How to shrink my FPGAs

Optimizing Tile Interfaces and the Configuration Logic in FABulous FPGA Fabrics

King Lok Chung, Nguyen Dao, Jing Yu and Dirk Koch

king.chung@student.manchester.ac.uk, [nguyen.dao, jing.yu, dirk.koch]@manchester.ac.uk

Computer Science, The University of Manchester, UK

ABSTRACT

Commercial FPGAs from major vendors are extensively optimized, and fabrics use many hand-crafted custom cells, including switch matrix multiplexers and configuration memory cells. The physical design optimizations commonly improve area, latency (=speed), and power consumption together. This paper is dedicated to improving the physical implementation of FPGA tiles and the configuration storage in SRAM FPGAs. This paper proposes to remap configuration bits and interface wires to implement tightly packed tiles. Using the FABulous FPGA framework, we show that our optimizations are virtually for free but can save over 20% in area and improve latency at the same time. We will evaluate our approach in different scenarios by changing the available metal layers or the requested channel capacity. Our optimizations consider all tiles and we propose a flow that resolves dependencies between the CLBs and other tiles. Moreover, we will show that frame-based reconfiguration is, in almost all cases, better than shift register configuration.

CCS CONCEPTS

• **Hardware** → *Programmable logic elements*;

KEYWORDS

FPGA, reconfigurable computing, FPGA virtualisation, partial re-configuration

ACM Reference Format:

King Lok Chung, Nguyen Dao, Jing Yu and Dirk Koch. 2022. How to shrink my FPGAs — Optimizing Tile Interfaces and the Configuration Logic in FABulous FPGA Fabrics. In *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '22)*, February 27–March 1, 2022, Virtual Event, USA. ACM, New York, NY, USA, 11 pages. <https://doi.org/10.1145/3490422.3502371>

1 INTRODUCTION

The major FPGA vendors have large design teams with experienced engineers for designing extensively optimized FPGA fabrics which deliver high quality of results in terms of density (=area), latency,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

FPGA '22, February 27–March 1, 2022, Virtual Event, CA, USA

© 2022 Copyright is held by the owner/author(s). Publication rights licensed to ACM.

ACM 978-1-4503-9149-8/22/02...\$15.00

<https://doi.org/10.1145/3490422.3502371>

Figure 1: FABulous [20] integrates academic and industry tools for ASIC generation, emulation (not shown), and eFPGA configuration bitstream generation.

and power for FPGAs. The optimizations include a complex interplay between the architecture and the physical implementation to maximize customer utility.

To illustrate the complexity of designing FPGAs (or embedded FPGAs – eFPGAs), Figure 1 shows the FABulous ecosystem [20] that itself integrates several industry and open-source tools to generate the physical ASIC implementation of an FPGA fabric as well as a flow to compile user circuits into bitstreams. FABulous provides flows for physically implementing an FPGA fabric on a chip and for parametrizing the SymbiFlow [36] open-source tool suite that is integrating Yosys [35], nextpnr [29], ABC [3] and VPR [4] that is used to compile Verilog to bitstreams for the FPGA fabrics. Moreover, FABulous integrates BitMan [28] for the bitstream assembly and the Verilator [34] for simulation. The framework is very versatile and, for instance, for the ASIC implementation path, it can (besides some industry tools) use the OpenLane [10] tool suite, which together with the Skywater 130nm process, provides an entirely open design environment including the ASIC process design toolkit (PDK). The open Skywater 130nm process [30] is currently very popular in the open-source hardware community thanks to Google’s OpenShuttle program that resulted in about 80 MPW projects over the last six months [11].

In this paper, we are using FABulous to generate FPGA tiles and Yosys (vers. 0.9+4052) for logic synthesis. Only for the physical implementation of the FPGA tiles, we will use an industry tool (Cadence Innovus 15.2) as it still provides slightly better and more relevant results than OpenLane in its present version.

This paper deals mostly with the physical optimization of tiles consisting of primitives and routing resources that form the basic building blocks for stitching together an FPGA fabric. This paper is built on our FABulous eFPGA framework that aims at providing high-quality FPGA fabrics as close as possible to what major FPGA vendors can achieve. To assist this, we are adding physical optimizations for improved density, latency, and possibly power. The paper contributions include:

- **Configuration cell remapping.** In Section 4, we will modify the relative position of configuration memory cells in both the fabric netlist and the physical implementation with the goal to minimize congestion inside the logic tiles of an FPGA fabric.
- **Tile interface layout optimizations.** In Section 5, we will optimize the tile interface by defining the exact physical connections of a logic tile with the aim to shrink its area.
- **Heterogeneous tile optimizations.** In Section 6, we will optimize DSP and register file tiles.
- **A Study on frame-based versus shift-register configuration modes** in Section 8 examines the different cost factors and features for providing an optimal infrastructure to write configuration data into the fabric.

This paper is a contribution to the open-hardware movement to promote reconfigurable technology. The here presented optimizations are integrated into the FABulous project, available under: <https://github.com/FPGA-Research-Manchester/FABulous>

2 BACKGROUND AND RELATED WORK

As identified in the seminal textbook *Architecture and CAD for Deep-Submicron FPGAs* by Betz and Rose [5], there exist three major optimization loops in an FPGA ecosystem (see also Figure 1):

- (1) *Physical optimization of the FPGA ASIC.* This includes optimizations like full custom cell design for configuration storage and multiplexing, buffer and wire sizing, and exploring many physical constraints. The goal here is to provide the best performance and power at the lowest cost (=area).
- (2) *Fabric architecture optimization.* This includes the optimization of primitives (e.g., LUTs) and the routing fabric consisting of switch multiplexers and wires. Here, the goal is to design a fabric that is suited to implement user circuits.
- (3) *User design optimization,* which is tweaking user circuits to make best use out of a given fabric. This is the domain of a typical FPGA application engineer.

These optimization loops do interact heavily with each other. For instance, different fabrics will impact the physical ASIC implementation and the ability of the FPGA CAD tools, and the user circuit optimization effort relates to the number of FPGA resources (and ultimately the implementation cost).

This paper is mostly focusing on physical implementation aspects of FPGA fabrics. This research has a long history. For example, custom cells were deployed in [1, 20, 31, 32] to improved design

	type	height	width	area	utilization
std cell	CLB	219 μ m	219 μ m	47,961	81.8%
	REG	219 μ m	214 μ m	46,866	84.1%
	DSP	443 μ m	185 μ m	81,955	80.9%
custom mux cell	CLB	215 μ m	215 μ m	46,225	60.7%
	REG	215 μ m	217 μ m	46,655	64.3%
	DSP	435 μ m	188 μ m	81,780	56.7%

Table 1: Comparing smallest quadratic CLB tiles using standard cell and custom multiplexers designs.

metrics, or the work in [15] discussed the effect of floorplanning on FPGA performance (latency). [7] discusses if pass-gates or transmission gates are better suited for building custom multiplexers for FPGAs, and customizing and optimizing the architecture of FPGAs are studied deeply in [24, 27].

In this work, we will in particular investigate if a remapping of configuration bits can improve area, which is not covered in literature so far. Furthermore, we are tailoring the optimization of interfaces to the specific needs to build FPGA tiles.

3 CUSTOM CELLS

Most academic works (e.g., [14, 17, 18, 26]) and even some industry solutions [8] do only use standard cell designs for their FPGA fabrics. Standard cell fabrics are easier to port to different process nodes, but usually come with poorer performance and density. However, because FPGAs require a large number of multiplexers and corresponding configuration memory cells, it is promising to optimize these two components. In the Soft++ eFPGA work [1], the authors refer to these as *tactical cells*, meaning that just optimizing a small number of cells can gain substantial area savings (e.g., 25% in [1]).

The OpenFPGA framework [31] provides for some processes custom multiplexers (Mux4, Mux2) and dedicated configuration D-flip-flops. FABulous provides custom multiplexers for TSMC 180nm and Skywater 130nm processes but currently has no optimized configuration cells. The reason for this is that FABulous can use frame-based reconfiguration where the configuration data is stored in latches rather than expensive D-flip-flops as needed for shift-register configuration in OpenFPGA. For instance, the custom configuration cells provided with OpenFPGA are not smaller than standard cell latches. This means that FABulous uses about the same area for configuration storage per cell than openFPGA, without the need (and risk) to design a custom configuration cell.

To examine the impact of custom cells in FABulous, we took the CLB logic, register file tiles (REG) and the DSP tiles from the FABulousSky git repository [32], and we implemented them in the smallest possible bounding boxes. For this, we used the provided netlists and we started optimizing the CLB logic tiles first by trying a number of different square bounding boxes to implement a CLB tile. The numbers reported are for the smallest successful implementation. Then we took the height of that CLB tile to implement the register file (REG) and multiplier (DSP) tiles. Note that the DSP tiles are double the height of a CLB tile plus a 5 μ m gap that we have

used between stitched tiles inside a fabric. For the REG and DSP tiles, we tried different widths to explore the smallest bounding box that will result in a successful implementation. We repeated the process, but this time used the custom multiplexers as provided by FABulous.

The benchmark CLB provides 8 Lattice-like LUT-4 with a routing fabric that was adopted from a Spartan-3 routing fabric. The CLB resources include:

resources	Mux4	Mux2	DFP	latch
count	376	46	8	586

The results are listed in Table 1. For the used Skywater 130 process, FABulous provides only a custom Mux4 which is $6.44 \times 2.72\mu\text{m}^2$ versus $12.42 \times 2.72\mu\text{m}^2$ for the standard cell Mux4¹. In the standard cell design, all Mux4 contributed to the area $376 \times 12.42 \times 2.72\mu\text{m}^2 = 12,749\mu\text{m}^2$ or 26.6% of the CLB area. This area can be shrunk by using custom multiplexers that save 48% of the Mux4 area. Ergo, the savings could be up to $6138\mu\text{m}^2$. However, our experiments showed that we could only save about one-quarter of that ($1736\mu\text{m}^2$), and that the use of custom multiplexers is decreasing utilization instead. The trend that we found for the CLB tile also holds for the register file (REG) and DSP tile: replacing the standard cell multiplexers with custom counterparts is only marginally improving area while the core utilization of the tile is going down. Note that the tile implementation leaves a fence of $5\mu\text{m}$ around the border of the bounding box unused in order to fight congestion at the tile border that arises from the strict tile bounding box definition. This causes some distortion to the utilization numbers that are listed in Table 1 (denoting the average utilization over the full tile area).

These experiments indicate that the physical implementation is bound by wire congestion rather than the available area for mapping the cells on the chip. This particular experiment shows that providing an optimized custom multiplexer is barely showing enough improvement to justify the effort. We will therefore target optimizations that fight congestion in the following in order to take full advantage of custom multiplexers.

4 CONFIGURATION CELL PLACEMENT

The netlist comprising the primitives and switch matrix multiplexers of a tile defines a place and route problem. In addition, a tile usually requires hundreds of configuration bits (about 500-1000 bits for a typical CLB) that form their own netlist and corresponding place and route problem. In frame-based reconfiguration mode, that configuration bit netlist is defined by select (frame-strobe) and data (frame-data) wires. In shift-register configuration mode, it is the sequential order of configuration bits and the configuration clock tree that define a netlist. The two netlists are connected by the outputs of the configuration bit cells that control multiplexers and primitives. Even these configuration bit output wires are not timing critical, they will still create a force during place and route of the tile when the primitives and routing fabric have to be implemented together with the configuration bit cells. Therefore, the mapping of configuration bits into frames and frame positions (for

Figure 2: Reducing congestion by swapping the order of the configuration bits that are controlling the switch matrix multiplexers.

frame-based configuration - FBC) or the order inside a configuration shift-register (for shift-register configuration SRC) can impact area, performance, and power figures of a tile.

However, the exact mapping of configuration bits into frames or into a specific shift-register order can be decided arbitrarily. A different mapping of physical configuration bits will result in the same FPGA functionality when we propagate this into the bitstream image for the specific tile. The basic idea of this optimization is sketched in Figure 2 for an SRC example. Here the physical placement of configuration bits is influenced by the tile configuration ports (c_{in} and c_{out}), which, in turn, influences the placement of the shown multiplexers. By swapping the configuration bits such that c_0 controls M_1 and c_1 controls M_0 , congestion can be improved. A corresponding swapping of the configuration bitstream will then implement the same user circuit on the fabric.

While it is correct to say that configuration bits can be arbitrarily remapped, support for partial reconfiguration, however, may benefit from an optimized bitstream encoding. For instance, the fewer frames a LUT function requires to be written to the FPGA, the faster that LUT could be reconfigured individually [21]. Or it is desirable to separate routing resources from primitives when using distributed memory (i.e. when LUTs can be used as small memories or shift registers). Moreover, the configuration mapping may impact power consumption during partial reconfiguration [2]. However, even the market leader Xilinx is mostly omitting these partial reconfiguration-related bitstream encoding constraints in their fabrics, and we will not further discuss the impact of the bitstream encoding on partial reconfiguration in this paper.

4.1 Algorithm and Implementation

To improve the congestion and the physical implementation, we take the EDA tool's report files (which contain coordinates of all primitives) and we use that information to optimize the tile netlist (at technology mapped level). The optimized netlist file is then fed back into the EDA tool to revamp a better and denser placement. We iterate through this process for a number of times to further improve the tile or give up after 100 iterations.

4.1.1 Frame-based Configuration - FBC. In an SRAM memory, we use word lines to select some data from a memory array and bitlines that carry the actual data. As shown in Figure 3a) Frame-based reconfiguration works similar to that and the bit lines (frame_data)

¹The Skywater standard cell library provides a Mux4 primitive that is $9.66 \times 2.72\mu\text{m}^2$, but the Yosys tool was falling back to use a Mux2 and some logic instead (using an accumulated size of $12.42 \times 2.72\mu\text{m}^2$). We found a similar behavior when studying the OpenFPGA GDS in [25].

Figure 3: Configuration storage. a) Frame-based configuration (FBC) using a frame register and latches. b) Shift register configuration (SRC) using DFFs.

are fed from a frame register. This register forms a wide configuration data word that may span the entire height of the device (like in Xilinx Spartan-3 FPGAs) or a region (like in recent Xilinx devices). A major advantage of frame-based reconfiguration is that it allows using cheap latches for memorizing the configuration bits. This is normally well offsetting the cost for the frame register (as shown in Section 8). A tile, like a CLB, will usually require multiple frames, which are sequentially written by enabling the target frame (through `frame_strobe` signals). However, while an SRAM memory is very regular and tightly packed for improved density, the individual configuration cells of FPGA tiles do usually form an irregular grid and some grid positions may be left unused.

The grid of the tile configuration memory is defined by the frame data and frame strobe signals and each configuration memory cell corresponds to a unique data-strobe crosspoint, as shown in Figure 4a). This implies that the number of configuration bit cells is bound by the number of frame-data and frame-strobe wires. Because a regular grid is an efficient way to build a memory array, we will map the configuration bit latches to the closest grid points. Therefore, we are (re)assigning a data-strobe grid coordinate to each configuration cell by manipulating the technology mapped tile netlists, as shown in Figure 4b).

We explored two methods for optimizing the netlist: 1) disconnecting all configuration cells from the grid and mapping them again more regular, and 2) implementing the tile without initial configuration data and strobe signal connections (and therefore removing the force induced by the grid) and jumping to method 1) for setting the grid connections. We found 2) to be more difficult to implement and not delivering better results and we will focus on method 1) for the remainder of this paper.

For mapping the configuration cells to grid points, we build a matrix denoting the Manhattan distance between each of the memory cells and the location of the grid point. We then use the Google Optimization tool (OR tools [13]) to solve the assignment problem on that matrix. The used mixed integer programming solver minimizes the distance between each of the memory cells and the grid locations while assigning each of the memory cells a coordinate. An illustration of the algorithm's operation is shown in Figure 5. The example shows that the algorithm can handle clusters of configuration latches by spreading out to grid points in the neighborhood, which, in turn, may require some latches to be

Figure 4: a) initial FBC. b) optimized FBC. c) derived optimization for SRC.

Figure 5: Illustration of the grid point mapping algorithm.

mapped to further distend grid positions. However, this ensures that we always find a feasible solution (as long as we have at least as many grid points as we have configuration latches).

4.1.2 Shift register Configuration - SRC. As shown in Figure 3b), in SRC, the configuration cells are daisy-chained inside each tile and tile-by-tile. There are several variants to implement this approach. The long serial shift register can be split into multiple parallel and correspondingly shorter chains for improving configuration speed. Some approaches include bypass logic that allows skipping entire tiles from the shifting. This can improve configuration speed and most important it reduces power consumption as it can reduce dynamic power from potentially thousands or millions of configuration bits toggling at each configuration clock cycle. In this paper, we are focusing on single long baseline shift registers.

SRC is easier to implement than FBC as it uses less tile-to-tile routing resources and does not need a frame register or a decoder for frame strobe signals. However, the configuration D flip-flop cells (DFFs) are more expensive than latches and the configuration clock has to be routed with skew in mind. A detailed comparison FBC versus SRC will be provided in Section 8.

As described in the introduction to this section, we can reorder the configuration DFFs inside the configuration shift register to improve the physical implementation of the tiles (see Figure 2). Computing this order is computing a (minimum) Hamiltonian path, which is a well-examined problem. We ran experiments using the OR tools [13] library which gave us some improvements but not as good as hoped for. Instead we found that we can adapt the grid-point mapping from FBC by daisy-chaining the configuration memory cells along the frame strobe signals to form a meander-shape that is

	initial		optimized	
	height,width	area	height,width	area
FBC	215 x 215	$46,225\mu m^2$	193 x 193	$37,249\mu m^2$
SRC _{DFE}	253 x 253	$64,009\mu m^2$	249 x 249	$62,001\mu m^2$
SRC _{CFF}	229 x 229	$52,441\mu m^2$	196 x 196	$38,416\mu m^2$

Table 2: Area improvement through configuration cell placement for frame-based configuration (FSB) and shift register configuration using standard cell D flip-flops (SRC_{DFE}) and custom flip-flops as in [31] (SRC_{CFF})

Figure 6: Frame data and frame strobe connections before and after the optimization of the FABulous CLB tile.

traversing down and up the original frame strobe signals, as shown in Figure 4c).

4.2 Evaluation

We evaluated the optimizations on a CLB netlist with 586 configuration bits [32].

4.2.1 Improvement of FBC CLB Tiles. For FBC, the tile provides 32 frame data and 20 frame strobe wires. Therefore, the grid point mapper has 640 possible grid points to choose from (leaving 54 positions unused). One run of the grid point mapper took 8s on an Intel(R) Xeon(R) CPU E7- 4830 running Python 3.8 and on Ubuntu 20.04.1 LTS. As a reference, implementing a tile netlist in Innovus took 7 to 8 minutes.

The results are summarized in Table 2. Starting from a $215 \times 215\mu m^2$ FBC tile, our approach shrank a physical implementation without design rule violations to $193 \times 193\mu m^2$. This corresponds to a 19.42% reduction in area that we achieved just by rewiring frame data and frame strobe signals. $193 \times 193\mu m^2$ was the smallest tile found after running our optimization for 100 iterations, which took about 12.5 hours to run. Figure 6 shows the changes to the CLB tile netlist connections before and after the first optimization run.

Additionally, we recorded how our method is improving utilization over the iteration runs. The optimization flow is shown in

Figure 7: Optimization flow.

Figure 7. We start from the smallest tile size that was implemented without optimization (i.e. $215 \times 215\mu m^2$) and reduced the width and height by $1\mu m$, apply our netlist transformations and run the physical implementation. If the implementation fails, we run the netlist optimization again on the new placed design and rerun this process until we do not improve further. The improvement is shown in Figure 8. As can be seen, most of the optimization potential was reached after 50 optimization runs (which corresponds to about 6 hours execution time).

4.2.2 The Impact of Metalization Layers. For our experiments, we used the 130nm Skywater process, which provides 5 metal layers. However, so far, we used only 4 metal layers for the implementation of our tiles. It is common practice for embedded FPGAs (eFPGAs) to reserve one metal layer for the top level integration. This is needed for connecting clock trees, power lines, and in many cases, the system that is surrounding the eFPGA will have some signals that have to route across the eFPGA. Leaving one metal layer reserved is the default mode in FABulous to provide users with a robust design experience.

Nevertheless, if we would use the FPGA fabric as a stand-alone version instead of an eFPGA, we can use the top metal layer to support the physical implementation of the fabric. To measure the impact of this on density, we repeated the experiment for the FBC tile with all metal layers enabled while running our optimization loop. With one extra metal layer, the size of the tile can be further reduced to $190 \times 190\mu m^2 = 36,100\mu m^2$, which is another 3.2% improvement.

As a reference, we repeated the experiment by using all metal layers but without our optimization. In this case, the CLB tile size reduced to $202 \times 202\mu m^2 = 40,804\mu m^2$ versus $37,249\mu m^2$ when sparing one metal layer but running our optimizations. This shows that bitstream remapping helps substantially more for improving density than the extra metal layer. The cost of sparing one metal layer is 3.2% in density and this is the cost paid to greatly simplify the integration of an eFPGA fabric into an SoC. Reserving one metal layer will in most cases allow stitching eFPGA fabrics together from fully implemented hard macros (GDS files). This also means that the optimization effort that we spend on optimizing a tile can benefit

Figure 8: Improving utilization by configuration bit remapping over optimization iterations. When a tile implementation is successful we reduce the tile width and height by $1\mu\text{m}$ and rerun the optimization loop.

multiple chips. We can therefore conclude that this optimization comes virtually for free.

4.2.3 Impact of Routing Channel Capacity. In the previous paragraph, we examined how the available metal layers impact the tile implementation. In this paragraph, we will examine how an increasing demand of wires impacts the physical implementation of a CLB tile.

The basic CLB used in the experiments had been designed for small FPGA fabrics. This is reflected in the fabric architecture graph by providing fewer long distance wires, which are normally needed for linking different modules together on a larger FPGA. Subsetting an FPGA architecture graph was, for example, used by Xilinx in their Spartan-3 family where the smallest devices do not provide hex wires [19]. The congestion in an FPGA fabric is related to the number of wires and their length. For instance, if, at a tile, we have 3 wires of length 4 starting in one direction, there will be $3 \times 4 = 12$ wires forming the interface between two adjacent tiles, as shown in Figure 9. This is quantified by the channel capacity that is described in more detail in Section 5.

To examine the effect of adding long distance wires, we added 4 wires in each north, east, south, west direction and set the length to a distance of 4, 8, and 12 tiles. In all three experiments, we used the same adjacency and we only changed the wire span. The baseline CLB tile with FBC configuration has 507 wire connections at its border. The three experiments will add 64, 128, and 192 wires respectively to the tile. The logic resources (i.e. standard cell primitives) of the three tile netlists are the same except for buffers that FABulous is adding on the long distance wires to meet design rule constraints.

The results are listed in Table 3. We can see that the tiles became substantially larger despite that the underlying netlist is almost identical in terms of resources. This reflects the increased stress on the metal stack that is rising with the span. This means that in these experiments, the tile area is bound by the metalization resources rather than the silicon capacity to accommodate gates.

While the tile sizes increase from increasing wire spans, our bitstream remapping is able to offset some of the area increase. For

span		size	area	utilization	C_H	C_V	wire length
4	ref	220x220	48400	57.58%	6.47%	4.07%	1.01E+05
	opt	200x200	40000	70.87%	10.70%	6.52%	9.24E+04
8	ref	237x237	56169	51.02%	6.43%	4.48%	1.25E+05
	opt	216x216	46656	62.14%	8.60%	5.53%	1.14E+05
12	ref	240x240	57600	51.81%	6.35%	4.12%	1.44E+05
	opt	228x228	51984	57.70%	7.49%	5.21%	1.36E+05

Table 3: Impact of increasing wire resources on the CLB FBC implementation. Length and area are in μm and μm^2 . C_H and C_V denotes the horizontal and vertical wire congestion as reported by Innovus. Wire span 4, 8, 12 will add 64, 128, 192 wires respectively to the tile.

wire span 4, 8, and 12, we measured an area improvement of 17.4%, 17.0%, and 9.8% respectively. The reduction for the baseline tile was 19.4% (see Paragraph 4.2.1). This shows that the optimization gets less effective when routing stress increases. For this experimental setting, this behavior is expected because our optimizations remap the bitstream. However, in most cases, long distances wires just route straight through a tile without any multiplexing. This means that there is no direct impact from configuration cells in the implementation of long distance wires.

The results emphasize the importance to design a balanced architecture graph that optimizes both, the silicon density and the use of the metal stack [5]. This is in particular a concern for embedded FPGAs that are commonly more constrained with the number of usable metal layers. For instance, if the physical implementation is heavily congested, then extra multiplexing for tapping wires or using wires bidirectional may become attractive optimization options [23]. While this topic is beyond the scope of this paper, the here proposed optimizations will work orthogonal to such architectural improvements and will provide some extra gain in density virtually for free.

4.2.4 Improvement of SRC CLB Tiles. We repeated the experiments using the SRC configuration mode in the otherwise identical CLB tile. Here we run the experiments two times. In the first run, we generated the CLB in FABulous with a configuration shift register using Skywater standard cell D flip-flops (SRC_{DFF}). Because a flip-flop is about double the size of a latch, OpenFPGA suggests using a custom configuration cell instead of a DFF to offset the area overhead for their shift register configuration. However, OpenFPGA is currently not providing custom configuration cells for Skywater 130. The closest available custom configuration flip-flop is reported for a 45nm process with about 60% the size of a DFF [31]. In this paper, we mostly investigate physical implementation aspects. In order to investigate the impact of a custom configuration cell, we replaced the DFFs from the first run with latches for a second run (SRC_{CFF}). This netlist is not functional, but reassembles well the physical optimization problem when using custom flip-flops (and assuming some very good optimization levels). In both runs, the configuration clock was implemented as a separate clock tree.

The results are recorded in Table 2 for comparison against FBC. The standard cell version of the shift register approach (SRC_{DFF}) is

Figure 9: The channel capacity is the number of wires crossing a cut between any two adjacent tiles.

17,784 μm^2 (38.5%) larger than the corresponding tile using frame-based configuration. This is much more than what we would expect for fundamentally replacing 586 latches with D flip-flops and rewiring these cells. This is a strong indicator that the scan chain is irregular and causing congestion. This observation holds correspondingly for the custom cell shift register version (SRC_{CF}) that due to substituting the D flip-flops with latches uses the same core area for all the physical cells, but is still 13.4% larger than FBC.

Because the bitstream remapping is optimizing congestion, we expect this method to work well for tiles that use SRC and that are very irregular. In fact, bitstream remapping helped to reduce the area of the congested SRC_{CF} example by 36.5%. However, the results also show that even this great improvement and using custom reconfiguration cells cannot beat the optimized FBC that can use off-the-shelf latches for configuration and is still 3.1% smaller.

5 TILE INTERFACE LAYOUT OPTIMIZATION

The *channel capacity* denotes the number of wires that cross a cut between any two tiles (see Figure 9 for an explanation). The channel capacity depends on the number of wires and their span (i.e. the routing distance in tiles). A typical CLB has a channel capacity of about 200–400 wires in both horizontal and vertical directions. In a tile-based FPGA layout, all these channel wires have to be connected to wire pins at the border of each tile. In addition, an FPGA fabric needs some global signals, like clocks and wires for configuring the fabric, which can have a significant impact on the physical implementation. For instance, the FBC tile from the previous section uses 32 wires in the horizontal direction for distributing configuration data and 20 wires in the vertical direction for the frame strobe signals. This contributes to 10% of the 507 IO wires of a CLB tile. The tile interfaces must be designed such that stitching of different tile types (e.g., CLBs with DSPs) is possible. This means that, for example, an output pin at the left tile border has to align with a correspondingly aligned input pin at the right tile border of each tile that we will ever use for stitching.

As shown in Section 3, our FPGA tiles suffer from congestion, and in this section, we will optimize the tile interface to generate a denser packed (better utilized) CLB tile. This problem is not entirely new, and in [6], simulated annealing was used to optimize the physical wire layout of an on-FPGA interface between a static

Figure 10: Tile interface optimization. top) view at the north/south tile border that will use M2 and M4 for tile pins. bottom) longer wires (aggregated input and output side) will use M4 shorter wires M4. Pin positions are selected based on horizontal component.

FPGA system and partially reconfigurable modules. While the wire allocation problem is solved for a user circuit running on an FPGA rather than for the FPGA fabric itself, the problem is closely related to our tile interface optimization.

5.1 Algorithm and Implementation

As a starting point, we analyzed the default FABulous tile layout, which is based on sorting wires by direction and name. To some extent, this is similar to what some scan-chain insertion tools can do (like the DFT in the Synopsys Design Compiler). For a scan chain, this strategy of arranging scan flops by sorting them by name can produce good results as this commonly reflects hierarchies and the vectorization of signals. However, for optimizing a fine-grained and hugely irregular tile netlist, this approach will likely not improve congestion. This is why we developed a dedicated optimization method for the tile interface optimization.

As with the configuration remapping in Section 4, the interface optimization extracts data generated by the EDA tool. This includes the floorplan (with detailed placement information of each standard cell) and the tile netlist (which is mapped to standard cells and may use custom primitives). The optimization problem has two dimensions: 1) the metal layer (we use M1 and M3 for horizontal routing and M2 and M4 for vertical routing) and 2) the position of an interface wire pin along the tile border. In each optimization round, we are computing a new optimized pin placement that is implemented by modifying the physical constraint files of the tile. The optimized interface design is then fed back into the EDA tool with the aim to reduce congestion for allowing a smaller tile area. The optimization follows the flow shown in Figure 7 and instead of optimizing the bitstream mapping, we are now optimizing the tile interface.

The operation of the algorithm is shown in Figure 10. The pin allocation algorithm will consider corresponding pins on opposite sides of the tile, hence forming a pin pair. This is because an input

Figure 11: Improving utilization by tile pin remapping over optimization iterations. When a tile implementation is successful we reduce the tile width and height by $1\mu\text{m}$ and rerun the optimization loop.

pin at, for instance, the north border will be an output at the south border. First, we compute the shortest lengths between a pin pair and any connected primitive for the input and output side separately. Our algorithm will consider only the shortest paths. The strategy here is that in a spanning tree, we want to reach the first node and branch eventually from there to other nodes.

Then, we compute the sum of the lengths for each pin pair and sort them by their combined lengths. The longer combined paths will be mapped to the higher metal layers and the shorter paths to the lower metal layers.

Next we compute the exact pin position in each metal layer. For north-south borders, we will compute the combined horizontal component and feed this into a table that is used by Google Optimization tool (OR tools [13]) to compute the pin positions. This is related to the configuration bit grid point computation but uses only one dimension here. The algorithm will automatically resolve conflicts if multiple paths want to use the same pin pair. The process is repeated correspondingly for the east-west tile borders by incorporating the combined vertical component of each wire.

5.2 Evaluation

We ran the experiment in the same way that we evaluated the configuration bit remapping, and we reduced the tile borders by $1\mu\text{m}$ after each successful implementation until we eventually fail to optimize further. For 120 iterations in total, the optimization took about 10 hours, and it reduced the tile size from $215 \times 215\mu\text{m}^2$ to $199 \times 199\mu\text{m}^2$, a 14.3% reduction in area. Figure 11 shows the progression of the optimization loop over the iterations. Results for applying tile IO pin optimizations to other tiles will be presented in Section 6.1.

6 OPTIMIZING HETEROGENEOUS FABRICS

Modern FPGAs usually provide different types of resources to support the implementation of user circuits that require memory or multiplications. Those functions are expensive to implement just from LUTs [22]. The physical implementation of one FPGA fabric tile type introduces some dependencies between other tile types, and this section is devoted to optimizing such other tiles. The dependencies include the height of the base tile and the horizontal interface as defined by the CLBs. The CLBs will be optimized first. This includes the configuration bitstream remapping (Section 3) and

tile	netlist		size ¹	area ²	congestion
CLB	std	ref	219x219	47961	81.78%
		opt	218x218	47742	84%
	custom	ref	215x215	46225	60.73%
		opt	193x193³	37249	77.24%
REG	std	ref	246x193	47478	83.82%
		opt	243x193	46899 ⁵	84.76%
	custom	ref	245x193	47285	63.27%
		opt	225x193	43425 ⁶	69.28%
DSP	std	ref	206x391 ⁴	80546	80.62%
		opt	200x391	78200 ⁷	83.51%
	custom	ref	190x391	74290	61.64%
		opt	172x391	66861 ⁸	69.30%

¹ in $\mu\text{m} \times \mu\text{m}$;

² in μm^2

³ best available CLB tile; its constraints will be propagated to the REG and DSP tiles

⁴ height is $2 \times 193\mu\text{m} + (2 - 1) \times 5\mu\text{m}$

⁵ 1.2%; ⁶ 8.2%; ⁷ 2.9%; ⁸ 10.0% (improvement)

Table 4: Heterogeneous FPGA tile optimizations.

the tile interface optimizations (Section 5). The dependencies induced by this will now be propagated to the DSP and REG (memory) tiles.

Consequently, when applying our optimizations, we have to implement the CLB firstly and propagate the height but also the (horizontal) interface constraints to the other tiles (see also Section 4.2.4). In the case of the register file tile (REG), the CLB height and the left and right tile border interfaces are propagated from the CLB. In the case of a tile spanning over n multiple vertical adjacent tiles (like the DSP), $n \times \text{height}(\text{CLB}) + (n - 1) \times \text{tile_gap}$ is propagated as the height, and the CLB interface is propagated to each elementary cell (e.g., two times in our DSP example where a DSP uses the height of two CLBs and provides two switch matrices to the routing fabric).

The results in Table 4 are for tiles using FBC configuration mode. The numbers confirm the low improvement of our optimizations for the standard cell designs because these problems are not very congested (see also Section 3). When using custom cells, congestion is rising and allows our optimizations to kick in. However, the improvements (8.2% and 10.0%) are below the numbers that we reported for the CLBs in Section 4 and Section 5. We believe the reason for this is that these specialized tiles provide overall more regularity inside the netlist, which helps Innovus with the physical implementation such that, for example, the remapping of configuration bits is working somewhat less effective. Nevertheless, the improvements are still significant and can be gained fully automated.

Figure 12: Area against the baseline FBC with custom multiplexers (Mux4).

6.1 Optimizing Heterogeneous Tile Interfaces

We applied IO interface optimization where we remap tile pins to optimize the DSP and register file tiles. This optimized the area of the register file tile (REG) from $245 \times 193 \mu\text{m}^2$ to $225 \times 193 \mu\text{m}^2$ and for the DSP tile from $190 \times 391 \mu\text{m}^2$ to $173 \times 391 \mu\text{m}^2$. Interestingly, these are almost identical improvements that we achieved by the configuration cell remapping.

7 SIDE-BY-SIDE COMPARISON

We collected the area results of our different optimizations in Figure 12. We took the default FABulous CLB tile with frame-based configuration as the baseline. We found that our present configuration bit remapping and tile pin remapping had been able to gain the same improvement in area (-19.4%). We believe that both optimizations are hitting a utilization level (about 85% core utilization) from which Innovus stops improving any further.

We also applied both optimizations together, which improved the tile to -21.7% , the best improvement we achieved by our optimization over the baseline tile.

The figure also contrasts frame-based versus shift register modes for building an FPGA CLB tile. As can be seen, shift register configuration results in much bigger tiles, which can only be partially offset by custom configuration cells if the design is congested. Similar to the custom multiplexer optimization (see Section 3), using custom configuration cells is mostly resulting in an underutilized tile. However, when rearranging the configuration bits, we can reduce the tile size close to what we achieved for FBC mode.

7.1 Timing Improvement

Even our main objective in this paper was to improve density, a denser (=smaller) design will likely also improve latency as wire lengths get shorter. Moreover, our optimizations fight congestion meaning that the total wire length will be reduced. As a metric, we measured the latency of all tile input pins to all tile output pins. We recorded the worst-case latency as this is most relevant for improving critical path delay in a user circuit. As can be seen in Figure 13, the latency improvements are minor than the area improvements. This is expected as the wire length is only one contributor to the overall wire propagation delay. From the previous

Figure 13: Timing comparison of different optimized CLBs.

paragraph, we saw that our optimizations can improve area similar good as adding an extra metal layer. However, for latency, the additional metal layer can cut latency over the baseline by 12.4% versus 6.1% that we achieve just from optimizations.

7.2 FABulous versus OpenFPGA on Sky130

The latest Google Shuttle run [12] on a Skywater 130nm process includes both a FABulous [32] and an OpenFPGA FPGA design [25]. Both designs implement a CLB with 8 LUT-4. The openFPGA CLB allows fracturing the LUTs into two LUT-3, which is essentially tapping into the LUT multiplexer tree. The CLB resources compare as follows:

	resources
OpenFPGA	1300xMUX2 / 530xDFF
FABulous	376xMUX4 / 46xMUX2 / 8xFF / 586xlatch

Considering that 1xMUX4 is about 3xMUX2, we can say that both CLBs are similarly complex and state a similar physical optimization problem. Figure 14 compares the Shuttle-2 implementations against our optimized FABulous FPGA tile. We can see that our optimizations deliver a CLB that is 1/3 smaller than the OpenFPGA CLB tile.

8 FRAME-BASED VERSUS SHIFT-REGISTER CONFIGURATION MODES

Even though shift-register configurations are easier to implement, there are multiple advantages of frame-based configuration modes over shift-register configuration, the most important include:

- **Latches for configuration storage**, which saves area.
- **Partial reconfiguration**. While SRC could be implemented using tile bypass multiplexers, this is not suited for PR because it is commonly required to configure routing resources individually (e.g., to route a static signal through a partial region). SRC would not allow this as all configuration bits are shifted together, while FBC can update each individual configuration bit separately.
- **Lower power consumption during configuration**. A CLB tile uses about 500 to 1000 configuration bits (about two orders of magnitude more configuration memory bits than

Figure 14: Area comparison of Shuttle-2 tapeouts of OpenFPGA and FABulous CLB tiles and the optimized tile.

the number of user flip-flops in a CLB). Therefore, shifting in a configuration may potentially toggle large numbers of flops simultaneously with a corresponding dynamic power footprint. Moreover, in shift register configuration, a configuration is only valid if it reaches its final position. However, during the configuration process, a configuration may cause many unwanted intermediate configurations like ring oscillators or even short circuits, as shown in [2].

- **Fast configuration blanking and configuration wild-carding.** For resetting the configuration, SRC usually requires us to shift in a blanking stream, which takes as long as the configuration. Frame-based reconfiguration normally allows writing the configuration from the frame register to multiple frames, which can implement a fast configuration multicasting (e.g., the multiple framewrite feature (MFWR) in Xilinx FPGAs [37]. This mode is available in Xilinx FPGAs and FABulous FPGAs and can be used for fast configuration blanking or some configuration compression schemes.
- **Versatility.** Frame-based configuration allows for more optimizations, like combined cells for configuration storage and multiplexing where it may be undesirable to provide the configuration state outside the cell. Moreover, FBC will be mandatory for non-volatile configuration storage options, like emerging ReRAM technology.

Table 2 showed that an FBC CLB tile is 3.1% smaller than an optimized SRC CLB. A frame register costs about 30% the width of a CLB tile, and the frame select logic 20% of the height of a CLB. Therefore, after 10 columns and 7 rows of CLBs, the frame-based configuration would be cheaper to implement, even considering that custom configuration cells could save half the configuration cell area (which is an optimistic estimate for the achievable optimization).

9 CONCLUSION

This paper proposed *configuration cell remapping* and *tile interface pin remapping* as optimization techniques for shrinking FPGA tiles. Combining both techniques gained an area improvement of 21.7%. We evaluated our approach over various different scenarios, including changing the number of metal layers and examining routing fabrics that use different channel capacities. In this paper, we provide a complete optimization approach that includes all tiles, including logic, memory arithmetic, or any other specialized tile. The numbers collected in this paper represent a full week (24/7) of

execution time on a large server, including failed runs, probably close to a month.

As a further contribution, we contrasted frame-based configuration mode versus shift-register configuration mode. Our results show that frame-based configuration is winning in all aspects for any realistic fabric size.

The here presented optimizations can be applied to FPGA tiles that include hierarchies, and in fact, the used FABulous tiles do all include hierarchies. While we used the FABulous FPGA framework to run our experiments, there is no fundamental obstacle for users to integrate our optimizations into other frameworks.

With more than 20% improvement in area, our results are better than what we had expected when starting this project. This improvement was made possible by reversing the physical optimization loop as shown in Figure 1. Normally, a model is refined over multiple abstraction levels to produce the physical ASIC implementation in a strict top-down manner. In our flow, we modify the physically implemented netlist at the bottom and propagate the outcome back to the model (e.g., in the form of a bitstream remapping table). While this approach may not work well for general ASIC design, it is an interesting avenue for future research for implementing FPGA fabrics.

With this work, we contribute to the growing open-hardware community and empower users to integrate high-quality FPGAs (and reconfigurable fabrics in general) into their designs. For instance, the RISC V movement has created much interest in (open) hardware design, and eFPGAs can add exciting features to corresponding systems (e.g. [9] where a RISC V CPU had been coupled with an embedded FPGA for runtime reconfigurable custom instruction set extensions).

This work integrates into a larger ecosystem of open-source projects, such as SymbiFlow [36], Yosys [35], nextpnr [29], VPR [4], OpenRAM [16], OpenLane [10] and the Verilator [34], but also the open PDK from Skywater [30]. Our next step will include replacing Innovus with OpenLane to provide an open-everything FPGA experience with the aim to provide a quality of results close to or, perhaps, even better than what industry tools achieve due to customization and optimizations.

ACKNOWLEDGMENT

This work is part of the project FORTE [33], which is supported by the UK Engineering and Physical Sciences Research Council (EPSRC) grant EP/R024642/1.

REFERENCES

- [1] V. Aken'Ova and R. Saleh. 2006. A "Soft++" eFPGA Physical Design Approach with Case Studies in 180nm and 90nm. In *IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures (ISVLSI'06)*.
- [2] C. Beckhoff, D. Koch, and J. Torresen. 2010. Short-Circuits on FPGAs Caused by Partial Runtime Reconfiguration. In *20th International Conference on Field Programmable Logic and Applications (FPL)*.
- [3] Berkeley Logic Synthesis and Verification Group. [n.d.]. ABC: A System for Sequential Synthesis and Verification. <http://www.eecs.berkeley.edu/~alanmi/abc/>.
- [4] Vaughn Betz and Jonathan Rose. 1997. VPR: a New Packing, Placement and Routing Tool for FPGA Research. In *Field-Programmable Logic and Applications (FPL)*. Springer Berlin Heidelberg, 213–222.
- [5] Vaughn Betz, Jonathan Rose, and Alexander Marquardt (Eds.). 1999. *Architecture and CAD for Deep-Submicron FPGAs*. Kluwer Academic Publishers, Norwell, MA, USA.
- [6] Jeff Carver, Neil Pittman, and Alessandro Forin. 2008. *Technical Report MSR-TR-2008-111 – Relocation and Automatic Floor-planning of FPGA Partial Configuration Bit-Streams*. Technical Report. Microsoft Research.
- [7] Charles Chiasson and Vaughn Betz. 2013. Should FPGAs abandon the pass-gate?. In *23rd International Conference on Field programmable Logic and Applications (FPL)*. 1–8.
- [8] Menta Corporation. 2018. *Menta eFPGA Website*. Retrieved April 1, 2020 from <https://www.menta-efpga.com/>
- [9] Nguyen Dao, Andrew Attwood, Bea Healy, and Dirk Koch. 2020. FlexBex: A RISC-V with a Reconfigurable Instruction Extension. In *International Conference on Field-Programmable Technology (FPT)*. IEEE.
- [10] eFabless. 2021. <https://openlane.readthedocs.io/en/latest>.
- [11] eFabless. 2021. <https://efabless.com/projects/public>.
- [12] eFabless. 2021. https://efabless.com/projects/shuttle_name/MPW-2.
- [13] Google. 2015. <https://github.com/google/or-tools>.
- [14] B. Grady and J. H. Anderson. 2018. Synthesizable Heterogeneous FPGA Fabrics. In *2018 International Conference on Field-Programmable Technology (FPT)*. 222–229.
- [15] Brett Grady and Jason H. Anderson. 2019. Physical Design Considerations for Synthesizable Standard-Cell-Based FPGAs. In *Proceedings of the 10th International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies*. 1–6.
- [16] Matthew R. Guthaus, James E. Stine, Samira Ataei, Brian Chen, Bin Wu, and Mehedi Sarwar. 2016. OpenRAM: An Open-Source Memory Compiler. In *ICCAD*.
- [17] Jin Hee Kim and J. H. Anderson. 2015. Synthesizable FPGA Fabrics Targetable by the Verilog-to-Routing (VTR) CAD flow. In *Proceedings of the 25th International Conference on Field Programmable Logic and Applications (FPL)*.
- [18] 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.
- [19] Dirk Koch. 2012. *Partial Reconfiguration on FPGAs: Architectures, Tools, and Applications*. Springer.
- [20] Dirk Koch, Nguyen Dao, Bea Healy, and Andrew Attwood. 2021. FABulous: an Embedded FPGA Framework. In *Proceedings of the 29th ACM/SIGDA international symposium on Field-programmable gate arrays (FPGA)*.
- [21] Amit Kulkarni and Dirk Stroobandt. 2016. How to Efficiently Reconfigure Tunable Lookup Tables for Dynamic Circuit Specialization. *International Journal of Reconfigurable Computing* 2016 (2016).
- [22] Martin Langhammer and Gregg Baekler. 2018. High Density and Performance Multiplication for FPGA. In *2018 IEEE 25th Symposium on Computer Arithmetic (ARITH)*. 5–12.
- [23] G. Lemieux, E. Lee, M. Tom, and A. Yu. 2004. Directional and Single-Driver Wires in FPGA Interconnect. In *IEEE International Conference on Field- Programmable Technology (IEEE Cat. No.04EX921)*. 41–48.
- [24] Ang Li and David Wentzlaff. 2021. PRGA: An Open-Source FPGA Research and Prototyping Framework. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*. 127–137.
- [25] LNI. 2021. <https://github.com/linis-uofu/SOFA>.
- [26] Prashanth Mohan, Oguz Atli, Onur O Kibar, and Ken Mai. 2020. A Top-Down Design Methodology for Synthesizing FPGA Fabrics Using Standard ASIC Flow. In *Proceedings of the 28th ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*.
- [27] Kevin E Murray, Oleg Petelin, Sheng Zhong, Jia Min Wang, Mohamed Eldafrawy, Jean-Philippe Legault, Eugene Sha, Aaron G Graham, Jean Wu, Matthew JP Walker, et al. 2020. VTR 8: High-performance CAD and Customizable FPGA Architecture Modelling. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 13, 2 (2020), 1–55.
- [28] K. Pham, E. Horta, and D. Koch. 2017. BITMAN: A Tool and API for FPGA Bitstream Manipulations. In *Design, Automation, and Test in Europe (DATE)*.
- [29] D. Shah, E. Hung, C. Wolf, S. Bazanski, D. Gisselquist, and M. Milanovic. 2019. Yosys+nextpnr: An Open Source Framework from Verilog to Bitstream for Commercial FPGAs. In *IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*.
- [30] Skywater Technology. 2021. <https://github.com/google/skywater-pdk>.
- [31] Xifan Tang, Edouard Giacomin, Baudouin Chauviere, Aurélien Alacchi, and Pierre-Emmanuel Gaillardon. 2020. OpenFPGA: An Open-Source Framework for Agile Prototyping Customizable FPGAs. *IEEE Micro* 40, 4 (2020), 41–48.
- [32] University of Manchester. 2021. <https://github.com/FPGA-Research-Manchester/FABulous-Sky---a-heterogeneous-FPGA-fabric-in-Skywater130>.
- [33] University of Southampton, Imperial College London, and University of Manchester. [n.d.]. *The FORTE (Functional Oxide Reconfigurable Technologies) project website*. Retrieved Nov. 15, 2020 from <http://www.forte.ac.uk/>
- [34] Veripool. 2021. <https://www.veripool.org/verilator>.
- [35] Clifford Wolf. [n.d.]. Yosys Open Synthesis Suite. <http://www.clifford.at/yosys/>.
- [36] Claire Wolf, David Shah, Aliaksei Chapyzenka, Karol Gugala, and Tim Ansell. [n.d.]. SymbiFlow Open source flow for generating bitstreams from Verilog. <https://github.com/SymbiFlow/>.
- [37] Xilinx Inc. 2015. *UG470 - 7 Series FPGAs Configuration User Guide*.