# Synthesizable Standard Cell FPGA Fabrics Targetable by the Verilog-to-Routing CAD Flow



JIN HEE KIM and JASON H. ANDERSON, University of Toronto

In this article, we consider implementing field-programmable gate arrays (FPGAs) using a standard cell design methodology and present a framework for the automated generation of synthesizable FPGA fabrics. The open-source Verilog-to-Routing (VTR) FPGA architecture evaluation framework [Rose et al. 2012] is extended to generate synthesizable Verilog for its in-memory FPGA architectural device model. The Verilog can subsequently be synthesized into standard cells, placed and routed using an ASIC design flow. A second extension to VTR generates a configuration bitstream for the FPGA, where the bitstream configures the FPGA to realize a user-provided placed and routed design. The proposed framework and methodology makes possible the silicon implementation of a wide range of VTR-modeled FPGA fabrics. In an experimental study, area and timing-optimized FPGA implementations in 65nm TSMC standard cells are compared to a 65nm Altera commercial FPGA. In addition, we consider augmenting the generic standard-cell library from TSMC with a manually designed and laid-out FPGA-specific cell. We demonstrate the utility of the custom cell in reducing the area of the synthesized FPGA fabric.

CCS Concepts: • **Computer systems organization** → **Reconfigurable computing**; • **Hardware** → Hardware description languages and compilation

Additional Key Words and Phrases: Field-programmable gate array

**ACM Reference Format:**
Jin Hee Kim and Jason H. Anderson. 2017. Synthesizable standard cell FPGA fabrics targetable by the Verilog-to-Routing CAD flow. ACM Trans. Reconfigurable Technol. Syst. 10, 2, Article 11 (April 2017), 23 pages.
DOI: http://dx.doi.org/10.1145/3024063

## 1. INTRODUCTION

Standard cell design methodologies are prevalent in the design of modern digital ICs, owing to the high costs associated with manual layout and increasingly complicated design rules in deep sub-100nm technologies. Today, entire processors [Fluhr et al. 2014] and other digital blocks, such as PLLs [Deng et al. 2015], are mainly synthesized from RTL, as opposed to hand designed at a lower level of abstraction. Field-programmable gate arrays (FPGAs) are one of the few remaining classes of digital IC incorporating a considerable amount of custom layout. The core logic and interconnect tiles in commercial FPGAs are laid out manually, motivated by intense pressure to optimize area, delay, and power in the underlying circuitry, as such tiles are stamped out hundreds to thousands of times on each die. In this article, we consider implementing FPGAs in standard cells and assess the gap between a synthesized standard cell and a full custom commercial FPGA implementation.

Authors' address: J. Kim and J. H. Anderson, Department of Electrical and Computer Engineering, University of Toronto, 10 King's College Road, Toronto, Ontario M5S 3G4, Canada; emails: {kimjin14, janders}@ece.utoronto.ca.

DOI: http://dx.doi.org/10.1145/3024063

To realize a standard cell FPGA implementation, we have developed a synthesizable FPGA fabric generator within the widely used open-source Verilog-to-Routing (VTR) [Rose et al. 2012] toolsuite from the University of Toronto. VTR is capable of modeling and mapping circuits into a wide variety of different FPGA architectures. Our generator produces synthesizable Verilog for VTR's in-memory FPGA device model. As such, our generator is not locked into a single FPGA architecture but rather is able to produce Verilog for a spectrum of different FPGAs, for example, with different numbers of look-up tables (LUTs) per logic block, different numbers of tracks per routing channel, or even different switch block connectivities. In addition to producing synthesizable Verilog, we have also extended VTR to produce a configuration bitstream for a user design implemented within the synthesizable FPGA. Whereas the conventional approach used by commercial vendors involves adding CAD support for each new FPGA device, in our case we have built "silicon support" for an existing and established FPGA architecture/CAD evaluation toolsuite—VTR.

In addition to the advantages associated with synthesis versus custom layout, the proposed synthesizable FPGA fabric generator brings several benefits. First, it enables VTR-modeled FPGAs to be realized in silicon, democratizing access to FPGA technology. Specifically, our VTR-based approach circumvents a major impediment to the development of new FPGAs, namely the complexity and cost associated with building CAD tools that can map user circuits into them. Second, the synthesizable FPGAs can be easily ported to new process technologies by resynthesizing using a new cell library, and similarly fabrics with different area/performance/power trade-offs can be realized easily by changing constraints provided to the ASIC toolflow. Third, the FPGA fabrics we generate are straightforward to incorporate into a system on chip (SoC)—the FPGA module can be instantiated within the surrounding circuitry, and the layout shape/aspect ratio of the FPGA tiles can be tailored according to the overall SoC floorplan. Fourth, the tool can potentially be used in FPGA architecture research to rapidly evaluate the area/performance/power consequences of architectural decisions via analysis of low-level standard cell implementations. The latter benefit may be particularly useful in synthesizing and evaluating domain-specific programmable fabrics (e.g., fabrics having significantly less routing richness or having directionally biased routing).

We synthesize FPGA fabrics into TSMC 65nm standard cells. Through constraints supplied to the ASIC design tools (Synopsys Design Compiler and Cadence Encounter), we produce area-optimized, timing-optimized, and balanced FPGA fabric implementations. In an experimental study, we supply VTR with an architecture model closely resembling Altera's Stratix III device and compare the area and delay of the synthesized standard cell FPGA with Stratix III, which is also implemented in 65nm. Then we incorporate a custom-laid-out 16-to-1 MUX cell to the standard cell library and study its impact on area efficiency. The contributions of this article are as follows:

(1) An FPGA fabric generator, built within VTR, capable of producing synthesizable Verilog RTL for a variety of architectures
(2) A configuration bitstream generator for the synthesizable FPGAs
(3) An area/performance comparison between several synthesized standard cell FPGAs and, to the authors' knowledge, the first published study comparing a full custom commercial FPGA with a synthesized standard cell FPGA
(4) A custom standard cell, designed specifically for FPGAs and its integration into the TSMC library, producing an extended cell library
(5) An area/performance analysis of an FPGA synthesized into the extended library.

Compared to the preliminary conference publication [Kim and Anderson 2015], this article includes an additional fabric (referred to as the MUX4-optimized fabric), wherein the automatically generated Verilog has been altered to provide better technology mapping results. For instance, Design Compiler makes better use of the cells in

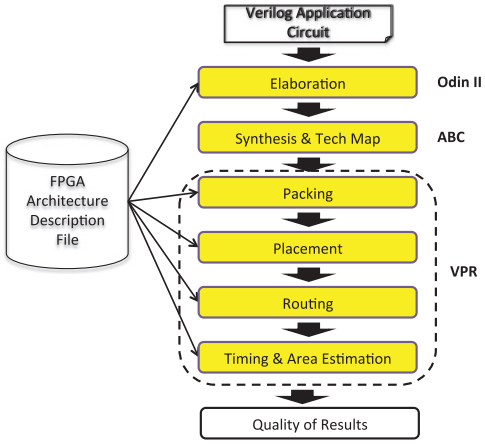


Fig. 1. VTR flow.

the target library. This provides a significant improvement to the previously considered timing-optimized fabric. Another direction pursued is the design and incorporation of a custom-designed FPGA-specific cell into the TSMC library. The custom cell is demonstrated to provide considerable benefits in area.

The remainder of this article is organized as follows. Section 2 describes related work and provides a background for subsequent sections. The VTR-based synthesizable fabric and bitstream generation is introduced in Section 3. Section 4 describes the ASIC flow we used to produce a standard cell implementation. The experimental study appears in Section 5. The custom standard cell design and its usage is described in Section 6. Conclusions and future work are offered in Section 7.

## 2. BACKGROUND AND RELATED WORK

### 2.1. Verilog-to-Routing

VTR [Rose et al. 2012] is an open-source FPGA architecture evaluation/CAD framework from the University of Toronto, comprising RTL synthesis, logic synthesis, packing, placement, routing, and timing/power analysis, as shown in Figure 1. The inputs to VTR are (1) a description of an FPGA architecture and (2) an application benchmark for implementation in the FPGA. The architectural description is written in human-readable XML, and through this an architect can specify both the interconnect and logic architecture of the target FPGA. VTR's internal CAD algorithms are adaptive to the architecture file input in the sense that specific architectural details are not hard coded into the algorithms themselves—the algorithms are designed to do a reasonably good job of implementing the application benchmark in a range of architectures. Note that prior to the current work, the VTR flow terminated at the routing stage. This means that it was not possible to realize a silicon implementation of a VTR-modeled architecture. Our work extends VTR to produce synthesizable Verilog for VTR's in-memory architectural device model, as well as a bitstream for the application benchmark implemented in the device.

### 2.2. FPGA Architecture

VTR is able to model *island-style* FPGAs [Betz et al. 1999]—two-dimensional arrays of logic blocks with horizontal and vertical routing channels surrounded by a ring of I/Os. The key architectural components necessary to understand the concepts presented in this wokr are shown in Figure 2. Switch blocks allow horizontal and vertical routing tracks to be programmably connected with one another; connection blocks allow logic

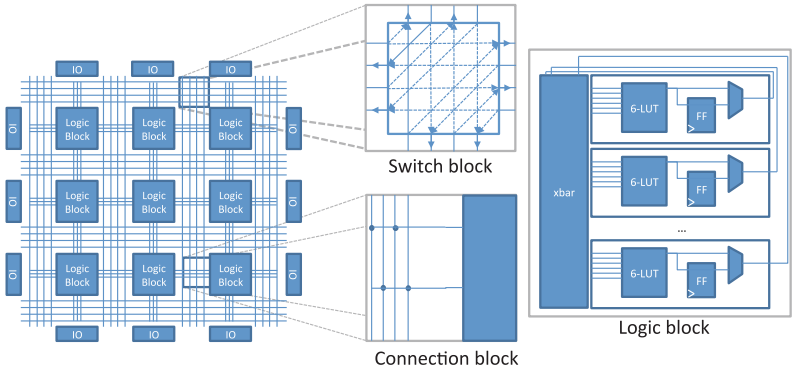


Fig. 2. FPGA architectural components.

block pins to connect to adjacent routing tracks. Logic blocks generally contain one or more LUTs and flip-flops (FFs), and an internal crossbar for making local connections.

With respect to routing, VTR allows one to change the number of tracks per channel, wire directionality, the wire segment lengths and relative frequency of wires of a given length, the connectivity between horizontal and vertical wires, and the way wires connect to logic block pins. For logic, VTR permits heterogeneity, where columns of blocks may be of different types, such as LUT-based soft logic blocks, DSP blocks, and memories. Within each of these types, an architect has a wide range of choices. For example, with soft logic blocks, one can vary the number of of LUTs per block, whether the LUTs are fracturable [Altera 2015] versus nonfracturable, the richness of the internal local crossbar, and the number of FFs. VTR also supports the notion of *modes*, which represent mutually exclusive ways in which a block may function. For example, a fracturable LUT may operate in single-output mode (implementing a single logic function) or dual-output mode (implementing two logic functions).

### 2.3. Related Work

There has been prior work on automating the creation of FPGAs. One approach is to use circuit generators for creating an FPGA. An example of this is GILES [Kuon et al. 2005], wherein the FPGA circuitry is directly generated rather than synthesized by an ASIC flow. Another approach is to use ASIC tools to synthesize, place, and route the circuits into standard cells.

Several recent works bear similarity to our own in that they propose to synthesize FPGA fabrics targetable by VTR. Chaudhuri et al. [2008] focuses on embedding a reconfigurable FPGA in an SoC and enhance the area and performance through floorplanning [Chaudhuri et al. 2007]. This work achieved an area of $0.6u\text{m}^2$ for an $8 \times 8$ FPGA in a $0.13u$m process. Liu [2014] studies the impact of the FPGA architectural parameters on the synthesized components of the FPGA. The author reported an area of $0.026\text{mm}^2$ per tile with a 2ns critical path with channel width of 160 and 10 six-input LUT per logic block using a 65nm process. In both of these works, there is little detail on the issues that arise from using ASIC design tools. Moreover, none of these works show a suite of benchmark designs being verified as functional within the synthesized fabric, nor do they compare the synthesized standard cell implementation with a commercial FPGA.

In another work, Aken'Ova [2005] investigated island-style FPGAs and improved area and delay gap by using "tactical cells" [Aken'Ova et al. 2005] and floorplanning [Aken'Ova and Saleh 2006]. The authors thoroughly describe architecture changes and solutions to overcome ASIC design flow obstacles encountered. They reported that

their fabric synthesized using an extended standard cell library was 2× to 2.8× larger than a custom FPGA and 10% slower. The custom FPGA area and delays were estimated using VPR. However, there is little discussion on the generation of the architecture and bitstream. A very recent work, FPRESSO [Zgheib et al. 2016], used an ASIC flow to synthesize portions of an FPGA into customized cell libraries for the purpose of extracting timing and area estimates to inform VTR models. Conversely, we focus on the synthesis of complete fabrics into a foundry's standard cell library.

Other work has focused on standard cell implementations of application-specific FPGA architectures. An early work by Phillips and Hauck [2002] synthesized the reconfigurable-pipelined datapath (RaPiD) [Ebeling et al. 1996] architecture using standard cells. The authors observe that customizing the architecture for domain-specific applications, as well as including some FPGA-specific standard cells into the library, improves area and performance. Kafafi et al. [2003] synthesize a combinational and directional architecture and report a large area difference relative to a custom layout design. Wilton et al. [2007] synthesize a datapath-oriented FPGA fabric with a directional routing architecture. Unlike these past works, which deal with nonstandard FPGA architectures, we focus on architectures that resemble today's commercial FPGAs and are already supported by the VTR framework.

## 3. ARCHITECTURE AND BITSTREAM GENERATION

VPR [Betz and Rose 1997] is the portion of the VTR flow that performs packing, placement, and routing. From the user-supplied architectural description, VPR builds an in-memory representation of the entire FPGA device, including all logic and interconnect. The packing, placement, and routing steps in VPR implement the application benchmark in an in-memory FPGA device model. Our synthesizable Verilog generator is built within VPR and executes at the end of the routing step. Essentially, our generator code "walks" the in-memory device model to produce synthesizable Verilog, and likewise by examining the application benchmark's implementation in the device, we produce a configuration bitstream for the FPGA. We elaborate on these steps in the following.

```
module fle ( clk, reset, in, config, out);

input clk, reset;
input [7:0]in;
input [66:0] config;
output out;

wire in_4, in_5, lut4_0, lut4_1, lut4_2, lut4_3, lut5_0, lut5_1, lut6;
wire out_lut_0, out_lut_1, out_ff_0, out_ff_1;

mux2_inner fracture_lut_in_4 ( .config(config[64]), .in({in[4], in[2]}), .out(in_4) );
mux2_inner fracture_lut_in_5 ( .config(config[64]), .in({in[5], in[3]}), .out(in_5) );

lut4 lut4_0 ( .config(config[15:0]), .in(in[3:0]), .out(lut4_0) );
lut4 lut4_1 ( .config(config[31:16]), .in(in[3:0]), .out(lut4_1) );
lut4 lut4_2 ( .config(config[47:32]), .in({in_5, in_4, in[1:0]}), .out(lut4_2) );
lut4 lut4_3 ( .config(config[63:48]), .in({in_5, in_4, in[1:0]}), .out(lut4_3) );

mux2_inner lut5_0_0 ( .config(in[6]), .in({lut4_1, lut4_0}), .out(lut5_0) ); //[6][3][2][1][0]
mux2_inner lut5_0_1 ( .config(in[6]), .in({lut4_3, lut4_2}), .out(lut5_1) );
mux2_inner lut6 ( .config(in[7]), .in({lut5_1, lut5_0}), .out(lut6) ); //[7][6][3][2][1][0]

mux2_inner fracture_lut ( .config_in(config_in[64]), .in({lut5_0, lut6}), .out(out_lut_0) );
mux2_inner lut5_1 ( .config_in(in[7]), .in({lut4_3, lut4_2}), .out(out_lut_1) ); //[7][5][4][1][0]

ff ff_0 ( .clk(clk), .reset(reset), .D(out_lut_0), .Q(out_ff_0) );
ff ff_1 ( .clk(clk), .reset(reset), .D(out_lut_1), .Q(out_ff_1) );

mux2_inner mux_bypass_0 ( .config(config[65]), .in({out_ff_0, out_lut_0}), .out(out[0]) );
mux2_inner mux_bypass_1 ( .config(config[66]), .in({out_ff_1, out_lut_1}), .out(out[1]) );

endmodule
```

Listing 1. Fracturable LUT Verilog.

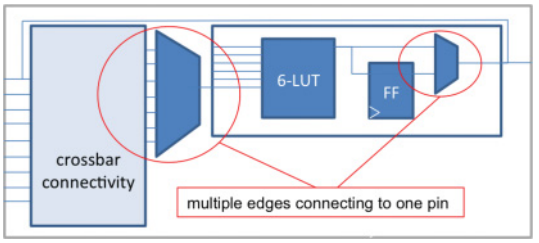


Fig. 3. MUX inference within a logic block.

### 3.1. Generating Synthesizable Verilog

As a first step, we handwrote Verilog for two FPGA primitives: an FF and a Stratix III–like fracturable LUT (see Section 5). Subsequently, we automatically generate Verilog for the entire FPGA device, a stuctural netlist of these primitives, as well as other primitives that are generated by our generator code: multiplexers (MUXes) of any size and LUTs with any number of inputs. An example of primitive Verilog (fracturable LUT) is shown in Listing 1. It uses 67 configuration bits in which 64 bits are used for the LUT configuration, 1 bit is used for selecting the mode, and 2 bits are used for optional FF on the output. The generation must handle the following: logic blocks, intra-logic block routing, inter-logic block routing, and configuration cell memory.

*Logic Blocks*. Logic blocks in VPR are represented in memory hierarchically as a tree. The tree root represents the entire logic block, nodes at intermediate levels of the tree represent levels of hierarchy in the block, and the leaves represent the primitives (LUTs and FFs). We generate the Verilog for each logic block by first traversing to the leaf nodes. We then move up the tree, and as we visit each node in the hierarchy, its child nodes are defined and instantiated in the output Verilog. The synthesizable Verilog generated for a logic block has the same hierarchy specified by the architect in the architecture file.

*Intra-logic Block Routing*. Routing within a logic block is stored in memory as a graph, where nodes represent pins (on primitives or on intermediate levels of hierarchy) and directed edges represent connections between pins. For a given pin, if there is more than one incoming edge, a routing MUX is inferred. This means that in this case, the pin can be driven from one of several pins that connect to the data inputs of the MUX. The select inputs to the MUX will be driven by configuration cells (discussed in the following). Figure 3 highlights examples of routing MUX inference within a logic block. Crossbars with varying degrees of connectivity can be generated, as VPR only creates edges in its in-memory model for those connections that exist in the architecture. This implies that the fabric generator is not tied to a specific style of intra-logic block routing.

*Inter-logic Block Routing*. Routing that connects the logic blocks is likewise represented in memory as a graph in VPR. In this case, the nodes represent the wire segments and pins. Edges represent programmable connections between such conductors. As earlier, where there exists more than one edge to a node, MUXes are inferred. These MUXes correspond to the connectivity within switch blocks and connection blocks (see Figure 2). VPR does not model the inter-logic block routing hierarchically—there is no notion of a switch block or connection block within VPR's in-memory model. Consequently, for inter-logic block routing, each MUX is instantiated in our Verilog without hierarchy.

*Configuration Cells*. As MUXes that implement programmable connectivity are being instantiated, configuration cells that drive their select inputs must also be instantiated and attached accordingly. We use "fully encoded" MUXes, meaning that a 4-to-1 MUX will have two configuration bits. Other styles of MUX (e.g., flattened MUXes that use more configuration cells and have fewer levels from input to output) are left to consider in future work. We use an FF to implement each configuration cell. Then, the FF cells are connected in a chain as a shift register. Similarly, for the LUTs in logic blocks, we instantiate configuration cells to hold the LUT's truth table contents. Future work involves considering alternatives to the configuration cell shift register, whose weaknesses include rendering the entire device useless if a single FF is faulty, potentially increased area to avoid hold time violations, and invalid/illegal configuration states at intermediate points in the shifting.

### 3.2. Bitstream Generation

The configuration bitstream is an ordered sequence of zeros and ones that configures the FPGA according to the implementation of the application benchmark. Since the configuration FF cells are connected together in a chain, the `0/1` values shifted in for the benchmark's implementation must align exactly with the ordering of cells in the chain. Thus, to create the configuration bitstream for a design, our generator walks the device model in precisely the same order as is used to generate the synthesizable Verilog. The in-memory implementation of an application benchmark within the device is used to assign `0/1` values in the bitstream. For example, consider a 4-to-1 interconnect MUX whose inputs are numbered 0, 1, 2, 3. The path selected through the MUX will be controlled by two configuration cells. Assuming that VPR has routed a signal through input #1, the two configuration cell values in the stream will be `01`. There were two challenges worth highlighting, on which we elaborate in the following.

*Input and Output Equivalence*. VPR supports input and output pin equivalence (essentially "pin swapping"). This means that as we generate the bitstream, we have to account for any change in the ordering of the inputs or outputs that may have occurred during routing. For example, consider a MUX within the intra-logic block crossbar. At the packing stage, VPR may have used the $i^{th}$ input to the MUX for a logic signal; however,instead the VPR router may end up using the $j^{th}$ input for the signal (e.g., for timing/routability reasons). During bitstream generation, we account for such changes by examining the routing paths actually used by nets and do not rely on the packed (prerouted) netlist.

*Fracturable LUTs*. When LUTs are not fracturable, we may assume that unused inputs are grounded and configure the LUT truth table accordingly. However, with fracturable LUTs, we must account for inputs that are shared between the LUTs. For example, fracturable LUTs in Altera commercial devices have eight inputs, where two inputs are shared between the two LUTs. When one of the shared inputs is used in the first LUT but unused in the second LUT, we can no longer assume that input will be grounded when we specify the truth table for the second LUT. The truth table for the second LUT must be set in such a way that the unused input is a "don't care": the LUT function must be correct regardless of whether the unused input is a zero or a one. This involves replicating the truth table contents for both possible logic states of the unused input.

### 3.3. Functional Correctness

Figure 4 shows the verification flow. We developed a testbench where the original application benchmark RTL is simulated in ModelSim with random vectors. Within

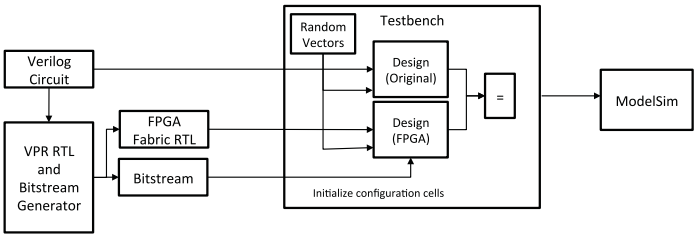


Fig. 4. Synthesizable FPGA verification flow.

the same testbench, the FPGA device RTL, configured with the generated bitstream, is simulated with the same random vectors. Output values are checked for equality with each vector applied. Note that this verification flow was used to check correctness at all stages of the standard cell implementation: RTL generated by our VPR generator, post-technology mapping with Synopsys (discussed later), and post-layout with Cadence (also discussed later).

### 3.4. Architectures Supported by the Tool

Presently, our tool is able to generate synthesizable Verilog for FPGAs comprised of LUT/FF-based logic blocks, interconnect, and I/Os. Support for other types of blocks, such as DSP or RAM blocks or configurable I/Os with varying signally standards, is left as future work. We support LUTs that are either fracturable or nonfracturable. In fact, LUT fracturability is the only form of VTR "modes" [Rose et al. 2012] supported by our tool. The modes feature in VTR allows an architect to describe mutually exclusive functionality for a given block. However, the specification of modes to VTR does not contain information regarding how such functionality should be implemented in hardware, nor is it obvious how it could be inferred automatically by a tool such as ours.

Aside from these limitations, our tool supports Verilog/bitstream generation for all VTR-targetable architectures—made possible by the approach described earlier, which walks VTR's in-memory device model. For example, we are able to handle any number of LUTs per logic block, any switch block/connection block connectivity, wire segments of various lengths, and fully or partially populated crossbars within logic blocks.

## 4. STANDARD CELL ASIC IMPLEMENTATION

We use an ASIC design flow to synthesize, place, route, and analyze the FPGA circuit, as summarized in Figure 5. We used Synopsys Design Compiler to synthesize the FPGA to standard cells. Cadence Encounter is used for placement and routing. Synopsys PrimeTime is used for timing analysis.

### 4.1. Synthesis to Standard Cells

There are many ways of describing a given hardware circuit in Verilog. For example, the circuit can be described structurally or behaviorally. Another flexibility concerns the specification of more complex blocks, which can be in a hierarchical or flat style. A hierarchical approach would entail the instantiation of modules within other modules, whereas a flat approach would instantiate and interconnect modules at one level of hierarchy.

Recall from Section 2 that an architecture in VPR is described as a collection of modules that are grouped together to form a larger module in the architecture file. Therefore, we opted for structural and hierarchical Verilog as the best choice for representing the components of an FPGA, except the primitives. The primitives

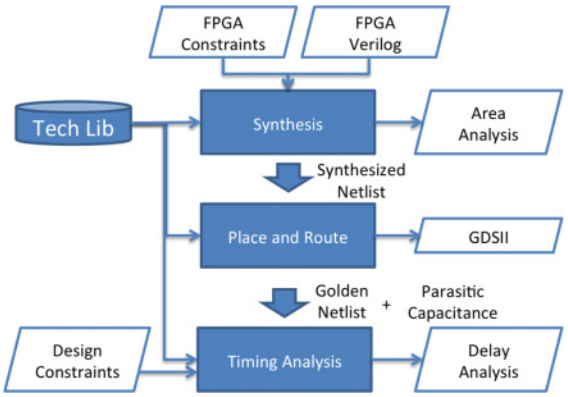


Fig. 5. ASIC design flow.

are described behaviorally to avoid library dependencies. FFs are straightforward to define, but LUTs and MUXes can be defined in various ways. Since LUTs and MUXes have a very similar hardware implementation, we studied the effect of Verilog style on the area and performance of MUXes. First, a 4-to-1 MUX was written in three different styles: *always*, *assign*, *and case*. When compiled, we observed that all of the styles of Verilog mapped to a 4-to-1 MUX in the standard cell library (i.e., all Verilog styles mapped to a single standard cell). Then a 16-to-1 MUX was written with hierarchy using the previous 4-to-1 MUX and also written in a flat manner with one large case statement. With the hierarchical approach, five 4-to-1 MUX cells were used in the standard cell implementation, whereas the flat design resulted in many logic gates of different types being used. This demonstrated that Synopsys Design Compiler is unable to recognize large MUX designs, which led to underutilization of MUX cells in the standard cell library and a poor area-performance trade-off. However, it is important to note that using the 4-to-1 MUX hierarchical method may be biased to the particular standard cell library used (TSMC 65nm), as well as biased to the synthesis tool (Design Compiler).

We evaluated several different synthesis strategies: top down, "uniquify," or bottom up. The top-down method is a push-button approach where the entire design is synthesized in "one shot." However, since it processes the whole design at once, it is too runtime and memory intensive to be a viable approach for a large design. In fact, for a $20 \times 20$ FPGA with 300 tracks per channel, Design Compiler could not successfully synthesize using the top-down approach. The uniquify approach allows one to break up the design and compile each instance separately. This approach worked; however, it is again runtime intensive, as each instance of the same Verilog module (e.g., a 6-LUT) is compiled individually. We therefore chose the bottom-up approach, in which each required Verilog module is synthesized just once, and the synthesized instances are stitched together to compose the overall synthesized design.

Although the bottom-up method produces a more regular implementation and brings runtime benefits, its weakness is that each type of module is synthesized in isolation—that is, outside of the context of the other modules it connects to when instantiated in the overall FPGA. For example, consider that for a length-16 wire, it may be truncated at the edge of the FPGA, depending on the location from which it is driven. Length-16 wires truncated at different points will all present different load capacitances, and it is undesirable to synthesize a separate/different driver to be used for each variant of truncated length-16 wire. To handle these issues that arise from routing MUXes driving various-length wires, we did the following: (1) we synthesized a single MUX

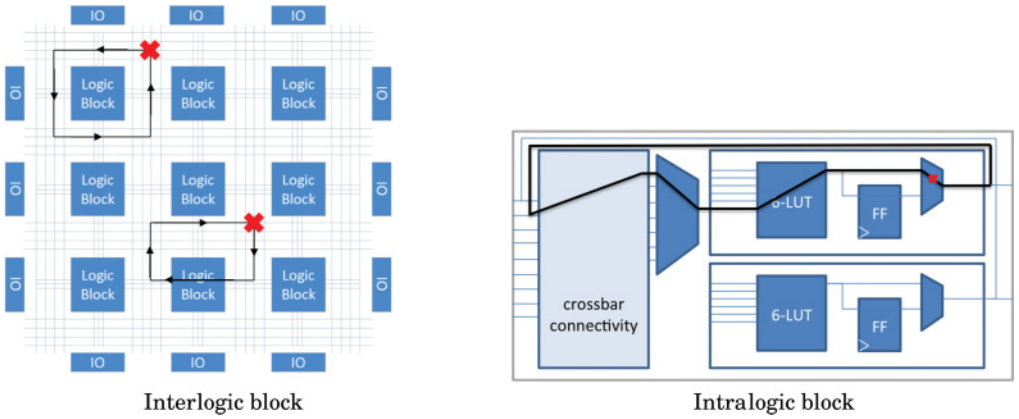


Fig. 6. Breaking combinational loops in routing.

of each size; (2) we inserted a fixed-size buffer[1] on the output of each MUX to create a consistent load on the MUX output; and (3) we inserted a fixed-size buffer every two tiles on inter-logic block interconnect wires, ensuring a roughly uniform load for each buffer. Because the FPGA is floorplanned, wirelengths and therefore capacitive loads are fairly predictable. This allows us to insert a buffer of the appropriate size for the wire load; specifically, the buffer size was selected according to the standard cell databook, which specifies the load that each buffer size can drive.

Design Compiler accepts area and timing constraints, permitting one to trade off performance versus area for a single RTL design by changing constraints. In our experimental study (Section 5), we synthesize area-optimized, timing-optimized, and balanced FPGAs. Optimizing for area is straightforward: we direct Design Compiler to achieve a target area of zero. Optimizing for timing is more involved, owing to the fact that FPGAs contain many combinational loops before being programmed. In other words, the ASIC tools are synthesizing an *unconfigured* FPGA and "see" many paths from a cell output, through routing, and back again to the same cell—combinational loops that are not present when the FPGA is configured for a practical/real application. Such loops are problematic for timing analysis, and they must be "broken" prior to timing-constrained synthesis. The loops exist within both inter- and intra-logic block routing, and combinations of these. Figure 6 show examples of combinational loops and how we break such loops (via automatically generated constraints provided to Synopsys). It is clear that inter-logic block routing will contain many of these loops, and to ensure that all loops are broken, all of the MUXes in the top level will be disabled during top-level timing analysis. This means that when compiling the MUXes, timing constraints are still applied. Within a logic block, there are various places the loop can be broken. We decided to break the loop at the output MUX, which selects between the combinational or registered path. This choice ensures that paths through the FF will be timing optimized. The loops are broken using the `set_disable_timing` constraint. In essence, after breaking such loops and by using the bottom-up synthesis approach, we are able to produce a timing-optimized implementation of each module; however, all possible timing paths through the overall FPGA (i.e., across modules) are not optimized globally. Nevertheless, results in the next section demonstrate significantly improved

[1]The Cadence Encounter router also has capabilities for automatic buffer insertion (command `optDesign`); however, because of the size of the design being placed and routed, the router-based buffer insertion repeatedly crashed on our server. We therefore opted to insert buffers during synthesis.

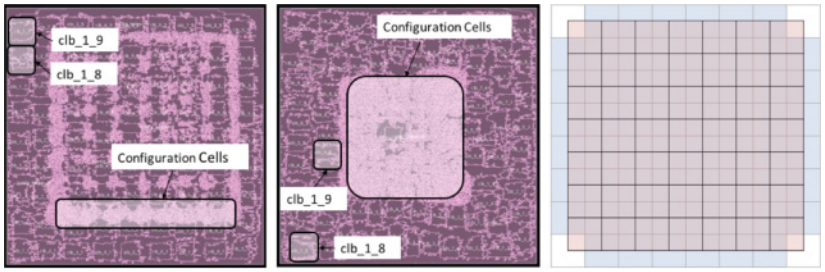


Fig. 7. Floorplanned (left) and unfloorplanned (center) layout and floorplan grid (right).

performance in the timing-optimized implementations. Note that timing constraints are only applied to paths through which logic signals may propagate in an application implementation. We do not apply timing constraints to the configuration cells or paths to and from such cells. The content of such cells only changes when the device is configured; hence, they are not performance critical.

### 4.2. Place and Route

Placement and routing proceeds in a flat manner, allowing optimization across the module boundaries. To help the placer, we guide our design using floorplanning. By default, we set floorplanning constraints assuming a chip aspect ratio of 1 (square die) and 85% utilization (as discussed by Kuon and Rose [2007]). Note that total cell area is known after synthesis to standard cells, making it possible to define a die size with any given utilization ratio.

We found that floorplanning was mandatory to ensure that the physical layout of logic and routing tiles, in terms of ordering in the horizontal and vertical dimensions, matched with that assumed by VPR. Without this, Encounter produced layouts where, for example, logic blocks that VPR saw as adjacent were actually placed far apart in the layout.

Figure 7 is an example of how configuration cells will drift toward each other due to their connectivity and how two logic blocks that are intended to be adjacent to one another can get separated without floorplanning. For floorplanning the individual modules, we evenly divide up the chip and constrain our logic blocks and the connection MUXes connected to these logic blocks in the appropriate areas. On top of this grid, we overlay another grid to floorplan the switch MUXes in the appropriate areas. The Cadence placer allows one to control the rigidity of the floorplanning constraints, specifically whether cells are allow to enter/exit each floorplanning region. We set this to the most flexible scheme possible, where the floorplanning constraints are used as a guide to the placer, but cells may exit/enter the specified regions. All of the floorplanning `TCL` commands are automatically generated at the same time the Verilog description of the FPGA is generated.

Once the designs have been placed and routed, parasitic capacitances are extracted for use by PrimeTime to obtain accurate post-layout timing analysis. In addition, at this point, a GDSII file can be written that contains all mask information.

### 4.3. Timing Analysis

Synopsys PrimeTime is used for post-layout timing analysis of (1) specific paths within the implementation or (2) an application benchmark programmed on the FPGA. PrimeTime accepts as input the design, annotated with parasitic capacitance information, as well as a Synopsys design constraints (SDC) file. The SDC file specifies which timing paths should be ignored. For (1), we ignore all paths but the specific paths we wish to

Table I. FPGA Architecture Parameters

| Parameters | Values | Parameters | Values |
|---|---|---|---|
| FPGA dimensions | 20 × 20 | K, LUT size | 6 |
| N, number of LUTs/logic block | 10 | Crossbar connectivity | 50% |
| W, Channel width | 300 | L, Wire length | 4 (87%), 16 (13%) |
| Fs, Switch block connectivity | 3 | Switch block type | Wilton |
| $Fc_{in}$, Input connectivity | 0.055 | $Fc_{out}$, Output connectivity | 0.1 |

analyze (see the next section) and run timing analysis to obtain their delay. For (2), finding the critical path of an application benchmark implemented within the fabric, the process is more involved. Commercial FPGA vendors provide static timing analysis tools that analyze the performance for user designs implemented in their FPGAs, using delay models of the underlying fabric. To mimic the behavior of such tools for an application implemented within our synthesized fabric, we devised the following approach. During bitstream generation (Section 3.2), we have precise knowledge about which FPGA resources are used versus unused. For each unused resource, we automatically generate an SDC constraint to disable timing analysis through the resource. When PrimeTime is invoked to analyze performance of the FPGA device configured with the application bitstream, PrimeTime "sees" only those paths in the used part of the FPGA (which should be free of combinational loops, assuming well-designed circuits). The critical path reported by PrimeTime is then analogous to that reported by the timing analysis tools of commercial FPGA vendors. It is important to note that once the FPGA device has been synthesized, placed, and routed, timing analysis can be done for any application benchmark by providing PrimeTime with the bitstream and SDC file for that benchmark. Meaning, it is not necessary to synthesize, place, and route the FPGA device on an individual benchmark-by-benchmark basis.

A challenge we had to deal with regarding PrimeTime arose due to our bottom-up synthesis strategy and the delay model of the standard cells. PrimeTime reported warnings (RC-009) that in some cases, timing results may be inaccurate because cell drive resistance was too small in comparison with the impedance of the driven network. Recall that in the bottom-up synthesis style, in some cases Synopsys technology mapping must select cells of a certain size without global context/knowledge of the total RC load driven by such cells. This mainly occurred for large cells driving long interconnect wires, and we were able to eliminate all warnings through the buffer insertion discussed previously.

## 5. EXPERIMENTAL STUDY

Table I summarizes the parameters of the FPGA architecture [Brown et al. 2012] that we synthesized into commercial TSMC 65nm standard cells. The architecture is designed to resemble Altera's Stratix III FPGA, which is also fabricated in TSMC's 65nm process, allowing us to make a (roughly) apples-to-apples comparison. The architectural parameters are from a recently published Stratix IV architecture capture by Murray et al. [2013], where the authors attempted to model Stratix IV within VTR.[2] Our synthesized FPGA has dimensions of 20 × 20 logic blocks, with 10 fracturable LUTs per block. There are 300 routing tracks/channel, where 87% of tracks span 4 tiles and 13% span 16 tiles. $Fc_{in}$/$Fc_{out}$ refer to the fraction of adjacent tracks to which a logic block input/output pin may programmably connect. Within the logic block, the crossbar is 50% populated. We are using fracturable 6-LUTs with eight inputs, which

[2]Although Stratix IV is on a more advanced process than Stratix III, the soft logic block and routing architectures are similar.

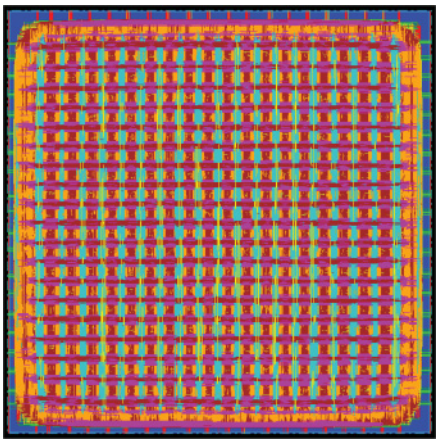

Fig. 8. Synthesized FPGA.

implies two shared inputs in dual-output mode, similar to the extensive architecture described in Luu [2014]. Such LUTs can implement any single function of up to six variables, or any two functions that together use no more than eight unique variables. We reinforce that although in this study we focus on a particular architecture that is comparable with Stratix III, our generator is able to automatically produce RTL for a wide variety of VTR-supported architectures.

We synthesized three variants of the architecture described in Section 4, area optimized, timing optimized and balanced, using Synopsys Design Compiler (syn_vH2013.03) and Cadence Encounter (v09.12). We used the wire-load model TSMC8K_Lowk_Conservative. For the area-optimized fabric, we directed Synopsys to minimize area and imposed no timing constraints. For the timing-optimized fabric, we conversely directed Synopsys to minimize delay and imposed no area constraints. For the balanced fabric, we took the midpoint of the achieved delays between the area and timing-optimized fabrics and set these as the target delays for Synopsys. As an additional study to the three synthesized fabrics, we also synthesized a timing-constrained fabric that used a hierarchical style Verilog MUX design optimized to use standard cell 4-to-1 MUX in the TSMC library, referred to as MUX4 optimized, as discussed in Section 4.1. To synthesize each of these fabrics, the runtime of Synopsys Design Compiler using the bottom-up method was 3 to 5 hours, depending on the fabric type. Cadence Encounter, when run with 85% utilization, required approximately 20 to 24 hours of runtime using two threads. Higher utilization rates resulted in much slower place and route. Last, timing analysis using Synopsys PrimeTime took around 40 minutes per benchmark due to the number of parasitic annotations and disabled paths. Figure 8 shows one of the synthesized FPGA fabric layouts.

In a first set of experiments, we examine the area and performance (of specific paths) of the synthesized FPGA and compare to analogous area and performance data for Stratix III. This first set of experiments is thus agnostic to any particular application design being implemented within the fabric—it is a fabric-to-fabric comparison. In a second set of experiments, we compare the performance of application benchmark designs implemented on our fabric to those same designs implemented on Stratix III.

We consider various combinational and sequential benchmarks from the MCNC benchmark suite [Yang 1991]. Since we are using the full VTR flow, we omitted some designs from the 20 largest MCNC benchmarks where VTR swept away unconnected nodes (as these circuits caused problems for our verification flow that relies on I/O

Table II. Area of Synthesized FPGA

| FPGA Fabric | No. of Std. Cells | Total Area ($mm^2$) | Tile Area ($mm^2$) | Ratio (Synthesized/ Stratix III) | Ratio (Synthesized/ Custom) |
|---|---|---|---|---|---|
| Area optimized | 3,577,520 | 12.65 | 0.0316 | 1.5 | 1.1 |
| Timing optimized | 7,521,616 | 25.72 | 0.0643 | 2.9 | 2.3 |
| Balanced | 5,298,588 | 16.89 | 0.0422 | 1.9 | 1.5 |
| MUX4 optimized | 6,527,880 | 21.82 | 0.0546 | 2.5 | 1.9 |
| Custom | 2,335,100 | 11.35 | 0.0284 | 1.3 | 1.0 |

matching). In addition to the MCNC circuits, we added a finite state machine (FSM) that detects a pattern and an adder connected to a shift register. These latter two circuits were used mainly for debugging purposes. We use the MCNC circuits in this initial study, as these can be simulated with random vectors and verified with the flow in Figure 4. Other benchmark suites, such as the VTR suite, contain DSP blocks and RAMs, and they are more challenging to simulate/verify owing to the circuits having reset/control inputs.

### 5.1. Area Analysis

We compare the tile area of our synthesized FPGA to Altera's Stratix III. The tile area of our FPGA was obtained by dividing total die area by the number of logic blocks ($20 \times 20 = 400$). Table II summarizes the tile area of the four synthesized architectures. The Stratix III LAB tile area is reported to be $0.0221mm^2$ by Wong et al. [2011]. The area-optimized fabric resulted in the smallest tile area of $0.0316mm^2$, which is $1.5\times$ bigger than Stratix III. As expected, the timing-optimized, balanced, and MUX4-optimized fabrics were larger: $2.9\times$, $1.9\times$, $2.5\times$ bigger than Stratix III, respectively. We were encouraged by the area of the synthesized fabrics, especially the area-optimized fabric, which is relatively close to Stratix III.

Several factors contribute to the area difference versus Stratix III. First, there are architectural differences. For example, our architecture does not support carry chains, nor are our MUXes fully decoded. Second, our implementation uses only those standard cells in the TSMC library. In commercial FPGAs, pass transistors or transmission gates are commonly used to implement MUXes and LUTs; however, we use full CMOS implementations of these primitives. Likewise, we are also using FFs for the configuration cells rather than SRAM cells (as we expect is done in a commercial device). Perhaps most importantly, the Stratix III LAB is custom laid out.

Delving further into the area results, Figure 9 shows the breakdown of area into logic, inter- and intra-logic block routing, and configuration for each fabric type. In the area-optimized design, configuration cells built of costly FFs in our case, occupy a large portion of the area: 42% of the total. It is likewise not surprising that routing comprises 50% of the fabric area, as we are using standard cell–based MUXes instead of pass transistor–style MUXes.

In the timing-optimized FPGA fabric, we observe that configuration cells are reduced to 21% of the total area. This is because the configuration area is kept constant by applying no timing constraints to the configuration cells (not performance critical). Routing area has increased to 67% of the area, and logic area increased to 13%. Remember that in the timing-optimized fabric, we inserted extra buffers on the inter-logic block wires. However, buffer area is not appreciable: 2% of the total. The MUX4-optimized FPGA fabric has a similar breakdown to the timing-optimized FPGA fabric, except the inter-logic block routing takes up less of the total area. The reduction comes from using the hierarchical style of Verilog to match available TSMC standard cell: 4-to-1 MUX.

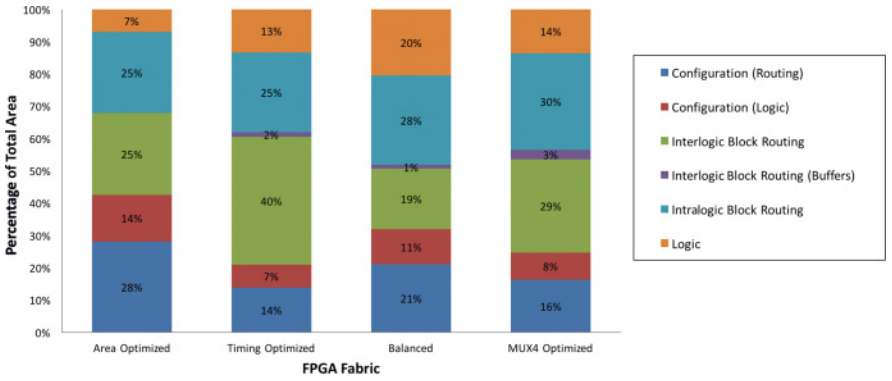


Fig. 9. Area breakdown.

Table III. Architecture Delay

| FPGA Fabric | $L_0$ (ns) | Ratio (Synthesized/ Stratix III) | $L_4$ (ns) | Ratio (Synthesized/ Stratix III) | $L_{16}$ (ns) | Ratio (Synthesized/ Stratix III) |
|---|---|---|---|---|---|---|
| Area optimized | 3.71 | 5.1 | 7.32 | 7.1 | 17.23 | 11.2 |
| Timing optimized | 1.79 | 2.5 | 2.90 | 2.8 | 4.92 | 3.2 |
| Balanced | 1.34 | 1.8 | 3.73 | 3.6 | 7.32 | 4.8 |
| MUX4 optimized | 1.62 | 2.2 | 3.56 | 3.7 | 5.03 | 3.3 |
| Custom | 4.36 | 6.0 | 7.50 | 7.3 | 17.97 | 11.7 |
| Stratix III | 0.73 | 1.0 | 1.03 | 1.0 | 1.54 | 1.0 |

In the balanced FPGA fabric, both timing and area constraints were applied; however, we give a more relaxed timing constraint to the routing circuitry to save area. This leads to logic taking up 31% (logic + config for logic) and routing taking up 69% (routing + config for routing + buffers) of the total area. Note that in the balanced fabric, we keep the LUT's timing constraint aggressive, as the LUT takes up a small portion of the total area.

### 5.2. Timing Analysis

We first examine the delay using Synopsys PrimeTime (pts_C2009.06) of commonly used paths in the synthesized fabrics and Stratix III (application-agnostic analysis). Specifically, we looked at the following three paths:

(1) $L_0$: FF → crossbar → LUT → FF (within a logic block)
(2) $L_4$: FF → length-4 wire → crossbar → LUT → FF (a path of length 4)
(3) $L_{16}$: FF → length-16 wire → crossbar → LUT → FF (a path of length 16).

Table III provides a summary of the average delay of these paths in six different areas of the FPGAs (in the four corners of the fabric and also on the middle of the left and right sides). In the synthesized fabrics, we manually selected the six paths by creating an SDC file that reports the delay for each. In doing so, we are assured that our analysis reflects the use of a length-4 or length-16 wire accordingly. For the Altera Stratix III delays, we use Altera's LogicLock feature to place two connected FFs 4 or 16 logic blocks away from one another and then use Altera's TimeQuest tool to ascertain the path delay. Since the routing architectures differ, Stratix III paths are not exactly matching the described paths; however, this is the closest estimation. Note that for the timing results in this article, we use the 1.1V slowest speedgrade delay model for Stratix III and compare to the 0.9V slowest-process-corner delay model for analysis of our fabric.

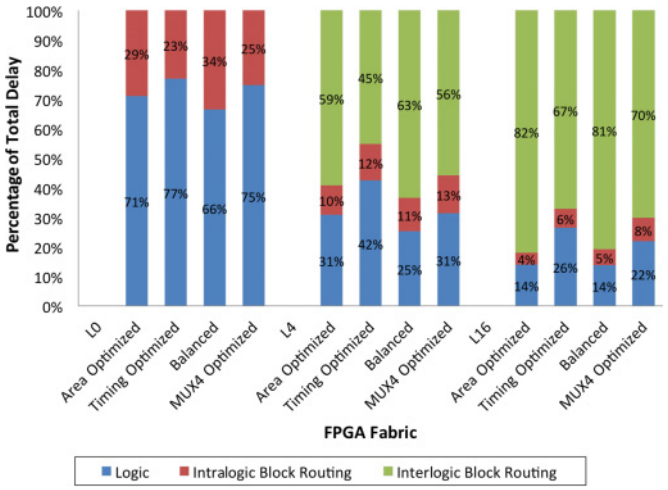


Fig. 10. Delay breakdown.

Comparing our timing-optimized implementation to Stratix III, the delay is 2.5× to 3.2× slower, with the gap being larger for longer wire lengths. The balanced fabric is 1.8× to 4.8× slower, the area-optimized fabric is 5.1× to 11.2× slower, and the MUX4-optimized fabric is 2.2× to 3.7× slower than Stratix III. A portion of the performance gap is a consequence of the 0.2V higher supply voltage used for the commercial device compared to the synthesized fabric. We expect that Altera has incorporated features into the design/layout of their device to ensure reliability at the higher supply. In addition, we believe the reason the delay gap versus Stratix III grows with wire length is related to the difficulty in handling long wires in the ASIC toolflow.

The relative results between the synthesized fabrics are as expected: the area-optimized fabric overall is slower than the balanced fabric, the balanced fabric is slower than the MUX4-optimized fabric, and MUX4-optimized fabric is slower than the timing-optimized fabric, except for the $L_0$ path. $L_0$ reflects timing within a logic block; the inter-logic block routing MUXes are not included in the delay. The slightly lower delay in the balanced fabric and MUX4-optimized fabric may be due to the heuristic nature of ASIC mapping, placement, and routing tools. It may also be because in our fabric (unlike Stratix III), one MUX drives the output of a logic block to both feedback and inter-logic block routing paths. That output MUX is timing optimized in our balanced fabric implementation, yet it sees to be a smaller load than in the timing-optimized fabric implementation (owing to smaller gates and less load capacitance in the balanced implementation). Similar to the area analysis, Figure 10 shows a delay breakdown for the three types of paths for all architectures. The $L_4$ and $L_{16}$ path delays are dominated by routing delay. This confirms that we need to use optimizations such as buffering to reduce the routing delay.

When we compare the timing-optimized fabric and the MUX4-optimized fabric (also optimized for timing), we observe that the differences in their delays are small, yet the area consumed by the MUX4-optimized fabric is considerably less than that consumed by the timing-optimized fabric. Recall that the difference between these fabrics relates to the MUX implementations: in the MUX4-optimized case, the MUXes were specified in a way that allowed Design Compiler to infer one of the available MUX standard cells. Overall, the MUX4-optimized fabric appears to offer an attractive area/performance trade-off.

Table IV. Critical Path Delay (ns) of Designs on FPGA

| Benchmark Circuits | Area Optimized | Timing Optimized | Balanced | MUX4 Optimized | Custom | Stratix III |
|---|---|---|---|---|---|---|
| alu4 | 67.80 | 22.29 | 32.47 | 27.14 | 65.76 | 5.29 |
| apex4 | 74.62 | 23.06 | 36.08 | 29.50 | 71.50 | 5.27 |
| des | 68.65 | 21.58 | 31.26 | 27.32 | 66.76 | 6.70 |
| ex1010 | 103.59 | 31.36 | 48.09 | 38.15 | 97.95 | 7.25 |
| ex5p | 68.68 | 21.86 | 31.54 | 27.36 | 66.49 | 5.46 |
| misex3 | 74.78 | 22.59 | 34.32 | 29.76 | 73.74 | 5.28 |
| pdc | 111.02 | 33.72 | 51.66 | 42.55 | 106.15 | 7.21 |
| seq | 69.79 | 22.66 | 32.66 | 27.54 | 67.46 | 5.74 |
| spla | 112.73 | 34.92 | 51.30 | 41.39 | 101.20 | 6.65 |
| diffeq | 69.42 | 23.28 | 29.74 | 27.77 | 69.98 | 4.39 |
| dsip | 35.94 | 11.71 | 17.75 | 14.20 | 33.06 | 5.92 |
| elliptic | 103.42 | 32.45 | 44.02 | 39.22 | 102.67 | 6.91 |
| frisc | 118.72 | 38.06 | 52.37 | 47.17 | 119.04 | 7.87 |
| tseng | 60.43 | 20.05 | 25.92 | 24.42 | 62.25 | 4.52 |
| addshift16 | 37.01 | 13.43 | 16.61 | 15.86 | 39.88 | 4.31 |
| fsm | 5.75 | 1.71 | 2.85 | 2.72 | 5.19 | 1.11 |
| Geo. mean | 63.13 | 20.10 | 28.97 | 25.13 | 61.30 | 5.25 |
| Ratio (synthesized/ Stratix III) | 12.0 | 3.8 | 5.5 | 4.8 | 11.7 | 1.0 |

In the second part of the performance study, we looked at how benchmark circuits perform on the synthesized FPGA versus Stratix III. The same Verilog file is passed to each tool (VTR and Quartus II) for implementing the circuits on the FPGAs. We use the SDC file generated with the bitstream to "program" our synthesized FPGA, as discussed in Section 4.3. Note that the results of this experiment are not solely reflective of the fabric speed but also of the differences in architectures and in the CAD tools supporting the architectures: open-source VTR versus Altera's Quartus II. Table IV lists the critical path delays reported by the tools. The reported critical path delays do not include clock skew nor I/O cell delays (only core logic and routing).

In combinational designs (top part of the table), both designs were given input-to-output delay constraints. On average, there is a nearly 3.8× increase in delay between the timing-optimized and Stratix III FPGAs (see the geometric mean row at the bottom of the table). The delay gap between the two FPGAs increased from our preceding architecture delay study, likely due to the weaknesses of the open-source VTR flow versus Quartus II. In the sequential designs (bottom part of the table), the critical path delays reported include register-to-register and I/O paths. Most circuits show a similar increase in delay as seen with the combinational designs; however, the *dsip* and *fsm* benchmarks showed smaller increases of 1.5× to 2×. The critical paths of these circuits have fewer logic levels compared to the other designs.

It is worthwhile to reinforce that one of the key advantages of a synthesizable FPGA fabric is that it permits the type of exploration done here, namely the ability to realize fabrics with different area/delay trade-offs from a single RTL source simply by changing constraints provided to the ASIC tools. Such an exploration is highly costly if manual layout is required for each fabric.

## 6. ADDING A CUSTOM CELL

Standard cell libraries contain a collection of cells that can be combined to realize any function described by the designer in RTL. These cells are not optimized for any particular application, so it is expected that there would be an area and delay overhead

Table V. Detailed MUX Usage Breakdown

| MUX Size ($x$-to-1) | Area ($u$m$^2$) | Instances (#) | Routing Area (%) |
|---|---|---|---|
| 2 | 5.04 | 840 | 0 |
| 3 | 9.36 | 11,760 | 3 |
| 10 | 30.96 | 1,320 | 1 |
| 11 | 39.96 | 1,680 | 2 |
| 12 | 34.20 | 2,160 | 2 |
| 13 | 41.04 | 8,844 | 11 |
| 14 | 42.84 | 19,696 | 26 |
| 15 | 40.32 | 19,696 | 25 |
| 16 | 40.68 | 22,232 | 28 |

connected with the use of standard cells for a specific application. For example, most of the cells use the fully complementary CMOS style of circuit design, where each gate is constructed using an NMOS pull-down network and a PMOS pull-up network, each having the same number of transistors.

The FPGA routing network contains many MUXes to support flexible connectivity between logic elements. Building MUXes out of fully complementary CMOS standard cells is not particularly area efficient, and consequently the synthesized FPGA fabric area is dominated by routing. In Section 5, the results showed that routing consumed more than half of the synthesized FPGA area. In this section, we consider using a custom cell for more efficient implementation of wide MUXes rather than using standard cells as done in prior sections.

Table V lists MUX sizes used in the area-optimized standard cell fabric described in prior sections, the number of instances of each size, and the percentage of total routing area consumed. When we look at the breakdown of different MUX sizes, it is clear that we will benefit the most from adding a 16-to-1 MUX with area less than $40u\text{m}^2$ since the majority of cells are between 14-to-1 and 16-to-1 MUX. The added 16-to-1 MUX can also be used to implement smaller MUXes (e.g., 14-to-1) by tieing unused data inputs to constants.

Although we decided to customize the 16-to-1 MUX using a pass transistor implementation instead of full CMOS gates, it is worth mentioning that this choice is specific to the Altera Stratix–like architecture that we are synthesizing. Architectures having different architectural parameters will lead to different sizes of MUXes being used. In addition, the MUX usage breakdown only reflects inter-logic block routing. Intra-logic block routing uses large MUXes as well. Ideally, for FPGAs, we believe that it would be desirable to have custom cells available for a variety of MUX sizes. For this study, however, we consider solely adding a single custom cell, owing to the time-intensive custom cell design and manual layout.

### 6.1. Cell Design and Integration

To design a 16-to-1 MUX, we first consider its encoding. A MUX can be fully encoded or one-hot encoded. A fully encoded MUX will have fewer select signals (configuration cells) but will have several levels of logic, whereas a one-hot encoded MUX will have more select signals but will only have one level of logic. To compromise, we divided the 16-to-1 MUX into two levels: four 4-to-1 MUXes in a first stage, then another 4-to-1 MUX to select from the first level. The first level of MUXes use one-hot encoding, and the second level of MUX is fully encoded. In all, six configuration cells are used, and there are three levels of pass transistors. Figure 11 is the transistor-level circuit of the 16-to-1 MUX that we designed.

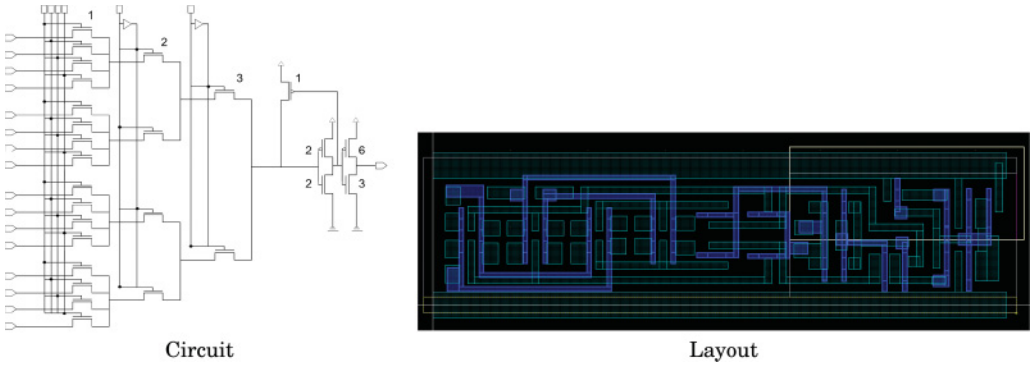


Fig. 11. MUX16 transistor-level circuit and custom layout.

We considered whether to implement the MUX using transmission gates or NMOS pass transistors. Although Chiasson and Betz [2013] suggested using transmission gates when there is no gate boosting, we found that pass transistors resulted in a better layout when limited by the standard cell pitch. In particular, to allow the cell to be used alongside the TSMC standard cell library, we must follow the layout rules of the library, including the cell height (pitch). Since we are using the NMOS pass transistor style of circuit design, we include a small pull-up PMOS at the end of the NMOS tree to restore degraded $V_{DD}$ from the NMOS-based MUX. It is important to make sure that the PMOS is small enough to turn it off on a falling transition; otherwise, the node could be stuck at $V_{DD}$. In the end, we chose to use the smallest possible PMOS width in the TSMC process.

To size all of the transistors, Hspice was used iteratively. We sized the transistors to achieve robust functionality and lower delay than that achieved with using the generic standard cell library, with extra margin added to accommodate parasitic capacitance arising from layout. The iterative approach resulted in the use of minimum-width transistors for the first level of NMOS, 2× the minimum width for the second level, and 3× the minimum width for the third level. Using the same method, the two inverters forming the output buffer were sized as 2× minimum width (PMOS and NMOS are sized the same since the turning on of NMOS is slower due to the pull-up network) in the first stage and 3× minimum width in the second stage to drive a load of 50fF in the worst case.

### 6.2. Layout

The layout was done using the TSMC 65nm kit in Cadence Virtuoso, adhering to the pitch, metal, and pin location requirements of the TSMC cell library. Since the standard cells are handled automatically by the place and route tools, there are layout requirements so that neighboring cells can be abutted (for power and ground distribution) and provide enough space for metal to reach the pins. Since we have 16 inputs to route to the NMOS pass transistors, the cell was both metal and transistor limited. Figure 11 is the resulting layout where the left two thirds of the layout contains all of the NMOS pass transistors and the remaining one third of the layout is where the pull-up PMOS and all necessary inverters are placed. Our design is very long horizontally due to the vertical pitch requirements of the library. One can imagine that a transmission gate–based design would have been even longer (hence larger) given the required pitch. Note that some layers are omitted in the figure, owing to confidentiality requirements of TSMC. Layout verification was done using Mentor's Calibre LVS (layout vs. schematic) and DRC (design-rule check) tool. Mentor's Calibre PEX tool was used to extract parasitic

capacitance. The 16-to-1 MUX cell was 13.68$u$m$^2$ compared to the 40$u$m$^2$ when using the generic standard cell library, which is 66% smaller.

### 6.3. Cell Integration

We characterized the cell to create a library element for Synopsys Design Compiler and a library exchange format (LEF) file for Cadence Encounter. Synopsys Design Compiler and PrimeTime use Liberty files, `.lib`, to map circuits into standard cells. The Liberty file contains cell functionality information and a table of delays with input transition times and output loads as parameters. We are using the TSMC library based on the worst-case conditions (0.9V), so we simulated our post-layout Hspice netlist (generated by Mentor Calibre PEX) with the same $V_{DD}$ and gathered data for a list of input transition times and output loads similar to those used for the TSMC cells. We used HspiceD with scripts to simulate our cell. For each input pin, a script was generated that would sweep varying input transition times and output loads. The script measured the rise/fall transition times and rise/fall delays. The script included our MUX netlist with its parasitics annotated. The characterization results were then turned into the liberty file format using another script.

When we provided Synopsys Design Compiler with the updated library, it was unable to automatically infer usage of the new MUX for any style of input Verilog that we tried. We simply could not "make" Design Compiler choose to use the cell. Therefore, we modified our Verilog generator to force the usage of our cell by manually instantiating it in the Verilog. Concretely, as we generate Verilog, if a MUX of size greater than 8 is needed in the FPGA fabric, we instantiate the new 16-to-1 MUX cell. For example, for a 17-to-1 MUX, we would instantiate the new MUX as well as a 2-to-1 MUX. Since we are using a partially encoded MUX, we also had to modify our bitstream generator to match the underlying MUX implementation for functional correctness. We used Cadence Encounter as before to place and route the FPGA fabrics containing the new cell.

Last, the same Liberty file was provided to Synopsys PrimeTime for timing analysis. However, we found that our cell characterization resulted in nonincreasing segments in the piecewise linear delay with respect to input slew rate and output load. The TSMC library into which we incorporated our cell used a LUT approach to estimating delay and expected an increase in delay with an increase in input transition time and output load; however, our characterization showed a trend that did not follow the expectations, possibly due to the presence of the level-restoring buffer on the MUX output (having the weak pull-up), resulting in warnings in PrimeTime. To verify that it was not the layout of the cell that produced the warnings, we generated the same library element using the prelayout Hspice netlist and observed the same warnings. However, despite the warnings, the delay results appear consistent with expectations (see the following).

### 6.4. Results

We synthesized the FPGA architecture described in Section 5 using the new 16-to-1 MUX cell. Table II presents the tile area of all the FPGA fabrics studied previously, as well as the area-optimized fabric using the custom cell (referred to as custom fabric). We observe an improvement of 11% relative to the area-optimized fabric synthesized with the "vanilla" standard cell library. We were encouraged that the addition of a single cell could achieve such a significant impact to overall area. This follows closely with previous work [Phillips and Hauck 2002], which reported an area savings of 18.9% to 9% by adding four tactical cells. Aken'Ova et al. [2005] reported an area savings of 42% with six additional cells, where the significant area savings may be due to the addition of the SRAM cell.

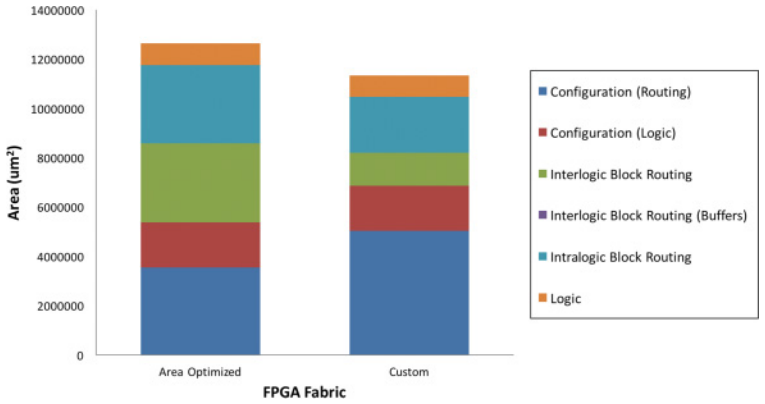


Fig. 12. Area breakdown of an area-optimized FPGA and an area-optimized FPGA with a custom cell.

Figure 12 compares the area-optimized FPGA to the FPGA incorporating the custom cell. One can observe an increase in configuration area of routing due to our partially decoded MUX design but a decrease in overall routing area. As expected, the configuration for logic and logic area are unaffected, as the custom cell was not used within the LUTs—this is left to future work.

The delays of a fabric incorporating the custom cell are presented in Tables III and IV. The architectural delays of the area-optimized fabric using the custom cell are slightly greater than the area-optimized fabric using solely the TSMC library, with the delay ratio being highest in the L0 case when the presence of the custom cell represents a more significant portion of the total delay (due to the absence of inter-logic block wire delay). On the other hand, the delays of critical paths in the benchmarks are slightly smaller when the custom cells are used. Overall, the fabric with the custom cells appears to provide roughly similar performance as the area-optimized fabric considered previously, yet it does so with less area. The delay discrepancies may be due to the Synopsys PrimeTime warnings discussed earlier. The use of buffers in conjunction with the custom cells would likely be beneficial to the performance of fabrics incorporating them. We leave as future work the incorporation of custom cells into other fabrics, such as the timing-optimized and balanced variants.

## 7. CONCLUSIONS AND FUTURE WORK

In this work, we propose to automatically generate synthesizable FPGA fabrics within the open-source FPGA CAD tool, VTR. The fabrics we generate are then synthesized, placed, and routed using a standard ASIC design flow into a commercial standard cell library. We synthesized four variants of an FPGA fabric (modeled on Altera's Stratix III) into 65nm TSMC standard cells: timing-optimized, area-optimized, balanced, and timing-optimized implementation with a hierarchical MUX design. We compared the tile area of our smallest FPGA fabric (area optimized) to Altera's Stratix III and found that our fabric used 1.5× more area. Our timing-optimized fabric required 3× more area than Stratix III. With respect to performance, the critical paths of designs implemented in our timing-optimized fabric are nearly 3.8× longer on average than in Stratix III; however, in some benchmarks, the delay gap was as low as 1.5×. Overall, we are encouraged by the silicon area and performance of our fabric relative to that of Altera, especially considering that Stratix III is custom laid out and undoubtedly highly optimized. We then explored augmenting a standard cell library with an FPGA-specific cell. As a proof of concept, we added a 16-to-1 MUX cell

into the generic TSMC 65nm standard cell library. The layout of the cell was done in Cadence Virtuoso and characterized for integration with the rest of the standard cells. The cell was 66% smaller than a generic standard cell version of a 16-to-1 MUX. This resulted in an 11% tile area reduction compared to the area-optimized fabric with similar delays as the area-optimized fabric. To the best of our knowledge, this work represents the first comparison of a standard cell FPGA implementation to a commercial FPGA. The proposed VTR-based synthesizable FPGA generator opens the door to actual silicon implementation of FPGAs targetable by an established CAD tool.

In the future, the architecture and bitstream generation can benefit from accepting all architectures supported by VTR, including those with DSP blocks and memories. This includes the notion of modes being supported, where a designer can specify a DSP block that can function in two different modes (e.g., as an $18 \times 18$ bit multiplier or two $9 \times 9$ bit multipliers). The generator will have to automatically infer an architecture that can perform all modes described. Further work is also needed to support designs with multiple clocks and with configurable I/Os organized into banks that can programmably operate with different signaling standards (as in commercial devices). In terms of the ASIC design flow, clock tree synthesis should be explored. In our designs, we synthesized the clock with the rest of the logic signals and ignored clock skew. To realize low clock skew and higher performance, clock trees are required. In addition, power consumption should also be assessed. With this information, the trade-offs between area, delay, and power can be understood. Finally, we plan to "close the loop" in the VTR toolflow by back-annotating component-wise delays from the ASIC standard cell layout into the VTR architectural model, thereby allowing place and route of an application in VTR to proceed in a manner cognizant of delays within the synthesized fabric.

## REFERENCES


Victor Aken'Ova. 2005. *Bridging the Gap Between Soft and Hard eFPGA Design*. Master's Thesis. University of British Columbia.

Victor Aken'Ova, Guy Lemieux, and Resve Saleh. 2005. An improved "soft" eFPGA design and implementation strategy. In *Proceedings of the 2005 IEEE CICC Conference (CICC'05)*. 179–182.

Victor Aken'Ova and Resve Saleh. 2006. A "soft++" eFPGA physical design approach with case studies in 180nm and 90nm. In *Proceedings of the 2006 ISVLSI Conference (ISVLSI'06)*. IEEE, Los Alamitos, CA.

Altera. 2015. *Stratix III ALM Logic Structure's 8-Input Fracturable LUT*. Technical Report. Altera Corporation. https://www.altera.com/products/fpga/features/st3-logic-structure.html.

Vaughn Betz, Jonathan Rose, and Alexander Marquardt. 1999. *Architecture and CAD for Deep-Submicron FPGAs*. Kluwer.

Vaughn Betz and Jonathan Rose. 1997. VPR: A new packing, placement and routing tool for FPGA research. In *Proceedings of the 1997 FPL Conference (FPL'97)*. 213–222.

S. D. Brown, R. J. Francis, J. Rose, and Z. G. Vranesic. 2012. *Field-Programmable Gate Arrays*. Springer International Series in Engineering and Computer Science, Vol. 180. Springer Science & Business Media.

S. Chaudhuri, J.-L. Danger, and S. Guilley. 2007. Efficient modeling and floorplanning of embedded-FPGA fabric. In *Proceedings of the 2007 FPL Conference (FPL'07)*. IEEE, Los Alamitos, CA, 665–669.

S. Chaudhuri, S. Guilley, F. Flament, P. Hoogvorst, and J.-L. Danger. 2008. An 8x8 run-time reconfigurable FPGA embedded in a SoC. In *Proceedings of the 2008 DAC Conference (DAC'08)*. 120–125.

Charles Chiasson and Vaughn Betz. 2013. Should FPGAs abandon the pass-gate? In *Proceedings of the 2013 FPL Conference (FPL'13)*. IEEE, Los Alamitos, CA, 1–8.

W. Deng, D. Yang, T. Ueno, T. Siriburanon, S. Kondo, K. Okada, and A. Matsuzawa. 2015. A fully synthesizable all-digital PLL with interpolative phase coupled oscillator, current-output DAC, and fine-resolution digital varactor using gated edge injection technique. *IEEE Journal of Solid-State Circuits* 50, 1, 68–80.

C. Ebeling, D. Cronquist, and P. Franklin. 1996. RaPiD reconfigurable pipelined datapath. In *Field-Programmable Logic Smart Applications, New Paradigms and Compilers*. Lecture Notes in Computer Science, Vol. 1142. Springer, 126–135.

E. Fluhr, J. Friedrich, D. M. Dreps, and M. M. Ziegler. 2014. POWER8: A 12-core server-class processor in 22nm SOI with 7.6tb/s off-chip bandwidth. In *Proceedings of the 2014 ISSCC Conference (ISSCC'14)*. IEEE, Los Alamitos, CA, 96–97.

N. Kafafi, K. Bozman, and S. J. E. Wilton. 2003. architectures and algorithms for synthesizable embedded programmable logic cores. In *Proceedings of the 2003 FPGA Conference (FPGA'03)*. ACM, New York, NY, 3–11.

Jin Hee Kim and Jason H Anderson. 2015. Synthesizable FPGA fabrics targetable by the Verilog-to-Routing (VTR) CAD flow. In *Proceedings of the 2015 FPL Conference (FPL'15)*. IEEE, Los Alamitos, CA, 1–8.

Ian Kuon, Aaron Egier, and Jonathan Rose. 2005. Design, layout and verification of an FPGA using automated tools. In *Proceedings of the 2005 FPGA Conference (FPGA'05)*. ACM, New York, NY, 215–226.

Ian Kuon and Jonathan Rose. 2007. Measuring the gap between FPGAs and ASICs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 26, 2, 203–215.

Hao Jun Liu. 2014. *Archipelago—An Open Source FPGA with Toolflow Support*. Master's Thesis. University of California at Berkeley.

J. Luu. 2014. *Architecture-Aware Packing and CAD Infrastructure for Field-Programmable Gate Arrays*. Ph.D. Dissertation. University of Toronto.

K. E. Murray, S. Whitty, S. Liu, J. Luu, and V. Betz. 2013. Titan: Enabling large and complex benchmarks in academic CAD. In *Proceedings of the 2013 FPL Conference (FPL'13)*. IEEE, Los Alamitos, CA.

S. Phillips and S. Hauck. 2002. Automatic layout of domain-specific reconfigurable subsystems for system-on-a-chip. In *Proceedings of the 2002 FPGA Conference (FPGA'02)*. ACM, Los Alamitos, CA, 165–173.

J. Rose, J. Luu, C. W. Yu, O. Densmore, J. Goeders, A. Somerville, K. B. Kent, P. Jamieson, and J. Anderson. 2012. The VTR project: Architecture and CAD for FPGAs from Verilog to Routing. In *Proceedings of the 2012 FPGA Conference (FPGA'12)*. ACM, New York, NY, 77–86.

S. Wilton, C. H. Ho, B. Quinton, P. H. W. Leong, and W. Luk. 2007. A synthesizable datapath-oriented embedded FPGA fabric. In *Proceedings of the 2007 FPGA Conference (FPGA'07)*. ACM, New York, NY, 33–41.

H. Wong, V. Betz, and J. Rose. 2011. Comparing FPGA vs. custom CMOS and the impact on processor microarchitecture. In *Proceedings of the 2011 FPGA Conference (FPGA'11)*. ACM, New York, NY, 5–14.

S. Yang. 1991. *Logic Synthesis and Optimization Benchmarks User Guide Version 3.0*. Microelectronics Center of North Carolina.

Grace Zgheib, Manana Lortkipanidze, Muhsen Owaida, David Novo, and Paolo Ienne. 2016. FPRESSO: Enabling express transistor-level exploration of FPGA architectures. In *Proceedings of the 2016 FPGA Conference (FPGA'16)*.

Received April 2016; revised October 2016; accepted November 2016