Design, Layout and Verification of an FPGA using Automated Tools

Ian Kuon, Aaron Egier and Jonathan Rose
Department of Electrical and Computer Engineering,
University of Toronto
Toronto, Ontario, Canada M5S 3G4
{ikuon,aegier,jayar}@eecg.utoronto.ca

ABSTRACT

Creating a new FPGA is a challenging undertaking because of the significant effort that must be spent on circuit design, layout and verification. It currently takes approximately 50 to 200 person years from architecture definition to tape-out for a new FPGA family. Such a lengthy development time is necessary because the process is primarily done manually. Simplifying and shortening the design process would be advantageous since it could reduce the time to market for new FPGAs while also enhancing architecture explorations. One way to accomplish this is through automation and, in this paper, we describe our efforts to automate the entire process by making use of a previously developed set of tools that assist in the creation of the repeatable FPGA tile [25]. Our aim is to demonstrate the feasibility of a CAD flow that uses an input FPGA architecture description to generate a layout that can be sent for fabrication. We prove the feasibility of this proposition by actually designing and fabricating a complete FPGA. Initial functional testing of the FPGA appears promising but is inconclusive at this time. Through this architecture to layout process, we investigate the issues that are faced in the architecture selection, circuit design, layout and verification of such an automatically produced FPGA. We found that there are significant savings in design time. As well, we demonstrate that we can produce a layout using automated tools that is only 36% larger than a commercial FPGA device layout. Given the significant time savings and the relatively minor area penalty, we feel that this work demonstrates that automated layout of FPGAs is practical and advantageous.

Categories and Subject Descriptors

B.7.2 [**Integrated Circuits**]: Design Aids – *layout, placement and routing, verification* B.7.1 [**Integrated Circuits**]: Types and Design Styles – *gate arrays*

General Terms

Design, Verification

Keywords

FPGA, PLD, programmable logic, automatic layout.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
FPGA'05, February 20–22, 2005, Monterey, CA, USA.
Copyright 2005 ACM 1-59593-029-9/05/0002 $5.00.

1. INTRODUCTION

The creation of a new FPGA is a complex and daunting task because the very reasons that make FPGAs attractive to designers present associated challenges to those who produce the FPGAs - every pre-fabricated structure must work correctly in every conceivable use, and must be implemented as efficiently as possible so that the cost of programmability does not outweigh its benefits. The creation of an FPGA requires an increasingly complex design process, in which the architecture of the FPGA must be determined, appropriate electrical circuits for implementing this architecture must be developed, the layout of that circuit must be performed and the final design must be verified to be correct. This design process takes approximately 50 to 200 person years [25].

The reason this process is extremely time-consuming is that much of it is currently performed manually. It is essential that the final layout allow the implementation of circuits that are as fast and as low power as possible, while occupying the smallest area to reduce cost. For these reasons, FPGA manufacturers have relied on manual design since it is believed to deliver higher quality layouts than are generated by automated layout tools.

On the other hand, there are many benefits that could come from automation of this design process. First, automation would reduce the time to market for new FPGAs, which currently take at least two years from conception to first-silicon availability, of which the design portion takes at least one third. Second, an automated design flow could also lower the entry barrier for new FPGA manufacturers – instead of requiring a team of hundreds to create the design in a reasonable amount of time, a handful of engineers could produce a hopefully similar quality design in even less time using automated design tools. Finally, if the layout process was quick and automated, FPGA architects would have access to more detailed information about the area and speed impact of their architectural decisions. This improved accuracy could lead to the development of more efficient architectures.

Previous researchers [25][27][16] have proposed several ways of automating the design process for various kinds and circumstances of programmable logic, which we review in Section 2. However, none of this past work fully considered the implications of using automated layout tools to fabricate a complete FPGA. In the present work, we employ and enhance one of these previous approaches to architect, design, floorplan, lay out, verify, and fabricate a complete FPGA. Our goal is to demonstrate that automated design tools can be used successfully to create an FPGA, and that it can be done with significantly reduced manual labor. We also compare the area of automatically

and manually generated layouts to determine if the savings in engineering costs are being offset by increased production costs.

This paper is organized as follows: Section 2 describes previous work on automated design methodologies for FPGA creation. In Section 3, we describe the development of the architecture to be implemented. In Section 4, we outline the automated process of taking that FPGA architecture and producing a layout that can be taped out for fabrication. Before actually taping out the design, extensive verification is necessary, which is described in Section 5. Section 6 reports on the functionality, required design time and area of the FPGA and Section 7 concludes.

2. Related Work

There have been a number of prior works concerning the automated creation of FPGAs.

Kafafi *et al.* propose a new architecture and an implementation using standard cells to construct an FPGA from a VHDL input description [16]. Their context is the creation of FPGAs embedded within a non-programmable ASIC. The layout for this logic, called a *synthesizable embedded programmable logic core*, is easily created using commercial synthesis, placement and routing tools. The authors adopted this approach to allow for easy integration in system-on-a-chip applications and they target a very specific architecture that is suitable only for small embedded cores. Nevertheless, this approach could also simplify the design of standalone FPGAs. The disadvantage of this approach is that, as feared, it incurs a significant penalty in terms of area. In [16], the authors estimate that the automatically generated layout is 6.4 times larger than an equivalent manually created layout. Wilton and Wu report that a core created with this methodology did function successfully [35].

Phillips and Hauck also employed standard cells to implement programmable logic in [27] and [26]. They focused on the creation of *domain-specific reconfigurable systems* in which the amount of configurability can be reduced for the particular application domain. The authors do offer a comparison between unreduced automated and full-custom layouts. Their finding was that the automated approach yielded a layout that was 42% larger and 64% slower than a manual design. By reducing functionality to that required for a specific domain, the authors successfully demonstrate that smaller and faster layouts can be created automatically. This flexibility to create varying designs highlights one of the benefits of automated design that we hope to achieve as well. Phillips and Hauck do not report any fabricated chips resulting from this work.

The above works take as their starting point a fairly FPGA specific architecture. The goal in the present work is to automate a fairly general class of FPGA architectures. Also, in both cases, automation clearly simplified the design process but the result was detrimental in terms of area and speed. These results are comparable to past research that measured the performance gap between standard cell and custom designs. In [15], Dally and Chang find that, depending on the level of automation, automated layouts are between 64% and 1350% larger than custom layout areas. The speed of the implementation also suffers although with only a little manual effort automated tools came within 11% of the speed performance of a custom layout. Based on these results, we consider area the most important factor facing automated design flows.

An alternative to using standard cells and supporting tools is to use tools created specifically for FPGA design and layout. This approach was used in [25]. These tools take an FPGA architectural specification as the primary input and take advantage of FPGA-specific optimizations during the layout process. As a result, they produce smaller layouts than would otherwise be possible. Our work uses these tools in the design of our FPGA. However, before describing the basic functionality of these tools, we will review the overall FPGA layout approach that will be used in this work.

2.1 Tile-based FPGA Layout

We will focus exclusively on island-style FPGAs, as illustrated in Figure 1. A common (but not universal) approach ([9][22][33]) to laying out this style of FPGA is to take advantage of regularity of the array and build a tileable structure that can be replicated. This is done by grouping the basic logic block and the adjacent routing resources into a tile as shown in Figure 2. Such an approach does place some restrictions on the architectures that can be created. A detailed discussion of these can be found in [24].

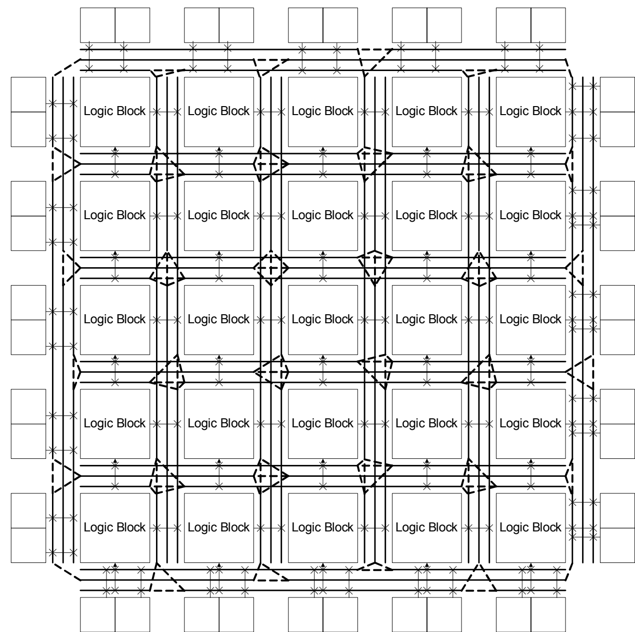


Figure 1 - Island-style FPGA

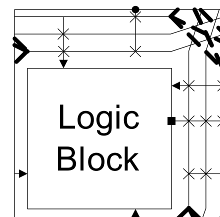


Figure 2 - Individual Tile

The tile layouts are connected simply by abutment. While this approach provides less opportunity for chip-wide optimization, the layout is much more manageable. We use this tile-based approach in the present work, as did [25].

This tile-based approach also has the advantage that it can be readily adapted to handle heterogeneous FPGA architectures. In such architectures, the logic block could simply be replaced by other features such as multipliers or memory and the layout of these blocks would be constrained to match the connections of a normal tile containing a logic block. If necessary, these heteregenous blocks could occupy the space of multiple logic block tiles. In this work, we focus on uniform FPGAs consisting exclusively of logic blocks but, since our tools are already capable of handling constrained layouts, extending this work to handle heterogeneity would be straightforward.

2.2 The GILES Layout Tools

While laying out a single tile is easier than laying out the entire array, it is still a daunting task. Tiles typically contain on the order of 10,000 transistors [25] and, given that the same layout will be used repeatedly, obtaining a compact layout area that can operate at a high speed is essential. The GILES tools described in [25], [6], [14], and [24] are designed to automate the layout process of this single tile while maintaining reasonable speed and area. Our work uses these tools to assist in the process of making a complete FPGA. The inputs to these tools are an architecture description and a description of the available cells. These are used to produce a single tile layout that can be used to create part of an FPGA. The steps in this process are shown in Figure 3.

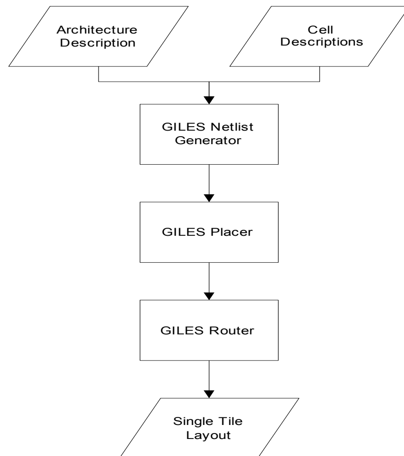


Figure 3 - GILES Tools Layout Flow

The architecture description is specified using the VPR Architecture Description Language [2][3]. The VPR Architecture Generator [2][3] uses that architecture description to produce an internal data structure defining the FPGA that will be created. Based on that structure, the GILES Netlist Generator produces a cell-level netlist describing the circuitry needed to implement that architecture. It is important to remember that this netlist does not describe the entire FPGA and, instead, it only describes a single tile of the FPGA.

To produce the physical layout of the tile, the GILES Placer and Router are used. The GILES Placer takes the cell-level netlist as an input. A simulated annealing-based algorithm [17] is then used to perform simultaneous placement *and* compaction. This custom placement approach also allows the use of FPGA-specific optimizations such as allowing the logical equivalence of configuration SRAM outputs to be leveraged [25].

Once placement is complete, the GILES router takes the placed FPGA tile and outputs the mask-level layout for the tile. Previously, in [25], routing was performed using a custom detailed router based on a negotiated congestion algorithm [11] [5]. This router is useful for experimental work since it always routes a design successfully. When the router fails to find a valid routing initially, rows and/or columns of space are inserted in the most congested regions of the design and the routing algorithm is re-applied. We make use of this router for the results presented in Section 6. However, this router is not as flexible as most commercial routers. For example it is unable to handle partial routing blockages on a layer (it needs the entire layer to be free). For this reason, the GILES tools were augmented to also function using the Cadence Chip Assembly router [7][12]. This is the primary approach we use in this work.

3. ARCHITECTURAL DECISIONS

The first step in creating any FPGA is selecting its architecture. In this section we describe the architectural choices made, and the method for making the choices.

One of the first decisions that must be made is to select the silicon process that will be used for our design. We decided to use a 0.18 μm process from TSMC with 6 metal layers and 1 polysilicon layer. This mature technology was selected to allow us to focus on the functionality of our prototype design instead of having to worry about the increasingly complex design rules and the lower yields of the latest processes. However, using a less aggressive process technology does not make our results less relevant since the automated design tools can be adapted to any process.

The next consideration is to determine the tool flow that will be used for implementing circuits on the final FPGA. Since the GILES Layout tools are based on VPR [4], our CAD flow for using the FPGA will also be VPR-based. The input circuits to this CAD flow will be BLIF descriptions of circuits that were synthesized and mapped from the original BLIF descriptions using SIS with Flowmap [10][28] to the 4-LUTs we use in this work. T-VPACK [21] will then be used for clustering and VPR for placement and routing on the FPGA. This choice constrained the range of architectures that we could consider to basic logic element-based FPGAs with multiple-driver based routing. It was beyond the scope of this work to also augment VPR to handle more recent architectural features such as carry chains and unidirectional/direct-drive routing [20][38].

There has been a great deal of research aimed at finding efficient architectures for VPR-style FPGAs [1][5]. For our architecture, the logic block was selected to be a cluster containing three 4-input LUTs; this is considered to be reasonably area-efficient and fast in [1], but creates a small enough tile that a reasonable number can be replicated in the silicon area available to us. The number of cluster inputs and the output connection block flexibility were set based on the conclusions from [5]. These conclusions called for the number of cluster inputs to be equal to 2N+2 where N is the cluster size. Therefore, for our cluster size of three, there should be 8 inputs. The authors also recommended that the flexibility of the connection block is equal to 1/N which gives a flexibility of 0.333 for our architecture.

Next, we decided to avoid non-buffered pass-transistor routing because of the electrical complexity it adds to the design. Instead,

a fully buffered bi-directional routing network will be used. For this case, the results from [5] suggested that routing tracks with only segments of length 4 would be an appropriate routing architecture.

The next key question was to determine the number of tracks in each routing channel, as this has a dramatic effect on the tile area. We employed an experimental approach to determine a quantity that would render the resulting FPGA routable. In these experiments, 117 MCNC benchmark circuits [23], which were previously mapped to 4-LUTs, were run through the complete T-VPACK/VPR flow and the number of benchmark circuits that routed for a range of possible track counts was measured. The circuits selected were those we approximated would fit in the silicon area allocated to our project since silicon area limits our LUT and input/output count.

The results of this experimentation to determine a suitable track count are shown in Figure 4. Based on these results, we set the number of tracks per channel to be 20 since it is the minimum track count that allowed all the benchmark circuits to successfully route.

The input connection block flexibility was also determined experimentally. Since it has a less pronounced effect on the tile area, we selected the minimum flexibility, 0.6, that offered consistent routability for all the benchmark circuits. Some lower flexibility values were able to route the circuits; however, it was necessary to repeatedly run VPR with varying input seeds before all the designs could be routed.

The complete final parameters that were used for the FPGA are summarized in Table 1.

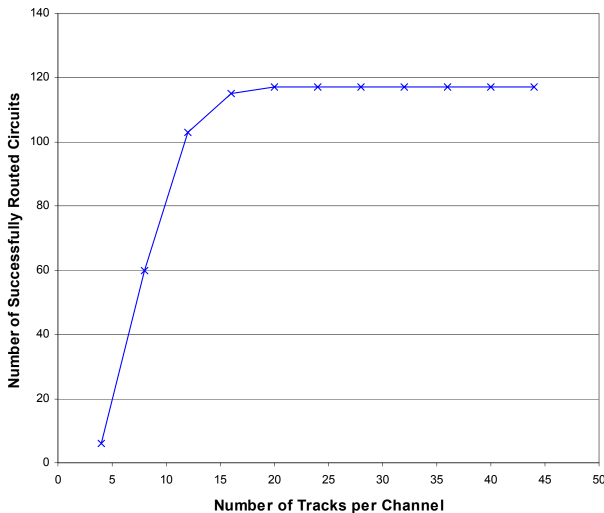


Figure 4 - Number of Routed Circuits over Varying Track Count

Table 1 - Architecture Parameters

Parameter	Value
LUT Size, k	4
Cluster Size, N	3
Number of Cluster Inputs, I	8
Number of Tracks Per Channel	20
Track Length	4
% Buffered Tracks	100
$F_{c,\ input}$	0.600
$F_{c,\ output}$	0.333
$F_{c,\ pad}$	0.600
Array Size, n_x x n_y	8 x 8
Pads per row/column	2
Total Number of LUTs	192
Total Number of I/O's	64

As was mentioned previously, the tile-based layout methodology restricts the architecture that can be selected. Among the tileable architectures, there are also limitations in how these architectures are created by the VPR Architecture Generator. The current version of VPR produces architectures that are in fact not tileable. In particular, the connection from the routing tracks to the logic cluster is not created such that every cluster's connections match the physical implementation created when a single tile is replicated. This is described in detail in [18]. To address this deficiency in dealing with tile-based layouts, VPR was updated appropriately.

4. ARCHITECTURE TO LAYOUT – AUTOMATICALLY

With the architecture defined, the process of creating a complete FPGA automatically can proceed. The goal was to go from architecture description to layout with as little manual intervention as possible.

4.1 Netlist Generation

The first step in the architecture to layout process takes the architecture description as input and produces a cell-level netlist of the tile as output. A portion of an example input architecture description is shown in Figure 5. The processing of this description is done using the GILES Netlist Generator described previously in Section 2.

```
subblocks_per_clb 3
subblock_lut_size 4

#parameters needed only for detailed routing.
switch_block_type subset
Fc_type fractional
Fc_output 0.333333333333333
Fc_input 0.6
Fc_pad 0.6
# Uniform channels. Each pin appears on only one side.
io_rat 2        #2 pads per row or column

chan_width_io 1
chan_width_x uniform 1
chan_width_y uniform 1

...
```

Figure 5 - Architecture Description Excerpt

At this point in the process, it was necessary to decide how the transistors in the design will be grouped into cells. The initial GILES tools described in [25] selected the grouping into cells without any consideration of the area impact of this choice. The work in [12] explored a range of grouping possibilities and this design makes use of the improved cell groupings discovered in that work. A key conclusion from that work is that SRAM configuration cells are most efficiently grouped in a 4x4 configuration of 16 bits. This conclusion depends on the ability of our layout tools to leverage the functional equivalence of the configuration bits during placement and compaction. This conclusion may be relevant to manual layout regimes as well.

4.2 Cell Layout

Once it was known which cells were required for the design, the cells had to be created. This is the one step in the design process that must be done manually. However, it is possible that tools such as Cadabra from Synopsys [30] could be used for this step in the future. The fifteen cells required for the design are listed in Table 2. All the cells are relatively small in size ranging from an inverter with two transistors to a group of SRAM bits requiring eighty transistors. The 4 word line by 4 bit line array of SRAM bits is constructed by replicating the layout of a single 5-transistor SRAM bit and, therefore, is not as complex as might be implied by the eighty transistors in the cell. Clearly, the effort required to layout these cells is significantly less than would be needed for a complete FPGA tile containing thousands of transistors.

Table 2 - Manual Cell Layouts

Cell	# of Transistors	Size (μm^2)
1x Inverter with pullup	3	15.25
2x Inverter	2	13.07
4x Inverter	2	15.68
4x Buffer	4	18.30
4x Buffered Switch	5	18.30
PMOS Pullup	3	13.07
4-LUT	30	86.25
2-input MUX	2	8.71
11-input MUX	20	47.92
12-input MUX	22	52.27
20-input MUX	38	109.77
Flip-flop	15	47.04
Flip-flop with enable	19	54.89
AND gate	6	17.42
4x4 SRAM	80	209.09

4.3 Periphery Tiles

The process above focused on the creation of a single tile containing a logic block and its adjacent routing. As described earlier, a layout of the tile can be replicated to form a large array of logic blocks connected by routing. This, however, is only part of the typical island-style structure shown in Figure 6.

In particular, there is extra routing required along the left and bottom edges of the tile array to provide connectivity to those I/O cells and ease routing congestion. Additional tiles are needed to provide that routing. In total, eight different tiles are required. This includes the main tile described above, four edge tiles and three corner tiles.

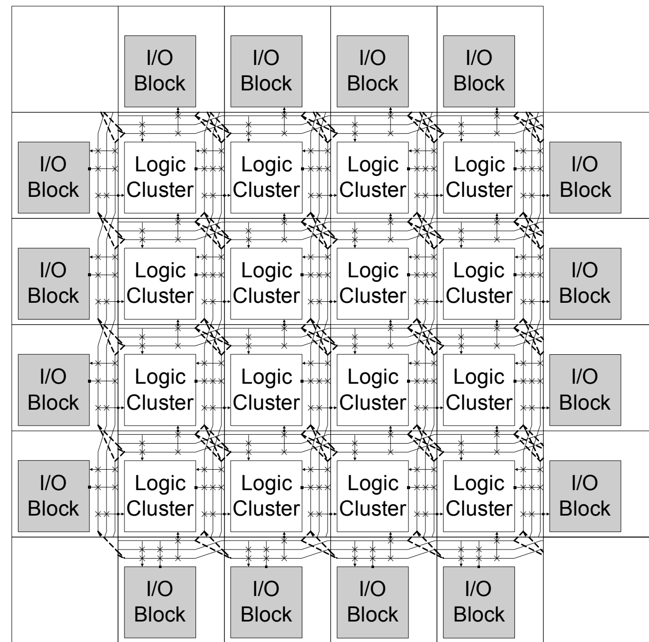


Figure 6 - Tiled Island-style FPGA

With this tile-based approach, the goal is to connect the tiles by abutment. To enable such connections, it is necessary to constrain the placement of the inter-tile connection ports. Normally, those ports are free to move which allows for improved routing. Constraining the placement of these ports restricts the amount of optimization available to the placer and will likely lead to a less area-efficient design. To minimize the impact of this effect on the total area of the design, the tiles must be created in a specific order, allowing the most frequently used tile (the main tile) the most freedom during its placement by laying it out first. The next most frequently used tiles are the edge tiles. Each of these tiles abuts the main tile on one side and it is those connections to the main tile that must be constrained during placement of the edge tiles. To accommodate this, the GILES placer, which normally performs both placement and compaction, was modified to handle the constrained placement of the inter-tile connection ports and to compact the tile only in the unconstrained dimension. Finally, the corner tiles each connect to two of the periphery tiles and are therefore constrained on both edges.

4.4 Tile Layouts

Each of the eight tiles required for the design was laid out using the GILES Placer [25] and the Cadence Chip Assembly [7] router. Without automation this process can take many person-years but with the automated placer and router the task is completed in less than half a day.

The power grid was added to each tile placement before routing. It consists of alternating power (VDD) and ground (VSS) vertical and horizontal stripes on the top two metal layers. The Cadence Chip Assembly router automatically connects the power grid to all the cells in the tile. However, left on its own, the router prefers to make connections between the cells on the lower metal layers and brings up only a few thin connections to the power grid. This will restrict the current and slow down the FPGA. It could also cause reliability problems due to electromigration. One solution is to tell the router to create thicker power connections but this

would increase congestion. Instead, we force the router to make more connections up to the power grid by dividing the power grid into regions with each region having its own logical VDD and VSS nets. Physically, these logical nets are all connected electrically. The cells located in each region connect to the VDD and VSS nets for that region only. The router connects the VDD and VSS nets in each region and brings up at least one connection to the power grid per region.

To calculate how many power regions were needed, we determined the maximum current draw of the main tile is 63.6 mA if all the cells switch at the same time. Assuming only a quarter of the cells switch simultaneously, the current draw is 15.9 mA. At least 57 power connections are needed to supply this current. To be cautious, we created 80 power regions. These calculations are sufficient for the purposes of our prototype chip but, in the future, we would like to simulate the power grid to verify how many regions are needed. The power grid and its regions are generated automatically using Cadence's scripting language, SKILL [8], so it is easy to change the number of power regions in the future.

Once a tile layout was created, it was checked to ensure it conforms to the process design rules. Typically, DRC errors can be introduced by both the placer and the router. In our case, the placer treats all cells as black boxes. Therefore, to avoid DRC problems, the cell boundaries were sized such that all placements without overlapping cell boundaries are DRC clean. This caused space between cells that would not exist in manually laid out tiles. However, we eliminated some space by using larger cells [12] and future work may be able to reduce it further without violating design rules.

The Cadence Chip Assembly router was configured for our process's design rules. However, the final routing output by the router did not satisfy all these rules when the design was congested. In our trials, typically a small number (under 20) of violations would be found. These errors were fixed manually.

4.5 Configuration SRAM Programmer

With the tiles now laid out, the design of the logical functionality seen by a user of the FPGA is complete. However, before actually using the FPGA, each SRAM bit in the design must be programmed to the state required for the circuit being implemented on the device. This task is performed by a dedicated configuration SRAM programmer.

As the tiles were being created, the GILES tools connected the configuration SRAM bits to the necessary programmable elements and connected all the bits in a tile together to form a memory array of word and bit lines. There are far too many lines to connect off chip and for this reason, dedicated on-chip circuitry is needed to control these connections to the SRAM bits. Modern FPGAs have many advanced programming features such as the partial reconfiguration capabilities described in [37] and [38] but these features are not needed for the test chip we are creating. Instead, we adopted the simple strategy illustrated in Figure 7. Here the word and bit lines from all the tiles are connected together to form one large array. Two shift registers control the word lines and the bit lines. These shift registers are embedded in the periphery tiles.

The programmer contains the control logic for these shift registers. This controller was written in Verilog. The design was synthesized using Synopsys Design Compiler and it was placed and routed using Cadence Silicon Ensemble PKS. The standard cells in the design were from the Diplomat-18 Standard Cell library from Virtual Silicon Technology [34].

Programming of the device proceeds as follows. First, the bitstream is sent to the programmer and serially loaded into the bit line shift register. The programmer configures the word line shift register to assert one word line when the contents of the bit shift register are valid. This process repeats until all the word lines have been programmed.

4.6 Bitstream Generation

The programmer handles the task of configuring the SRAM bits in the design based on the bitstream supplied to it. However, we need a way to generate this bitstream. Doing so requires a complete FPGA CAD flow such as Altera's Quartus II or Xilinx's ISE that can take a description of the circuit to be implemented and output a programming file for the device. Our solution to this challenge is based on the VPR toolset [5]; the process we use is shown in Figure 8.

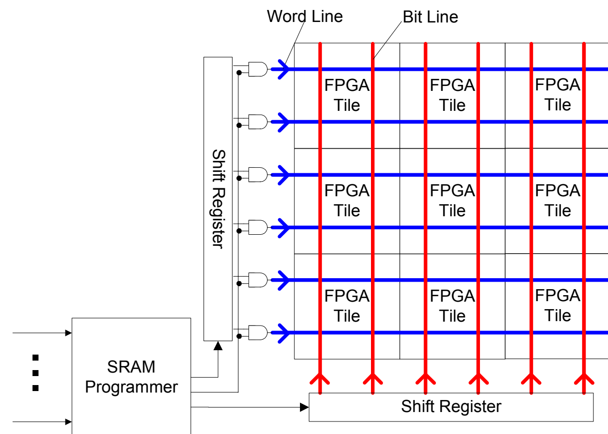


Figure 7 - Configuration SRAM Programmer

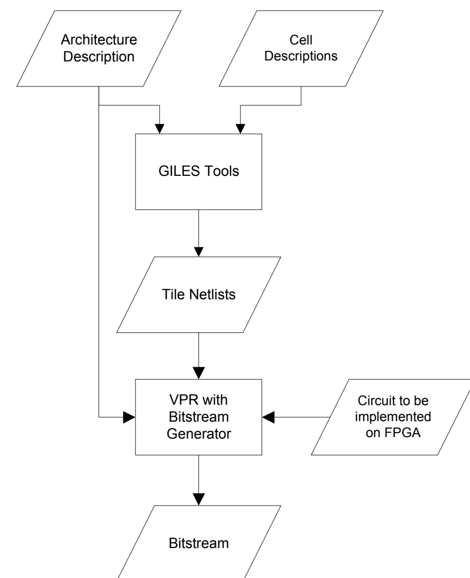


Figure 8 - Bitstream Generation CAD Flow

It is interesting to consider an issue that arose due to one of the features of the GILES tools: its ability to take advantage of logical equivalencies in the FPGA. The netlist of the tile is first defined by the FPGA tile netlist generator. However, the placer alters this netlist to improve the quality of the placement. One such alteration involves the connections to configuration SRAM bits, which are logically equivalent under bitstream re-ordering. Similarly, all the inputs to a multiplexer are logically equivalent and it is only important that the available connections remain unchanged. Making such changes during placement is unusual since with standard cell tools the only changes that would usually be made during placement and routing are buffer insertions and logically equivalent swaps of cell inputs whereas the GILES placer will perform logically equivalent swaps of the cells themselves. The changes that our placer makes impact the bitstream that is to be generated and, therefore, a bitstream cannot be created until after the placement is complete. At that point the bitstream generator must reconstruct the logical structure of the FPGA. This is not a trivial task since resources like routing tracks are all very similar. To ease this process the GILES placer was updated to maintain information about the logical FPGA structure. With that information rebuilding the logical structure from the tile netlists is significantly easier.

Once the logical structure information is reconstructed, generating the bitstream is straightforward using a modified version of VPR. Placement and routing are first done normally using VPR. The output from the router is a set of routing resource graph edges that were used by the circuit. The bitstream generator configures the bitstream such that the programmable elements corresponding to those edges are turned on. Note also, that all the basic logic elements in the logic cluster are equivalent and that the router determines which specific logic element is used. Therefore, the bitstream generator only configures the logic elements after the routing information is processed. All the other programmable elements in the design are left in a default safe state for the bitstream. In this state, all unused input and output pads are configured as inputs and unused routing track drivers are disabled to avoid contention.

4.7 Arrays, Clock and Power-Up Signals

The work described in the previous sections provided all the necessary pieces required for the FPGA that was created. The floorplan shown in Figure 11 illustrates how the tile layouts and the programmer connect together. The input/output pads for the design are also from a Virtual Silicon Technology library [34]. These pad cells were manually laid out to form the pad frame for our FPGA. The array of the eight different FPGA tiles was constructed automatically by abutting the tiles to form the structure shown in the figure. This was done using Cadence's scripting language, SKILL [8]. As described earlier, the layout of the programmer was generated automatically using commercial place and route tools. The connections between the FPGA array, the programmer and the pad frame were done by the Cadence Chip Assembly Router with two exceptions: the clock and a global programming signal. We discuss both of these signals now.

The tile-based approach works well for most of the features in the FPGA. However, clocks must be treated differently if the design is to function quickly and reliably. With clock signals, it is important to create a low-skew network. A frequently used structure for ensuring this is known as an H-tree. With this structure, the distance (and therefore delay) to all end points (registers) is the same but implementing this structure as shown in Figure 9 requires that the clock enter each tile from alternating sides of the tile. Instead of creating two types of tile we decided to create a single tile with a small horizontal channel reserved for the clock routing. After the tile layouts are arrayed, the clock routing is created by a SKILL script and the appropriate side of the tile is used.

One potentially problematic area with FPGAs is that their programmability introduces the possibility of contention. In our architecture, all the routing tracks can be driven by many different drivers. A valid programming bitstream ensures that only one of these drivers is on but when the device is powered up the programmable SRAM bits initialize to an undetermined state. As a result, it is possible to have a case as shown in Figure 10(a) where multiple drivers are driving a routing track with different values. The drivers would then require large supply currents. To avoid having to design our power grid to handle such large currents, we instead opted to prevent this contention. All drivers where contention is possible can be disabled by a global programming signal as shown in Figure 10(b). This signal is set to 0V until a valid programming bitstream is loaded into the design. For normal operation of the design, the signal is then set to VDD. This signal must be distributed throughout the tile array. We automatically route the signal again using a SKILL script. In this routing we take advantage of the routing channel that was left for the clock. Since the clock only enters on one side of the tile, the other side remains available and this global programming protection signal connects through this channel.

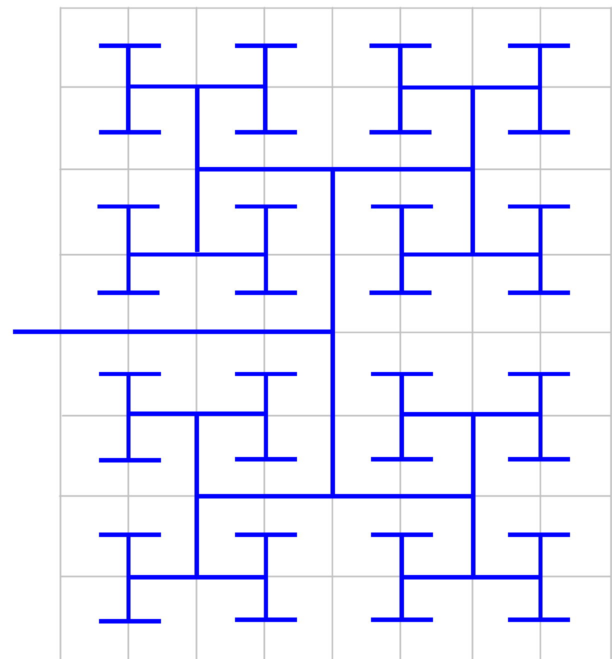
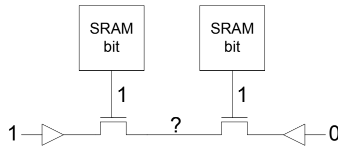
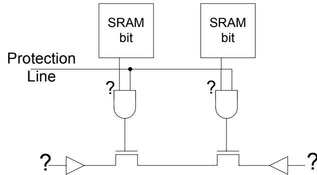


Figure 9 - Clock H-Tree



(a) Possible Contention prior to Configuration



(b) Approach used to prevent contention

Figure 10 - Power On Circuitry

One final concern after routing is antenna rules [29][13]. These rules ensure that during manufacturing the routing in the design does not accumulate a charge large enough to damage the gates of transistors connected to that routing. The charge accumulates during processing steps involving ion etching and it depends on the area of the metal or polysilicon connected to a transistor gate. Any antenna violations are corrected by connecting reverse-biased diodes to the substrate since these allow the routing to be discharged. This check could not be meaningfully performed on the single tile since connections to other neighboring tiles can vary the amount of diffusion area relative to the metal/polysilicon area, which can either resolve or introduce antenna errors. Therefore, only once the array of tiles was created were the antenna rules checked. On the order of two hundred errors were observed initially but with the manual addition of only eleven diodes in the replicated tiles all the errors were corrected.

With all these steps complete, a final layout for the device was generated. A SPICE-style netlist of the complete design that others may find useful is available at:

http://www.eecg.toronto.edu/~jayar/pubs/ATL/powell.html.

However, cells taken from our standard cell library are not included to satisfy our non-disclosure agreements.

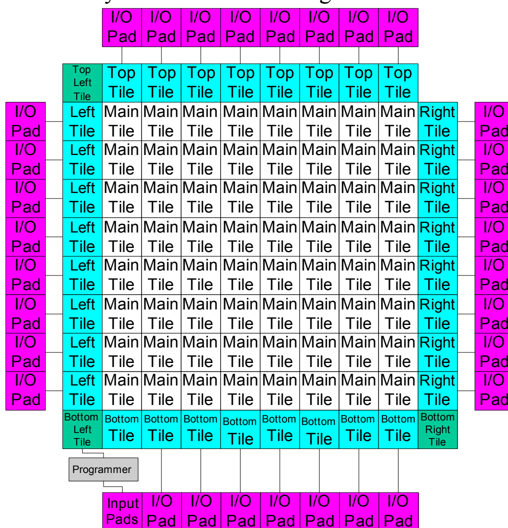


Figure 11 - Complete FPGA Floorplan

5. VERIFICATION

Prior to taping out this design, thorough verification was essential and we now describe these verification efforts. With over 300,000 transistors in the design, verification is not a trivial task. To simplify the problem, verification was done at several levels.

5.1 Logical Functionality

The first verification step taken was to confirm the correct logical behavior of the design. This is an important step because neither the tools that create the FPGA nor the CAD flow to use the FPGA had ever been tested for the correct logical functionality. The first test involved checking that all the logical connections expected by a user of the FPGA are present. This was done by comparing the routing resource graph in VPR with the tile netlists. Each edge in the graph is a programmable element that must be present in the design and, therefore, we confirmed that there was a programmable device in the netlists corresponding to each routing resource edge.

Once we fixed gross errors of missing programmable elements, we proceeded to simulate the logical behavior of the design. This is necessary to confirm that we are generating correct bitstreams for the device. For this test, we created a Verilog description of the entire design. To generate bitstreams, we started from a BLIF description of the circuit mapped to 4-LUTs. T-VPACK and the modified version of VPR were then used for clustering, placement, routing and bitstream generation. In the Cadence NC-Sim simulator, we simulated the programming of the design. Once programming was complete we applied random stimulus to the design and observed the output from the FPGA to compare it to our expectations for each test circuit. Through this testing, we were able to uncover logical problems with the generated bitstream. For example, an improper initialization of default states for unused components led to the inadvertent creation of ring oscillators throughout the design. Once we corrected these problems, we were satisfied that the bitstream generator functioned correctly and the logical behavior of the design was correct.

5.2 Electrical Functionality

Verifying the logical behavior of our FPGA is not sufficient since the design is actually being sent for fabrication in silicon. We must ensure that the design is electrically sound. First, each cell was simulated in HSPICE to ensure that it functioned. Such testing alone might be adequate for a simple digital design but the complexity of our design requires larger-scale simulations. In particular, due to the exclusive use of NMOS pass transistors trees for multiplexers and tri-state drivers, signal voltages are degraded below the full VDD level. (The use of NMOS-based multiplexers as opposed to transmission gates is standard in FPGAs to reduce area and capacitive loading.)

To confirm proper electrical functionality in the face of these varied voltages, simulation is required. Simulating a single tile is not useful since the electrical interaction with neighboring tiles must be considered. Instead, the safest approach is to simulate the entire FPGA in the same manner that it will be used in silicon. The standard tool for electrical simulations is HSPICE but with 358,374 transistors in this design the capabilities of HSPICE are far exceeded [31]. Instead, Synopsys Nanosim [32] was used. This is a SPICE-like simulator that can handle millions of

transistors while maintaining accuracy within a few percent of HSPICE [32].

The simulation process was similar to that used for verifying the logical functionality since we expect the same behavior from the transistor-level implementation of the design. The configuration bitstream was first applied and then the output from random input stimulus was compared to expectations.

It is interesting to describe the problems uncovered by these simulations. One problematic area, as expected, resulted from the degradation in voltage seen when passing through an NMOS pass transistor tree. If the voltages were left below the VDD rail then static power would become a concern since downstream PMOS transistors would not be turned off completely. To avoid such problems, PMOS pull-up transistors were used as level restorers to raise the voltage back to VDD. The sizing of those pull-ups is a delicate task since the transistors must be weak enough to allow the node to be pulled down but not be so weak that they waste area and slow down the device. The simulations uncovered cases where the pull-ups were in fact too strong and pulling down the node was not possible. As a result, the pull-up transistors were resized and the design was re-simulated. Once issues such as these were resolved the complete design was found to be functional and ready to be taped out.

6. RESULTS

Before describing the results, we first revisit the goals for this work. The primary aim was to demonstrate the viability of automated FPGA design. Clearly, the most important consideration is functionality since if automated FPGA design is to be feasible it must maintain the same level of functionality as manually generated designs. Once functionality has been assured then the feasibility of automated design rests on the cost savings it can introduce. We will look at the savings in engineering time since this translates to a reduction in non-recurring engineering expenses. Past automation efforts have failed because of a feared increase in area. Increased area translates into increased production costs and therefore we also examine the area impact of this design methodology.

6.1 Functionality

We believe the successful simulation of the entire design accomplished the goal of demonstrating that a complete FPGA can be created using automated tools. The complete layout of our design is shown in Figure 12. The core of the chip consisting of an 8 x 8 array of clusters with three 4-input LUTs occupies an area of 1041 μm by 1225 μm. The total die area is 5.634 mm^2. A photo of the fabricated die is shown in Figure 13.

The fabricated chip has been successfully powered-up. The current drawn after power-up is 17 mA which is close to our expectations and the programmer responds correctly during programming. Unfortunately, after programming, the chip draws more current than was anticipated and the test circuit does not function as expected. We believe the chip is being configured improperly but we have not yet determined whether the problem is with the chip or the test procedure. Testing is ongoing to determine the nature of the problem and we remain optimistic that it can be resolved.[1]

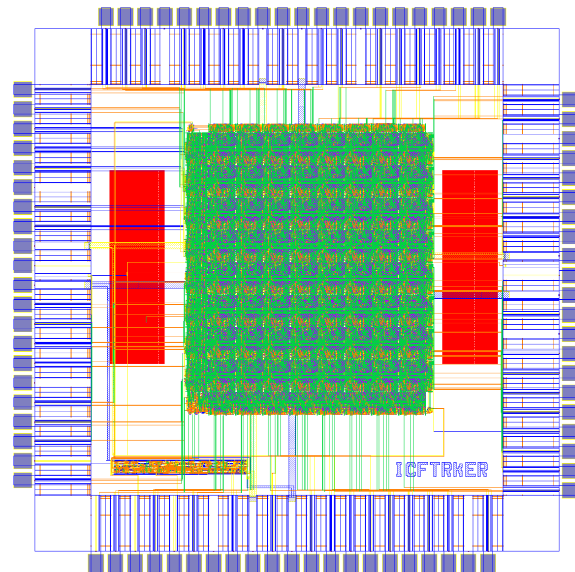


Figure 12 – Complete FPGA Layout

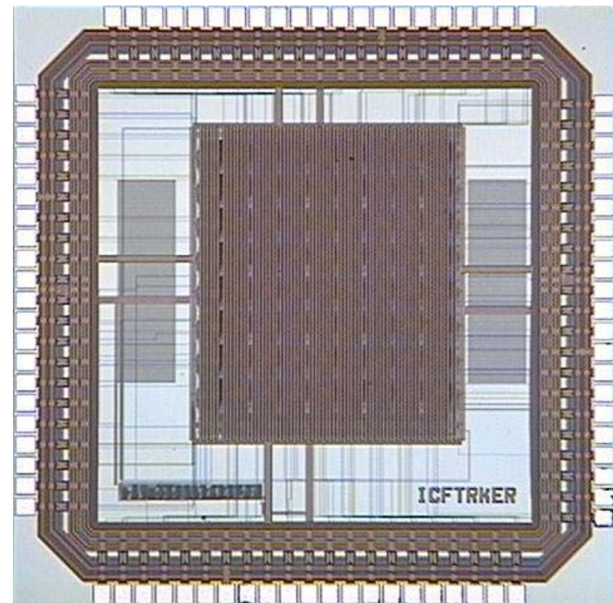


Figure 13 – Die Photo

6.2 Cost Savings

6.2.1 Design Time

The primary goal in automating the FPGA design process was to reduce the design time from the over 50 person years that are currently required to create an FPGA. To determine if we succeeded in this goal, we tracked the time spent constructing our FPGA. The time required for each stage from conception to tape-out is summarized in Table 3.

[1] Once testing is complete, the full test report will be posted at http://www.eecg.toronto.edu/~jayar/pubs/ATL/powell.html.

Table 3 – Time Required for FPGA Design

Step	Time Required (person-weeks)
Architecture Exploration	2
Circuit Design and Optimization	8
Cell Layout	6
Tile Layout	0.5
Programmer Synthesis, Placement and Routing	2
Power Planning	1
Clock Network Simulation	1
DRC/Antenna Fixing	0.5
Pad Frame Layout and Connections	1
Verification	12
Total	34

These times are in part estimates because at the same time the FPGA was being created the automated layout tools were also being enhanced. These enhancements would not be necessary for any future designs.

A few steps in the process are particularly interesting. Architecture exploration involved the investigation to determine what specific architecture would be used. For a commercial design, this step is extremely important and greater time would be allocated for this study. Similarly, the circuitry for the FPGA was not optimized to the same extent commercial designs are optimized. However, our automation efforts were not focused on either of these stages. Instead, the automated tools simplified the cell layout and the tile layout steps. For a commercial design many person years might be spent on these two steps alone while thanks to automation we spent only 6.5 person weeks completing this portion of the design. The significant time savings at this stage in the process demonstrates the success of our automated tools in reducing the design time. In total, the entire process took approximately 8 person months. Much of this work is parallelizable and two graduate students were able to complete the work over a period of 4 months.

It is important to note that modern commercial FPGAs have evolved to include more advanced logic blocks and features in addition to the basic logic block. The basic logic element now only constitutes 31% of the silicon area in a typical commercial device [19]. Our design consists solely of basic logic element clusters and simple input and output connections. Therefore, the 8 person months required for our design cannot be directly compared to the 50 to 200 person years of effort needed for a new commercial FPGA. Nevertheless, we believe the layout of the logic elements constitutes a significant portion of a typical design cycle given their importance. With our automated design tools, this time will be significantly reduced.

6.2.2 Area

The past efforts at automated layout of programmable devices in [26] and [16] both indicated a significant area penalty was incurred due to the automated layout methodology. For the GILES tools, the work in [25] attempted to address these concerns by obtaining an approximate comparison to commercial devices. They found that their automated layouts were 47% and 97% larger than comparable manually laid out Xilinx Virtex-E and Altera Apex 20K400E devices respectively. However, this comparison, by the authors' own admission, was very approximate. To address this concern, we revisit the question of automated versus manual layout area.

The layout area of the device created for this work does not offer a useful point of comparison since there are no manual layouts with a comparable architecture and process technology. Generating our own manual layout would be exceedingly time consuming – avoiding such arduous work is one of the goals of the project. It would also not offer a convincing comparison point since the layout capabilities of the authors certainly do not match those available in industry where many person years of work are dedicated to the task.

Instead, for fair comparison, we chose to compare the layout area to that of a commercial device, the Xilinx Virtex-E [38]. For this comparison we accurately captured the architecture of the Virtex-E using Xilinx FPGA Editor [36]. It was necessary to make some assumptions about the structure of the features in the architecture. The assumptions that were made were based on patents filed by Xilinx and standard design practices. One factor that significantly impacts area is buffer sizing. To improve the accuracy of our comparison, we simulated our version of the routing resources in HSPICE and selected buffer sizes that minimized the area delay product for those resources. We call our representation of Xilinx's design, the Virtex-E capture. A more detailed description of the process used to generate the capture can be found in [18].

Our capture of the Virtex-E is very similar to the actual device. The sum of the cell areas in our capture gives the total active area in our design. If we assume that the tile area of the actual Virtex-E is equal to its active area, we find that the active area of our capture is 11% larger than the actual Virtex-E's active area. (This assumption is reasonable is reasonable based on [5]) The number of configuration SRAM bits in our capture was within 3% of the number of configuration SRAM bits in the actual device. Given this similarity, it is fair to compare the final layout area of the two designs.

For the layout produced automatically, the GILES layout tools are used. Instead of the Cadence Chip Assembly router used for fabricating the chip, we used a custom router that is part of the GILES tools since it is better able to handle high congestion designs. The results from this comparison are summarized in Table 4. It is clear the layout produced manually by Xilinx is the densest but it is very encouraging that the area used by the automated layout tools is only 36% larger than Xilinx's design. This is an improvement over the initial result with the automated layout tools which yielded a layout that was 198% larger [18]. The improved area resulted from more efficient use of metal layers and better grouping of transistors into cells. This result is also an improvement over the prior results obtained in [16] and [26]. This clearly demonstrates that there is promise to this automated FPGA design process at least in some market segments since the increase in production costs due to area can be offset by savings in design time. Furthermore, with a concerted industrial effort we believe that achieving an automated layout that is on par with a manual layout is possible.

Table 4 - Virtex-E Layout Area Comparison

	Area	% increase relative to the Xilinx Virtex-E
Actual 0.18 µm Virtex-E Tile	35,462 μm^2	-
0.18 µm Virtex-E Tile created with GILES layout tools	48,282 μm^2	+36%

7. SUMMARY

We have described a complete automated CAD flow for creating new FPGAs, and used it to design and tape-out a complete FPGA. We have shown that this automation significantly reduces the design time of an FPGA and only suffers from a moderate increase in silicon area. We believe that, with additional effort, the area penalty of automated design can be reduced. Our experience leads us to speculate that it may be possible to achieve better-than-human layouts (because the tools can be given exposure to a broader optimization space than humans can deal with) that will also be fast and power-efficient.

In the future, we plan to explore the impact of this automated methodology on circuit speed and power. This current work focused on functionality and neither speed nor power were optimized. Future work will address this issue by automatically evaluating the impact of transistor sizing.

8. ACKNOWLEDGMENTS

The authors would like to thank Ketan Padalia, Ryan Fung and Mark Bourgeault for their initial work on the GILES layout tools. We are also grateful for the technology and silicon access that was provided by the Canadian Microelectronics Corporation (CMC) for this project. The authors also appreciate Jaro Pristupa's assistance in ensuring easy access to the technology kits and tools we used in this work. Finally, we would like to thank NSERC and Altera for their generous funding of this project.

9. REFERENCES

[1] E. Ahmed and J. Rose. The effect of LUT and cluster size on deep submicron FPGA performance and density. In *Proceedings of the 2000 ACM/SIGDA eighth international symposium on Field programmable gate arrays*, pages 3–12. ACM Press, 2000.

[2] V. Betz and J. Rose. Automatic generation of FPGA routing architectures from high-level descriptions. In *Proceedings of the 2000 ACM/SIGDA eighth international symposium on Field programmable gate arrays*, pages 175–184. ACM Press, 2000.

[3] V. Betz and J. Rose. *Automatic generation of programmable logic device architectures*, October 2003. US Patent 6,631,510.

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

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

[6] M. Bourgeault, J. Slavkin, and C. Sun. *Automatic transistor-level design and layout of FPGAs*. Bachelor's thesis, University of Toronto, 2002. Available online at: http://www.eecg.toronto.edu/~jayar/pubs/ATL/Bourgeault_Slavkin_Sun_2002_Project.pdf.

[7] Cadence, Virtuoso Chip Assembly Router. Datasheet available online at: http://www.cadence.com/datasheets/4886_virtuosoCAR_DSfnl.pdf.

[8] Cadence. SKILL Programming Language, http://www.cadence.com

[9] P. Chow, S. Ong Seo, J. Rose, K. Chung, G. Paéz-Monzón, and I. Rahardja. The design of a SRAM-based field programmable gate array-part ii: Circuit design and layout. *IEEE Trans. on VLSI* Systems, 7(3):321–330, Sept 1999.

[10] J. Cong and Y. Ding, FlowMap: An Optimal Technology Mapping Algorithm for Delay Optimization in Lookup-Table Based FPGA Designs. *IEEE Trans. On CAD*. pp. 1-12, Jan. 1994.

[11] C. Ebeling, L. McMurchie, S. A. Hauck and S. Burns, Placement and Routing Tools for the Triptych FPGA, *IEEE Trans. on VLSI*, 4(3):473-482, Dec. 1995.

[12] A. Egier. *Enhancing and Using an Automatic Design System for Creating FPGAs*. Master's thesis, University of Toronto, 2005.

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

[14] R. Fung. *Optimization of transistor-level floorplans for field-programmable gate arrays*. Bachelor's thesis, University of Toronto, 2002. Available online at: http://www.eecg.toronto.edu/~jayar/pubs/ATL/ryan_fung_2002_thesis.pdf.

[15] W. J. Dally and A. Chang. The role of custom design in ASIC chips. In *Proceedings of the 37th Design Automation Conference*, pages 643–647. ACM Press, 2000.

[16] N. Kafafi, K. Bozman, and S. J. E. Wilton. Architectures and algorithms for synthesizable embedded programmable logic cores. In *Proceedings of the 2003 ACM/SIGDA eleventh international symposium on Field programmable gate arrays*, pages 3–11. ACM Press, 2003.

[17] S. Kirkpatrick, C. Gelatt and M. Vecchi, "Optimization by Simulated Annealing," *Science*, May 13, 1983, pp. 671 –680.

[18] I. Kuon. Automated FPGA Design, Verification and Layout, Master's thesis, University of Toronto, 2004. Available online at: http://www.eecg.toronto.edu/~jayar/pubs/theses/Kuon/IanKuon.pdf.

[19] P. Leventis, *et al.* Cyclone: a low-cost, high-performance FPGA. In *Proceedings of the IEEE 2003 CICC*, pages 49–52, September 2003.

[20] D. Lewis, *et al*. The Stratix™ routing and logic architecture. In *Proceedings of the 2003 ACM/SIGDA eleventh international symposium on Field programmable gate arrays*, pages 12–20. ACM Press, 2003.

[21] A. Marquardt, V. Betz and J. Rose. Timing-Driven Placement for FPGAs. In *Proceedings of the 2003 ACM/SIGDA eleventh international symposium on Field Programmable gate arrays*, pages 37-46.

[22] B. Nguyen, O. P. Agrawal, B. A. Sharpe-Giesler, J. T. Wong, H. M Chang, and G. H. Tran. *Tileable and compact layout for super variable grain blocks within FPGA device*, November 2000. US Patent 6,154,051.

[23] LGSynth93 MCNC Benchmarks. Obtained from http://www.eecg.toronto.edu/~lemieux/sega/ccts_blif.tar.gz.

[24] K. Padalia. *Automatic transistor-level design and layout placement of FPGA logic and routing from an architectural specification*. Bachelor's thesis, University of Toronto, 2001. Available online at: http://www.eecg.toronto.edu/~jayar/pubs/ATL/ketan_padalia_2001_thesis.pdf.

[25] K. Padalia, R. Fung, M. Bourgeault, A. Egier, and J. Rose. Automatic transistor and physical design of FPGA tiles from an architectural specification. In *Proceedings of the 2003 ACM/SIGDA eleventh international symposium on Field programmable gate arrays*, pp. 164–172. ACM Press, 2003.

[26] S. Phillips, Automatic Layout of Domain-Specific *Reconfigurable Subsystems for System-on-a-Chip*. Master's Thesis, Northwestern University, 2001.

[27] S. Phillips and S. Hauck. Automatic layout of domain-specific reconfigurable subsystems for system-on-a-chip. In *Proceedings of the 2002 ACM/SIGDA tenth international symposium on Field-programmable gate arrays*, pages 165–173. ACM Press, 2002.

[28] E. M. Sentovich *et al*. SIS: A System for Sequential Circuit Analysis, Tech. Report No. UCB/ERL M92/41, University of California, Berkeley, 1990.

[29] H. Shin, C. Hu. "Plasma-etching induced damage in thin oxide." In *IEEE/SEMI Advanced Semiconductor Manufacturing Conference and Workshop*, pp 79-83, 1992.

[30] Synopsys Cadabra. Product description available at: http://www.synopsys.com/products/ntimrg/cadabra_ds.html.

[31] Synopsys. HSPICE frequently asked questions. http://www.synopsys.com/products/mixedsignal/hspice/hspice_faqs.html.

[32] Synopsys. Nanosim. Product Description available at: http://www.synopsys.com/products/mixedsignal/nanosim/nanosim.html.

[33] D. Tavana, W. K. Yee, and V. A. Holen. *FPGA architecture with repeatable tiles including routing matrices and logic matrices*, October 1997. US Patent 5,682,107.

[34] Virtual Silicon Technology. Diplomat-18 standard cell library, 2003. http://www.virtual-silicon.com/.

[35] S. Wilton and J. Wu, Private Communication

[36] Xilinx. FPGA Editor, http://toolbox.xilinx.com/docsan/xilinx5/help/fpga_editor/fpga_editor.htm.

[37] Xilinx. Virtex series configuration architecture user guide, September 2000. XAPP151 (v1.5).

[38] Xilinx. Virtex-E 1.8v field programmable gate arrays production product specification, July 2002. DS022-1 (v2.3).