Brigham Young University
BYU ScholarsArchive

All Theses and Dissertations

2017-07-01

Academic Packing for Commercial FPGA Architectures

Travis D. Haroldsen
Brigham Young University

Follow this and additional works at: https://scholarsarchive.byu.edu/etd

Part of the Electrical and Computer Engineering Commons

BYU ScholarsArchive Citation

Haroldsen, Travis D., "Academic Packing for Commercial FPGA Architectures" (2017). *All Theses and Dissertations*. 6526.
https://scholarsarchive.byu.edu/etd/6526

This Dissertation is brought to you for free and open access by BYU ScholarsArchive. It has been accepted for inclusion in All Theses and Dissertations by an authorized administrator of BYU ScholarsArchive. For more information, please contact scholarsarchive@byu.edu, ellen_amatangelo@byu.edu.

Academic Packing for Commercial FPGA Architectures

Travis D. Haroldsen

A dissertation submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Brent E. Nelson, Chair
Brad L. Hutchings
Michael J. Wirthlin
Doran K. Wilde
James K Archibald

Department of Electrical and Computer Engineering

Brigham Young University

Copyright © 2017 Travis D. Haroldsen

All Rights Reserved

ABSTRACT

Academic Packing for Commercial FPGA Architectures

Travis D. Haroldsen
Department of Electrical and Computer Engineering, BYU
Doctor of Philosophy

With a few exceptions, academic packing algorithms for FPGAs are typically applied solely to theoretical architectures. This has allowed the algorithms to focus on the basic components of packing while abstracting away many of the details dictated by real hardware. As commercially available FPGAs have advanced, however, the academic algorithms and architectures have diverged significantly from their commercial counterparts.

In this dissertation, the RapidSmith 2 framework is presented. This framework accurately reflects the architecture of Xilinx FPGAs and provides support for integrating custom tools into the commercial CAD tools. Using this framework, the RSVPack packing algorithm is implemented. The RSVPack algorithm can accept a design synthesized using the commercial Xilinx CAD tools, pack designs which make use of the many features of commercial FPGA architectures and return the packed designs to the Xilinx CAD tools to be placed and routed in their software. This enables researchers to isolate the packing portion of the algorithm from the commercial flow and evaluate different packing techniques while allowing the high-quality commercial tools to perform the remainder of the flow. Integrating the RSVPack algorithm the commercial flow shows RSVPack produces packing which lead to circuits with minimum clock periods within 10%, on average, of circuits generated using the pure Xilinx flow.

Included in this work is a novel table lookup-based algorithm which RSVPack utilizes to quickly determine the routability of a cluster. This algorithm performs 5 times faster on average than the current academic alternatives. Finally, using RSVPack, this dissertation explores various techniques for improving the quality of packing for Xilinx circuits. Together, this demonstrates the potential for academic research into FPGA CAD tools for commercial architectures.

Keywords: FPGA, packing, Xilinx, algorithms

ACKNOWLEDGMENTS

This path has taken far longer than I imagined when I started back in 2011[1]. First and foremost, I need to express profound gratitude to my wife, Amanda. Even with the countless late nights and the many Saturdays that I have spent on campus instead of with her and our children, she has been amazingly patient and supportive of me throughout this entire process. I would also like to thank my children for their patience and bright smiles that always bring me cheer. I express gratitude to my parents. From my childhood, they have instilled in me a love and respect for learning that has enabled me through this work.

I would also like to recognize my colleagues I have worked with in the Configurable Computing Laboratory. I have enjoyed the many conversations I have had with them and each of them has made the work more enjoyable. In particular, I would like to acknowledge Josh Monson and Jon-Paul Anderson who have been some of the few constants throughout my studies. I appreciate the advice and recommendations both have freely offered me. I would also like to single out Thomas Townsend whose help has been instrumental in preparing RapidSmith 2 for public release.

I would like to thank my adviser, Brent Nelson. I appreciate his patience and the counsel he has provided at times when the path forward looked hopeless. Lastly, I would like to thank Brad Hutchings who for the last couple of years has served as a second adviser to me. I appreciate his input and the insights he has brought as I have pursued my studies.

[1]This work was supported in part by the I/UCRC Program of the National Science Foundation within the NSF Center for High-Performance Reconfigurable Computing (CHREC), Grant No. 1265957

TABLE OF CONTENTS

List of Tables vi

List of Figures vii

Chapter 1 Introduction 1
1.1 Contributions of this Work 4
1.2 Dissertation Organization 5

Chapter 2 Background and Related Work 6
2.1 Xilinx FPGA Architecture 6
2.1.1 Tiles 6
2.1.2 Sites 8
2.2 FPGA CAD Flow 11
2.2.1 Traditional CAD Flow 11
2.2.2 Xilinx ISE CAD Flow 13
2.2.3 Academic CAD Frameworks 14
2.2.4 Academic Packing Algorithms 16
2.3 Xilinx Design Language and RapidSmith 18
2.3.1 XDLRC File Format 19
2.3.2 XDL 19
2.3.3 XDL/ISE Interoperability 20
2.3.4 RapidSmith 1 21

Chapter 3 RapidSmith 2 23
3.1 Subsite Device Representation 24
3.1.1 Building Site Templates 25
3.2 Design Netlist 28
3.2.1 Route Trees 29
3.3 XDL Compatibility 30
3.3.1 Integrating Custom Packers into ISE with RapidSmith 2 32
3.3.2 Non-Xilinx Architectures in RapidSmith 2 32

Chapter 4 RSVPack: A Packer for Xilinx FPGAs 33
4.1 RSVPack Overview 34
4.1.1 Pack Units 34
4.2 RSVPack Algorithm 35
4.2.1 Seeding Clusters 35
4.2.2 Seeding a Cluster 35
4.2.3 Cluster Legality Validation 38
4.2.4 Conditional Mode 44
4.2.5 Manual Pairing of LUTs and FFs 45
4.3 Table-Lookup Routing Feasibility Improvement 46

4.3.1 Routing Feasibility: VTR's AAPack versus RSVPack 46
4.3.2 Run Time Improvement 49
4.4 Identifying Packing Rules for Xilinx Architectures 51
4.5 Integrating RSVPack into the Xilinx ISE CAD Flow 52
4.5.1 Pure ISE Flows 53
4.5.2 RSX Flow 54
4.5.3 RSV Flow 54
4.6 RSVPack Performance 55
4.6.1 Benchmark Set 55
4.6.2 RSVPack Results 56
4.6.3 Summary 59

Chapter 5 Experimenting with Packing Xilinx FPGAs Using RSVPack 61
5.1 Experiment 1: Impact of Packing on Circuit Quality 61
5.1.1 Experiment Setup 61
5.1.2 Results 62
5.2 Experiment 2: Cluster Underutilization 63
5.2.1 Two-Hop Deep Cell Selector 65
5.2.2 Experiment Setup 66
5.2.3 Results and Conclusion 66
5.3 Experiment 3: Tile Versus Site Level Packing 67
5.3.1 Background 68
5.3.2 Experiment Setup 69
5.3.3 Results and Analysis 70
5.3.4 Experiment Conclusions 73
5.4 Summary 74

Chapter 6 Conclusion 75
6.1 Summary of Research 75
6.2 Future Work 76

REFERENCES 78

Appendix A Publications and Documentation 84

Appendix B RapidSmith 2 Changelist 85

Appendix C XDL-Unpacking and XDL-Packing Algorithms 86
C.1 Design Unpacking 86
C.2 XDL-Packing 88

Appendix D Checks Required for Packing for a Virtex 6 Device 92

LIST OF TABLES

2.1 CAD Flow Stage Inputs and Outputs 11

4.1 Routing Connectivity Lookup Table for Figure 4.7a 48
4.2 Average Runtimes for Each Outcome of Routing Feasibility Checking (in microseconds) 50
4.3 Benchmarks Used to Analyze RSVPack 55
4.4 Number of Used Slices and Percent of LUT5s Merged 56
4.5 Clustering of LUT/FF Pairs and Nets Exposed From Slices 57
4.6 Minimum Periods of Benchmarks Implemented with Different Flows (in ns) 58
4.7 Total Wire Length of Benchmarks Implemented with Different Flows (in Thousands of Tiles Travelled) 58
4.8 Implementation Runtimes for each Flow in Seconds (Median of 20 Runs) 60

5.1 Influence of 2-Hop Deep Cell Selector on Benchmarks 66
5.2 Used Slices for RSV-Site and RSV-Tile Implemented Benchmarks 70
5.3 HPWL, Used Wire Length, and Minimum Periods of RSV-Site and RSV-Tile Implemented Benchmarks 70
5.4 Run Times for RSV-Tile and RSV-Site in Seconds 72
5.5 Used Wire Length and Minimum Periods of RSX-Site and RSX-Tile Implemented Benchmarks 72

LIST OF FIGURES

1.1 A Traditional Academic LE (1.1a) vs a Virtex 6 LE (1.1b) 2
1.2 The Traditional CAD Flow 3

2.1 Hierarchy of a Xilinx Tile 6
2.2 Device Arranged as a Grid of Tiles 7
2.3 Connectivity Inside a Switch Box 8
2.4 A Xilinx Virtex 6 CLB and Adjacent Switch Box 9
2.5 A Xilinx Virtex 6 Logic Element 10
2.6 Correspondence Between the Traditional and Xilinx ISE CAD Flows 12
2.7 Sample of an XDLRC Device Description 19
2.8 A 4-Bit Adder Netlist (a) and its XDL Representation (b) 20
2.9 Steps in the ISE Tool Chain Where Designs can be Converted to XDL 21

3.1 The Extended Device Representation in RapidSmith 2 24
3.2 The Site Level Hierarchy (shaded) of Xilinx FPGAs 25
3.3 An XDLRC Primitive_def (left) with its RapidSmith 2 Representation (right) 26
3.4 A Polarity Mux Schematic (left) and its XDLRC Representation (right) 28
3.5 Relationship Between RapidSmith 2 Netlist API Classes 29
3.6 Subsite and Intersite Sections of a Route Tree 30
3.7 Converting Between an XDL Instance and RS2's Cell-Based Netlist 30
3.8 Flow for a Custom Packer Using RapidSmith 2 32

4.1 Possible Invalid Packings for Clusters With Multiple Carry Chains 39
4.2 Examples of Legal and Illegal Fracturable LUT Usage 42
4.3 Example of an A5LUT that must be Conditionally Packed with a Carry Chain 43
4.4 Carry chain element using both DO and CO pins must be packed with a FF 43
4.5 Example of Entering and Resolving Conditional Mode 44
4.6 Example of Packing LUT/FF Pairs Together 45
4.7 Example for Determining Routing Feasibility 48
4.8 Projected Run Times Using Pin Counting 50
4.9 Implementation Flows used in this Work 53
4.10 Densities of Slices Packed with Xilinx and RSVPack (average of benchmarks) 57

5.1 Average Increase in Minimum Period Against Packing Quality 62
5.2 Average Increase in Wire Length Against Packing Quality 63
5.3 Progression of Packing Density of Clusters over Packing Process 64
5.4 Example of a Pocket Forming Around a Cell (B) During Packing 64
5.5 Utilization of LEs in Slices when Using Single-Hop and Two-Hop Cell Selectors . . . 67
5.6 Progression of Packing Density of Clusters over Packing Process with Two-Hop Cell Selector 67
5.7 Flows for Comparing Tile-Level and Site-Level Based Packing 69
5.8 Non-Overlapping Carry Chains Paired Together in a CLB 74

C.1 Converting between XDL and RapidSmith 2 with the Design-Unpacker 86

C.2 A BEL entry in the `unpack.xml` 87
C.3 A BEL entry in the `pack.xml` 90

CHAPTER 1. INTRODUCTION

Publicly available research into FPGA CAD algorithms and frameworks is important in opening up exploration of novel concepts relating to the FPGA CAD flow to the academic research community. The availability of this information and these tools enables new techniques to both the general CAD flow as well as special applications, such as FPGA reliability and security that may not be addressed by the FPGA vendor tools. Because the commercial vendors do not release their algorithms and internal tools to the public, it is necessary for people interested in researching new algorithms and flows to develop their own tools. For the remainder of this dissertation, the public information on CAD tools available in the public research literature will be referred to as "academic".

Academic research into CAD algorithms for commercial FPGA architectures has been hindered by a lack of CAD infrastructure for these devices. As commercial architectures have added advanced features and capabilities to the configurable logic blocks (CLBs), academic frameworks and algorithms have settled for working with FPGA architectures, like those used in [1], that resemble less and less the architectures available from the commercial vendors.

The traditional architecture used in academic research is comprised of a collection of CLBs made up of multiple LUT/flip-flop (FF) pairs, called logic elements or LEs, like the one shown in Figure 1.1a. In contrast, the LEs found in recent commercial architectures, like the one in Figure 1.1b, contain many special-purpose components connected with sophisticated routing and often interact with other LEs in the same CLB. Additionally, each of the LEs in the CLBs in these architecture have slight variations leading to an irregular structure in how they interconnect.

Packing algorithms for FPGAs are especially affected by the added complexity of the CLBs. *Packing* is a step in the traditional FPGA CAD flow (Figure 1.2) and which is responsible for assembling the LUTs, FFs, and other basic components into relatively-placed structures, called *clusters*, that map onto the CLBs. Packing is, therefore, responsible for handling the highly-

constrained routing environment and other constraints associated with the CLBs, thus simplifying the task of the subsequent global placer. Packing also acts as a localized placement step in the flow and uses algorithms suited to grouping closely related logic to reduce the number of nets that spread to different CLBs.

The added complexity found in the commercial CLBs but which is not present in the academic CLBs adds many requirements to the packing algorithms. Additional requirements include verifying that a valid routing path exists between elements in the CLBs and ensuring various design rule constraints, such as requirements that certain LUTs in the CLB be used, of the CLBs are satisfied. The traditional academic architectures typically do not require these routing checks as valid routes always exists. Additionally, the academic architectures typically do not have any design rule constraints that must be met.

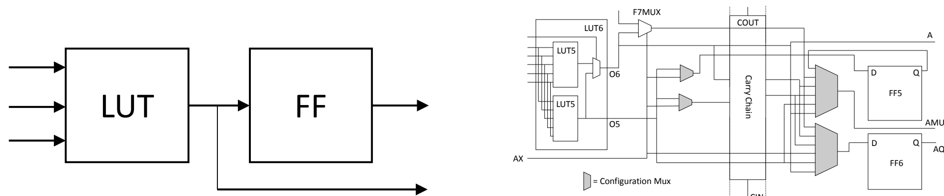


Figure 1.1: A Traditional Academic LE (1.1a) vs a Virtex 6 LE (1.1b)

The discrepancy between commercial architectures and the architectures used in academic research is partly due to the lack of publicly available frameworks capable of representing the more complex commercial architectures. To work with commercial architectures, packers depend on a faithful representation of the CLB architecture and on input from good front-end synthesis and technology-mapping tools that can target the features of the CLBs.

Additionally, a framework that accurately represents a commercial architecture opens up the possibility of integrating the packer into a commercial flow. Integrating academic algorithms into the vendor's tools chain allows the particular step being explored to be evaluated in conjunction with the vendor tools. As the commercial vendor tools are generally of high-quality, using the vendor tools helps isolate the exact impact a new technique to a specific part of the CAD flow has on the final circuit. In contrast, when using a purely academic flow, different inefficiencies in the tools may compound, possibly hiding the effect of a new technique to a certain part of the

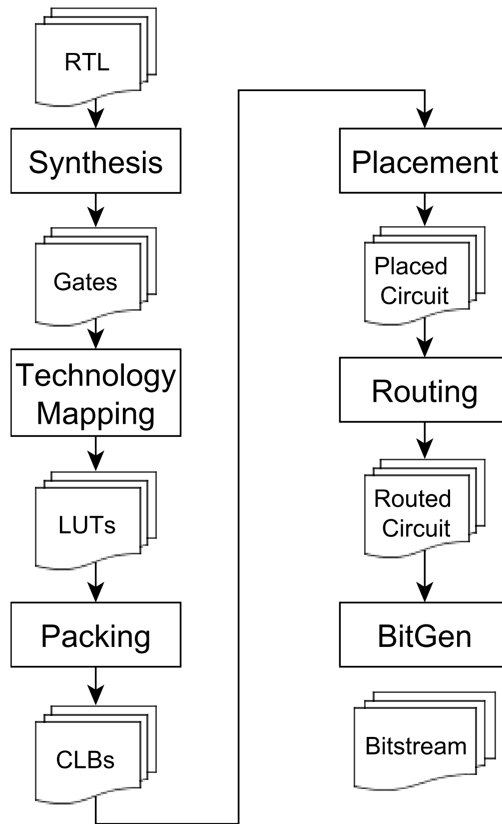


Figure 1.2: The Traditional CAD Flow

flow. Integration of academic tools with the commercial flows also provides access to the different analysis tools provided by the vendors.

Through use of the Xilinx XDL language and tools [2], the RapidSmith framework [3] has created a pathway for integrating custom CAD tools with the commercial Xilinx CAD flow. With RapidSmith, designs can be exported from the Xilinx ISE tool chain, modified, and re-imported back into the ISE tool chain. This has enabled large amounts of research that is targeted at Xilinx FPGAs that previously could not be performed [4–16]. This pathway, however, is limited to modifying fully packed designs. While modifying the packing of designs is possible in RapidSmith, the work of manipulating the text attributes is tedious and difficult to automate. This, along with the lack of a traversable routing graph of the internal structure of the sites makes creating custom packers using RapidSmith infeasible.

As part of this work, I present RapidSmith 2 [17], which provides improvements to the original RapidSmith framework that enable the creation of custom packers for Xilinx FPGAs.

Packing algorithms created using the RapidSmith 2 framework can be integrated into the Xilinx tool flow, enabling the user to evaluate the packing portion of the flow in conjunction with the ISE CAD flow. Furthermore, once integrated into the ISE tool chain, the timing and power impacts of the algorithms can be evaluated using the vendor software and the circuits produced with the tool can be programmed onto physical hardware.

Using RapidSmith 2, I have developed the RSVPack packing algorithm capable of creating legal packings of designs for Xilinx FPGAs. This algorithm is integrated into the Xilinx ISE CAD flow; it accepts a Xilinx technology-mapped netlist as input and returns a packed netlist for Xilinx to place and route. The resulting circuit is then evaluated using the Xilinx timing analysis tool. This dissertation will describe the RSVPack algorithm and present solutions to the various challenges related to packing a commercial architecture. Using RSVPack, this dissertation then explores the impact packing has on circuits targeted at Xilinx FPGAs and evaluates different techniques for improving the quality of packed circuits coming from the packer.

1.1 Contributions of this Work

This work presents the following contributions:

- It provides the RapidSmith 2 framework, an extension to RapidSmith, that enables the creation and modification of the packing of designs targeted at Xilinx FPGAs. This framework is publicly released online as open source.
- It presents the RSVPack packing algorithm that is capable of packing a design targeted at a Xilinx FPGA. The algorithm in this work is adapted for the Xilinx Virtex 6 and 7 FPGA architectures and differs from previous algorithms in that it:
  - identifies and performs a set of checks to ensure all unique requirements of the Virtex 6 architecture, in addition to general requirements for packing, are satisified,
  - uses a new method to quickly determine if a legal routing exists for a cluster which runs 5 times faster than the speculative packing approach described in [18].
- It describes approaches to discovering design rule requirements of the architecture and creating rules to ensure those requirements are addressed by the packer.

- It presents two flows for integrating the RSVPack algorithm into the Xilinx CAD flow. One flow uses a Xilinx placer while the other uses an academic simulated annealing placer and provides more visibility into and control of the placement to the user.
- It compares circuits packed using both RSVPack and Xilinx's packing using wire length, clock rate and tool run time as metrics.
- It evaluates different techniques for improving packing for designs targeted at Xilinx devices including:
  - comparing the impact of packing at different levels of the device hierarchy and
  - comparing the impact of looking at cells both one and two hops away from the cluster.

1.2 Dissertation Organization

This dissertation is divided into six chapters:

- **Chapter 2** provides background necessary for understanding the remainder of the dissertation. Included is a review of the traditional FPGA and Xilinx ISE CAD flows, the XDL tool set, and RapidSmith. This chapter also reviews prior work in academic FPGA frameworks, CAD flows and packing algorithms.
- **Chapter 3** describes the extensions to RapidSmith published in RapidSmith 2 and how they enable creating and modifying the packing of Xilinx-synthesized designs.
- **Chapter 4** presents the RSVPack packing algorithm and describes the challenges involved with packing a commercial FPGA and the solutions to those challenges. It then presents RSX and RSV flows for integrating the RSVPack algorithm into the Xilinx ISE CAD flow. Finally, it presents the performance of the algorithm when using the two flows.
- **Chapter 5** details experiments that have been performed using RSVPack to analyze the best approaches for packing for Virtex 6 FPGAs.
- **Chapter 6** summarizes the results and contributions of this dissertation and gives suggestions for future work in this area.

CHAPTER 2. BACKGROUND AND RELATED WORK

2.1 Xilinx FPGA Architecture

Xilinx structures their FPGAs into three levels of hierarchy. The elements at each level, from top to bottom, are called *tiles*, *primitive sites* or simply *sites*, and *Basic Elements of Logic* or BELs (Figure 2.1).

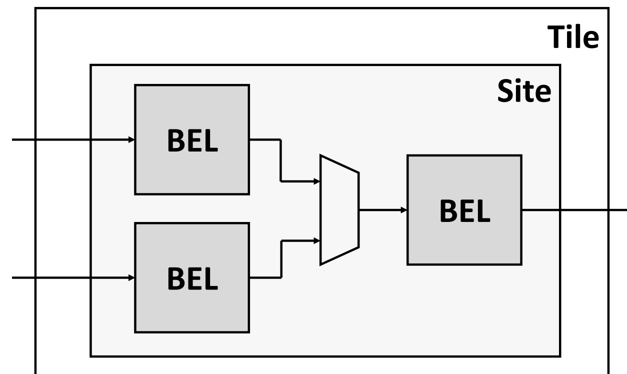


Figure 2.1: Hierarchy of a Xilinx Tile

2.1.1 Tiles

Xilinx architectures are organized as a two-dimensional grid of tiles with columns alternating between logic performing tiles and one or more associated interconnect tiles, also called switch boxes (Figure 2.2). The logic performing tiles contain a collection of different functional components. Examples of tiles include the CLBs, DSPs (hard multipliers), and BRAMs (on-chip memories).

In general, each column consists of a single type of tile replicated from the bottom to the top of the chip. With the exception of a few dedicated carry chain wires which connect adjacent tiles

(not shown), all routing between the logic performing tiles in the device is accomplished through the switch boxes.

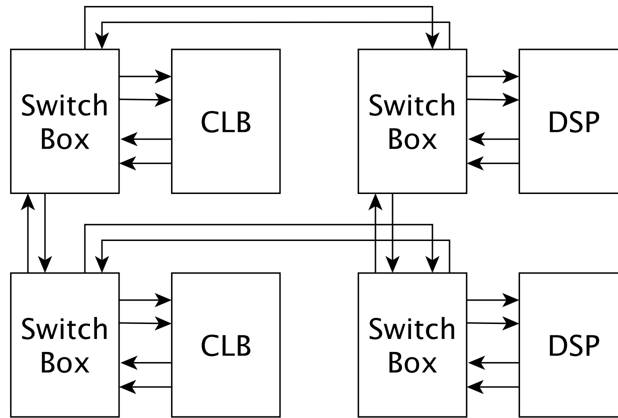


Figure 2.2: Device Arranged as a Grid of Tiles

Switch boxes contain the wires that make up the general routing fabric of the FPGA. Each switch box is made up of wires that a) connect the switch box to its associated logic performing tile, b) connect the switch box to other nearby switch boxes, and c) connect to other wires within the switch box (Figure 2.1.1). Connections between wires within the switch box can be controlled by enabling or disabling a programmable interconnect point (PIP). The PIPs provide most of the routing flexibility of the FPGA.

Unlike other FPGA architectures [19,20], Xilinx FPGAs do not contain an explicit routing crossbar inside the logic performing tiles. Instead, outputs from the logic performing tiles must all exit to the switch box, even if their nets immediately return to the same tile they came from. This implies that the resources for routing elements in the same CLB (local routing resources) are shared with the global routing resources.

Most connections coming from the logic performing tiles connect to wires that leave the switch box. However, the switch box does provide a limited number of routing paths that connect the outputs of the logic performing tiles directly back to their inputs without requiring the net to leave and return to the switch box. In the case of slices, which will be discussed ahead, many of these connections connect the two different slices in the CLBs. A packer that makes use of these connections can create circuits that use fewer wires and run faster.

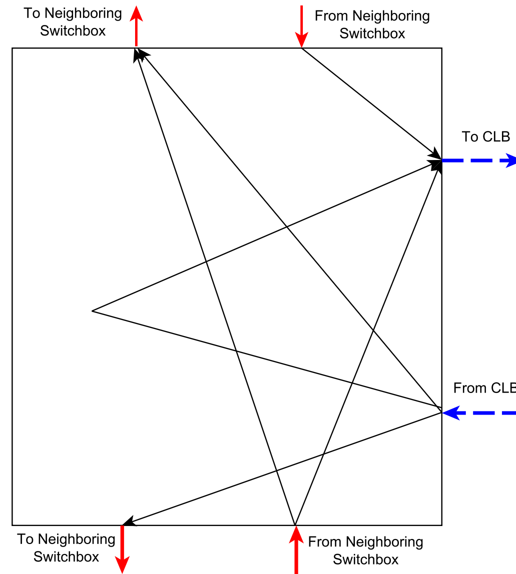


Figure 2.3: Connectivity Inside a Switch Box

2.1.2 Sites

Each of the logic performing tiles is comprised of one or more sites. The sites in turn are composed of a number of BELs connected by muxes. All of the functional circuitry is contained within the sites with the containing tile only connecting wires from the switch boxes to the sites. The functionality of a site is determined by its type, with the type determining the pins, BELs, routing and possible configurations of the site. For example, sites of type SLICEL and SLICEM contain LUTs, FFs and other associated logic, while sites of type IOB contain the input/output circuitry for the FPGA. In some cases, a site can be represented as one of multiple possible site types. For example, some slices in the device can be of type SLICEM, which allows the LUTs to be configured as LUTRAMs, or as SLICEL, which does not allow LUTs to operate as LUTRAMs.

In the Virtex 6 architecture which is used in this work, many of the sites, such as the DSPs and BRAMs, consist of a single BEL with a set of configuration properties. The configuration properties of these BELs are determined by the synthesis and technology-mapping tools and packing these resources into a site is trivial. The most common sites, the *slices*, however, have their contents exposed and the selection and arrangement of the LUTs, FFs and other resources in slice can be determined by the packer.

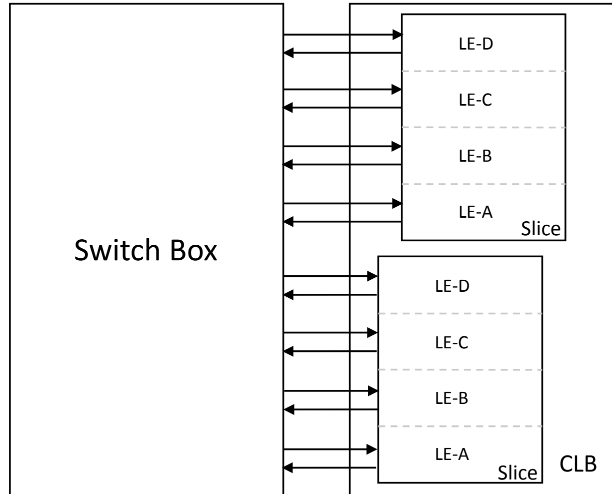


Figure 2.4: A Xilinx Virtex 6 CLB and Adjacent Switch Box

Slices

Slices are sites in Xilinx FPGAs which contain the LUT/FF pairs. In recent Xilinx FPGAs, slices also contain other structures, such as carry chain circuitry and other circuitry, all connected together by an intricate routing environment. In Virtex 6, each CLB tile contains two independent slices. All routing between the two slices must pass through the adjacent switch box. Each slice in turn contains four similar, but not identical, units of closely coupled resources referred to as logic elements (LEs). The Virtex 6 slice and CLB structure and the LE are shown in Figures 2.4 and 2.5, respectively.

The features in each LE include:

- A 6-input LUT (6-LUT) which can be treated as two 5-input LUTs (5-LUTs) sharing the lower five pins.
- LUTRAM capabilities for the LUTs in SLICEM type slices. When a LUT is configured as a LUTRAM, the LUT can operate as either a 32 deep by 1 bit wide RAM or as a 16-bit shift register (SRL). The LUTs in the SLICEMs are also connected with additional routing facilitating the combining of multiple LUTRAMs into larger or more complex RAM or shift register types.

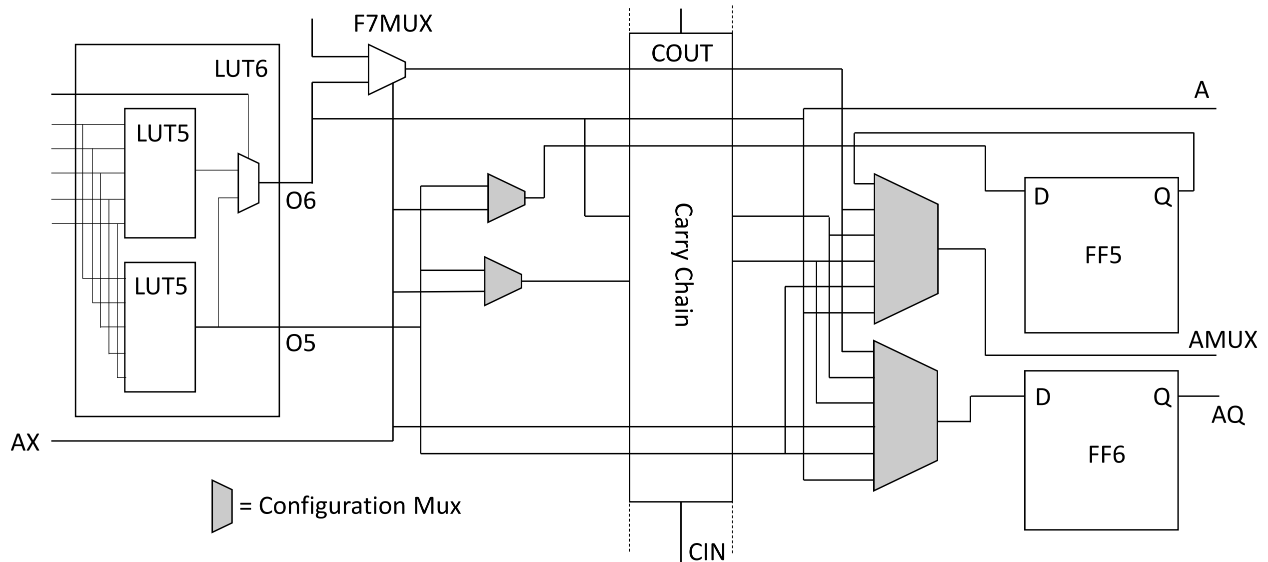


Figure 2.5: A Xilinx Virtex 6 Logic Element

- An F7 or F8 mux, depending on the LE in the slice, that can combine either two or four 6-LUTs in the slice together to perform a 7 or 8-input logic equation. Each slice contains two F7 muxes which feed into one F8 mux. These muxes couple two adjacent LEs.

- A carry logic component which is shared between all LEs in the slice. This component provides additional circuitry which can be used in conjunction with the LUTs to implement chains of adder circuitry. The carry component supports a carry-in signal and a carry-out signal to chain multiple slices into larger adders. The carry logic could be broken into 4 instances of a 2-to-1 mux and a 2-input XOR gate – one instance for each LE – but we treat it as a single unit to match the representation made available from Xilinx.

- Two FFs, one – the FF5 – with a D input configured to come from the O5 LUT output or the AX site input pin and the other – the FF6 – shared between the different components in the LE. The FF6 supports additional functionality including acting as a latch or a 2-input AND gate.

- A dedicated output (A) for the O6 LUT output and an output (AMUX) shared between the different components in the slice. The AMUX output is frequently a point of contention when trying to fill the slice during packing.

To achieve the best possible circuits, it is important for a packer support using each of these features.

2.2 FPGA CAD Flow

2.2.1 Traditional CAD Flow

The FPGA CAD flow converts a user design expressed at the register transfer level (RTL), usually described in a hardware description language, into a placed and routed bitstream for a specific FPGA. A produced circuit is valid if all of the components can be placed and routed using the resources available in the FPGA and if it satisfies all of the user provided constraints. User constraints typically include power and clock rate requirements.

To simplify the task, CAD flows are typically broken up into multiple steps. The steps in the traditional flow are synthesis, technology mapping, packing, placement and routing (left side of Figure 2.6 and Table 2.1). The combination of these steps is referred to as the *implementation* process and the resulting circuit as the implemented design.

Table 2.1: CAD Flow Stage Inputs and Outputs

Step	Input	Output
Synthesis	RTL netlist	netlist of gates
Tech Mapping	netlist of gates	netlist of cells
Packing	netlist of cells	netlist of clusters
Placement	netlist of clusters	placed circuit
Routing	placed circuit	routed circuit
BitGen	routed circuit	bitstream

Synthesis is the first step in the CAD flow and involves translating a design expressed in RTL into a gate level netlist. **Technology Mapping** translates the gate level netlist from synthesis into a netlist of LUTs, FFs, and other specialized components present on the FPGA. In this work, the nodes of the netlist produced by the technology mapper are called *cells*. Each cell performs a different logical or memory function and can be placed onto a single BEL in the device.

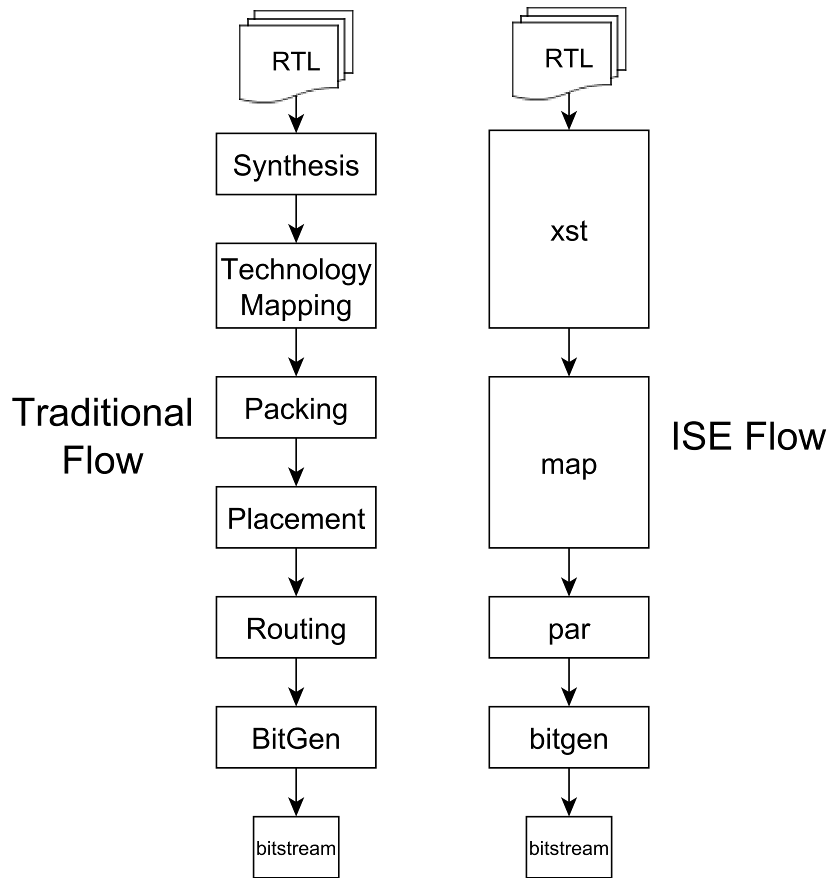


Figure 2.6: Correspondence Between the Traditional and Xilinx ISE CAD Flows

Packing groups the *cells* into structures called *clusters*. Clusters represent an unplaced, logical instance of a physical structure in the FPGA into which cells are packed. The cells are packed into relative locations within the cluster determined by the associated physical structure for the cluster. Possible physical structures that can be represented by a cluster include, but are not necessarily limited to, the different sites and tiles in a device. With the exception of the experiment in Section 5.3, clusters in this work are based on the sites in the device.

Packing serves as an early localized placement phase in the flow and uses algorithms crafted for assembling clusters of related logic [18]. Packing serves three purposes: first, it identifies and groups related logic into a single logical group to reduce the amount of routing between different tiles; second, it separates the rich intersite routing from the constrained routing environment inside the slices; and last, it ensures the many architectural requirements of the slices are satisfied for the

circuit. The second and third purposes simplify the job of the subsequent global placer allowing it to perform its role more efficiently .

Placement finds physical locations on the FPGA for each of the clusters created during packing. Simulated annealing is a popular algorithm used for placement, though analytical placement algorithms have been proposed that scale better with the increasing sizes of FPGAs [21, 22]. **Routing** finds valid, non-conflicting paths in the device's programmable routing network connecting net sources to their sinks. **Bitgen** completes the flow by converting a placed-and-routed circuit into a bitstream that can be programmed into an FPGA's memory to configure the device to perform the desired function.

This research focuses primarily on maximizing the clock frequency of the circuits. This is for two reasons: first, many designs focus on throughput and the ability of the tools to achieve a faster clock rate allows the designer to increase the operating frequency of the design. Second, the ability to find a better circuit improves the chances that a provided set of constraints can be satisfied. Achieving the desired operating frequency is one of the hardest challenges in circuit design and the better the solution the tools can find, the faster a designer can complete his or her circuit.

2.2.2 Xilinx ISE CAD Flow

Prior to its release of Vivado, Xilinx used its ISE software suite for its CAD flow. This flow generally follows the traditional FPGA CAD flow but occasionally merges multiple steps into a single process. The algorithms used in each process are treated as trade secrets and are not public knowledge. The ISE CAD flow and its general relationship to the traditional CAD flow is shown in Figure 2.6. The ISE flow diverges from the traditional flow in that ISE groups synthesis and technology mapping into a single *xst* executable and also groups packing and placement into a single *map* executable. *par* routes the placed designs.

Prior to Virtex 5, ISE contained distinct packing and placement phases with packing being performed in *map* and placement done in *par*. While placement for newer devices has been moved into *map*, the old placer in *par* can still be used to re-place circuits targeted at the newer devices though the tool reports that the feature is deprecated. The work presented in this dissertation takes advantage of the placer in *par* to avoid using a custom placer in most cases.

2.2.3 Academic CAD Frameworks

Commercial CAD tools are almost always proprietary with their techniques and frameworks not being available to the academic community. To support open research into FPGA CAD, a few public frameworks have been developed and released that model FPGAs with varying degrees of capabilities. These frameworks provide architectural descriptions necessary for developing automated CAD algorithms.

Versatile Place and Route

Versatile Place and Route (VPR) [1] is one such framework which supports both FPGA architecture and CAD research. VPR provides users the ability to describe FPGA architectures with various amounts of available routing and sizes of CLBs and a suite of CAD tools to implement designs onto these architectures. This suite of CAD tools includes packing, placement, and routing algorithms but contains no automated front-end synthesis. With the framework, researchers can evaluate the effects of changes to the architectures such as varying the number of routing tracks leaving a tile or the number of LEs in the CLBs and can explore new CAD algorithms.

VPR is, however, limited to only representing logic blocks consisting basic LUT/FF pairs. This has prevented VPR from representing commercial architectures; limiting it to exploring only basic FPGA architectures. Missing from the basic architectures supported by VPR are carry chains, specialized components and the routing connecting these features. The lack of these features meant that packing algorithms targeting VPR architecture only needed to identify which LUT/FF pairs should be packed together without needing to support other components are determining if such blocks even could legally be packed together. Nevertheless, the availability of VPR, along with its accompanying CAD suite, has led to it serving as the backbone to academic FPGA CAD research for several years [23–26].

Verilog-to-Routing

In recent years, the Verilog-to-Routing (VTR) framework [27], an update to VPR, has been released to address the increasing disconnect between the FPGA model in VPR and commercial

FPGAs. The update adds support for modeling many of the complex hardware features found in current FPGAs and a CAD flow to support these features. This release includes:

- the ability to represent advanced features including “fracturable” LUTs (LUTs that can be treated as a single larger input LUT or as multiple smaller input LUTs), carry chains, and arbitrary routing in the logic blocks,
- integration with the ODIN-II [28] and ABC [29] synthesis and technology-mapping tools, and
- back-end implementation tools which support for the complex features found in the logic blocks.

This update substantially improves the ability of VTR to support recent architectures. However, VTR still does not support full integration with any commercial CAD flows. Further, VTR cannot fully model Xilinx architectures without extensive modifications.

Virtex 6 in VTR

Hung et al., in [30] and [31] describe work towards representing a Virtex 6 device in VTR and integrating the VTR CAD flow with Xilinx ISE. As part of this work, Hung created a VTR architectural description of a subset of the Virtex 6 FPGA, implemented designs on this architecture using the VTR CAD flow and imported the designs into the ISE tool chain.

This work had the following limitations. First, the VTR portion of the flow could not accept a Xilinx synthesized netlist. The input had to come from VTR’s own front-end synthesis. Hung found that VTR produced a the lower quality synthesis than Xilinx and this lower quality synthesis result accounted for a significant deterioration in the average critical path delay [31]. Second, the modeled slice lacked important features including:

- the F7/F8 muxes (see Figure 2.5),
- the LUTRAM and SRL capabilities of the SLICEMs, and
- the slice clock enable and set/reset signals.

While these device features likely could be described in VTR's architectural model, they are not supported by VTR's synthesis and therefore would be unavailable to the rest of the flow even if they were implemented. Features that VTR's synthesis tool cannot target include the LUTRAM capabilities of the LUTs and the F7/F8 muxes. Making use of these features is important in reducing the area and delay required to implement a design. For example, a 7-input function can be implemented with two LUTs and an F7 mux but requires three LUTs if the F7 mux is not used – a 50% increase in area. In addition, the lack of these features masks many additional challenges inherent in packing commercial CLB structures such as verifying that incompatible features are not used within the same slice.

The Xilinx synthesis output contain cells that directly map to these features. As such, to fully integrate a custom packer into the ISE flow, the packer must be able to properly accept and handle these features.

2.2.4 Academic Packing Algorithms

Over the years, researchers have proposed many different packing algorithms for FPGAs. Most of these packing algorithms use either a seed-based or a partitioning-based heuristic. In the seed-based approach, clusters are constructed by seeding a new cluster with an unpacked cell and then greedily filling the cluster until it is complete. These algorithms are useful as they have relatively simple implementations and are easy to modify and explore new optimization criteria.

The VPack and T-VPack packing algorithms [32] from the VPR CAD suite are seed-based algorithms. These algorithms work by first packing the LUTs and FFs into basic LUT/FF LE pairs and assembling the clusters from these LE pairs. Many published packing algorithms build on these algorithms by altering the metrics used for identifying good seeds and LEs to add to the cluster [33–36].

In partitioning-based packing algorithms, the design is repeatedly split into smaller partitions until the partitions can fit into the sites. In contrast to seed-based packing, partitioning algorithms provide a global approach to packing and can often yield circuits with less wire length than seed-based algorithms [37] [38]. While partitioning algorithms work well when packing simple LEs into clusters, these algorithms struggle to handle the constraints present in commercial devices.

Packing for More Complex FPGAs

Most publicly available research into packing algorithms has been based on logic blocks comprised of the basic LEs found in VPR. Recent commercial FPGAs, however, use logic blocks which include structures such as carry chains, multiple FFs per LE, dual-output "fracturable" LUTs and high-input multiplexers connecting LEs. [39] explores how the use of these complex features can improve general circuit quality.

The AAPack algorithm [40] is an update to T-VPack included in the release of VTR to handle the more complex logic blocks supported by the VTR architectural model. AAPack, like T-VPack, uses a seed-based packing heuristic. However, instead of initially building the simple LE structures used in VPR, AAPack is designed to assemble its clusters from a set of cells. One of VTR's primary focuses is on being compatible with the large number of complex cluster structures describable by VTR's architectural description capabilities.

The AAPack algorithm works by seeding a cluster with a single cell and then greedily filling the cluster with related cells. When a cell is absorbed into the cluster, the algorithm must confirm that the elements inside the cluster can be legally routed. Verifying that valid routes exist for all nets in the cluster can be a slow process so AAPack uses a set of techniques such as speculative packing and pin counting to avoid performing a full routing feasibility check after packing each cell whenever possible. These optimizations are described in [18].

As part of my research, I have found that difficulties emerge when applying AAPack to Xilinx architectures. First, AAPack does not contain any mechanism for checking for device specific requirements. For example, with Virtex 6, FFs can be either rising edge or falling edge sensitive, but all FFs in a slice must be configured the same. AAPack contains no built-in means of enforcing this rule.

Second, while speculative packing and pin counting is an improvement over conventional techniques for determining if a cluster can be legally routed, the complexity of the routing inside the slices lead to pin counting mispredicting routability at a high rate limiting its usefulness. This is further discussed in Section 4.3.

Last, to support some structures in the slices, the packer must enter into an illegal state before resolving to a legal state. AAPack supports "prepacking" that will enable predefined patterns to be defined to ensure multiple cells are packed together, but this requires the user to identify all

such cases and moves some of the decision making outside of the main packing algorithm. The packer described in this work instead uses a conditional mode described in Section 4.2.3 to support these structures without needing to predefine each of them.

In contrast to the pure seed-based approach of AAPack, [41] merges partitioning-based packing and seed-based packing to pack VTR-described FPGAs. The proposed algorithm initially partitions the design into smaller units and then uses AAPack to pack the partitions into clusters. This allows a seed-based packer to handle the constraints on the slices while using a partitioning algorithm to split the cells into clusters. This approach also allows for increased parallelism that is difficult to achieve with a seed-based algorithm.

Combining Packing and Placement

Most academic CAD flows separate packing and placing into independent steps. Some research has looked at either guiding the packing with placement information or at modifying the packing during placement. In [42], a simulated-annealing placer is augmented with an additional move type that swaps LEs. This allows the placer to move LEs that hinder the placer from finding better results. The authors extend this work in [43] to include duplication of logic during placement leading to modest improvements in maximum clock frequency. Lastly, [44] improves T-VPack by guiding the packer with predicted placement information obtained from a quick, low-quality placer.

2.3 Xilinx Design Language and RapidSmith

Ideally, academic CAD tools should be integratable into commercial flows. This would enable the tools to make use of the different tools available in the commercial toolchains and allow for generation of bitstreams of designs created using the academic tools.

However, compatibility with commercial CAD tools and architectures has been hampered by the closed nature of commercial tools. Xilinx ISE, though, exposes much of the information stored in its proprietary device and circuit description files through the Xilinx Design Language (XDL) tools. Through XDL tools, users can obtain information about the architecture of a Xilinx FPGA in a text-based XDLRC format and convert both partially and fully-implemented designs stored in the proprietary native circuit description (NCD) format to a human readable XDL format

and back. Combined, these tools have enabled researchers to integrate custom CAD tools into the Xilinx flow and produce bitstreams of circuits employing their custom modifications [6, 9].

2.3.1 XDLRC File Format

The XDL tools expose the components of a Xilinx FPGA and its layout in the XDLRC format. This format is human-readable and lists every tile, site and wire in a device along with their positions in an x-y grid. Additionally, it describes the BELs, muxes and other configurations in the sites in a *primitive_defs* section. The XDLRC format, however, does not provide any delay information for the wires or components in the device. A portion of an XDLRC file is shown in Figure 2.7.

```
(tiles 211 179
  (tile 2 26 CLBLM_X9Y198 CLBLM 2
    (primitive_site SLICE_X12Y198 SLICEM internal 50
      (pinwire A1 input CLBLM_M_A1)
      (pinwire A2 input CLBLM_M_A2)
    )
    (primitive_site SLICE_X13Y198 SLICEL internal 45
      (pinwire A1 input CLBLM_L_A1)
      (pinwire A2 input CLBLM_L_A2)
    )
    (wire CLBLM_BYP_B0 1
      (conn INT_X9Y198 BYP_B0)
    )
    (pip CLBLM_X9Y198 CLBLM_BYP_B0 -> CLBLM_L_AX)
  )
  ...
)
```

Figure 2.7: Sample of an XDLRC Device Description

2.3.2 XDL

The XDL tools expose the design netlist of a technology-mapped user design in the XDL format. The XDL format is a human-readable representation of the Xilinx NCD circuit description format and presents designs as a set of instances interconnected by nets. Designs in XDL format may optionally contain placement and routing information [2].

The instances in XDL are logical representations of structures that correspond to sites on the FPGA. The configurations of LUTs and flip-flops in the design are stored as text attributes in the configuration of the instances. Figure 2.8b presents a simple XDL netlist of the 4-bit adder shown

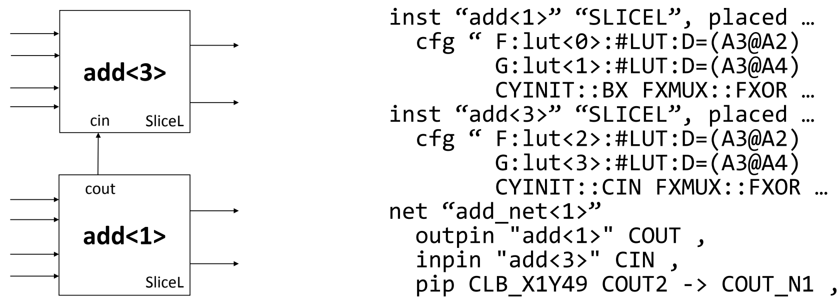


Figure 2.8: A 4-Bit Adder Netlist (a) and its XDL Representation (b)

in Figure 2.8a which contains two instances (slices) connected by a carry chain. The attributes F and G in the XDL are attributes which describe LUTs in the slices and their configurations and the attributes CYINIT and FXMUX describe the routing in the slices.

2.3.3 XDL/ISE Interoperability

While XDL is useful for exploring and manipulating Xilinx implemented circuits, the true power of XDL is in its interoperability with Xilinx ISE. The XDL tools provide the ability to convert designs back-and-forth between Xilinx's proprietary NCD format and public XDL format. As shown in Figure 2.9, this conversion can occur either between placement and routing or after routing. If desired, a user could also create their own netlist from scratch to give to Xilinx.

Since Virtex 5, ISE no longer separates packing and placement into different processes. However, Xilinx still provides access to a placer in *par* which can be used to re-place a design which can be enabled by removing the `P3_PLACED` design attribute from the XDL. This adds an additional entry point back into the ISE flow between packing and placement.

The ability to extract designs from different phases of ISE's CAD flow, to make modifications and then to reimport them into ISE opens the way for many novel tools targeted at commercial architectures. These tools include alternative CAD flows, circuit analysis software and post-placement/routing circuit modification techniques. The ability to reimport the design at several possible stages of the CAD flow allows researchers to operate on a design at their desired stage and allow Xilinx to handle all other phases of the flow.

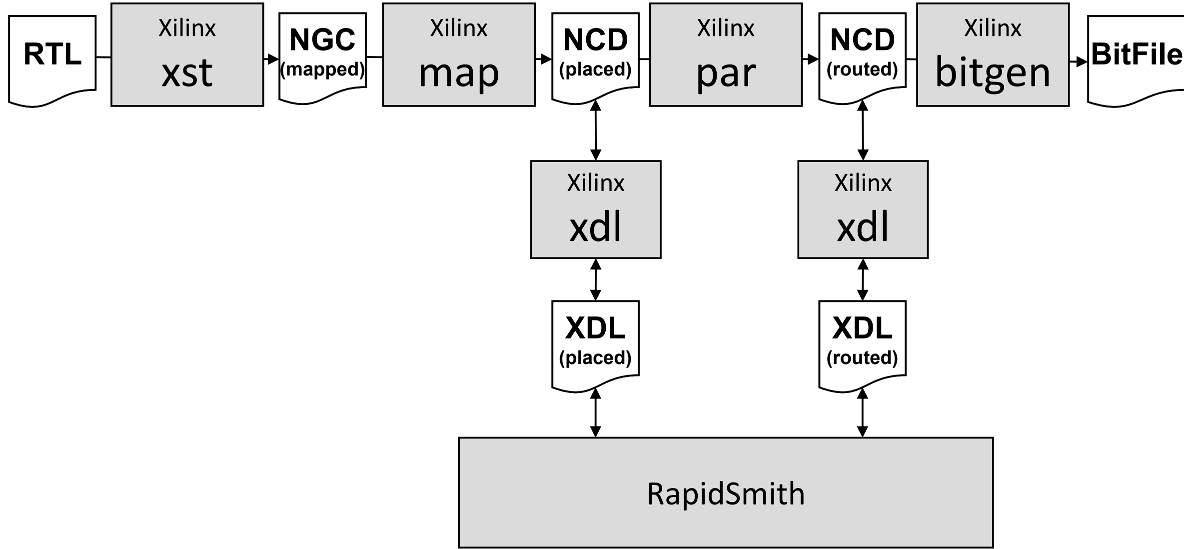


Figure 2.9: Steps in the ISE Tool Chain Where Designs can be Converted to XDL

2.3.4 RapidSmith 1

RapidSmith 1 [3], or RS1 for short, is a Java-based open source framework for viewing and modifying XDL-based netlists and XDLRC device descriptions. Like Torc [45], the RS1 framework enables exploration of new algorithms for and approaches to the CAD flow for Xilinx FPGAs. Such exploration includes, among other things, novel algorithms for floorplanners, placers, and routers, and enables research in areas such as FPGA security [15], reliability [12, 13] and productivity [7, 9, 46].

To permit CAD exploration, the RS1 framework provides both a logical netlist to represent designs and a physical description of the FPGA's resources. The physical device description is obtained from processing the XDLRC device description files and is presented in the Devices API. This API exposes the tiles and sites in the device and their physical locations. Wires and their connections are stored in a graph structure that enables traversing the routing resources of the FPGA. RS1 does not include support for exploring the sub-site structures.

The logical netlist found in the Design package of RS1 enables easy manipulation of designs. RS1 can both import its designs from and export its designs to XDL in both placed and placed-and-routed forms. Like XDL, the netlist structure in RS1 is based on a collection of instances connected by nets. RS1 provides methods for easily creating and modifying these netlists,

placing the instances and routing the nets. These methods make it simple to write sophisticated tools for analyzing, creating, and modifying designs for Xilinx FPGAs.

Designs in RS1 are based on Xilinx architectures and, because of XDL, can be returned to the Xilinx ISE tool flow. By being interoperable with the ISE CAD suite, RS1 allows researchers to focus on the problems of interest and leave other aspects of the flow to the Xilinx tools. By working with designs for real devices, RS1 provides researchers with the flexibility of a research tool while preserving the rigor, detail and accuracy required by a modern FPGA and allows researchers the ability to see their work be realized as bitstreams for physical FPGAs.

Despite its usefulness, RS1 does have a significant limitation; namely, that due to its close relationship with XDL, it only provides visibility down to the site level. While the instances do contain the information stored as text attributes, manipulating these attributes in any significant way can be tedious. Further, the lack of physical information about the device at this level hinders any exploration of tools and algorithms attempting automated manipulations of the BEL-level netlist. This limitation makes using RS1 impractical for creating and modifying the packing of a design.

CHAPTER 3. RAPIDSMITH 2

While the XDL language, with the support of RapidSmith 1, provides the ability to view, create and modify the packing of a design through manipulating the attributes of an instance, this process is error prone and tedious. By themselves, the text attributes in the instances provide little context to the type of resource they represent and to the connectivity between the elements. As such, to directly modify these attributes requires the user to have a deep understanding of the meaning of each of the attributes and the structure of the sites. The lack of a traversable netlist structure describing the connections between these elements also complicates automating modifying the packing of the instances.

To support the creation of a packing algorithm for Xilinx FPGAs, I have updated RS1 to expose the internal circuitry of the sites and decomposed the instances into a netlist of the individual components. This update, called *RapidSmith 2* [17], or RS2 for short, provides the ability to explore the BELs in the sites and the connections between them, a netlist of the individual components in the instance (cells), and a process of converting between this netlist format and XDL. RS2 extends the device structure from RS1 to include the BELs in the sites and the routing connecting them (Figure 3.1). The instance-based netlist from RS1 has been replaced with a new cell-based netlist. Through all of the changes made to RapidSmith, compatibility with XDL is preserved allowing designs to be imported from and exported to Xilinx ISE.

The most recent version of RS2 adds support for Vivado integration. This will enable RapidSmith to continue supporting the latest Xilinx architectures. The source code for the latest release of RS2 is distributed online at
https://github.com/byuccl/RapidSmith2 under the GPLv3 license. Due to Vivado only recently being supported in RS2 and the frequent changes to RS2 as support for Vivado was added, the

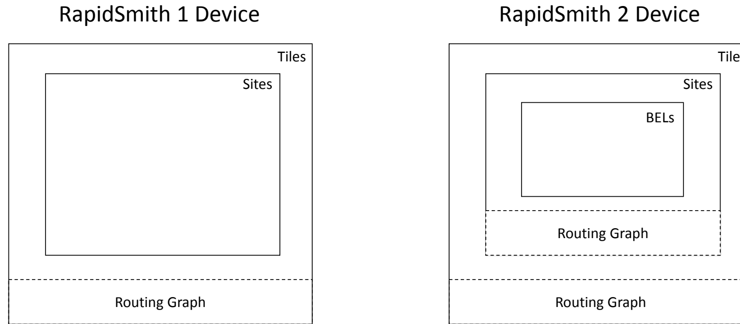


Figure 3.1: The Extended Device Representation in RapidSmith 2

work described in this dissertation uses an older version of RS2. This means that the work in this dissertation focuses only on ISE, XDL, and an ISE compatible architecture.[1]

3.1 Subsite Device Representation

RS2 adds new classes to device representation in RS1 to expose the subsite device structures (the shaded portion in Figure 3.2). The additional classes represent the BELs, wires, and connections that exist within the sites in the Xilinx device hierarchy. These new structures allow users to programmatically explore the components and connections within the sites without the need for an in depth understanding of the structure of the site.

The subsite structure is represented as a collection of BELs for each site connected by a traversable routing graph. This graph provides the connections between the wires, BELs and site pins inside the site and can be traversed using standard graph algorithms. The routing graph is built using the same data structure used to represent the intersite routing in RS1 [47].

In the routing graph, routing muxes in the sites are treated as a set of PIPs each connecting to the same output wire. The pins on the BELs each attach to a wire on the routing graph. Wires that reach the edge of the site connect to site pins which act as the boundary between the subsite and tile level hierarchies. This makes it easy to transition between one hierarchy and the other while providing a distinct break between the two routing graphs.

[1]The differences between the version of RS2 used in this work and the released version are minor and should not affect any of the results in this dissertation. Differences between the versions are listed in Appendix B. This algorithms described in this work are currently being ported to the latest version of RS2 and will be updated to support Vivado and the new FPGA architectures supported by Vivado.

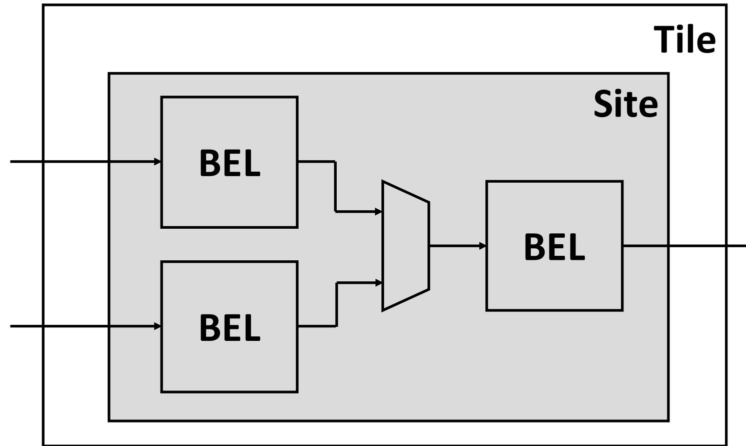


Figure 3.2: The Site Level Hierarchy (shaded) of Xilinx FPGAs

As many BELs and connections within the site will not be used in a normal design, RS2 conserves memory usage by creating a single representation, stored in a *site template*, for each site type in the device. This template contains all of the information about the BELs and routing for its particular site type. Upon requesting a BEL or wire in a site, RS2 will dynamically create the desired object for the site. This limits the subsite resources that exist in memory to only the resources that are used by the design and makes it easier to represent sites that may be represented as different types (refer to Section 2.1.2). When the type of a site is changed, the site simply updates its backing template to the template for the new site type.

3.1.1 Building Site Templates

The site templates are built primarily from the information contained in the `primitive_defs` section of an XDLRC device description. The `primitive_defs` section of the XDLRC file describes the various components, muxes and configurations for each type of site in a device. RS1 built a parse tree of this section but performed no further processing of the elements in this section. To my knowledge, no other publicly available tools provide an extensive break down of the components in this section and their meanings.

The `primitive_defs` section contains a `primitive_def` for each site type in the device. All sites that are configured as that type use the structure and properties described in the corresponding definition. Inside each `primitive_def` is a description of the different components,

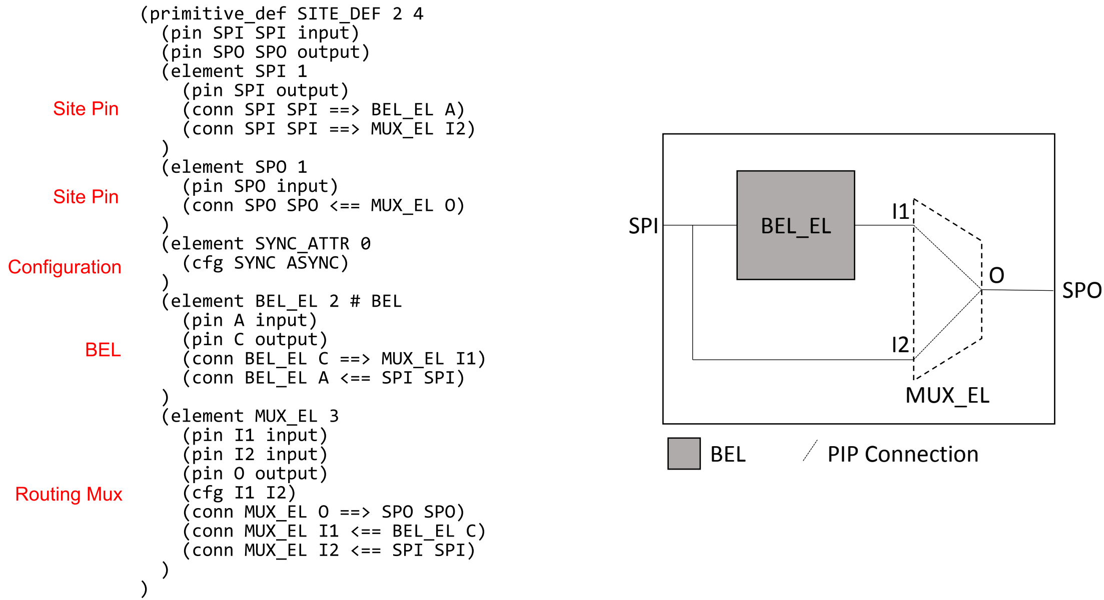


Figure 3.3: An XDLRC Primitive_def (left) with its RapidSmith 2 Representation (right)

pins, and configurations for that site type. Each element describes a unique property or component of the site along with the pins on the component and the connections between the elements, but the exact meaning of each element in the description is not always immediately clear.

RS2 processes each `primitive_def` to build the site templates for the device. Each element in a `primitive_def` is translated into a different structure in the corresponding template. Figure 3.3 shows an example of a `primitive_def` converted into a site template. The elements in the `primitive_def` look similar to one another and XDLRC provides little help in determining what kind of structure each element describes. However, with a little analysis, the elements can be categorized as one of the four types that RS2 translates in a specific way – types for each element in Figure 3.3a are on the left. The element types and the manner in which they are processed are:

- **Site pin** elements (elements SPI and SPO in Figure 3.3) which correspond to pins on the edge of a site. RS2 creates a site pin object for each site pin element. The site pin objects connect to both a wire outside of the site, defined in the description of the site in the XDLRC, and a wire inside the site.

- **Site configuration** elements (element SYNC_ATTR in Figure 3.3) which describe configurable attributes of the BELs or of the site as a whole. While these configuration attributes

are used in the design representation, they provide no useful context for the site templates and are ignored in the site representations.

- **BEL** elements (BEL_EL in Figure 3.3) which describe BELs in the site. RS2 creates a BEL object for each BEL element in a `primitive_def` and connects each pin on the BEL to a unique wire in the routing graph. BELs in RS2 are black boxes and, if needed, it is up to the user to understand the functionality of each BEL.

- **Routing mux** elements (MUX_EL in Figure 3.3) which describe statically-configurable muxes in a site. No distinct mux objects are created to represent the muxes. Instead, mux elements are translated into connections within the routing graph for the site. As with BELs, RS2 connects each pin on the mux to a unique wire. It then creates a PIP connection from each input wire to the output wire in the routing graph.

The site pin, BEL, and routing mux elements in XDLRC each contain a set of `conn` elements which describe how the different elements are connected within the site. RS2 uses these `conn` elements to create the connections between the wires in the routing graph for the template.

While most of the information needed to build the site templates is obtained from the `primitive_defs` section of the XDLRC device description, some additional information is required to fill in missing details and correct some seemingly erroneous descriptions in the XDLRC. This information must be assembled by the user for each supported device family and is stored in a `family_info.xml` document. This additional information provided in this file includes:

- the different site types that each site can be represented as,

- a set of corrections to elements that are mislabeled as BELs in the XDLRC but function as configuration muxes, and

- an identification of a class of muxes, *polarity muxes*, that optionally invert a signal entering the site. These muxes, such as the mux shown in Figure 3.4, are problematic as the inverters on the input pins of the muxes are not explicitly represented, making the two inputs look identical. Furthermore, the configurations for these muxes in the design netlist are stored as a property on the cells that are driven by these muxes. To handle these polarity muxes, RS2

removes the mux elements in the XDLRC and replaces them with configuration elements (which are ignored during the site template creation process).

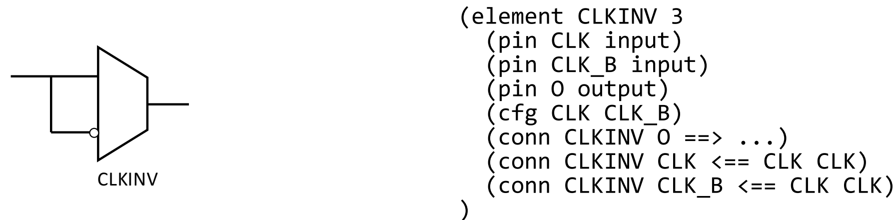


Figure 3.4: A Polarity Mux Schematic (left) and its XDLRC Representation (right)

When building the site templates, RS2 reads in the additional information stored in the `family_info.xml` file and applies it to the templates being built. Additional information on the device representation and the `family_info.xml` file can be found in the RS2 documentation [48].

3.2 Design Netlist

RS2 uses a new netlist structure to represent user designs at the BEL-level (RS1 used a site-level netlist). The RS2 class names and relationships for its logical netlist representation are presented in Figure 3.5. Cells, logical components that can be placed onto BELs, serve as the basis of this netlist (CellDesign). Each cell in the netlist is backed by a *library cell* (LibraryCell) which defines the type, configuration, and valid placement locations for the cell.

All library cells used in a design are contained in a *cell library*. RS2 supports user-defined cell libraries which allows it to accept inputs from different possible input sources (e.g. an ISE, Vivado, or academic synthesis tool's synthesized netlist output). Each cell library is specific to a single FPGA family and is described in an XML file. This XML file contains an entry for each library cell in the cell library, with each entry defining the pins on the corresponding cell, the BELs the corresponding cells can be placed onto, and the mapping between the pins on the cell to the pins on the BELs.

The pins on the cells (CellPins) are connected by nets (CellNets). In contrast to the multi-leveled device representation (tiles, site, BELs), the design netlist in RS2 is a purely flat structure.

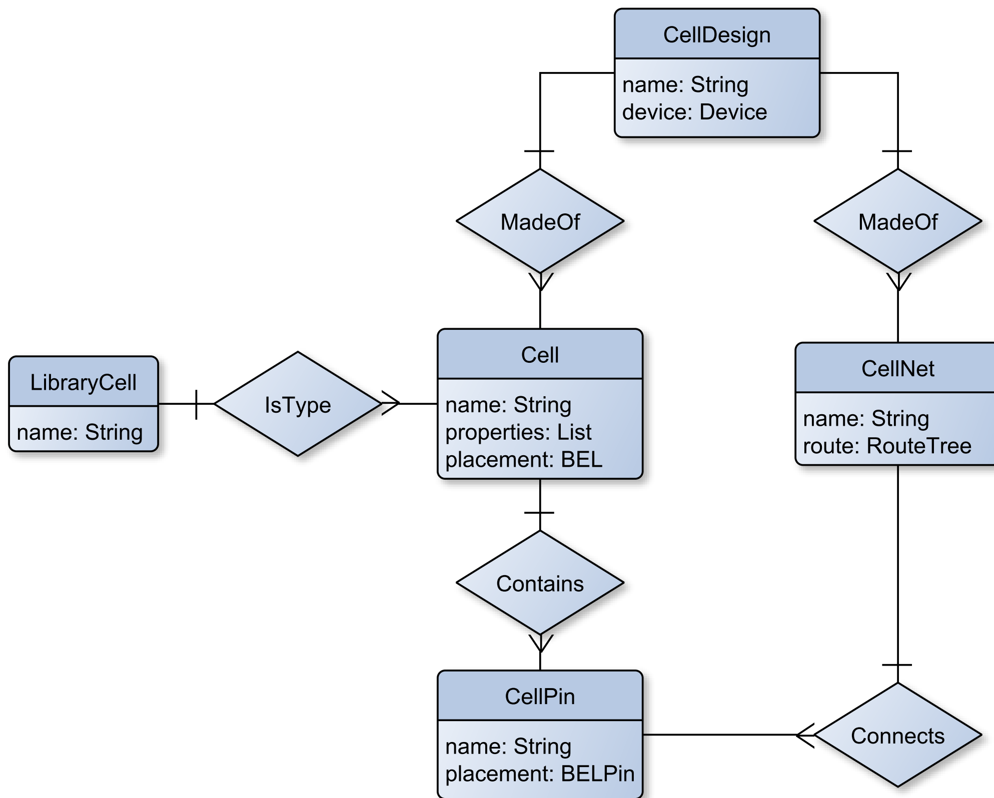


Figure 3.5: Relationship Between RapidSmith 2 Netlist API Classes

The cells and nets in RS2 provide fields to specify their physical placements and routes, respectively. Each cell can be associated with a BEL the cell is placed on. Each net can be associated with a tree structure defining the physical routing for the net.

3.2.1 Route Trees

To simplify exploring and modifying the routing of circuits, RS2 provides a new data structure for representing routing of nets. In RS1, the routing of nets was defined by providing a set of enabled PIPs used in the route. While sufficient to represent the route of a net, this representation was inappropriate for following a route from the source to a sink or vice versa. To improve the ability to traverse the physical routing of the nets in a design, RS2 adds a *route tree* structure for representing routing. Route trees are basic tree structures with nodes representing the wires used in the routes and edges representing the connections between wires. The route trees simplify traversing and building routing in RS2 designs by allowing the routes to be traversed using any

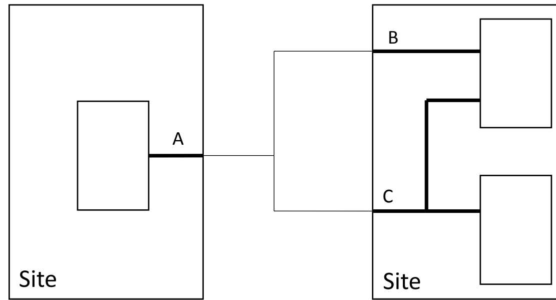


Figure 3.6: Subsite and Intersite Sections of a Route Tree

standard tree traversal algorithm and allowing branches to easily be added or pruned from the tree to modify the route.

To enable packed-but-unplaced designs, multiple route trees can be associated with a given net. This means that multiple independent route trees will exist for different portions of the site-level routing. For example, sections A, B and C of the routing in Figure 3.6 will be individual route trees assigned to the net. If desired, a router can later fill in the intersite routing. This is done by creating a new route tree representing the intersite routing and connecting it to each of the subsite route trees.

3.3 XDL Compatibility

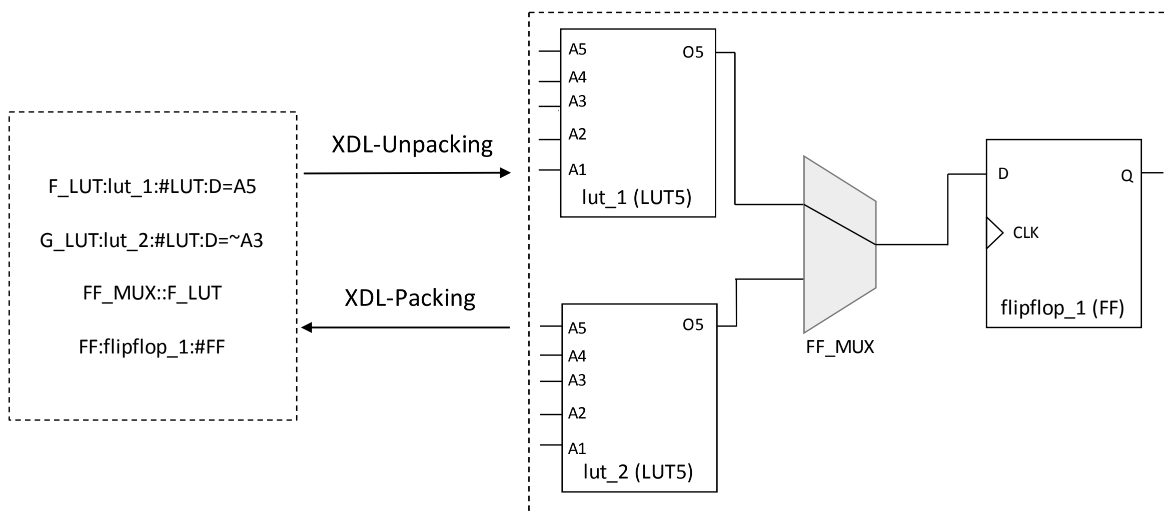


Figure 3.7: Converting Between an XDL Instance and RS2's Cell-Based Netlist

RS2 preserves the same compatibility with XDL that existed in RS1. Converting between XDL and RS2's cell-based netlist is accomplished through the XDL-Packer and XDL-Unpacker tools. The XDL-Unpacker breaks apart the attributes in instances from XDL's netlists into individual cells while the XDL-Packer builds instances from RS2's cell-based netlists.

In XDL-unpacking, the attributes in the XDL instances are translated into the cells, properties, and routing that make up an RS2 netlist. For example, in Figure 3.7, the F_LUT, G_LUT, and FF attributes in the XDL description (left) are translated into three cells (lut_1, lut_2, and flipflop_1 respectively) in the netlist (right) and the FF_MUX attribute is translated into a PIP connecting the F_LUT output to the FF input through the FF_MUX mux in the slice. When complete, the returned netlist will be a functionally equivalent cell-based representation of the original XDL-based netlist.

XDL-Packing, in turn, converts the cell-based netlist from RS2 into an instance-based XDL netlist. The cells and their properties and the intrasite routing connections are translated into corresponding attributes in the XDL instances.

Together, XDL-unpacking and XDL-packing are important for the flows presented later in this dissertation. With ISE, the circuits produced by `map` will be packed and placed. To perform a custom packing on these circuits, the circuit will be XDL-unpacked and packing and placement generated by `map` will be discarded. Once discarded, the circuit will essentially be identical to the netlist generated by `xst`. Figure 3.8 shows how XDL-unpacking and XDL-packing can be used to implement a custom packer in RS2 and integrate it into the Xilinx ISE flow.

Alternatively to accepting a placed XDL netlist from ISE, RS2 could be configured to accept an EDIF netlist provided by the synthesis output. In the case of packing, this would allow a custom packer to directly accept the synthesis results from ISE instead of unplacing and unpacking a placed design coming from `map`. However, the netlist properties of the cell provided by the synthesized EDIF netlist do not match the properties required by XDL. While it would be possible to map the EDIF properties to their corresponding XDL attributes, such a task is beyond the resources available to this project.

With the Vivado support available in the public release of RS2, the cell properties provided by EDIF are preserved through the entire tool chain. As such, when packing with newer devices supported by Vivado, the packer can accept the Xilinx synthesis results as input.

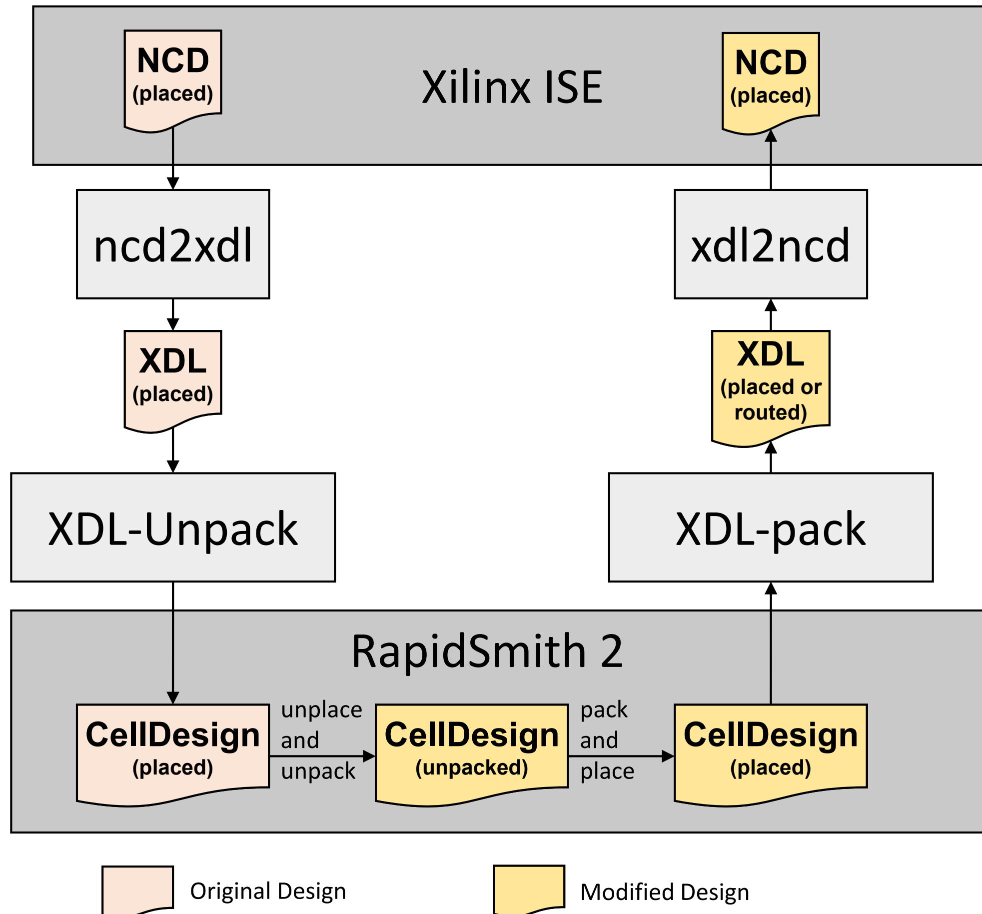


Figure 3.8: Flow for a Custom Packer Using RapidSmith 2

3.3.1 Integrating Custom Packers into ISE with RapidSmith 2

3.3.2 Non-Xilinx Architectures in RapidSmith 2

RS1 and RS2 are designed to target Xilinx architectures. As such, the structures used in RS2 are patterned off of those found in Xilinx architectures – specifically the structures found in the XDLRC descriptions. However, RS2 contains no hard-coded FPGA components or features and should be flexible enough to represent a variety of FPGA architectures. However, these architectures would likely need to be finessed to match the three level hierarchy (tiles, site, BELs) used by Xilinx. At this time, I do not know of any attempts to represent non-Xilinx architectures in either RS1 or RS2 and cannot definitively say whether competing architectures could be represented in RS2.

CHAPTER 4. RSVPACK: A PACKER FOR XILINX FPGAS

In traditional FPGA CAD flows, a packing step is employed to group individual components of a technology-mapped design into relatively-placed structures called clusters. This step acts as a localized placement phase, bundling closely related logic together to shorten routing paths between elements and to reduce the total amount of global routing required for the circuit. Along with abstracting away the constrained routing environment and other complexities of the CLB internals, packing helps to simplify the subsequent global placement.

To address the challenges of packing a commercial architecture, this dissertation presents the RSVPack packing algorithm. The RSVPack algorithm is specifically targeted toward Xilinx FPGAs. By focusing on a particular architecture, the algorithm can be tailored to address the specific challenges and requirements associated with the targeted architecture. To address the specific requirements of each Xilinx family, the algorithm allows for a set of checks to be defined that will enforce the different requirements for the clusters. The work described in this disseration specifically focuses on the Virtex 6 architecture. Due to similarities between successive generations, the algorithm should be portable to more recent Xilinx families by updating the set of checks to address the requirements of the new architecture.

A primary objective of RSVPack is to be intregratable into the Xilinx ISE CAD flow. As such, RSVPack must:

1. accept a synthesized and technology-mapped netlist from ISE,
2. accurately represent the architecture of a Xilinx device, and
3. create a set of packed clusters that ISE can import.

This chapter will describe the RSVPack packing algorithm [49] and will demonstrate its abilities by integrating the algorithm into the Xilinx ISE toolchain. A set of benchmarks will be

implemented using this flow and the results will be analyzed using timing information obtained from ISE.

4.1 RSVPack Overview

The packing algorithm takes as input an unplaced, technology-mapped netlist. In this particular work, the input comes from a Xilinx XDL netlist which has been unplaced and unpacked by RS2 (refer to Section 3.3.1). RSVPack is designed to support all of the components used in this netlist including the LUTRAMs, shift-registers and the F7 and F8 muxes.

The netlist that comes from Xilinx already identifies which resources should be mapped to special purpose hardware, for example, whether a RAM should be implemented using hard block RAMs or as LUTRAMs. The sole role of this packing algorithm is to group these components into clusters.

The RSVPack algorithm uses a seed-based, greedy heuristic similar to the AAPack algorithm [18]. This heuristic leads to a straightforward implementation and allows for simple testing to ensure the generated clusters are legal. However, as is often the case with greedy heuristics, decisions are only locally optimal possibly leading to non-optimal global packing or can even lead to situtations where the remainder of the circuit cannot be legally packed. Care, therefore, must be taken when choosing cells to add to a cluster to avoid using resources that will be required by future clusters.

RSVPack does not have any timing-driven capabilities. This is due to a lack of wire and component delay values for Xilinx devices; Xilinx keeps these values proprietary. While timing information for the devices could be estimated and used to improve RSVPack, performing this task is left as future work.

4.1.1 Pack Units

The clusters produced by the packer are made to be placed on regular, reoccurring components of an FPGA henceforth referred to as *pack units*. RSVPack is designed to work with different sets of pack units based on the different levels of hierarchy of the FPGA. The two most obvious levels of hierarchy for pack units are the tile and site levels. This is because both levels contain

all of the BELs and inflexible interconnect necessary for creating a legal packing. Through experimentation described later in Section 5.3, RSVPack has been found to perform better with site-level (slice) pack units than with tile-level (CLB) pack units.

The pack units used by RSVPack are generated from RS2's device representation and include information on the BELs and routing in the pack units. Templates describing each pack unit are generated prior to running RSVPack and describe:

1. the BELs in the pack unit,
2. the routing connecting the BELs in the pack unit,
3. the BELs in the pack unit that can act as power or ground sources,
4. the wires that enter and leave the pack unit, and
5. the carry chains connections coming into or going from the pack unit.

To simplify validating the routability of clusters, the switch boxes in front of the pack units are treated as full crossbars connecting each output of a pack unit to every input of the pack unit. No cases have been observed where this assumption has led to illegal clusters. The routing configuration inside the switch boxes is determined by the final routing stage of the CAD flow.

4.2 RSVPack Algorithm

4.2.1 Seeding Clusters

The RSVPack algorithm (shown in Algorithm 1) works by repeatedly seeding a new cluster with an unpacked cell and filling the cluster with other unpacked cells until no more cells can be packed into the cluster. The algorithm completes after all cells have been packed into a cluster.

4.2.2 Seeding a Cluster

RSVPack begins each cluster by choosing a seed from the remaining unpacked cells. RSVPack chooses the remaining cell with the most external pins as the seed. In the case of a tie, RSVPack chooses randomly chooses one of the tied cells. This seed could potentially seed valid

Algorithm 1 RSVPack Algorithm

```
1: function PACK_DESIGN(netlist, device)
2:     clusters ← a collection of clusters
3:     while unpacked cells remain do
4:         seed ← select_seed_cell(netlist)
5:         for each pack_unit in device.pack_units
6:             cluster ← MAKE_CLUSTER(pack_unit)
7:             success ← PACK_CLUSTER(cluster, seed)
8:             if success then
9:                 cluster.cost ← COMPUTE_CLUSTER_COST(cluster)
10:                best_cluster ← lower cost of best_cluster and cluster
11:            end if
12:        end for
13:        add best_cluster to clusters
14:    end while
15:    return clusters
16: end function
17:
18: function PACK_CLUSTER(cluster, seed)
19:    if PACK_CELL(cluster, seed) is INVALID then
20:        return INVALID
21:    while more candidates cells to add to cluster do
22:        nextCell ← best nearby cell
23:        if PACK_CELL(cluster, nextCell) is INVALID then
24:            disqualify nextCell
25:    end while
26:    return VALID
27: end function
28:
29: function PACK_CELL(cluster, cell)
30:    while untried BELs for the cell remain do
31:        BEL ← best remaining BEL for cell
32:        place cell at BEL
33:        validate cell/BEL pair
34:        if cluster is valid then
35:            return VALID
36:        else
37:            unplace cell
38:            invalidate BEL as a location for cell
39:        end if
40:    end while
41:    return INVALID            → no legal placement for cell in cluster
42: end function
```

clusters based on multiple pack units – for example, a LUT cell could be the seed for clusters based on a SLICEL or a SLICEM pack unit. Rather than predicting which pack unit will yield the best cluster given the seed, RSVPack creates clusters for all possible pack units for that seed and chooses the highest quality cluster from the resulting valid clusters.

The quality of the completed clusters for a given seed is measured by a cost function which looks at how well the resources in the cluster are utilized and how many unique nets enter the cluster. In the case of slices, this will cause the SLICEL type cluster to be preferred over the SLICEM type cluster when the contents of the clusters are identical. The SLICEL cluster type is preferable to the SLICEM cluster type in these cases because clusters of type SLICEL type can be placed onto both SLICEL and SLICEM type sites. SLICEM type clusters can only be placed on SLICEM type sites.

While this approach can lead to longer pack times, the increase is not prohibitive as each seed is compatible with at most two possible pack units.

Filling the Clusters

For each pack unit to be explored, the algorithm proceeds to fill the cluster by selecting an unpacked cell, choosing a BEL in the cluster, packing the cell at the chosen BEL and repeating this process until the cluster is considered complete (method *pack_cell* in Algorithm 1). A cluster is considered complete when either it is full or no more connected cells are available to pack into the cluster.

RSVPack uses the attraction function from non-timing driven AAPack to determine which cell to add to the cluster – namely, it prioritizes cells that share many nets with the current cluster. More information about this attraction criteria can be found in [40]. In contrast to AAPack, the selection criteria limits its search to cells that are connected to a cell already in the cluster.

Once a cell is selected, the packer chooses a location to place the cell. The BEL selection seeks to minimize the number of global routing resources required to route the cluster. This includes prioritizing BELs which fully absorb nets inside a site. The BEL selection also prioritizes the LUT6 and FF6 BELs in the LEs over the LUT5 and FF5 BELs as the LUT6/FF6 BELs have faster paths leaving the slices. Through experimentation, it has been found that beyond pairing

LUTs and FFs and utilizing the fracturable LUTs in the slices, the particular LE a cell is placed in does not have a significant impact on circuit quality.

After each cell is packed into the cluster, the algorithm ensures that the cluster is legal in its current state or can be modified to be so. The Virtex 6 slices contain many requirements which must be met for the cluster to be legal. Cluster legality is checked through a series of tests. Some of these tests are for rules general to all architectures (such as a routability check) while others are specific to an architecture. Cluster validation tests for the Virtex 6 architecture are described in Section 4.2.3.

If the cell/BEL combination being evaluated is determined to result in an illegal cluster by any of the checks, the cell is unpacked from its location and another BEL is chosen for the cell. This process continues until either a valid BEL is identified for the cell or all possible locations are exhausted, at which point a new cell is selected to try to add to the cluster. The previous cell is invalidated for the current cluster and will not be evaluated for the cluster again.

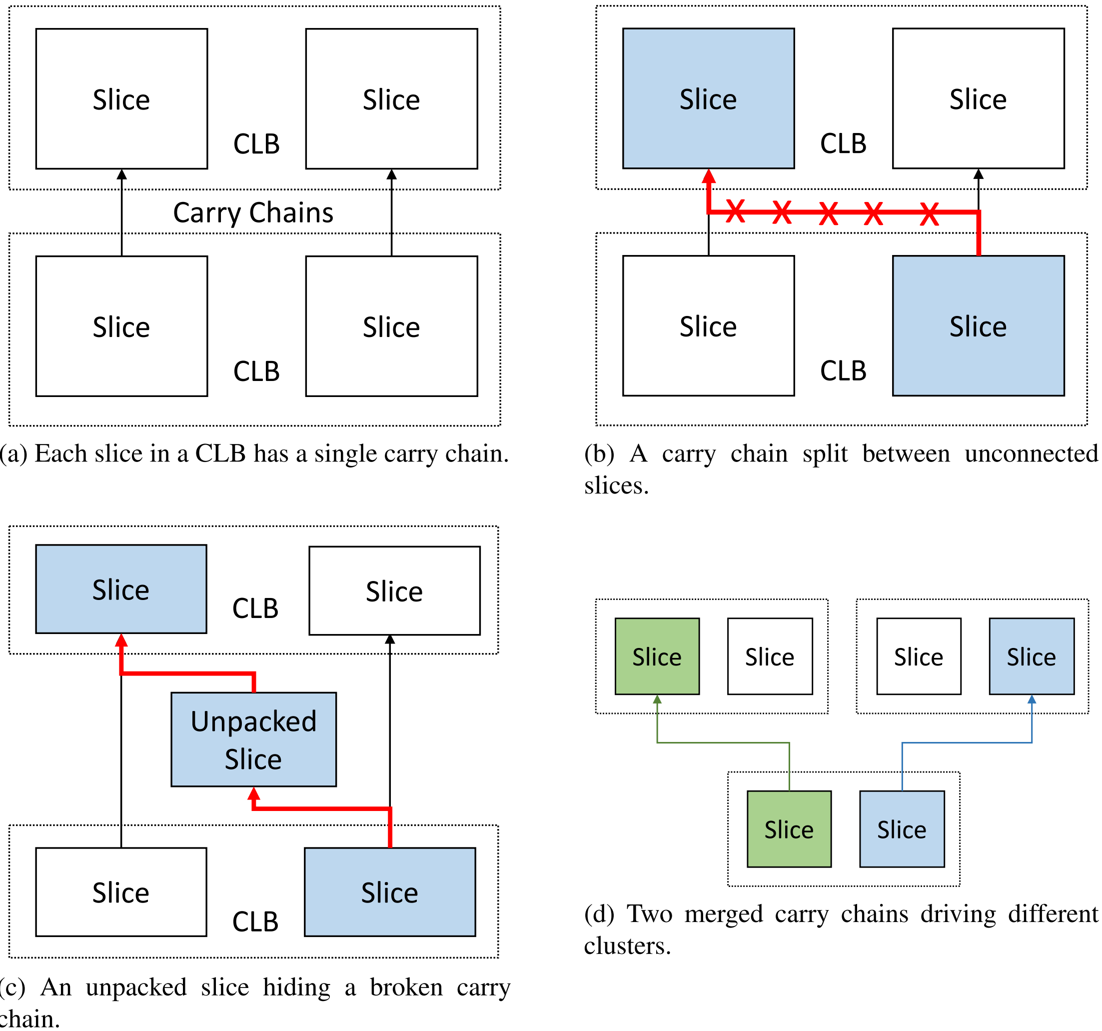
4.2.3 Cluster Legality Validation

Control Sets Consistency

Some properties of the clusters, such as whether the FF reset logic is high or low-asserted, are configurable at the site level instead of being independently configurable for each BEL. These properties, called *control sets*, require that all cells in the site share the same configurations. This check ensures that the packer does not allow cells requiring conflicting properties to be placed in the same site.

Routing Feasibility

Routing feasibility checks ensure that the cluster is packed in a manner that can be routed. This includes checking that all routing can be made including 1) connections between cells in the cluster, 2) connections coming to or from cells in different clusters connected through the general routing and 3) connections between cells forming carry chain connections. To perform



(a) Each slice in a CLB has a single carry chain.

(b) A carry chain split between unconnected slices.

(c) An unpacked slice hiding a broken carry chain.

(d) Two merged carry chains driving different clusters.

Figure 4.1: Possible Invalid Packings for Clusters With Multiple Carry Chains

this check in a timely manner, RSVPack uses a table lookup-based validation algorithm described in Section 4.3.

Carry Chain Preservation

Carry chains provide a fast, dedicated connection between arithmetic units. As the arithmetic chains can be long – a 32-bit adder requires a chain of 8 CLBs – and can often be the critical path in a design, utilizing these paths is critical for creating high-quality circuits. Xilinx uses carry chains in their DSPs, BRAMs and CLB tiles. In the case of the CLB, both slices contain their own distinct carry chain connected to their corresponding slice in an adjacent CLB (see Figure 4.1a).

Carry chains break the notion of independent clusters. Typically, the clusters can be built independent of one another. While optimizing the circuit quality may lead to one cluster impacting the building of another, the clusters could largely be packed independently of one another. In the case of carry chains, however, the building of clusters involved in the chain are dependent on one another as the carry chain leaving one cluster must be able to correctly enter the adjacent cluster. When packing, care must be taken to ensure the elements in the chain are packed such that the chain is compatible between clusters. As part of this, the packing must be configured to use the correct wires leaving the sites.

The work being described in this dissertation works with site level pack units. However, if a tile level pack unit is chosen to pack onto, handling carry chains becomes even more complicated. As the tiles contain multiple carry chains, the cells involved in the chain must be packed so that the chain is preserved when going between different clusters. If not packed correctly (see Figure 4.1b), the circuit could suffer significantly reduced quality or not even be realizable. While the routing feasibility can detect such illegal configurations, the checker only performs a local validation and situations such as in Figures 4.1c-4.1d can arise that the routing feasibility checker is unable to prevent.

As a result, RSVPack uses different strategies to ensure carry chains are properly handled. First, RSVPack will not pack any cell involved in a partially-packed carry chain unless it is adjacent to a previously packed cell. This defers packing carry chain cells until their required positioning can be determined from their previously-packed neighbors and prevents an intermediate unpacked cell in the carry chain from hiding that two other cells in the chain have been packed in incompatible locations such as in 4.1c. Second, when two cells from different carry chains are packed into the same CLB as in the lower half of 4.1d, RSVPack merges the adjacent carry chain cells to ensure that they are packed together later.

LUTRAM Validation

The Virtex 6 architecture is designed to allow for multiple LUTRAM cells to be ganged together to create deeper memories. To support this, each of the LUT BELs in the SLICEM slices handle the upper two bits of the write address pins differently. This limits the allowable locations for the different cells in deeper LUTRAMs.

The input netlist RSVPack uses is a flattened structure – the hierarchical LUTRAM cells were previously broken into individual pieces during XDL-unpacking. When the input design is unpacked, the XDL-unpacker tags each cell in a LUTRAM with properties indicating which LUTRAM the cell is a part of and the additional placement requirements of the cell.

This check ensures that LUTRAMs are appropriately assembled. In addition to making sure each LUTRAM cell is packed onto a valid LUT BEL, this check also ensures that all cells in a single LUTRAM are packed into the same slice. This is necessary as the write address inputs for all cells in the slice are shared with the read address inputs on the D-LUT. Last, this test checks that the D6LUT is occupied when any of the cells in the slice are used as LUTRAMs – a requirement of the architecture.

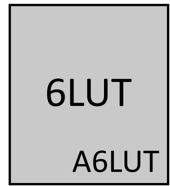
Fracturable LUT Usage

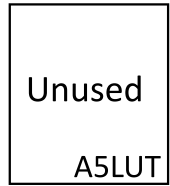
The LUTs in the Virtex 6 can either be used as a single 6-input LUT or as two 5-input LUTs. XDL, and hence RSVPack, represents the LUTs as two BELs, a 6LUT BEL and a 5LUT BEL. The 5LUT can only be used as a 5-input LUT while the 6LUT can be used as either a 5-input or a 6-input LUT. When the 5LUT BEL is used, though, the 6LUT can only be configured as a 5-input LUT.

This rule guarantees that the 6LUT BEL is not used as a 6-input LUT when the LUT5 BEL is used. Additionally, it makes sure that the LUT5 and LUT6 BELs are both configured either as standard LUTs or as LUTRAMs. Figure 4.2 shows examples of legal and illegal usages of the fracturable capabilities of the LUTs.

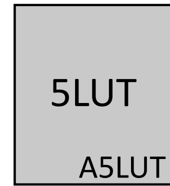
Lookahead Validation

When looking at a single cluster, it is possible that RSVPack will add a cell to a cluster that is needed for another cluster but is not caught by the other checks. Figure 4.3 shows an example of this situation. If, in the example, the DI0 pin is driven by a different source than the CYINIT pin, then the source of the DI0 pin (cell A for future reference) must be packed onto the A5LUT in the same cluster as the carry chain cell.

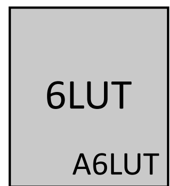


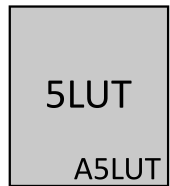


(a) Using only the A6LUT is legal.

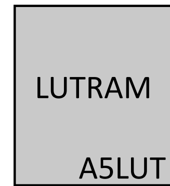


(b) Using both LUTs as 5LUTs is legal.





(c) Pairing a 6LUT and a 5LUT is illegal.



(d) Pairing a LUT and a LUTRAM is illegal.

Figure 4.2: Examples of Legal and Illegal Fracturable LUT Usage

In most cases, the routing feasibility check will detect when the routing requires that a cell be packed with another cell and will force the cells to be merged together. However, in the above case, the routing feasibility checker does not know that the CYINIT pin will require the AX site input pin. Instead, the routing feasibility check sees the AX pin coming from the general routing fabric as a possible route to the DI0 pin. It thus allows cell A to be packed into another cluster as long as cell A can drive the general routing fabric.

In the example where cell A is packed into a different cluster than the carry chain, the routing feasibility check will initially assume that the cluster that cell A is packed into is allowable. This illegal packing will not be found until the accompanying carry chain is packed and no legal

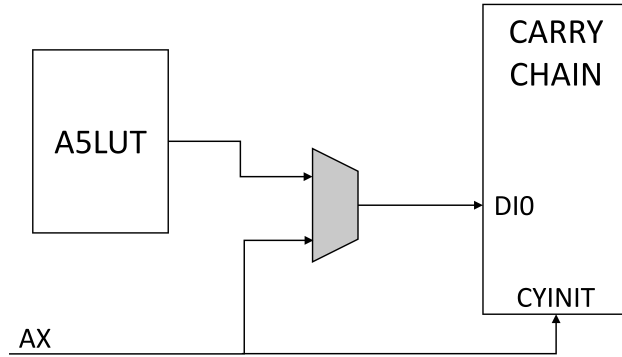


Figure 4.3: Example of an A5LUT that must be Conditionally Packed with a Carry Chain

route is found for both nets driving the CYINIT and DI0 pins. At this point, though, no legal clustering will be possible without undoing the packing in a previously created cluster.

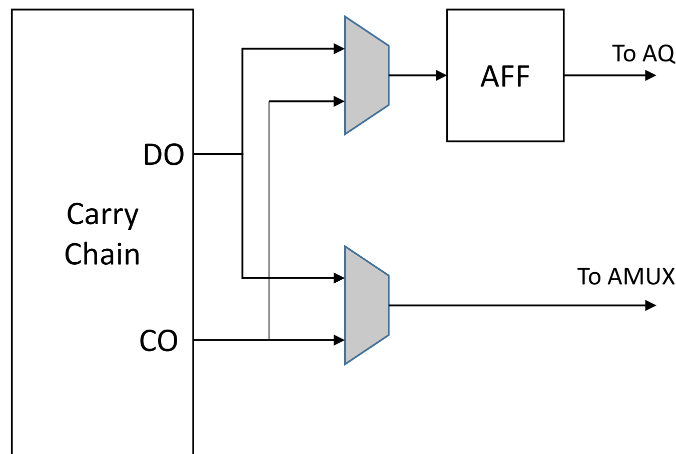


Figure 4.4: Carry chain element using both DO and CO pins must be packed with a FF

I have identified two cases of this occurrence in the Virtex 6 architecture. The first instance is discussed in the above example. The second instance occurs when both the sum and carry pins of a carry chain leave the cluster and the flip-flops cells driven by both of these pins are already packed in another cluster (see Figure 4.4). To handle these cases, a custom check is employed that identifies these two patterns in the design and forces the appropriate cells to be packed together.

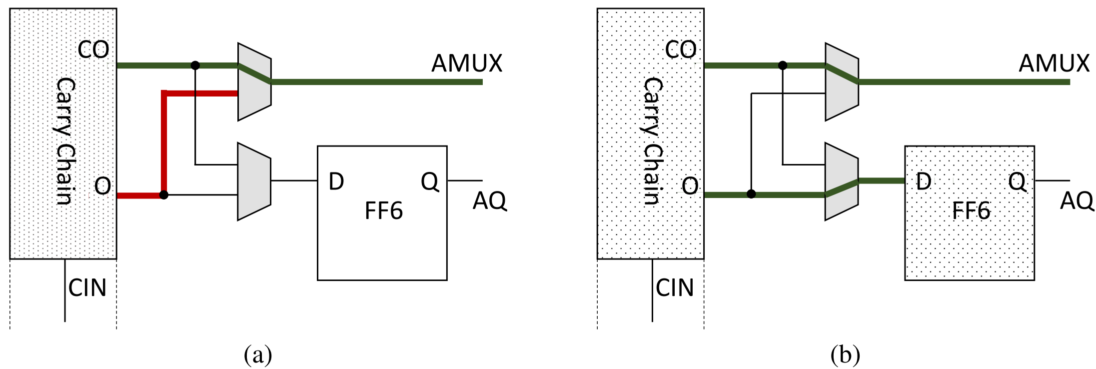


Figure 4.5: Example of Entering and Resolving Conditional Mode

4.2.4 Conditional Mode

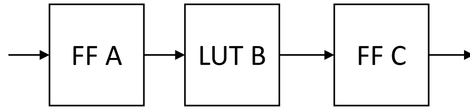
While filling clusters, it is possible for the cluster to enter a state where it is no longer legal in its given configuration; however, it may become legal later in the process as a result of absorbing additional cells. Rather than immediately invalidating this configuration, the algorithm enters into a conditional mode.

For example, upon adding a carry chain element that uses both the O and CO outputs (Figure 4.5a), there will be insufficient outputs to route the cluster. This causes RSVPack to enter conditional mode. Upon adding the flip-flop that is driven by the O output to the cluster (Figure 4.5b), the cluster is again routable and RSVPack returns to its normal mode.

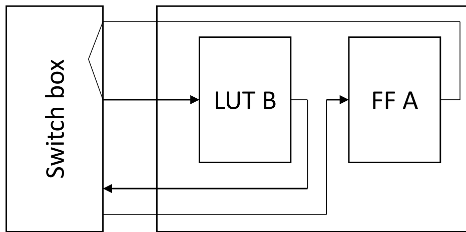
While in conditional mode, the algorithm continues to add cells to the cluster under the assumption that the cluster will eventually return to a legal configuration. If RSVPack is unable to return the cluster to a legal state by absorbing more cells, it will roll back the cluster to a previous checkpoint, invalidate the cell/BEL pair that caused it to enter conditional mode, and resume normal operation. As returning to a legal state may require adding more than one cell to the cluster, the checkpoint and roll back mechanisms operate in a recursive manner.

The recursive nature of the conditional mode mechanism can potentially lead to exponential run times. To limit the time spent in conditional mode, RSVPack guides the cell selection process to cells that will quickly bring the cluster to a legal or illegal state. This guidance is provided by the different validation tests described in section 4.2.3.

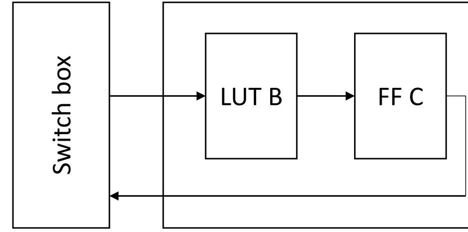
The amount of time RSVPack spends in conditional mode varies widely based on design, ranging from less than 1% in several benchmarks to over 60% in one tested benchmark. Condi-



(a) The example circuit



(b) FF A is packed with LUT B



(c) FF C is packed with LUT B

Figure 4.6: Example of Packing LUT/FF Pairs Together

tional mode resolves to a legal circuit between 1% and 28% of the time, depending on the circuit. Conditional mode is required to guarantee that all designs can be successfully packed.

4.2.5 Manual Pairing of LUTs and FFs

RSVPack uses an additional optimization to improve final circuit quality. The base algorithm described in Section 4.2 does a poor job pairing associated LUTs and FFs into the same LE. This is because the cell selector does not account for where in the cluster a cell can be packed when it is making its decision. This can lead to cases such as the following example occurring.

When LUT B in Figure 4.6a, is packed into a slice, both FFs A and C share a single net with the cluster and will have the same attraction to the cluster. If FF A is chosen (Figure 4.6b), the output of the FF must leave the slice to drive the input of LUT B. This leads two three nets which must route through the switch box. In contrast, if FF C is chosen (Figure 4.6c), the net connecting the output of LUT B and the input of FF C is fully encapsulated in the slice and only two nets must route through the switch box.

Without help, RSVPack does not prioritize FF C over FF A in this example. In this optimization, when a LUT or FF is absorbed into the cluster, the other cell in the LUT/FF pair will also be absorbed into the cluster at the complementary BEL if the location is unused. This helps

with grouping LUT/FF pairs. In the case that the complementary location is already occupied, the LUT/FF pair will be broken apart. This means that it is still possible that some LUT/FF pairs will still be packed into different LEs.

4.3 Table-Lookup Routing Feasibility Improvement

4.3.1 Routing Feasibility: VTR's AAPack versus RSVPack

The Virtex 6 slice contains a highly-specialized, constrained, routing structure. The routability of the cluster must be confirmed each time a cell is added to the cluster. A simple method to verify that a cluster is routable is to simply attempt to route the cluster. If a valid route is found, the cluster is routable; otherwise, the cluster in its current state is not routable. However, running a full router in the inner loop of the algorithm leads to very slow run times. For example, a reentrant version of the popular PathFinder routing algorithm [50] took on average 136ms to build each cluster – this would lead to very long run times.

AAPack's Approach for Feasibility Checking

To avoid the long routing checks, AAPack uses a combination of speculative packing and pin counting to avoid running PathFinder each time a cell is added to a cluster [18]. In AAPack's approach, a cluster is first packed using a fast but optimistic pin counting algorithm to determine cluster routing feasibility. The pin counting algorithm works by comparing the number of nets entering and leaving a cluster and against the number of available input and output pins on the cluster, respectively. This check can be quickly performed but does not guarantee that a valid route exists.

Once a cluster is built, AAPack uses the PathFinder algorithm to determine if the cluster is truly routable or if the pin counting algorithm returned a false positive result. If the cluster is unroutable, the cluster is rebuilt using PathFinder to check routing feasibility after each cell is added.

When pin counting produces correct results, the time to build each cluster using this scheme is significantly reduced. However, as pin counting returns more false positive results, the speedup

quickly reduces with the algorithm effectively building each cluster twice in the worst case. With the Virtex 6 architecture, pin counting, on average, returns a false positive rate of 25%. Most of these false positives revolve around the A and AMUX outputs (refer to Figure 2.5). Pin counting sees both outputs available for use, but the A output is dedicated to the O6 output leaving it inaccessible to the other elements. This limits the improvements gained by using pin counting on the Virtex 6 architecture.

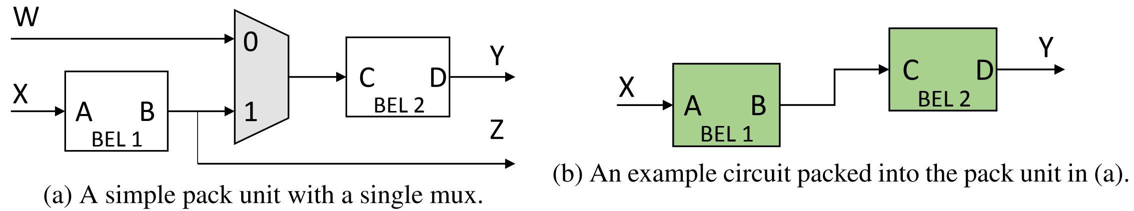
RSVPack's Approach for Feasibility Checking

When analyzing routing times associated with PathFinder, it was found that most of the time is spent determining that a cluster is unroutable. This long run time comes from the nature of using PathFinder to determine routability. PathFinder never actually determines that a cluster is not routable. Instead, that determination is made when Pathfinder is unable to find a valid route after several attempts. As such, while PathFinder is usually quick to find a legal route for a routable cluster, it is slow to determine that a cluster is unroutable. Additionally, attempting to add a cell to a cluster results in an unroutable cluster over 80% of the time, compounding this issue with PathFinder.

In response to this information, a fast algorithm for determining routing was developed. This algorithm utilizes a pre-computed routing feasiblity table to determine routability. This approach quickly compares the current cluster's configuration against all valid routing configurations of the pack unit. To do this, the routing feasibility table is pre-populated with the information about the available routing for each enumeration of configurations as dictated by the various routing mux configurations. As cells are added to a cluster, the resulting cluster is compared against all of the table's entries to see if the cluster's connectivity is compatible with any of those entries. If one or more compatible entries exist, the cluster is routable. If none exist, the cluster is not routable.

Each table entry contains, for each BEL pin in the pack unit, an indication of what pins that pin drives or what pin is driving it. These sources and sinks can be either external source/sink pins or other BEL pins in the pack unit.

Figure 4.7a shows an example pack unit consisting of 2 BELs, two external inputs, two external outputs, and a single routing mux. Since there is a single 2:1 routing mux in the pack unit, the resulting table contains two entries as shown in Table 4.1. The top entry of the table describes



(a) A simple pack unit with a single mux.

(b) An example circuit packed into the pack unit in (a).

Figure 4.7: Example for Determining Routing Feasibility

Table 4.1: Routing Connectivity Lookup Table for Figure 4.7a

Mux CFG	Pin	Source/Sinks
0	A	Driven by X
	C	Driven by W
	B	Drives Z
	D	Drives Y
1	A	Driven by X
	C	Driven by B
	B	Drives C and Z
	D	Drives Y

the pack unit connectivity when the routing mux is configured with a '0' on its select. In this case, the following driver/driven mappings are: D→Y, B→Z, C←W and A←X.

Figure 4.7b shows an example of a packed cluster that is being tested to see if it is routable. In this case, a cell from the design has been placed onto BEL1, another cell has been placed onto BEL2, and the design's netlist implies that BEL1's pin B should be connected to BEL2's pin C.

Examining the top row of Table 4.1 shows that, in this configuration, C←W and B→Z. This contradicts the routing shown in Figure 4.7b and therefore this table entry is marked as invalid. In the second row of the table we see that W is unconnected, B→C and B→Z. This is compatible with the routing required by the packed cluster of Figure 4.7b and therefore this row is considered valid.

A cluster is considered routable if there is at least one entry that is valid in the table. To perform the overall check, the algorithm iterates over all the entries in the table until a valid entry is found. If no valid entry is found, the cluster is unroutable. Advantageously, previously invalidated table entries do not later become valid as new cells are added to the cluster. Thus, the algorithm

increasingly runs faster as new cells are added to the cluster since the number of table entries that need to be checked shrinks over time as entries are marked invalid.

Reducing Table Size with Pin Groups

The total number of entries in the table for a pack unit is the number of possible permutations of routing mux configurations in the pack unit. For a Virtex 6 slice, a naive version of the table may contain over a billion entries. To make the size of the table manageable, the pack unit is broken into groups of independent routing resources called *pin groups*. These pin groups are sets of connected pins, wires and muxes that form a set that is independent from all other pin groups in the pack unit. Any change to the usage of a pin in a group may affect the routability of other pins in the same group, but will not change the routability of any other pin groups.

The pin groups are created by merging together all of the pins and routing muxes that share wires in a cluster. In Figure 4.7a, pins B and C, input W and output Z are connected by wires through the mux and therefore form a pin group. Pins A and D have their own independent sets of wires and therefore each forms its own individual pin group.

Each pin group is represented by its own routing feasibility table. By breaking the pack unit into pin groups, the total number of entries in the tables decreases to a manageable size. For a Virtex 6 SLICEM, the most complex pack unit in the architecture, there are 36 pin groups with a combined 854 table entries. Across the entire device, there are 2135 pin groups with a combined 4580 rows. Most of the tables contain very few entries with the largest table containing 432 entries.

To determine routing feasibility, the previously described algorithm is applied to each pin group in the pack unit in turn. If any affected pin group is found to contain no valid table entries, the entire cluster is considered unroutable. As pin groups are independent of one another, only pin groups that are affected by the addition of the cell need to be checked, further reducing the amount of work required each iteration.

4.3.2 Run Time Improvement

Table 4.2 presents the run times incurred when performing a single routing feasibility check with 1) the table-based feasibility checker and 2) PathFinder. The provided numbers are an average

Table 4.2: Average Runtimes for Each Outcome of Routing Feasibility Checking (in microseconds)

	Routable	Conditional	Unroutable	Overall
PathFinder	230	1008	4117	1577
Table Based	63	160	38	61
Speedup (x)	3.7	6.3	109.1	25.7

across a set of benchmarks. The PathFinder-based checker declares the cluster unroutable if it is unable to find a valid circuit after 6 iterations. For all possible outcomes, the table-based checker is faster but it is substantially faster when the cluster is unroutable – the most common case. Overall, the table-based checker runs 25 times faster than the PathFinder-based checker.

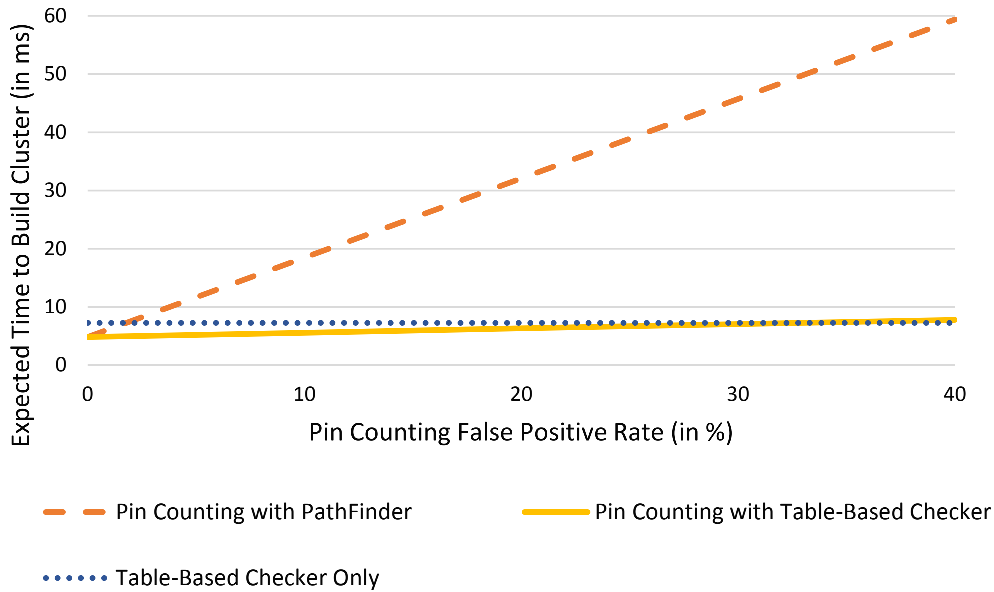


Figure 4.8: Projected Run Times Using Pin Counting

To compare the table-based approach against the speculative packing and pin counting approach used by AAPack, the expected run times of building a cluster using pin counting and speculative packing can be projected using a simple equation. Assuming pin counting runs in zero time, when pin counting returns the correct result, the time to build a cluster is $T_s = B$ where B is the time to build a cluster without any routing feasibility checking. When pin counting yields an incorrect result, the time to build a cluster is $T_f = B + (B + F)$ where F is the cumulative time

spent in a routing feasibility checker on the cluster. In this case, the cluster is built twice, the first time with pin counting (when it failed) and the second time with a full routing feasibility checker.

The average time to build a cluster with speculative packing given a false positive rate on pin counting of R is $T = (1-R) \times T_s + R \times T_f$. Combining the equations, an average time to build a cluster using speculative packing is

$$T = B + R(B + F). \tag{4.1}$$

Figure 4.8 shows the projected growth in time to construct a slice cluster, as computed by Equation 4.1, as the false positive rate from pin counting increases. Values for B and F are taken from average run times of RSVPack over a set of benchmarks. The scheme that combines pin counting and PathFinder incurs more run time than using only the table-based approach as soon as the false positive rate hits 2%. With the observed false positive rate of 25%, the table-based approach is 5 times faster.

If AAPack's speculative approach were to be used with the table-based checker in place of PathFinder, this approach would perform slightly faster than the table-based approach alone up to a 33% false positive rate. However, the improvement in this case would be only marginally faster; using the table-based checker alone is sufficient without speculative packing.

4.4 Identifying Packing Rules for Xilinx Architectures

The sites, the slices in particular, in Xilinx devices have several requirements that must be adhered to to produce legal circuits. Unfortunately, many of these requirements are not well documented and must be found through trial and error. In addition, as previously discussed, the greedy nature of RSVPack can lead to cells that must be packed with another cell to be packed prematurely. This leads to the dependent cell not being able to be legally packed into a cluster causing an error. Both of these situations must be addressed by device specific rules for the packer.

In the process of creating and testing RSVPack, I have discovered many of these architectural requirements. The discovery process typically consisted of trying to pass a benchmark through the entire back-end flow, from packing through routing. In the process of implementing a benchmark, I would encounter a new issue, track down the cause of the issue, and implement a

new rule to detect and prevent an illegal cluster from being created. Once a design was successfully implemented, I would proceed to a new design that used new features that would expose new requirements to address. As different designs use different features, it is necessary to test the rules on several designs to exercise the many components and features of the architecture.

When discovering requirements and creating checks for a new architecture, illegal clusters can be found in two ways. First, the packer may fail to complete due to encountering an unpackable configuration. These cases always occur because a cell that another cell was dependent on was previously packed into another cell. This type of issue can be one of the most difficult to decipher as identifying the source of the error usually requires simulating the construction of the failed cluster. Once the source of the error is identified, the issue is rectified by creating a new check that binds the dependent resources together. This is achieved by making the packing of each resource conditional on the other resource. The conditional mode in RSVPack will then ensure that the dependent cells are packed together.

The second area that can expose issues with the packed design is the conversion of the packed XDL design to the NCD format. When resources on the FPGA are used in an illegal manner, the conversion tool will fail with an error message indicating the XDL instance the issue appeared in and a message describing what failed. These messages are not always clear, but will at minimum direct the user to the location of the error. The errors reported in this conversion stem from a rule of the architecture not being satisified and is the best way that I have found to discover the undocumented architectural rules and are addressed by creating a check that enforces the requirement when building the clusters.

Each architecture will have its own set of unique requirements. However, with Xilinx architectures, many of the architectural requirements apply to multiple families of FPGAs. A list of requirements that I have found for Virtex 6 are listed in Appendix D.

4.5 Integrating RSVPack into the Xilinx ISE CAD Flow

As mentioned, one of the primary goals of the RSVPack algorithm is to make it possible to measure the impact of academic algorithms on commercial Xilinx devices. Figure 4.9 shows four different flows that can be used to implement circuits onto Xilinx FPGAs. The first two flows

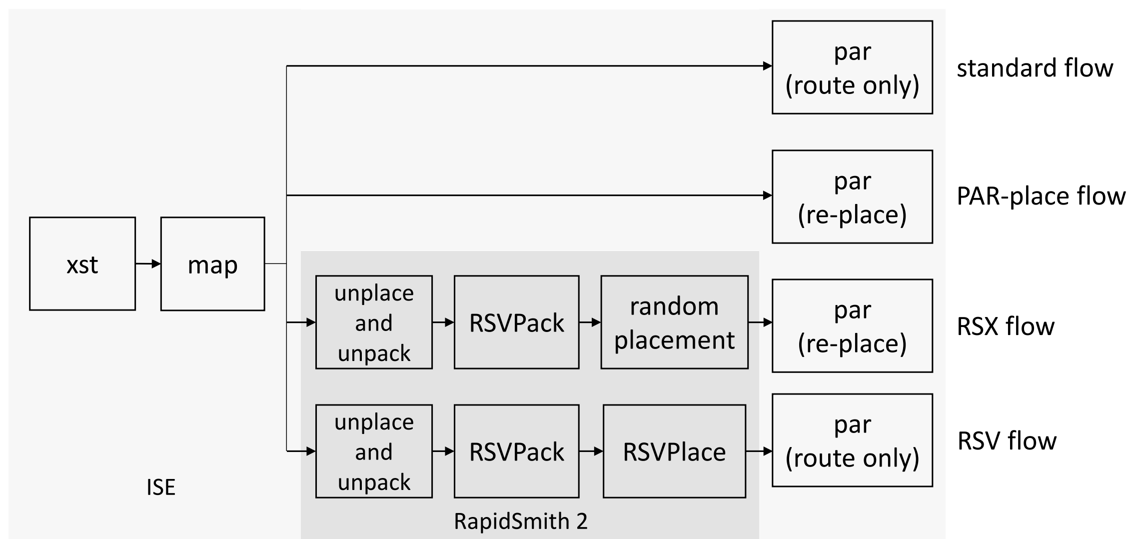


Figure 4.9: Implementation Flows used in this Work

(Standard and PAR-Place flows) use ISE tools to fully implement a design. The last two flows (RSX and RSV flows) uses RS2 to integrate RSVPack into the ISE flow.

4.5.1 Pure ISE Flows

The standard Xilinx flow begins by synthesizing a design using `xst` and then packing and placing the synthesized netlist with `map`. The placed design is then routed by `par`, completing the implementation process. This is the flow typical ISE users use to implement their designs.

While Xilinx no longer officially supports placement in `par`, removing a design attribute in the XDL representation of a design will cause `par` to re-place the design. This work takes advantage of this placer to allow Xilinx to place the RSVPack-ed designs. As the placer is proprietary, the exact algorithm it uses is not known; however, it is known that it performs a site-level placement of a packed design. It does not appear to use the same placement engine found in `map`.

The PAR-place flow (second row) is a modification from the standard flow that uses the placer in `par` to re-place the design without modifying the packing. This is not a typical flow but it is included to provide a direct comparison between a circuit packed by Xilinx, in `map`, and packed by RSVPack.

4.5.2 RSX Flow

The RSX flow is a custom flow utilizing the RSVPack packing algorithm. In this flow, the packing is the only step performed outside of ISE – placement is handled by the placer in `par`. This allows RSVPack to perform the packing of the design while letting Xilinx perform all other parts of the flow.

In the RSX flow, the netlist to be packed is imported into RS2 via the placed design coming from `map`. RS2 immediately unplaces and upacks the design. The unpacked netlist is then packed with RSVPack. `par` can only accept a placed design even when if it is going to re-place the design itself. To support this, the RSX flow applies a quick random placement to the packed design. The random placement also serves as an initial seed to the placer in `par` with `par` returning a different final placement for each initial random placement.

Upon finishing re-placing the design, `par` will immediately route the design. The Xilinx router has the capability to relocate a small number of slices while routing the circuit. However, this movement appears limited to moving entire slices and, after much experimentation, the router has not been observed to make any modifications to the packing of the slices in any of the RSVPack packed designs.

4.5.3 RSV Flow

The RSV flow works similar to the RSX flow but instead of using `par` to place the design, the RSV flow uses an internally developed simulated annealing placer (RSVPlace) to place the design. In this flow, `par` is only used to route the design. RSVPlace does not have access to delay information for the device and uses a half-perimeter wire length cost function in place of delay estimation. As such, circuits implemented with the RSV flow will be significantly lower quality than those implemented with the RSX flow.

Though resulting in lower quality circuits than the RSX flow, the RSV flow is important in enabling future research. First, this flow provides more visibility and control over the placement than is available with the placer in `par`. This enables experimentation that cannot be done using the RSX flow such as exploring the effects of using either smaller or larger pack units in packing and placing designs. Second, RSVPlace plays a role in providing a full suite of CAD tools for

Xilinx devices in RS2. It is especially important in updating RSVPack for newer Xilinx devices supported only by the new Vivado tool suite as Vivado provides no means of placing a user-packed design.

4.6 RSVPack Performance

4.6.1 Benchmark Set

To evaluate RSVPack, each of the flows is used to implement a set of FPGA benchmark designs acquired from a collection of sources including the VTR 7 [51] and Titan23 [52] benchmark suites, Xilinx Coregen [53], and OpenCores.org [54]. These benchmarks are medium sized benchmarks created for implementation on FPGAs. Some benchmarks were modified from their original forms to make them compatible with Xilinx FPGAs. The benchmarks are presented in Table 4.3. Each of the benchmarks is synthesized with `xst` and targeted to use up to 50% of the available LUTs on the device.

Each design is implemented 20 times with each run using a different seed for the placer. A timing constraint to `par` is decreased for each run until the minimum clock period achievable for each benchmark is reached. Reported numbers are from the run with the lowest minimum clock period for each benchmark and flow.

Table 4.3: Benchmarks Used to Analyze RSVPack

Name	Application	Source	Part	5LUTs[1]	% of Device
viterbi	Math	OpenCores	lx75t	11171	12.0
microblaze	Processor	Xilinx	lx75t	16439	17.7
divider	4 Floating Point Dividers	Xilinx	lx75t	20568	22.1
bgm	Finance	VTR	lx75t	26566	28.5
lu8peeng	Math	VTR	lx75t	33834	36.3
mcml	Medical Physics	VTR	lx130t	63373	25.4
cholesky_mc	Matrix Decomposition	Titan	lx195t	75425	30.2
dart	On Chip Network Simulator	Titan	lx195t	92345	37.0

[1] 6LUTs are counted as 2 5LUTs.

4.6.2 RSVPack Results

Slice Utilization

Table 4.4 shows the number of slices used by RSVPack compared to Xilinx. On average, RSVPack uses 19% more slices than Xilinx to pack a design. Some of this comes from a much poorer ability to join multiple LUT5s into a single fracturable LUT pair in an LE (see % Merged LUT5s in Table 4.4). Much of it, though, comes from low LE utilization of many of the slices. As shown in Figure 4.10, on average, compared to Xilinx, RSVPack-packed designs contain almost double the number of slices utilizing only a single LE.

Table 4.4: Number of Used Slices and Percent of LUT5s Merged

Benchmark	Used Slices			% Merged LUT5s		
	Xilinx	RSVPack	Ratio	Xilinx	RSVPack	Ratio
viterbi	2293	3129	1.36	38.5	36.1	0.94
microblaze	4482	6332	1.41	53.3	27.6	0.52
divider	4273	6501	1.52	73.5	60.0	0.82
bgm	5763	4630	0.80	42.5	32.5	0.76
lu8peeng	6943	8748	1.26	65.9	25.9	0.39
mcml	14097	17207	1.22	50.0	37.4	0.75
cholesky_mc	18319	20559	1.12	67.2	55.0	0.82
dart	21209	17617	0.83	41.7	34.1	0.82

Much of this increase in slice usage comes from lower LE utilization in the later stages of RSVPack. Figure 4.10 shows the distribution of slice density averaged across all benchmarks. RSVPack tends to create slice clusters utilizing only a single LE much more frequently than Xilinx. In five of the eight benchmarks, more than 20% of the slices packed by RSVPack contain only a single LE. In contrast, when packed by Xilinx, none of the benchmarks ever have more than 17% of slices containing a single LE. This significant number of low-utilized slices spreads out the designs leading to more routing. This issue is further explored in Section 5.2.

Even with the manual optimization to improve LUT/FF pairing discussed in Section 4.2.5, as seen in Table 4.5, RSVPack still splits many LUT/FF pairs. One of the added costs of separating the LUT from its FF pair it that this separation means that the net connecting the two must leave

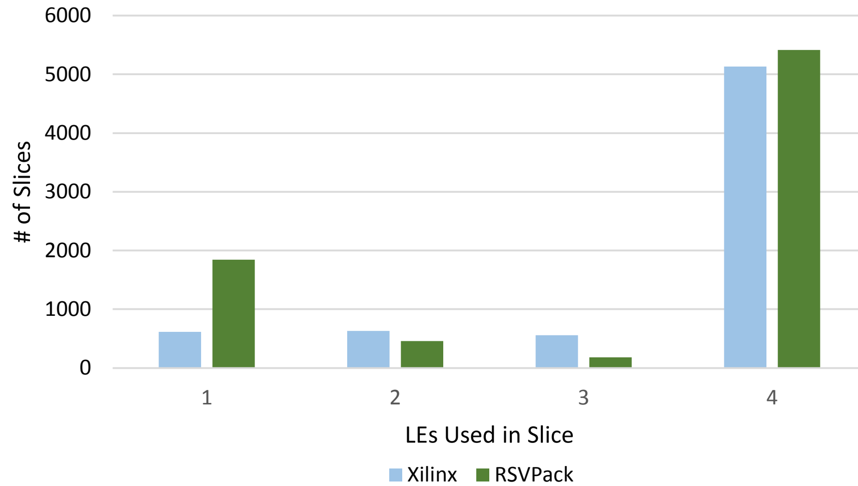


Figure 4.10: Densities of Slices Packed with Xilinx and RSVPack (average of benchmarks)

Table 4.5: Clustering of LUT/FF Pairs and Nets Exposed From Slices

Benchmark	% Paired LUT/FF Pairs			Exposed Nets		
	Xilinx	RSVPack	Ratio	Xilinx	RSVPack	Ratio
viterbi	53.2	47.3	0.89	11466	11757	1.03
microblaze	91.4	65.5	0.72	17517	19066	1.09
divider	100.0	94.4	0.94	27621	28769	1.04
bgm	96.6	63.6	0.66	20999	21939	1.04
lu8peeng	71.8	70.1	0.98	28838	28872	1.00
mcml	99.2	68.5	0.69	78581	83176	1.06
cholesky_mc	99.7	72.6	0.73	114090	126959	1.11
dart	99.1	43.8	0.44	82404	98810	1.20

the slice and enter the switch box. This occurs even if the FF is packed into another LE in the same slice. This has the effect of increasing the routing delay on the net containing the LUT/FF pair and adding extra congestion in the switch box. This separation is reflected in the number of exposed nets from the slices shown in Table 4.5.

Circuit Quality

The minimum clock period and total wire length (measured in tiles traveled) are reported in Table 4.6 and 4.7, respectively. These values are obtained from the fully placed and routed circuits with the minimum clock period coming from the Xilinx generated timing report. The reported

results come from the run with the best minimum clock period for each flow. These two metrics provide a view of how the packing affects the final circuit quality.

Table 4.6: Minimum Periods of Benchmarks Implemented with Different Flows (in ns)

	Standard	PAR Place	RSX	RSV
viterbi	5.77	6.22	6.68	9.19
microblaze	6.58	7.00	7.13	9.19
divider	3.20	3.18	3.67	5.18
bgm	7.56	7.73	8.28	10.58
lu8peeng	43.99	47.30	50.59	57.33
mcml	26.17	27.23	28.28	38.08
cholesky_mc	9.42	9.45	9.49	13.23
dart	9.84	11.26	11.64	20.76

Table 4.7: Total Wire Length of Benchmarks Implemented with Different Flows (in Thousands of Tiles Travelled)

	Standard	PAR Place	RSX	RSV
viterbi	186	179	191	164
microblaze	281	312	351	351
divider	223	268	287	374
bgm	279	331	291	268
lu8peeng	474	547	538	537
mcml	1020	1499	1402	1575
cholesky_mc	1345	1959	1564	2124
dart	1962	2297	2195	2466

The RSX flow performs well compared to Xilinx. The RSX flow averages a 5% higher minimum clock period and is usually within $\pm$ 10% of total wire length of the PAR Place flow. The RSV flow averages a 53% higher minimum clock period and uses 28% more wire length than the PAR Place flow demonstrating that Xilinx does a much better job than the academic placer. Observing this difference further illustrates the advantage of analyzing individual tools integrated into the commercial flows.

Run Time

Run times for the flows are presented in Table 4.8. The presented run times exclude the time spent in synthesis which is identical for all flows. The times for the RSX flow ignore `map` and `xdl` conversion times required for obtaining an XDL design description and instead treat RSVPack as a direct substitute for the equivalent Xilinx tool. The PAR Place flow counts the time to place the circuit twice – placement is performed in both `map` and `par`. Though not fair to compare flows against the PAR Place flow for this reason, the results are provided both for completeness and for a relative measure of the time spent in `par` to both place and route a circuit.

RSVPack runs in a reasonable time frame with many of the designs taking only a few minutes to complete and the largest completing in about 20 minutes. The overall time required by the RSX flow is within 1.5 times that of the standard Xilinx flow. The lower quality results of the RSV flow are especially noticeable as `par` spends significantly longer routing designs in this flow than when employing the standard Xilinx flow.

4.6.3 Summary

This work demonstrates that the RSVPack algorithm is capable of creating valid packing for Xilinx flows. Additionally, the results show that while academic packing algorithms still lag behind their commercial counterparts, the algorithms can be competitive.

Table 4.8: Implementation Runtimes for each Flow in Seconds (Median of 20 Runs)

Benchmark	Flow	map	RSVPack	RSVPlace	par	Total
viterbi	Standard	155			195	349
viterbi	PAR Place	155			419	574
viterbi	RSX		83		444	527
viterbi	RSV		80	50	477	607
microblaze	Standard	275			201	476
microblaze	PAR Place	275			361	636
microblaze	RSX		128		457	585
microblaze	RSV		132	129	278	538
divider	Standard	749			263	1012
divider	PAR Place	749			347	1096
divider	RSX		124		332	455
divider	RSV		120	132	246	498
bgm	Standard	366			123	488
bgm	PAR Place	366			305	671
bgm	RSX		106		233	339
bgm	RSV		104	134	175	412
lu8peeng	Standard	339			315	654
lu8peeng	PAR Place	339			871	1210
lu8peeng	RSX		194		967	1161
lu8peeng	RSV		188	290	485	963
mcml	Standard	984			1269	2253
mcml	PAR Place	984			1567	2552
mcml	RSX		316		1120	1436
mcml	RSV		302	752	7101	8154
cholesky_mc	Standard	986			1146	2133
cholesky_mc	PAR Place	986			2352	3338
cholesky_mc	RSX		532		3068	3599
cholesky_mc	RSV		503	1107	3134	4744
dart	Standard	923			1147	2070
dart	PAR Place	923			2070	2993
dart	RSX		1238		2029	3267
dart	RSV		1245	1166	13851	16261

CHAPTER 5. EXPERIMENTING WITH PACKING XILINX FPGAS USING RSV-PACK

RSVPack enables experimentation with packing using Xilinx FPGAs. As mentioned, this experimentation can be performed with just the packer while allowing Xilinx to handle all other parts of this flow. This chapter will look at different experiments evaluating the effect of packing on Xilinx FPGAs and ways that the packing can be improved. These experiments include:

1. exploring the impact that packing has on final circuit quality,
2. attempts to improve the utilization of the slice clusters, and
3. evaluating the tradeoffs of site level packing against tile level packing

5.1 Experiment 1: Impact of Packing on Circuit Quality

RSVPack performs reasonably well at creating valid packings for Xilinx FPGAs. These packings lead to circuits that run about 10% slower than their Xilinx-generated counterparts showing that RSVPack can be improved upon. Using RSVPack, we can further explore how packing quality can influence the final quality of a circuit. This experiment explores how circuit quality decreases as the quality of packing decreases.

5.1.1 Experiment Setup

To explore the impact of packing on the final circuit quality, this experiment augments the cell selector in RSVPack to produce lower quality packings. The degradation in packing quality is achieved by occasionally selecting a random cell to add to a cluster. Specifically, in this experiment, the RSVPack cell selector is modified so that every Nth cell returned by the cell selector is replaced with a random cell from the design – in all other cases, it acts as the normal cell selector. The returned cell is analyzed the same as all other cells and will not be added to the cluster if no legal

location exists for the cell in the cluster. With this selector, the quality of the packing can be degraded by increasing the frequency that a random cell is returned by the selector.

For this experiment, each of the previously discussed benchmarks were re-run with the RSX flow with RSVPack using the augmented cell selector with frequencies of a random cell being chosen of .25%, .5%, 1%, 2% and 3%. At frequencies greater than 3%, many of the benchmarks struggle to route.

5.1.2 Results

The most dramatic observable effect is the impact on whether the packed and placed circuits can even be routed. The likelihood that a circuit can be routed decreases as the rate of random cells selected increases. Both the *cholesky_mc* and *mcml* benchmarks cease to be routable when the selector reaches a rate of 3% randomness. The *dart* benchmark is unroutable for any frequency greater than 1%.

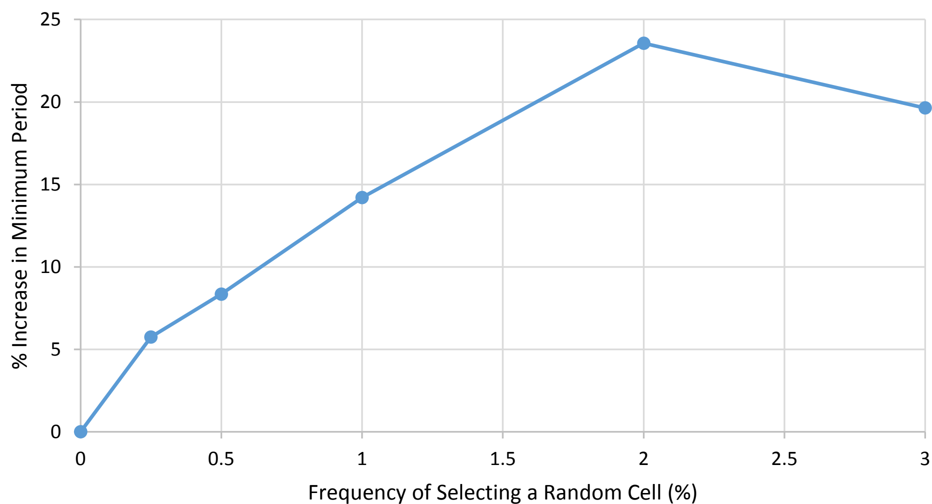


Figure 5.1: Average Increase in Minimum Period Against Packing Quality

Figures 5.1 and 5.2 show the increase in minimum period and total wire length, respectively, as the frequency of selecting a random cell increases. The graphs show the averages across all benchmarks – unroutable circuits are not included. The minimum period increases about 25% while the total wire length approaches a 70% increase. Altogether, this demonstrates that the qual-

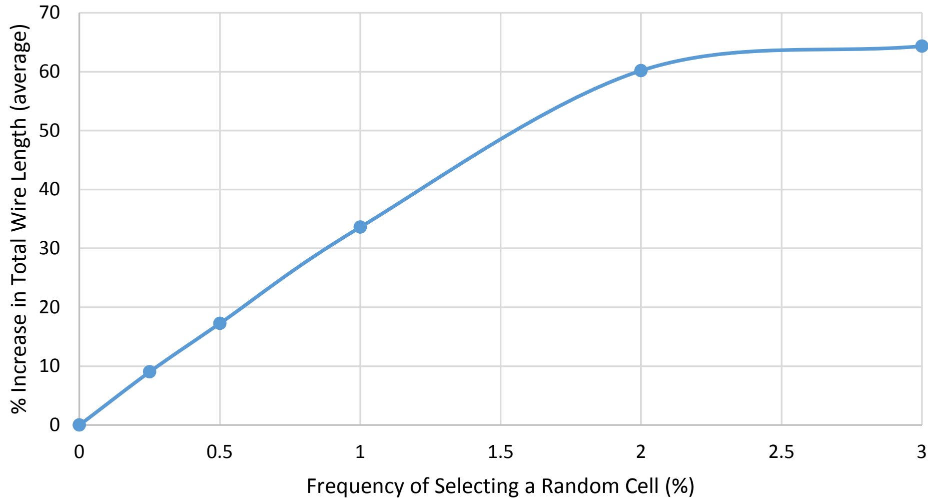


Figure 5.2: Average Increase in Wire Length Against Packing Quality

ity of the packing on a circuit has a noticeable influence on the final circuit for Xilinx FPGAs. It also shows that the minimum period and number of required routing resources will slowly degrade in concert with packing quality to the point that the circuit can no longer be successfully routed.

5.2 Experiment 2: Cluster Underutilization

To limit the number of cells RSVPack searches and to avoid adding unrelated logic to the clusters that can reduce circuit quality (see Section 5.1), the cell selector for RSVPack only looks at cells that share one or more nets with a cell already in the cluster. Such cells are considered to be one hop from the cluster. Considering cells only a single hop from the cluster reduces the run time of the packer – it only looks at adding a subset of the cells in the design – and allows the algorithm to scale with increasing design size.

As a side effect of not considering unrelated logic, and stemming from the locally-greedy nature of RSVPack, many clusters, especially those which are packed late in the process, are underutilized. Figure 5.3 shows how the clusters are increasingly less utilized as the RSVPack progresses. Early in the packing process, most of the clusters are being packed with at least 8 cells. About midway through the process, however, the density of the cells quickly drops off finishing with most slice clusters containing only between one to three cells at the end.

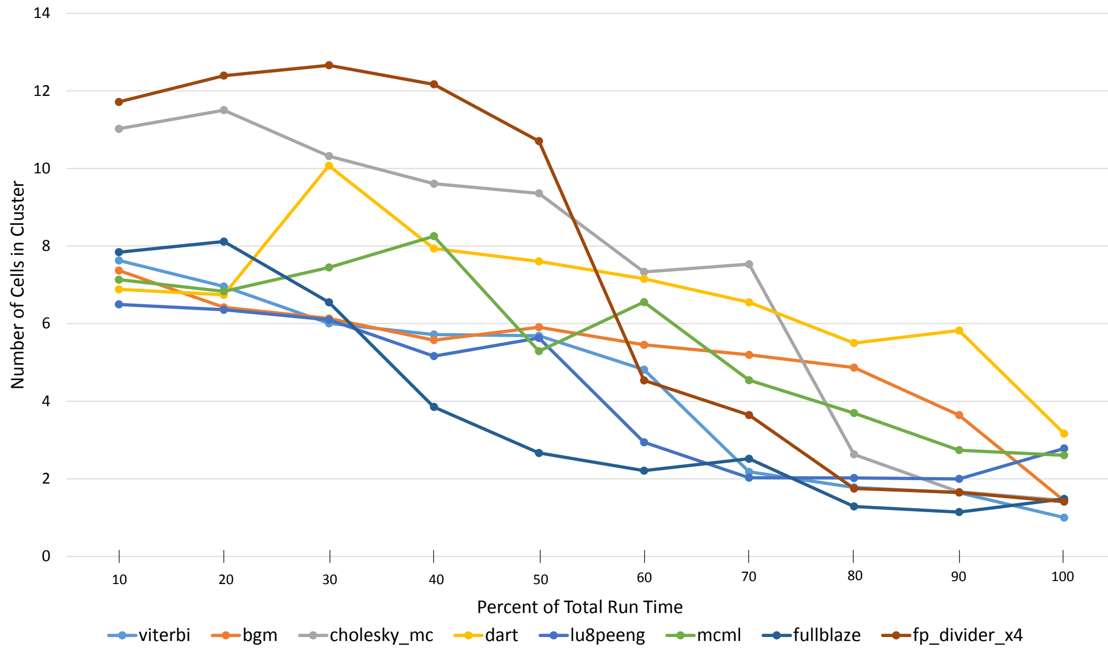


Figure 5.3: Progression of Packing Density of Clusters over Packing Process

The lower utilization over time stems from pockets of cells forming which are fully surrounded by packed cells. Figure 5.4 shows an example of one of these pockets. In the figure, cell B connects to cells A, C and D, each of which are already packed into clusters in this example. When cell B is selected to seed a cluster, all of the neighboring cells (A, C and D) are already packed leaving cell B to occupy the cluster alone.

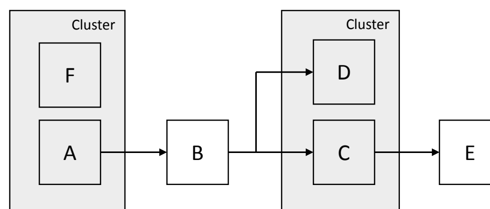


Figure 5.4: Example of a Pocket Forming Around a Cell (B) During Packing

The large number of cells which are underutilized increases the required number of slices and CLBs. More slices means the circuit is more spread out leading to longer routes and higher minimum periods. In the worst case, underutilization of the slices and CLBs can lead to larger designs.

5.2.1 Two-Hop Deep Cell Selector

Returning to Figure 5.4, while cell B has no adjacent unpacked cells left to be clustered with, the unpacked cell E presents a nearby candidate to be clustered with. Though cells B and E do not share any nets, by sharing a neighbor – in this case cell C – the cells should likely end up near one another in the final circuit and thus clustering them together is likely to be beneficial to the circuit and will help reduce the number of logic resources used in the device.

Cells such as B and E, which both connect to a shared cell but not each other, are considered two hops apart. To reduce the number of lightly utilized clusters, the cell selector used by RSVPack can be modified to consider these cells during packing. This *two-hop cell selector* allows RSVPack to search beyond the immediately adjacent connected cells for candidates to absorb into the clusters.

The two-hop cell selector used in this experiment operates in two modes. While unpacked adjacent cells remain for RSVPack to consider, the two-hop cell selector operates identically to the single hop cell selector and only considers adjacent cells. After all adjacent cells have been evaluated, the cell selector enters into its second mode, two-hop mode. In two-hop mode, the selector searches for and suggests cells which are two hops from the current cluster. Unlike the default one-hop cell selector, the two-hop cell selector stops after three LEs in the cluster are utilized. This helps prevent fully filling the clusters with only tangentially-related logic.

The two-hop cell selector limits the number of cells that are evaluated when in two-hop mode. The number of cells that are reachable in a certain number of hops is approximately exponential to the fan out of the circuit. This means that if a design has an average fan out of 20, the number of cells two hops away will likely be in the hundreds. This exponential growth can lead to much longer run times for only marginal gains. To limit the increase in run time, the two-hop cell selector stops considering additional cells after 50 consecutive cells could not be added to the cluster. Due to diminishing returns of packing cells that are more hops away, the two-hop cell selector will not search three or more hops away from the cluster.

5.2.2 Experiment Setup

Each of the benchmarks from Section 4.6.1 are implemented using both the standard one-hop and two-hop cell selectors. As before, numbers are taken from the best performing implementations, as measured by minimum clock period, after twenty runs of the CAD tool.

5.2.3 Results and Conclusion

Table 5.1: Influence of 2-Hop Deep Cell Selector on Benchmarks

Benchmark	Used Slices			Total Wire Length			Min Period (ns)		
	One Hop	Two Hop	% Decrease	One Hop	Two Hop	% Decrease	One Hop	Two Hop	% Decrease
viterbi	3129	2386	23.7	191	179	6.5	6.68	6.65	0.5
microblaze	6332	4929	22.2	351	324	7.6	7.13	7.65	-7.2
divider	6501	6182	4.9	287	271	5.7	3.67	3.52	4.1
bgm	4630	4453	3.8	291	302	-3.7	8.28	8.01	3.2
lu8peeng	8748	6822	22.0	538	612	-13.7	50.59	49.28	2.6
mcml	17207	16048	6.7	1402	1324	5.6	28.28	27.95	1.2
cholesky_mc	20559	18239	11.3	1564	1576	-0.7	9.49	9.57	-0.8
dart	17617	16596	5.8	2195	2339	-6.5	11.64	12.24	-5.2

As seen in Table 5.1, using the two-hop cell selector leads to decreases in the number of used slices with some benchmarks using up to 25% fewer. This decrease comes from having fewer slices that contain only a single LE (Figure 5.5). The number of LUTs with a single LE used drops up to 60%. The two-hop cell selector produces circuits with, on average, minimum periods and total wire lengths that are equivalent to the one hop cell selector.

Compared to the one-hop cell selector, the two-hop cell selector does a better job of maintaining consistency in the LE usage throughout the packing process (see Figure 5.6). Nevertheless, the two-hop cell selector does decrease in cluster density at the end.

Overall, the two-hop cell selector significantly reduces the number of slices required to pack a design while not causing consistent increases in wire usage and minimum period. The decrease in the number of required slices allows for designs to potentially fit on smaller devices. In the future this technique could be combined with approaches for managing the maximum density of a cluster, such as those used in [36], to better control how densely the slices are packed.

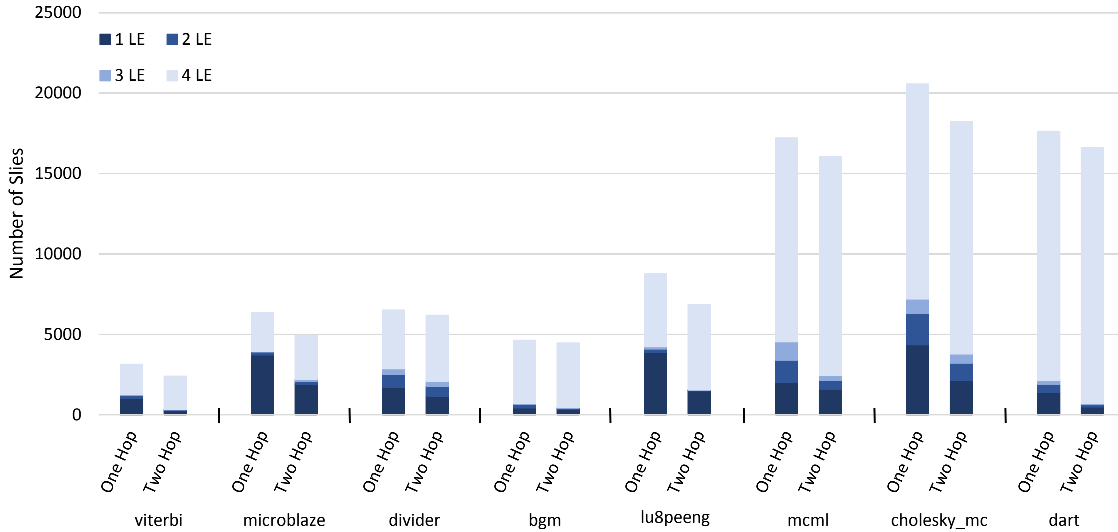


Figure 5.5: Utilization of LEs in Slices when Using Single-Hop and Two-Hop Cell Selectors

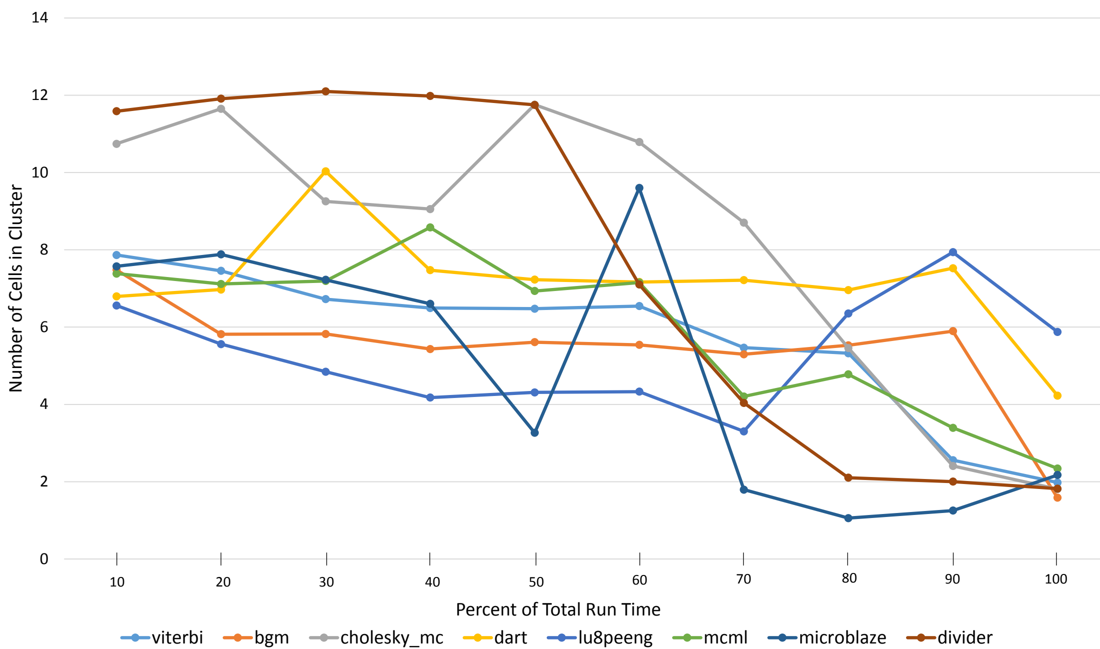


Figure 5.6: Progression of Packing Density of Clusters over Packing Process with Two-Hop Cell Selector

5.3 Experiment 3: Tile Versus Site Level Packing

Pack units are typically based on a naturally occurring level of hierarchy in the device. Xilinx has two such levels of hierarchy: tiles and sites. RSVPack is designed to work with either

of these levels of hierarchy. Up to this point, RSVPack has targeted site-level pack units. This experiment will explore the tradeoffs between packing onto site-level and tile-level pack units.

5.3.1 Background

The CLBs (tiles) in Virtex 6 each contain two slices (sites). The slices in each CLB are independent of one another except that they share a switch box. Because slices in a CLB share a switch box, a net connecting to both slices in the CLB uses only marginally more routing resources in the switch box than if the net connected to only one of the slices. Additionally, the switch boxes contain paths connecting the outputs of the slices to some of the pins on the other slice. These direct paths require no additional routing resources, such as bounce pins, and are often faster than other paths connecting the pins. Making use of these paths can reduce the amount of routing congestion in the switch box and lead to less routing delay. This would seem to encourage taking both slices into account when packing a design.

Another factor in packing is the granularity of the created clusters. Site-level clusters are finer granularity than tile-level clusters. The finer granularity provides more flexibility for the placer while increasing the number of clusters the placer must place by 2x.

RSVPack, by its nature, works with less contextual information than the global placer. The placer knows of the physical locations of each cluster in the circuit and can use this information to estimate routing delay and congestion for the circuit. Packing is performed before this information is available. In general, it is best to postpone decisions until later information is known, but this presents a tradeoff between algorithm complexity, run time and final circuit quality. Packing into sites instead of tiles postpones more of the decisions while increasing the workload for the placer.

The factors discussed above provide reasons why site-level and tile-level packing might each be be preferable. This experiment is designed to empirically determine which hierarchical level is best for packing with RSVPack.

5.3.2 Experiment Setup

Figure 5.7 shows the flows that are used to compare site-level and tile-level pack units. The names of the flows are appended with "-Site" and "-Tile" to distinguish which pack unit is used by which flow. The benchmarks from Section 4.6.1 are implemented with each of these flows.

Due to significant differences between RSVPlace and the placer in `par`, the RSV-based flows are analyzed independently of the RSX-based flows. Though, as has been shown, the RSV flow performs significantly worse than the RSX flow, the RSVPlace placement algorithm in the RSV flow can be configured to place the tile-level pack unit based clusters. In contrast, the placer in PAR used by the RSX flow is a site-level placer and will break the CLB clusters into individual slices during placement.

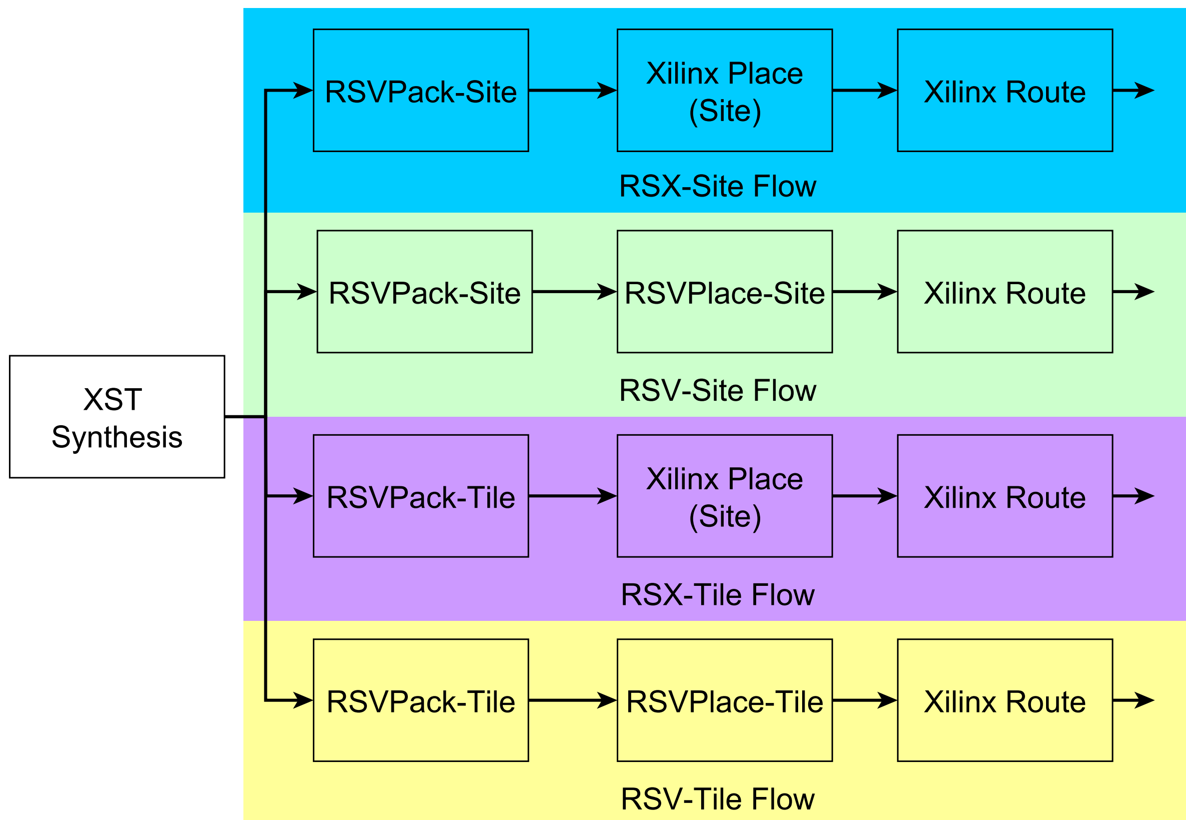


Figure 5.7: Flows for Comparing Tile-Level and Site-Level Based Packing

5.3.3 Results and Analysis

RSV-Tile versus RSV-Site

Table 5.2: Used Slices for RSV-Site and RSV-Tile Implemented Benchmarks

Benchmark	Used Slices		
	RSV-Site	RSV-Tile	$\frac{Site}{Tile}$
viterbi	3129	2858	1.09
microblaze	6332	5771	1.10
divider	6501	6997	0.93
bgm	4630	4494	1.03
lu8peeng	8748	8665	1.01
mcml[1]	17207	17388	0.99
cholesky_mc	20559	20434	1.01
dart	17617	17699	1.00

[1] Due to lack of CLB resources, the *mcml* benchmark is targeted for the xc6vlx195t part when using the RSV-Tile flow.

Table 5.3: HPWL, Used Wire Length, and Minimum Periods of RSV-Site and RSV-Tile Implemented Benchmarks

Benchmark	Half-Perimeter Wire Length (Thousands)			Total Wire Length (Thousands)			Min Period (ns)		
	RSV-Site	RSV-Tile	$\frac{Site}{Tile}$	RSV-Site	RSV-Tile	$\frac{Site}{Tile}$	RSV-Site	RSV-Tile	$\frac{Site}{Tile}$
viterbi	132	164	0.80	164	186	0.88	9.19	9.69	0.95
microblaze	290	315	0.92	351	365	0.96	9.19	12.52	0.73
divider	400	1195	0.33	374	942	0.40	5.18	8.62	0.60
bgm	248	357	0.70	268	336	0.80	10.58	12.53	0.84
lu8peeng	470	674	0.70	537	681	0.79	57.33	88.23	0.65
mcml	1514	3917	0.39	1575	—	—	38.08	—	—
cholesky_mc	2048	3218	0.63	2124	3020	0.70	13.23	18.08	0.73
dart	2242	3089	0.73	2466	—	—	20.76	—	—

The resource utilization by RSVPack for both tile-level and site-level placement is shown in Table 5.2. RSV-Site and RSV-Tile use about the same number of slices. However, tile-level

packing often leads to more CLBs being used. The same low utilization discussed in Section 5.2 appears in the tile-level packing but with whole tiles being underutilized. In the case of the *mcml* benchmark, tile-level packing required a larger part to fit all of the tiles. The difference in the CLB usage between tile and site level packing is heavily dependent on how densely the site-level placer places the slices, however.

RSV-Site significantly outperforms RSV-Tile in the final circuit quality (Table 5.3). With RSVPack-Tile, three of the benchmarks, *cholesky_mc*, *mcml* and *dart*, could not be routed due to routing congestion; wire length and minimum period are not presented for these benchmarks. In contrast, all of the benchmarks were successfully implemented with RSV-Site. In the successfully implemented benchmarks, RSV-Site produces circuits with minimum periods which are up to 66% faster and use 23% less wire length on average.

The Half-Perimeter Wire Length (HPWL) cost function used by RSVPlace shows RSV-Site producing placements with an average reduction of 35% over RSV-Tile produced placements. The *divider* and *mcml* circuits especially stand out with reductions of nearly 70%. These benchmarks differ from the others in that they make heavy use of the carry chains. This will be discussed later in this section.

In run time (Table 5.4) RSV-Tile runs up to 33% faster than RSV-Site (26 minutes vs 40 minutes for *dart*). These times are not large and likely do not justify the significant decrease in quality.

RSX-Tile versus RSX-Site

Both RSX-Tile and RSX-Site use Xilinx PAR to place and route the packed designs. PAR always places at the site-level and will break the tile-level clusters produced by RSVPack-Tile into site-level clusters during placement. Despite this, examining the RSX flows helps to show the effect that coupling the two sites in a tile together has on the potential quality of a circuit.

In contrast to the RSV flows, all benchmarks were successfully placed and routed with both the RSX-Site and RSX-Tile flows. Results from the experiment are presented in Table 5.5. Changes in total wire length vary depending on the benchmark, but across all of the benchmarks, RSV-Site implemented benchmarks see an average decrease of 5% compared to RSV-Tile imple-

Table 5.4: Run Times for RSV-Tile and RSV-Site in Seconds

Benchmark	Flow	Pack	Place	Total
viterbi	RSV-Site	80	50	130
	RSV-Tile	130	26	156
microblaze	RSV-Site	130	131	261
	RSV-Tile	184	73	257
divider	RSV-Site	120	132	252
	RSV-Tile	172	70	242
bgm	RSV-Site	104	134	238
	RSV-Tile	137	57	195
lu8peeng	RSV-Site	190	322	512
	RSV-Tile	307	155	462
mcml	RSV-Site	302	752	1053
	RSV-Tile	436	336	771
cholesky_mc	RSV-Site	503	1107	1610
	RSV-Tile			0
dart	RSV-Site	1245	1166	2411
	RSV-Tile	1123	483	1605

Table 5.5: Used Wire Length and Minimum Periods of RSX-Site and RSX-Tile Implemented Benchmarks

Benchmark	Total Wire Length (Thousands)			Min Period (ns)		
	RSV-Site	RSV-Tile	$\frac{Site}{Tile}$	RSV-Site	RSV-Tile	$\frac{Site}{Tile}$
viterbi	191	177	1.08	6.68	6.55	1.02
microblaze	351	384	0.92	7.13	7.43	0.96
divider	287	328	0.87	3.67	3.59	1.02
bgm	291	305	0.95	8.28	8.46	0.98
lu8peeng	538	619	0.87	50.59	51.91	0.97
mcml	1402	1352	1.04	28.28	28.06	1.01
cholesky_mc	1564	1750	0.89	9.49	9.50	1.00
dart	2195	2211	0.99	11.64	12.09	0.96

mented benchmarks. The differences in minimum period between the flows are negligible and average out to less than 1%.

Effect on Carry Chains

As mentioned, the *divider* and *mcml* benchmarks performed significantly worse with the RSV-Tile flow than the RSV-Site flow compared to the other benchmarks. The dramatic difference, however, is not reflected in their corresponding RSX flows. These two benchmarks, unlike the other benchmarks, are largely made up of arithmetic circuitry with over 25% of the LUTs in the circuits being involved in carry chain logic.

Carry chains are interesting structures in FPGA architectures. First, carry chains lead to very inflexible packing – the carry chain BEL forces at least 4 LUTs to be packed together at defined locations. Second, whereas most clusters can be placed independently of one another, clusters which are part of the same carry chain must be placed relative to one another. Thus, when a circuit contains a 32 bit adder, the 8 slices involved in the adder are all positioned relative to one another, effectively an 8 slice cluster.

Packing CLBs compounds the greater influence carry chains exert on the circuit. In the Virtex 6 architecture, each CLB contains two slices, and thus two carry chains. When packing tiles, if both carry chains in the CLB are used, all cells involved in either carry chain must be placed relative to one another. Further, the two carry chains packed into a cluster may have significant portions of each carry chain which, like the two carry chains seen in Figure 5.8, do not overlap.

The combining of the carry chains explains the dramatic increase in wire length seen in the *divider* and *mcml* benchmarks between the RSV-Site and RSV-Tile flows. When the paired carry chains from the tile-level clusters are broken up by the placer in `par`, the large discrepancy seen in these designs disappears and the circuits from both flows use similar amounts of routing resources.

5.3.4 Experiment Conclusions

This experiment shows that placing site-level clusters is superior to placing tile-level clusters. With access to the physical location of each cluster, the additional flexibility the site-level placer has over the tile-level placer yields much improved circuits. Additionally, the possibility

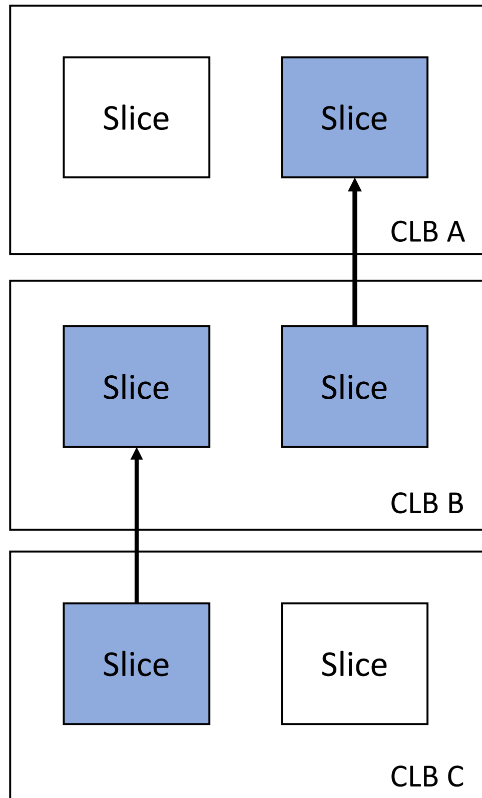


Figure 5.8: Non-Overlapping Carry Chains Paired Together in a CLB

of multiple carry chains being packed together with a tile-level packer can significantly degrade results. The added run time for site-level placement is also small. For these reasons, site-level packing is preferable to tile-level packing.

5.4 Summary

This chapter has shown different experiments that are enabled by RSVPack and RapidSmith 2. By decreasing the quality of the packing, the first experiment showed that poor packing can lead to unroutable circuits and can impact clock periods by at least 25%. The second experiment presented a way to reduce the number of slices used by a design while maintaining the quality of the final circuits. The last experiment showed that packing and placing sites leads to better quality circuits than packing and placing tiles on the Virtex 6 architecture.

CHAPTER 6. CONCLUSION

6.1 Summary of Research

The following is a summary of the research presented in this dissertation and its major contributions:

Chapters 1 and 2 provided motivation and background for this research. The need for an academic packer that can fit into a commercial tool frame was addressed. A survey of the different approaches to packing algorithms and the CAD frameworks which enable these algorithms was conducted. The survey also discussed previous work toward integrating one of these frameworks into the Xilinx tool flow and the limitations of that work.

Chapter 3 described the updates made to RapidSmith included in RapidSmith 2. The updates included interpreting and representing the subsite data structure provided in XDL and creating a netlist of cells to represent designs represented at the BEL level. As part of this representation, a method for integrating RapidSmith 2 into the Xilinx ISE CAD flow was also presented. This integration enables integrating custom packers into the Xilinx ISE flow.

Chapter 4 presented the RSVPack algorithm. This algorithm is unique in its ability to accept a Xilinx-synthesized netlist, perform packing on the netlist and return the packed results to be placed and routed by Xilinx. The changes to existing algorithms and checks required to support the Virtex 6 architecture were discussed. The packer and flow were demonstrated on a set of medium-sized benchmarks and produced circuits with minimum clock periods that are within 5% of Xilinx packed circuits. Finally, a novel table-based algorithm that determines cluster routability that runs 5 times faster than previous approaches was presented.

Chapter 5 detailed different experiments that have been performed using RSVPack and its related flows. These experiments demonstrated the impact that packing has on the quality of the final circuits, established that packing and placing sites leads to better results than packing and

placing tiles and showing that looking at cells two hops away from the current cluster can decrease slice usage while not decreasing general circuit quality.

6.2 Future Work

This work lays the foundation for exploring novel approaches for packing algorithms targeting Xilinx FPGAs. The latest release of RapidSmith 2 adds support for integrating custom tools into the Xilinx Vivado CAD suite. RSVPack is being updated to work with the latest version of RapidSmith 2. As part of this update, RSVPack will target the new architectures supported only by Vivado and will be integrated into the Vivado CAD flow. When complete, these updates to RSVPack will be released as part of the RapidSmith 2 project.

The following topics are areas where the RSVPack algorithm can be improved:

- Add timing-driven features to the algorithm to keep long chains of combinational logic located close together. As Xilinx does not make its wire delay model public, timing-driven features will likely involve developing a method to approximate wire delay.

- Make slice utilization more uniform through the entire packing process. Though the two-hop cell selector discussed helps make the densities of slices more consistent throughout the duration of packing, at the end, the LE utilization of the slices does still decrease. In addition to making this more uniform, it is important to identify how densely slices should be packed in a circuit.

- Explore techniques for modifying the packing during placement. Packing is performed early in the implementation process and is prone to making decisions that lead to poor circuit quality. By allowing the placer to rearrange some of the packing, the placer can separate components in clusters and place each component where it fits best on the circuit. Modifying the packing during placement will require the placer to perform the same checks that were performed by RSVPack.

Finally, with RSVPack and RSVPlace, the RapidSmith project now has two of the three components of a complete back-end FPGA implementation. Completing a router would provide a full back-end CAD flow allowing researchers to explore new algorithms and approaches for each

of the different components of the FPGA CAD flow and witness the implementations programmed onto physical hardware.

REFERENCES

[1] V. Betz and J. Rose, "VPR: A New Packing, Placement and Routing Tool for FPGA Research," in *Proceedings of the 7th International Workshop on Field-Programmable Logic and Applications*, ser. FPL '97. London, UK, UK: Springer-Verlag, 1997, pp. 213–222. [Online]. Available: http://dl.acm.org/citation.cfm?id=647924.738755 1, 14

[2] C. Beckhoff, D. Koch, and J. Torresen, "The Xilinx Design Language (XDL): Tutorial and use cases," in *6th International Workshop on Reconfigurable Communication-Centric Systems-on-Chip (ReCoSoC)*, June 2011, pp. 1–8. 3, 19

[3] C. Lavin, M. Padilla, J. Lamprecht, P. Lundrigan, B. Nelson, and B. Hutchings, "RapidSmith: Do-It-Yourself CAD Tools for Xilinx FPGAs," in *Field Programmable Logic and Applications (FPL), 2011 International Conference on*, Sept 2011, pp. 349–355. 3, 21, 84

[4] A. Otero, E. de la Torre, and T. Riesgo, "Dreams: A Tool for the Design of Dynamically Reconfigurable Embedded and Modular Systems," in *Reconfigurable Computing and FPGAs (ReConFig), 2012 International Conference on*, Dec 2012, pp. 1–8. 3

[5] D. Peterson, O. Bringmann, T. Schweizer, and W. Rosenstiel, "StML: Bridging the Gap between FPGA Design and HDL Circuit Description," in *Field-Programmable Technology (FPT), 2013 International Conference on*, Dec 2013, pp. 278–285. 3

[6] C. Lavin, B. Nelson, and B. Hutchings, "Impact of Hard Macro Size on FPGA Clock Rate and Place/Route Time," in *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*, Sept 2013, pp. 1–6. 3, 19

[7] T. Haroldsen, B. Nelson, and B. White, "Rapid FPGA Design Prototyping through Preservation of System Logic: A Case Study," in *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*, Sept 2013, pp. 1–7. 3, 21, 84

[8] L. Gantel, M. Benkhelifa, F. Lemonnier, and F. Verdier, "Module Relocation in Heterogeneous Reconfigurable Systems-on-Chip using the Xilinx Isolation Design Flow," in *Reconfigurable Computing and FPGAs (ReConFig), 2012 International Conference on*, Dec 2012, pp. 1–6. 3

[9] B. Hutchings and J. Keeley, "Rapid Post-Map Insertion of Embedded Logic Analyzers for Xilinx FPGAs," in *Field-Programmable Custom Computing Machines (FCCM), 2014 IEEE 22nd Annual International Symposium on*, May 2014, pp. 72–79. 3, 19, 21

[10] A. Das, S. Venkataraman, and A. Kumar, "Improving Autonomous Soft-error Tolerance of FPGA through LUT Configuration Bit Manipulation," in *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*, Sept 2013, pp. 1–8. 3

[11] M. Abdelfattah, L. Bauer, C. Braun, M. Imhof, M. Kochte, H. Zhang, J. Henkel, and H. Wunderlich, "Transparent Structural Online Test for Reconfigurable Systems," in *On-Line Testing Symposium (IOLTS), 2012 IEEE 18th International*, June 2012, pp. 37–42. 3

[12] M. Wirthlin, J. Jensen, A. Wilson, W. Howes, S. Wen, and R. Wong, "Placement of Repair Circuits for In-field FPGA Repair," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '13. New York, NY, USA: ACM, 2013, pp. 115–124. [Online]. Available: http://doi.acm.org/10.1145/2435264.2435286 3, 21

[13] H. Pham, S. Pillement, and S. Piestrak, "Low-overhead Fault-tolerance Technique for a Dynamically Reconfigurable Softcore Processor," *Computers, IEEE Transactions on*, vol. 62, no. 6, pp. 1179–1192, June 2013. 3, 21

[14] A. Sari, D. Agiakatsikas, and M. Psarakis, "A Soft Error Vulnerability Analysis Framework for Xilinx FPGAs," in *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-programmable Gate Arrays*, ser. FPGA '14. New York, NY, USA: ACM, 2014, pp. 237–240. [Online]. Available: http://doi.acm.org/10.1145/2554688.2554767 3

[15] O. Soll, T. Korak, M. Muehlberghuber, and M. Hutter, "EM-based Detection of Hardware Trojans on FPGAs," in *Hardware-Oriented Security and Trust (HOST), 2014 IEEE International Symposium on*, May 2014, pp. 84–87. 3, 21

[16] B. L. Hutchings, J. Monson, D. Savory, and J. Keeley, "A Power Side-channel-based Digital to Analog Converter for Xilinx FPGAs," in *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-programmable Gate Arrays*, ser. FPGA '14. New York, NY, USA: ACM, 2014, pp. 113–116. [Online]. Available: http://doi.acm.org/10.1145/2554688.2554770 3

[17] T. Haroldsen, B. Nelson, and B. Hutchings, "RapidSmith 2: A Framework for BEL-level CAD Exploration on Xilinx FPGAs," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '15. New York, NY, USA: ACM, 2015, pp. 66–69. [Online]. Available: http://doi.acm.org/10.1145/2684746.2689085 3, 23, 84

[18] J. Luu, J. Rose, and J. Anderson, "Towards Interconnect-adaptive Packing for FPGAs," in *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-programmable Gate Arrays*, ser. FPGA '14. New York, NY, USA: ACM, 2014, pp. 21–30. [Online]. Available: http://doi.acm.org/10.1145/2554688.2554783 4, 12, 17, 34, 46

[19] Altera. FPGA Architecture White Paper. [Online]. Available: https://www.altera.com/en_US/pdfs/literature/wp/wp-01003.pdf 7

[20] Microsemi. UG0574 User Guide RTG4 FPGA Fabric. [Online]. Available: https://www.microsemi.com/document-portal/doc_download/134407-ug0574-rtg4-fpga-fabric-user-guide 7

[21] M. Gort and J. Anderson, "Analytical Placement for Heterogeneous FPGAs," in *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, Aug 2012, pp. 143–150. 13

[22] M. Xu, G. Grewal, and S. Areibi, "StarPlace: A New Analytic Method for FPGA Placement," *Integr. VLSI J.*, vol. 44, no. 3, pp. 192–204, Jun. 2011. [Online]. Available: http://dx.doi.org/10.1016/j.vlsi.2011.02.001 13

[23] Y. Sankar and J. Rose, "Trading Quality for Compile Time: Ultra-fast Placement for FPGAs," in *Proceedings of the 1999 ACM/SIGDA Seventh International Symposium on Field Programmable Gate Arrays*, ser. FPGA '99. New York, NY, USA: ACM, 1999, pp. 157–166. [Online]. Available: http://doi.acm.org/10.1145/296399.296449 14

[24] E. Bozorgzadeh, S. Ogrenci-Memik, and M. Sarrafzadeh, "RPack: Routability-Driven Packing for Cluster-Based FPGAs," in *Asia South Pacific Design Automation Conference*, 2001, pp. 629–634. 14

[25] J. S. Swartz, V. Betz, and J. Rose, "A Fast Routability-Driven Router for FPGAs," in *Proceedings of the 1998 ACM/SIGDA Sixth International Symposium on Field Programmable Gate Arrays*, ser. FPGA '98. New York, NY, USA: ACM, 1998, pp. 140–149. [Online]. Available: http://doi.acm.org/10.1145/275107.275134 14

[26] M. G. Wrighton and A. M. DeHon, "Hardware-Assisted Simulated Annealing with Application for Fast FPGA Placement," in *Proceedings of the 2003 ACM/SIGDA Eleventh International Symposium on Field Programmable Gate Arrays*, ser. FPGA '03. New York, NY, USA: ACM, 2003, pp. 33–42. [Online]. Available: http://doi.acm.org/10.1145/611817.611824 14

[27] J. Luu, J. Goeders, M. Wainberg, A. Somerville, T. Yu, K. Nasartschuk, M. Nasr, S. Wang, T. Liu, N. Ahmed, K. B. Kent, J. Anderson, J. Rose, and V. Betz, "VTR 7.0: Next Generation Architecture and CAD System for FPGAs," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 7, no. 2, pp. 6:1–6:30, Jul. 2014. [Online]. Available: http://doi.acm.org/10.1145/2617593 14

[28] P. Jamieson, K. B. Kent, F. Gharibian, and L. Shannon, "Odin II - An Open-Source Verilog HDL Synthesis Tool for CAD Research," in *2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*, May 2010, pp. 149–156. 15

[29] A. Mishchenko. ABC: A System for Sequential Synthesis and Verification. [Online]. Available: http://www-cad.eecs.berkeley.edu/~alanmi/abc 15

[30] E. Hung, F. Eslami, and S. Wilton, "Escaping the Academic Sandbox: Realizing VPR Circuits on Xilinx Devices," in *Field-Programmable Custom Computing Machines (FCCM), 2013 IEEE 21st Annual International Symposium on*, April 2013, pp. 45–52. 15

[31] E. Hung, "Mind the (Synthesis) Gap: Examining Where Academic FPGA Tools Lag Behind Industry," in *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*, Sept 2015, pp. 1–4. 15

[32] A. S. Marquardt, V. Betz, and J. Rose, "Using Cluster-based Logic Blocks and Timing-driven Packing to Improve FPGA Speed and Density," in *Proceedings of the 1999 ACM/SIGDA Seventh International Symposium on Field Programmable Gate Arrays*, ser. FPGA '99. New York, NY, USA: ACM, 1999, pp. 37–46. [Online]. Available: http://doi.acm.org/10.1145/296399.296426 16

[33] E. Bozorgzadeh, S. O. Memik, X. Yang, and M. Sarrafzadeh, "Routability-driven Packing: Metrics and Algorithms for Cluster-based FPGAs," *Journal of Circuits Systems and Computers*, vol. 13, pp. 77–100, 2004. 16

[34] S. T. Rajavel and A. Akoglu, "MO-pack: Many-objective Clustering for FPGA CAD," in *Proceedings of the 48th Design Automation Conference*, ser. DAC '11. New York, NY, USA: ACM, 2011, pp. 818–823. [Online]. Available: http://doi.acm.org/10.1145/2024724.2024908 16

[35] A. Singh and M. Marek-Sadowska, "Efficient Circuit Clustering for Area and Power Reduction in FPGAs," in *Proceedings of the 2002 ACM/SIGDA Tenth International Symposium on Field-programmable Gate Arrays*, ser. FPGA '02. New York, NY, USA: ACM, 2002, pp. 59–66. [Online]. Available: http://doi.acm.org/10.1145/503048.503058 16

[36] W. Feng, J. Greene, K. Vorwerk, V. Pevzner, and A. Kundu, "Rent's Rule Based FPGA Packing for Routability Optimization," in *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-programmable Gate Arrays*, ser. FPGA '14. New York, NY, USA: ACM, 2014, pp. 31–34. [Online]. Available: http://doi.acm.org/10.1145/2554688.2554763 16, 66

[37] Z. Marrakchi, H. Mrabet, and H. Mehrez, "Hierarchical FPGA Clustering Based on Multi-level Partitioning Approach to Improve Routability and Reduce Power Dissipation," in *2005 International Conference on Reconfigurable Computing and FPGAs (ReConFig'05)*, Sept 2005, pp. 4 pp.–25. 16

[38] W. Feng, "K-way Partitioning Based Packing for FPGA Logic Blocks Without Input Bandwidth Constraint," in *2012 International Conference on Field-Programmable Technology*, Dec 2012, pp. 8–15. 16

[39] T. Ahmed, P. D. Kundarewich, J. H. Anderson, B. L. Taylor, and R. Aggarwal, "Architecture-Specific Packing for Virtex-5 FPGAs," in *Proceedings of the 16th International ACM/SIGDA Symposium on Field Programmable Gate Arrays*, ser. FPGA '08. New York, NY, USA: ACM, 2008, pp. 5–13. [Online]. Available: http://doi.acm.org/10.1145/1344671.1344675 17

[40] J. Luu, J. Anderson, and J. Rose, "Architecture Description and Packing for Logic Blocks with Hierarchy, Modes and Complex Interconnect," in *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '11. New York, NY, USA: ACM, 2011, pp. 227–236. [Online]. Available: http://doi.acm.org/10.1145/1950413.1950457 17, 37

[41] D. Vercruyce, E. Vansteenkiste, and D. Stroobandt, "Runtime-Quality Tradeoff in Partitioning Based Multithreaded Packing," in *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, Aug 2016, pp. 1–9. 18

[42] G. Chen and J. Cong, "Simultaneous Timing Driven Clustering and Placement for FPGAs," in *In Proceedings of the International Conference on Field Programmable Logic and Its Applications*, 2004, pp. 158–167. 18

[43] ——, "Simultaneous Timing-driven Placement and Duplication," in *Proceedings of the 2005 ACM/SIGDA 13th International Symposium on Field-programmable Gate Arrays*, ser. FPGA '05. New York, NY, USA: ACM, 2005, pp. 51–59. [Online]. Available: http://doi.acm.org/10.1145/1046192.1046200 18

[44] D. Chen, K. Vorwerk, and A. Kennings, "Improving Timing-Driven FPGA Packing with Physical Information," in *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, Aug 2007, pp. 117–123. 18

[45] N. Steiner, A. Wood, H. Shojaei, J. Couch, P. Athanas, and M. French, "Torc: Towards an Open-source Tool Flow," in *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '11. New York, NY, USA: ACM, 2011, pp. 41–44. [Online]. Available: http://doi.acm.org/10.1145/1950413.1950425 21

[46] C. Lavin, B. Nelson, and B. Hutchings, "Improving Clock-rate of Hard-macro Designs," in *Field-Programmable Technology (FPT), 2013 International Conference on*, Dec 2013, pp. 246–253. 21

[47] C. Lavin, M. Padilla, J. Lamprecht, P. Lundrigan, B. Nelson, B. Hutchings, and M. Wirthlin, *RapidSmith: Technical Report and Documentation*, January 2014. [Online]. Available: http://rapidsmith.sourceforge.net/doc/TechReportAndDocumentation.pdf 24

[48] B. Nelson, T. Haroldsen, and T. Townsend, *RapidSmith 2: A Library for Low-level Manipulation of Vivado Designs at the Cell/BEL Level*, May 2017. [Online]. Available: https://github.com/byuccl/RapidSmith2/blob/master/doc/TechReport.pdf 28, 84

[49] T. Haroldsen, B. Nelson, and B. Hutchings, "Packing a modern xilinx fpga using rapidsmith," in *2016 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, Nov 2016, pp. 1–6. 33, 84

[50] L. McMurchie and C. Ebeling, "PathFinder: A Negotiation-Based Performance-Driven Router for FPGAs," in *Field-Programmable Gate Arrays, 1995. FPGA '95. Proceedings of the Third International ACM Symposium on*, 1995, pp. 111–117. 46

[51] J. Luu, J. Goeders, M. Wainberg, A. Somerville, T. Yu, K. Nasartschuk, M. Nasr, S. Wang, T. Liu, N. Ahmed, K. B. Kent, J. Anderson, J. Rose, and V. Betz. Verilog-to-Routing 7.0. [Online]. Available: https://code.google.com/p/vtr-verilog-to-routing/ 55

[52] K. E. Murray, S. Whitty, S. Liu, J. Luu, and V. Betz, "Timing-Driven Titan: Enabling Large Benchmarks and Exploring the Gap Between Academic and Commercial CAD," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 8, no. 2, pp. 10:1–10:18, Mar. 2015. [Online]. Available: http://doi.acm.org/10.1145/2629579 55

[53] Xilinx. Xilinx CORE Generator System. [Online]. Available: https://www.xilinx.com/products/design-tools/coregen.html 55

[54] OpenCores.org. Viterbi HDL Code Generator. [Online]. Available: https://opencores.org/project,vhcg 55

[55] T. Haroldsen, B. Nelson, and B. Hutchings, "Integrating a Custom Packer into the Xilinx CAD Flow with RapidSmith," *Microprocessors and Microsystems*, submitted for review. 84

[56] B. White and B. Nelson, "Tincr - A Custom CAD Tool Framework for Vivado," in *2014 International Conference on ReConFigurable Computing and FPGAs (ReConFig14)*, Dec 2014, pp. 1–6. 84

[57] T. Townsend and B. Nelson, "Vivado Design Interface: An Export/Import Capability for Vivado FPGA Designs," in *Field Programmable Logic and Applications (FPL), 2017 27th International Conference on*, Ghent, Belgium, Sep 2017, to appear. 84

APPENDIX A. PUBLICATIONS AND DOCUMENTATION

Work discussed in this dissertation is also published in [17], [49], and [55]. During my doctoral work, I have also published the following paper [7].

This work builds on the RapidSmith 1 project described in [3]. Work towards interfacing RapidSmith 2 with Vivado is presented in [56] and [57]. The documentation for RapidSmith 2 is available online at [48].

APPENDIX B. RAPIDSMITH 2 CHANGELIST

The version of RapidSmith 2 used in this work differs from the publicly released version in the following ways:

- The public release of RapidSmith 2 separates the intersite route tree from the intrasite route trees.
- The public release of RapidSmith 2 adds support for hierarchical cells. This primarily changes how distributed RAMs (LUTRAMs) are represented in RapidSmith 2.
- In the public release of RapidSmith 2, all nets sourced by ground or power connect to the same net. In this work, a new net is created for each ground or power source.
- The public release of RapidSmith 2 adds support for the LUTs to act as route throughs – ie. unused LUTs can be treated as wires. In this work, to use a LUT as a pass through, a LUT cell programmed to pass one input through unchanged must be created in the netlist.
- The public release of RapidSmith 2 allows cells to use *pseudo pins* to connect nets to unmapped BEL pins. For example, an 5-input LUT cell placed on a 6LUT BEL must connect the A6 pin on the BEL to VCC. As the 5-input LUT cell only has 5 inputs, a pseudo pin is added to the cell which maps to the A6 pin on the BEL. In this work, after packing, the type of the cells were changed to allow the required connections.

None of these changes should influence the results provided in this work.

APPENDIX C. XDL-UNPACKING AND XDL-PACKING ALGORITHMS

XDL-unpacking and XDL-packing are tools to translate an XDL-based design into a cell-based design used by RapidSmith 2 and back, respectively. To simplify the conversion process, the algorithms work from the RapidSmith 1 instance-based netlists instead of processing the XDL directly. The full flow to convert a Xilinx NCD netlist to RapidSmith 2's cell-based netlists is shown in Figure C.1.

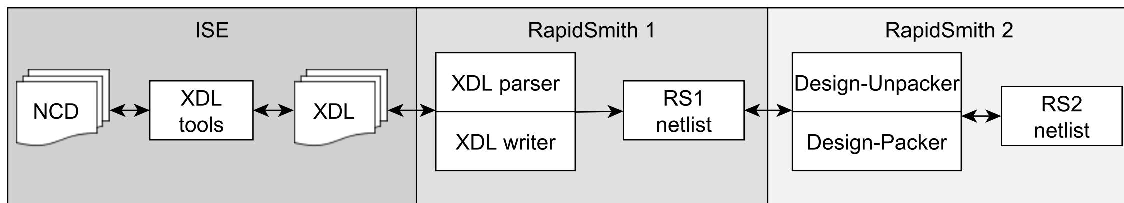


Figure C.1: Converting between XDL and RapidSmith 2 with the Design-Unpacker

C.1 Design Unpacking

XDL-unpacking, is the process of interpreting the attributes in the instances in an XDL design and translating them into a netlist of cells. This process involves identifying which attributes in the instance should be translated into cells, determining the type for each of these cells, associating each configuration attribute in the instance with the right properties on the appropriate cells, and translating the mux attributes into the route tree structure. The XDL-Unpacking algorithm is shown in Algorithm 2.

To guide the translations of XDL attributes to BELs and properties, the XDL-unpacker references an `unpack.xml` file which specifies how every BEL in the device should be XDL-unpacked. The `unpack.xml` file, shown in Figure C.2, describes how each attribute relating to a BEL in the XDL should be unpacked. For trivial BELs, the corresponding entry in the `unpack.xml`

```xml
<root>
  <bel>
    <!-- The type of BEL to unpack being defined. -->
    <id>
      <primitive_type>SLICEL</primitive_type>
      <name>A5FF</name>
    </id>
    <!--
      Specifies how this section should be processed.
      Options include:
        automatic - all information for unpacking is contained in this file
        manual - unpacking BELs of this type should be done by an external class
            specified in a <class> tag
     -->
    <mode>automatic</mode>
    <!-- Defines the type for the cell -->
    <cell_type>FF_INIT</cell_type>
    <!--
      Declares the attributes to be added as properties of the cell.  If rename
      tag is specified, the property should have the specified name.
     -->
    <attributes>
      <attribute>
        <name>A5FFINIT</name>
        <rename>INIT</rename>
      </attribute>
    </attributes>
    <!--
      Declares how to map the pins on the BEL to the pins on the cell.
      Currently only direct is supported.
     -->
    <pins mode="direct"/>
  </bel>
</root>
```

Figure C.2: A BEL entry in the `unpack.xml`

file specifies the type for the cell that is created, a mapping of the pins on the BEL to the pins on the cell, and the XDL property attributes that relate to the configuration of the cell (e.g. attribute CFFSR in the XDL is translated to the SR property for the FF cell located at the CFF BEL).

In most cases, the description in the `unpack.xml` is sufficient for translating the XDL BEL attributes to cells. However, in some cases, the type of the cell is a property of both the BEL the cell is located on and how the cell is used. For example, the A6LUT attribute in a SLICEM instance may translate to a cell of type LUT6, LUT5, SRL, or LUTRAM based on how many pins are connected to the cell and the associated configuration attributes for the cell. In these cases, a more complex analysis is required to determine the type of the cell. To allow for this analysis, `unpack.xml` points to a software hook that the XDL-unpacker can call to determine the types and properties for the corresponding cells.

In the above cases where a BEL attribute could be translated into more than one possible cell type, the XDL-unpacker chooses the cell type that provides the cell with the most placement flexibility. For example, in the previous A6LUT example, if the BEL uses only five of the six available input pins, the LUT5 cell type will be preferred over the LUT6 type. This allows the created cell to be placed on both 5LUT and 6LUT BELs in the device; a LUT6 cell could only be placed on a 6LUT BEL. Similarly, the LUT5 and LUT6 cell types are preferred over the SRL and LUTRAM types. In all cases, however, the selected cell type is guaranteed to have all of the functionality required to describe the resource.

As each BEL attribute in the XDL needs to be translated into a cell from the cell library, the XDL-unpacking process is closely tied to a cell library. To simplify the process, the XDL-unpacker utilizes a cell library that closely resembles the BELs in XDL. This allows for most BELs in the device to have a direct mapping to a single cell type from the library, leading to a straight-forward translation.

To translate the subsite routing, the XDL-unpacker performs a tree traversal algorithm starting at the outputs of each used BEL in a site to build the corresponding route tree. When traversing the routing path, upon reaching a PIP in the graph, the algorithm will determine if the PIP is enabled by matching the PIP to its associated attribute in the XDL and checking the value of the attribute to see if the PIP is enabled. If the PIP is enabled in the XDL, the traversal continues. Otherwise, the traversal stops at the disabled PIP.

Once complete, the algorithm returns a functionally equivalent cell-based representation of the original XDL design. This representation includes all cells in the design, route trees representing the subsite routing, and, in the case of a fully routed XDL design, the route trees representing the intersite routing.

C.2 XDL-Packing

XDL-packing converts the cell-based netlist in RS2 into an RS1 instance-based netlist which can be converted to XDL. When complete, the XDL-Packer will return a functionally equivalent XDL design of the input cell-based netlist. Though, XDL-packing does not need as much outside information as XDL-unpacking does, it still uses an `pack.xml` to provide the mapping

Algorithm 2 Design-Unpacker Algorithm

```
 1: function CONVERT(design)
 2:     cellDesign ← a new cell-based design
 3:     for each net in design.nets
 4:         cellNet ← a new CellNet
 5:         if net is routed then
 6:             routeTree ← buildRouteTree(net)
 7:             add routeTree to cellNet
 8:         end if
 9:         add cellNet to cellDesign
10:     end for
11:     for each instance in design.instances
12:         for each attribute in instance.attributes
13:             if attribute represents a BEL in the design then
14:                 cell ← a new Cell
15:                 cell.type ← type as determined by unpack.xml
16:                 cell.properties ←getProperties(attribute, instance, unpack.xml)
17:             end if
18:         end for
19:         for each sourcePin in used inputs to the site
20:             wire ← sourcePin.wire
21:             net ← the external net connected to sourcePin
22:             routeTree ← buildRouteTreeFrom(wire,net)
23:             add routeTree to net
24:         end for
25:         for each sourcePin in source pins on used BELs
26:             wire ← sourcePin.wire
27:             net ← a new CellNet
28:             routeTree ← buildRouteTreeFrom(wire,net)
29:             add routeTree to net
30:         end for
31:     end for
32:     return cellDesign
33: end function
```

```
34: function BUILDROUTETREEFROM(wire, net)
35:     for each sinkWire driven by wire
36:         if connection from wire to sinkWire is a PIP then
37:             if PIP is enabled in the instance then
38:                 buildRouteTreeFromwire, net
39:         else
40:             buildRouteTreeFromwire, net
41:         end if
42:         if wire connects to a used BEL pin then
43:             add the associated cell pin as a sink to net
44:         if wire connects to a used site pin then
45:             merge net with the external net connected to the site pin
46:     end for
47: end function
```

from the names of the properties on the cells to their corresponding XDL attribute names. The arrangement of the `pack.xml` file is shown in Figure C.3.

The XDL-packing algorithm is presented in Algorithm 3. Because XDL requires all components to reside within an instance, XDL-packing only works with placed netlists. It would be possible to translate a packed-but-not-placed input netlist, but RS2 does not have any native support for such netlists at this time.

```
<root>
  <bel>
    <!-- The type of BEL to unpack being defined. -->
    <id>
      <primitive_type>SLICEL</primitive_type>
      <name>A5FF</name>
    </id>
    <!--
      Declares the attributes to be mapping from property names to attribute
      names for cells placed on this BEL
     -->
    <attributes>
      <attribute>
        <name>INIT</name>
        <rename>A5FFINIT</rename>
      </attribute>
    </attributes>
  </bel>
</root>
```

Figure C.3: A BEL entry in the `pack.xml`

Algorithm 3 Design-Packer Algorithm

```
 1: function CONVERT(cellDesign)
 2:     design ← a new instance-based design
 3:     for each site in cellDesign.usedSites
 4:         instance ← a new instance
 5:         add instance to design
 6:         set type of instance based on the BELs used in the site
 7:         for each cell in cellDesign.cells placed in site
 8:             bel ← cell.bel
 9:             attribute ← a new attribute
10:             attribute.name ← bel.name
11:             attribute.value ← cell.attributes[cell.name]
12:             add attribute to instance
13:
14:             for each property in properties of cell
15:                 attribute ← a new attribute
16:                 attribute.name ← name based on property.name and cell.bel
17:                 attribute.value ← property.value
18:                 add attribute to instance
19:             end for
20:         end for
21:     end for
22:     for each cellNet in cellDesign.nets
23:         traverse each route tree in cellNet
24:             if connection is a subsite PIP then
25:                 instance ← the instance at the site of the PIP
26:                 attribute ← a new attribute
27:                 attribute.name ← name of the mux matching PIP
28:                 attribute.value ← property.value
29:                 add attribute to instance
30:             else if the net enters or leaves the site then
31:                 net ← the net in the design or a new net
32:                 add the instance pin to the net
33:             else if  connection is a tile-level PIP then
34:                 create and add the PIP to net
35:             end if
36:         end traversal
37:     end for
38:     return design
39: end function
```

APPENDIX D. CHECKS REQUIRED FOR PACKING FOR A VIRTEX 6 DEVICE

Below is a list of checks that must be performed when packing a design for a Virtex 6 FPGA to produce a legal circuit. The checks that prevent cells from being packed prematurely consist of:

- When the CYINIT pin on a carry chain element is driven by a non-power input, the cell sourcing the DI0 pin on the carry chain element must be packed with the carry chain element.
- When both the O and CO pins on a carry chain element are used, at least one of the FFs driven by the O and CO pins on a carry chain element must be packed together with the carry chain element.
- All cells in a multi-cell LUTRAM must be packed together. Failing to do so may lead to violations of other architectural requirements.

The checks that cover the discovered design rules for the archiecture consist of:

- When both the 5LUT and 6LUT components of a LUT are used, cells placed on both components must be the same type of cell (e.g. LUTs, SRL16, SPRAM32, DPRAM32).
- When both the 5LUT and 6LUT components of a LUT are used, either both components must be 5-input LUT types or the equation of the cell placed on the 5LUT must be identical to the equation of the cell placed on the 6LUT when the highest index pin (A6) is low.
- The cells that are part of a multi-cell LUTRAM must be placed at BELs that perform the appropriate write address checking for the cell.
- When a LUT in a slice is used as a LUTRAM, the D6LUT must be occupied.