Automatic Generation of FPGA Routing Architectures from High-Level Descriptions

Vaughn Betz and Jonathan Rose[1]
{vaughn, jayar}@rtrack.com

Right Track CAD Corp.,
720 Spadina Ave., Suite #313
Toronto, ON, M5S 2T9

Dept. of Electrical and Computer Engineering,
University of Toronto, 10 King's College Road
Toronto, ON, M5S 3G4

Abstract

In this paper we present a "high-level" FPGA architecture description language which lets FPGA architects succinctly and quickly describe an FPGA routing architecture. We then present an "architecture generator" built into the VPR CAD tool [1, 2] that converts this high-level architecture description into a detailed and completely specified flat FPGA architecture. This flat architecture is the representation with which CAD optimization and visualization modules typically work. By allowing FPGA researchers to specify an architecture at a high-level, an architecture generator enables quick and easy "what-if" experimentation with a wide range of FPGA architectures. The net effect is a more fully optimized final FPGA architecture. In contrast, when FPGA architects are forced to use more traditional methods of describing an FPGA (such as the manual specification of every switch in the basic tile of the FPGA), far less experimentation can be performed in the same time, and the architectures experimented upon are likely to be highly similar, leaving important parts of the design space completely unexplored.

This paper describes the automated routing architecture generation problem, and highlights the two key difficulties — creating an FPGA architecture that matches all of an FPGA architect's specifications, while simultaneously determining good values for the many unspecified portions of an FPGA so that a high quality FPGA results. We describe the method by which we generate FPGA routing architectures automatically, and present several examples.

1. Introduction

In order to develop a high-quality FPGA architecture, one must evaluate the utility of a huge number of architectural tradeoffs and decisions. Typically one "implements" (using a synthesis flow) a set of benchmark circuits in each FPGA architecture (or architecture variant) of interest, and determines the area required and speed achieved by these circuits in each of the architectures [2, 3]. The architecture which leads to circuit implementations with the best combination of area, delay, and perhaps other parameters such as power, is the best FPGA architecture, and is laid out and manufactured.

The architecture of an FPGA specifies both the structure of its logic block and its programmable routing; in this paper we are focusing on the routing architecture portion of an FPGA. To implement circuits in each FPGA routing architecture of interest, one requires both a CAD tool set incorporating sufficiently flexible internal data structures and algorithms that it can target each of these architectures, and a method of describing each FPGA architecture to this CAD tool set. In this paper we are concerned with the second of these requirements — how can one *conveniently* and *quickly* describe an FPGA routing architecture to a CAD tool. If one cannot describe architectures quickly to a CAD tool, the number of architectures with which one can experiment will be quite limited. The net result will be that many portions of the design space remain unexplored, many architecture decisions are not tested with real benchmark circuits run through a real CAD flow, and the final FPGA is not as fully optimized as it could have been.

The most brute-force method of describing an FPGA routing architecture to a CAD tool is to create a directed graph (which we call a *routing-resource graph*) that fully specifies all the connections that may be made in the FPGA routing. This is a very general representation of FPGA routing, and is generally the data structure used internally by the routing tool. It is not very practical to specify this routing-resource graph manually, however, as the routing resource graph for a typical FPGA is 10 MBytes - 200 MBytes in size. Essentially, this is too low-level a description for an FPGA architect to use conveniently.

A more practical alternative is to design a basic tile (a single logic block and its associated routing) manually, and create a program to automatically replicate and stitch together this tile into a routing-resource graph describing the entire FPGA. Even the manual creation of a basic tile is too time-consuming for many purposes, however. A typical tile contains several hundred programmable switches and wires, so it can take hours or days to describe even one tile. Furthermore, such a hand-crafted tile is designed for one value of routing channel width, W (the number of tracks in a channel). In many architecture experiments one must vary W in order to see how routable a given FPGA architecture is, or to determine the minimum value of W that allows some desired fraction of application circuits (say 95%) to route successfully. With a tile-based approach, one must hand-craft one tile for each different value of W, for each architecture. An FPGA designer will often wish to investigate hundreds of different FPGA architectures, and tens of W values for each of these architectures, resulting in thousands or tens of thousands of these basic tiles.

There has been some prior work in describing FPGA routing at a higher level of abstraction. In [4], Brown et al developed an FPGA router for use with island-style FPGAs. In order to quickly investigate FPGAs with different numbers of routing switches,

1. This work was performed at the University of Toronto. Vaughn Betz is now with Right Track CAD, and Jonathan Rose is now at both Right Track CAD and the University of Toronto.

they localized all the code that interacted with switch patterns to two routines, $F_c()$ and $F_s()$. By rewriting these two routines, a user can target their router (CGE) to an FPGA with a different switch pattern. The later SEGA router [5] used the same method to allow retargeting to different FPGAs.

In the Emerald CAD system [6], an FPGA's routing is described by means of WireC schematics — essentially schematics annotated with C-like code that describe switch patterns. The Emerald system can convert these WireC schematics into routing-resource graphs for use by its FPGA router.

While both CGE, SEGA and Emerald reduce the labour required to specify FPGA's routing, they still require considerable effort. Instead of specifying every switch in a basic tile of an FPGA, one writes code (in either C or WireC) to generate all the switches in a basic tile. If the user writes sufficiently general code, it may be possible to change the channel width, W, and have the basic tile adapt properly, but again, it is the user's task to write this (often non-obvious) code.

In this paper we describe a different method of specifying FPGA routing architectures; this specification method has been built into the Versatile Place and Route (VPR) CAD tool [1, 2]. As Figure 1 shows, a user describes an FPGA to VPR via a concise, easily understandable list of parameters. Essentially, the FPGA is described to VPR in a specialized and simple FPGA architecture description language. VPR then uses an internal "architecture generator" to create the routing-resource graph with which the router, graphics, and statistics routines all work. We have designed the architecture description language such that a single architecture description can always be used to generate an FPGA with any value of channel width, W. We can make an analogy with the levels of abstraction possible in software development: manually describing an FPGA by specifying every switch in its basic tile is like programming in machine code (binary), while using WireC or CGE is akin to assembly language programming. We are proposing the use of architecture descriptions that are more like high-level languages; they are easy for humans to create and understand, but require more interpretation by the CAD tools (compilers).

The remainder of this paper is organized as follows. In the next section, we describe the routing architecture description language. Section 3 describes the routing-resource graph used internally by VPR to represent a routing architecture. Section 4 shows how we convert from the easily understood architecture description language input to VPR into the detailed routing-resource graph. We highlight the two major difficulties in this procedure: (i) it is often difficult to meet all the specifications listed by the user, and (ii) VPR must automatically build the portions of the architecture that are left unspecified in a way that results in the best overall FPGA. In Section 5, we show some examples of automatically generated FPGA routing architectures, and Section 6 presents our conclusions and suggestions for future work.

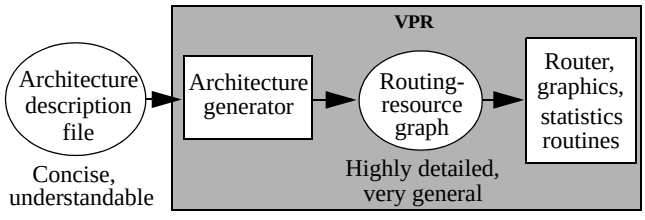


Figure 1: A concise architecture description is converted to a detailed graph description.

2. Architecture Description and Parameterization

We want architecture descriptions to be easy to create, so we tried to parameterize architectures in ways that are intuitive to FPGA researchers. By parameterizing architectures we also make it easier to describe results to other FPGA architects and researchers, and to understand why one architecture is better than another. (Simply showing that one 100 MB routing-resource graph is superior to most others does not allow one to describe results to others very easily!) Indeed, the choice of parameterization is itself a key step in architecture exploration.

Our architecture description is currently intended for use with island-style FPGAs, although it could be extended to other types of FPGAs. Figure 2 shows a typical island-style FPGA. The architecture description file specifies:

- The number of logic block input and output pins,
- The side(s) of the logic block from which each input and output is accessible,
- The logical equivalence between the various input and output pins (e.g. all look-up table inputs are functionally equivalent),
- The number of I/O pads that fit into one row or column of the FPGA,
- The switch block [7] topology used to connect the routing tracks (i.e. which tracks connect to which at a switch block),
- The number of tracks to which each logic block input pin connects, $F_{c,input}$ [7],
- The number of tracks to which each logic block output pin connects, $F_{c,output}$,
- The F_c value for I/O pads, $F_{c,pad}$, and
- One or more *wire segment types*. For each segment type, one specifies:
  - The fraction of tracks in a channel that are of this segment type,

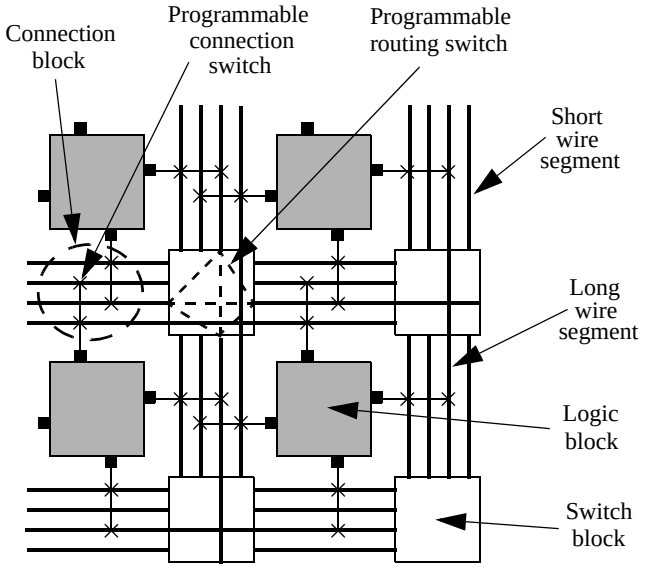


Figure 2: An island-style FPGA (from [2]).

- The segment length (number of logic blocks spanned by a wire segment),
- The type of switch (pass-transistor or tri-state buffer, drive strength, etc.) used to connect a wire segment of this type to other routing segments,
- The switch-block internal population of this segment type (discussed below), and
- The connection-block internal population of this segment type (discussed below).

Note that the segmentation distribution (the fraction of routing tracks of each length), is specified as part of the wire type definitions.

Two of the parameters listed above, switch-block and connection-block internal population, may not be familiar to many FPGA researchers. These two terms were introduced by Chow et al in [8]. They indicate whether or not routing wires and logic blocks, respectively, can connect to the interior of a wire segment that spans multiple logic blocks, or if connections to a wire can be made only at its ends. In [8], a wire segment is either completely internally populated or completely depopulated. We allow *partial depopulation* of the interior of a wire segment. For example, a length five segment spans five logic blocks. If we specify a connection-block population of 100%, this wire segment can connect to all five logic blocks it passes, so it is fully internally populated. If the connection-block population is 40%, it can only connect to the two logic blocks at its ends, so it is internally depopulated. If we specify a connection-block population of 60%, however, the wire can connect to the two logic blocks at its ends and one logic block in its interior, so it is partially internally depopulated. Figure 3 illustrates the four possible values of connection-block population for a length five wire. Switch-block population is specified in a similar, percentage, form.

Notice that we specify the distribution of wire types as fractions of the channel width, W, rather than as an absolute number of tracks of each type. For example, one might say there are 20% length = 2 wires and 80% length = 5 wires. This allows a user to attempt routing with different W values, to determine the routability of an architecture, without changing the architecture file. Similarly, the various F_c values can be specified either as absolute numbers (e.g. 5 tracks), or as a fraction of the tracks in a channel (e.g. 0.2·W).

The number of tracks per channel, W, and the size of the logic block array size can be specified on the command line. If one or more of these parameters is not specified, the VPR router will determine the minimum value(s) needed to fit the circuit in the specified FPGA architecture.

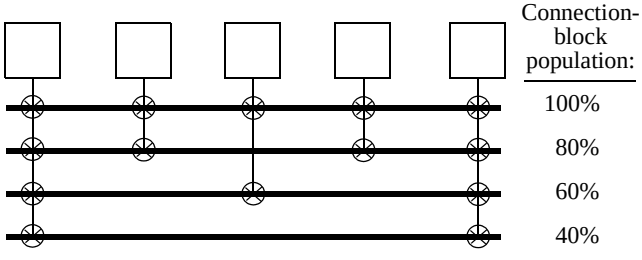


Figure 3: Possible connection-block population values for length 5 wire segments.

Finally, to allow extraction of the delay of routed nets and path-based timing-analysis, one must specify various timing parameters in the architecture description file. These include:

- The input and output capacitance, equivalent resistance, and intrinsic delay of each type of switch used in the routing; as many switch types as desired can be defined.
- The capacitance and resistance of each type of wire segment,
- The delays of all the combinational and sequential elements within each logic block, and
- The delays of the I/O pads.

3. The Routing-Resource Graph

While the architecture parameters listed above are easy for FPGA architects to understand and specify, they are not appropriate for use as an internal architecture representation for a router. Internally, VPR uses a routing-resource graph [9] to describe the FPGA; this is more general than any parameterization, since it can specify arbitrary connectivity. It also makes it much faster to determine connectivity information, such as the wires to which a given wire segment can connect, since this information is explicitly contained in the graph.

Each wire and each logic block pin becomes a node in this routing-resource graph and each switch becomes a directed edge (for unidirectional switches, such as buffers) or a pair of directed edges (for bidirectional switches, such as pass transistors) between the two appropriate nodes. Figure 4 shows the routing-resource graph corresponding to a portion of an FPGA whose logic block contains a single 2-input, 1-output look-up table (LUT).

Often FPGA logic blocks have logically equivalent pins; for example, all the input pins to a LUT are logically equivalent. This means that a router can complete a given connection using any one of the input pins of a LUT; changing the values stored in the LUT can compensate for any re-ordering of which connection connects to which input pin performed by the router. We model this logical equivalence in the routing-resource graph by adding *source* nodes at which all nets begin, and *sink* nodes at which all net terminals end. There is one source node for each set of logically-equivalent output pins, and there is an edge from the source to each of these output pins. Similarly, there is one sink node for each set of logically-equivalent input pins, and an edge from each of these input pins to the sink node.

To reduce the number of nodes in the routing-resource graph, and hence save memory, we assign a capacity to each node. A

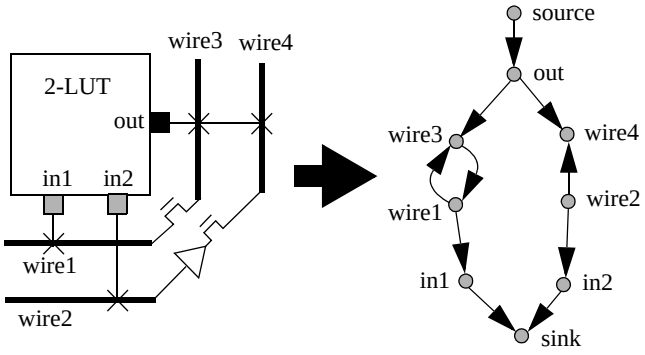


Figure 4: Modelling FPGA routing as a directed graph.

node's capacity is the maximum number of different nets which can use this node in a legal routing. Wire segments and logic block pins have capacity one, since only one net may use each. Sinks and sources can have larger capacities. For example, in a 4-input LUT, there is one group of four logically-equivalent inputs, so we have one sink of capacity four. If we could not assign a capacity of four to the sink, we would be forced to create four logically-equivalent sinks and connect them to the four input pins via a complete bipartite graph ($K_{4,4}$), wasting considerable memory.

To perform timing-driven routing, timing analysis, and to graphically display the architecture we need more information than just the raw connectivity embodied in the nodes and edges of the routing-resource graph. Accordingly, we annotate each node in the graph with its type (wire, input pin, etc.), location in the FPGA array, capacitance and metal resistance. Each edge in the graph is marked with the index of its "switch type," allowing retrieval of information about the switch intrinsic delay, equivalent resistance, input and output capacitance and whether the switch is a pass transistor or tri-state buffer.

4. Automatic Routing Architecture Generation to Match Specified Parameters

As Section 1 described, there are compelling reasons to allow designers to specify architectures in an understandable, parameterized format, and for the routing tools to work with a more detailed, graph-based, description. We therefore need the capability illustrated in Figure 1: a tool that can automatically generate a routing-resource graph from a set of specified architecture parameters. This is a difficult problem for two reasons:

1. We want to create a *good* architecture with the specified parameters. That is, the unspecified properties of the architecture should be set to "reasonable" values.
2. Simultaneously satisfying all the parameters defining the architecture is difficult. In some cases, the specified parameters conflict and overspecify the FPGA, making it impossible to simultaneously satisfy all the specified constraints.

The next section gives a brief overview of our architecture generation approach, while Sections 4.2 and 4.3 illustrate the two difficulties mentioned above.

4.1. Architecture Generation Approach

We generate routing architectures in two phases. First, we build all the unique switch patterns required by the architecture, and one vertical and one horizontal routing channel. Typically the unique switch patterns consist of one connection box (logic block pins to routing wires switch pattern) for each side of the logic block, another connection box for IO blocks, and the switch block (routing wire to other routing wire switch pattern) specified by the user.

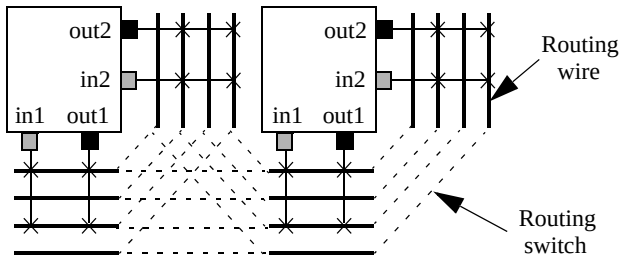
Next, we create the entire FPGA by replicating variants of these basic switch patterns and the canonical channels. As Section 4.3 describes, creating the entire FPGA is more complex than simply replicating these switch patterns and the basic channels across the FPGA; they must be stitched together in a more involved way.

4.2. Creating a Good FPGA Despite Unspecified Architecture Parameters

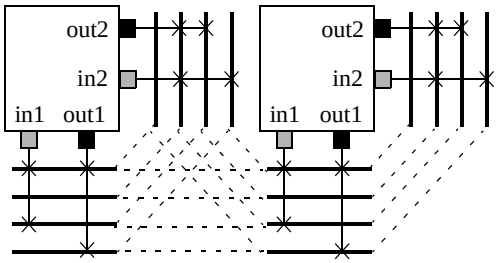
If we require a user to specify every conceivable parameter, and every interaction between these parameters, describing an architecture will be very time-consuming. Instead, we want to allow users to specify the important parameters, and have the architecture generator automatically adjust other parameters of the architecture so that a good FPGA results. For example, we require that a user specify the number of tracks to which input and output pins can connect, $F_{c,input}$ and $F_{c,output}$, rather than requiring a user to specify the complete connection block switch pattern. This certainly simplifies the task of describing an FPGA, but it means that VPR must generate a good connection block switch pattern automatically.

Let us consider this connection block problem in more detail. We decided that the switch pattern chosen should:

- Ensure that each of the W tracks in a channel can be connected to roughly the same number of input pins, and roughly the same number of output pins,
- Ensure that each pin can connect to a mix of different wire types (e.g. different length wires),
- Ensure that pins that appear on multiple sides of the logic block connect to different tracks on each side, to allow more routing options,
- Ensure that logically-equivalent pins connect to different tracks, again to allow more routing options, and
- Ensure that pathological switch topologies in which it is impossible to route from certain output pins to certain input pins do not occur. Figure 5 shows one example of a patho-



(a) Nets starting at out2 can only reach in2, nets starting at out1 can only reach in1



(b) Nets starting at either output can reach either input; vastly improved routability

Figure 5: Example connection block patterns: (a) pathologically bad; (b) good.

logically bad switch pattern — some logic block output pins cannot drive any tracks that can reach certain input pins.

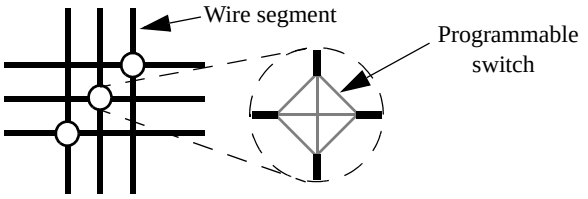
Clearly this is a complex problem. In essence, the proper connection block pattern is a function of $F_{c,input}$, $F_{c,output}$, W, the segmentation distribution (lengths of routing wires), the logical equivalence between pins, and the side(s) of a logic block from which each pin is accessible. The last condition is also a function of the switch block topology. The architecture generator uses a heuristic algorithm that attempts to build a connection block that satisfies the five criteria above, but it will not necessarily perfectly satisfy them all for all architectures.

4.3. Matching All the Architecture Specifications

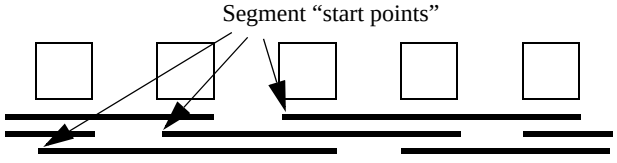
The second difficulty in generating an architecture automatically is simultaneously meeting all the user-defined specifications. We will illustrate this difficulty with an example that shows it often takes considerable thought to simultaneously satisfy the specifications. Consider an architecture in which:

- Each channel is three tracks wide.
- Each wire is of length 3.
- Each wire has an internal switch block population of 50%. That is, routing switches can connect only to the ends of a wire segment (2 of the 4 possible switch block locations).
- The switch block topology is *disjoint* [10]. In this switch block, wires in track 1 always connect only to other wires in track 1, and so on. This is the switch block topology used in the original Xilinx 4000 FPGAs [11].

Figure 6 shows the disjoint switch block topology, and a channel containing 3 wires of length 3. Notice that the "start points" of the wire segments are staggered [12]. This enhances routability, since each logic block in the FPGA can then reach a logic block two units away in either direction using only one wire segment. It also arises naturally in a tile-based layout, so staggering the start points of the segments in this way makes it easier to lay out the FPGA. A tile-based FPGA layout is one in which only a single logic block and its associated routing (one vertical channel segment and one horizontal channel segment) have to be laid out — the entire FPGA is created by replication of this basic tile.



(a) Disjoint switch block



(b) Each channel contains 3 wires of length three.

Figure 6: Architecture specification: (a) disjoint switch block; (b) segmentation distribution.

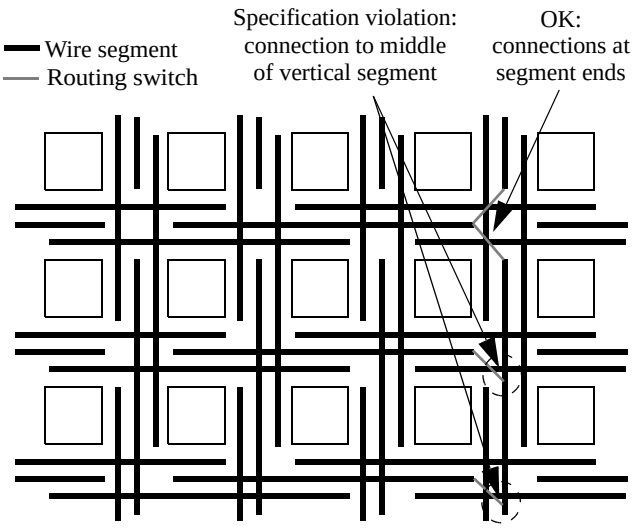


Figure 7: Replicating one channel causes the horizontal and vertical constraints to conflict.

The most straightforward way to create an FPGA with this architecture is to create one horizontal channel and one vertical channel, and replicate them across the array. Switches are then inserted between horizontal and vertical wire segments which the switch block and internal population parameters indicate should be connected. Figure 7 shows the results of such a technique, where only a few of the routing switches have been shown for clarity. Notice that this FPGA *does not* meet the specifications. By inserting routing switches at the ends of the horizontal segments, we are allowing connections into the middle of vertical segments. However, our specifications said that segments should have routing switches only at their ends. If we do not insert switches at the ends of the horizontal segments, however, we cannot connect to the ends of the horizontal segments, so the specifications are again violated. We call this problem a conflict between the *horizontal constraints* and the *vertical constraints.*

The solution to this problem is shown in Figure 8. Instead of simply replicating a single channel, the "start points" of the segments in each channel have to be adjusted. As Figure 8 shows, this allows the horizontal and vertical constraints to be simultaneously satisfied. The specification for the FPGA has been completely realized — every segment connects to others only at its ends, and the switch block topology is disjoint. Figure 9 shows how one can implement this architecture using a single layout tile. This is an additional bonus of this "segment start point adjustment" technique — we not only meet our specifications fully, but create an easily laid-out FPGA.

In order to describe the adjustment of the segment start points more clearly, let us define an FPGA coordinate system. Let the logic block in the lower left corner of the logic block array have coordinates (1,1). The logic block to its right has coordinates (2,1), and the logic block above it has coordinates (1,2), as Figure 8 shows. A horizontal channel has the same y-coordinate as the logic block below it, and a vertical channel has the same x-coordinate as the logic block to its left. We also number the tracks within each channel from 0 to 2, with track 0 being the bottommost track in a horizontal channel, or the leftmost track in a vertical channel.

The proper adjustment shifts the start point of each segment back by 1 logic block, relative to its start point in channel j, when

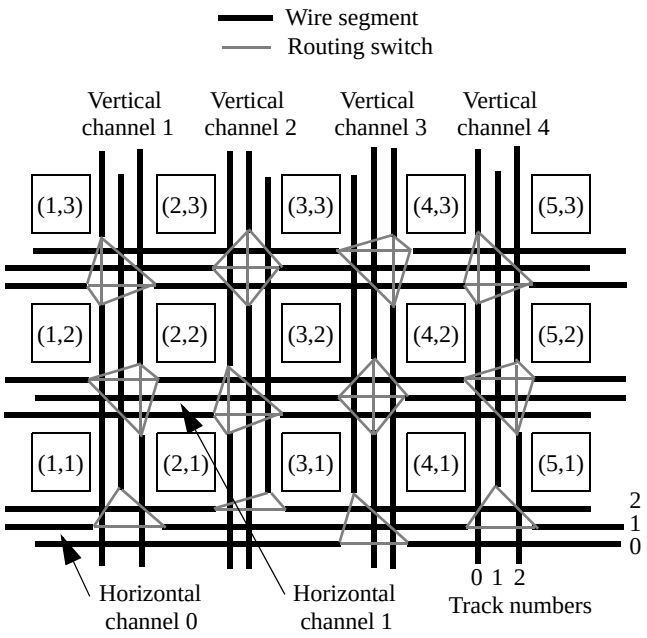


Figure 8: Adjusting the segment start points allows both the horizontal and vertical constraints to be satisfied. The FPGA coordinate system is also shown.

constructing channel j+1. For example, in Figure 8, the left ends of the wire segments in track 0, horizontal channel 0 line up with the logic blocks that satisfy

$$(i + 2)\ modulo\ 3 = 0\,, \qquad (1.1)$$

where i is the horizontal (x) coordinate of a logic block. In channel 1, track 0, however, the left ends of the wire segments line up with logic blocks that satisfy:

$$(i + 3)\ \ modulo\ 3 = 0 \qquad (1.2)$$

A similar shifting back of start points must be performed in the vertical channels — the start point of each segment in channel i+1 is moved back one logic block relative to its start point in channel i.

The shifting of segment start points above allows the horizontal and vertical constraints on an FPGA to be met if either of the following two conditions is met:

- The disjoint switch block topology is used. The segmentation distribution and segment internal populations can have any values. Or,
- All segments are fully switch-block populated. The segmentation distribution and switch block topology can have any values.

If either of these conditions is satisfied, the shifting of segment start points also makes a tile-based layout possible if one additional constraint is satisfied: the number of tracks of length L is divisible by L, for all segment lengths L.

We have not yet found a method to simultaneously satisfy the horizontal and vertical constraints when a switch block topology other than disjoint is used with internally-depopulated segments. It is an open question as to whether there is any method of satisfying both sets of constraints in this most general case. In cases where we cannot make the horizontal and vertical constraints agree, there are locations in the FPGA where a vertical wire wishes to connect to a horizontal wire, but the horizontal wire does not want a switch there, or vice versa. We resolve this conflict by inserting the switch, preferring to err on the side of too many switches in the routing, rather than too few.

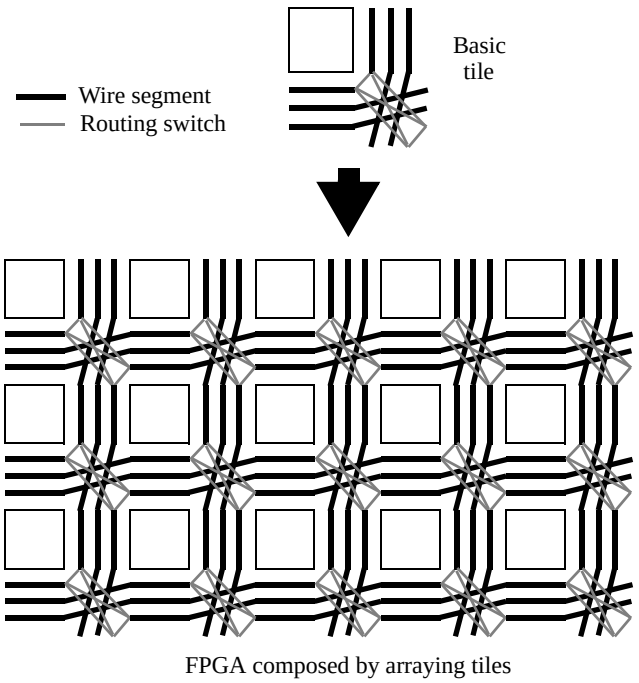


Figure 9: Tiled layout to implement FPGA of Figure 8.

5. Examples of Automatically Generated Routing Architectures

Figure 10 shows a routing architecture description file for an FPGA in which the logic block is a 4-input look-up table plus a register. For a precise description of the architecture description file format, see [13]. Notice that the file is indeed concise — only 38 non-comment lines, 12 of which specify timing and area model information. While this is a simple FPGA architecture, even quite complex FPGA architectures can be described in less than 100 lines with our architecture description language.

Figure 11 shows the FPGA VPR generates to match the architecture specification of Figure 10, when the desired channel width, W, is 10 tracks. These pictures of the FPGA architecture come directly from VPR's built-in graphics. Figure 11a shows the entire FPGA; in this case the FPGA generated consists of a 17 x 17 array of logic blocks, surrounded by IO pads on all four sides. A circuit has been mapped into this FPGA, and logic blocks that were are being used by this circuit are shown as grey squares, while unused logic blocks are shown as white squares. In this case, two IO pads fit into the width or height occupied by a logic block.

Figure 11b is a close-up of a small part of this FPGA so the switch pattern is visible. The black lines are routing wires, while the small black squares are logic block input and output pins. Switches between logic block pins and routing wires are shown as x's. The routing switches in switch blocks are shown as grey lines between routing wires; a small triangle indicates the switch is a tri-state buffer, while a circle indicates it is a pass transistor. In this

```
io_rat 2                    # 2 IO pads per row or column
chan_width_io 1             # All channels the same width.
chan_width_x uniform 1
chan_width_y uniform 1

# 4-input LUT.  LUT inputs first, then output, then clock.
inpin class: 0 bottom       # Equivalence class 0 is LUT inputs
inpin class: 0 left
inpin class: 0 top
inpin class: 0 right
outpin class: 1 bottom      # Output. Not equivalent to anything
inpin class: 2 global top   # Clock.

switch_block_type subset # Also called disjoint switch block.
Fc_type fractional          # Fc values are relative to W
Fc_output 1
Fc_input 1
Fc_pad 1

# Definitions of different types of routing wires.

segment frequency: 0.2 length: 1 wire_switch: 0 opin_switch: 1 \
        Frac_cb: 1. Frac_sb: 1. Rmetal: 4.16 Cmetal: 81e-15

segment frequency: 0.4 length: 2 wire_switch: 2 opin_switch: 2 \
        Frac_cb: 1. Frac_sb: 1 Rmetal: 4.16 Cmetal: 81e-15

segment frequency: 0.4 length: 4 wire_switch: 2 opin_switch: 2 \
        Frac_cb: 1. Frac_sb: 1 Rmetal: 4.16 Cmetal: 81e-15

# Definitions of different types of routing switches.

# Pass transistor switch.
switch 0  buffered: no  R: 196.728 Cin: 20.574e-15 \
        Cout: 20.574e-15 Tdel: 0

# Logic block output buffer driving pass-transistor-switched
# wires.
switch 1  buffered: yes  R: 393.47 Cin: 7.512e-15 \
        Cout: 20.574e-15 Tdel: 524e-12

# Switch used as a tri-state buffer within the routing, and also
# as the output buffer driving tri-state buffer switched wires.
switch 2  buffered: yes  R: 786.9 Cin: 7.512e-15  \
        Cout: 10.762e-15 Tdel: 456e-12

# Used only by the area model.
R_minW_nmos 1967
R_minW_pmos 3738

# Timing info below.  See manual for details.
C_ipin_cblock 7.512e-15
T_ipin_cblock 1.5e-9
T_ipad 478e-12              # clk_to_Q + 2:1 mux
T_opad 295e-12              # Tsetup
T_sblk_opin_to_sblk_ipin 0.
T_clb_ipin_to_sblk_ipin 0.
T_sblk_opin_to_clb_opin 0.

subblocks_per_clb 1
subblock_lut_size 4
T_subblock T_comb: 546e-12 T_seq_in: 845e-12 \
        T_seq_out: 478e-12
```

Figure 10: Example architecture description file.

architecture, we specified that each logic block pin is accessible from only one side. One of the four look-up table input pins is accessible from each side of the logic block, while the output pin can drive routing tracks below the logic block. As specified, each LUT input pin can be reached by every track adjacent to it and the logic block output can drive each track adjacent to it (i.e. $F_{c,input}$ = W and $F_{c,output}$ = W). Notice that there are no switches connecting the clock pin (in the upper right corner of the logic block) to routing wires; the architecture description in Figure 10 specified that the clock must be routed on a special dedicated resource. Also, as we specified, 20% of the routing wires connect to each other via pass transistor switches, while the remainder of the switches in a switch block are tri-state buffers.

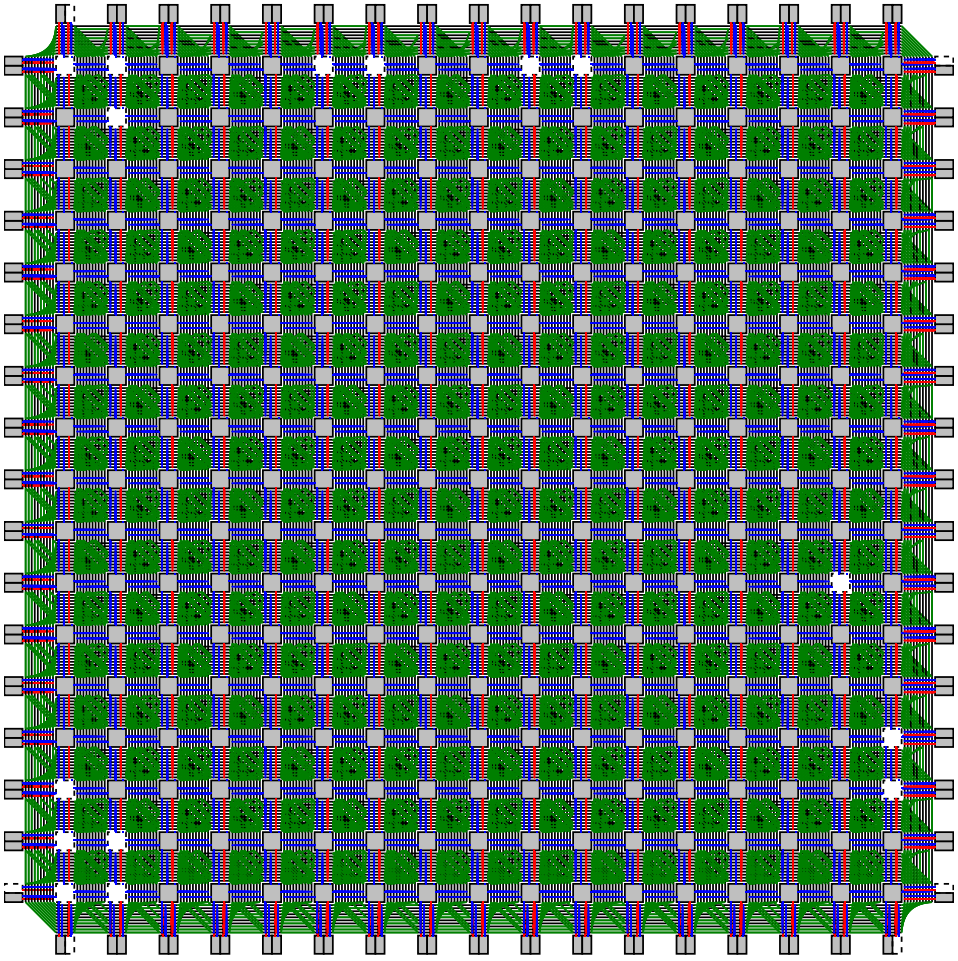
Figure 12 is a less cluttered view, in which the routing switches are not displayed, of the same FPGA. The segmentation distribution of the FPGA is clearly visible: 20% of the routing tracks are length 1 wires (span one logic block before terminating), 40% are length 2 wires, and 40% are length 4 wires. Notice that the "starting points" of the longer wires are staggered to enhance routability.

Figure 13 shows a small portion of an FPGA with a more complex logic block. In Figure 13, the logic block is a "logic cluster" [14, 15] containing four 4-input LUTs and four registers. It has ten logic inputs, four outputs, and one clock input. In this FPGA the connection block switch pattern is more sparse — each of the ten "regular" logic block inputs can connect to only half the tracks in the routing channel beside it, while each of the four outputs can connect to only one-quarter of the routing tracks adjacent to it. All ten logic inputs are logically equivalent in this logic block, as are all four logic outputs. Notice that VPR takes advantage of this logical equivalence in creating switch patterns. The switch patterns for the outputs, for example, ensure that every track can be driven by one of the outputs, and that each output can drive roughly the same number of wires of each type as the other outputs can. Notice also that while the switch pattern looks quite regular, it is not perfectly regular (i.e. the switch pattern for each pin is not merely an offset version of the switch pattern for the other pins). For routing architectures like this, perfectly regular switch patterns often result in some input pins not being reachable from some output pins, which reduces routability. Consequently, VPR checks if its switch pattern generator has created perfectly regular switch patterns which will "disconnect" some inputs from some outputs, and perturbs the switch pattern to solve this problem if necessary.

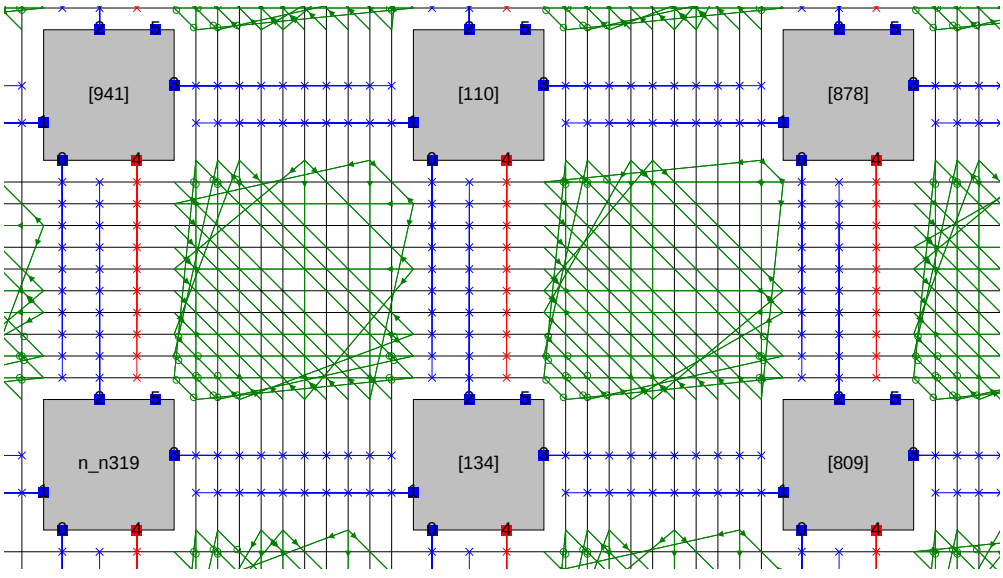
The VPR routing architecture generator is CPU-efficient, requiring only 15 seconds of CPU time on a 300 MHz UltraSparc to build the routing-resource graph of a large (8300 4-input LUTs) FPGA.

6. Conclusions and Future Work

We believe that the automatic generation of FPGA architectures to match a set of specifications is both a key technology for the development of high-quality FPGA architectures and a fertile area for future research. Automatic FPGA architecture generation allows FPGA architects to quickly and easily perform "what-if" experiments on a huge range of FPGA architectures, resulting in a more fully optimized final FPGA architecture to go to manufacturing. Without an easy method to specify FPGA architectures, on the other hand, FPGA architects are likely to experiment with far fewer, and much more similar, architectural ideas, increasing their chance of becoming trapped in a "local minimum" in the FPGA architecture search space.



(a) Entire FPGA



(b) Close-up view

Figure 11: Graphical view of an example FPGA routing architecture; logic block is a four-input LUT + register.

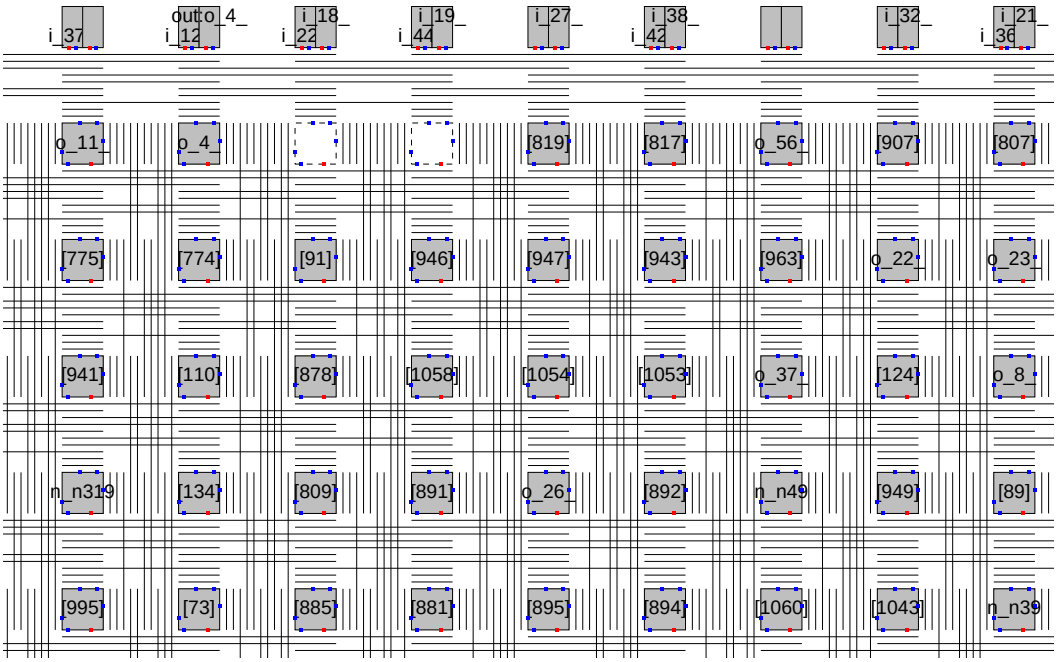


Figure 12: Segmentation distribution of an example FPGA; logic block is a four-input LUT + register.

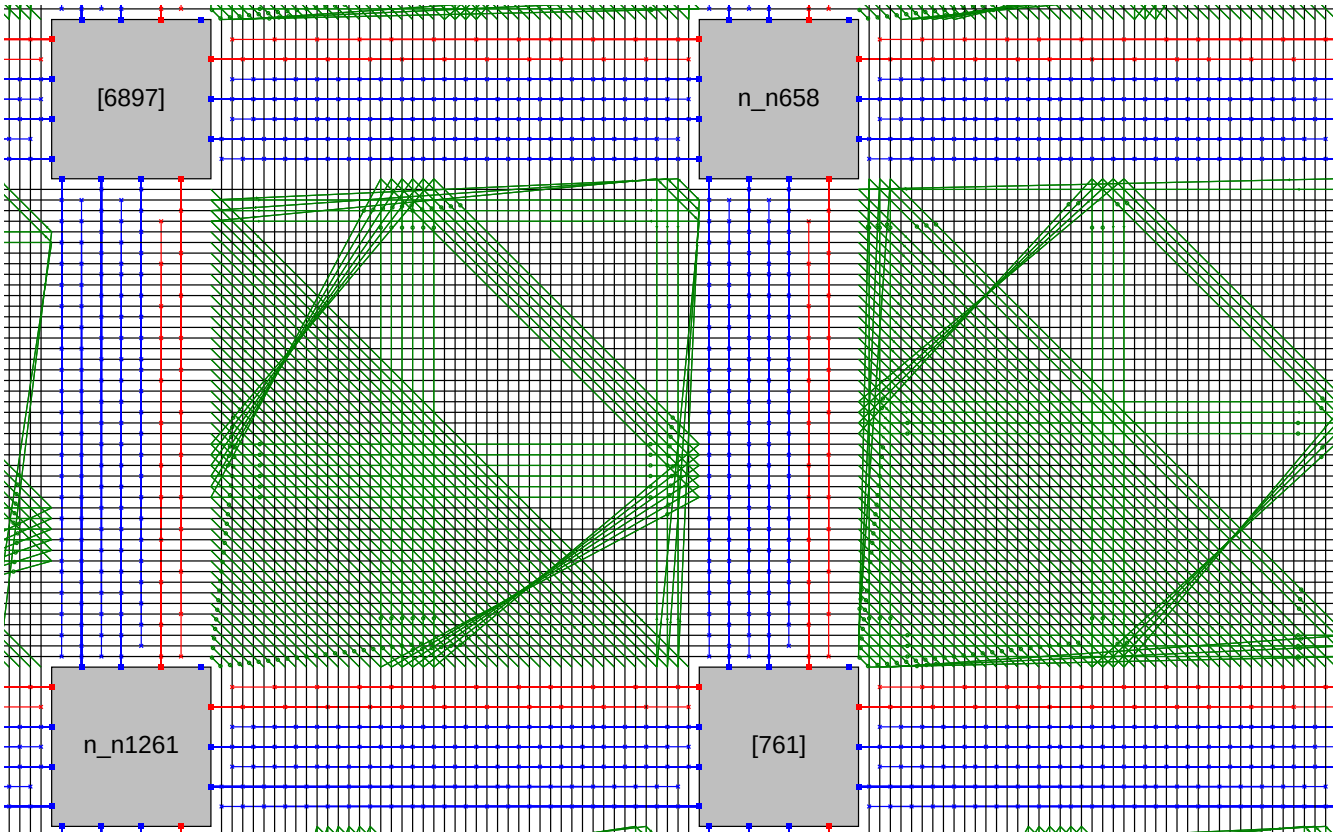


Figure 13: Routing architecture of a more complex FPGA; each logic block contains 4 LUTs + 4 registers.

In this work, we have presented a method of parameterizing and describing island-style FPGAs, and described the method by which we turn these succinct architecture descriptions into a fully-specified FPGA architecture. The two key difficulties in this architecture generation procedure are meeting all the specifications of the FPGA architect, and choosing unspecified parameters intelligently in order to create the most routable FPGA.

While our current architecture generation tool can match all the desired specifications for a wide variety of architectures, there are circumstances where it cannot match all the specifications. Future research can investigate ways to ensure that the generated FPGAs match all their specifications not only by developing better ways of generating architectures from the parameters we have listed in this work, but also by searching for new methods of parameterizing FPGA architectures such that all the specifications can always be satisfied. Similarly, much more work remains to be done in terms of choosing unspecified architectural parameters intelligently enough that the best routability always results.

Finally, throughout this work we have focused on homogenous (one type of routing channel and function block) island-style FPGA architectures. Other styles of architectures can be specified via a high-level description and automatically generated as well, however. In fact a research project at the University of Toronto recently enhanced the VPR architecture generator to allow it to generate Altera-like "long-line" FPGAs [16]. Another important improvement to the architecture generator described in this work would be to allow the automatic generation of heterogeneous FPGAs — that is, FPGAs with several different types of routing channels, or several different types of function blocks. For example, the Lucent Orca FPGAs contain two different types of routing channels [17], while many commercial FPGAs contain two different types of function blocks: logic blocks and RAM blocks.

Acknowledgments

This work was supported by the Information Technology Centre of Ontario, the Walter C. Sumner Foundation, an NSERC 1967 Scholarship, a V. L. Henderson Fellowship and the Canadian Microelectronics Corp.

References

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

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

[3] S. Brown, R. Francis, J. Rose and Z. Vranesic, *Field-Programmable Gate Arrays*, Kluwer Academic Publishers, 1992.

[4] S. Brown, J. Rose, and Z. Vranesic, "A Detailed Router for Field-Programmable Gate Arrays," *IEEE Trans. on CAD*, May 1992, pp. 620 - 628.

[5] G. Lemieux, and S. Brown, "A Detailed Router for Allocating Wire Segments in FPGAs," *ACM/SIGDA Physical Design Workshop*, 1993, pp. 215 - 226.

[6] D. Cronquist and L. McMurchie, "Emerald — An Architecture-Driven Tool Compiler for FPGAs," *ACM Symp. on FPGAs*, 1996, pp. 144 - 150.

[7] J. Rose and S. Brown, "Flexibility of Interconnection Structures for Field-Programmable Gate Arrays," *JSSC*, March 1991, pp. 277 - 282.

[8] P. Chow, S. Seo, J. Rose, K. Chung, G. Paez and I. Rahardja, "The Design of an SRAM-Based Field-Programmable Gate Array, Part I: Architecture," June 1999, pp. 191 - 197.

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

[10] G. Lemieux, S. Brown, D. Vranesic, "On Two-Step Routing for FPGAs," *ACM Symp. on Physical Design*, 1997, pp. 60 - 66.

[11] H. Hseih, et al, "Third-Generation Architecture Boosts Speed and Density of Field-Programmable Gate Arrays," *CICC*, 1990, pp. 31.2.1 - 31.27.

[12] M. Khellah, S. Brown and Z. Vranesic, "Minimizing Interconnection Delays in Array-Based FPGAs," *CICC*, 1994, pp. 181 - 184.

[13] V. Betz, "VPR User's Manual, Version 4.22," *Available from http://www.eecg.toronto.edu/~vaughn/vpr/vpr.html*, Nov. 1998.

[14] V. Betz and J. Rose, "Cluster-Based Logic Blocks for FPGAs: Area-Efficiency vs. Input Sharing and Size," *CICC*, 1997, pp. 551 - 554.

[15] A. Marquardt, V. Betz and J. Rose, "Using Cluster-Based Logic Blocks and Timing-Driven Packing to Improve FPGA Speed and Density," *ACM Symp. on FPGA*s, 1999, pp. 37 - 46.

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

[17] B. K. Britton et al., "Second Generation ORCA Architecture Utilizing 0.5 µm Process Enhances the Speed and Usable Gate Capacity of FPGAs," *IEEE Int. ASIC Conf.*, Sept. 1994, pp. 474 - 478.