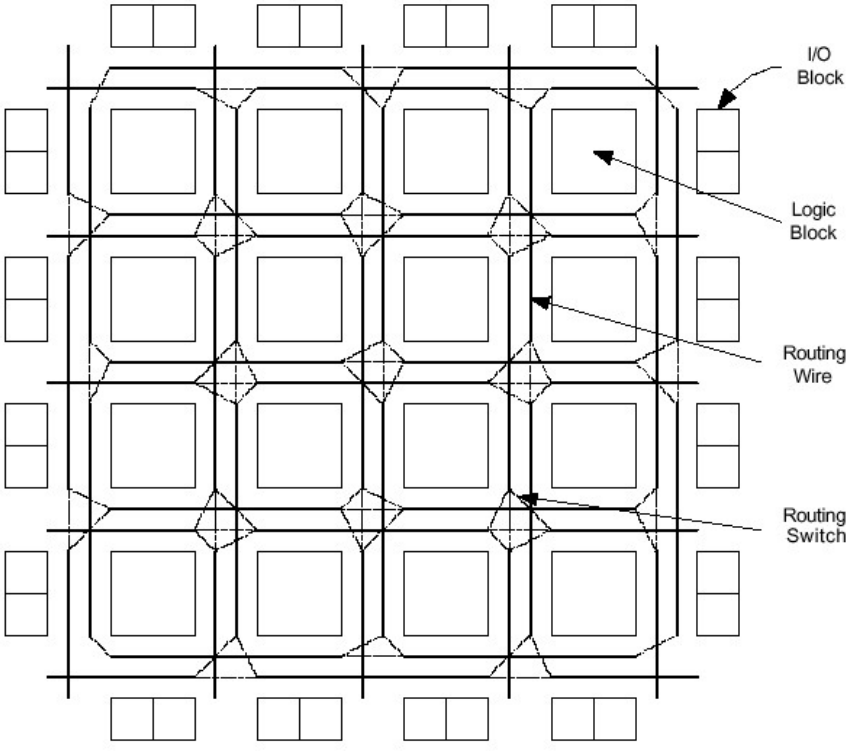


Figure 4: High level view of FPGA

At the user-design level, the positions of the blocks and routing are fixed. During Integrated Circuit (IC) design, each block location is undetermined and the routing paths

are not defined. The main task of IC design is to determine the optimal positions of the blocks and to connect them with routing. In addition to fixing the location of each block and the routing between them, the details of each block must also be developed. Once basic architecture parameters are defined (e.g. the number of LUTs to a LAB, the number of inputs to a LUT, the number of horizontal and vertical wires, the connectivity patterns, etc.), the optimization of placement is an arduous task requiring significant effort by many engineers. Most of this work is done manually using Computer Aided Design (CAD) tools. Any change in the architecture parameters requires a complete reworking of the entire design, thus making experimentation with different architectures expensive in terms of resources.

Since the task of laying out a complete FGPA is a very large problem, tools, like Versatile Place and Route (VPR) and VPR-LAYOUT, allow architectural engineers to specify a sub-section of the FPGA, a FPGA tile, which are building blocks of the overall chip design. Once basic tiles have been defined, VPR replicates them appropriately to achieve the desired FPGA. This project focuses on automating the layout within each tile.

Automated Transistor Layout (ATL) is the first step in automating the tile layout process. ATL places the components, or cells, within each tile. Our project will define the transistor level layout within each cell and provide the routing between them.

2.2 CAD in FPGA

Implementing a circuit in an FPGA is highly complex due to the shear number of circuit elements involved. CAD tools exist to assist designers in this task. An FPGA user will typically provide a high-level circuit specification using a hardware description language (HDL) or schematic entry. Then CAD programs will convert this abstraction of the circuit into a detailed programming file that dictates the circuit map in the FPGA. This procedure is normally divided into three sequential sub-processes (synthesis, placement and routing) to keep the complexity tractable as shown in Figure 5. VPR is a CAD tool that implements placement and routing for FPGAs. It provides the transition of data flow from high-level architecture description files, in order to generate the necessary detailed programming file on which ATL operates. This section provides a description for each sub-processes and how VPR tackles each process.

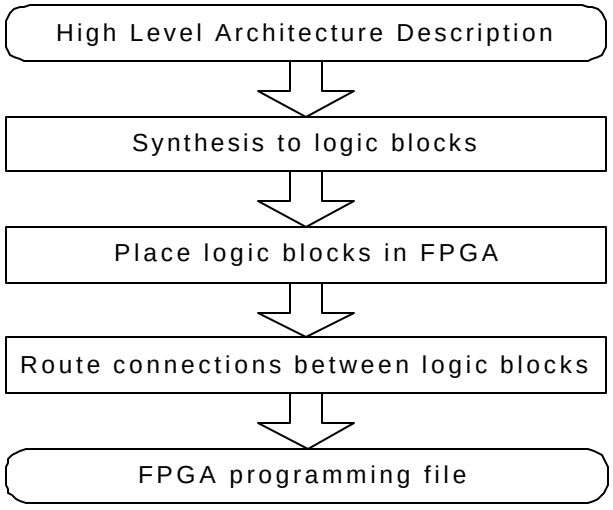


Figure 5 FPGA CAD flow

2.2.1 Synthesis

Synthesis is the automatic conversion of a hardware description language model into a netlist of logic blocks, governed by a set of design criteria, such as area and speed. Synthesis first converts logic-level HDL into basic gates, and then undergoes the logic synthesis process. Logic synthesis involves performing logic optimization to reduce area and/or delay, mapping the optimized netlist of basic gates to look-up tables (LUT) and packing the LUTs into logic blocks.

The first two stages of the logic synthesis have been extensively studied. Good algorithms and tools are publicly available. The problem raised in the last stage, Logic Block Packing, is a form of clustering. Clustering and partitioning are inherently the same problem; both divide a netlist into smaller pieces. While partitioning is the process of dividing a circuit into only a few pieces at a time, clustering breaks the circuit down into many small pieces in one step, as opposed to recursively partitioning into a few partitions in each step. Partitioning is also known as a top-down approach and clustering as a bottom-up approach. Clustering has been studied at length. However, many methods are not capable of constraining simultaneously on the maximum number of inputs, the number of clocks and the number of LUTs and registers in a logic block, which are key in logic block packing.

VPR claims to be the first publication work that describes algorithms targeting at "cluster-based" logic blocks. Two packing approaches are presented: a basic algorithm named VPack and a timing-driven algorithm named T-VPack. The complexity of VPack

is O(k_{max}*K*n), where k_{max} is the maximum number of terminals on a net, K is the number of inputs to each LUT and n is the number of LUT's plus the number of registers in the circuit. The author also claims that T-VPack outperforms VPack in terms of both circuit speed and routing area required.

2.2.2 Placement

Placement is the task of placing modules adjacent to each other to minimize area or cycle time. Two main algorithms that have been developed are min-cut (partitioning-based) and simulated annealing. The Min-cut algorithm is a recursive procedure that partitions the group of blocks to be placed into two subgroups with the minimum number of signal interconnections, until the leaf cells are reached. In simulated annealing, the movement of modules is likened to thermal annealing. Modules are initially placed randomly, and the "temperature" of the layout is estimated according to measurements such as area and timing. As the layout "cools", the overall rating of the layout improves. For each proposed movement, the rating is calculated. A proposed movement can proceed only if the resulted rating is improved.

VPR incorporates the simulated annealing algorithm for the placement, because it is much easier to add new optimization goals or constraints to a placer based on such an algorithm. The algorithm incorporates three enhancements over conventional placement algorithms that use simulated-annealing approach: a new annealing schedule that is adaptive to the current layout, a linear congestion cost function that provides better

results that all other alternatives in a reasonable computation time, and an incremental net bounding box update method that reduces placement CPU time by a factor of over 5.

2.2.3 Routing

After each logic block in a circuit has been assigned a location, a router is needed to determine how to connect all the logic block input and output pins required by the circuit. Routing a connection corresponds to finding a path between the logic blocks. The path is preferred to be as short as possible to comply with the limited number of wires in a FPGA. A route for a net should not take up resources that another net needs.

Modern routers study net interactions and perform routing in parallel. The overall procedure divides the routing area into smaller pieces and uses a global router to assign each net to a few routing areas. A detailed router will then proceed to place the actual wires.

VPR performs either global routing or combined global-detail routing for FPGAs and incorporates a routability-driven and a timing-driven router. The timing-driven router also uses routability as a consideration. While many path finding algorithms are available, both routers were developed based on a Pathfinder negotiated congestion algorithm. The Pathfinder algorithm produces excellent results due to two innovations: allowing overuse of routing resources, and allowing congestion to gradually be resolved and timing to be directly optimized.

The routability-driven router outperforms all other routers in terms of routing circuits with a minimum amount of routing. However, the routers cannot be compared on the basis of timing due to lack of standard benchmarks in this area.

2.3 Previous works

VPR has become quite renowned in industry and widely referenced in many papers relating to FPGA research. This acclaim is due to relative performance of the algorithms presented compared to the performance of similar software packages. Our project is part of an on-going research project at the University of Toronto. We extend an existing CAD placement tool, ATL, which has its basis in VPR.

2.3.1 Architecture Generation

The versatility of VPR lies in its ability to place and route almost any FPGA architecture. This is possible because VPR does not make any inherent assumptions about the FPGA architecture and instead provides a mechanism for users of VPR to specify FPGA architectures. Because of the complexity of the architecture of an FPGA, a language for expressing different architectures to the software is defined. This standard language is able to capture the most essential design features of a vast class of FPGA architectures. The thesis notes that by enforcing certain architectural constraints, an architect only needs to define a subset of an FPGA – an FPGA "tile" – to specify variations. By taking advantage of the ability to create an FPGA from a small set of tiles, VPR produces a flexible and efficient connectivity structure for FPGA architectures.

Using VPR's infrastructure and the succinct architecture representation format, it is possible to design and evaluate a wide selection of *viable* FPGA devices. The term viable implies that the device is capable of implementing real-world designs.

The specification provided in the architecture file also contains pertinent information about the fundamental electrical parameters that characterize the wires and switches inside an FPGA. VPR uses this information to compute area and delay estimates of the FPGA if it were manufactured by a semiconductor fabrication facility. The software then uses the delay estimates and tile connectivity structures to generate an internal representation of the entire FPGA. This data structure, known as the routing resource graph (RR graph), is a weighted, directed graph that correlates to the input architectural description.

The RR graph implicitly contains the transistor-level structure of the FPGA. For example, nodes in the graph classified as "logic cell sources" can represent the look-up table structure outlined in Figure 6.

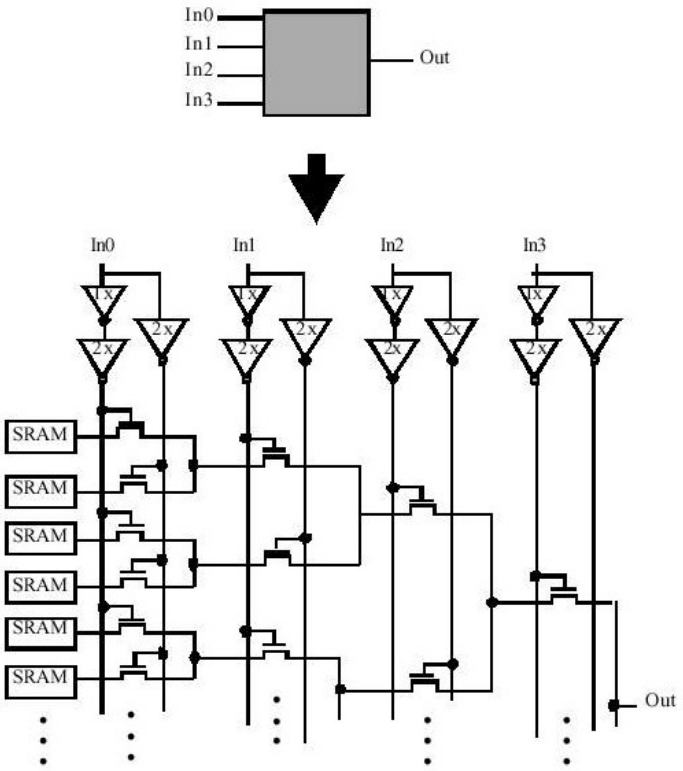


Figure 6: Lookup Table Schematic

As illustrated in Figure 6, the logic cell source not only represents transistors, but additional logical transistor groupings such as static RAM cells, buffers, and multiplexers. VPR supports architectural parameters that affect the transistor-level structure of the lookup-table, allowing users to vary the physical implementation of an FPGA's fundamental digital logic components by changing a small number of parameters. Other parameters supported by VPR can vary the number of routing wires in a channel and the amount of connectivity between the transistors and the routing fabric.

To evaluate the performance of a given architecture, the CAD tools in VPR use approximate area and delay information for an anticipated physical layout of the FPGA.

These estimates are based on abstract models that characterize an FPGA's electrical components. Under ideal conditions, one would produce a complete layout for each FPGA architecture of interest to obtain precise area measurements and accurate delay values [4]. This statement represents one of the primary goals for our project.

2.3.2 Netlist Generation

VPR_LAYOUT is a software program that is used by VPR to generate two equivalent representations of an FPGA tile using an architectural description as input. VPR_LAYOUT creates these representations by extracting the transistor-level information embedded in VPR's routing resource graph into two different file formats. The overall flow of VPR_LAYOUT is depicted in the figure below. As depicted, VPR_LAYOUT is a direct extension to VPR.

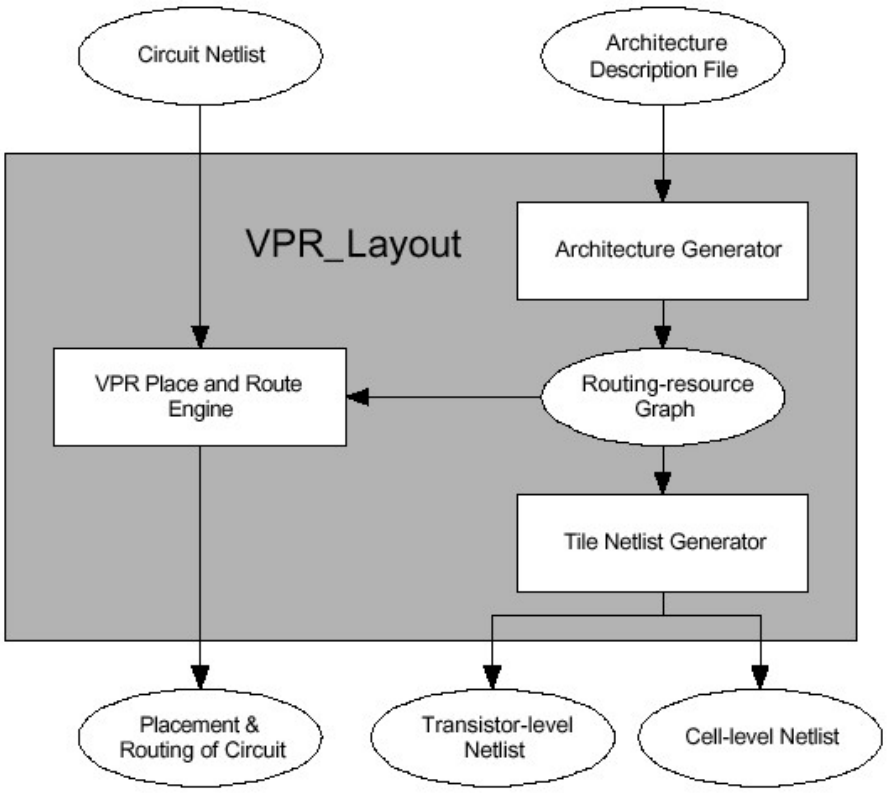


Figure 7: Overall Flow of VPR_LAYOUT

One file VPR _LAYOUT produces is the transistor-level netlist. This netlist captures the structure of all transistors that form a single FPGA tile. An FPGA, like any other VLSI system, can be decomposed into many interconnected digital logic elements such as static RAM (SRAM), lookup tables (LUT), and multiplexers (MUX). Each digital logic component can be created by a set of transistors. Combining these details, we state that the transistor-level netlist contains the most fundamental information about the functionality of an FPGA tile. It is the structural relationship of the transistors that defines the behaviour of an FPGA or any other digital device. The transistor-level netlist is needed by one module in our project, intra-cell layout, to generate the compact layout for each cell.

In an FPGA, connections between transistors are not limited to a single tileable region. VPR_LAYOUT represents connections at the tile boundary by an abstract entity called a "port". Therefore, in the context of VPR_LAYOUT, an FPGA tile consists of a set of transistors and ports that have a specific connectivity pattern. In addition to the connectivity information between transistors and ports, the transistor-level netlist contains tileability constraint data for the ports. The concept that an FPGA can be built using a small number of "tiles" replicated in a grid-like fashion implies that there are restrictions in positioning ports on the sides of a tile. These constraints are enforced by VPR's architectural generator and inferred from its routing resource graph. The netlist contains restrictions for a port's side and constraints that enforce two ports to be positioned directly opposite each other on the tile's perimeter. Figure 8 shows an example where two ports have fixed sides and a coupling constraint between them.

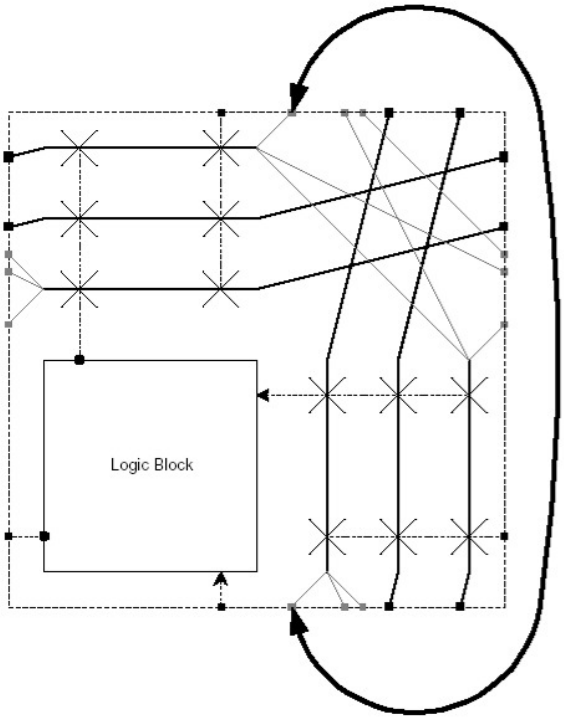


Figure 8: Port aligned for tileability [2]

The second equivalent representation that VPR_LAYOUT produces of an FPGA tile is the cell level netlist. This type of netlist groups all of the transistors into logical "block types" that directly correspond to the types of digital logic elements used to construct an FPGA tile. Each instance of a digital logic element is known as a "cell". The abstract concept of a cell is useful for dividing the problem of producing a high quality electrical layout into two sub-problems.

The first sub-problem is the creation of an effective layout of the transistors within a cell. The second sub-problem is the implementation of an algorithm that produces a layout of the cells within an FPGA tile and defines legal routing connections between them. This was the motivation of the automated CAD layout tool, ATL, which is the basis of our project. Our project interprets the cell level netlist for requirement parameters needed for the next stage, routing.

2.3.3 Placement

The CAD tool created in [2] performs the automatic layout placement of the netlists generated by VPR_LAYOUT. This tool, ATL (Automatic Tile Layout), uses the information in the netlists to estimate the physical size of the tile and produce an initial solution to the placement problem. A simulated annealing algorithm, based on the implementation in [4], is the core of the placement engine for ATL. One of the most important factors in simulated annealing algorithms is the cost function; this function is used to score the benefits of each placement and to decide whether a proposed arrangement is accepted or rejected. The cost function used by ATL tries to minimize the estimated total wire length required to make the connections specified in the cell level netlist.

Our project is an extension on ATL to provide an automated CAD tool for the complete tile layout process. ATL provides the crucial cell layout information required by the inter-cell router. Figure 2 (page 5) and Figure 3 (page 6) show the modified ATL design flow.

3 Intra-cell Layout

The intra-cell layout process takes the cell descriptions used in the Architecture Description and defines the locations of the transistors and routing required to implement each cell. Although an FPGA tile may have many hundreds of cells, there are only a handful of distinct cell types. Within the architectures investigated for the FPGA tile there are five main types of cells. They are:

- SRAM
- Look-up Table
- Buffer
- Multiplexer
- Flip-flop

These cells, when interconnected in the cell-level netlist, fully defining the functionality of the FPGA tile. This section of this report will provide a sample schematic and layout for the SRAM cell. The complete cell library of schematics and layout is found in Appendix A..

The original version of ATL already operated on cell-level netlists, with estimated sizes and port locations, to find good placements. Defining the actual layout of the cells prior to the inter-cell placement should yield better placements and lead to a better routed tile.

3.1 Layout Process

The overall intra-cell layout process involves defining a schematic for each cell type at the transistor level, verifying the functionality of the cell, defining a compact layout while conforming to design rules. The final layouts are read into ATL to update the cell-level netlists used in the inter-cell placer to improve upon the previous estimates.

3.1.1 Schematic Entry

In order to guarantee that the final layout for a given cell will provide the intended functionality, a digital representation of the cell needs to be created. Creating a transistor-level representation of the cell and simulating its performance confirms that the representation implements the intended functionality of the cell. Any tool that creates a netlist suitable for gate-level simulation is appropriate at this stage. Using SUE provides this functionality. In addition, SUE is integrated tightly with MAX, a layout CAD tool, and can provide verification that the layout matches the functionality of the schematic. Figure 9 is the schematic representation of the SRAM cell.

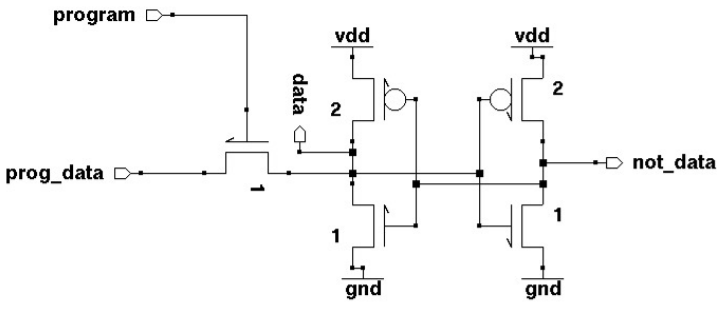


Figure 9: Schematic of an SRAM cell

Once the necessary cells have been created and verified, the actual layout process can begin.

3.1.2 Manual Layout

Performing manual layouts of the cells allows the FPGA architect to bring their experience to the design process and optimize each cell for both area and speed. Because each cell is independent, every cell can be optimized without worrying about affecting neighboring cells. During the manual layout process, certain constraints must be met. First, the cell must provide the desired functionality. Second, the layout must use no more than two layers of metal. This constraint allows 2 layers of metal within the cells, 4 layers of metal dedicated to the routing between the cells and 2 additional layers reserved for power, ground and clock nets. Third, the layout must meet the design rules for the process within the cell. Finally, the cells must allow abutment of cells without causing design rule violations.

Using the third party utility MAX, the custom layouts for each of the cell type was created. By using this CAD tools and the previously created schematics, the layout can be matched to the previously verified circuit representation of the cell. In addition, this CAD tool, like many others, provides design-rule checking thus increasing the possibility of a high fabrication yield. Figure 10 is the pictorial representation of the SRAM cell with MAX. This layout has no design rule errors and fully implements the functionality of the SRAM cell. Further work on this cell can further optimize the layout area required to realize the cell.

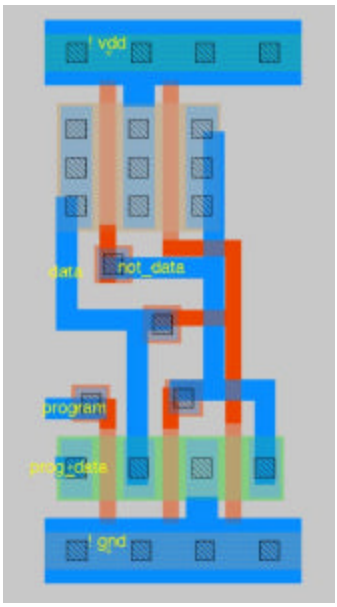


Figure 10: Layout of SRAM cell

The layout for any cell can be extracted into a SPICE netlist and simulations can be performed upon this netlist to ensure the timing characteristics of the cell are inline line with the timing requirements. This information could, in future extensions of this project, be used to make timing estimates of the tile.

Performing manual layout within a project that is designed to automate the layout and design process might seem to negate the benefits of automation. However, the limited number of distinct cell types within a tile means the number of manual tasks is small.

3.1.3 Layout Parser

The final layouts for each cell produced through manual layout must be imported into ATL. The layout parser acts as the entry point into ATL. Each cell layout is

translated into the internal intra-cell layout structure, *l_cells*. The implementation of the intra-cell layout structure in ATL is a simple data structure that uses straightforward linked-lists of structures in C. The main data structure, *l_cell*, contains cell identifier, the dimensions of the cell, a list of ports with locations relative to the lower left corner of the cell, and the internal structure of the cell. The data structures for the ports, *l_port*, and the internal structure of the cell, *l_intNode* are separate C structures within the *l_cell.* The graphical interface requires all the information contained within the *l_cell* to accurately draw the internal representations of the cell. The inter-cell placer and router require only the cell size and port locations as both treat the cells as black boxes.

To allow for easy expansion of cell types, there is a two-level file structure. The first level file is a cell list file. This file contains the filenames of each individual cell layout files to be included, the second level file. In the original implementation, the second tier of files were completely hand generate representations of the cell. In the final implementation of the layout parser, these files are the MAX layout files. Although the support still exists for the hand-generated layout files, the verification and validation of the cells created this way is not guaranteed in the process flow.

The layout parser provides more than simply reading a data file. The units of measure in MAX files are specified in microns and use specific technology processes. For example, layouts can be created in 0.25µm or 0.18µm process design rules. ATL expects layouts to be created under scalable CMOS design rules. The basic unit of measure in ATL is λ-based; a conversion between the two scales must be performed.

The final function of the layout parser is to convert the original cell-level netlist representation of the FPGA tile into one that replaces the estimated cell areas and port positions with those determined through the layout process. Since the inter-cell placer treats the cell as a simple block with dimensions and port locations, the newly sized cells can be interchanged with the original without affecting the placement algorithms. This also means that a mixture of estimated and actual cell information can be used during the development phase.

3.2 Inter-cell Placement

Inter-cell placement module determines the locations of each instance of the cells within the tile. ATL's placement routine uses the cell-level netlist representation of the FPGA tile to determine the cell types and connectivity within the cell. Cell dimensions and port locations are determined by the Intra-cell layout process and provided via the Layout Parser. A simulated annealing algorithm gradually improves cell placement while minimizes the expected wire lengths and wire congestion. The end result of the Inter-cell Placer is a placed cell-level netlist. The placed cell netlist defines not only the connectivity of the cells within the tile but also the orientation and position of each cell within the tile. The placed cell-level netlist is the starting point for the Inter-Cell Router, which has only to add the metal connecting the ports of the cells.

4 Inter-cell Routing

Ultimately, a VLSI layout is physically unrealizable unless an electrically legal path exists for each logical connection specified in the circuit netlist. The purpose of the router is to define the dimensions, position, and orientation of many metal segments that collectively connect up the logic design, while meeting the constraints imposed by the process design rules.

This section will describe how the inter-cell routing module of ATL works. The "inter-cell" qualifier indicates that this module only considers connections that involve different cells. The intra-cell layout module is responsible for defining the connections that are localized within a single cell. We begin by describing the goals and constraints placed on the routing algorithm and the high-level design decisions that were made at the onset of the project that balance these considerations. We then explain how the silicon area available for inter-cell routing is internally represented and why the representation is both valid and well suited to software routing algorithms that we created. Next, we describe the implementation details of a routability-driven router that was developed to solve the inter-cell routing problem. Finally, we present qualitative and quantitative evidence that the router successfully generates a high-quality, electrically legal routing for all the circuits available in our test suite.

4.1 Position in CAD Flow

Figure 11 illustrates where the ATL router fits into the CAD flow. The router's input is a netlist that describes the position of each cell within the FPGA tile and the position of each port on the perimeter. This information comes from the output of the ATL placer and will be referred to as the "netlist placement". The netlist placement is either read in from an output file or is generated by invoking the ATL placer. The ATL router is invoked once this information is obtained. Once complete, the router produces a compact output file of all the metal segments required to route the circuit and a collection of routing statistics about the solution is has generated. Examples of these statistics include the total length of wire used in the routing attempt, the distribution of wire usage over the routing area, and any connections that the router could not successfully route. By creating a "feedback" loop between the placement and routing modules, this information can be utilized by the placer to improve the overall quality of the layout and the success rate of subsequent routing attempts in the event of an initial routing failure.

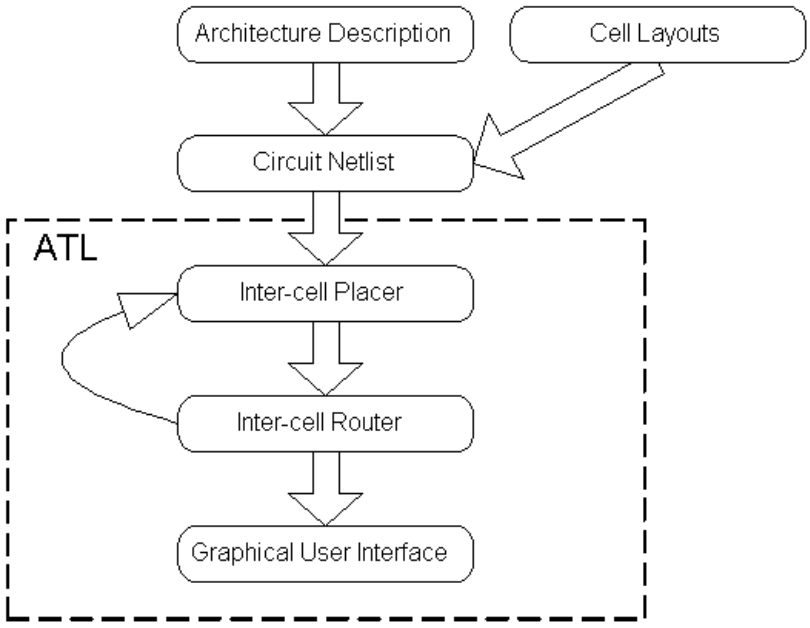


Figure 11: Router Position in ATL CAD Flow

4.2 Routing Goals & Constraints

Although electrical legality is the only direct constraint imposed on the router's operation, the definition of a high-quality routing algorithm is based on a combination of the routed circuit's performance, the success rate (i.e. routability) that the algorithm has in generating a legal routing, and the computational resources required by the routing algorithm. A balance of these factors is necessary to produce a tool that is useful in the FPGA design process in both academia and industry.

The most prominently used measurement for qualifying synchronous circuit performance is the maximum delay between any two registers in the logic design. The inverse of this delay represents the maximum frequency for the clock (denoted F_{max}) at which the circuit can operate successfully. However, our project focuses on generating a routing for an FPGA tile, which is a platform for ***other*** digital logic designs. Therefore, the true goal is to produce a layout that enables the FPGA to be able to implement other logic designs with the highest F_{max}. This implies that equalizing the delays on all the paths will not necessarily to the best performance for the FPGA. Although there are a substantial number of factors that need to be considered for the evaluation of the FPGA's performance, the standard method is to generate a delay profile for several representative paths of the FPGA tile. Some paths of interest, based on FPGA CAD tool research are listed below.

- The delay between the LUT inputs and the logic element flip-flops.
- The delay between the output of a logic element and the input of another logic element in the same tile.

- The delay between the output of a logic element and the edge of the tile for each of the wire types available (e.g. short vertical, short horizontal, long vertical, long horizontal).
- The delay to traverse the entire width/height of the tile.

Based on the high-level architecture, the FPGA designer needs to give guidance to the CAD tool about the relative importance for each of the representative paths of the FPGA tile. VPR & VPR_LAYOUT, the software tools that transform the high-level architecture description into the circuit netlist that ATL uses, currently do not support the specification of timing constraints. These applications need to be modified in order to propagate the FPGA designers' timing requirements to the routing module. In the absence of this specific timing information, the ATL router attempts to equalize the delay on all paths. However, the router is designed such that timing information could be incorporated with minimal modifications to the core algorithmic structure. In order to effectively and efficiently monitor the routing quality with respect to timing, a fast and accurate net delay extractor and a path-based timing analyzer need to be developed. The details of this process are well known and thoroughly discussed in [1].

The effort spent by the router to create a routing that minimizes the delay for the majority of connections is wasted if the solution is not electrically legal. A potential layout is worthless if there is even one connection that does not have a legal path. Although the inability to route a single path does not appear to have significant ramifications, the routing task is sufficiently complex and massive to preclude "hybrid"

solutions involving a software routing algorithm that produces a layout that is almost legal and a cleanup phase performed by hand. Therefore, the most important goal for the routing engine is to generate solutions where 100% of the connections are legally routed – even at the expense of increasing the average/maximum connection delay.

A final consideration for CAD tools that cannot be overlooked is the amount of computational resources that are required to generate a solution. In order for an automated layout tool to be useful to FPGA architects, the application must be able to produce a layout within a reasonable amount of CPU time and utilizing a reasonable amount of physical RAM. These aspects were important considerations in the design of the router since similar commercial CAD tools for VLSI place-and-route have substantial hardware requirements [10]. We expect that a high-quality design and memory utilization strategy is necessary to ensure our router can function effectively with a reasonable amount of hardware resources. Our initial expectations regarding hardware utilization anticipate that the inter-cell routing module could route the largest circuit in our test suite in 15 minutes on a 1 GHz processor using, at a maximum, 256 MB of physical RAM. These computational constraints serve as an initial filter for various data representations and heuristics that were considered in the design phase of the routing module.

4.3 Routing Grid

The router is required to connect up the cell pins of the logic design. This task is accomplished by defining the position of the metal segments that carry the logic signals

across the FPGA tile in order to implement the high-level FPGA architectural functions. An important aspect of the inter-cell routing component of ATL is the internal representation of the area available for routing wires. Since the router algorithm constantly requests information about the "status" of the metal area, the accessibility and memory efficiency of the metal area's internal representation is paramount in the design of the inter-cell routing module.

The goals and constraints defined for the router in the previous subsection emphasize that the success of the router is a more important consideration than minimizing the amount of computational resources required by the routing heuristic. Two dominant representations of the metal surface have emerged for detailed routing algorithms in VLSI CAD tools [11]. The first of these representations is a grid that partitions the metal surface into equally sized regions, called nodes. The routing grid representation maintains explicit information as to which metal connection(s) occupy each routing grid node. The second representation records the state of the metal surface in a list of wires – each representing a single metal segment. The grid data structure requires more physical memory, since information is maintained about all locations on the routing surface, while the edge list approach only maintains information about the locations that are currently occupied by a routing connection. However, the grid representation has a distinct speed advantage since a constant time algorithm is available to identify whether a particular region of the routing surface is occupied. In contrast, the edge list approach requires a linear search through a small set of edges to answer a similar 'occupancy' query. Based on these reasons, we decided to use the grid approach

to represent the metal surface in the router. The rest of this sub-section provides details as to the exact relationship between the routing grid and the physical metal surface, how the information from the netlist placement is transferred to the routing grid, and how the routing grid serves as a convenient abstraction to shield the routing algorithm from actively considering layout design rules.

4.3.1 Design Rule Considerations

Current technological limitations enforce certain restrictions on the relative placement and connectivity for IC components in the layout. These restrictions are necessary to ensure that the layout has a reasonable chance of surviving the fabrication stage without any defects. Our project must consider these constraints in order for the generated layouts to be representative of a legal solution to the VLSI routing problem. Each VLSI process technology has a detailed set of process "design rules" that define the minimum distances between different types of metal wires that **should** be respected in order for the resulting layout to have a reasonable chance to pass the fabrication phase without experiencing a fault created by process variations. A single fault in a VLSI circuit renders the entire fabricated circuit useless. Therefore, it is critical that the routing solution respects the process design rules.

The design rules for a process are specified in terms of absolute distance, that is, µm. However, since each process size defines a unique set of design rules, it is unlikely that a layout in one process is transferable to another process. An alternate to using physical distances in the specification of design rules is to represent the constraints as

"Scalable CMOS Design Rules" [8]. This approach involves defining all design rules in terms of integer multiples of a base unit, λ, which is equivalent to ½ the minimum transistor length for the VLSI fabrication process. The primary advantage to this approach is that layouts that abide by the scalable design rules can be implemented in a wide range of semiconductor fabrication processes. Unfortunately, processes in the deep sub-micron range ($\lambda \leq 0.35\mu m$) have additional constraints due to quantum effects and fabrication limitations. An additional disadvantage is that the design rules must be conservative. These factors indicate that layouts generated by scalable design rules generally require more silicon area than equivalent designs implemented using absolute physical distances. However, since all the scalable design rules are specified in integer multiples, we believe that using them will greatly simplify the core of the routing algorithm. Since the primary goal of our project is to determine the feasibility of an automated approach to transistor-level design and layout of FPGAs, it was decided that we would be able to answer this question more quickly by using scalable design rules.

The routing grid is a three-dimensional structure. It is composed of an arbitrary number of metal layers – a parameter specified by the user to ATL. Each metal layer is divided into a grid of "squares", called routing grid nodes. Each routing grid node represents an 8λ x 8λ region of silicon area available for metal interconnect. The size of a node is chosen to be the minimum area such that two wire segments carrying different electrical signals can be positioned in adjacent routing grid nodes without violating any constraints imposed by process design rules. The scalable design rules for a 0.13μm process specify that the minimum width for metal wires on all layers except metal 1 is 4λ

and that the minimum spacing between wires is also 4λ. However, since the 1st layer of metal is reserved for intra-cell routing, these scalable design rules are layer-invariant. If the scalable design rules were not used, it would not be possible for the routing grid to have uniform design rules on each layer of metal. In that case, the structure of both the routing grid and routing algorithm would be substantially more complex.

There are significant advantages in having layer-invariant design rules. First, uniform dimensions for all routing grid nodes simplify the legality in using vias – the VLSI interconnect component that connects two pieces of metal on different layers. Several design rules that involve via legality do not play factor when the design rules on adjacent layers are equivalent. A second benefit is that each routing layer will have an equal number of routing grid nodes. This permits the three coordinates of a routing grid node (layer, x, and y) to be easily represented by an "encoded" index. All data structures that reference routing grid nodes utilize the encoded routing grid index, as opposed to explicitly defining the three coordinates. This results in substantial memory savings for ATL.

The inter-cell routing algorithm selects routing grid nodes to connect up the logic design. This information is not sufficient to generate a complete layout that can be implemented on silicon. The representation of the metal region inside each routing grid node must be exactly defined. This involves specifying the occupancy of each 1λ x 1λ unit of metal. This information is extracted based on the immediate connectivity of a routing grid node. In order to support all possible connectivity configurations, each

routing grid node has been divided into non-overlapping sub-regions. Figure 12 shows the breakdown of the metal area of a single routing grid node into the different sub-regions. In this diagram, each square represents a 1λ x 1λ unit of metal.

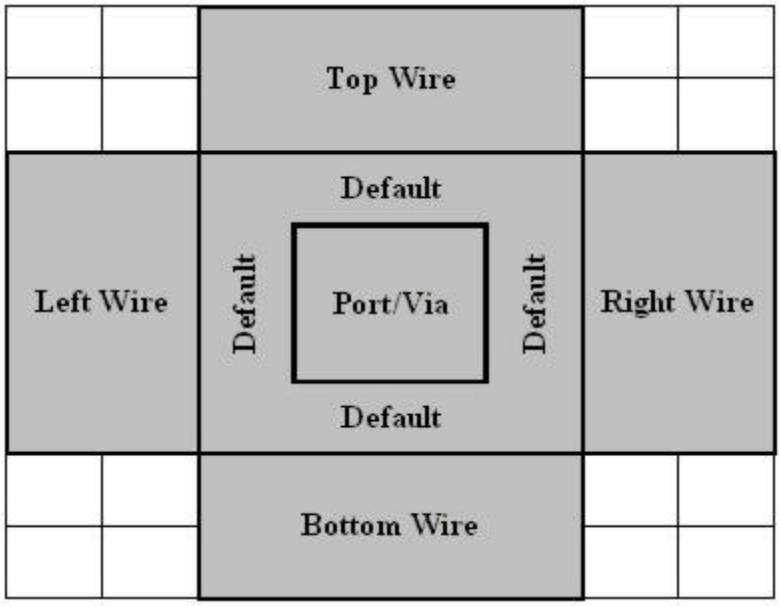


Figure 12: Relationship between Routing Grid Nodes and the Metal Area

For all grid nodes that are used in the routing of the circuit, the "default" 4λ x 4λ sub-region located in the centre of the available metal area used. The 2λ x 2λ port/via sub-region is used if there is either a port or a via in that node. These layout choices respect the design rules involving via enclosure. The other four sub-regions are used if the nodes on the same metal layer that are adjacent to the node in question are used in the routing of the same net. Since each sub-region represents a distinct piece of metal, a routing grid node may contain any combination of the sub-regions. Figure 13 shows the additional sub-regions that are used for each node in the routing of a sample net on one layer of metal. Each square in the diagram represents a single routing grid node. The letters L/R/T/B represent that the "Left Wire", "Right Wire", "Top Wire", and "Bottom

Wire" sub-regions, respectively, are used in that routing grid node. It can be easily seen that the resulting metal segment is unbroken and respects all design rules considered by the router.

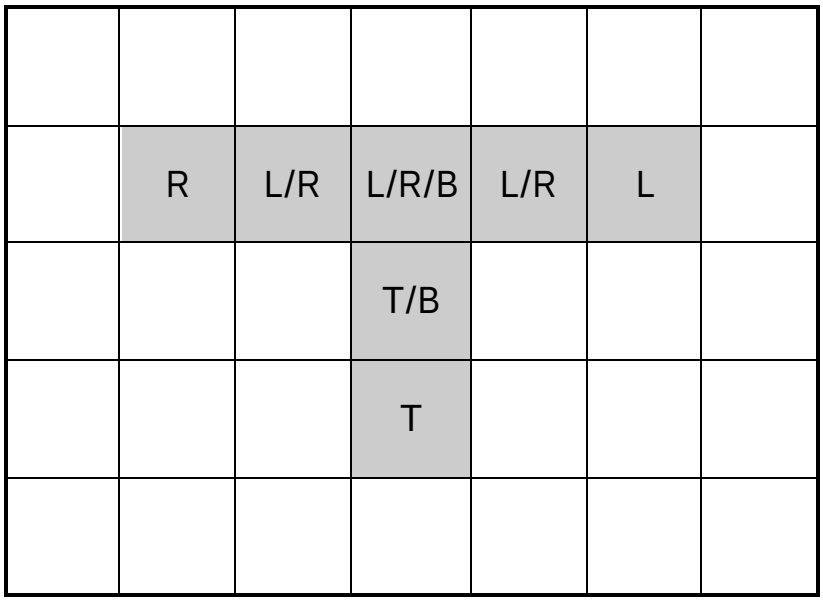


Figure 13: Underlying metal representation for routing grid nodes

4.3.2 Coordinate Transformation

The inter-cell router only deals with connections that are between the cells. The intra-cell layout module defines the connections that are fully contained inside a single cell. In order to obtain a relatively compact layout, two metal layers have been dedicated to intra-cell routing. It was decided that the routing module exclusively use the metal layers above metal 2 to connect the cells together. Since intra-cell layout is restricted to metal 1 and metal 2, all the "other" metal layers are unoccupied (i.e. have no metal segments in them) when the routing phase commences. This design decision was made since cells occupy the majority of the area in the tile (>90%) leaving only a small amount

of available metal area on the bottom two layers of metal routing. Additionally, there would be significant complexity introduced by allowing the router to use the same metal layers as intra-cell layout since the explicit design rule checking would be required to ensure electrical legality. To accomplish this, the internal layout of each of the cells would need to be exposed to the routing module. Therefore, restricting the router in this manner allows all cells to be treated as "black boxes". Because of this, the overall layout will be electrically legal provided that the router generates a solution that completes all the connections specified by the netlist.

In order to create an electrically legal routing, a conversion of the positional information from the placer coordinate system to the router coordinate system is required. It is imperative that this transformation maintains the aspect ratio of the FPGA tile, the relative positions of the cells to the FPGA tile, and the relative positions of the cell pins inside the cells – all specified by the placement phase. Provided that each cell pin is specified at a unique location, it is possible for the routing grid to be identical in size to the placer grid. This implies that each location on the **placer** grid represents an $8\lambda \times 8\lambda$ area on the metal surface. If a unity mapping between the coordinate systems is used, the intra-cell layout information is exactly preserved. If a unity mapping is not preserved, then the coordinate transformation will "skew" any implicit intra-cell layout information in the cells. It should be noted that the uniqueness requirement does not apply to ports on the perimeter of the tile since these ports, unlike cell pins, are not bound to a single layer of metal. Therefore, it is possible to resolve multiple tile ports at the same location into different layers of metal without losing any intra-cell layout information. The only

limitation is that the number of tile ports at any given layer is less than or equal to the number of metal layers available for inter-cell routing.

Limiting the input netlists to the routing module to respect the constraints defined above would reduce the usefulness of the router. In order for the routing module to be able to accept netlists that specify multiple pins at the same tile location, the routing grid needs to be larger than the placement grid. We use the concept of grid granularity to represent the relationship between the sizes of the placement grid and the routing grid. Specifically, defining a grid granularity of g specifies that the dimensions of each metal layer on the routing grid are g times larger than the dimensions of the placement grid. For example, if the placement grid size is 150 x 200 units and a granularity of 2 is specified, the routing grid size would be 300 x 400 units. However, the metal area that each routing grid node represents is 8λ x 8λ, regardless of the granularity of the routing grid.

The value of g is limited to integer values greater than or equal 1, since a transformation that preserves the location information in the placer netlist cannot be defined for non-integer values of grid granularity. The only limitation on selecting grid granularity is that the cell pins that appear on the placed netlist must be able to be uniquely resolved into routing grid nodes. Since the size of the routing grid is proportional to the square of grid granularity, it is extremely important to keep this value as small as possible.

The netlists that were produced by the original ATL code had multiple cell pins appearing at a single placer grid location since it did not utilize specific intra-cell layout information. Although the placer can operate successfully in the presence of "overlapping" pins, the router is unable to since it is electrically impossible to place more than one pin in an 8λ x 8λ area of silicon. The concept of grid granularity enables the routing module to transform these netlists to the routing grid and try to successfully route the circuit. By increasing grid granularity, any placed netlist can be routed by the routing module, irrespective of whether an exact intra-cell layout has been defined. However, once an intra-cell layout has been defined, the exact locations of the cell pins are known, relative to the cell, and are guaranteed to be non-overlapping. Therefore, a grid granularity of 1 (unity mapping) can be used for all netlists that implicitly contain intra-cell layout information. Furthermore, all values of grid granularity other than 1 will distort any implicit intra-cell layout information contained in the netlist.

The validity of the solution produced by ATL is intimately dependent on the fact that the intra-cell layouts for all cell types in the FPGA tile are specified according to an 8λ x 8λ "grid". Specifically, the size of each cell on the netlist corresponds to a layout of the exact size assumed by the router using a unity mapping to translate the placer coordinate system to the router coordinate system. Additionally, the positions of the ports for each cell type must be located at the region available for ports, as specified by Figure 12. For example, if the layout for an SRAM cell is 32λ x 48λ, the size of the SRAM cell on the netlist must be 4 units by 6 units. Ports on the SRAM cell must be 2λ x 2λ and the lower left-hand corner of the ports can only be positioned at one of the

locations specified by the following relationship: $[(8x + 3)\lambda, (8y + 3)\lambda]$, where $x \in [0,3]$ and $y \in [0,5]$. If both these conditions hold and the grid granularity is set to 1, then the routing grid will be an exact representation of the silicon area used to layout the SRAM cells. If the layout for all netlist cells satisfies these conditions, the routing module will contain the "true" view of the entire chip.

Before the routing algorithm is discussed, we present the procedure for resolving multiple pins at the same location in the placement grid. The original pin positioning code developed in [2] places the pins on the perimeter of the block and tries to group different pin classes to be representative of a potential layout. The procedure used to assign routing grid coordinates to each block pin begins by classifying each placer grid coordinate to one of nine regions based on the various sides of a cell directly reachable from the placer grid coordinate. Figure 14 outlines the classifications for a coordinates inside a cell that has dimensions 4 x 5 (in terms of the placer grid).

Top Left	Top Middle	Top Middle	Top Middle	Top Right
Middle Left	Middle	Middle	Middle	Middle Right
Middle Left	Middle	Middle	Middle	Middle Right
Bottom Left	Bottom Middle	Bottom Middle	Bottom Middle	Bottom Right

Figure 14: Classification of block coordinates for the pin placement algorithm

For each coordinate classification that represents a placer grid coordinate adjacent to an edge of a cell, block pins can be positioned in the routing grid spaces that map into that single placer grid coordinate. Assuming a grid granularity of 3, a single square in the placer grid represents a 3x3 region of routing grid nodes. It was deemed that the non-corner squares can support three block pins; one pin on the "edge" of the block at each routing grid coordinate. The rationale for this decision is that block pin congestion would be significant if pins were placed two rows deep – the 2nd row of pins (i.e. the pins closer to the middle of the block) would be forced to use a via in order to connect to another block pin outside the cell. The corner squares can support four block pins, one pin for each router grid square on the perimeter, but not on the direct corner of the block. This decision seemed obvious since it would be difficult to imagine a layout that would have a pin at the direct corner of the block. Figure 15 illustrates the valid pin positions for each of the nine coordinate classifications, assuming a grid granularity of 3.

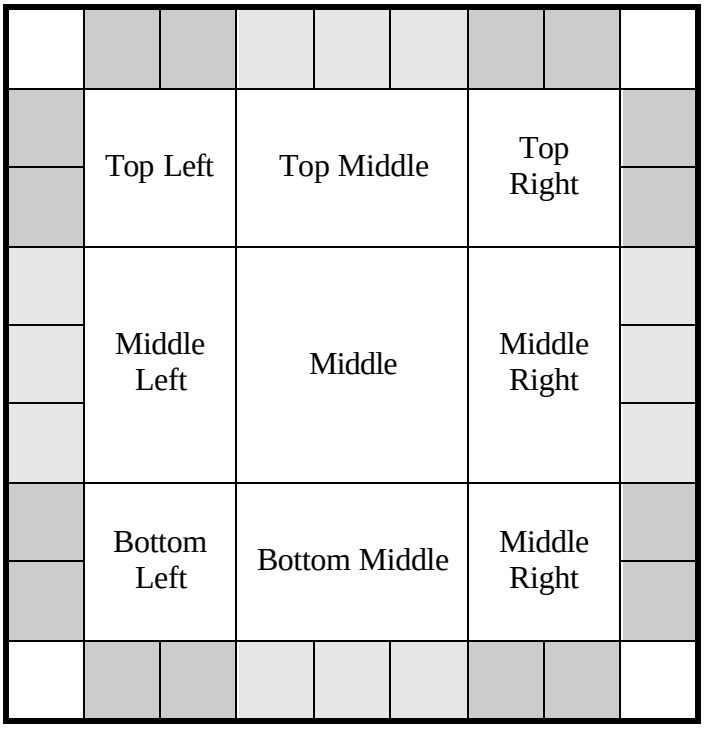


Figure 15: Pin positions used to resolve contentions on the placer coordinate system

For each cell, the algorithm assigns positions to each block pin based on its coordinate classification and on the number of pins that have already been assigned to that square on the placer grid. The algorithm has been designed to be flexible in both the granularity of the router grid coordinate system and the legality of the block pin positions for each coordinate classification.

4.3.3 Routing Grid Abstraction

The primary purpose of defining the routing grid and a complicated mapping between the coordinate systems in ATL is to isolate the routing algorithm from actively considering issues involving design rule legality. If the router can find the sequences (i.e. paths) of routing grid nodes that should be used to electrically connect the logic design together such that every routing grid node is used, at most, one time. The underlying metal representation of the circuit is extracted based on node connectivity, as was previously illustrated in Figure 13. The size of each routing grid node is specified such that the design rules involving metal spacing, metal width, and via enclosure are all met if two wires carrying different electrical signals are positioned in adjacent grid nodes. Since the routing algorithm can only select the sequence of nodes for a given position, it is guaranteed that the metal wires inferred from the path selections made by the router are legal, with respect to the design rules being considered.

4.4 Routability-Driven Router

The routing algorithm is an extension of the classical maze router approach [12], with various elements incorporated and adapted from routing algorithms for FPGAs [4],[9]. Although the FPGA routing algorithms that we have examined do not define silicon metal wires, they operate on a "resource graph" [9] – an equivalent of the routing grid. This sub-section will discuss the key aspects of the routing algorithm, the primary reasons for its success, some additional enhancements to improve the performance and speed of the algorithm, and the results it has obtained. The routability-driven router does not actively consider the delay of each path. Instead, its only focus is to generate a legal solution. Since the routing algorithm we developed is the first known work attempting to perform a combined global-detailed route at the transistor level for an FPGA tile, the emphasis was placed on feasibility. However, connection delay is given secondary consideration, since the router is trying to minimize wirelength, which has an indirect correlation with the delay of a routing path.

4.4.1 Algorithm Structure

Using terminology developed in the previous sub-sections, the routing problem can be restated as follows: determine, for each net, a sequence of routing grid nodes that electrically connects all the terminals of that net, such that no routing grid node is used by more than one net. The routing algorithm also tries to minimize wirelength used and balance the delay for all the connections in the circuit. A net that needs to connect more than two terminals together is called a net with multiple fanout. The starting point, called the source, is required to connect to several destination terminals, called sinks. All cell

pins that are sources represent an electrical connection to the drain of a transistor. Similarly, all cell pins that are sinks represent an electrical connection to the gate of a transistor.

The routing of a single connection essentially involves running Dijkstra's algorithm [13] on the routing grid trying to find the shortest path (lowest total cost) between the source and the sink nodes. The routing grid has implicit edges between adjacent routing grid nodes. The weight on each edge in the entire graph is set to unity. The routing of a net essentially consists of routing several two-pin connections, each having the same source node. However, since all sinks on the same net are electrically equivalent, the router can "re-use" any routing grid nodes that have been selected in the routing paths for connections on the same net. Therefore, the routing of high-fanout nets is a tree-like structure, as opposed to a series of isolated connections.

An important term used in the description of the routing algorithm is the "wavefront" of the net. The wavefront is the set of nodes currently being considered by the search algorithm. When a new connection is considered, the wavefront consists of the nodes that have already been selected for this net. The algorithm then considers routing grid nodes adjacent to the nodes currently in the wavefront – in effect, expanding the extent of the wavefront. This process continues until the sink being searched for is included into the wavefront of the net.

The computational complexity of Dijkstra's algorithm and the traditional maze router approach is $O(n^2)$, where '*n*' is the number of nodes in the routing grid. Several FPGA tiles we are routing have over one million routing grid nodes. The size of these tiles imply that a heuristic having a computational complexity $O(n^2)$ will be unacceptably lengthy. The creators of Pathfinder describe an essential enhancement to improve the speed of the routing algorithm. The simple breadth-first approach of Dijkstra's algorithm is extended to use an A*, or directed, search [9]. By considering an additional cost term (in addition to path distance) in the algorithm that is based on the distance to the target, the execution time of the algorithm can be greatly reduced. When this approach is run in the absence of congestion, the computational complexity is O(n) provided that the costs of any two adjacent nodes are ordered such that the node closer to the target has a lower cost than the node farther from the target.

The traditional maze router approach [12] does not allow the overuse of routing resources. This fact exposes the severe limitation that one routing path may block another path. Since the FPGA tile has thousands connections, this path "blocking" is inevitable. In order to overcome this limitation while still using an algorithm that routes the netlist connections serially, our routing algorithm uses the negotiated congestion principle that is present in both the Pathfinder [9] and VPR [3] routing approaches. The crux of negotiated congestion is allowing the overuse of routing resources and using the cost function to allow congestion to gradually be resolved as the algorithm progresses. These algorithms repeatedly rip-up and re-route every net in the circuit until all congestion is resolved – this idea is due to Nair [15]. Ripping-up and re-routing every

net in the circuit once is called a *routing iteration*. ATL continually performs routing iterations until the circuit can be legally routed without any congestion or a maximum number of iterations have been reached, at which point the router declares the circuit cannot be legally routed. Figure 16 contains the pseudo-code for the routability-driven routing algorithm used in ATL.

Let: **RT(i)** be the set of nodes, n, in the current routing of net(i);

PriorityQueue be a set of { TotalCost(n), PathCost(n) } pairs for each node, n, in the current wave expansion, sorted on TotalCost

StoredTotalCost be arrays with one entry per routing grid node, with all entries initialized to a huge number

PresCost and **AccCost** be arrays with one entry per routing grid node representing the current congestion and accumulated congestion costs of using that node in the routing of a connection

```
while (overused resources exist) { /* Illegal Routing? */
  for (each net, i) {
    rip-up routing tree RT(i) and update affected PresCost values;

    RT(i) = NetSource(i);

    for (each sink, j, of net(i)) {
      PriorityQueue = RT(i) at { TotalCost(n) = α(DistanceToTarget(n)),
               PathCost(n) = 0 };

        while (sink(i,j) not found) { /* wave expansion */
          Remove lowest TotalCost node, m, from PriorityQueue;

          if (TotalCost(m) < StoredTotalCost(m)) {

            StoredTotalCost(m) = TotalCost(m);

            for (all adjacent nodes, n, of node m) { /* expand node m */

              PathCost(n) = PathCost(m) + PresCost(n) * AccCost(n);
              TotalCost(n) = PathCost(n) + α(DistanceToTarget(n));

                   if (TotalCost(n) < StoredTotalCost(n)) {
                Add n to PriorityQueue at { TotalCost(n), PathCost(n) };
              }
            }
          }
        }  /* end wave expansion */

        for (all nodes, n, in path from RT(i) to sink(i,j)) { /* backtrace */
          Update PresCost(n);
          Add n to RT(i);
        }

        for (all nodes, n, expanded during previous wave expansion) {
          StoredTotalCost(n) = HugeNumber;
        }
      }
    }

  } /* end net routing */

  Update AccCost(n) for all n;
  Update PresCostMultiplier & PresCost(n) for all n;

}  /* end routing iteration */
```

Figure 16: Pseudo-code of the routability-driven routing algorithm

4.4.2 Routing Representations

An examination of the pseudo-code contained in Figure 16 reveals that the routing algorithm uses two major data structures: a priority queue to maintain the list of nodes, sorted by total cost, that are currently in the wavefront expansion and a routing tree to identify the connectivity pattern of the nodes currently contained in the routing of a net. A heap was chosen to implement the priority queue, since the routing algorithm only wants to examine the node in the wavefront expansion with the lowest cost.

The only complicated operation that is performed on these data structures is the addition of the nodes in the path from RT(i) to the sink(i, j). In order to discuss this operation, the structure of the routing tree needs to be explained. The physical structure of the routing tree is intuitive; it contains the routing grid node indices for all of the connections in a given net in a tree-like format. "Join points" are the only nodes in the routing tree that have more than one child node. At the start of the routing of each net, the routing tree consists of exactly one node – the routing grid node representing the source of the net. After the routing algorithm determines the path for each connection on the net (all nets have at least one connection), the routing tree is updated to reflect the new path by examining the sequence of heap elements selected to be part of the new path. The 1st heap element considered by the routing algorithm will be a node that exists in the routing tree at the beginning of the wavefront expansion algorithm. Due to this fact, it is possible to correctly update the tree and identify the join point for the new segment of the net's routing.

In addition to the heap and routing tree data structures, we decided that there should be a common data structure for representing a sequence of routing grid nodes. This was a necessary step since the intra-cell layout, inter-cell routing, and graphical editor modules were developed in parallel. We decided to use a linear linked list data structure of routing grid nodes to communicate the sequence of interconnect segments used to route the each connection in the netlist. This linear linked list of routing grid nodes will be called the "traceback" for a net. The traceback is composed of segments, where each segment identifies the new routing grid nodes selected by the algorithm for each successive connection in a net. In order to identify join points, each segment of the traceback structure needs to repeat one of the routing grid nodes in a previous segment of the net. Additionally, a "separator" element in the traceback structure is needed to identify the beginning of a new segment. The equivalent traceback data structure for a potential routing for a 3-pin net on a sample routing grid is displayed in Figure 17.

1	2	3	4	5	6	7
8	9	10	11	12	Pin C 13	14
15	16	17	18	19	20	21
22	Pin A 23	24	25	26	27	28
29	30	31	32	33	34	35
36	37	38	Pin B 39	40	41	42
43	44	45	46	47	48	49

Associated Traceback: 23→24→25→32→39→XX→25→26→27→20→13

Figure 17: Representation of the Traceback Data Structure

The routing tree and the traceback are essentially two different representations of the same information; that is, they contain the routing grid nodes used to connect the different terminals of a single net. Both representations are useful since each has unique advantages based on the position in the program flow that needs to access current routing information. The routing tree is more convenient representation when the routing engine is active since it is less cumbersome of identifying the join points in a net's routing. Additionally, according to [4], the routing tree structure is more appropriate for storing timing information, a future goal for the ATL router. However, the traceback data structure provides a more compact representation, it is easier to traverse, and most importantly, the traceback is segmented into a series of connections that mimics the ordering of connections in the cell-level netlist. This correlation is required so that there is a logical linkage between the routing and the circuit description. The routing tree data structure does not have this essential relationship. Figure 18 shows the contents of each of the data structures as each connection in the net of Figure 17 is routed.

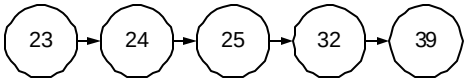
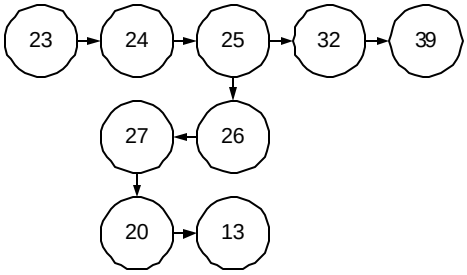
Algorithm Stage	Routing Tree	Traceback
Before 1st connection		Nothing
After 1st connection		23→24→25→32→39
After 2nd connection		23→24→25→32→39→ *(sep.)*→25→26→27→ 20→13

Figure 18: Comparison of Routing Tree and Traceback Representations

4.4.3 Cost Function

As shown in the pseudo-code algorithm, the routing algorithm assigns a "cost" to a routing grid node being considered in the wavefront expansion based on the cost of the previous node on the path being considered, the current congestion of that node (*PresCost*), and the historical congestion of that node (*AccCost*). In essence, the cost reflects the circuit's demand to use a specific point of the metal area and the total amount of metal on the path required to reach this node. It is critical to realize that the algorithm considers not only the cost of the routing grid node currently being considered in the directed breadth-first search, but also the cost of every routing grid node on the "best-

case" path to reach current point on the wavefront. The notion of the "best-case" path is guaranteed since the router will never add a node to the priority queue if another path has been considered that has a lower cost to that node.

As already described, congestion arises when two nets occupy the same routing grid node. Therefore, when considering a routing grid node to be used for the connection being routed, congestion will occur if one or more nets already occupy that node. The formula for the value of the present congestion cost, *PresCost*, is given by Equation 1, where *Occ(n)* represents the occupancy of the routing node being considered and *PresCostMultiplier* represents the "severity" of congestion at the stage in the routing algorithm. The lower bound on present cost is 1.0, representing the intrinsic cost of using this routing element.

$$PresCost(n) = 1 + Occ(n) \cdot PresCostMultiplier$$

Equation 1: Present Congestion Cost for Routing Grid Nodes

The principle of negotiated congestion is realized by setting *PresCostMultiplier* to be unity in the first routing iteration and increase it exponentially for all routing iterations that end with congestion remaining. Specifically, it was determined that defining $PresCostMultiplier = b^{iter}$, where *iter* is the current iteration number, and $\beta = 1.15$, provides a balance between trying and removing congestion and minimizing the wirelength used by the circuit. In order to increase the importance of congestion as the algorithm progresses, the constant β must be greater than unity. As β increases, the router tries to remove the congestion more quickly and, in most cases, produces a solution that requires more wirelength, or declares the circuit to be not routable.

The accumulated, or historical, cost "future" cost of the path represents the total pressure experienced on a specific routing grid node over ***all*** the routing iterations. This term in the cost function helps the process of resolving node congestion by identifying the nodes that should be avoided. This indicates to the router that a node that has continuously been congested should only be used when it results in significant savings (i.e. much lower cost function). This cost function term is especially important when the routing of a net is "forced" to use a congested node and all the congested nodes have equal present congestion. In order for the solution to move towards convergence, the router must select the node that has less historical congestion. Equation 2 specifies the formula for adjusting the accumulated cost of a routing grid node *at the end of a routing iteration*. At the beginning of routing, *AccCost* is set to unity. It was found that $a = 0.25$ provides a good balance between the multiplicative weight of historical congestion and present congestion when a node is being considered in the wavefront expansion.

$$AccCost(n) = AccCost(n) + a \cdot Occ(n)$$

Equation 2: Historical Congestion Cost for Routing Grid Nodes

The final aspect of the cost function is the relative weights of the "explored" to the "unexplored" portion of the path being considered. Equation 3 specifies the formula for the total cost assigned to a node *n*, a neighbour of node *m*, the node currently being considered in the expansion process. The function *DistanceToTarget*(*n*), is the minimum Manhattan (i.e. rectilinear) distance to the sink node of the current connection being routed. It should be noted that the Manhattan distance to the target is equal to the lower

bound of *PathCost*(*target*) - *PathCost*(*n*). This is one of the requirements of the A* search algorithm enhancement [14].

$$TotalCost(n) = \left[PathCost(m) + PresCost(n) \cdot AccCost(n)\right] + a \cdot DistanceToTarget(n)$$
$$PathCost(n) = PathCost(m) + PresCost(n) \cdot AccCost(n)$$

Equation 3: Expansion Costs for Routing Grid Nodes

The constant a determines the "directedness" of the routing algorithm. For values of a greater than 1, a routing node that is closer to the target than any node in the wavefront will have the lowest cost, provided it is not currently congested. Since the lowest cost node is always pulled from the heap first, the algorithm will only explore nodes that move towards the target. This is only true if all nodes being explored are not congested. The effect of congestion alters the selection of routing grid nodes in the expansion process. The cumulative effect of all three fundamental equations involved in the router cost function is that the router identifies a solution as quickly as possible when there is no congestion, but considers alternate paths depending on the severity of the congestion.

4.4.4 Speed Enhancements

The amount of CPU time required for the algorithm presented in the previous sub-sections is heavily influenced by the amount of free space in the routing grid and the number of metal layers that the router is allowed to use to route the circuit. Allocating a larger size to the routing grid – in effect, increasing the percentage of "free space" in the routing grid – or increasing the number of layers available to the router dramatically

reduces the CPU time required to route the circuit. The explanation for this phenomenon is based on the schedule for *PresCostMultiplier*. The routing algorithm cost function schedule increases this multiplier exponentially as the algorithm progresses. Therefore, in a situation that is a deterministically unroutable, the algorithm will continually explore (in futility) an increasing number of paths before declaring the "best" path, based on total cost, to be a congested one. For scenarios that are on the threshold of being routable, the router requires many routing iterations in order to resolve all the congestion. In contrast, a situation that is "easily" routable requires significantly fewer routing iterations and can be routed in a fraction of the time. Adding a new metal layer to the routing grid for a circuit on the threshold of being routable can decrease the CPU time required by an order of magnitude. Adding two metal layers can decrease the CPU time by two orders of magnitude. However, after a certain point, no further time reductions can be obtained by adding additional metal layers.

One of the simplest speed enhancements made to the routing algorithm that was implemented is to identify whether a net is currently legal and instructing the routing algorithm not to perform a re-route of a net unless at least one routing grid node in that net is currently congested. This does not degrade the quality of the routing since, generally, the total wirelength used to route a net increases as the router tries more diligently to remove congestion. If the router finds a solution with no congestion when the weight assigned to congestion is low, then we can declare the initial legal solution to be the "best" legal solution in the routing context and no further routing attempts are needed for that net. If the router cost function picks a routing grid node on a legally

routed net to be used by congested net, then the legally routed net becomes congested again and it will be re-routed in the next routing iteration.

The time saved from not re-routing legal nets is substantial. On average, over 90% of the nets are legally routed when the router reaches the "halfway" iteration mark of the legal solution. That is, if the router takes 50 iterations to find a solution, 90% of the nets are legally routed at iteration number 25. However, the speedup from this enhancement is not proportional to the percentage of nets that are routed since most of the **legally** routed nets are 2-pin nets, whereas the remaining congested nets have a significant number of pins. An important point of this speed enhancement is that it does not introduce any degradation in the routability of any circuit. That is, this speed enhancement will not prohibit the router from finding a legal routing for the circuit, if one exists.

Another enhancement was made to the core of the routing algorithm after several circuits took an unexpectedly long time to route a single net. After some investigation, it was discovered that the router was using an extremely large amount of memory to route a connection and was increasing the CPU time of the routing algorithm due to the swapping of memory to and from the hard drive. The situation that occurred was that thousands of copies of many *congested* routing grid nodes were being placed onto the routing heap. Although a node is added to the heap only if the cost to that node is less than the lowest path cost to that node. However, the lowest path cost, *StoredTotalCost*(*n*), is only set after a path to that node is considered. Since the routing

grid nodes causing the problems were congested, the costs of these nodes were high enough so that they were never being pulled from the heap. Therefore, each of the thousands of expansions to these nodes was added to the heap. This is plausible since the number of paths between any two nodes is exponential based on the distance between the nodes.

The "enhancement" that was developed to fix this problem exploited the principle that the relative costs for the expansion a single routing grid node are only dependent on their historical costs. That is, the costs assigned to the neighbours a routing grid node will be at a minimum for the minimum cost path that reaches that node. Therefore, it is not necessary to expand that node for any path other than the minimum cost path. We define the "active heap element" of a routing grid node to be the element on the routing heap representing the minimum cost path to that node. Additionally, we change the requirements for a routing grid node to be added to the routing heap to include the condition that the path cost to that node is less than the path cost of the active heap element for that node, if one exists. This solution bounds the number of elements that can appear on the routing heap, limiting the memory used by the routing algorithm, and fixing the original problem.

Another modification that was considered to improve the speed of the routing algorithm is modify the routing grid node expansion process to add nodes onto the heap that are not directly connected to the node being expanded. The additional nodes that are added to the heap are in the same line as the nodes being expanded, closer to the target,

and a significant distance from the node being considered. This process is dubbed "beam routing" since to the position of the additional nodes being considered are in a straight line, as a beam of light, in the direction of the target. Figure 19 shows the locations of nodes being considered by the expansion of the node *X*.

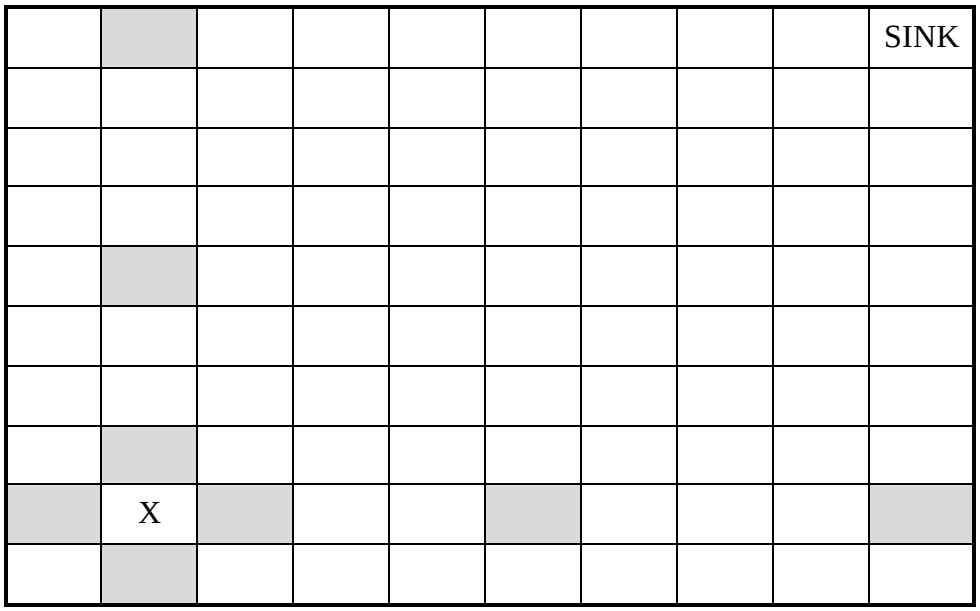


Figure 19: Routing Grid Nodes Considered by Beam Routing Approach

Although the beam routing approach speeds up the routing of a single net by reducing the number of elements added to the heap, it does not work well in the presence of congestion. Therefore, special precautions were made to ensure that beam routing expansion does not double-back on itself and specify an illegal routing path. Experimental results have shown that beam routing has a negative effect on the CPU time required by the router since the computational effort required for determining the situations where beam routing can be legally applied is greater than the amount of time that is saved by exploring the nodes far away from the target. Therefore, it can be

concluded that this approach, which appeared to look extremely promising in the theoretical stage, has no utility in practice and that the router achieves a better solution by methodically examining the routing grid one node at a time instead of trying to take shortcuts.

One final technique to reduce the CPU time required is the addition of logic to predict whether the routing algorithm will eventually fail and terminate the algorithm once an accurate prediction of failure can be made. As previously discussed, the router takes significantly more CPU time for circuits that are not routable. This enhancement will improve the average time of routing a circuit. However, it will not improve the amount of time to route circuits that have a congestion-free solution or are extremely close to have a legal solution. The prediction code maintains a history of the number of overused routing grid nodes for all of the previous routing iterations. The code that makes the decision whether to terminate the algorithm considers the number of overused routing grid nodes and the slope of the historical numbers for overall congestion. For unroutable solutions, the router will terminate in less than one-third the time than letting the router attempt the maximal number of routing iterations.

4.4.5 Performance Enhancements

The routing algorithm structure presented in Section 4.4.1 focuses exclusively on resolving congestion and minimizing wirelength. The main limitation of the cost function is that the path cost term is exclusively influenced by congestion *at the node being explored*. The routability of a logic circuit can be improved by providing

additional information to the cost function about the decisions it can make that influence the *global congestion* of the circuit. For example, it was found that the congestion near the edge of the circuit is particularly high since all connections that travel between cells are required to connect to the perimeter of the FPGA tile. Therefore, it is useful to instruct the router to avoid using routing grid nodes near the perimeter of the chip unless absolutely necessary. Additionally, by reducing the number of vias and bends in the circuit, the overall routability can be improved. Finally, some VLSI routing approaches [19] specify that routing the majority of routing connections on the same layer in the same direction (i.e. horizontally or vertically) for any give same layer allow more routing tracks to be used in the available routing area. However, it is important that each of these considerations be treated as guidelines and not as hard-and-fast rules. For example, restricting all connections in metal 2 to route vertically would limit the search space of the routing algorithm.

During the incorporation of these concepts into the router cost function, it was decided that the fundamental structure of the algorithm must not be perturbed due to its proven success in solving complex routing problems. Specifically, any transformations applied to the cost function must not affect the negotiated congestion aspect of the algorithm. Furthermore, it was realized that the cost function is structured so that two (or three) routing grid nodes adjacent to the node being explored would be assigned equal cost, assuming that there is no straight line from the node being considered to the target. Figure 20 shows the costs assigned to each node by the routing algorithm for a congestion-free circuit during the wave expansion for a 2-pin net. It is assumed that the

multiplier for the *DistanceToTarget* function, a, is set to 1.1. As seen in the diagram, for each routing grid node being expanded, both nodes that are closer to the sink are given equal costs. Since the nodes have equal costs, the routing heap "randomly" selects one of the two nodes to be expanded. It was recognized that the introduction of a small difference in the cost of routing grid nodes to reflect routing decisions on a global level would achieve the desired result, while not reducing the search space of the routing algorithm.

						SINK	
		9.6					
	11.8	9.7	9.6				
	11.9	9.8	9.7				
	12.0	SRC	9.8				
		12.0					

Figure 20: Costs Assigned in Wave Expansion Algorithm

The routing algorithm was enhanced with several "bias factors" that introduce a small perturbation in the cost function to reflect global routing decisions that improve the overall routability of the circuit. A "bias factor" is a condition applied to the expansion of a routing grid node that introduces a small difference in the cost assigned to its neighbours that has an effect on the global routability of the circuit. Each bias factor is assigned a value, slightly greater than unity, which represents the penalty that should be

applied to a routing path that violates the bias factor condition. The overall penalty applied to the node, *BiasFactorPenalty*(*n*), is the product of all the individual bias factor penalties that are applied to the path implied by the routing grid node being considered. Equation 4 represents the formulae for the *PathCost* and *TotalCost* variables that account for the effect introduced by bias factors.

$$PathCost(n) = PathCost(m) + PresCost(n) \cdot AccCost(n) \cdot BiasFactorPenalty(n)$$

$$TotalCost(n) = PathCost(n) + a \cdot DistanceToTarget(n)$$

Equation 4: Expansion Costs for Routing Grid Nodes with Bias Factor Considerations

Four bias factors are introduced into the router cost function:

- *Against Routing Flow*: This bias factor guides the router to route all paths on the same layer in the same direction.
- *Against Wire Direction*: This bias factor attempts to limit the number of "bends" in routing paths. Zigzagging paths are commonplace if this bias factor is not considered.
- *Via Usage*: Routing paths that use vias are given a small penalty. Although via use is inevitable, this factor tries to reduce the overall number of vias in the circuit.
- *Edge Usage*: Routing paths that are within 10% of the perimeter of the tile are slightly penalized.

Figure 21 shows the costs assigned to each node by the routing algorithm that considers the "against routing flow" bias factor for a congestion-free circuit during the

wave expansion for a 2-pin net. It is assumed that penalty for a routing path that violating this bias factor is 1.01 and that the vertical direction is the "flow" for this metal layer. It can be seen from the diagram that the node costs are no longer equal and the router chooses the nodes that route the circuit in the vertical direction.

						SINK	
		9.6					
	11.81	9.7	9.61				
	11.91	9.8	9.71				
	12.01	SRC	9.81				
		12.0					

Figure 21: Costs Assigned in Wave Expansion Algorithm with Routing Flow Bias Factor

Another improvement that reduces the overall wirelength of the circuit is explicitly determining the order that connections of a multiple fanout net should be routed. The netlist lists the destinations of a net in a random order. Naïvely routing the connections by their position in the netlist data structure can generate extremely poor results. The situation portrayed in Figure 22 illustrates a situation that can occur when the connection terminating at $SINK_1$ is routed before the connection terminating at $SINK_2$. When the routing tree is added to the heap when the 2^{nd} connection is being considered, the routing algorithm recognizes that $SINK_1$ as the closest node to $SINK_2$. In this scenario, the overall routing path to $SINK_2$ is extremely poor. By changing the order

of the connections, this effect can be avoided. Figure 23 shows the resulting paths when $SINK_2$ is routed before $SINK_1$. The wirelength savings from ordering the sinks of a net are proportional to the number of pins on the net. However, over 90% of the nets in the circuits being considered by the ATL are 2-pin nets and do not experience any benefit from sink ordering.

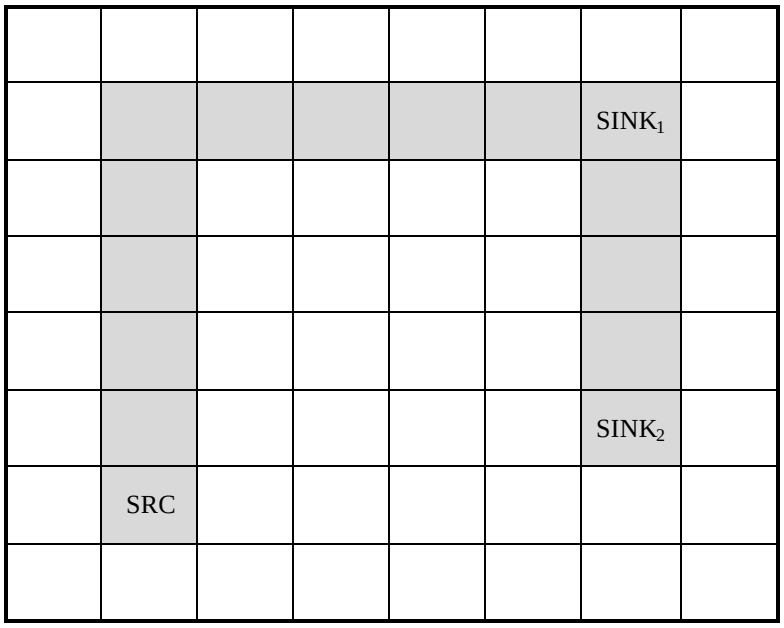


Figure 22: Naive Routing of 3-pin Net

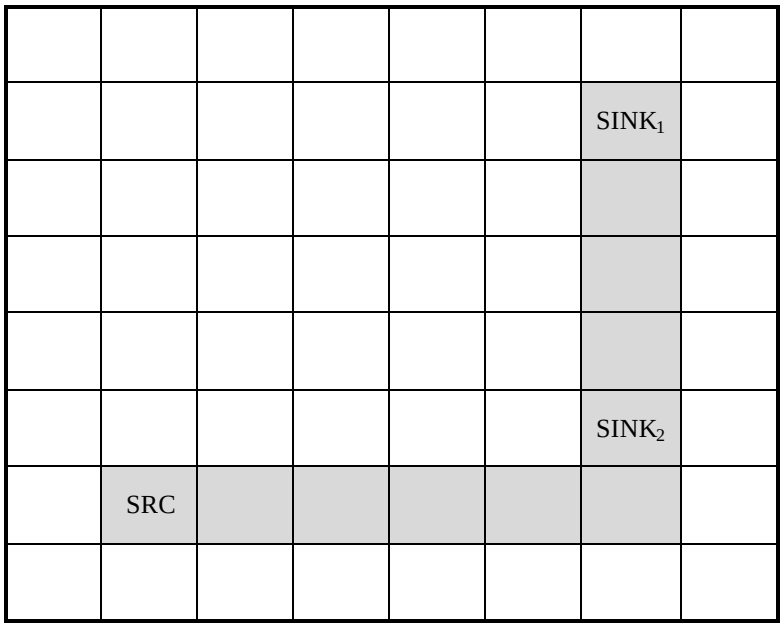


Figure 23: Routing of 3-pin Net with Explicit Sink Ordering

One final performance enhancement was the introduction of a "clean-up" routing iteration that re-routes every connection after the router has found a legal solution. By inspecting the routing visually, using the graphical user interface that was developed, it was found that several nets had obvious paths that required less bends. These sub-optimal routes were not corrected since these nets were never re-routed once they were free of congestion. Additional space for the sub-optimal path was created when the routing algorithm re-routed another path that was previously blocking the sub-optimal path. An additional routing iteration where all nets are re-routed is performed to fix these sub-optimal routes while maintaining the speed advantage gained from not re-routing legal nets. On average, this enhancement reduces the total wirelength and number of vias required by the circuit by one-tenth of a percentage point with a negligible increase in routing CPU time. Additionally, this re-routing phase does not affect the routability of

the circuit since the router will select the same path for a net connection if no improvement for that connection can be found.

4.4.6 Validation Module

One of the most important tasks for any software system that attempts to solve a real world problem is the validation of the solution produced by the application. A dedicated routing checker was developed that verifies that the routing module produces an electrically legal routing at the routing grid node level. The routing checker does not consider the legality of the underlying metal representation. Section 4.3.2 describes how the underlying metal representation is inferred from the utilization of a routing grid node and the necessary conditions to ensure that the metal representation corresponds to the implicit intra-cell layout information in the netlist. In addition to validating the routing solution against the design rules, the checker serves the purpose of increasing our confidence that the routability-driven router algorithm is "correct" and that the data structures used to represent the routing are mutually consistent. A stand-alone verification module lowers the chance that invalid or incomplete data is passed to other modules in the program flow; in our program, the graphics module uses the results of the router.

The following major tests are performed in the stand-alone routing checker:

- All the destination ports are connected in the routing of each net.
- The routing of each net is a tree.
- No portion of the circuit is electrically shorted.

- The occupancy specified by the traceback data structure agrees with the occupancy count annotated on the routing grid.
- Each adjacent segment in a net's routing is logically connected to each other. Specifically, the sub-regions in each of the routing grid nodes reflect the connectivity of that node.

4.5 Placer & Router Communication Loop

It is desirable that each invocation of the ATL application by the user will return a legal layout to the user. In the cases where the routing algorithm fails to find a legal route, the placement needs to be modified in order for subsequent routing attempts to succeed. An iterative loop has been developed between the inter-cell placement and inter-cell routing modules so that an electrically legal layout will eventually be identified after one or more unsuccessful routing attempts. An important component in this iterative loop is the quality of information about the previous routing attempt that is transferred to the placer. This component allows the placer to adjust for congested regions that it did not consider in its initial placement attempt. Specifically, the router records the overall congestion for all routing iterations, the exact nets that could not legally be routed at the end of the attempt, and the level of congestion over each cell in the FPGA tile.

The types of modifications made to the placement are primarily dependent on the amount of congestion at the end of the previous routing attempt. If the router fails by a significant margin, the placer might need to increase the size of the tile in order to give

the router additional space to resolve the congestion. If the router can obtain a solution with only one or two congested routing grid nodes, it may be possible for the inter-cell placer to use the same tile size, but reposition the cells to reduce the localized congestion.

Another alternative that is available to ATL, in the attempt to create an electrically legal solution, is to increase the number of metal layers available for inter-cell routing. The results produced by the routing algorithm indicate that an extra layer of metal can produce an electrically legal solution for a placement that incurred several hundred routing grid nodes of congestion in its previous routing attempt. Unfortunately, the number of metal layers available for the layout of an FPGA tile is usually fixed, so this solution is not viable in all situations.

4.6 Specialized Net Routing

The routing algorithm developed considers the connections specified on the input netlist. However, the power signals (i.e. VDD/GND) and clock signals are not included in the netlist and, because of that, not routed. In order for the circuit to be functionally correct, these global nets must be routed. The routing algorithm can be adapted with minimal effort to route the power signals, but is not well suited to define a high-quality clock routing since it is critical to route these nets with low skew.

A typical routing structure for power and ground nets is to use thick, interleaved wires on the upper metal layers to carry these global signals [16]. This structure is displayed in Figure 24. This configuration is most beneficial when standard cells are

used, since the power and ground rails run directly over the locations that require the signals. The routing of the power and ground nets in the ATL application requires more effort than the standard cell layout style since the inter-cell placement module does not arrange the cells into rows. The router requires knowledge of the interleaved power/ground routing structure in the upper metal layers in order to adapt the routing algorithm to consider these specialized nets. These nets can be routed by first identifying a number of points on the top metal layer that are directly beneath the power rails and using these nodes as the "starting" point for the routing tree.

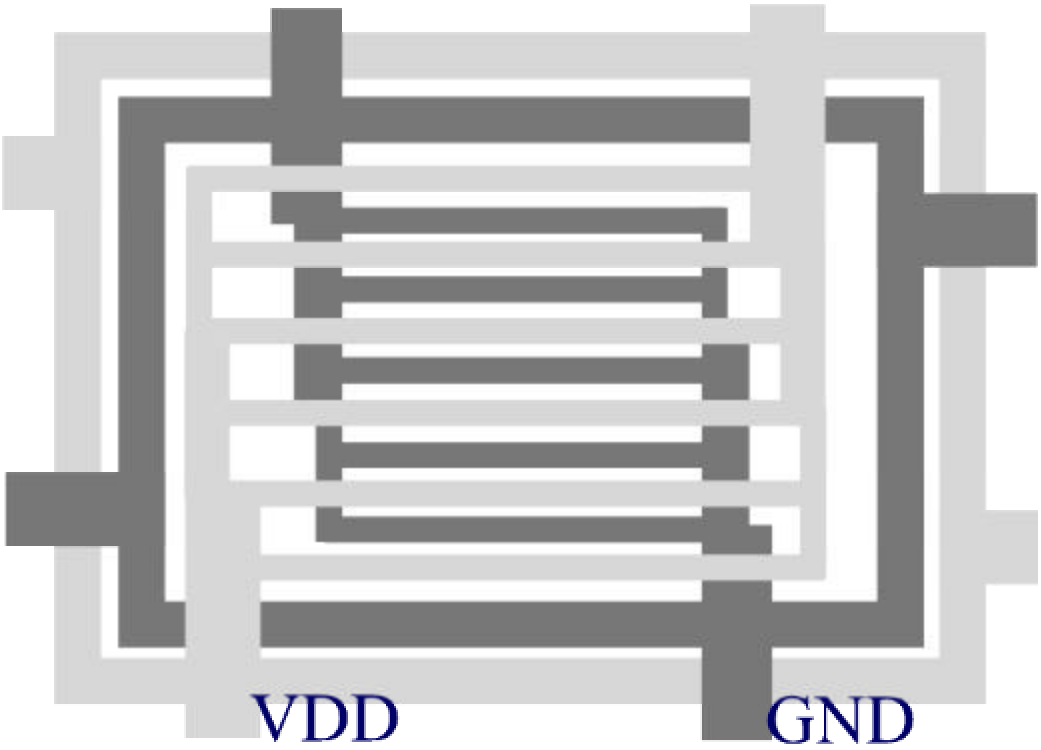


Figure 24: Traditional Power/Ground Rail Layout

Although it possible to identify similar "starting" points for clock nets on the top metal layer reserved for inter-cell routing, the problem is fundamentally different since it is important to balance the delay between the connections. Currently, the router attempts

to minimize wirelength for all the connections. Additional logic is required in the routing algorithm to ensure that the skew in the clock net does not impact the circuit operation. A bias factor might help in equalizing the delay between the different clock connections. However, the routing algorithm might perform in an unpredictable manner since the bias factor would directly contradict the dominant term in the cost function. Since the netlist does not contain clock connections, it is impossible to realize clock routing for the FPGA tile. Therefore, this task is left to future extensions to the ATL application.

4.7 Routing Results

Ten netlists representing different types of FPGA tiles were used to test the success rate, overall quality, and speed of the routing algorithm. Given that this is the first CAD tool developed that generates a complete layout for an FPGA tile, there is no commercial application that can be used as a reference point for the routing algorithm. An additional complication in estimating the results produced by ATL is the incompatibility in the input circuit file, the cell-level netlist, which was developed specifically for this tool. However, since ATL performs a transistor-level layout of a digital logic design, it is possible to convert the netlist to a standard format, such as EDIF [17], that several commercial CAD tools can process. Although a direct comparison with another tool cannot be made at this time, useful quantitative and qualitative assessments of the router's quality can be derived from analyzing the router results. Finally, the exact layout sizes obtained by FPGA companies are closely guarded information and cannot be used for evaluating the performance of the router.

The primary goal of this project is to evaluate the feasibility of an automated solution to the layout problem for FPGA tiles. This goal translates to the task of defining quality intra-cell layouts for all the cells that can appear in a tile, deriving actual port positions and dimensions of the cells, and using this information to place and route these cells in the tile. The most crucial aspect is to use the exact positional information from the intra-cell layout phase in the routing grid. Specifically, this implies that a grid granularity (see section 4.3.2) used to build the routing grid is set to unity. All the results presented in this section have been run using a grid granularity defined as unity.

The secondary goal is to evaluate the router's quality by comparing both the number of metal layers and additional space required by the routing algorithm against approximate numbers for similar types of layout problems. It should be noted that the algorithm always identifies an electrically legal solution given enough metal layers and a significant space buffer between all cells in the tile. Therefore, the results presented in this section defines a solution with two numerical quantities: the number of metal layers required and the area increase required over the original tile size provided by the placement module. All area values are expressed as a ratio. For example, if a placement requires a 220 x 220 grid to successfully route, but the inter-cell placer generated a placement using a 200 x 200 grid, the area increase would be $\left(\frac{220}{200}\right)^2 = 1.21$.

The benchmark circuits used to test the router represent logic array blocks [18] containing varying amounts of configurable logic and programmable routing. The VPR architecture generator [4] is the software tool that parses the high-level architectural

description for an FPGA tile and identifies the logical connections required to implement the tile. As part of [2], Padalia created ten circuits to test the original ATL application. These circuits are used to test the quality of the router. The name of each circuit identifies the number of 4-LUT cells contained in the cluster. An appropriate amount of programmable routing is added to each tile, based on the number of lookup tables in the tile. Table 4 contains pertinent information as to the number of transistors and nets in each of the ten benchmarks used to evaluate the quality of the router.

Circuit	# Transistors	# Cells	# Nets	Placer Grid Size
tile_1x4	3093	1197	836	156 x 177
tile_2x4	3762	1365	1007	254 x 231
tile_3x4	4359	1483	1146	296 x 310
tile_4x4	4854	1603	1274	333 x 270
tile_5x4	5621	1711	1435	399 x 411
tile_6x4	6336	1853	1586	461 x 450
tile_7x4	7023	1931	1700	514 x 477
tile_8x4	7592	2011	1783	533 x 520
tile_9x4	8459	2108	1908	590 x 575
tile_10x4	9342	2245	2057	640 x 640

Table 4: Physical Information on Benchmark Circuits

An important aspect in CAD tool design is the amount of CPU time required to complete the operation. As previously discussed, the length of time required to route a circuit is greatly affected by the stress placed on the router. Routing the same circuit with one more metal layer finishes over an order of magnitude faster. Table 5 identifies the low-stress and high-stress routing times for the ten circuits in our benchmark suite[1]. The results indicate that the smaller circuits route incredibly quickly, but the algorithm takes a

[1] Athlon 1000 MHz Processor with 256 MB of PC-100 SDRAM used for all CPU time results

considerable amount of time to find a solution for the larger benchmark circuits. It seems clear that the algorithm is not well suited for circuits containing more than ten thousand transistors and will definitely not scale to designs containing hundreds of thousands of transistors.

Circuit	Low-Stress Routing (CPU s)	High-Stress Routing (CPU s)
tile_1x4	3	18
tile_2x4	11	144
tile_3x4	22	238
tile_4x4	37	493
tile_5x4	73	796
tile_6x4	86	974
tile_7x4	124	1357
tile_8x4	190	2158
tile_9x4	233	2785
tile_10x4	306	4051

Table 5: CPU Time Required By The Router

After running the benchmark circuits with many different sets of parameters for the routing bias factors, cost function weighting factors, and the "directedness" of the algorithm, a single set that consistently produced decent results was chosen to use as the basis of the evaluation of the routing algorithm. Table 6 presents the minimum additional area required to generate a legal routing for each of the benchmark circuits, using several different selections for the number of metal layers available for inter-cell routing[2].

[2] The area bloat becomes excessive when circuits above tile_3x4 are routed on 3 layers of metal

Circuit	Metal Layers	Router Area Ratio
tile_1x4	3	1.124
	4	1.024
	5	1.000
tile_2x4	3	1.257
	4	1.089
	5	1.018
tile_3x4	3	1.335
	4	1.122
	5	1.052
tile_4x4	4	1.231
	5	1.105
	6	1.034
tile_5x4	4	1.265
	5	1.117
	6	1.049
tile_6x4	4	1.294
	5	1.144
	6	1.055
tile_7x4	4	1.319
	5	1.171
	6	1.061
tile_8x4	4	1.351
	5	1.190
	6	1.069
tile_9x4	4	1.377
	5	1.231
	6	1.072
tile_10x4	4	1.410
	5	1.262
	6	1.078

Table 6: Experimental Routing Results for Benchmark Circuits

Two major observations can be made from the results presented in the previous table. First, all circuits are fully routable in four layers of dedicated inter-cell routing metal. After recognizing that two layers are devoted to intra-cell layout and that two metal layers are sufficient for routing power, ground, and clock nets, it can be concluded that the entire layout produced by ATL requires eight layers of metal. This number is comparable with the number of layers required by modern VLSI processes. Therefore, in this respect the router produces a feasible and quality solution. Second, the amount of additional area required in order to legally route the circuit is considerable (41% for the largest circuit), but decreases substantially for each additional metal layer allowed to be used by the routing algorithm. However, it should be noted that achieving an electrically legal layout in a few hours for an FPGA tile having the same complexity as modern FPGA devices is a substantial achievement.

4.8 Summary

In this section we described the routing portion of the Automated Transistor Layout CAD tool. A description of the representation for the silicon area available for metal wiring was presented along with a justification of its validity. A routing algorithm was developed that utilizes several different attributes of previous CAD approaches that generate high quality results along with many new ideas formulated using specific knowledge and concepts from both the VLSI and FPGA domains.

The success of the ATL routing algorithm demonstrates that an automated approach to the transistor level layout of an FPGA is feasible. The quality of the results

indicates that there is potential for using this tool to aid FPGA architects in the design and layout of the FPGA tile. However, further investigation needs to be performed in order to accurately compare the layouts produced by ATL against other software applications that generate detailed transistor-level layouts for VLSI circuits.

5 Graphical User Interface

The goal of the graphical editor is to provide a visualization of the entire FPGA tile as designed by ATL. This includes showing the placement of blocks, the routing between blocks as well as the transistor layout within the blocks. Further requirements include the ability to edit, save and load routing information.

5.1 GUI Functionality

5.1.1 Inter Cell Placement

The original ATL was already capable of drawing placement. However, the visualization engine was insufficient and not easily extendable to accommodate the additional requirements. Therefore, the majority of the block placement visualization code needed to be rewritten. Blocks are drawn using rectangles and can be distinguished by the different colours used. Block placement is two-dimensional so visualizing the block placement is quite straightforward.

5.1.2 Inter Cell Routing

The amount of routing nodes used for the complete routing of an FPGA tile is very large and visualizing the layout is complex. Further there are multiple layers of overlapping metal and so a mechanism by which this can be visualized needs to be defined. The selected implementation uses a different colour for each layer of metal. Overlapping layers of metal are represented using transparency effects such that higher

layers do not hide the layers beneath them. The connections between routes and the ports that connect the routing to blocks is also shown.

The visualization is user-interactive. The user can select to display any subset of layers through keyboard input. Routing nodes and nets can be selected and displayed in isolation as well – this is useful for identifying a long meandering route that can then be corrected. An example of this can be seen the figure below.

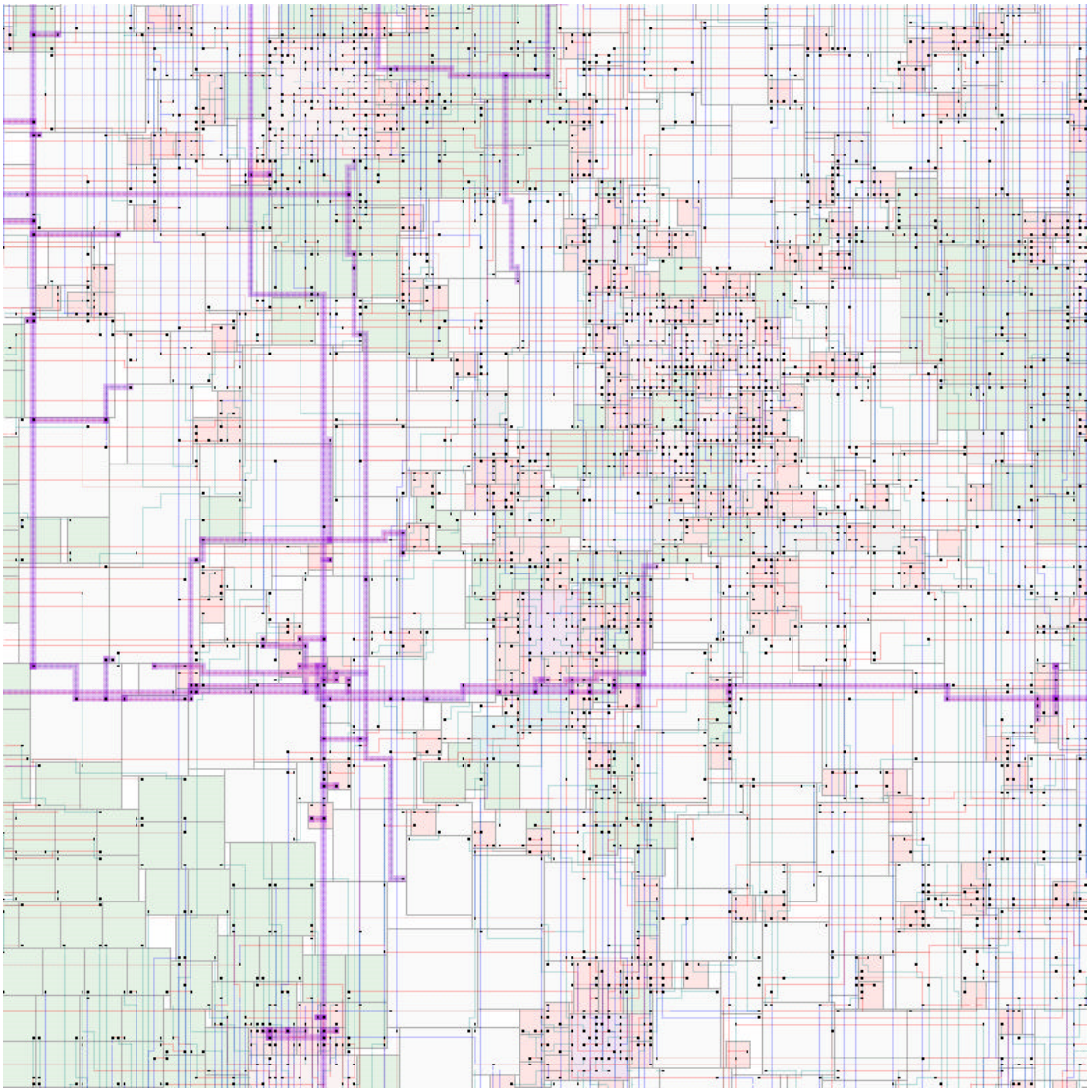


Figure 25: Inter-cell Routing Showing Selected Nets

An intuitive interface to edit routes by adding and deleting nodes is provided for the user. This usage is similar to that of other layout tools in industry (for example MAX). To add to a net, the user simply clicks on the net they want to extend. That net will become active and a wire on the metal layer that the user clicked will be drawn from the selected point to where the user next clicks. To change layers, the user simply uses the keyboard – the vias required for connection the different metal layers are created automatically by the tool.

In a complex design it is very easy to create erroneous routes and it can be cumbersome to identify and correct simple errors. ATL attempts to facilitate this process by supporting the automatic correction of certain erroneous routes. This includes the automatic removal of loops and extraneous paths. Routes that ATL cannot automatically resolve are identified to the user by highlighting the erroneous net.

5.1.3 Intra Cell Placement

The intra cell placement is visualized right on top of the inter cell placement. That is, instead of simply drawing the outline of rectangular blocks using different colours, the actual transistor layout of each block type is drawn. Drawing the internal transistor layout and routing for a complete tile can be quite slow because of the immense quantity of detail. However, when the visible area is smaller, such that only a portion of the tile is visible, the drawing speed becomes more bearable.

The inter cell placement can be toggled on and off through keyboard input. This feature is available to the user throughout the CAD flow – from initial placement to the final routed design.

5.2 Graphical Interface

The old ATL interface has several buttons on the right side of the display. A selected subset of these buttons and their associated functionalities were replicated in the new ATL. The buttons that were not ported over were deemed superfluous and more time was dedicated to adding additional features. This includes the ability to select, add and delete routing nodes.

Navigating around the ATL interface is quite simple and can be accomplished through both the keyboard and the mouse. The arrow keys allow the current viewport to be displaced while the "+" and "-" keys zoom in and out. This corresponds to the "U", "D", "L", R", "Zoom In" and "Zoom Out" buttons. The window feature allows the user to select an area to zoom into. Zoom fit instantly jumps the user back out to the initial viewport.

The export to postscript feature was superceded by the ability to take a snapshot of the display to an image file. Multiple image formats are supported, including PNG, JPG and GIF.

Figure 26 shows the old and new interface buttons.

Several keyboard inputs are also supported. A summary of the keys recognized include:

- `GUI Keyboard Input Legend:`
- `A: display all layers`
- `Z: display no layers`
- `J: toggle display of internal transistor layout of blocks`
- `[0-9]: toggle display for layer X`
- `When in ADD mode:`
- `d: up a layer`
- `D: down a layer`

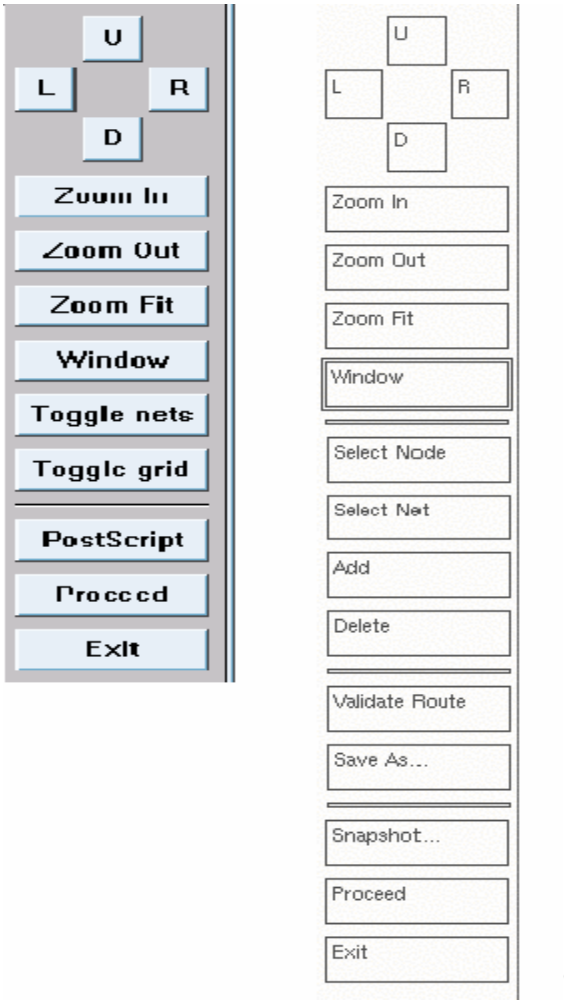


Figure 26: The old ATL interface vs. new OpenGL ATL

5.3 GUI Implementation

The graphical editor is implemented using OpenGL and has been tested under both Windows and Linux platforms. OpenGL is a mature graphics platform that is supported on multiple platforms. It features 32-bit RGBA (red-green-blue-alpha) colour, which enables the visual display of the routing to be more appealing. Further as an emerging standard many video cards have built-in hardware support for OpenGL which makes the graphics run faster.

Incorporating the GL Utility Toolkit (GLUT) eliminated the need for two files from the ATL project, atl_win_graphics.c and atl_x11_graphics.c. Previously, these were used to provide drawing functionality in Windows and Unix environments, respectively. A unified file, atl_graphics.c, was added to accomplish drawing in both Unix and Windows environments. This enhances code readability and maintainability while keeping cross platform compatibility.

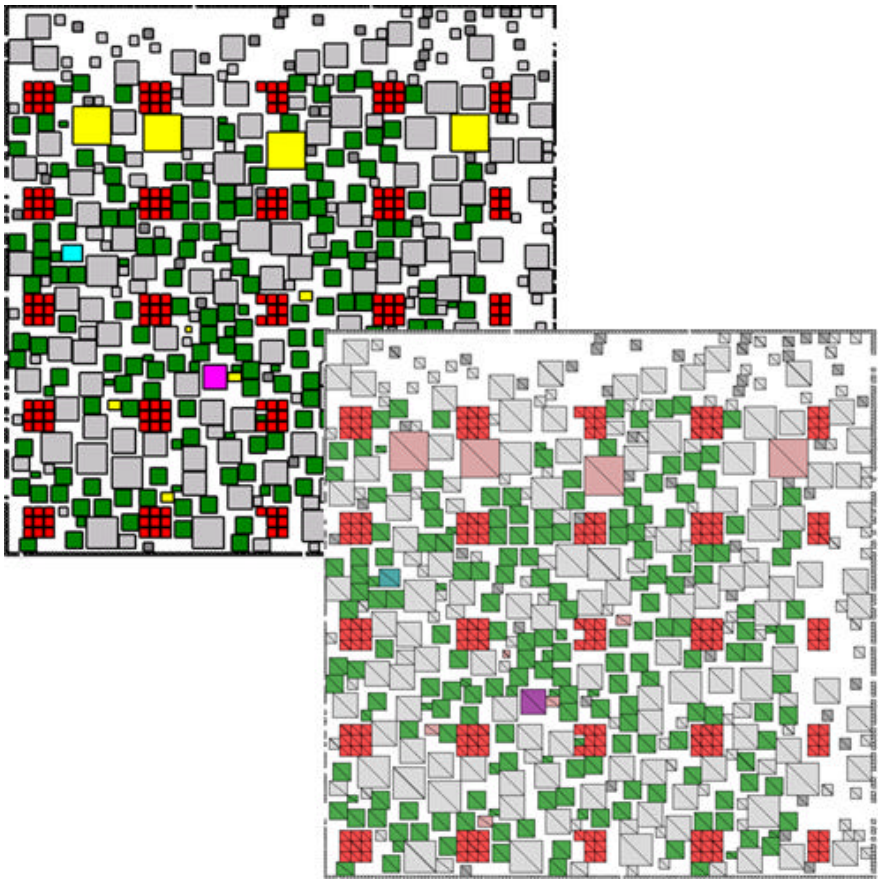


Figure 27: **Initial placement views (old vs. new) for "tile_1x4" circuit.**

The OpenGL view uses a double buffer system and therefore seems to be faster when refreshing the screen. In the old ATL program, the redrawing of each individual block can be seen when the window is being redrawn. Figure 27 shows the initial placement of the smallest test circuit under both the old and new ATL.

5.3.1 Interface Implementation

The existing export to Postscript button actually re-draws the placement using Postscript code. Using OpenGL, it is much simpler to export an image file than to generate Postscript code to represent the display. Hence, the export button was changed

to create an image file instead. Using the BitMapped Graphic Library (BMGlib), the new ATL interface can export PNG, JPG and TIF files.

In the interim report it was stated that there were problems with multiple sequential zoom requests. All such problems have been resolved and zoom now functions as expected under all conditions.

5.3.2 Inter Cell Placement Visualization Implementation

The inter cell placement consists primarily of coloured rectangles. Drawing all blocks in the cell level netlist is accomplished in O(N) time – loop over all blocks and draw a rectangle for each block. This was quite simple as GLUT provides primitives for drawing rectangles.

5.3.3 Routing Visualization Implementation

The inter cell routing consists of an immense number of routing nodes placed upon a routing grid. Each node on the routing grid can be flagged with the following properties:

```
#define ROUTE_GRID_NODE_TYPE_EMPTY          0x0000
#define ROUTE_GRID_NODE_TYPE_LEFT_WIRE      0x0001
#define ROUTE_GRID_NODE_TYPE_RIGHT_WIRE     0x0002
#define ROUTE_GRID_NODE_TYPE_TOP_WIRE       0x0004
#define ROUTE_GRID_NODE_TYPE_BOTTOM_WIRE    0x0008
#define ROUTE_GRID_NODE_TYPE_PORT           0x0010
#define ROUTE_GRID_NODE_TYPE_UPWARD_VIA     0x0020
#define ROUTE_GRID_NODE_TYPE_DOWNWARD_VIA   0x0040
#define ROUTE_GRID_NODE_TYPE_ILLEGAL        0xffff
```

Figure 28: Routing grid flags

Based on these types, each node on the routing grid is divided into a 3x3 grid and drawn using Table 7 as a reference. This 3x3 table shows the positions of possible contents inside the node. Using bit-wise comparisons of the current node property and the defined properties, information can be extracted to indicate exactly which pieces of the node need to be drawn. For occupied nodes on the routing grid, the centerpiece is always drawn. Vias and ports are drawn within the centerpiece. Adjacent routing nodes that form wires are drawn by filling in the appropriate entry as shown in Table 1. For example, If all of the wires adjacent to the centerpiece are used, the picture resembles a cross.

Always Empty	TOP_WIRE	Always Empty
LEFT_WIRE	CENTER PIECE (always drawn)	RIGHT_WIRE
Always Empty	BOTTOM_WIRE	Always Empty

Table 7: Position of metal segments in a routing grid node

Several problems related to the speed of the graphical editor were presented during the interim report and these have all been rectified. The largest tile data set available can be drawn in less than a second with all the routing displayed. This is considerable improvement over the minutes that the initial solution presented in the interim report required.

Because the number of routing nodes is significantly higher than the number of the number of blocks, an O(N) algorithm in terms of the number of used routing nodes is still too slow to be bearable.

The main source of the incredible speed increase is caused by a major code overhaul to draw nets instead of drawing nodes. This requires relatively complex preprocessing of the node connectivity based on the flags set on each node to create a new net-based data structure (DrawNet) storing only the end points of metal wires. This new data structure is created by reading the routing traceback and creating a list of endpoints that can be identified by comparing the flags assigned to each node. This data structure has a second benefit – it reduces the memory usage, as intermediary nodes do not have to be stored. The memory usage concern was mentioned in the interim report and was successfully resolved. Figure 29 (identical to Figure 17) shows an example of what information is stored in the DrawNet data structure for the routing of a 3-pin net.

Because the traceback does not change, computing the endpoints once at the beginning and caching these coordinates makes drawing the nets orders of magnitude faster than visiting all visible nodes and querying their properties.

1	2	3	4	5	6	7
8	9	10	11	12	Pin C 13	14
15	16	17	18	19	20	21
22	Pin A 23	24	25	26	27	28
29	30	31	32	33	34	35
36	37	38	Pin B 39	40	41	42
43	44	45	46	47	48	49

Associated Traceback: 23→24→25→32→39→XX→25→26→27→20→13
DrawNet Endpoints list: 23->25, 25->39,26->27,20->13

Figure 29: Drawing procedure Based On Traceback.

The modifications that a user makes to the routing are made in the graphics data structures. Two data structures are updated upon either a user add or delete command – an array of routing nodes that stores the properties of each node and the DrawNet structure. As such, the interface requires the user to add routing nodes to an existing net – routing nodes that do not belong to any net cannot be added. This simplifies the error recovery system considerably.

When the user issues a "Validate Route" command the route traceback data structure is reconstructed based on the graphics structure. This conversion process includes a simplistic error recovery system that can very quickly identify and remove some erroneous routes. This error recovery system is very fast and does not involve an

actual call to re-route the entire tile. Instead it traces the route that is displayed to the user by performing a depth first search starting from the source pin and searching along connected paths for the destination pins. During the depth first search, loops can be identified and removed. Further pins that cannot be found are flagged as a fatal error – the route is disconnected. In the event that a successful conversion is possible from the graphics structure to the route traceback data structure, the standard route checking routines are invoked. This verifies the occupancies and connectivity in a more extensive manner.

5.3.4 Intra Cell Placement Visualization

The internal transistor layout of each block can be quite complex and therefore drawing all transistors for an entire tile causes considerable slowdown to the graphics. This is alleviated somewhat by smartly only drawing the visible blocks, therefore in a zoomed in view the delay is negligible.

As the intra cell placement is presented as a set of rectangles, drawing this data is quite straightforward. The only minor problem resulted in the fact that the intra cell placement coordinate system did not match that of the inter cell placement. Because of these discrepancies in the coordinate systems, some scaling of the internal transistor layout must be performed. This scaling is done on a per block basis since there is also no consistency with regards to the coordinate system for the different block types. Figure 30 shows a zoomed in view of the tile with the internal transistor layout revealed. Figure 31 shows the same view but the internal transistor layout is hidden.

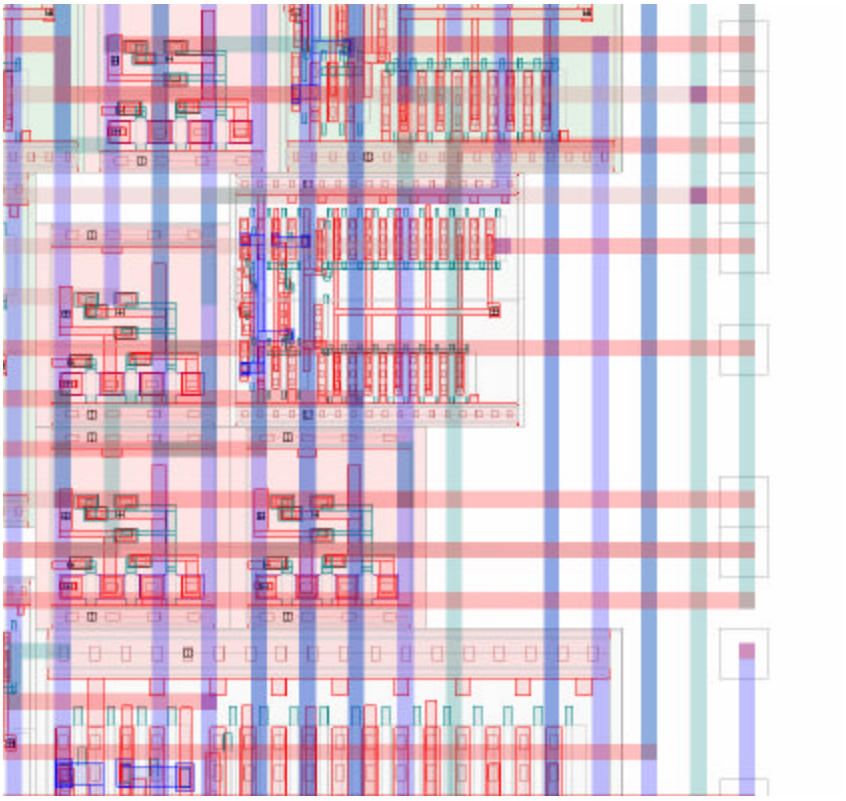


Figure 30: Inter-cell Routing With Transistor Level Layout displayed

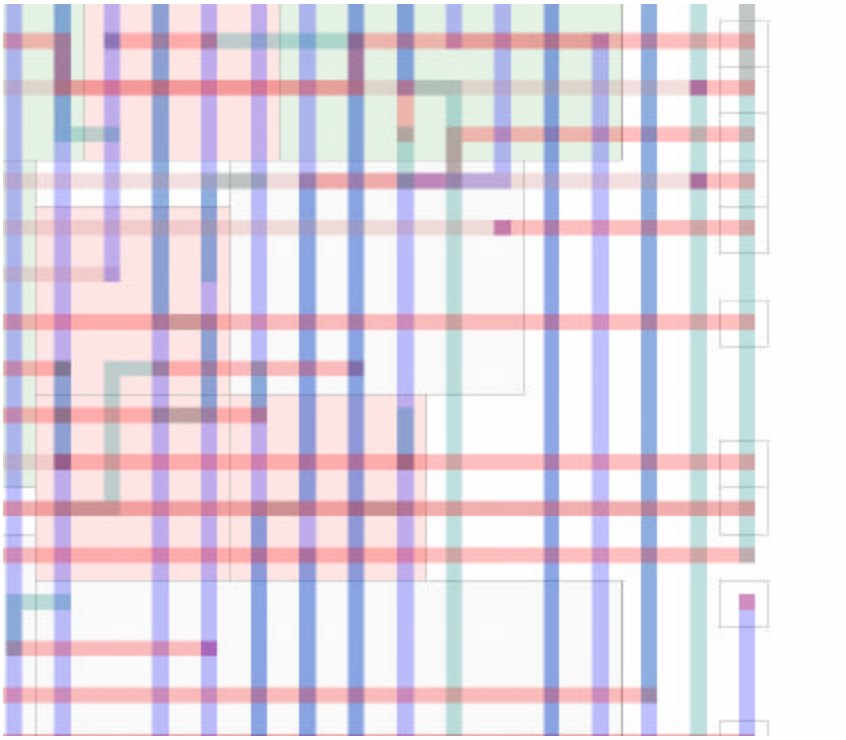


Figure 31: Inter-cell Routing With Transistor Level Layout hidden

6 Conclusions and Future Work

6.1 Summary and Contributions

Current design flow processes for Field Programmable Gate Array devices require an extensive amount of manual effort in order to define an electrically legal layout that satisfies all performance constraints. Due to escalating complexities in modern FPGA devices, many person-years of development time is required to develop a new architecture. An additional complication experienced in the early stages of design process is the inaccuracies in estimating the final performance of FPGA architectures, since the exact timing numbers can only be extracted after the layout is completed. Since decisions made at the beginning of the design cycle have a profound impact on the quality, performance, and routability of the architecture, the development of automated software tools that assist FPGA architects in *quantitatively* evaluating high-level architectural decisions is an essential step in improving the quality of FPGAs. An additional benefit is that CAD tools, in combination with human intuition and experience, can accelerate the design cycle for FPGA architectures from several person-years to a few person-months.

Betz *et al.* [4] identify three major factors that influence the performance of FPGA devices:

- The quality of the CAD tools used to map circuits into an FPGA

- The high-level architectural decisions of the FPGA, such as the global routing architecture, the detailed routing architecture, and the composition of the logic structures inside the FPGA
- The intrinsic quality of the transistor-level layout of the FPGA

Betz develops a detailed framework for investigating the first two of these three factors. VPR, Versatile Place-and-Route, contains a high-quality and highly *flexible* packer, placer, and router that position and connect the fundamental logic components that represent a digital design into an FPGA. Having high-quality CAD tools that map circuits into the FPGA are fundamental in accurately evaluating the effect of high-level architectural decisions on the overall performance of the architecture. This work points out that the area and delay information used in the tools to evaluate FPGA performance are developed using abstract models that estimate the area of the FPGA and delay between connections within the FPGA. The exact numbers cannot be obtained since a valid transistor-level layout is not available for these architectures.

Our design project is the extension of a CAD tool that performs the automated transistor-level design and layout of an FPGA – the unexplored factor that has a significant influence on the performance of FPGA devices. The ultimate goal of the CAD tool, ATL, is the creation of a high-performance electrically legal layout for an FPGA tile defined by a high-level architectural description. Ideally, ATL will generate a solution that uses a minimum amount of silicon area, has desirable timing characteristics,

and identify the solution in a short period of time. FPGA architects can utilize ATL, in concert with VPR, to obtain a *complete* CAD flow for evaluating high-level architectural decisions on the final performance of an FPGA.

We have added three important modules to ATL that increase the functionality of the application. First, a set of hand-optimized layouts was developed and functionally verified using commercial CAD tools. A parser was developed that converts these layouts into ATL's data structures and transforms the circuit netlist to reflect the information about the dimensions of each cell. Next, an inter-cell routing algorithm was created to define the width, position, and orientation of metal segments that electrically connect the terminals of the logic design, as specified by the circuit netlist. Finally, we produced a graphical user interface that is capable of displaying, editing, and saving both layout and routing information from the two other modules.

ATL is currently the only academic work that focuses on using domain specific knowledge to develop a transistor-level layout for an FPGA tile. Therefore, the feasibility of automated solution is an important question that our project is attempting to answer. In this paper, we have demonstrated that an automated tool can achieve compact, routable layouts for complex tile definitions in a few hours. This result proves there is potential for using automated tool to assist in the development of modern FPGA architectures.

6.2 Future Enhancements

Although we have demonstrated the feasibility of an automated solution, there are many additional considerations that are required to increase the quality and usefulness of ATL. There are several important components that need to be implemented before this tool can produce sufficient information for a semiconductor facility to generate a mask. The output of ATL needs to be converted to the standard mask definition language used in industry, GDS-II. This task is more complicated than a simple format translator due to the complex rules associated with the GDS-II standard. However, a considerable benefit of developing a GDS-II translator for ATL is that the output layout could be analyzed by commercial simulation and timing analysis applications to (1) validate the correctness and (2) compare the performance of ATL against alternative CAD tools that perform transistor-level layout. In order for ATL to generate layouts that can be implemented in silicon, the router needs to be enhanced to handle design rules involving exact distances instead of relying on scalable design rules. Additionally, more complex design rules need to be considered in the routing algorithm in order for the layouts to be implemented on deep sub-micron VLSI processes.

Several enhancements can be made to the routing module that would reduce the overall area (and potentially the minimum number of metal layers) required by the router. Modifications can be made to allow the router to use polysilicon for short connections and utilize the unused areas in the lower metal layers. Another enhancement to the tool would be the ability to generate a layout that considers “directed” timing requirements specified in the architecture file. For example, it is standard practice in recent FPGA

devices to skew the delays of the LUT inputs to increase the performance of the circuits implemented in FPGAs.

An additional extension to this project would be the ability to allow the floor planning of an entire FPGA based on various configurations of primitive tiles. This module would be responsible handling user specifications and the issues involved in the integration of various tile structures. This feature would allow ATL and its derivatives to be one step closer to generate the transistor-level layouts for *complete* FPGA architectures and produce legal mask definitions that are ready for the next phase in the fabrication process.

These examples of potential enhancements to the ATL application are a sample of the rich possibilities that exist in the field of automated transistor-level layout for FPGA devices and shed light on the potential that ATL will have on further research. This technology has the potential to have a tremendous impact on both the current and future design processes of FPGA devices. This work has taken an important step to the eventual realization of this technology.

7 References

[1] R. Hitchcock, G. Smith, and D. Cheng, "Timing Analysis of Computer-Hardware," *IBM Journal of Research and Development*, Jan. 1983, pp. 100 – 105.

[2] K. Padalia, "Automated Transistor-Level Design and Layout Placement of FPGA Logic and Routing from an Architectural Specification", *Undergraduate Thesis*, University of Toronto, 2001.

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

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

[5] C. J. Alpert et al., "Buffered Steiner Trees For Difficult Instances," *International Symposium on Physical Design*, 2001, pp. 4-9.

[6] Joobbani, Rostam, *An Artificial Intelligence Approach to VLSI Routing*. Hingham, Massachusetts: Kluwer Academic Publishers, 1986.

[7] V. Betz, "VPR and T-VPack: Versatile Packing, Placement and Routing for FPGAs", [Online demo application], [Cited Jan. 9, 2002], Available HTTP: http://www.eecg.toronto.edu/~vaughn/vpr/vpr.html. Last checked: December 2001.

[8] MOSIS Corporation, "MOSIS Scalable CMOS (SCMOS) Design Rules," [Online document], [Cited Jan. 9, 2002], Available HTTP: http://www.mosis.org/Technical/Designrules/scmos/scmos-main.html

[9] C. Ebeling, L. McMurchie, S. A. Hauck and S. Burns, "Placement and Routing Tools for the Triptych FPGA," *IEEE Trans. On VLSI*, Dec. 1995, pp. 473-482.

[10] Rubin, Steven M., *Computer Aids for VLSI Design*. Reading, Massachusetts: Addison-Wesley Publishing Company, 1987.

[11] Preas, Bryan T., and Michael J. Lorenzetti, *Physical Design Automation of VLSI Systems*. Menlo Park, CA: Benjamin/Cummings Publishing Company, 1988.

[12] C. Y. Lee, "An Algorithm for Path Connections and its Applications," *IRE Trans. Electron. Comput.*, Vol. EC=10, 1961, pp. 346-365.

[13] E. Dijkstra, "A Note on Two Problems in Connexion with Graphs," *Numerical Mathematics*, Vol. 1, 1959, pp. 269-271.

[14] F. Rubin, "The Lee Path Connection Algorithm," *IEEE Trans. Computers*, Sept. 1974, pp. 907-914.

[15] R. Nair, "A Simple Yet Effective Technique for Global Wiring," *IEEE Trans. On CAD*, March 1987, pp. 165-172.

[16] Weste, Neil H. E., and Kamran Eshragian, *Principles of CMOS VLSI Design*, 1st Ed., Reading, MA: Addison-Wesley, 1985.

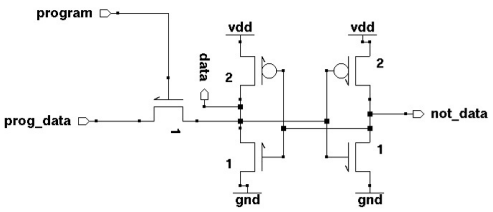
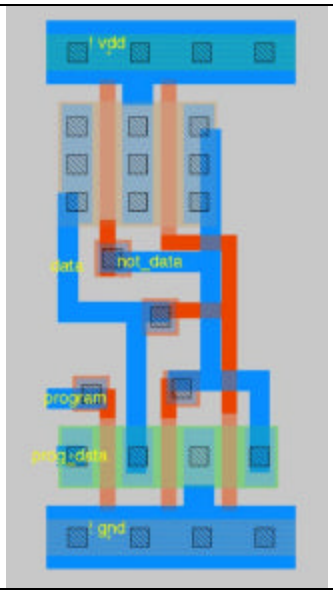
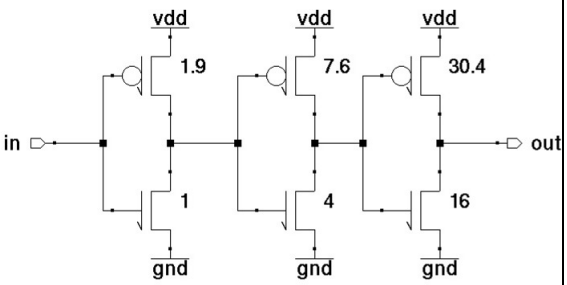
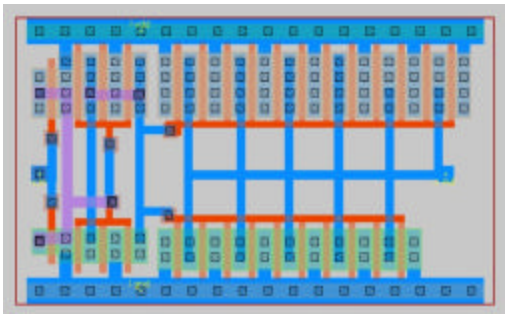
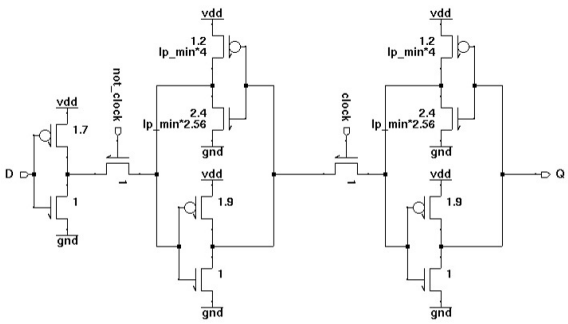
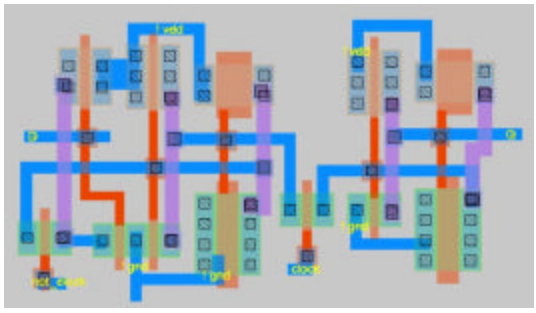
[17] Electronic Industries Alliance, "Electronic Data Interchange Format," [Online document], [Cited April 11, 2002], Available HTTP: http://www.edif.org/news.html

[18] Altera Corporation, "Altera Corporation Glossary (July 2001)," [Online document], [Cited April 11, 2002], Available HTTP: http://www.altera.com/literature/glossary/glossary.pdf

[19] M. Kaufmann and K. Mehlhorn, "Routing Problems in Grid Graphs," *Paths, Flows, and VLSI-Layout*, 1990, pp. 165-184.

APPENDIX A: Cell Schematic and Layout Library

The following table lists the major cell types used within the FPGA architectures considered in this project complete with a schematic representation and a sample layout .

Cell Name	Schematic Representation	Layout Representation
SRAM		
Buffer		
Flip-Flop		

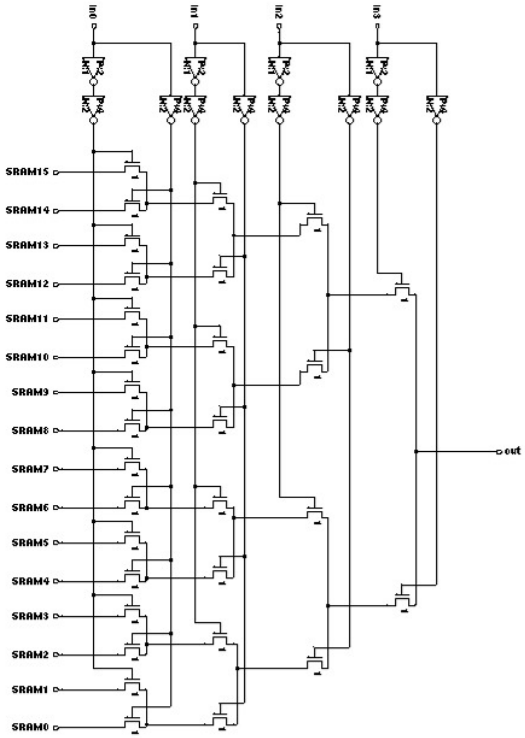
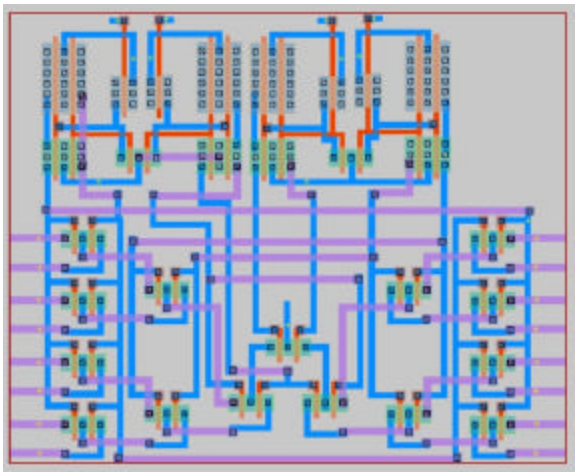
Look-up Table		

Table 8: Cell schematics and layouts