Automatic Transistor and Physical Design of FPGA Tiles From An Architectural Specification

Ketan Padalia, Ryan Fung, Mark Bourgeault,
Aaron Egier, and Jonathan Rose

Edward S. Rogers Sr. Department of ECE
University of Toronto
Toronto, Ontario, Canada M5S 3G4

ketan.padalia@utoronto.ca, ryan.fung@utoronto.ca, mbourgea@rogers.com
aegier@eecg.toronto.edu, jayar@eecg.toronto.edu

ABSTRACT

One of the most difficult and time-consuming steps in the creation of an FPGA is its transistor-level design and physical layout. Modern commercial FPGAs typically consume anywhere from 50 to 200 man-years simply in the layout step. To date, automated tools have only been employed in small parts of the periphery and programming circuitry. The core tiles, which are repeated many times, are subject to painstaking manual design and layout. In this paper we present a new system (called GILES, for Good Instant Layout of Erasable Semiconductors) that automatically generates a transistor-level schematic from a high-level architectural specification of an FPGA. It also generates a cell-level netlist that is placed and routed automatically. The architectural specification is the one used as input to the VPR [3] architectural exploration tool. The output is the mask-level layout of a single tile that can be replicated to form an FPGA array. We describe a new placement tool that simultaneously places and compacts the layout to minimize white space and wiring demand, and a special-purpose router built for this task.

GILES can place and route a tile consisting of four 4-input LUT logic cells and all of its programmable wires in a 0.18µm CMOS process using 8 layers of metal and 25983µm^2 of area. When we generate the layout of an architecture similar to the Xilinx Virtex-E FPGA (built in a 0.18µm process) GILES requires only 47% more area than the original. The layout area of an architecture similar to the Altera Apex 20K400E (also built in a 0.18µm process) constructed by GILES requires 97% more area than the original.

FPGA'03, February 23-25, 2003, Monterey, California, USA.

Keywords

FPGA, PLD, programmable logic, automatic layout

1. Introduction

The creation of a new FPGA requires a huge undertaking of manpower, starting with planning that creates the specification of the device features followed by a vast engineering effort to create high-quality implementations of those features, as well as the core programmable logic. A particularly labour-intensive and time-consuming part of this process is the transistor-level design and layout of the FPGA's masks. The latter typically takes from 9 months to a year, and currently requires the efforts of more than 100 people. It has been considered a manual task because only humans (as opposed to computers) were deemed capable of achieving the quality of results required. Highly efficient layouts are required because an FPGA tile is repeated many times, currently on the order of 10,000.

In this paper, we present a system that automates the transistor-level design and layout of the FPGA. It is based on the VPR [3] architecture exploration system, which is an architecture-retargettable packing, placement and routing system. FPGA architects use VPR to explore different architectural alternatives by changing the architecture across a spectrum of choices, and running a number of benchmark circuits through each architecture [2][3][4][6][7][17]. The output of VPR provides circuit speed and area requirements for each circuit implemented on each architecture, allowing the architect to determine the value of different choices.

One of the inputs to the VPR architectural exploration system is a file that describes the architecture of the logic block and its surrounding programmable routing. This compact, human-readable file specifies the number of lookup tables in a clustered logic block [4], the number of inputs to the logic block, the amount of connectivity from the main routing into the block and at the intersection of the main routing channels, the length of the routing wires, the types of programmable switches associated with different programmable routing wires, and the input/output pins and their connectivity to the general routing. It is sufficient to

describe an FPGA to the level needed to achieve any input circuit's packing, placement and detailed routing. It is also sufficient, with some small amount of added knowledge, to create the transistor-level circuit for the single FPGA tile that can be replicated to create an FPGA array. In this work we do exactly that, and create a higher, cell-level netlist describing the interconnections of basic units such as multiplexers, buffers, SRAM cells and, flip-flops.

We have built a placement/compactor for those cells, and a multi-layer router to connect them, in an attempt to fully automate the architecture-to-layout process. Figure 1 gives an overview of the flow of this system. Note that it relies on the input of manually performed cell-level layout.

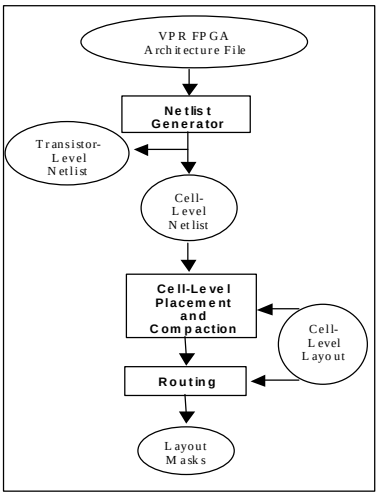


Figure 1 - Overview of Flow

Since this work is specific to FPGA layout, we will be able to use domain-specific knowledge in the new tools that are created – because we know the end target is an FPGA, we have extra knowledge about certain circuit elements that can be leveraged (an example of which is that all programming bits can be considered logically equivalent). While it may have been possible to use commercial layout tools to attempt the same work, it would have been impossible to make use of this kind of domain-specific knowledge.

There have been previous attempts at this type of work. Automated transistor-level layout of large systems has been an active area of research [8][10]. Cadabra [11] has a commercial tool for automating layout of standard cells, but that system is limited to cells on the order of 100, not the 10,000-20,000 transistors more typical in an FPGA tile. Azegami et al. [12] describe an FPGA built on top of a gate array, which benefits from automated layout. They worked with a single architecture (as opposed to the broad range of architectures that GILES can implement) and the implementation would suffer from the additional inefficiency of a gate array. Phillips and Hauck [13] describe a system for automating the layout of a specific architecture using a standard cell flow. This system allows portions of the architecture that are not required by a specific application to be eliminated, reducing area requirements, but does not automate a layout from a base specification as we do here.

This paper is organized as follows: the following section describes the generation of transistor- and cell-level netlists from the VPR FPGA architecture description file, including issues related to transistor sizing and tile replication. Section 3 describes the manual mask-level layout of the individual cells. Section 4 describes a novel combined placement and 2-D compaction algorithm that simultaneously minimizes area and wirelength. Section 5 describes a multi-layer router for forming the connections between cells. Section 6 gives a sample of the tool output across a number of architectures and gives a comparison to two commercial devices. Section 7 concludes and presents avenues for future work.

2. Netlist Generation and Tiling Issues

As discussed above, the input to the system is the same architecture description file used by the VPR FPGA architecture exploration system [3]. A description of that file format can be found online [17]. The outputs of the first phase of the system are two netlists. The first is the transistor-level design of a single logic and routing tile of the architecture described by the architecture file. The second is a netlist of higher-level cells that describe that same tile. These cells consist of SRAM programming bits, multiplexers, buffers, inverters, pass transistor switches, flip-flops and LUTs. The following sections describe the input and output in more detail and the issues that arise in their automated generation.

2.1 FPGA Architecture Input File

The primary input is a human-readable description of the FPGA architecture. The architecture file can be used to describe a wide variety of FPGA architectures. Figure 2 shows a section from the architecture file that would be used to describe the FPGA architecture depicted in Figure 3.

```
# Simple Architecture Description – 1 2-LUT cluster
# with all length-4 wires

# Logic architecture parameters
subblock_lut_size 2   # Using 2-LUT BLEs
subblocks_per_clb 1   # One 2-LUT per tile

# Routing architecture parameters
switch_block_type subset
Fc_output 1
Fc_input 1

switch 0 buffered: yes  R: 1000.0  Cin: 1.0e-15 \
Cout: 1.0e-15

# All buffered length 1 wires
segment frequency: 1.0 length: 1 wire_switch: 0 \
opin_switch: 0 Frac_cb: 1 Frac_sb: 1 \
Rmetal: 100.0 Cmetal: 1.0e-14

# Process parameters
R_minW_nmos 5000
R_minW_pmos 10000
```

Figure 2 – Example Architecture File Section

This architecture file specifies that every tile of the FPGA will have one 2-input look-up table (LUT) and a routing architecture with all buffered length-1 wires. It describes the

switch-box and connection-box connectivity parameters Fc and Fs [3], as well as the different types of switches used in the architecture (there is only one switch in this example). It also specifies process parameters that are used for sizing transistors. Figure 3 shows the cell-level netlist that represents the architecture from Figure 2. For simplicity, the set of routing switches in the top-left corner has been replaced by one buffered switch – in the real cell netlist, every routing switch would be created in this way. Also missing from the figure are the word lines and bit lines used to program the configuration bits.

2.2 Transistor-Level Netlists

To turn the architecture file specification into a transistor-level netlist that represents a tile of the FPGA, the netlist generator assumes specific transistor-level structures for each of the components in the FPGA. These are presented in detail in [1] and [3]. Figure 4 shows an example of the schematics we assume for the multiplexers in our FPGA tile.

Most of the transistors in our netlists are minimum size, except for those that are used to make buffers. The buffers are sized to provide a certain drive capability, as was done in [3]. The driver resistance information specified in the architecture file also affects the driver size.

2.3 Generation of Cell-Level Netlists

The netlist generator also creates cell-level netlists that abstract away the detailed transistor-level circuitry of the FPGA tile. These netlists are placed and routed to form a cell-level layout that, combined with the layout of the various cell types, form the final layout of the tile.

2.4 Tileability Constraints

For the layout to represent a tile that can be repeated to form a full FPGA core, we must make sure that the signals entering or leaving the tile are located so that they connect to the appropriate points on the replicated adjacent tiles on all 4 sides. To incorporate this information in the netlist, the netlist generator creates *ports* to represent the signals that enter or leave the tile. It also specifies constraints on those ports that say which edge of the tile each port must be placed on and indicates pairs of ports that must be kept opposite to each other (at the same vertical or horizontal position). These constraints ensure that adjacent tiles will have the appropriate wires connected together.

3. Manual Layout of Cells

We have chosen to work with a cell-level netlist in order to reduce the complexity of the transistor-level placement problem. As such, we require manual layout of the cells in the netlist. The total number of unique cells in a typical netlist is roughly 15, a relatively small number. For the results presented in Section 6 that compare directly to commercial devices, the full manual layout of each cell was done. For the other results, we approximate the area of the cells as an inflated function of the total minimum width transistor area [3]. We considered each minimum width transistor to occupy 2.25 squares in the routing grid (to allow for the intra-cell routing) and made sure there was also enough area to accommodate connections to inter-cell routing. We checked this approximation by comparing the layout of an FPGA tile using the estimated cell sizes against the tile area using actual layouts. The approximate layouts resulted in final layouts that were 35% to 75% optimistic compared to the actual cell layouts.

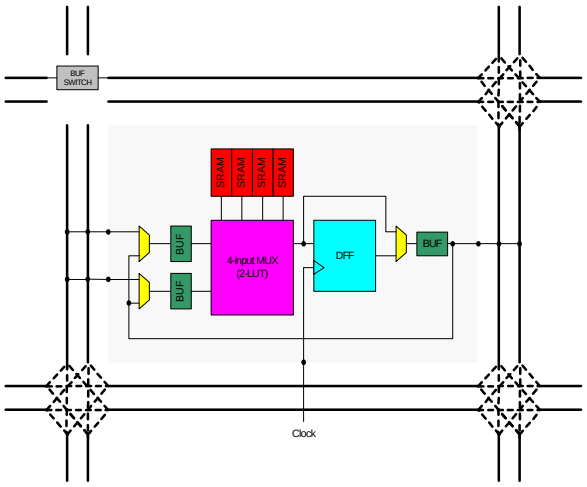
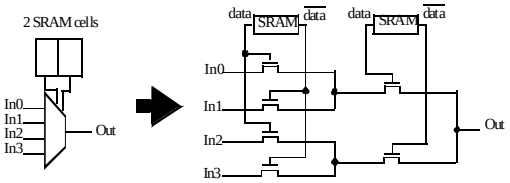


Figure 3 – Cell Netlist of Architecture Described in Figure 2



Schematic of a 4-input SRAM-controlled multiplexer.

Figure 4 – Multiplexer Schematic

Our layout methodology is to allow the cells to use all active layers and the first two metal layers to form connections. Other metal layers, as described in Section 5, are used for inter-cell routing.

4. Placement and Compaction

Once the cell-level netlist has been generated and the dimensions and pin positions of each cell are known, the cells are passed to the placement and compaction tool. An important strength of this work is that the placement and 2-D compaction are combined into one optimization step. In this section we describe the highlights of the algorithm, some details of our move generation and cost function, as well as an FPGA-specific algorithmic optimization. For a detailed treatment of the complete algorithm, see [5].

4.1 Goals, Constraints and Grid

The goal of the placement and compaction step is to determine the positions of cells and ports in a rectangular FPGA tile while respecting any tileability constraints of the form

described in Section 2.4. We allow up to two ports to be placed at a single port position since they can be routed to on different metal layers. If doubling the port density is not done, the port perimeter became the limiting factor of the cell size for reasonable architectures. The units of the cell dimensions are based on the routing grid used by the router described in Section 5. The routing grid is sized to the width of a routing track plus inter-track spacing (assuming worst-case via spacing).

4.2 Overview of Algorithm

The engine is based on the simulated-annealing algorithm [9][15]. An overview of the algorithm is shown in Figure 5.

The placement/compaction algorithm begins by optimizing a random initial placement of cells and ports. Initially, to optimize without being impeded by issues of cell overlap, a large-tile placement is used that keeps the cells spaced apart. Each cell occupies an MxM square large enough to accommodate the largest cell. The initial tile is kept as square as possible with enough space around the perimeter to accommodate all the ports. By spacing out the cells in this manner, a globally good positioning of cells can be found during this initial step. This technique, compared to an alternative starting with a random cell arrangement (in a tile sized 1.4 times the cell area [1]), reduces final wirelength by 33% (for the same run time and white space in the end).

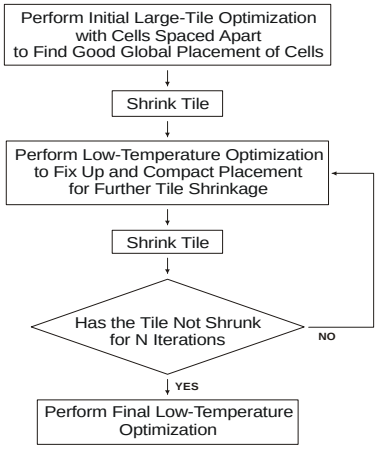


Figure 5 – Placement/Compaction Algorithm

After the initial optimization step, the algorithm alternates between tile shrinkage and low temperature simulated-annealing optimization steps. Tile shrinkage involves collapsing the port perimeter tightly around the cells in the interior of the tile. The cell and port positions relative to each other are preserved. Figure 6 illustrates the tile shrinkage operation. The low-temperature optimization steps have two goals: to optimize the placement of the cells to improve wirelength and to arrange and gather the cells in the interior of the tile such that tile shrinkage becomes possible.

There is an optimization trade-off present in the selection of the starting temperature of the low-temperature optimization steps. A higher temperature allows better "hill climbing" – perhaps better exploring the cost space to find better layouts. A lower temperature preserves more of the previous placement decisions. We found that the move acceptance ratio is a good measure of progress independent of the circuit (precise temperature values are a function of cost and hence are netlist-dependent). Therefore, we selected the starting temperature to be the temperature that achieves a particular acceptance ratio during the initial large-tile optimization. Experiments showed that an acceptance ratio of 0.29 yields good results.

4.3 Cost Function

The cost function of the annealing engine focuses on the estimated total wirelength of the placement, with several modifications to encourage the compaction of the cells. The wirelength cost is based on the semi-perimeter net bounding-box metric used in [3][16].

4.3.1 Tile Size Cost

The *tile-size cost* is used to encourage compaction, and operates on an imaginary bounding box enclosing all the cells (excluding ports) in the tile. It has two components: the first penalizes moves that increase the area of the imaginary bounding box. This encourages "crunching" of the cells away from the tile perimeter to facilitate tile shrinkage during the next compaction. The second cost component penalizes moves that increase the number of cells on the perimeter of the imaginary bounding box. This encourages evacuation of the imaginary bounding box perimeter, gradually making tile shrinkage possible. The tile-size cost function is defined as follows:

$$\begin{aligned}\text{cost} = (&\text{width}_{\text{imaginary bounding box}} \times \text{height}_{\text{imaginary bounding box}}) + \\ &(\text{num\_blocks}_{\text{left side}} + \text{num\_blocks}_{\text{right side}} + \\ &\text{num\_blocks}_{\text{top side}} + \text{num\_blocks}_{\text{bottom side}})\end{aligned}$$

The area of the imaginary bounding box is used for the first component instead of the perimeter to give it a greater weight since it is directly related to whether the tile can be shrunk or not.

A multiplier is applied to the tile-size cost so that it can be weighted with respect to the bounding-box wirelength cost. The multiplier is calculated so that the weighted tile-size cost is a certain multiple X of the bounding-box wirelength cost at the beginning of each low-temperature optimization step. The value of X is increased over the course of placement, between the low-temperature optimization steps, but is set to 0 for the final low-temperature optimization. Increasing the tile-size maximum cost fraction from 0.01 to 5 cuts tile white space in half, halves the run time, and improves wirelength by 3.5% (it turns out that completely "turning off" this cost causes placement run times to grow too large to get similar quality results since tile compaction occurs randomly over long periods of time).

4.4 Move Generation

We chose to implement a move generator that only proposes moves that maintain the legality of the placement. This means that cell overlapping is not permitted. This has the effect of reducing the size of the search space, but increasing the complexity of the move generator. We have two types of move: general moves and compaction-oriented moves.

4.4.1 *General Moves*

The general moves begin by proposing a translation for a random cell **A**. They then propose additional translations for other cells to make room for cell **A** and cells displaced by cell **A**. If a set of translations cannot be found that constitute a move that will preserve placement legality, the move is aborted.

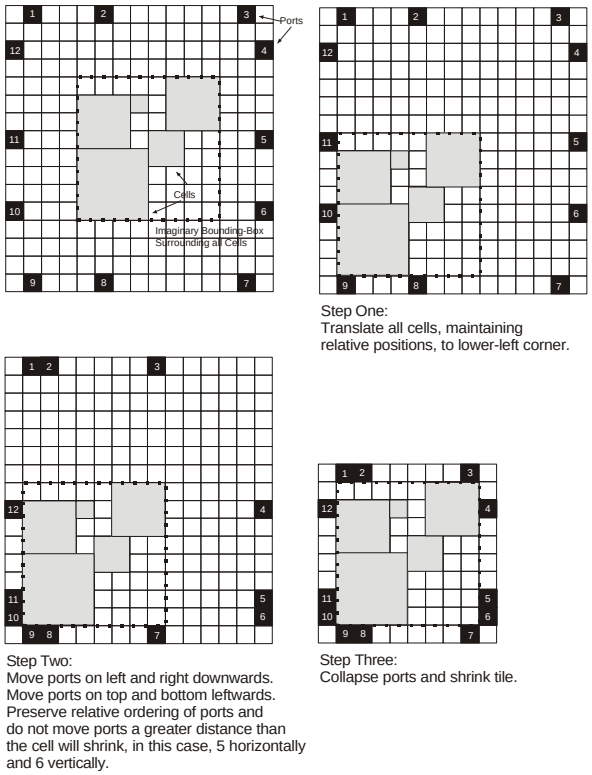


Figure 6 – Illustration of Tile Shrinkage

A range limiter is used to limit the distance a single move can transport a cell or port; it is kept constant during a placement temperature and is gradually reduced over the course of placement based on measures of move acceptance and the temperature update schedule [15].

4.4.2 *Compaction-Oriented Moves*

Two specialized moves facilitate tile compaction. The first, the *block-off-edge move*, is designed to encourage cells to move off of the perimeter of the imaginary bounding box monitored by the tile-size cost. This encourages the collapse of the imaginary bounding box, hopefully leading to eventual tile compaction. This type of move is needed because once the general locations of blocks settle during placement, the move acceptance function tends to reject moves that span a great distance (produce a large cost increase) and the range limiter of the general move generator tends to constrain translations (moves) to a local region. To shrink the tile, however, moves that transport cells across a large distance sometimes have to be accepted because the cells must be moved to locations with enough white space to accept them.

This move type addresses both cost arbitration and move generation issues that would otherwise prevent the type of long-distance move often needed for the sake of tile compaction. This move reduces the magnitude of cost increase associated with moves that successfully move cells off of the perimeter (to make acceptance of those moves more likely); it also removes the standard range-limit, imposing its own range-limit geared to explore moving a cell off of the perimeter of the imaginary bounding-box. Figure 7 illustrates a situation during placement when a block-off-edge move would move a cell on the perimeter of a tile a long distance to an empty area. This technique is an important one, as it cuts the final white space in half and reduces run-time by 19% without affecting wirelength.

A second specialized move is the *compaction move*. It is a move type involving many cells in a focused effort to take immediate advantage of gaps in the placement and for cells to move closer to the center of the tile. A series of inward one-unit shifts are proposed for all the cells involved in the move. This move type begins with cells closer to the centre so gaps are opened that outer cells can move in to. Each individual cell move is arbitrated with the same cost function as above; it is the pre-determined sequence of moves (as opposed to random choices) that makes these moves effective.

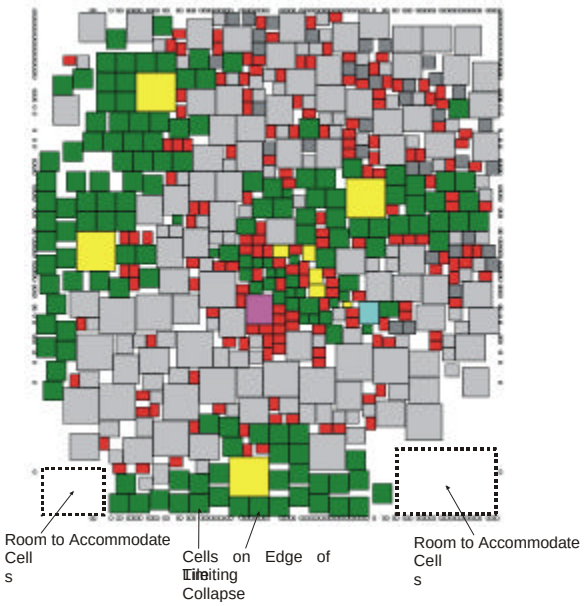


Figure 7 – Motivation for *Block-off-Edge* Move

Even with tile-compaction cost bonuses designed to benefit moves that lead to smaller tiles, combinations and sequences of those moves have to be proposed by the move generator for tile compaction to result. It is unlikely that the move generator will randomly be able to generate the proper sequences frequently. Compaction moves solve this by attempting to move blocks to the center of the tile one after another. By making compaction a priority of the move generator, we can do more efficient and effective tile compaction. During experimentation with this move type, we determined that multiple-unit (distance) shifts showed no advantage over single-unit shifts. Also, one-dimensional shifts were determined to be as effective as two-dimensional shifts.

Since compaction moves affect all cells, they are performed relatively infrequently. If compaction moves are not performed,

the run time of the tool to get equivalent quality increases significantly (because compaction occurs very slowly). When 4 compaction moves are performed per temperature instead of 0, the average run time was reduced by more than a factor of five and the average white space was reduced by a factor of 25.

4.5 FPGA-Specific Placement

Since this placement tool is designed for FPGA-based cells, we can leverage domain-specific knowledge. For example, the SRAM configuration bits of an FPGA are typically organized with word lines and bit lines just as in normal SRAMs. Word-and-bit-line assignments are arbitrary because FPGA SRAM programming can be adjusted to program the target SRAMs to any values for any fixed word and bit line assignment. In our placement algorithm, we used a technique called *SRAM reweaving* to leverage this. Our placement optimizer initially places SRAMs without considering costs for word and bit-line nets. After all the SRAMs are assigned good global positions (i.e. close to the cells they are attached to), word and bit lines can be assigned to SRAMs (rewoven) based on where they have been placed. We thereby leverage the arbitrariness of word and bit line assignments to minimize word and bit line length and to avoid consideration of programming line length when determining good positions for SRAMs.

The word and bit lines are ignored during the initial placement optimization, but are rewoven between the low-temperature optimizations. This technique saves 11% wirelength without affecting run time or white space.

5. Routing

The output from the placement phase provides the absolute position, size, and orientation of each cell or tile port in the cell-level netlist. These are passed to the routing step, which determines the detailed routing for connections between the cells. The input to the routing phase is the placed netlist, the dimensions of the tile, the number of metal layers available for inter-cell routing, and the metal and via wiring pitch (which is assumed to be the same for all layers).

The routing algorithm is an extension of the classical maze router approach [20], with elements incorporated and adapted from various FPGA routing algorithms [3][19]. In the next section we describe the layer planning and basic routing grid definition used by the router. We then proceed to describe the enhancements we made to the basic router algorithm for this tool. For a detailed treatment of the complete algorithm, see [21].

5.1 Layer Planning

We divide the routing connections into two distinct types: (1) intra-cell routing, which are connections localized within a single cell, (2) inter-cell routing, which consist of connections between two or more cells/ports within the FPGA tile. We also recognize there are specialized structures for power, ground, and clock nets (e.g. H-trees). We route these special signals between the cells, but not the distribution networks for the specialized structures (which are typically on the top metal layer(s)) or the routing to them. We assume a small amount of space between tiles and an additional metal layer is sufficient for these nets and the routing to them.

As described in Section 3, the layout for each cell type is manually created using the MAX [14] tool. Two metal layers (M1 and M2) have been allocated for all the intra-cell routing connections. The next n layers – where n is an input to the CAD tool – are used for inter-cell routing connections and for routing power/ground/clock signals between the cells. The link between intra-cell layouts (created manually) and inter-cell routing is realized by placing a via between metal 2 and metal 3 at the location of every cell pin.

We employ a uniform, three-dimensional routing grid to represent the metal layers available for inter-cell routing connections. Each layer of the structure is partitioned into a 2-D array of equally sized routing grid squares; the size of a grid square is chosen such that two wire segments carrying different electrical signals can be positioned in adjacent squares, as is common in maze-grid type routers [20]. For the results presented in this paper, the grid size is chosen to be appropriate for a 0.18 μm CMOS process, which, based on data from MOSIS [18], is a 0.66μm x 0.66μm region.

5.2 Router Algorithm & Enhancements

The router utilizes an iterative rip-up/reroute strategy that is combined with the negotiated congestion approach [15]. The cost function for using a routing grid node is based on the approach taken by VPR [3], but is modified to include various "tile-wide" routing directives, which ultimately provide better routing results.

First, we guide the router to route in only one orientation (horizontal or vertical) on any given layer by designating a preferred orientation for each layer. Second, we encourage the router to minimize the number of vias it uses. Third, experimentation determined that congestion was often found on the some edges of the tile, and so we encourage routes to avoid these edges. To achieve these goals, the cost function for each grid node used in a routing path is augmented by a penalty function for each routing directive. Specifically, the *PathCost* formula (see VPR [3]) is modified in the following manner to account for the effects introduced by bias factors:

$$PathCost(n) = PathCost(m) + PresCost(n) \cdot HistCost(n) \cdot \prod_{i=0}^{num\_factors} BiasFactorPenalty_i(n)$$

$BiasFactorPenalty_i(n)$ is the i^{th} penalty for using node *n*. Note that $BiasFactorPenalty_i(n)$ is always greater (or equal to) than unity.

If the router fails on a given placement, an outer loop of the algorithm determines the areas of the FPGA tile that are the most congested. White space is added to the original cell-level placement in this area, and another routing attempt begins.

6. Results

In this section we describe the use of GILES to build several tiles and make comparisons to two commercial devices. These comparisons will be based on a 0.18µm CMOS process [18]. Figure 8 shows the placement and compaction of a simple architecture that consists of four 4-input lookup table and flip flop basic logic elements, 16 length 4 tracks with buffered switches and 16 length 4 tracks with un-buffered switches, Fs=3, Fcin=0.56 and Fcout=1 [3]. It is interesting to look at this picture in colour (which you can do if you are reading this paper onscreen or have printed it in colour) with the legend given in Table 1 and see where different types of the cells are placed.

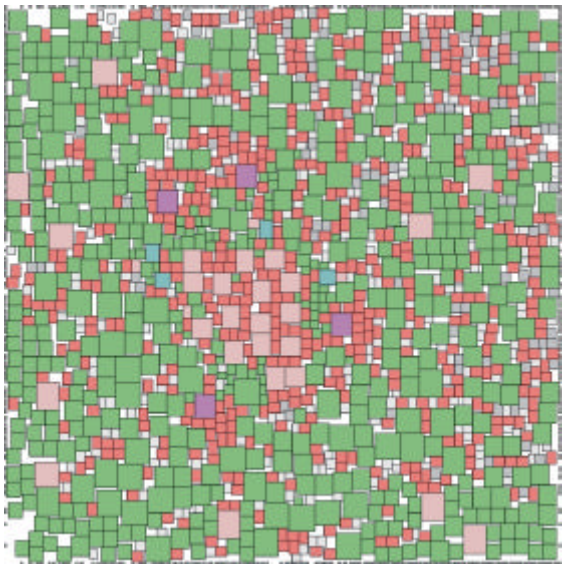


Figure 8 – Placement of 4x4-LUT Architecture

Cell Type	Colour
Buffer	Green
Configuration SRAM	Red
Multiplexer	Pink
LUT	Purple
Flip-Flop	Light Blue
Pass Transistor Switch	White
Buffered Switch	Grey

Table 1 – Colour Legend for Placement Picture

For this architecture we did not do the full transistor layout, but rather approximated cell areas as described in Section 3. Figure 9 shows the full placed and routed layout of this same tile. This tile required 7 layers of metal (excluding routing for the specialized networks but including power, ground, and clock routing between cells), and took up dimensions 84 x 81µm. Table 2 gives a set of results for ten automatically generated FPGA tiles. The table gives the number of 4-input lookup tables per cluster and the number of tracks per channel specified in the architecture file. Half the tracks are length four buffered segments and half are un-buffered. The number of metal layers indicated in Table 2 is one greater than the number the tool used; the extra layer is added to try to account for the specialized distribution networks, as discussed earlier.

The number of tracks is selected to be a reasonable number from our previous experimental experience. The fourth and fifth column give the value of the connection block input and output flexibility. In all ten examples, the switch block flexibility, Fs, is 3. We give the number of metal layers required to route the tile, including the metal used to route within the basic cells and the power and ground routing. The next three columns give the dimensions and area of the tile. The final column gives the runtime of the entire tool in seconds on a 1GHz Pentium 3 processor. Table 2 illustrates the power of our tool, which can layout radically different architectures.

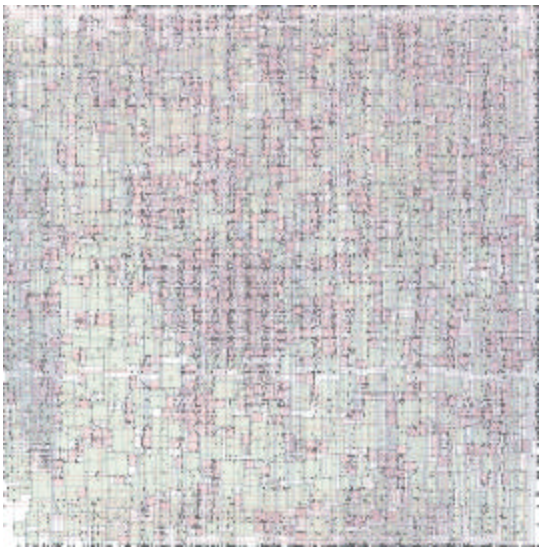


Figure 9 - Routed 4x4-LUT Architecture

# LUTs	# Track	Fc In	Fc Out	Metal Layers	Tile Width (um)	Tile Height (um)	Final Routed Area (um^2)	Total Run Time (s)
1	32	0.56	1.00	7	84	81	6805	113
2	56	0.44	0.50	8	115	108	12430	585
3	80	0.30	0.33	8	143	134	19100	1174
4	96	0.23	0.25	8	169	154	25983	4029
5	120	0.19	0.20	8	184	179	32935	4520
6	144	0.15	0.17	8	209	203	42392	8889
7	160	0.13	0.14	8	249	228	56821	18427
8	176	0.11	0.13	8	246	255	62717	14755
9	192	0.10	0.11	8	261	281	73126	23397
10	200	0.10	0.10	8	304	275	83557	30475

Table 2 – Architectural Specifcation and Layout Results for Ten Different FPGAs

6.1 Metal Layers – Area Tradeoff

One interesting use of GILES is to have it measure the trade-off between number of metal layers and the achieved final area. Figure 10 is a plot of the final placed and routed area achieved vs. number of inter-cell metal routing layers, for the first six

architectures of Table 2. It shows that area increases significantly once a lower-limit on metal layers is achieved.

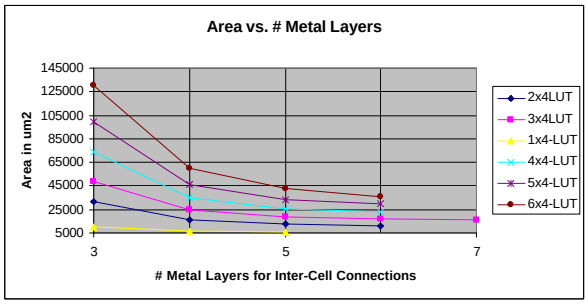


Figure 10 - Area vs. # Metal Layers

6.2 Comparison to Commercial Devices

In order to get a measure of the quality of our results, we have attempted to compare the automated layout of GILES with high-quality hand layout done for commercial devices. We created an input architecture file that resembles a commercial architecture and device, and compared the logic and routing tile areas between these proxy and real devices. Since our architecture file format isn't flexible enough to describe the exact architecture of the commercial devices, this is a *very* approximate comparison. In particular we are unable to reproduce the detailed logic elements (such as special internal muxing, lookup table RAM, extra muxing, etc.) of commercial devices. We chose to compare to the Xilinx Virtex-E device (built in 0.18µm CMOS process from UMC) and the Altera Apex 20K400E device (built in a 0.18µm CMOS process from TSMC). Table 3 gives the characteristics of the proxy architectures for these devices, as specified in the input architecture file to our automated layout system.[1]

Device	# LUTs per Cluster	Rout Type 1 Length	Rout Type 1 #	Rout Type 2 Length	Rout Type 2 #	Rout Type 3 Length	Rout Type 3 #	Fc In	Fc Out	Fs
Xilinx Virtex E	4	1	24	6	72	12	12	0.11	0.09	3
Altera Apex 20K400E	10	20	140	45	90	n/a	n/a	0.14	0.06	3

Table 3 - Characteristics of Proxy Architectures

Table 4 gives the measured dimensions and area of the tiles for each of the real commercial devices [22], the area achieved for the proxy architectures by GILES, the percentage difference in areas and the number of metal layers required to achieve the routing. The Virtex E proxy layout is only 47% larger than the commercial device. The proxy Apex20K400E is almost twice as large as the original. We believe these results represent an exciting first step in what could become a viable and competitive method of layout for FPGA tiles. Figure 11 is a picture of a fully placed and routed proxy Virtex-E architecture tile generated by GILES.

Device	Tile Actual X (um)	Tile Actual Y (um)	Measured Area (um^2)	Proxy Area (um^2)	% Diff	# Metal Layers
Virtex E	238	149	35462	52268	47%	8
Apex 20K400E	156	404	63161	124161	97%	8

Table 4 - Area of Commercial Tiles and Proxy Automated Layout Architectures

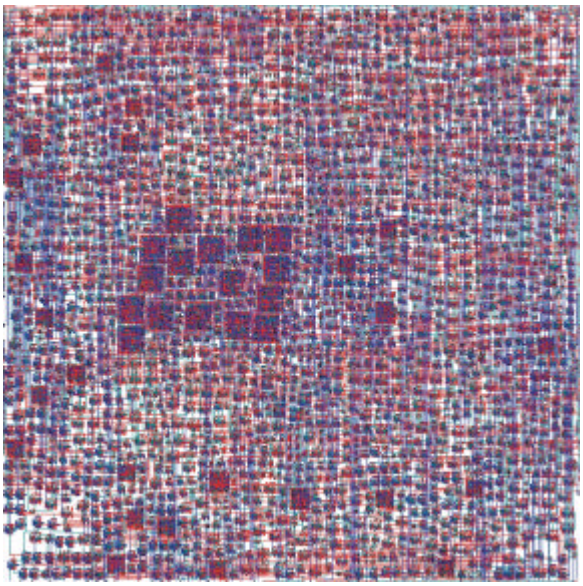


Figure 11 – Routed Proxy Virtex-E With Manually Laid Out Cells

7. Conclusions and Future Work

We have presented a CAD system, called GILES, that performs the automated layout of FPGA tiles from an architectural specification. We have shown that GILES is capable of producing a wide spectrum of different architectures on a modern IC process. A comparison with manually designed commercial devices showed that it can produce reasonably dense layouts. We intend to take this work forward in several directions: we will augment the tools to allow the production of actual programming bitstreams, and will automate the sizing of all gates, buffers and pass transistors as part of the architectural generation process, making the sizes architecture-dependent. The layout tool also needs to be timing-driven so as to optimize the wires associated with critical parts of the cells.

[1] Note that the VPR architecture exploration system is constrained to model architectures with the same number of tracks in the horizontal and vertical direction. In order to model the Apex 20K400E, which has a different number of tracks in each direction, we chose a quantity that would sum to the same total number of tracks in a tile. For the Virtex-E proxy architecture, all of the length six wires have connections at every channel intersection, whereas the real Virtex-E only makes connections in the middle. In addition, we can only model bi-directional segments and the Virtex-E employs some number of uni-directional wires. In both cases, these effects will make our area larger than the real Virtex-E device.

It would also be beneficial to tie the router back into placement to have the placement adjusted based on actual congestion. There are several optimizations we can make that should dramatically improve the final area, including using more aggressive metal spacing (with a smarter router) and using some of the first 2 layers of metal for inter-cell routing. Finally, we intend to fabricate a chip that is automatically generated in this way.

8. Acknowledgements

The authors would like to thank Vaughn Betz for providing the research and code base upon which this work is based, and much-appreciated guidance along the way. Elias Ahmed provided the 0.18 μm VPR 1-10 LUT architecture files and appropriate transistor sizing. William Chow and Chris Sun provided graphics packages, and Joshua Slavkin provided the manual layout infrastructure. Tomasz Czajkowski reverse-engineered important details of the Virtex-E architecture. David Galloway provided very helpful comments on this paper. Altera Corporation provided data on commercial chip die areas and a few architectural details of the APEX E. This work was funded by an NSERC research grant.

9. References

[1] K. Padalia, “Automated Transistor-Level Design and Layout Placement of FPGA Logic and Routing from an Architectural Specification”, Undergraduate Thesis, University of Toronto, 2001 http://www.eecg.toronto.edu/~jayar/pubs/ATL/ketan_padalia_2001_thesis.pdf

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

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

[4] V. Betz and J. Rose, “Cluster-Based Logic Blocks for FPGAs: Area-Efficiency vs. Input Sharing and Size,” in IEEE CICC 1997, Santa Clara, CA, pp. 551-554.

[5] R. Fung, “Optimization Of Transistor-Level Floorplans For Field-Programmable Gate Arrays”, Undergraduate Thesis, University of Toronto, 2001 http://www.eecg.toronto.edu/~jayar/pubs/ATL/ryan_fung_2002_thesis.pdf

[6] E. Ahmed and J. Rose, “The Effect of LUT and Cluster Size on Deep-Submicron FPGA Performance and Density,” in FPGA 2000, ACM Symp. FPGAs, February 2000, pp. 3-12.

[7] P. Hallschmid, S.J.E. Wilton, ``Detailed Routing Architectures for Embedded Programmable Logic IP Cores'', in the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, CA, Feb. 2001, pp. 69-74.

[8] A. Stauffer, R. Nair, “Optimal CMOS Cell TransistorPlacement: A Relaxation Approach”, Proc. ICCAD, 1988, pp. 364-367.

[9] S. Kirkpatrick, C. Gelatt and M. Vecchi, “Optimization by Simulated Annealing,” Science, May 13, 1983, pp. 671 – 680.

[10] T. Serdar and C. Sechen, “AKORD: Transistor Level and Mixed Transistor/Gate Level Placement Tool for Digital Data Paths”, International Conference on CAD, Nov. 1999, pp. 91-97.

[11] http://www.numeritech.com/ntproducts/

[12] K. Azegami, S. Kashiwakura, K. Yamashita, “Flexible FPGA Architecture Realized of General Purpose Sea of Gates, FPGA ’96.

[13] S. Phillips, S. Hauck, “Automatic Layout of Domain-Specific Reconfigurable Subsystems for System-on-a-Chip,” FPGA 2002.

[14] http://www.juniper.net/micromagic/max.html

[15] C. Sechen and A. Sangiovanni-Vincentelli, “The TimberWolf Placement and Routing Package,” JSSC, April 1985, pp. 510 – 522.

[16] Cheng, C. “RISA: Accurate and Efficient Placement and Routing Modeling”, ICCAD, 1994. pp. 690-695

[17] V. Betz, “VPR and T-Vpack: Versatile Packing, Placement and Routing for FPGAs”, http://www.eecg.toronto.edu/~vaughn/vpr/vpr.html

[18] MOSIS Corporation, “MOSIS Scalable CMOS (SCMOS) Design Rules,” http://www.mosis.org/Technical/Designrules/scmos/scmos-main.html

[19] 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.

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

[21] M. Bourgeault, J. Slavkin, and Y. Sun, “Automatic Transistor-Level Design and Layout of FPGAs”, Undergraduate Design Project, University of Toronto, 2002. http://www.eecg.toronto.edu/~jayar/pubs/ATL/Bourgeault_Slavkin_Sun_2002_Project.pdf

[22] Giles Powell and Srinvas Reddy, Altera Corporation, Private Communication.